



Advanced artefact analysis

Advanced dynamic 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 ComCERT¹ (Poland), S-CURE² (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.

Legal notice

Notice must be taken that this publication represents the views and interpretations of the authors and editors, unless stated otherwise. This publication should not be construed to be a legal action of ENISA or the ENISA bodies unless adopted pursuant to the Regulation (EU) No 526/2013. This publication does not necessarily represent state-of-the-art and ENISA may update it from time to time.

Third-party sources are quoted as appropriate. ENISA is not responsible for the content of the external sources including external websites referenced in this publication.

This publication is intended for information purposes only. It must be accessible free of charge. Neither ENISA nor any person acting on its behalf is responsible for the use that might be made of the information contained in this publication.

Copyright Notice

© European Union Agency for Network and Information Security (ENISA), 2015

Reproduction is authorised provided the source is acknowledged.

¹ Dawid Osojca, Paweł Weźgowiec and Tomasz Chlebowski

² Don Stikvoort and Michael Potter

Table of Contents

1. Training introduction	5
2. Introduction to OllyDbg	6
2.1 OllyDbg interface	6
2.2 Basic debugging and code navigation	15
2.3 Breakpoints	22
2.4 Execution flow manipulation	29
2.5 Plugins	32
2.6 Shortcuts	34
3. Unpacking artefacts	36
3.1 Packers and protectors	36
3.1.1 Introduction to packers and protectors	36
3.1.2 Unpacking steps	37
3.1.3 Finding the OEP	37
3.2 Unpacking UPX packed sample	38
3.3 Unpacking UPX with the ESP trick	48
3.4 Unpacking a Dyre sample	52
4. Anti-debugging techniques	63
4.1 Anti-debugging and anti-analysis techniques	63
4.2 Dyre - basic patching with OllyDbg	65
5. Process creation and injection	70
5.1 Process injection and process hollowing	70
5.2 Following child processes of Tinba banking trojan	70
5.2.1 First stage	71
5.2.2 Second stage	80
6. Introduction to scripting	87
6.1 Introduction to OllyDbg scripting	87
6.2 Decoding hidden strings in Tinba	88
7. Summary	95

Main Objective	<p>The aim of this training is to present methods and techniques of dynamic artefact analysis with the use of OllyDbg³ debugger package.</p> <p>Trainees will be following a code execution and unpack artefacts using the most efficient methods. In addition they will be tracing a malicious code execution. During the process trainees will learn how to counter the anti-analysis techniques implemented by malware authors.</p> <p>In the second part the trainees will study various code injection techniques and how to debug hollowed processes. At the end of the training they will be presented how to automate the debugging process.</p> <p>The training is performed using the Microsoft Windows operating system.</p>
Targeted Audience	CSIRT staff involved with the technical analysis of incidents, especially those dealing with the sample examination and malware analysis. Prior knowledge of assembly language and operating systems internals is highly recommended.
Total Duration	8-10 hours
Frequency	Once for each team member.

³ OllyDbg <http://www.ollydbg.de/> (last accessed 11.09.2015)

1. Training introduction

In this training you will learn practical elements of advanced dynamic analysis and debugging of malicious code. Using a debugger to analyse artefacts helps you to understand how the malicious code operates and gives you more details than the behavioural analysis. Moreover, if the original sample is packed then unpack it first with the help of a debugger if necessary before proceeding with the static analysis.

This training begins with the introduction to the OllyDbg debugger (v1.10)⁴, which will be used throughout later exercises. In the second part you will learn about packers and protectors and how to use a debugger to unpack binary samples. In the third part you will learn about various anti-debugging and anti-analysis techniques. You will also be presented how to perform basic code patching using a sample of Dyre malware⁵. The fourth part teaches various code injection techniques and how to debug hollowed processes. Finally, the training ends with a short introduction to debugging automation using OllyDbg scripting capabilities.

Except the introductory part, the samples used in this training are live malware samples. Consequently all analyses should be done in dedicated and isolated environments. After each analysis a clean virtual machine snapshot should be restored if not instructed otherwise. An Internet connection is not needed to complete this training.

When debugging malicious code accidental clicks might lead to an uninterrupted code execution and as a result you might need to repeat the entire exercise. To prevent this it is advisable to take snapshots of virtual machines after analysing major code parts or taking breaks. This way even if something goes wrong, you won't need to repeat the entire process because you will just need to restore the last snapshot.

⁴ OllyDbg <http://www.ollydbg.de/> (last accessed 11.09.2015)

⁵ Dyre: Emerging threat on financial fraud landscape

http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/dyre-emerging-threat.pdf (last accessed 19.10.2015)

2. Introduction to OllyDbg

In this part you will be introduced to the OllyDbg⁶ interface and its basic usage. This will make you ready to complete the rest of exercises from the Advanced Dynamic Analysis training.

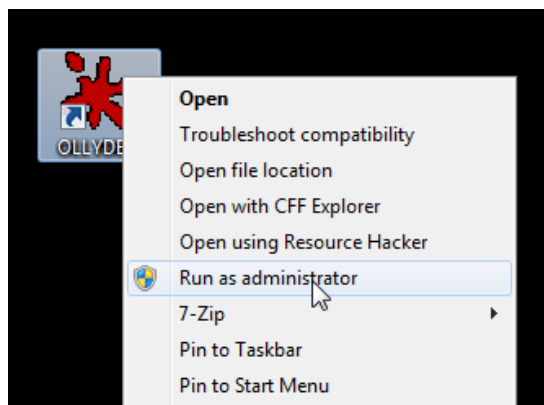
You will learn:

- How to use different views in OllyDbg
- How to navigate through the code
- Different methods of tracing executed instructions
- How to create different types of breakpoints
- How to manipulate execution flow of debugged program
- How to use plugins in OllyDbg

You will use the PuTTY executable⁷ which is a commonly used Secure Shell (SSH) client. This way you don't need to worry about accidentally execution and if it terminates you can execute it again without problems.

2.1 OllyDbg interface

First open OllyDbg debugger. Make sure to run it as Administrator.

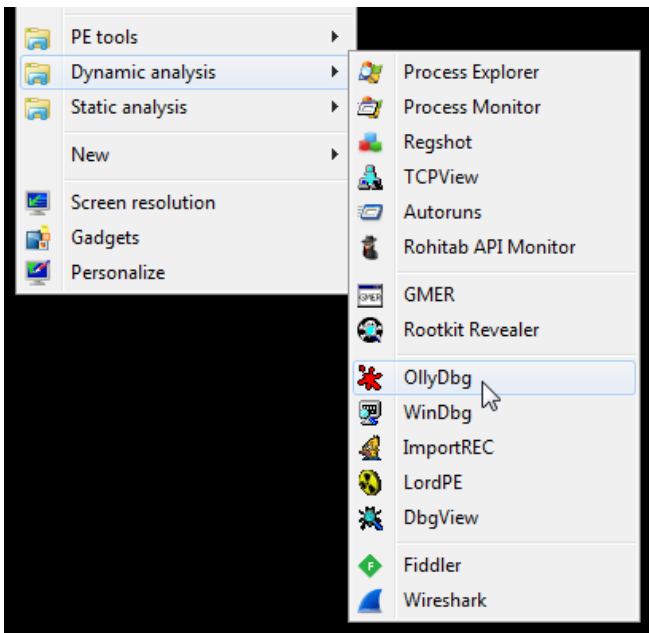


If you are using a Windows virtual machine prepared the same way as in the *Building artefact handling and analysis environment*⁸ training then you can also access OllyDbg using the context menu.

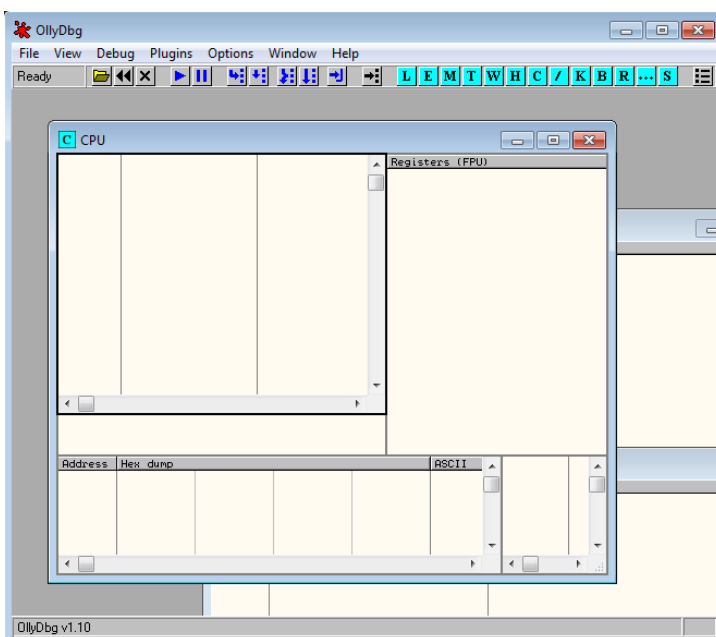
⁶OllyDbg <http://www.ollydbg.de/> (last accessed 11.09.2015)

⁷PuTTY: A Free Telnet/SSH Client <http://www.chiark.greenend.org.uk/~sgtatham/putty/> (last accessed 11.09.2015)

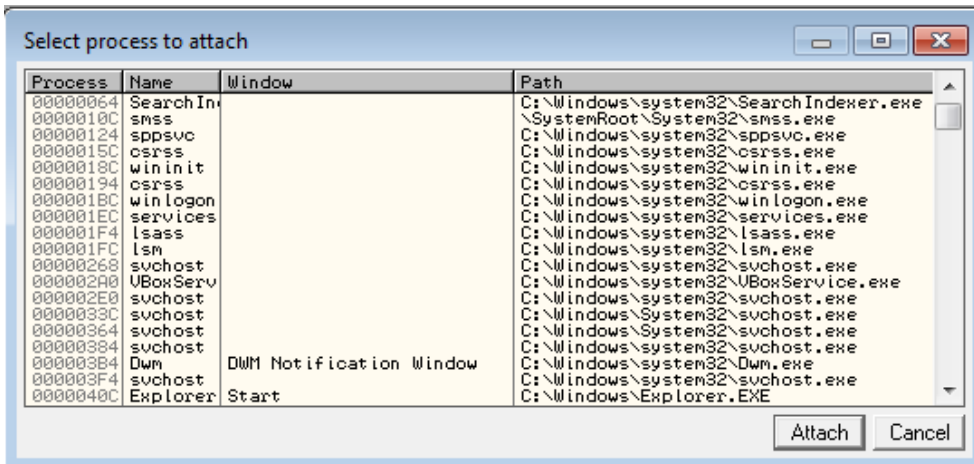
⁸Building artefact handling and analysis environment <https://www.enisa.europa.eu/activities/cert/training/training-resources/technical-operational#building> (last accessed 11.09.2015)



Now you should see the OlyDbg interface.

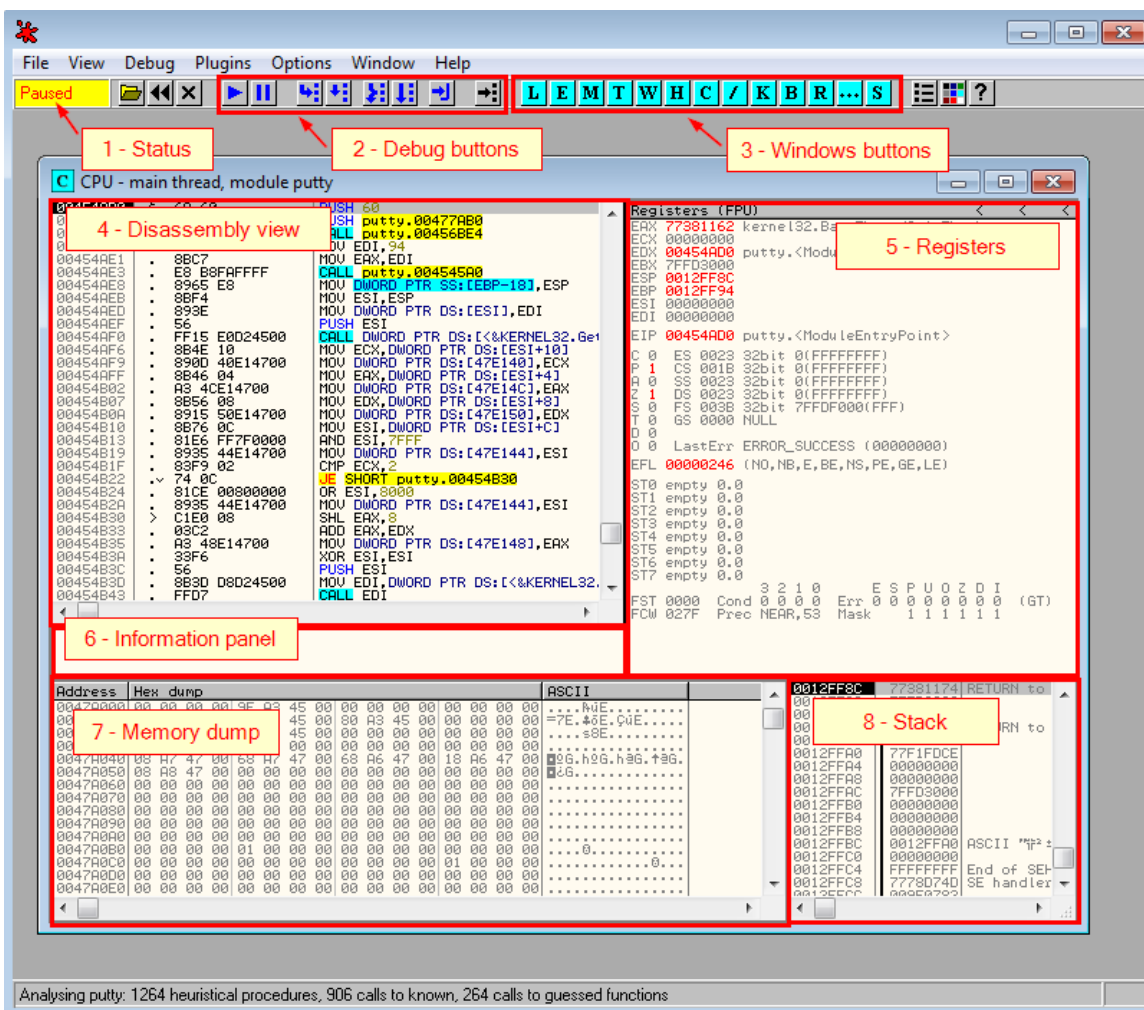


There are two ways to start a debugging process. Firstly, you can attach to the already running process. To do this, choose *File->Attach* and then choose the process of your interest. After attaching to the running process, OlyDbg should automatically break at the `ntdll.DbgBreakPoint` function.



The second way is to open an executable file using standard *File->Open* menu. This way OllyDbg will create a new child process with a debugged application (debuggee) and stop at the entry point of the executable (by default).

For example, open the **putty.exe** binary in OllyDbg. After a while OllyDbg should finish its initial analysis.



The Central part of OllyDbg is the CPU window. This is the window you will use most often during an analysis. It consists of five separate subpanels: *disassembly view*, *registers*, *information panel*, *memory dump* and *stack panel*.

Disassembly view (4) presents a listing with the disassembled code. It consists of four columns. The leftmost column shows the instruction address, the second column contains the hexadecimal representation of the instruction (machine code), the third column contains the assembly instruction and finally the fourth column is used to present comments and any additional information.

00454AEF	. 56	PUSH ESI	pVersionInformation = NULL
00454AF0	. FF15 E0D24500	CALL DWORD PTR DS:[&kernel32.GetVersionExA]	GetVersionExA
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	kernel32.BaseThreadInitThunk
00454B07	. 8B56 08	MOV EDX, DWORD PTR DS:[ESI+8]	
00454B0A	. 8915 50E14700	MOV DWORD PTR DS:[47E150], EDX	putty.<ModuleEntryPoint>
00454B10	. 8B76 0C	MOV ESI, DWORD PTR DS:[ESI+C]	

Registers view (5) presents the current state of CPU registers (for the currently selected thread).

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)

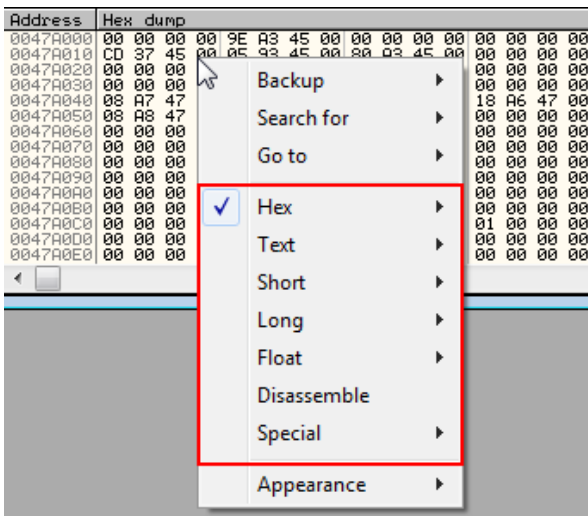
Information panel (6) is used to present additional information about the instruction selected in disassembly view (e.g. operation result, registers values).

EDI=00000000
EAX=77381162 (kernel32.BaseThreadInitThunk)

Memory dump (7) presents a dump of the chosen memory region.

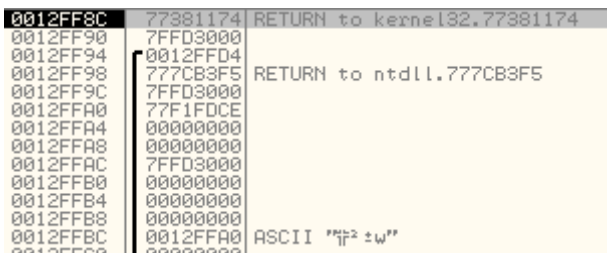
Address	Hex dump	ASCII
0047A000	00 00 00 00 9E A3 45 00 00 00 00 00 00 00 00 00RuE.....
0047A010	CD 37 45 00 05 93 45 00 80 A3 45 00 00 00 00 00	=7E.*5E.CuE....
0047A020	00 00 00 00 73 38 45 00 00 00 00 00 00 00 00 00s8E.....
0047A030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0047A040	08 A7 47 00 68 A7 47 00 68 A6 47 00 18 A6 47 00	8G.h0G.h3G.t3G.
0047A050	08 A8 47 00 00 00 00 00 00 00 00 00 00 00 00 00	8G.....
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

Besides the hexadecimal, you can choose other data representation formats by right-clicking on the memory dump panel and choosing required data representation from the context menu.

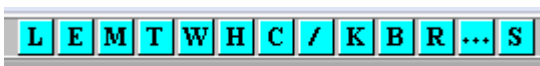


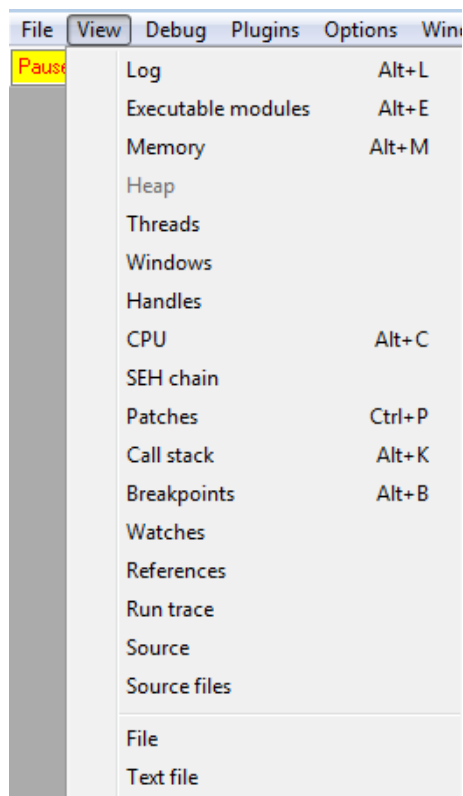
Take some time to check other data representations. At the end, restore the default format: *Hex->Hex/ASCII* (16 bytes).

Finally, *stack panel (8)* presents the stack state of the currently selected thread. The first column shows the memory address while the second column contains the value stored at the given stack address. Notice how the stack grows upward in the direction of lower memory addresses.



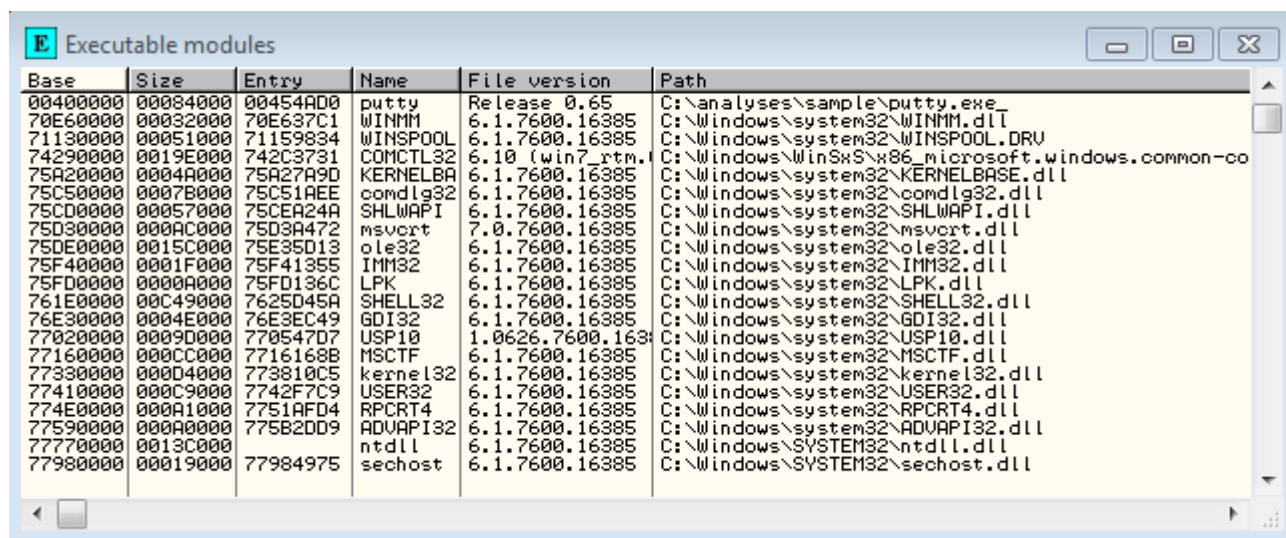
Besides the CPU window, OllyDbg offers few other windows used for different purposes. All windows can be accessed with windows buttons on the toolbar or *View* menu.



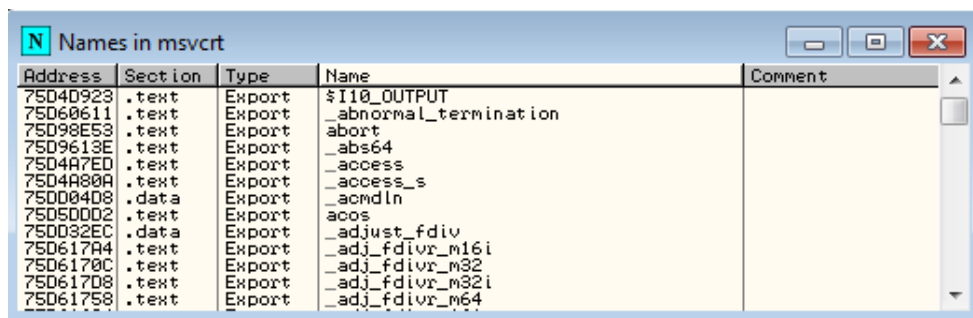


The more frequently used windows are: *Executable modules*, *Memory map*, *Threads*, *Handles*, *Call stack*, *Breakpoints*.

The *Executable modules* window presents all executable modules loaded in the address space of the debugged process. Usually, this would be a module of the executed binary and modules of loaded DLL libraries. You can double-click on any of the modules to immediately jump to this module in the disassembly view. You can also right-click on any of the modules to access context menu with additional operations.

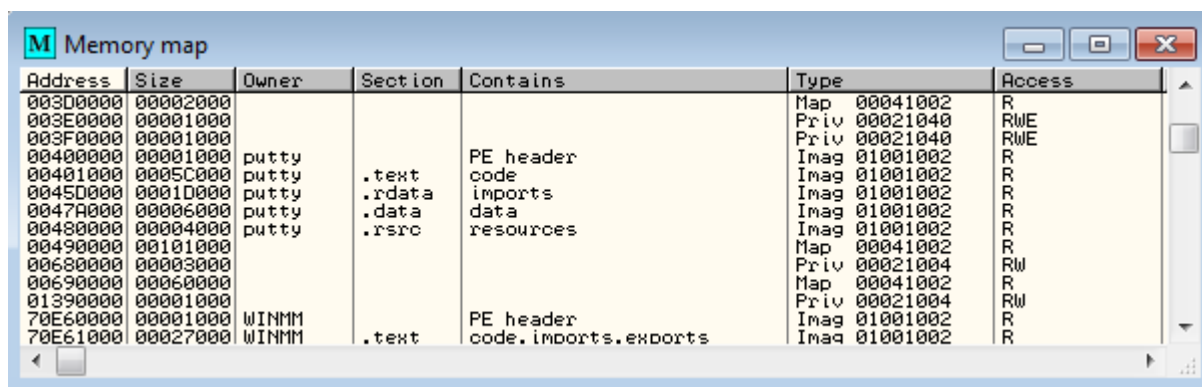


For example, right-click on *msvcrt* and choose *View names* to be presented with a list of all names defined in the *msvcrt* library (imports and exports).



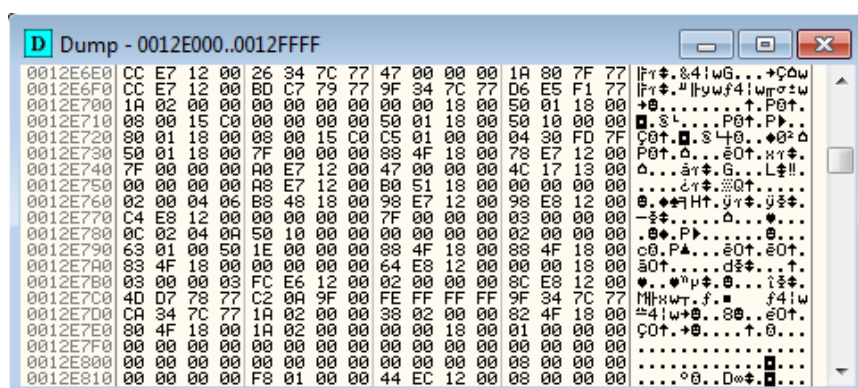
Address	Section	Type	Name	Comment
75040923	.text	Export	\$_I10_OUTPUT	
75060611	.text	Export	_abnormal_termination	
75098E53	.text	Export	abort	
7509613E	.text	Export	_abs64	
7504A7ED	.text	Export	_access	
7504A80A	.text	Export	_access_s	
75000408	.data	Export	_acmdln	
75050002	.text	Export	acos	
750032EC	.data	Export	_adjust_fdiv	
750617A4	.text	Export	_adj_fdivr_m16i	
7506170C	.text	Export	_adj_fdivr_m32	
75061708	.text	Export	_adj_fdivr_m32i	
75061758	.text	Export	_adj_fdivr_m64	

Memory map window presents the memory structure with all allocated memory regions in the address space of the debugged process. It is useful to track memory allocation operations done by the malicious code. Similarly as in the previous example you can right-click on any memory region to access the context menu with additional operations (dumping memory, searching memory, changing access rights, freeing memory, etc.).



Address	Size	Owner	Section	Contains	Type	Access
003D0000	00002000				Map 00041002	R
003E0000	00001000				Priv 00021040	RWE
003F0000	00001000				Priv 00021040	RWE
00400000	00001000	putty		PE header	Imag 01001002	R
00401000	0005C000	putty	.text	code	Imag 01001002	R
0045D000	0001D000	putty	.rdata	imports	Imag 01001002	R
0047A000	00006000	putty	.data	data	Imag 01001002	R
00480000	00004000	putty	.rsrc	resources	Imag 01001002	R
00490000	00101000				Map 00041002	R
00680000	00003000				Priv 00021004	RW
00690000	00060000				Map 00041002	R
01390000	00001000				Priv 00021004	RW
70E60000	00001000	WINMM		PE header	Imag 01001002	R
70E61000	00027000	WINMM	.text	code, imports, exports	Imag 01001002	R

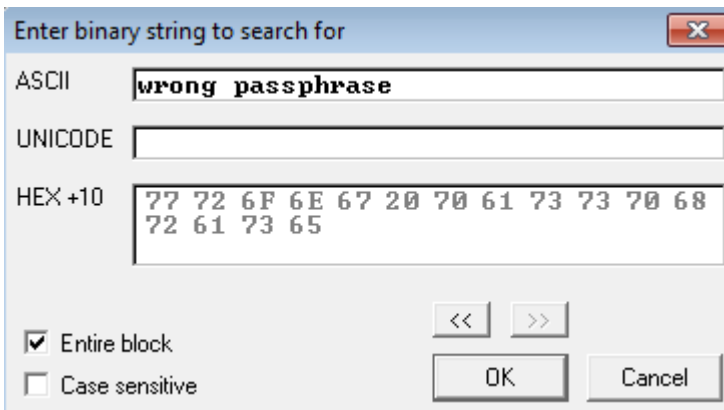
For example sometimes it is useful to open an additional dump window with a dump of the given memory region. To do this double-click on the memory region or select it and choose *Dump* option from the context menu.



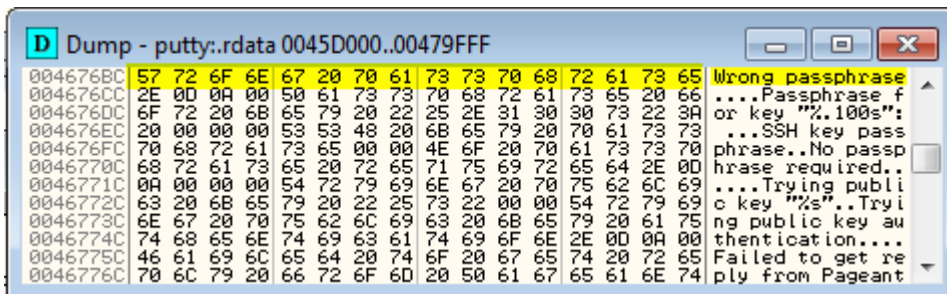
Address	Hex	ASCII
0012E6E0	CC E7 12 00 26 34 7C 77 47 00 00 00 1A 80 7F 77	f y + &4 w G...+ C w
0012E6F0	CC E7 12 00 BD C7 79 77 9F 34 7C 77 06 E5 F1 77	f y + &4 w f 4 w p o z w
0012E700	1A 02 00 00 00 00 00 00 00 00 18 00 50 01 18 00	+ 0+ P 0 +
0012E710	08 00 15 C0 00 00 00 00 50 01 18 00 50 10 00 00	0 .S .L.....P 0 + P 0 +
0012E720	80 01 18 00 08 00 15 C0 C5 01 00 00 04 30 FD 7F	C 0 + 0 .S .L 0 + 0 2 0
0012E730	50 01 18 00 7F 00 00 00 88 4F 18 00 78 E7 12 00	P 0 + 0 .e 0 + H y +
0012E740	7F 00 00 00 A0 E7 12 00 47 00 00 00 4C 17 13 00	0 ...a y + G ...L 0 !
0012E750	00 00 00 00 A8 E7 12 00 B0 51 18 00 00 00 00 00	0 ...c y + 0 0 +
0012E760	02 00 04 06 B8 48 18 00 98 E7 12 00 98 E8 12 00	0 ...0 + H T y + 0 0 0
0012E770	C4 E8 12 00 00 00 00 00 7F 00 00 00 03 00 00 00	- 0 + 00 ...0 ...0
0012E780	0C 02 04 0A 50 10 00 00 00 00 00 00 02 00 00 00	0 ...P 00 ...0 ...0
0012E790	63 01 00 50 1E 00 00 00 88 4F 18 00 88 4F 18 00	c 0 .P A ...e 0 + e 0 +
0012E7A0	83 4F 18 00 00 00 00 00 64 E8 12 00 00 00 18 00	e 0 + 0 ...d 0 + 0 + 0 +
0012E7B0	03 00 00 03 FC E6 12 00 02 00 00 00 8C E8 12 00	0 ...0 + p + 0 ...i 0 0
0012E7C0	4D 07 78 77 C2 0A 9F 00 FE FF FF FF 9F 34 7C 77	M H w T .f .0 ...f 4 w
0012E7D0	CA 34 7C 77 1A 02 00 00 38 02 00 00 82 4F 18 00	^ 4 w ^ 0 .8 0 ...e 0 +
0012E7E0	80 4F 18 00 1A 02 00 00 00 00 18 00 01 00 00 00	C 0 + ^ 0 ...+ 0 ...+ 0 ...0 ...0
0012E7F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000 ...0 ...0 ...0 ...0
0012E800	00 00 00 00 00 00 00 00 00 00 00 00 08 00 00 000 ...0 ...0 ...0 ...0
0012E810	00 00 00 00 F8 01 00 00 44 EC 12 00 08 00 00 000 ...D 0 + 0 ...0 ...0

Another operation you might try is searching all memory regions for a particular string or byte pattern. Let's say you know that somewhere in the memory the string 'wrong passphrase' is present, but you don't know the exact address nor in which memory region is it located.

To solve this problem, right-click anywhere in the memory map and choose *Search* (Ctrl+B) from the context menu. In the new window, type 'wrong passphrase' and click *Ok*.

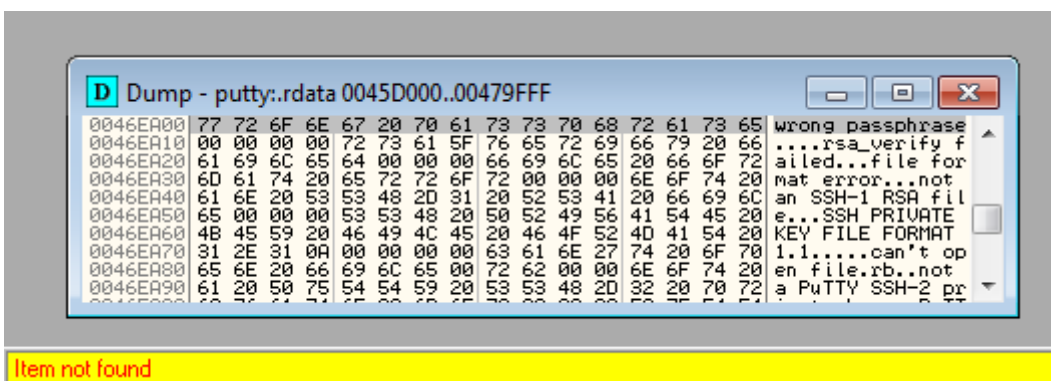


If the string is found OllyDbg will open a new Dump window with the position set on the string.



Here you see that the string was found at the virtual address `0x4676BC` which belongs to memory region `0x45D000-0x479FFF` (`putty:.rdata`).

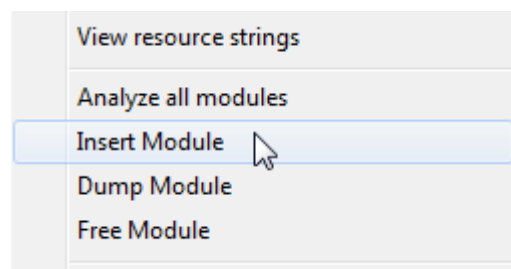
To keep searching for other occurrences of this string in this memory region click on *Dump* window (to make it active) and keep pressing `Ctrl+L`. When there is no more occurrences, OllyDbg will signal this with the *'Item not found'* message at the bottom of the window.



To continue searching for the string in other memory regions go back to *Memory map* window (make it active) and keep pressing `Ctrl+L`. If there is no more occurrences, OllyDbg will signal this with the same message at the bottom of the window.

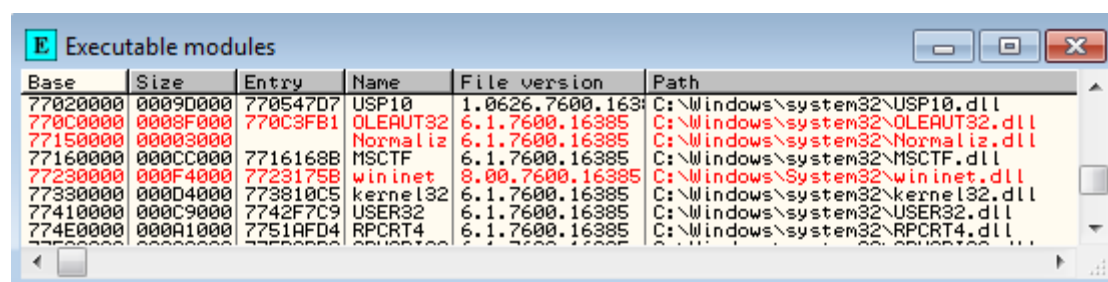
Threads window shows all threads of the current process. If the process has more than one thread, double-clicking on the thread would switch the context to this thread.

Next, right-click anywhere in the window and choose *Insert module* from the context menu (this operation is available only with Olly Advanced plugin).



In the *Open* dialog, choose *c:\Windows\System32\wininet.dll*. This way OllyDbg will load an additional module in the address space of the currently debugged process. Loading extra modules is sometimes useful in more advanced debugging when you want to load the DLLs with your custom code.

Now all the newly loaded modules should be marked with red font in the *Executable modules* window. Notice that besides the *wininet* module, a couple other DLLs were loaded. Those are the DLLs that were required by *wininet*.



The same rule of red-colouring new elements applies also to *Memory map* and various other views in OllyDbg. In general this is useful in tracking places in the code where new modules are loaded or new memory is allocated.

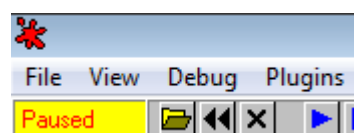
2.2 Basic debugging and code navigation

Start by loading the **putty.exe** sample as described in the previous exercise.

Each debugged process can be in one of the following states: *paused*, *running*, *terminated*, *tracing* and *animating*.

- *Paused* – program execution is paused, no instructions are being executed
- *Running* – program is freely running and debugger is not tracing its execution
- *Terminated* – debugged process has terminated
- *Tracing* – when instruction tracing was started (each executed instruction is logged)
- *Animating* – when instruction animation was started.

The current state of the debugged process can be read in the upper left corner of the OllyDbg window.



When the process is paused, the current position (the instruction pointer) is indicated by a black square in the disassembly view and by the value of EIP register.


```

CPU - main thread, module putty
00454AC7 . FF15 10AC4700 CALL DWORD PTR DS:[47AC10]
00454ACD . 59 POP ECX
00454ACE . 59 POP ECX
00454ACF . C3 RETN
00454AD0 . 6A 60 PUSH 60
00454AD7 . 68 B07A4700 PUSH putty.00477AB0
00454ADC . E8 08210000 CALL putty.00456BE4
00454AE1 . BF 94000000 MOV EDI,94
00454AE1 . 8BC7 MOV EAX,EDI

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
  
```

Whenever you get lost, double-click on the EIP register value to be instantaneously moved to the current position in the code. Remember that if the program has multiple threads, the current position will likely be different for each thread.

```

EDX 00454AD0 putty.<ModuleEntryPoint>
EBX 7FFD6000
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)
  
```

double click

When debugging a program you will spend most of the time on analysing disassembled instructions step-by-step. There are two modes of instruction stepping:

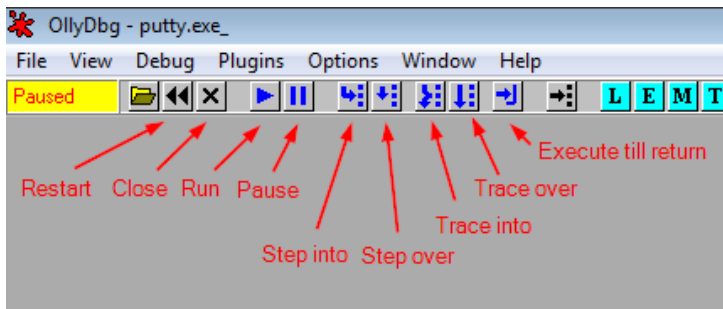
- *Step into* (F7) – executes current instruction and moves program execution to the next instruction. If the current instruction is a function call then the debugger steps into the call and starts stepping over instructions of the called function.
- *Step over* (F8) – behaves the same as *Step into* except if the current instruction is a function call, the debugger doesn't step into this call.

If you want to let the program run freely choose *Run* (F9). In the result, PuTTY will create its main window and present it to the user. If you want to pause the program execution then press F12 (*Debug->Pause*) while staying in OllyDbg. You can also restart the executable by pressing Ctrl+F2 (*Debug->Restart*).

Other useful debug operations are:

- *Run to selection* (F4) – causes OllyDbg to resume execution until the selected instruction
- *Execute till return* (Ctrl+F9) – executes the program until return from current function
- *Execute till user code* (Alt+F9) – executes program until user code

Debugging actions can be also accessed through the toolbar at the top of OllyDbg.



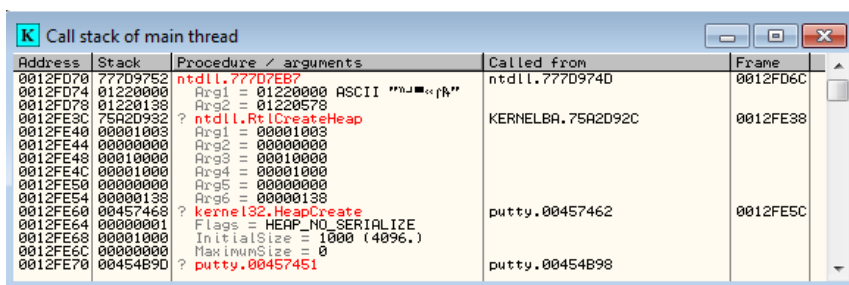
If you want to quickly pre-view the execution flow of a program (find loops, check which jumps are taken, etc.) you might decide to use the instruction trace or instruction animation functions. Both functions come in two forms: *Trace into/Trace over* and *Animate into/Animate over*.

Debug	Plugins	Options	Window	Help
Run				F9
Pause				F12
Restart				Ctrl+F2
Close				Alt+F2
Step into				F7
Step over				F8
Animate into				Ctrl+F7
Animate over				Ctrl+F8
Execute till return				Ctrl+F9
Execute till user code				Alt+F9
Open or clear run trace				
Trace into				Ctrl+F11
Trace over				Ctrl+F12
Set condition				Ctrl+T
Close run trace				

To see how the instruction animation works, restart PuTTY sample (*Debug->Reset*) and then choose *Debug -> Animate over* (Ctrl+F8). Observe what happens in the disassembly window.

You should see a short animation of executed instructions and after a few moments PuTTY's main window should appear.

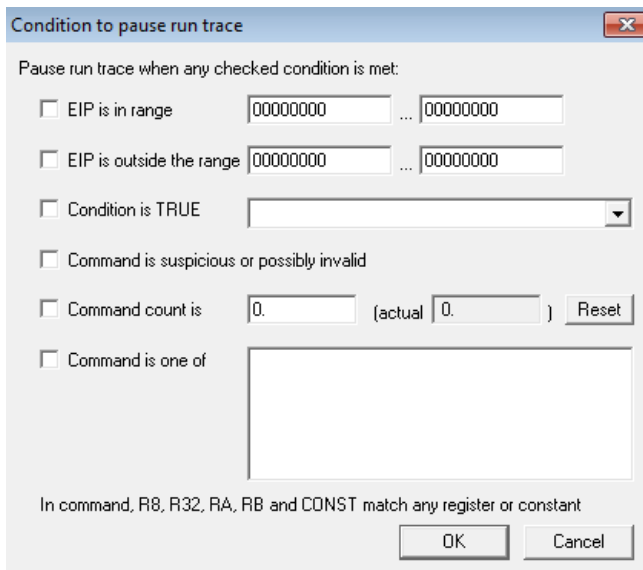
Close PuTTY and reset the sample. Now choose *Debug->Animate into*. This time instead of stepping over, the animation will step into each function call (including API calls). You can open the *Call stack* (Alt+K) window to observe all called functions in the real time.



Address	Stack	Procedure / arguments	Called from	Frame
0012FD70	77709752	ntdll.777D7EB7?	ntdll.777D9740	0012FD6C
0012FD74	01220000	Arg1 = 01220000 ASCII "m...jR"		
0012FD78	01220138	Arg2 = 01220578		
0012FE3C	75A2D932	? ntdll.RtlCreateHeap	KERNELBA.75A2D92C	0012FE38
0012FE40	00001003	Arg1 = 00001003		
0012FE44	00000000	Arg2 = 00000000		
0012FE48	00010000	Arg3 = 00010000		
0012FE4C	00001000	Arg4 = 00001000		
0012FE50	00000000	Arg5 = 00000000		
0012FE54	00000138	Arg6 = 00000138		
0012FE60	00457468	? kernel32.HeapCreate	putty.00457462	0012FE5C
0012FE64	00000001	Flags = HEAP_NO_SERIALIZE		
0012FE68	00001000	InitialSize = 1000 (4096.)		
0012FE6C	00000000	MaximumSize = 0		
0012FE70	00454B90	? putty.00457451	putty.00454B98	

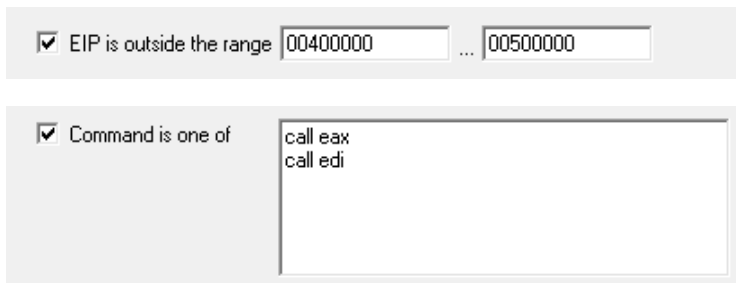
Animate into function usually takes some time until the program finishes execution. To stop it, use *Pause* (F12) function.

Next restart the sample again and choose *Debug->Set condition*.



In this dialog you can set conditions on how long the *Run trace* function should be running (conditions set here would also work for *animate* function). If you set more than one condition, *run trace* will be running until one of those conditions is met. It is important to note that if the condition is met inside the body of some called function and you are using *Trace over* function, it will not work.

For example set the following two conditions.



This would make instruction tracing stop either when the execution moves outside of the memory range *0x400000-0x500000*, or when the current command would be *call eax* or *call edi*.

Now open *Run trace* window (*View->Run trace*) and then choose *Debug->Trace over*. Execution should soon stop at the *call edi* instruction.

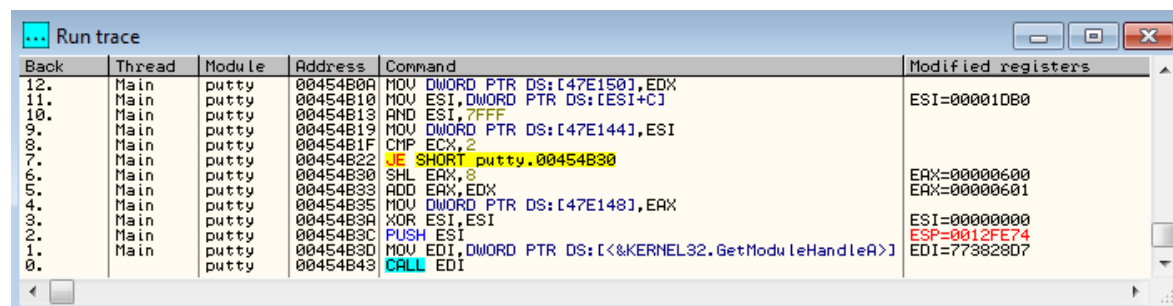
00454B3A	. 39F6	XOR ESI,ESI
00454B3C	. 56	PUSH ESI
00454B3D	. 8B3D 08D24500	MOV EDI,DWORD PTR DS:[&KERNEL32.GetModu
00454B43	. FF07	CALL EDI
00454B45	. 66:8138 4D5A	CMP WORD PTR DS:[EAX],5A4D
00454B4A	. 75 1F	JNZ SHORT putty.00454B6B
00454B4C	. 8B48 3C	MOV ECX,DWORD PTR DS:[EAX+3C]

This would also be indicated at the OllyDbg status bar in the bottom left corner.

Conditional pause: Command is call edi

Notice that execution hasn't stopped on the condition of *EIP* register being outside of the given memory range even though there were some API calls already made in the code. This is because you used the *Trace over* function and the API calls were stepped over. If you had used the *Trace into* function, execution would stop at the first API call.

Now take a look at the *Run trace* window. It contains all executed instructions with information about the instruction address, thread and modified registers. The last executed instruction is at the bottom of the window.



If you would like run trace to be logged to a file you should right-click on *Run trace* window and choose the *Log to file* option from the context menu (before executing *Run trace* function).

At this point you should know the basic debugging operations and functions. The next important thing to learn is how to navigate through the code.

First restart the PuTTY sample.

```

00454A00 $ 6A 60      PUSH 60
00454A02 . 68 B07A4700  PUSH putty.00477AB0
00454A07 . E8 08210000  CALL putty.00456BE4
00454ADC . BF 94000000  MOV EDI,94
00454AE1 . 8BC7        MOV EAX,EDI
00454AE3 . E8 B0FAFFFF  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

```

Whenever you see some call or jump instruction you can follow it (without executing) by clicking on this instruction and pressing *<Enter>*.

In this example follow a call to *putty.004545A0*. You should land at the function body.

```

0045459E | CC      INT3
0045459F | CC      INT3
004545A0 | $ 3D 00100000  CMP EAX,1000
004545A5 | .v 73 0E      JNB SHORT putty.004545B5
004545A7 | . F7D8     NEG EAX
004545A9 | . 03C4     ADD EAX,ESP
004545AB | . 83C0 04   ADD EAX,4
004545AE | . 8500     TEST DWORD PTR DS:[EAX],EAX

```

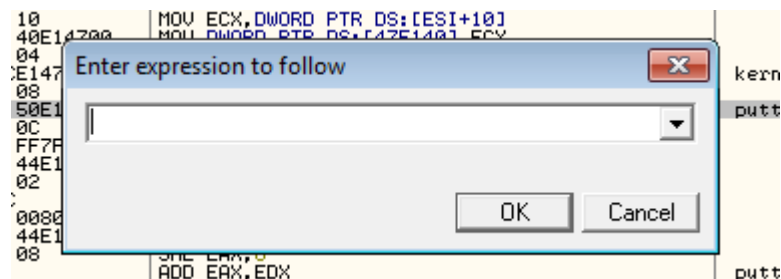
You can do the same with jump instructions.

One of the drawbacks of following calls and jumps in OllyDbg is the lack of a “Go back” function. That is, if you follow some jumps and calls, there is no easy way of going back to the previous position in a way that IDA Pro / IDA Free⁹ allows. You just need to remember what code you have followed or use the *Bookmarks* plugin (you will learn more about plugins in a later part of the exercise).

⁹ Freeware version of IDA v5.0 https://www.hex-rays.com/products/ida/support/download_freeware.shtml (last accessed 11.09.2015)

Another way of navigating through the code is using the *Go to expression* feature. It can be used to change the current position in disassembly view, memory dump or stack view – depending on which view is active.

Click on disassembly view and press Ctrl+G.

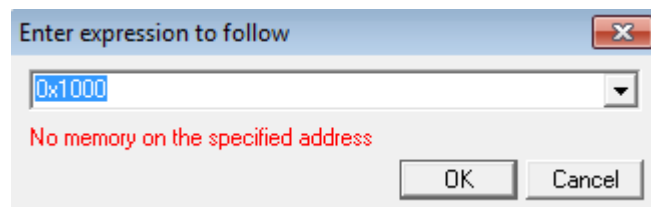


Type *eip* to be moved to the current location in the code (pointed by EIP register).

In *Enter expression to follow* dialog you can enter a wide range of expressions:

- registers: *eax, ebx, ecx*
- memory addresses: *0x401000*
- arithmetic expressions: *0x400000+2*0x1002, eax+0x1000*
- API functions names: *CreateFileA, WriteProcessMemory*
- Labels or other names used in program.

If the entered expression is invalid or the destination address doesn't exist in the address space of the debugged process you will see a proper error message.

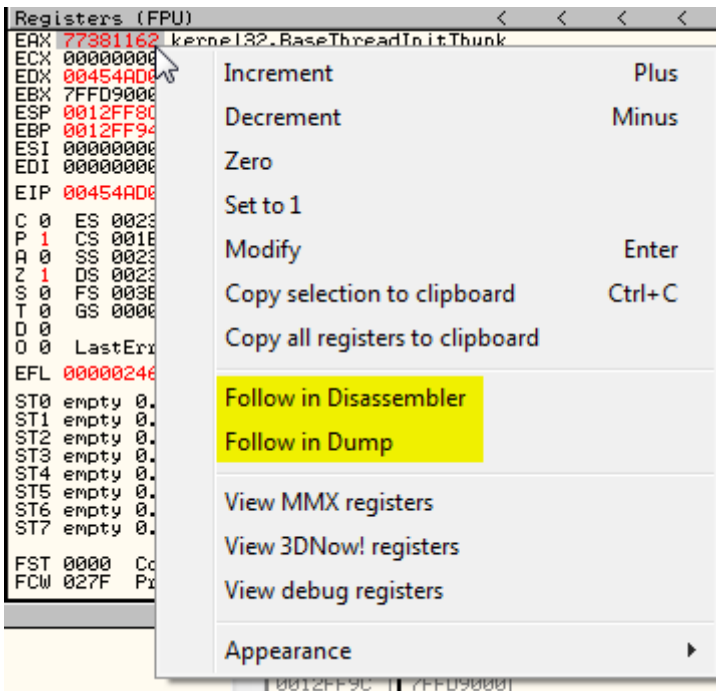


Additionally, if you want to find the address of a certain API function, but the module in which this function is located hasn't been loaded yet (it is being loaded at runtime as it is going to be called) you will also see an error message (*Unknown identifier*).

Another often used way of code navigation in OllyDbg is through context menus. You can click on various values in OllyDbg (register values, immediate values, stack stored values, strings) and in the context menu there will often be options like:

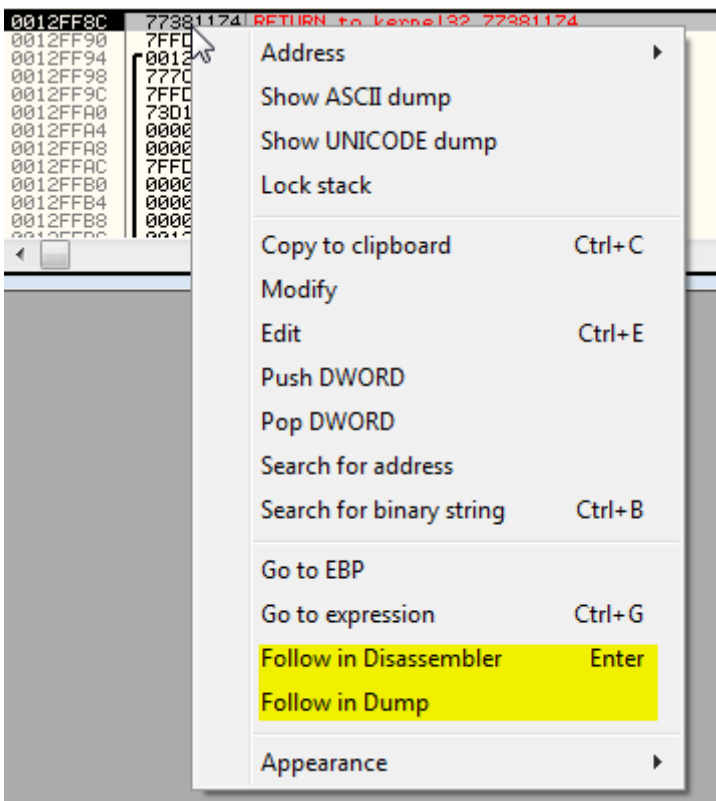
- *Follow in Disassembler*
- *Follow in Dump*
- *Follow in Stack*

For example, start clicking on registers values. If a register points to the existing address in the address space of the current program, there should be the following options: *Follow in Disassembler* and *Follow in Dump*.



If the register does not contain a valid address, these options won't be available. Additionally if a register points to the location on the stack (like in case of ESP register) there will be an option *Follow in Stack*.

You can do the same with values stored on stack.



2.3 Breakpoints

Breakpoints are crucial parts of any debugger. They allow to stop the program execution at a chosen moment allowing the user to analyse specific program functions.

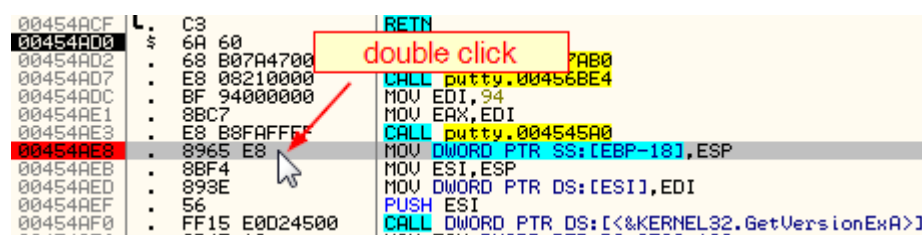
There are four types of breakpoints in OllyDbg¹⁰:

- Software breakpoints (INT 3 breakpoints)
- Hardware breakpoints
- Memory breakpoints
- Guarded pages

Software breakpoints work by inserting an INT 3¹¹ instruction in the place of the instruction on which the breakpoint is set. When the instruction is about to be executed, the interrupt is raised and the debugger steps in. The entire process is transparent to the user.

Setting software breakpoints actually modifies memory of debugged process. Thus when the debugged process was about to calculate the checksum of its own code, it might be different than expected. Some malicious code uses this as one of the anti-debugging techniques to detect if they are being debugged.

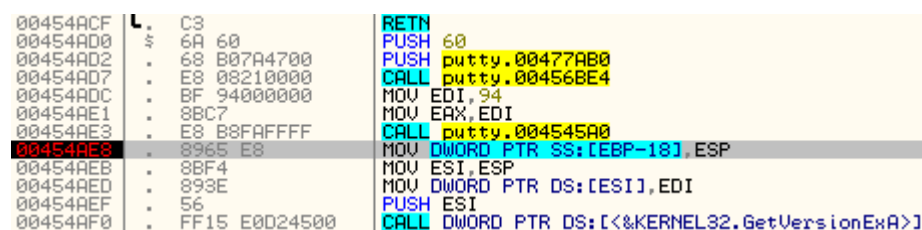
To set a software breakpoint, double-click in the second column next to the instruction or select an instruction and press F2. When the breakpoint is set this will be indicated by a red background of the instruction address.



```

00454ACF | L.  C3          | RETN
00454AD0 | $  6A 60       |
00454AD2 | .  68 B07A4700 |
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.GetVersionExA>]
  
```

Now press F9 (run) and the program should stop execution on this instruction (before executing it).



```

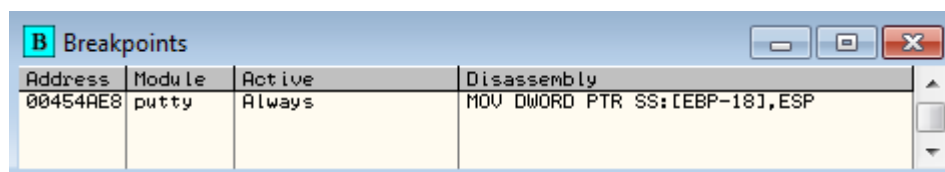
00454ACF | L.  C3          | RETN
00454AD0 | $  6A 60       | PUSH 60
00454AD2 | .  68 B07A4700 | PUSH putty.00477AB0
00454AD7 | .  E8 08210000 | CALL putty.00456BE4
00454ADC | .  BF 94000000 | MOV EDI,94
00454AE1 | .  8BC7        | MOV EAX,EDI
00454AE3 | .  E8 B8FAFFFF | CALL putty.004545A0
00454AE8 | .  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.GetVersionExA>]
  
```

To remove a breakpoint, repeat the same steps as when setting it.

You can view a list of all software breakpoints in the *Breakpoints* window.

¹⁰ http://www.ollydbg.de/Help/i_Breakpoints.htm (last accessed 11.09.2015)

¹¹ The INT 3 instruction is defined for use by debuggers to temporarily replace an instruction in a running program in order to set a breakpoint. [https://en.wikipedia.org/wiki/INT_\(x86_instruction\)](https://en.wikipedia.org/wiki/INT_(x86_instruction)) (last accessed 11.09.2015)

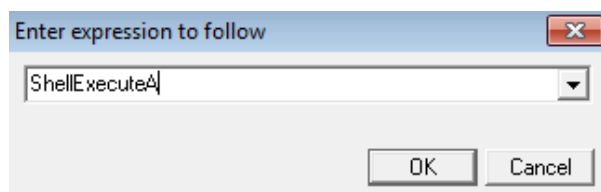


You can also use this window to remove or temporarily disable chosen breakpoints.

One way of using breakpoints is to set them on API functions. This allows to detect when a certain API function is called by malicious code and can be used to detect various operations done by malware. For example if you are interested in communication with C&C servers it is a good idea to set breakpoints on network related functions. And if you suspect that the process is injecting some code to other processes, you might set breakpoints on functions such as *WriteProcessMemory* or *CreateRemoteThread*.

Now you will set a breakpoint on *ShellExecuteA* function.

First click on disassembly view and use *Go to expression* (Ctrl+G) to find the address of *ShellExecuteA*.

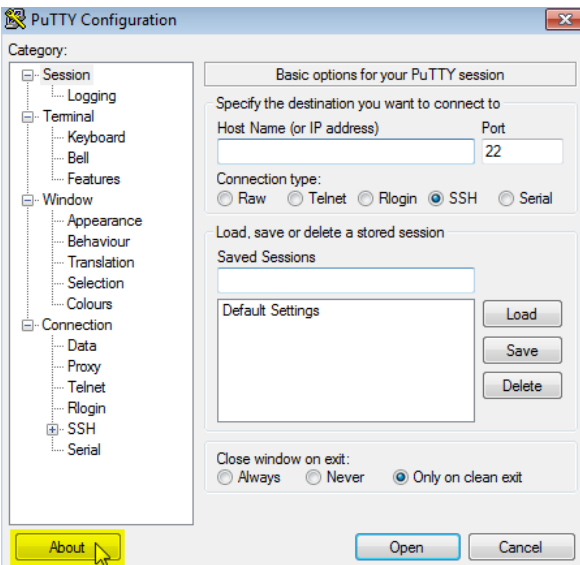


Then set breakpoint on the first instruction of *ShellExecuteA* (the one to which you were moved).

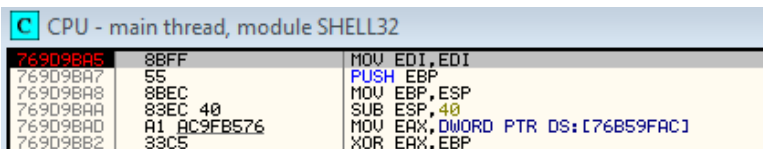
769D9BA5	8BFF	MOV EDI,EDI
769D9BA7	55	PUSH EBP
769D9BA8	8BEC	MOV EBP,ESP
769D9BAA	83EC 40	SUB ESP,40
769D9BAD	A1 AC9FB576	MOV EAX,DWORD PTR DS:[76B59FAC]
769D9BB2	33C5	XOR EAX,EBP
769D9BB4	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
769D9BB7	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]
769D9BBA	8B4D 0C	MOV ECX,DWORD PTR SS:[EBP+C]
769D9BBD	8B55 10	MOV EDX,DWORD PTR SS:[EBP+10]

If the PuTTY process was paused, resume execution (F9).

Next in the PuTTY window, click the *About* button and then the *Visit Web Site* button.

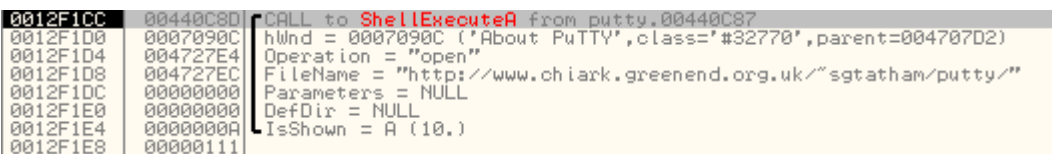


Now go back to OllyDbg. OllyDbg should break on a call to *ShellExecuteA* (on the previously set breakpoint).



Breakpoint at SHELL32.ShellExecuteA

Take a look at the stack view to see arguments passed to *ShellExecuteA*.



As you can see, after clicking *Visit Web Site*, PuTTY tries to open the http address <http://www.chiark.greenend.org.uk/~sgtatham/putty/> in the default system web browser.

You can also open the call stack window (*View->Call stack*, *Alt+K*) to check from where *ShellExecuteA* function was called.

Address	Stack	Procedure / arguments	Called from	Frame
0012F1CC	00440C80	SHELL32.ShellExecuteA	putty.00440C87	0012F1EC
0012F1D0	0007090C	hwnd = 0007090C ('About PuTTY',class='#32770',		
0012F1D4	004727E4	Operation = "open"		
0012F1D8	004727EC	FileName = "http://www.chiark.greenend.org.uk/		
0012F1DC	00000000	Parameters = NULL		
0012F1E0	00000000	DefDir = NULL		
0012F1E4	0000000A	IsShown = A (10.)		
0012F1F0	761286EF	putty.00440C2B	USER32.761286EC	0012F1EC
0012F1F4	0007090C	Arg1 = 0007090C		
0012F1F8	00000111	Arg2 = 00000111		
0012F1FC	000003EE	Arg3 = 000003EE		
0012F200	00070926	Arg4 = 00070926		
0012F21C	7611BAF1	? USER32.761286CC	USER32.7611BAEC	0012F218
0012F298	7611B98B	? USER32.7611BA48	USER32.7611B986	0012F294
0012F2E0	761390F9	USER32.7611B90C	USER32.761390F4	0012F2DC
0012F2FC	761286EF	Includes USER32.761390F9	USER32.761286EC	0012F2F8
0012F328	76128876	? USER32.761286CC	USER32.76128871	0012F324
0012F32C	761390D7	Includes USER32.76128876	USER32.761390D1	0012F39C

The second type of breakpoints are hardware breakpoints. In general, instead of changing program instructions in the memory as software breakpoints do, they use special processor registers (debug registers). On the x86 architecture there are four debug registers (DR0-DR3) used to store the linear address of breakpoints. Thus it is possible to set four hardware breakpoints at a time. Additionally, in contrast to software breakpoints, hardware breakpoints can be also used to break on memory read or write operations.

Hardware breakpoints are usually used when you want to detect when a certain memory address is being written to or when you know that the malicious code is trying to detect software breakpoints.

To get more information on differences between software and hardware breakpoints refer to the *Debugger flow control*¹²¹³ articles by Ken Johnson.

Now let's see how to set up hardware breakpoints: go to OllyDbg and restart the PuTTY sample.

00454A00	6A 60	PUSH 60	
00454A02	68 B07A4700	PUSH putty.00477AB0	
00454A07	E8 08210000	CALL putty.00456BE4	
00454A0C	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.GetVersionExA]	lpVersionInformation = NULL
00454AF6	8B4E 10	MOV ECX,DWORD PTR DS:[ESI+10]	GetVersionExA
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	kernel32.BaseThreadInitThunk

Next, step over until the instruction at 0x454AF9. As you can see some dword value is being written to the memory at the address 0x47E140.

```
ECX=00000002
DS:[0047E140]=00000000
```

Let's say you want to check at what place in the code this value will be used again.

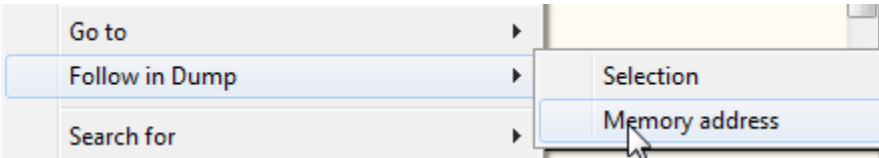
Right-click on this instruction and from the context menu choose *Follow in Dump->Memory address*.

¹²Debugger flow control: Hardware breakpoints vs software breakpoints <http://www.nynaeve.net/?p=80> (last accessed 11.09.2015)

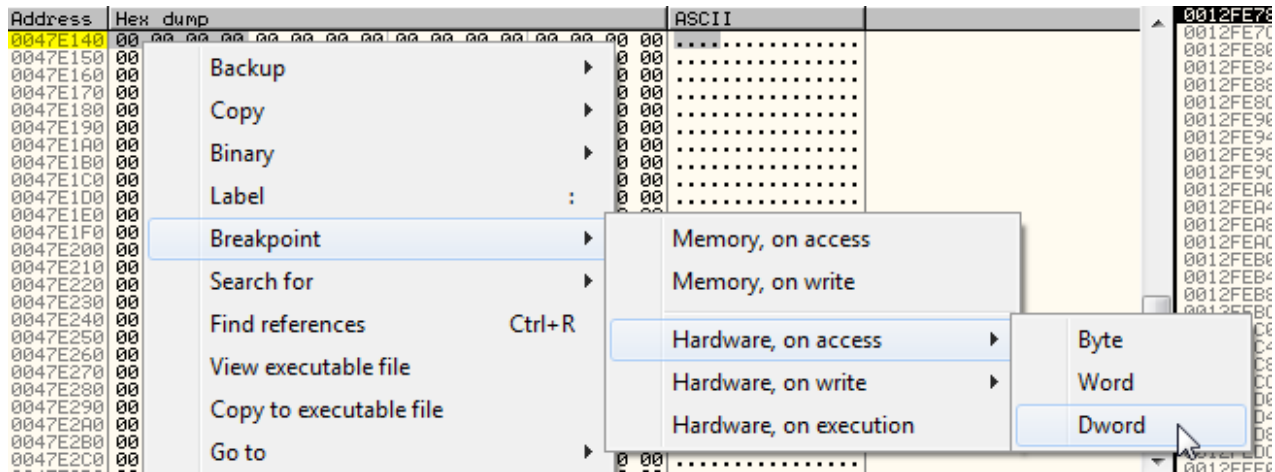
¹³Debugger flow control: More on breakpoints (part 2) <http://www.nynaeve.net/?p=81> (last accessed 11.09.2015)

```

00454AF0 . FF15 E0D24500 CALL DWORD PTR DS:[&&KERNEL32.GetVersionExA] GetVersionExA
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
  
```

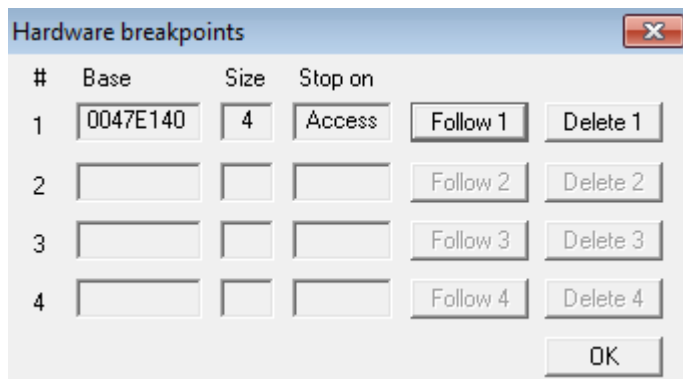


Now *Memory Dump* view should be centred on the *0x47E140* address. Select the first 4 bytes (dword) and right-click on them. From the context menu choose *Breakpoint->Hardware, on access->Dword*.



Now if at any place of the code this memory address would be accessed, the hardware breakpoint will hit and the program execution will be paused.

To view all currently set hardware breakpoints, choose *Debug->Hardware breakpoints*.



You can use this window to follow the memory address where the hardware breakpoint is set, or to delete the breakpoint.

After setting up a hardware breakpoint on *0x47E140*, resume the program execution (F9).

Almost immediately the program should break. As the message in the status bar shows, hardware breakpoint 1 was hit and EIP points to one instruction after `0x47E140` address was accessed.

Hardware breakpoint 1 at putty.0045743E - EIP points to next instruction

Scroll the disassembly view one line up to see the instruction accessing `0x47E140`.

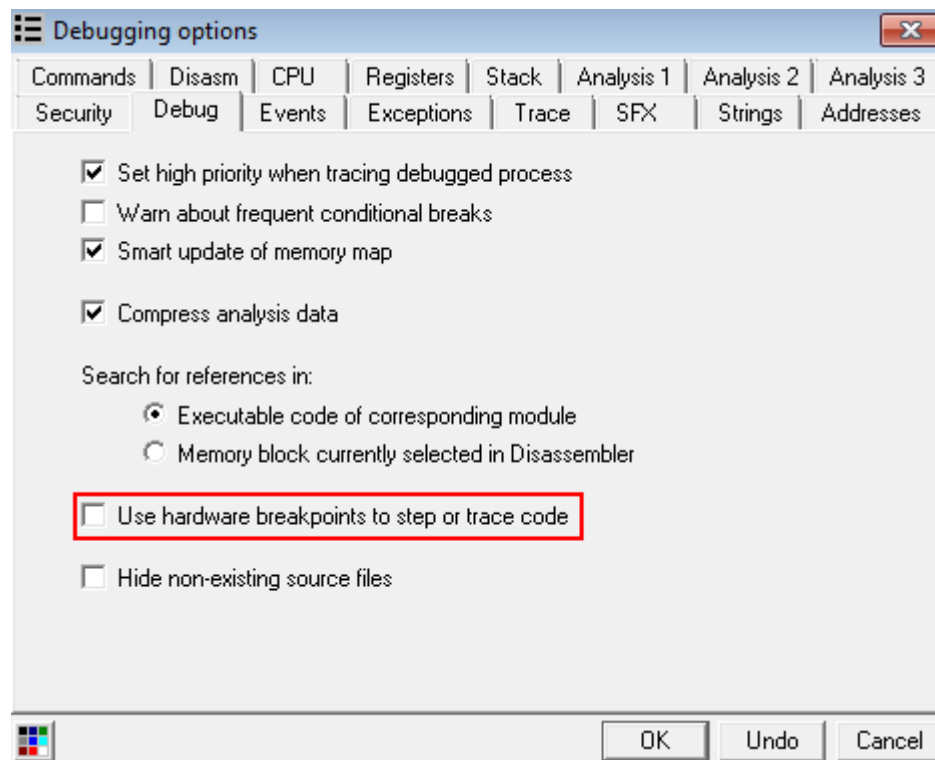
```

00457436 | . C3 RETN
00457437 | $ 833D 40E14700 02 CMP DWORD PTR DS:[47E140],2
0045743E | v 75 00 JNZ SHORT putty.00457440
00457440 | . 833D 4CE14700 05 CMP DWORD PTR DS:[47E14C],5
00457447 | v 72 04 JB SHORT putty.0045744D
00457449 | . 33C0 XOR EAX,EAX
0045744B | . 40 INC EAX
0045744C | . C3 RETN
0045744D | > 6A 03 PUSH 3

```

You can now remove the hardware breakpoint (it is not automatically removed after the sample reload).

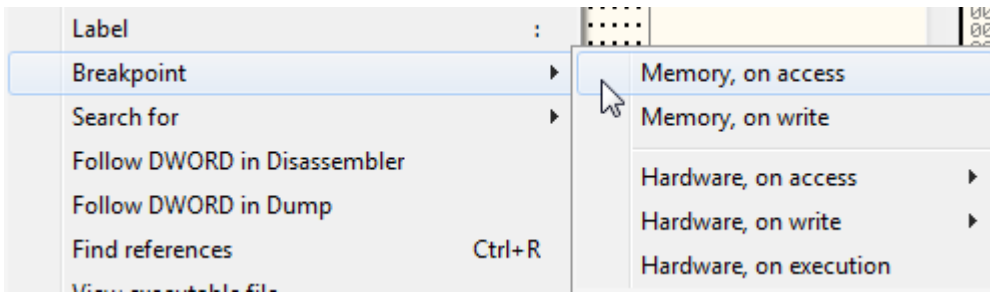
Hardware breakpoints can be used instead of software breakpoints, for instruction stepping or tracing. To configure this go to *Options-Debugging options->Debug* and select *“Use hardware breakpoints to step or trace code”*. Don’t select this option right now however, since in the remaining part of this training software breakpoints are used!



The third type of breakpoints are memory breakpoints. They can be used to detect memory read or write operations. They are set for memory pages and it is not possible to set them only for a byte, word or dword memory range. This makes them less accurate than hardware breakpoints but in contrast to hardware breakpoints, the number of memory breakpoints is not limited.

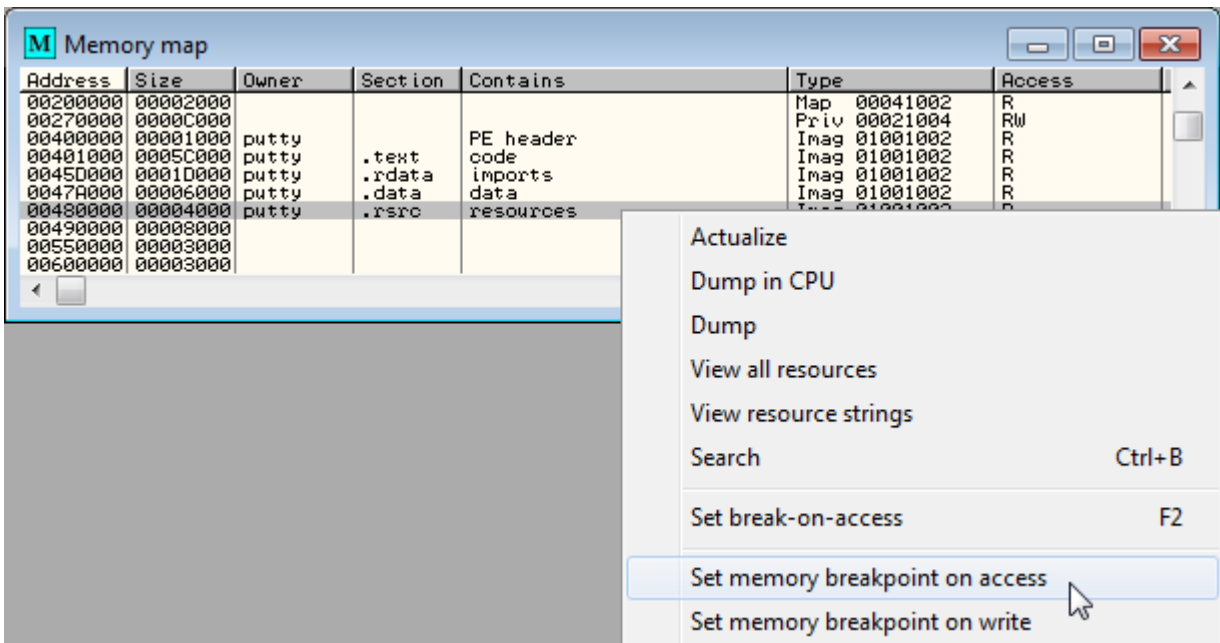
Typical usage for memory breakpoints is the detection of read or write operations on large memory blocks (for example newly allocated memory).

You can set memory breakpoints in a similar manner as hardware breakpoints by selecting some data in *Memory Dump* view and then choosing *Breakpoint->Memory*.

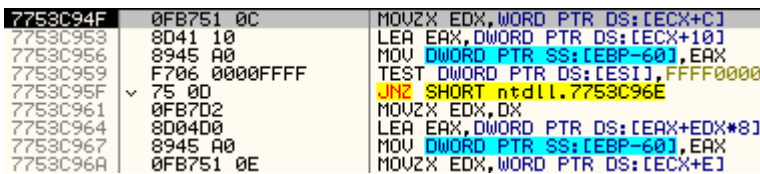


The second way of creating a memory breakpoint is using *Memory map* window.

Restart the PuTTY sample and open *Memory map* window. Then find PuTTY's *.resource* section, right-click it and from the context menu, choose *Set memory breakpoint on access*. Now if some code tries to access any data in *.resource* section, the breakpoint would hit.



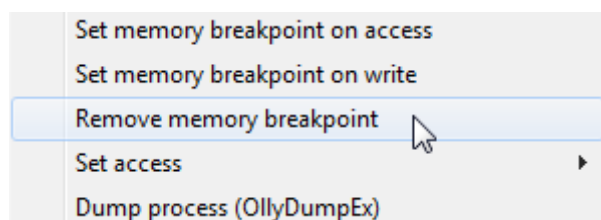
Next, resume the program (F9). The breakpoint should hit someplace in the system code.



If you check *Call stack* window you will see that the breakpoint was hit after a call to *CreateDialogParamA* from which *FindResourceExA* was called.

Address	Stack	Procedure / arguments	Called from	Frame
0012FD1C	77551D66	? ntdll.7753C8B5	ntdll.77551D61	0012FD18
0012FD3C	7597D519	? ntdll.LdrFindResource_U	kernel32.7597D513	0012FD38
0012FD94	76133E92	? kernel32.FindResourceExA	USER32.76133E8C	0012FD90
0012FD98	00400000	hModule = 00400000 (putty)		
0012FD9C	00000005	ResourceType = RT_DIALOG		
0012FDA0	0000006F	ResourceName = 6F		
0012FDA4	00000000	LanguageId = 0 (LANG_NEUTRAL)		
0012FDB4	0044154D	? USER32.CreateDialogParamA	putty.00441547	0012FDB0
0012FDB8	00400000	hInst = 00400000		
0012FDBC	0000006F	pTemplate = 6F		
0012FDC0	00000000	hOwner = NULL		
0012FDC4	00440E30	pDlgProc = putty.00440E30		
0012FDC8	00000000	lParam = 0		
0012FDD0	00448864	? putty.00441535	putty.0044885F	
0012FE64	00454C5B	? putty.0044882D	putty.00454C56	

To remove a memory breakpoint, go to the *Memory map* window, right-click on the memory region on which the memory breakpoint was set and select *Remove memory breakpoint*.



2.4 Execution flow manipulation

Besides the instruction stepping and execution flow analysis, debugging also allows you to change how a program actually executes. It is possible to change almost any aspect of program execution. OllyDbg allows you to overwrite executed instructions, change registers values, change FLAGS register as well as modify data on the stack or at any other memory address.

This might be useful to overcome some anti-analysis techniques or to check how malicious code would behave in other circumstances. However, any code or register manipulation must be done with care because otherwise it may lead to a crash of the debugged program.

Examples presented in this exercise are only intended to present how to do the execution flow manipulation and are not conducting any meaningful change.

First, restart the PuTTY sample and step over until the first jump instruction.

00454B19	. 8935 44E14700	MOV DWORD PTR DS:[47E144],ESI
00454B1F	. 83F9 02	CMP ECX,2
00454B22	. 74 0C	JE SHORT putty.00454B30
00454B24	. 81CE 00000000	OR ESI,0000
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

The red arrow next to the instruction tells that a jump will be made (this might be different on different systems).

You can force this jump not to be taken by changing then appropriate flag in the FLAGS register.


```
EIP 00454B22 putty.00454B22
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(4000)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_INSUFFICIENT_BUFFER (0000007A)
```

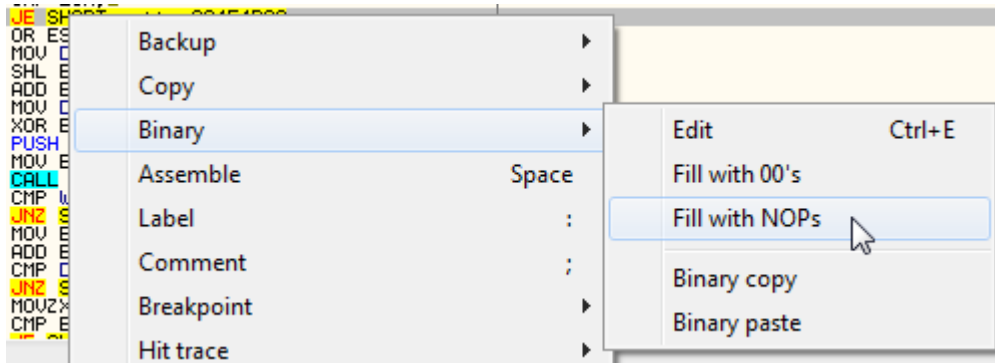
JE (jump on equality) is taken whenever the zero flag (Z) is set. To change the zero flag, double-click on the value next to it.

```
EIP 00454B22 putty.00454B22
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(4000)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_INSUFFICIENT_BUFFER (0000007A)
```

Now the jump won't be made (grey arrow).

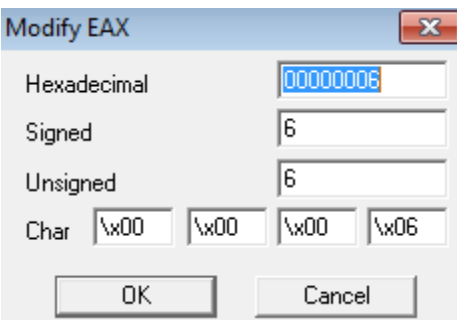
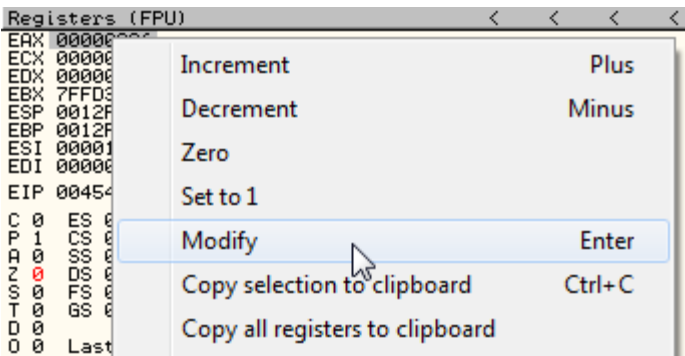
00454B19	.	8935 44E14700	MOV DWORD PTR DS:[47E144],ESI
00454B1F	.	83F9 02	CMP ECX,2
00454B22	.	74 0C	JE SHORT putty.00454B30
00454B24	.	81CE 00800000	OR ESI,8000
00454B2A	.	8935 44E14700	MOV DWORD PTR DS:[47E144],ESI
00454B30	>	C1E0 08	SHL EAX,8
00454B33	.	03C2	ADD EAX,EDX
00454B35	.	A3 48E14700	MOV DWORD PTR DS:[47E148],EAX

You can also change a jump to never be made by overwriting the jump instruction with NOP instructions. To do this, just right-click on the jump instruction and choose *Binary->Fill with NOPs*.

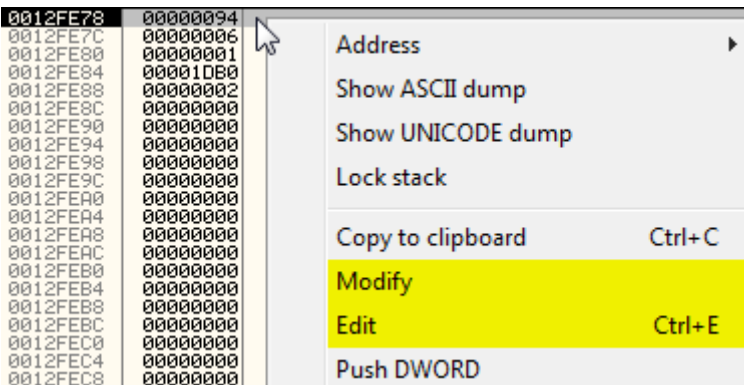


00454B19	.	8935 44E14700	MOV DWORD PTR DS:[47E144],ESI
00454B1F	.	83F9 02	CMP ECX,2
00454B22	.	90	NOP
00454B23	.	90	NOP
00454B24	.	81CE 00800000	OR ESI,8000
00454B2A	.	8935 44E14700	MOV DWORD PTR DS:[47E144],ESI

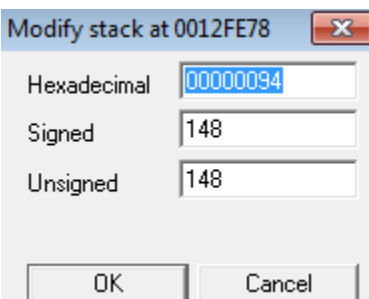
In a similar way as modifying the FLAGS register you can also modify other registers. To do this, right-click on the register value and choose *Modify*.

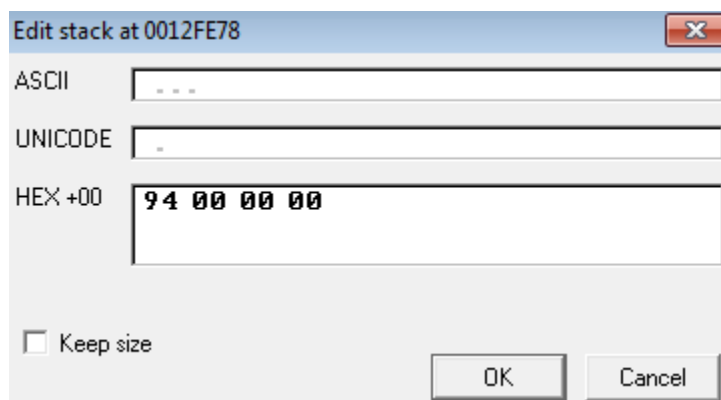


Values on the stack can be modified as well.

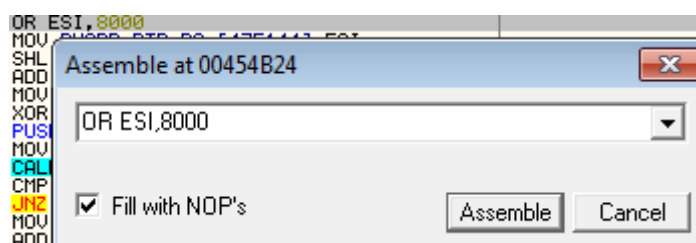


This time however there are two options: *Modify* and *Edit*. The difference between them is that *Modify* treats stack values as numbers while *Edit* treats stack values as group of bytes.





Besides modifying registers and data in the memory, it is also possible to change instructions that are executed. To achieve this just select the instruction you want to modify and press <space>.



This way you can edit the instruction operands or replace the instruction with a completely different one. However note that if a new instruction code would be longer than the code of the instruction that you are editing, then other instructions in the code would also be affected. If the new instruction code would be shorter, then the remaining bytes would be filled with NOP instructions.

2.5 Plugins

One very important aspect of OllyDbg are its plugins. OllyDbg has a very big plugin base contributed by many authors. Plugins are mainly used to introduce new features, to make debugging easier or to implement anti-anti-debugging techniques preventing OllyDbg from being detected.

Most of the popular plugins can be downloaded from the following websites:

- Collaborative RCE Tool Library¹⁴
- Tuts 4 You¹⁵
- OpenRCE.org¹⁶

¹⁴OllyDbg Extensions http://www.woodmann.com/collaborative/tools/index.php/Category:OllyDbg_Extensions (last accessed 11.09.2015)

¹⁵ OllyDbg 1.xx Plugins <https://tuts4you.com/download.php?list.9> (last accessed 11.09.2015)

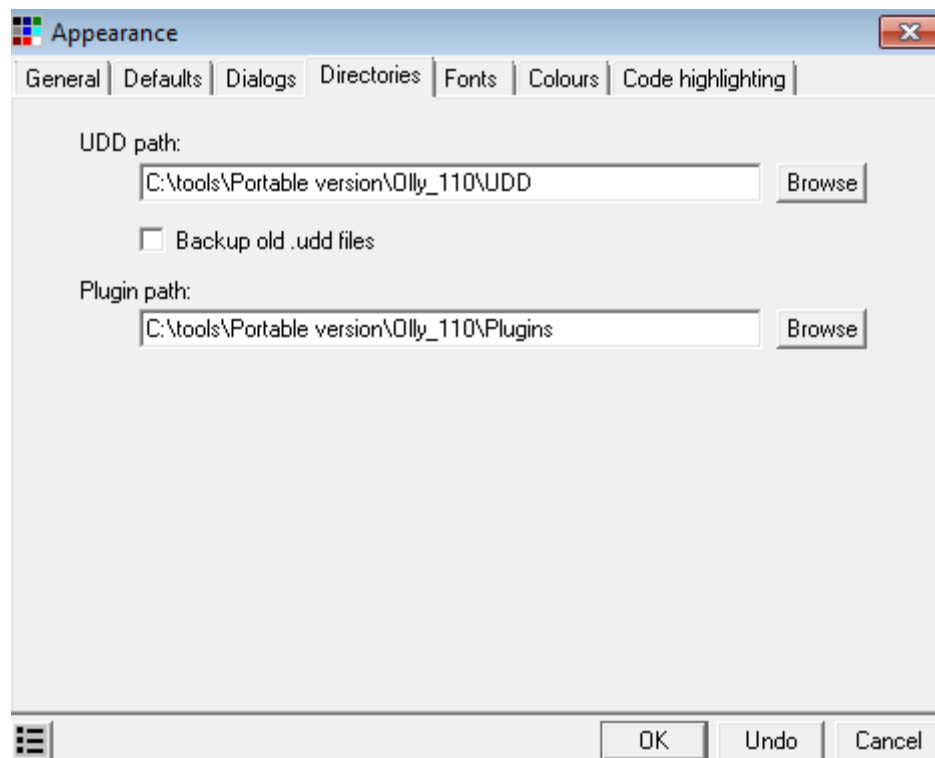
¹⁶ OpenRCE Hosted Downloads: OllyDbg Plugins http://www.openrce.org/downloads/browse/OllyDbg_Plugins (last accessed 11.09.2015)

OpenRCE Hosted Downloads: OllyDbg Plugins

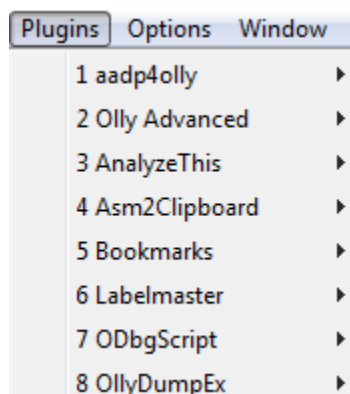
OllyDbg Plugins Downloads

Name	Author	Description Excerpt
Analyze This	Joe Stewart	Sometimes (especially when dealing with packers) you may need to run ...
Anti Anti Hardware Breakpoint	Matwood	This plug-in for OllyDbg was written to hook into NTDLL and restore the DRx ...
APIFinder	Tomislav Pericin	Simple plug-in that assists in the location of and breakpoint setting on ...
Asm2Clipboard	fatmike	copy asm code to clipboard
AttachAnyway	Joe Stewart	AttachAnyway is a PoC OllyDbg plugin designed to show how to remove a ...
Breakpoint Manager	Pedram Amini	OllyDBG has excellent breakpoint manipulation capabilities and can store ...
Catchal	mikado	Sometimes you don't know how to start a program correctly from OllyDgb. ...
CLBPlus!	Robert Ayrapetyan	CLBPlus! plugin extends standart capabilities of conditional log ...
Cleanup Ex	Gigapede	deletes all .udd, .bak files
CommandBar	Gigapede	SoftICE commands in a small bar on the bottom. Update: macro function, ...
DebugPlugin	TBD	loads OllyDbg and breakpoints on load plugin routine.
DeJunk	flyfancy	find/remove junkcode from packers. also customizable.
DllBreakEx	Epsilon3	For debugged proggies which loads a lot of dll, with this plugin you can ...

After downloading a plugin, unpack it and copy the plugin's .dll library to the OllyDbg's plugin directory (e.g. `c:\tools\Portable version\Olly_110\Plugins`). The exact location of the plugins directory can be checked in the *Options->Appearance->Directories* menu.



After plugin installation, restart OllyDbg. If the plugin is working, it should be available through the plugins menu.



Note that plugins created for OllyDbg v1.10 are not compatible with OllyDbg 2.xx and vice versa.

There are many useful plugins for OllyDbg and it is mostly up to your preference which to use. Among the plugins used in this training are.

- **aadp4olly** - tries to hide OllyDbg from most of the popular anti-debugger techniques.
- **Olly Advanced** – fixes some bugs in OllyDbg v1.10 and introduces new functions enhancing OllyDbg capabilities. It also implements various anti-anti-debugging techniques.
- **ODbgScript** – introduces scripting assembly-like language allowing to automate certain tasks.
- **OllyDumpEx** – memory and PE dumping plugin. It allows to dump PE image from the memory to the file. Frequently used for dumping unpacked binaries.
- **Bookmarks** – allows to insert bookmarks in the code to help quickly navigate to them later.

2.6 Shortcuts

Shortcuts are essential parts of OllyDbg. Thanks to the shortcuts you can perform many operations much faster, saving valuable time. This section lists the most commonly used shortcuts in OllyDbg.

Debugging:

OPERATION	SHORTCUT
Run	F9
Pause	F12
Restart debugged app	Ctrl+F2
Close debugged app	Alt+F2
Step into	F7
Step over	F8
Execute till return	Ctrl+F9
Execute till user code	Alt+F9
Pass exception to the program	Shift+F7/F8/F9

Animate into	Ctrl+F7
Animate over	Ctrl+F8
Trace into	Ctrl+F11
Trace over	Ctrl+F12

Windows and views:

OPERATION	SHORTCUT
CPU window	Alt+C
Memory map	Alt+M
Executable modules	Alt+E
Call stack	Alt+K
Breakpoints	Alt+B

Other operations:

OPERATION	SHORTCUT
Follow jump/call	Enter
Assembly instruction	Space
Edit memory	Ctrl+E
Add comment	; (semicolon)
Add label	: (colon)
Insert bookmark X	Alt+Shift+0..9
Go to bookmark X	Alt+0..9

3. Unpacking artefacts

3.1 Packers and protectors

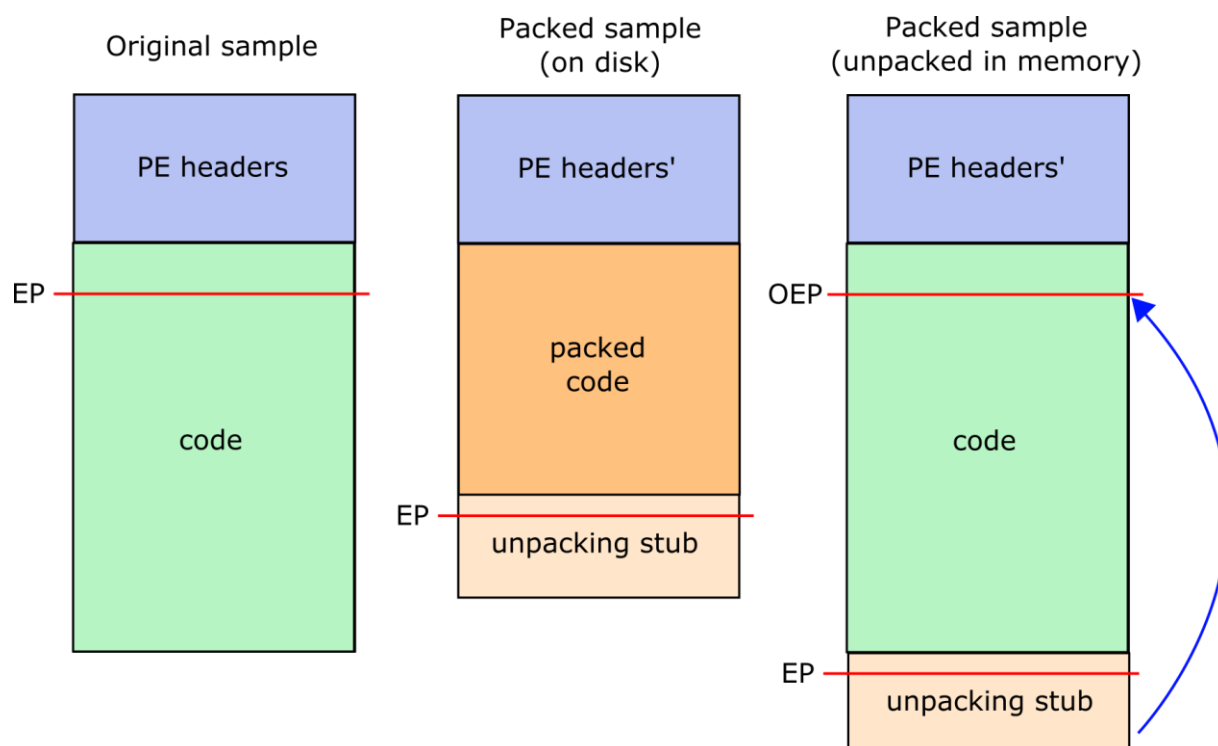
3.1.1 Introduction to packers and protectors

Packers are tools used to obfuscate other executables (usually malware) by rewriting their code. The resulting code is usually completely different from the original code and is impossible to analyse prior to unpacking it. After the execution of a packed binary, its code is unpacked at runtime to its original form, and the original code starts executing like it had never been packed.

Packers are serious problems in the IT security industry because one malware code can be packed (obfuscated) many times, each time resulting in seemingly completely different code. This makes signature based AV engines much less effective. Moreover, since each packer works differently there is no single unpacking algorithm.

Sometimes you might encounter names such as protector or crypter. They are often used interchangeably with the name packer to describe the same type of the tool. Using strict definitions, a *packer* is a tool which compresses a binary code making it smaller, a *protector* adds to the binary an additional protection layer (anti-emulation, anti-debugging, anti-sandbox) and a *crypter* encrypts the original binary code. Usually, one tool implements all those functions in one. For a convenience, only the term *packer* will be used in this document.

The scheme below presents a simplified version of how packers work.



The code of a packed binary is usually completely different from the code of the original binary. Packed code is often a block of highly compressed and encrypted data (with a high entropy). For obvious reasons, execution of such data is impossible. This is why a packer also needs to add to the binary an *unpacking stub*. The unpacking stub is a special code which sole role is to unpack and rebuild the original binary in the memory. After the execution of the packed

binary, the unpacking stub starts unpacking the code. When unpacking is finished and the import address table (IAT) is rebuilt, execution is transferred to the *Original Entry Point* (OEP).

When doing a malware analysis other than a behavioural or automatic analysis, it will be necessary to first unpack the malware sample. Otherwise you won't be able to analyse the original code. To detect if the sample was packed and what packer was used, you might use tools such as PEiD¹⁷ or ExeInfo PE¹⁸. If the packer used to pack the sample is well known, these tools should return its name. You can then search for an automatic unpacking tool for this particular packer¹⁹. Remember to always use unpacking tools in isolated environments.

3.1.2 Unpacking steps

If there is no automatic unpacking tool for the sample, it needs to be unpacked manually using a debugger. In general there isn't a single strategy or an algorithm for how to unpack binary files. Each packer and protector is slightly different and needs to be handled differently.

There are three stages of unpacking a binary file:

1. Finding OEP.
2. Dumping process image.
3. Rebuilding IAT and fixing EP.

When the unpacking stub starts executing, it will jump to the original entry point at some point. Finding the OEP is the first and often the most difficult task when trying to unpack the malware. There are few techniques that might help you finding the OEP, which will be presented in the next section.

After the OEP has been found, you need to dump the memory of the unpacked image of the original executable. Sometimes, the packer might utilize anti-dumping techniques²⁰: these are however not part of this training.

The last step is to rebuild the *Import Address Table* (IAT) and to fix the *Entry Point* (EP) address of the executable. This is necessary as packers usually modify PE headers when obfuscating the original code. A modified IAT table is often limited to just few most important entries and the EP points to the unpacking stub or some other code.

How to perform all these steps will be presented later in this training.

3.1.3 Finding the OEP

There are few techniques which will help you to recognize or find the OEP:

- Unpacking stubs often finish with indirect jumps or calls to the address stored in some register (for example *jmp eax* or *call eax*). If you see such instruction in the code, you should consider that this might be a jump to the OEP – especially if such an instruction is one of the last instructions in the unpacking routine.
- Unpacking stubs are often located in the PE file section rather than the code section. Sometimes unpacking stubs are also copied to newly allocated memory blocks outside of the original PE image.

¹⁷Binary Analysis / Editing <https://tuts4you.com/download.php?view.398> (last accessed 11.09.2015)

¹⁸ExeInfo PE http://www.woodmann.com/collaborative/tools/index.php/ExeInfo_PE (last accessed 11.09.2015)

¹⁹http://www.woodmann.com/collaborative/tools/index.php/Category:Automated_Unpackers (last accessed 11.09.2015)

²⁰Anti-Memory Dumping Techniques <http://resources.infosecinstitute.com/anti-memory-dumping-techniques/> (last accessed 11.09.2015)

Therefore if during unpacking you see a jump to the code section you should consider that this might be a jump to the OEP address.

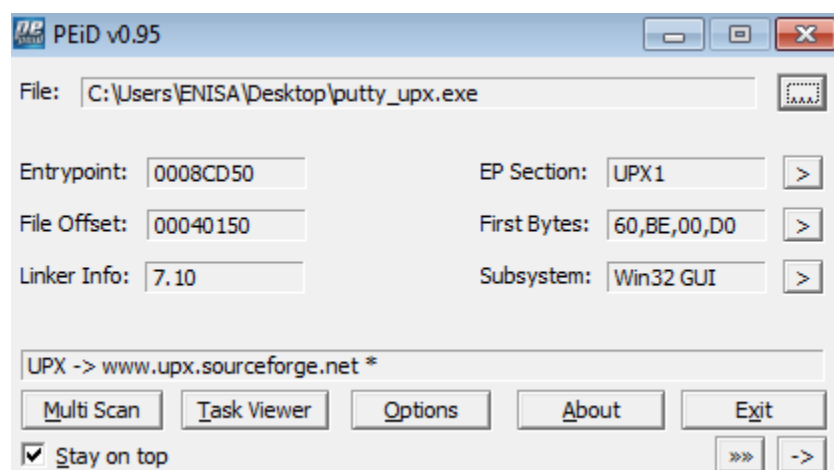
- When unpacking, the code section is usually overwritten with a new code. You can use memory breakpoints (on write) to detect unpacking loops. Then after unpacking finishes, try using memory breakpoints (on access) to detect moments when the execution will transfer to the unpacked code.
- Before unpacking finishes the program stack and registers are often restored to the initial state. If you see such behaviour in the unpacking routine, this might mean that soon there will be a jump to OEP.
- Compilers usually produce a similar entry point code for each created executable. Knowing how the entry point code produced by various compilers looks like can help you to recognize the OEP in packed samples.
- Many Windows applications at the beginning of the main routine call functions such as *GetCommandLine*, *GetModuleHandle*, *GetStartupInfo*, *GetVersion*, *GetVersionEx*. If you see such calls in the code this might mean that you have already reached the OEP. Additionally, one of the unpacking strategies might be to put breakpoints on those functions hoping they will be called at the beginning of the main routine.

3.2 Unpacking UPX packed sample

UPX²¹ is a fairly simple and commonly used packer. Samples that have been packed with UPX can be easily unpacked using publicly available tools, but during this exercise it will be shown how to manually unpack a UPX packed sample (*putty.exe*), to show the general concept of unpacking artefacts manually.

As an initial step, the packed code needs to be detected, similarly to the steps in the ENISA training material “Artefact analysis fundamentals”²². PEiD²³ is used to identify if the sample was packed and what packer was used. In some cases PEiD won’t properly identify if the sample was packed. In such situations, other checks and some manual assessment may be needed (for example checking embedded strings, inspecting IAT table or inspecting list of sections) – as described in “Artefact analysis fundamentals”.

In this case, PEiD reveals that *putty_upx.exe* sample was packed with UPX as seen in the following screenshot.

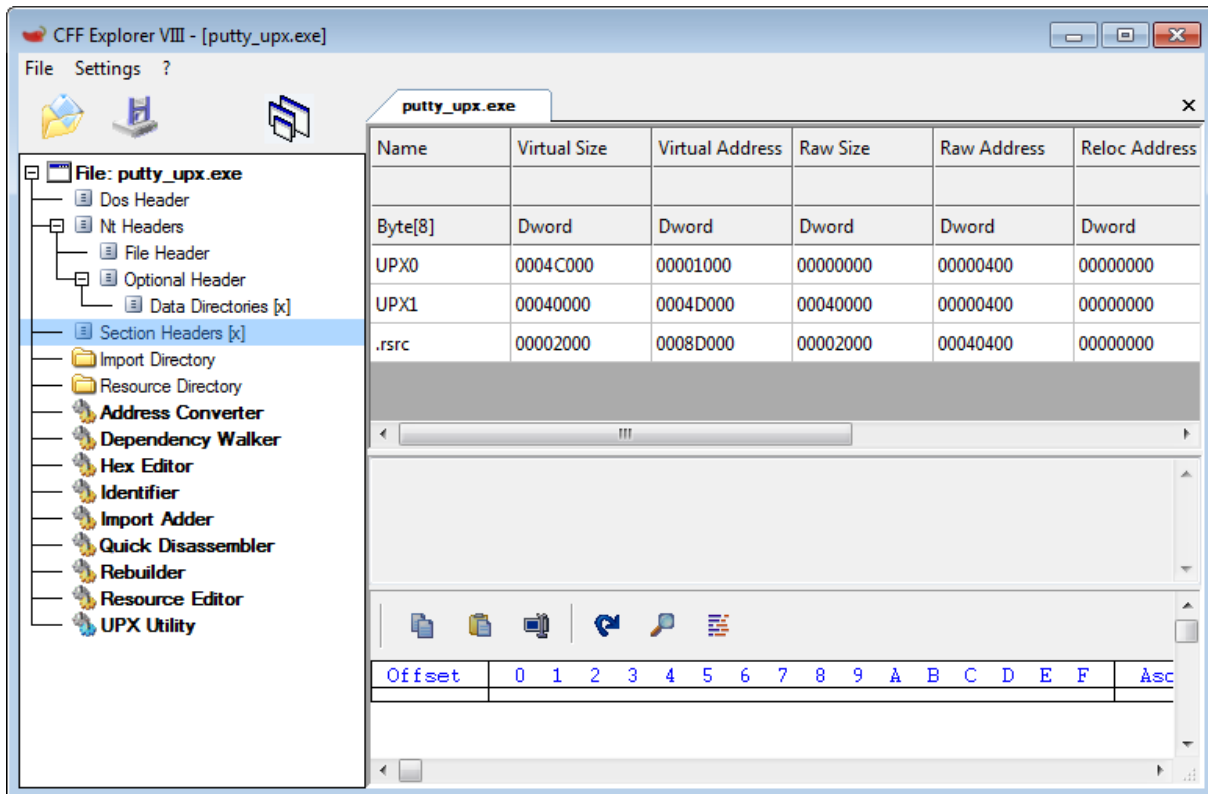


²¹UPX <http://upx.sourceforge.net/> (last accessed 11.09.2015)

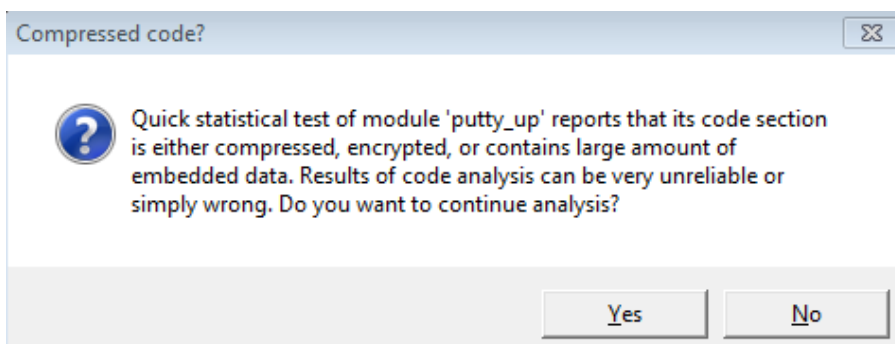
²² Artefact Analysis Fundamentals <https://www.enisa.europa.eu/activities/cert/training/training-resources/documents/artifact-analysis-fundamentals-handbook> (last accessed 11.09.2015)

²³ PEiD <http://www.aldeid.com/wiki/PEiD> (last accessed 11.09.2015)

To confirm the output of PEiD and to identify packing more specifically, CFF Explorer²⁴ is used. CFF Explorer is designed to make the PE editing as easy as possible. Beside PE headers viewing and editing CFF Explorer contains an integrated hex editor, simple disassembler and many other useful features. One distinct feature of UPX packed samples are two sections UPX0 and UPX1 within PE file, as seen on the screenshot below under the section headers part. Please note the Virtual Addresses of both sections²⁵, they will be referred later in this exercise.



Now, when you know that the sample was packed with UPX let's move forward towards the manual unpacking. To do this, the sample is opened in OllyDbg. OllyDbg should report that the sample looks like compressed or packed code and ask whether to continue with automatic analysis of this code. Answer "No".



²⁴ CFF Explorer suite <http://www.ntcore.com/exsuite.php> (last accessed 11.09.2015)

²⁵ Actually CFF Explorer presents the Relative Virtual Addresses (RVA) in the Virtual Address column in the sections list. To get actual Virtual Addresses of the sections, add to them the module base address (here 0x4000000) or use the 'Address Converter' feature from the CFF Explorer.

Execution of the executable should be paused at the entry point of putty_upx.exe (0x48CD50) which is located in the UPX1 section.

```

CPU - main thread, module putty_up
0048CD50 60 PUSHAD
0048CD51 BE 00D04400 MOV ESI,putty_up.0044D000
0048CD56 8DBE 0040FBFF LEA EDI,DWORD PTR DS:[ESI+FFFB4000]
0048CD5C 57 PUSH EDI
0048CD5D EB 0B JMP SHORT putty_up.0048CD6A
0048CD5F 90 NOP
0048CD60 8A06 MOV AL,BYTE PTR DS:[ESI]
0048CD62 46 INC ESI
0048CD63 8807 MOV BYTE PTR DS:[EDI],AL
0048CD65 47 INC EDI
0048CD66 01DB ADD EBX,EBX
0048CD68 75 07 JNZ SHORT putty_up.0048CD71
0048CD6A 8B1E MOV EBX,DWORD PTR DS:[ESI]
0048CD6C 83EE FC SUB ESI,-4
0048CD6F 11DB ADC EBX,EBX
0048CD71 72 ED JB SHORT putty_up.0048CD60
0048CD73 B8 01000000 MOV EAX,1
0048CD78 01DB ADD EBX,EBX
  
```

In order to reach the beginning of an unpacking routine, step over the function (Shortcut key F8) five times until the MOV EBX,DWORD PTR DS:[ESI] instruction (at the address 0x48CD6A).

```

CPU - main thread, module putty_up
0048CD50 60 PUSHAD
0048CD51 BE 00D04400 MOV ESI,putty_up.0044D000
0048CD56 8DBE 0040FBFF LEA EDI,DWORD PTR DS:[ESI+FFFB4000]
0048CD5C 57 PUSH EDI
0048CD5D EB 0B JMP SHORT putty_up.0048CD6A
0048CD5F 90 NOP
0048CD60 8A06 MOV AL,BYTE PTR DS:[ESI]
0048CD62 46 INC ESI
0048CD63 8807 MOV BYTE PTR DS:[EDI],AL
0048CD65 47 INC EDI
0048CD66 01DB ADD EBX,EBX
0048CD68 75 07 JNZ SHORT putty_up.0048CD71
0048CD6A 8B1E MOV EBX,DWORD PTR DS:[ESI]
0048CD6C 83EE FC SUB ESI,-4
0048CD6F 11DB ADC EBX,EBX
0048CD71 72 ED JB SHORT putty_up.0048CD60
0048CD73 B8 01000000 MOV EAX,1
0048CD78 01DB ADD EBX,EBX
0048CD7A 75 07 JNZ SHORT putty_up.0048CD83
0048CD7C 8B1E MOV EBX,DWORD PTR DS:[ESI]
0048CD7E 83EE FC SUB ESI,-4
  
```

Registers (FPU)

```

EAX 77381162 kernel32.BaseThreadInitThunk
ECX 00000000
EDX 0048CD50 putty_up.<ModuleEntryPoint>
EBX 7FFD0000
ESP 0012FF68
EBP 0012FF94
ESI 0044D000 putty_up.0044D000
EDI 00401000 putty_up.00401000
EIP 0048CD6A putty_up.0048CD6A
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
D 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
O 0
D 0 LastErr ERROR_INSUFFICIENT_BUFFER (0000007A)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
  
```

At this instruction, code is being read from the memory pointed by the ESI register. Take a look at ESI (source) and EDI (destination) registries. ESI points to the beginning of the UPX1 section while EDI points to the beginning of UPX0 (refer to previously checked Virtual Addresses in CFF Explorer). This suggests that some data will be read from UPX1, then processed and finally written to UPX0.

To see packed code follow in the hex dump (Ctrl+G) ESI register.

Address	Hex dump	ASCII
0044D000	37 EF FF FF 56 6A 0C 6A 01 E8 34 00 AB EA 8B F0	7n Uj.j034.%ni=
0044D010	0B 44 24 10 89 46 04 0C 14 08 A1 68 FE FD F7 FF	iD\$>eF. q ihm?#
0044D020	14 48 00 85 C0 59 59 74 12 83 30 60 16 00 75 09	9H.à.YYt*?=-.u.
0044D030	FF 35 6C 10 FF 00 59 A1 64 BF 33 F7 FF 39 74 04	5l? *Yich3? 9t?
0044D040	89 30 EB 06 89 35 30 0A 27 83 26 00 5E C3 56 CF	@0\$#5=. 'a%. ^lu?
0044D050	DD FE BF 88 23 85 F6 74 25 FF 76 08 FF 56 A4 06	l q i#3+?% v U#?
0044D060	56 A3 23 D0 AC F7 E7 DE B6 AA 9B B0 75 07 83 25	Uq#%*%r Hl-c%u. a%
0044D070	67 62 33 C0 39 05 28 7F BF FF 76 0F 96 C0 7A 8D	gb3?9#++?_ v*0?zi
0044D080	34 40 C1 E6 02 8B 8E 74 20 8D 86 78 0A 3B 08 7C	40+?0i?at i&x.;0!
0044D090	1C BF 8B C1 FF 57 83 C1 20 6A 08 51 8D BE 70 05	Lq?+ W3+ j00i?p?
0044D0A0	37 89 08 99 44 83 C4 0C 89 07 83 EF 6C F7 5F FF	7e00D3-.e. an l?
0044D0B0	74 24 10 80 48 69 8B 96 47 74 9C F7 67 F7 DE 04	t\$#H i0Gt&g% i?
0044D0C0	CA 2F 23 E4 3F 89 44 CA 04 FF 86 12 05 F6 0D 7E	*/#3?eD#? a?+?..
0044D0D0	F2 8B 0D 88 55 C9 74 69 83 7C 88 C6 62 5A FB B7	zi.eUrti&i&fbz7m
0044D0E0	07 FC 04 83 E8 01 75 58 8B 40 18 8B 00 14 04 28	*. #30uX i0+i. q0!
0044D0F0	4D BE 9B 0D 0A 66 8C 51 74 03 C3 51 50 A4 AF BF	M?c..fi0t? 0P&?>
0044D100	3D EC CD 19 16 71 14 05 5F EC 50 17 04 F2 38 0B	=>+q0! .wP?#>80
0044D110	AC 7B 80 15 D1 25 15 00 64 48 18 C7 82 FB DF BA	%C%3?%S. dHt ?e?
0044D120	05 81 01 1A 00 40 C3 83 C8 FF C3 85 05 90 1E 04	#i0+.0?#? h&e&#?
0044D130	B3 76 F7 07 48 04 68 08 25 46 79 65 1F 59 56 59	lv* .H0h0?FueVUYU
0044D140	3E C6 BE DF FE FF 55 8B EC 83 EC 10 53 57 8B 7D	?#?# Uio&wSwi?)
0044D150	08 68 C0 27 3A 33 DB 57 89 5D FC D8 62 9C 0D DB	h? :?3W3j?#?b. #
0044D160	7F 2D 25 75 3B 39 5D 0C 75 04 6A FE EB 2C 10 10	-?u;9l.u.0+j#?> >
0044D170	7C CA 75 44 B6 BB 6D 90 75 0C 3C 01 0B C2 0E D6	?uDh?i&u.<0?#?#
0044D180	EC 46 DF FE 3E 1C F7 44 A3 E4 20 6A 02 58 E9 1E	wF#>L%Du& j0X0?
0044D190	05 E7 5C 58 05 05 05 05 05 05 05 05 05 05 05	*?#?#?#?#?#?#?#?#

Then follow in the hex dump EDI register to see a clean memory where unpacked code will be stored.

Address	Hex dump	ASCII
00401000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004010A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004010B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004010C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004010D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004010E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004010F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Then press and hold for a few seconds the Step Over key (F8). You should observe in the hex dump the UPX0 section (pointed by EDI) being overwritten with the unpacked code.

Address	Hex dump	ASCII
00401000	56 6A 0C 6A 01 E8 34 00 AB EA 8B F0 8B 44 24 10	Uj.j0\$4.%Ri=iD\$▶
00401010	89 46 04 8B 44 24 14 89 46 08 A1 68 14 48 00 85	eF◊iD\$7eF ih7H.ã
00401020	C0 59 59 74 12 83 30 60 14 48 00 00 75 09 FF 35	LYvt◊7=7H..u.!
00401030	6C 14 48 00 FF D0 59 A1 64 14 48 00 85 C0 74 04	l7H. #Yid7H.ãt◊
00401040	89 30 EB 06 89 35 60 14 48 00 89 35 64 14 48 00	e0\$+960577H.e5d7H
00401050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Now scroll down over numerous jump instructions until you see three CALL instructions (at the addresses 0x48CE8A, 0x48CEA8, 0x48CEB9). Set breakpoints at those instructions to inspect what functions are called there. Resume execution (F9).

0048CE86	50	PUSH EAX	putty_up.0048EDE4
0048CE87	83C7 08	ADD EDI,8	
0048CE8A	FF96 78DD0800	CALL DWORD PTR DS:[ESI+8DD78]	kernel32.LoadLibraryA
0048CE90	95	XCHG EAX,EBP	
0048CE91	8A07	MOV AL,BYTE PTR DS:[EDI]	
0048CE93	47	INC EDI	putty_up.0048A008
0048CE94	08C0	OR AL,AL	
0048CE96	74 DC	JE SHORT putty_up.0048CE74	
0048CE98	89F9	MOV ECX,EDI	putty_up.0048A008
0048CE9A	79 07	JNS SHORT putty_up.0048CEA3	
0048CE9C	0FB707	MOVZX EAX,WORD PTR DS:[EDI]	
0048CE9F	47	INC EDI	putty_up.0048A008
0048CEA0	50	PUSH EAX	putty_up.0048EDE4
0048CEA1	47	INC EDI	putty_up.0048A008
0048CEA2	B9 5748F2AE	MOV ECX,REF24857	
0048CEA7	55	PUSH EBP	
0048CEA8	FF96 7CDD0800	CALL DWORD PTR DS:[ESI+8DD7C]	kernel32.GetProcAddress
0048CEAE	09C0	OR EAX,EAX	putty_up.0048EDE4
0048CEB0	74 07	JE SHORT putty_up.0048CEB9	
0048CEB2	8903	MOV DWORD PTR DS:[EBX],EAX	putty_up.0048EDE4
0048CEB4	83C3 04	ADD EBX,4	
0048CEB7	EB 08	JMP SHORT putty_up.0048CE91	
0048CEB9	FF96 8CDD0800	CALL DWORD PTR DS:[ESI+8DD8C]	kernel32.ExitProcess
0048CEBF	8BAE 80DD0800	MOV EBP,DWORD PTR DS:[ESI+8DD80]	kernel32.VirtualProtect
0048CEC5	8DBE 00F0FFFF	LEA EDI,DWORD PTR DS:[ESI-1000]	

At this point the unpacking routine is rebuilding the original Import Address Table (IAT) of the executable. It is done by loading necessary libraries, resolving addresses of used functions and storing them in the memory.

Put breakpoint at 0x48CEBF (outside IAT reconstruction loop) and press F9 a few times (5-10). In the stack window you can observe what functions are being loaded.

Address	Hex dump	ASCII
00461FE0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00461FF0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00462000	04 BE 98	Backup
00462010	33 B6 98	Search for
00462020	07 FC 97	Go to
00462030	96 1B 98	Hex
00462040	71 1B 98	Text
00462050	19 46 D6	Short
00462060	CA 7C B1	Long
00462070	BB 08 B2	Float
00462080	58 67 B1	Disassemble
00462090	DB 5D B1	Special
004620A0	77 F1 B1	Appearance
004620B0	06 F2 B1	Signed decimal
004620C0	4E C4 B3	Unsigned decimal
004620D0	25 C2 B3	Hex
004620E0	EE A6 B1	Address
004620F0	68 C9 B1	Address with ASCII dump
00462100	55 FE B1	Address with UNICODE dump
00462110	41 FC B1	
00462120	3F 6D AF	
00462130	48 BF A5	
00462140	02 45 A5	
00462150	B7 30 A6	
00462160	35 2B A7	
00462170	5E 0F 0C	

Address	Value	Comment
00461FF0	00000000	
00461FF4	00000000	
00461FF8	00000000	
00461FFC	00000000	
00462000	7598BED4	ADVAPI32.RegCloseKey
00462004	7598BC25	ADVAPI32.RegQueryValueExA
00462008	7597D2E0	ADVAPI32.RegOpenKeyA
0046200C	7597E504	ADVAPI32.GetUserNameA
00462010	7598B633	ADVAPI32.EqualSid
00462014	7598BA61	ADVAPI32.CopySid
00462018	7598B80C	ADVAPI32.GetLengthSid
0046201C	7598B82F	ADVAPI32.SetSecurityDescriptorDacl
00462020	7597FC07	ADVAPI32.SetSecurityDescriptorOwner
00462024	7598BB1D	ADVAPI32.InitializeSecurityDescriptor
00462028	7598B7DC	ADVAPI32.AllocateAndInitializeSid
0046202C	7597D3C1	ADVAPI32.RegCreateKeyA
00462030	75981B96	ADVAPI32.RegSetValueExA
00462034	759A0499	ADVAPI32.RegDeleteKeyA
00462038	7597D2BA	ADVAPI32.RegEnumKeyA
0046203C	7598194E	ADVAPI32.RegDeleteValueA
00462040	75981B71	ADVAPI32.RegCreateKeyExA
00462044	00000000	
00462048	73E1F437	COMCTL32.LBItemFromPt
0046204C	73E1F559	COMCTL32.DrawInsert
00462050	73D64619	COMCTL32.InitCommonControls
00462054	73E1E09E	COMCTL32.MakeReadyList

Next scroll down the assembly code until the characteristic JMP instruction at 0x48CEFC. Such unconditional jump instructions at the end of the unpacking routine often leads to the OEP thus it is always worth inspecting them (but keep in mind this is not the only way of jumping to the OEP). Put a breakpoint at this instruction and resume the execution (F9).

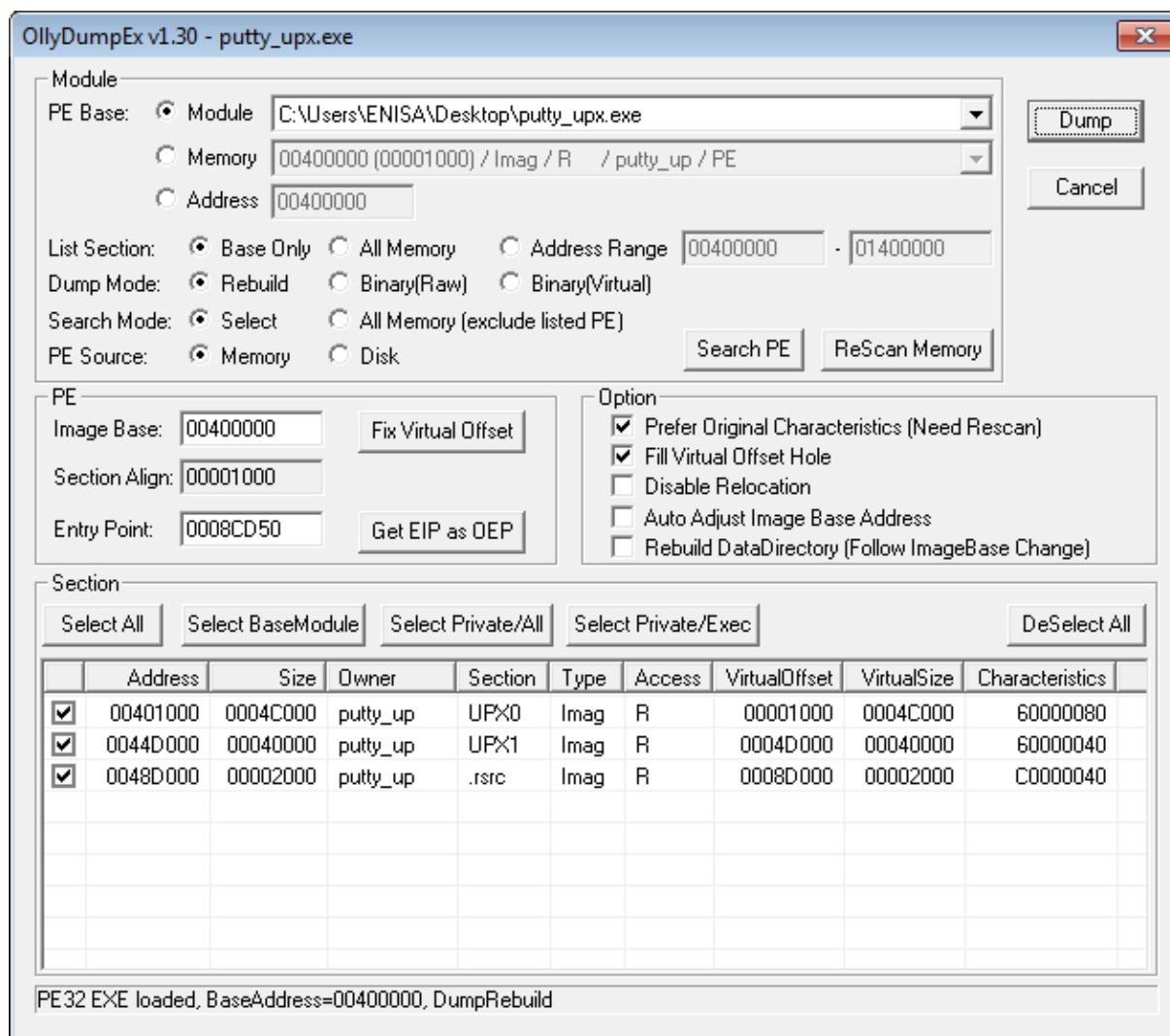
0048CEEB	FFD5	CALL EBX	
0048CEED	58	POP EAX	kernel32.75A61174
0048CEEE	61	POPAD	
0048CEEF	8D4424 80	LEA EAX, DWORD PTR SS:[ESP-80]	
0048CEF3	6A 00	PUSH 0	
0048CEFF	39C4	CMP ESP,EAX	
0048CF03	75 FA	JNZ SHORT putty_up.0048CEF3	
0048CF07	83EC 80	SUB ESP,-80	
0048CF0B	E9 DFD0FCFF	JMP putty_up.00459FE0	
0048CF0F	0000	ADD BYTE PTR DS:[EAX],AL	
0048CF13	0048 00	ADD BYTE PTR DS:[EAX],CL	
0048CF17	0000	ADD BYTE PTR DS:[EAX],AL	
0048CF1B	0000	ADD BYTE PTR DS:[EAX],AL	
0048CF1F	0000	ADD BYTE PTR DS:[EAX],AL	
0048CF23	0000	ADD BYTE PTR DS:[EAX],AL	

After reaching the breakpoint at the JMP instruction do a single step (F7/F8) to land at the OEP. In this case you can recognize the OEP by calls to functions such as *GetVersionExA* or *GetModuleHandleA*. Remember the address of the OEP (*0x459FE0*) because it will be needed later.

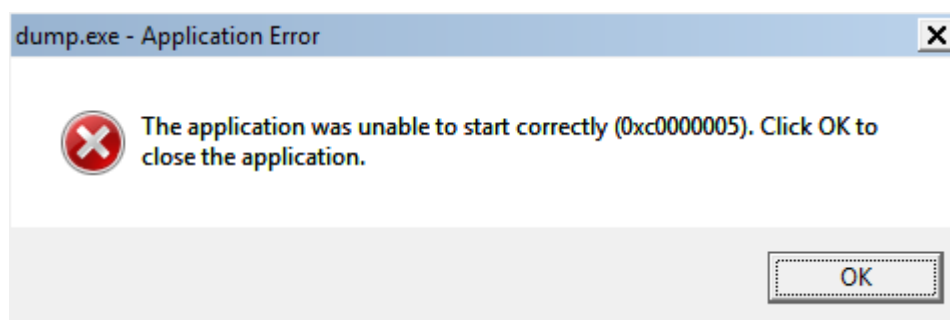
00459FE0	6A 60	PUSH 60	
00459FE2	68 E0DA4700	PUSH putty_up.0047DAE0	
00459FE7	E8 08210000	CALL putty_up.0045C0F4	
00459FEC	BF 94000000	MOV EDI, 94	
00459FF1	8BC7	MOV EAX, EDI	
00459FF3	E8 D8F9FFFF	CALL putty_up.004599D0	
00459FF8	8965 E8	MOV DWORD PTR SS:[EBP-18], ESP	
00459FFB	8BF4	MOV ESI, ESP	
00459FFD	893E	MOV DWORD PTR DS:[ESI], EDI	
00459FFF	56	PUSH ESI	
0045A000	FF15 E0224600	CALL DWORD PTR DS:[4622E0]	kernel32.GetVersionExA
0045A006	8B4E 10	MOV ECX, DWORD PTR DS:[ESI+10]	
0045A009	890D B8424800	MOV DWORD PTR DS:[4842B8], ECX	
0045A00F	8B46 04	MOV EAX, DWORD PTR DS:[ESI+4]	
0045A012	A3 C4424800	MOV DWORD PTR DS:[4842C4], EAX	
0045A017	8B56 08	MOV EDX, DWORD PTR DS:[ESI+8]	
0045A01A	8915 C8424800	MOV DWORD PTR DS:[4842C8], EDX	putty_up.<ModuleEntryPoint>
0045A020	8B76 0C	MOV ESI, DWORD PTR DS:[ESI+C]	
0045A023	81E6 FF7F0000	AND ESI, 7FFF	
0045A029	8935 BC424800	MOV DWORD PTR DS:[4842BC], ESI	
0045A02F	83F9 02	CMP ECX, 2	
0045A032	74 0C	JE SHORT putty_up.0045A040	
0045A034	81CE 00800000	OR ESI, 8000	
0045A03A	8935 BC424800	MOV DWORD PTR DS:[4842BC], ESI	
0045A040	C1E0 08	SHL EAX, 8	
0045A043	03C2	ADD EAX, EDX	putty_up.<ModuleEntryPoint>
0045A045	A3 C0424800	MOV DWORD PTR DS:[4842C0], EAX	
0045A04A	33F6	XOR ESI, ESI	
0045A04C	56	PUSH ESI	
0045A04D	8B3D D8224600	MOV EDI, DWORD PTR DS:[4622D8]	kernel32.GetModuleHandleA
0045A053	FFD7	CALL EDI	

Now the original (unpacked) putty.exe code is stored in the memory. In the previous steps you have observed how the unpacking routine was converting the packed code to its original form and how the Import Address Table was rebuilt. In the next step you will dump the unpacked process image to the executable file. To achieve this you will use the OllyDumpEx plugin which allows to dump a process image from the memory to the executable file in PE format.

When dumping unpacked putty.exe code use the default OllyDumpEx settings. Save dumped process as dump.exe. Don't close OllyDbg yet.

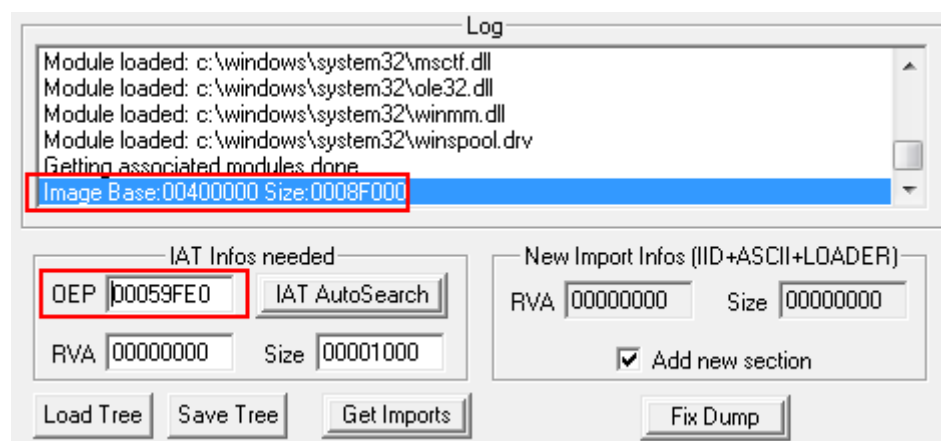
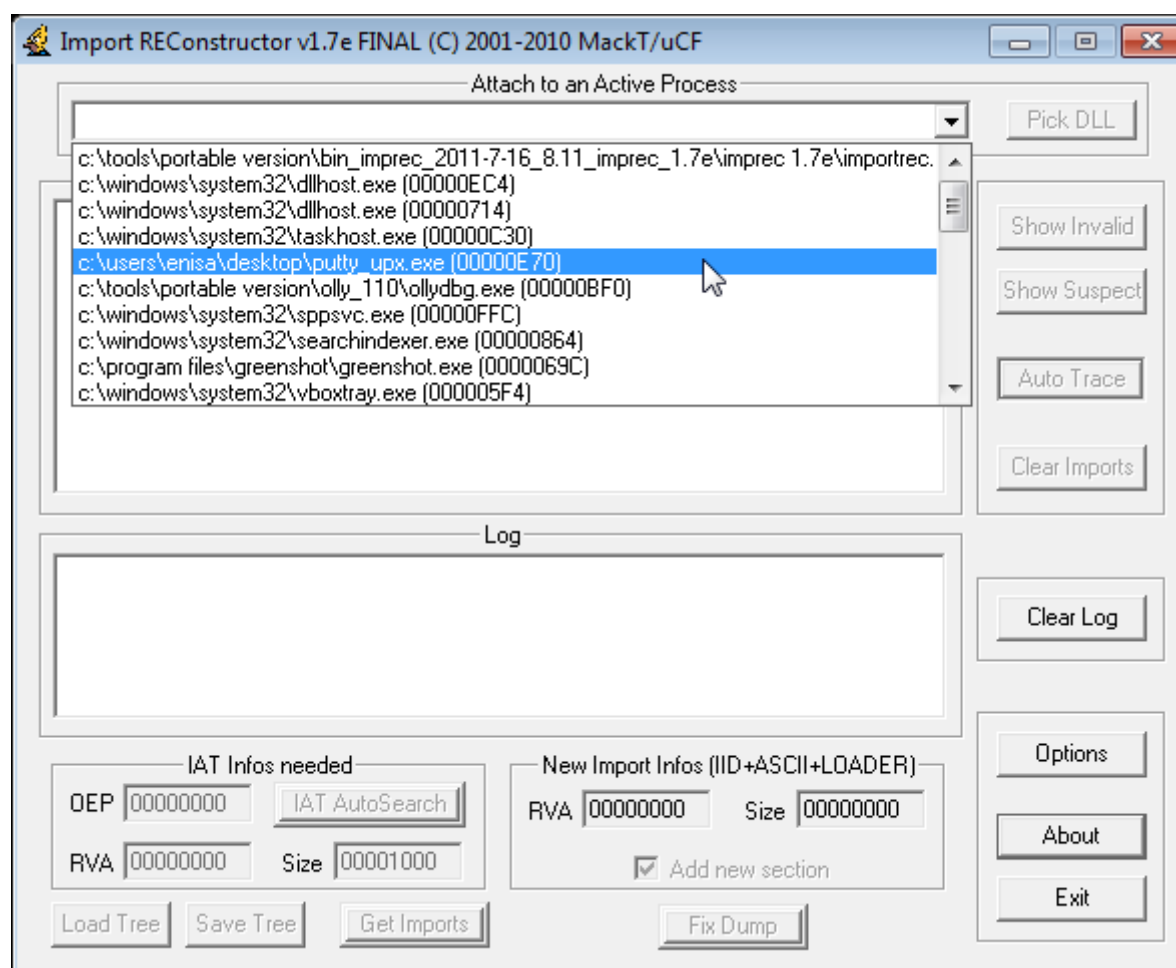


Now if you would try to execute *dump.exe* you will see an *Application Error*. That's because *dump.exe* still doesn't have the IAT reconstructed.

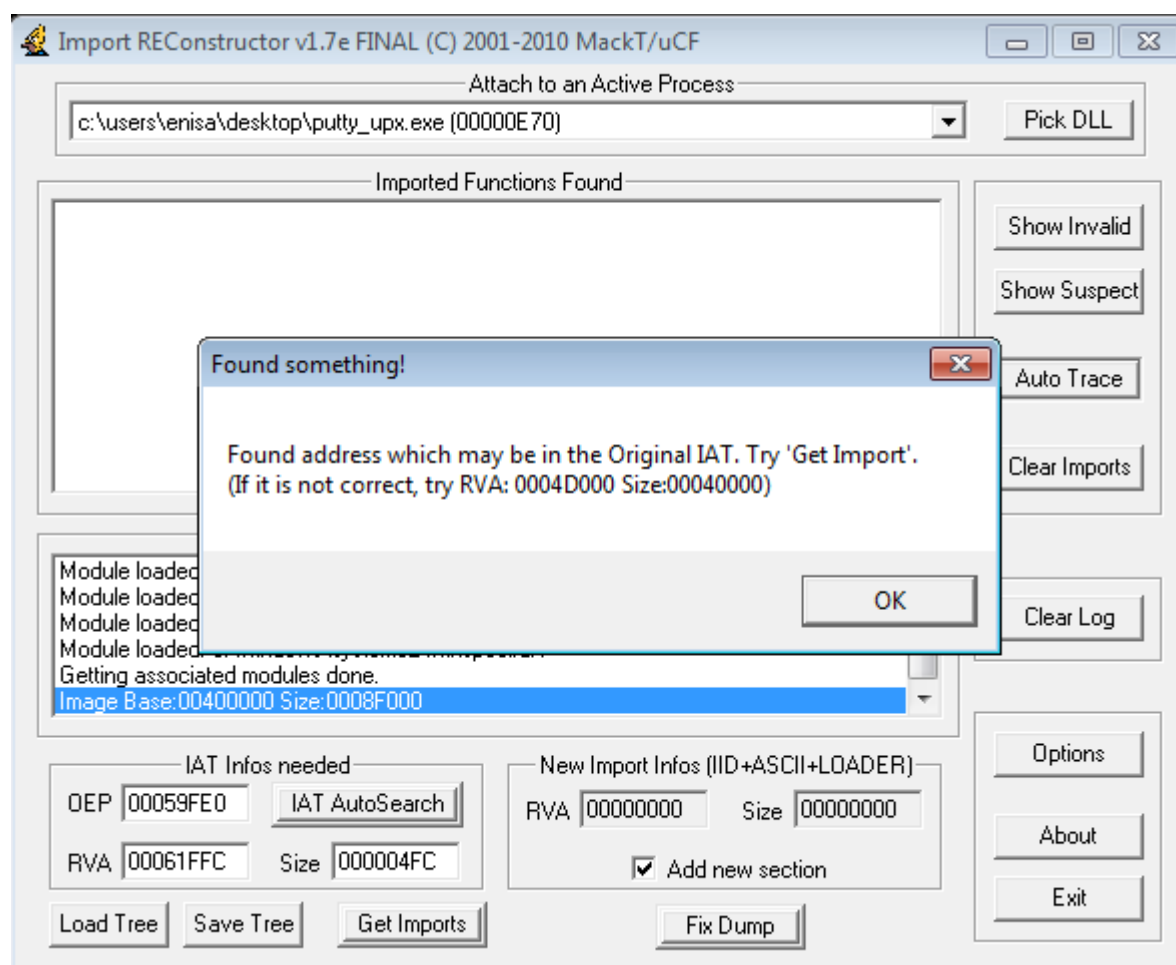


To reconstruct IAT you will use the ImpREC²⁶ tool. Run ImpREC (as Administrator) and from the scroll down menu at the top of the window choose the *putty_upx.exe* process.

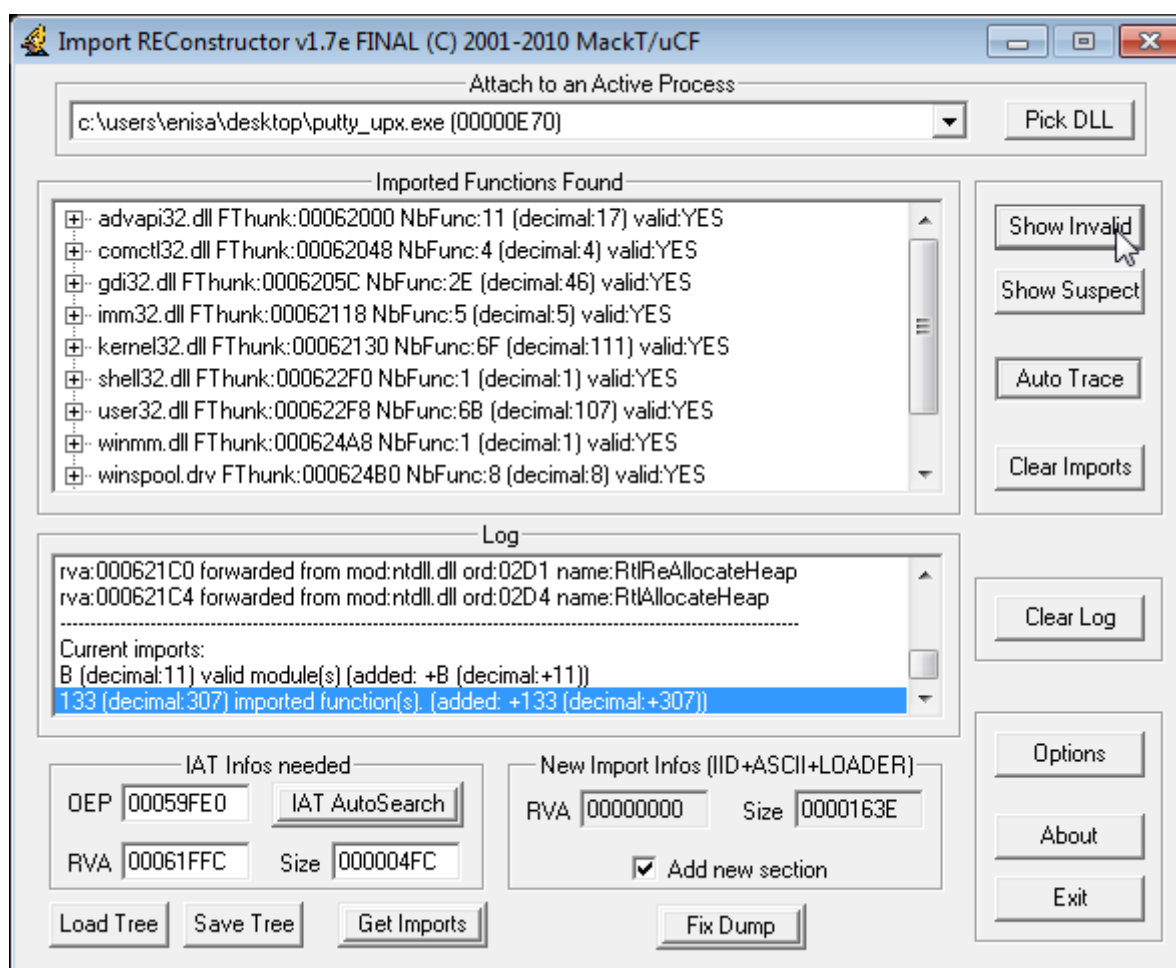
²⁶ImpREC <http://www.woodmann.com/collaborative/tools/index.php/ImpREC> (last accessed 10.10.2015)



Next in the “IAT Infos needed” panel enter the RVA address of the OEP (OEP address minus Image Base address, in this case $0x459FE0 - 0x400000 = 0x59FE0$) and click “IAT AutoSearch”. If the IAT is found you should see the appropriate message box. Otherwise you might need to try and manually enter RVA and Size of the IAT.



Next click *“Get Imports”*.



Click “*Show Invalid*” to see if there are any invalid functions. In this case there shouldn’t be any. Click “*Fix Dump*” and select *dump.exe* file. If everything goes right you should see a message that *dump.exe* was saved successfully (please note underscore in the name of the file name, the originale file wasn’t overwritten).

You can try to run *dump.exe* to check if it runs.

3.3 Unpacking UPX with the ESP trick

The manual unpacking of the UPX sample presented in the previous section was only intended for educational purposes. In most situations you are not interested in following each step of the unpacking routine and you only want to find the OEP the quickest way possible (unless there are anti-analysis or anti-debugging techniques used in the code preventing us from reaching the OEP). In this exercise you will use a simple ESP trick that will allow us to quickly track the address of an original entry point for a UPX packed sample.

Many packers try to preserve the state of the stack and registers from the start of the execution and restore it just before jumping to the OEP. This way after reaching the OEP, the application sees the stack and all registers as if it had never been packed. One way of achieving this is to push the content of all registers (using PUSHAD instruction) at the beginning of the unpacking stub and restore it just before the jump to the OEP. You can track this by putting a hardware breakpoint on the stack memory containing the saved registers and waiting till this memory will be accessed.

First, the same as in the previous exercise, open *putty_upx.exe* in OllyDbg.

```

CPU - main thread, module putty_up
0048CD50 60 PUSHAD
0048CD51 BE 00004400 MOV ESI,putty_up.00440000
0048CD56 8DBE 0040FBFF LEA EDI,DWORD PTR DS:[ESI+FFFB4000]
0048CD5C 57 PUSH EDI
0048CD5D EB 0B JMP SHORT putty_up.0048CD6A
0048CD5F 90 NOP
0048CD60 8A06 MOV AL,BYTE PTR DS:[ESI]
0048CD62 46 INC ESI
0048CD63 8807 MOV BYTE PTR DS:[EDI],AL
0048CD65 47 INC EDI
0048CD66 01DB ADD EBX,EBX
0048CD68 75 07 JNZ SHORT putty_up.0048CD71
0048CD6A 8B1E MOV EBX,DWORD PTR DS:[ESI]
0048CD6C 83EE FC SUB ESI,-4
0048CD6F 11DB ADC EBX,EBX
0048CD71 72 ED JB SHORT putty_up.0048CD60
0048CD73 B8 01000000 MOV EAX,1
0048CD78 01DB ADD EBX,EBX
  
```

Step over *PUSHAD* instruction (Shortcut key F8). Notice how the stack view and ESP register changes, as seen on the following screenshot.

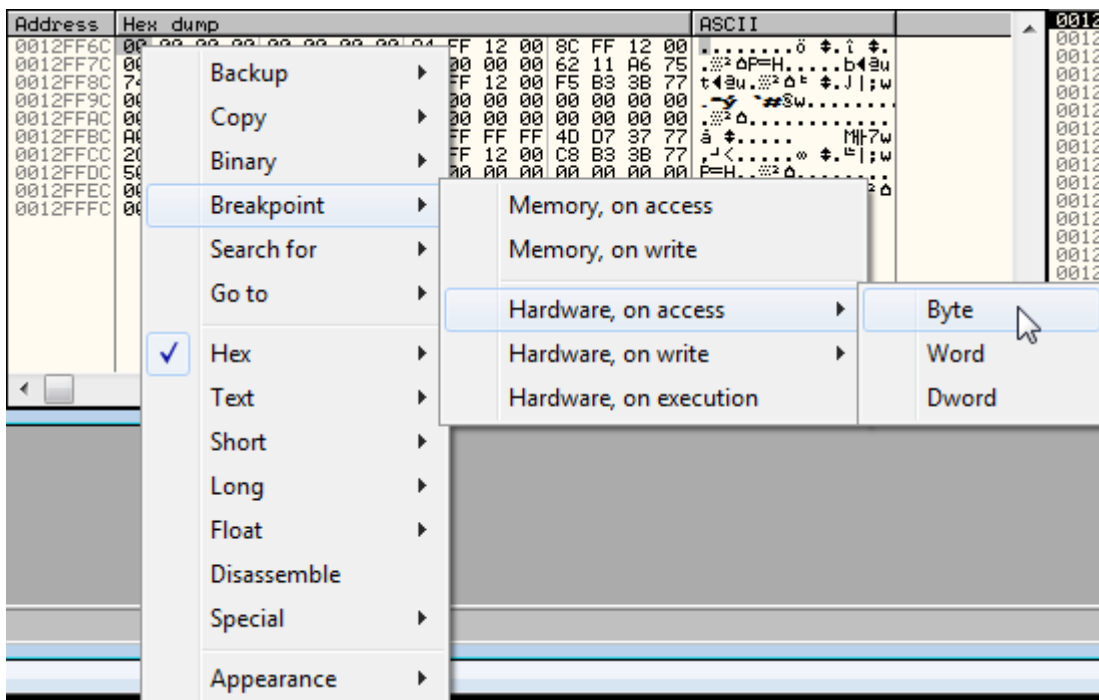
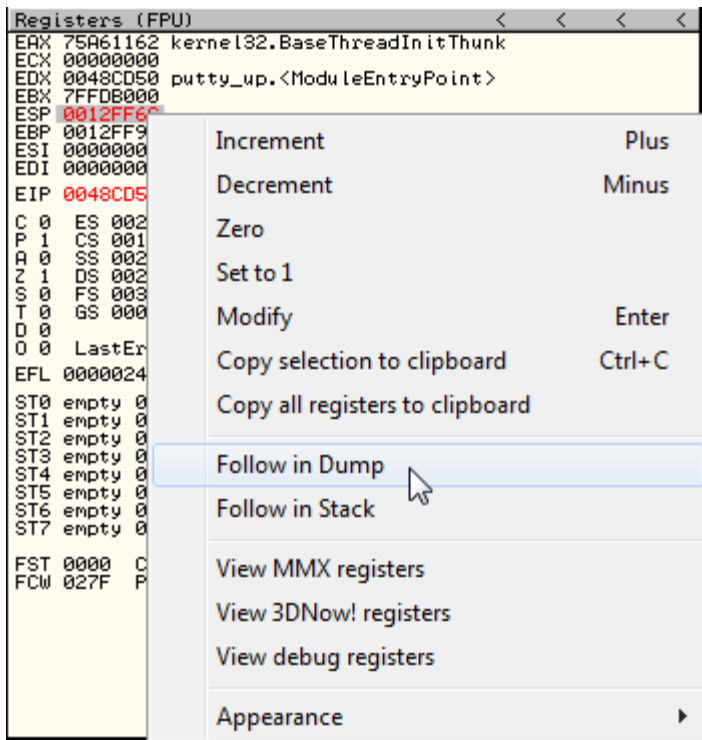
```

CPU - main thread, module putty_up
0048CD50 60 PUSHAD
0048CD51 BE 00004400 MOV ESI,putty_up.00440000
0048CD56 8DBE 0040FBFF LEA EDI,DWORD PTR DS:[ESI+FFFB4000]
0048CD5C 57 PUSH EDI
0048CD5D EB 0B JMP SHORT putty_up.0048CD6A
0048CD5F 90 NOP
0048CD60 8A06 MOV AL,BYTE PTR DS:[ESI]
0048CD62 46 INC ESI
0048CD63 8807 MOV BYTE PTR DS:[EDI],AL
0048CD65 47 INC EDI
  
```

```

0012FF6C 00000000
0012FF70 00000000
0012FF74 0012FF94 preserved registers
0012FF78 0012FF8C
0012FF7C 7FFD3000
0012FF80 0048CD50 putty_up.<ModuleEntryPoint>
0012FF84 00000000
0012FF88 77381162 kernel32.BaseThreadInitThunk
0012FF8C 77381174 RETURN to kernel32.77381174
0012FF90 7FFD3000
0012FF94 0012FFD4
0012FF98 777CB3F5 RETURN to ntdll.777CB3F5
0012FF9C 7FFD3000
0012FFA0 770D7559
0012FFA4 00000000
0012FFA8 00000000
  
```

Follow the ESP register in the hex dump and put a hardware breakpoint (on access) on the memory region pointed to by this register.

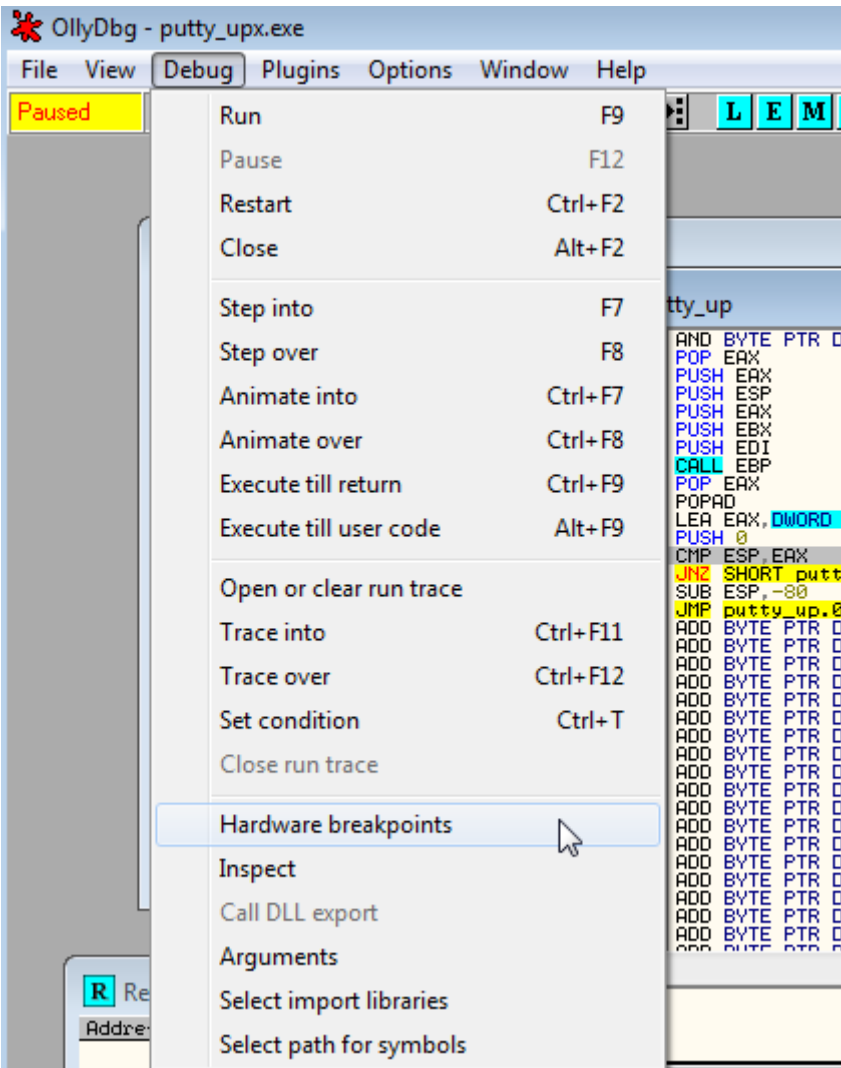


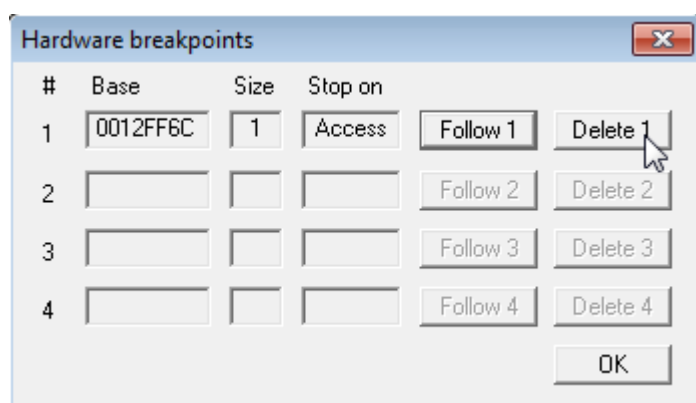
Next resume the execution (F9) and you should immediately land just before the jump to the OEP.

```

CPU - main thread, module putty_up
0048CEEF 8D4424 80 LEA EAX, DWORD PTR SS:[ESP-80]
0048CEFF 6A 00 PUSH 0
0048CF00 39C4 CMP ESP, EAX
0048CF01 ^ 75 FA JNZ SHORT putty_up.0048CEFF
0048CF02 83EC 80 SUB ESP, -80
0048CF03 ^ E9 DFD0FCFF JMP putty_up.00459FE0 ; Jump to the OEP
0048CF04 0000 ADD BYTE PTR DS:[EAX], AL
0048CF05 0048 00 ADD BYTE PTR DS:[EAX], CL
0048CF06 0000 ADD BYTE PTR DS:[EAX], AL
  
```

Remove the hardware breakpoint (Debug -> Hardware breakpoints, Delete).





Now put a breakpoint on the JMP instruction (0x48CEFC), and resume the execution (F9) until you reach it. Step over the JMP instruction (F8) and you should land at the OEP.

In the next step you would need to dump the unpacked process and reconstruct the IAT table in the same way as was described in the previous exercise. (Since it was already done in the previous exercise it is not necessary to do that here now.)

In this exercise you have seen that it is not always necessary to exactly follow the unpacking routine and that in various situations simple tricks can be used to reach the OEP. In this case you used the ESP trick to track the point where there is a jump to the original entry point. While UPX is a fairly easy packer and this trick hasn't eased our task significantly, there are more complex packers for which you can still use the same trick making the unpacking task easier.

3.4 Unpacking a Dyre sample

In this exercise the unpacking of the Dyre²⁷ malware will be presented. Dyre is a banking trojan and was packed using a more complex packer than UPX. Since it is a live malware sample, run it only in a controlled virtual environment and after the analysis restore a clean snapshot of the virtual machine. It is also advisable to forbid any network access while working on Dyre.

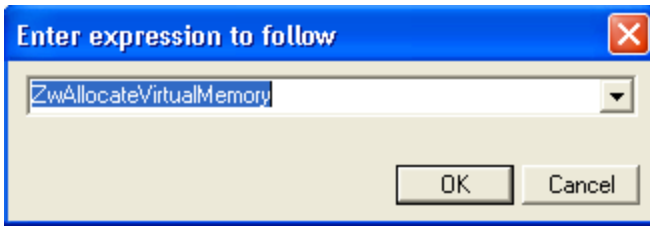
Open the file called *voiyhabs.exe* in the OllyDbg. You should see the entry point.

00401B3D	: C3	RETN	
00401B3E	: E8 EC150000	CALL	voiyhabs.0040312F
00401B43	: ^ E9 89FEFFFF	JMP	voiyhabs.004019D1
00401B48	> 8BFF	MOV	EDI,EDI
00401B4A	: 55	PUSH	EBP
00401B4B	: 8BEC	MOV	EBP,ESP
00401B4D	: 81EC 28030000	SUB	ESP,328

Put a breakpoint on *ZwAllocateVirtualMemory* (as described in the introduction to OllyDbg). This will allow us to track memory allocation operations. Packers often allocate new memory blocks to put unpacked code or unpacking stub there.

²⁷Dyre: Emerging threat on financial fraud landscape

http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/dyre-emerging-threat.pdf (last accessed 19.10.2015)



Now open the *Call stack* window (View->Call Stack, Alt+K) and press the resume execution (F9) a few times until you see a call to *HeapCreate* with flags set to 0x40000. *HEAP_CREATE_ENABLE_EXECUTE*²⁸ is a symbolic constant for 0x40000 meaning that all memory blocks from this heap will allow code execution. This suggests that this heap will be used to store the unpacking stub or some other executable code.

Address	Stack	Procedure / arguments	Called from	Frame
0012F89C	7C925E0C	? ntdll.ZwAllocateVirtualMemory	ntdll.7C925E07	
0012F97C	7C812C8F	? ntdll.RtlCreateHeap	kernel32.7C812C89	
0012F9A0	0040144D	? kernel32.HeapCreate	voiyhabs.00401447	0012F99C
0012F9A4	00040000	Flags = 40000		
0012F9A8	00139000	InitialSize = 139000 (1282048.)		
0012F9AC	00324000	MaximumSize = 324000 (3293184.)		
0012FF34	00401AE6	voiyhabs.00401AE6	voiyhabs.00401AE6	0012FF30
0012FF38	00400000	Arg1 = 00400000		
0012FF3C	00000000	Arg2 = 00000000		
0012FF40	00151F1A	Arg3 = 00151F1A		
0012FF44	0000000A	Arg4 = 0000000A		

Remove the breakpoint from *ZwAllocateVirtualMemory* and show the calling location of *HeapCreate* in the *voiyhabs* module.

0012F89C	7C925E0C	? ntdll.ZwAllocateVirtualMemory	ntdll.7C925E07
0012F97C	7C812C8F	? ntdll.RtlCreateHeap	kernel32.7C812C89
0012F9A0	0040144D	? kernel32.HeapCreate	voiyhabs.00401447
0012F9A4	00040000	Flags = 40000	
0012F9A8	00139000	InitialSize = 139000 (
0012F9AC	00324000	MaximumSize = 324000 (
0012FF34	00401AE6	voiyhabs.00401AE6	
0012FF38	00400000	Arg1 = 00400000	
0012FF3C	00000000	Arg2 = 00000000	
0012FF40	00151F1A	Arg3 = 00151F1A	
0012FF44	0000000A	Arg4 = 0000000A	

Actualize

Hide arguments Space

Follow address in stack

Show procedure Enter

Show call

Execute to return F4

Copy to clipboard ▶

Appearance ▶

Put a breakpoint on the instruction after the *HeapCreate* call and resume the execution (F9). Write down the address of the newly created heap (returned in *EAX* register, *0xDE0000* in this example, might be different).

00401444	> 57	PUSH EDI	MaximumSize = 324000 (3293184.) InitialSize = 139000 (1282048.) Flags = HEAP_CREATE_ENABLE_TRAC:
00401445	. 56	PUSH ESI	
00401446	. 50	PUSH EAX	
00401447	. FF15 C4704000	CALL DWORD PTR DS:[&&KERNEL32.He	
0040144D	. 894424 40	MOV DWORD PTR SS:[ESP+40], EAX	HeapCreate
00401451	. 3BC3	CMP EAX, EBX	

²⁸HeapCreate function <https://msdn.microsoft.com/en-us/library/windows/desktop/aa366599%28v=vs.85%29.aspx> (last accessed 11.09.2015)

```
Registers (FPU)
EAX 00DE0000
ECX 7C926090 ntdll.7C926090
EDX 7C97B380 ntdll.7C97B380
EBX 00000000
ESP 0012F9B0
EBP 0012FF30
ESI 00139000
EDI 00324000
EIP 00401440 voiyhabs.00401440
```

Remove the previously set breakpoint (0x40144D) and scroll down until you see a call to *RegisterClassEx* function (0x4018DE). Put a breakpoint on this function and resume the execution (F9).

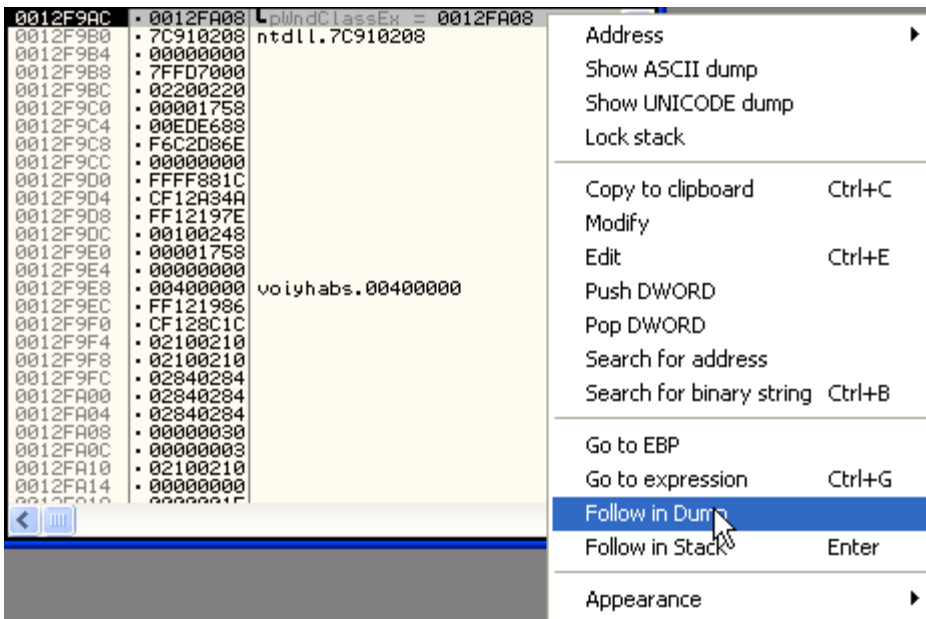
004018BC	50	PUSH EAX	
004018BD	C74424 7C 100000	MOV DWORD PTR SS:[ESP+7C],10	pWndClassEx = 0012FA08
004018C5	899C24 80000000	MOV DWORD PTR SS:[ESP+80],EBX	
004018CC	C78424 84000000	MOV DWORD PTR SS:[ESP+84],voiyhabs	ASCII "MY_EXCLUSIVE_CLASS"
004018D7	899C24 88000000	MOV DWORD PTR SS:[ESP+88],EBX	
004018DE	FF15 5C714000	CALL DWORD PTR DS:[<&USER32.Reg	RegisterClassExA
004018E4	53	PUSH EBX	lParam = 0
004018E5	53	PUSH EBX	pDlgProc = NULL
004018E6	53	PUSH EBX	hOwner = NULL
004018E7	68 E0030000	PUSH 3E8	pTemplate = 3E8
004018EC	FF7424 48	PUSH DWORD PTR SS:[ESP+48]	hInst = 02100210
004018F0	FF15 58714000	CALL DWORD PTR DS:[<&USER32.Crea	CreateDialogParamA
004018F6	6A 05	PUSH 5	ShowState = SW_SHOW
004018F8	50	PUSH EAX	hWnd = 0012FA08
004018F9	FF15 84714000	CALL DWORD PTR DS:[<&USER32.Show	ShowWindow

Next you will check the address of the window procedure in a registered window class. Hiding some code in a window procedure is a common technique used by a malware to hinder analysis and change the execution flow. If the window procedure points to an existing address it is good to put a breakpoint at this address.

The window procedure address is passed in a third field of the *WndClassEx* structure (*lpfnWndProc*) preceded by two INT values. This means that this address is a third DWORD value in the *WndClassEx* structure.

```
typedef struct tagWNDCLASSEX {
    UINT        cbSize;
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
    LPCTSTR     lpszClassName;
    HICON       hIconSm;
} WNDCLASSEX, *PWNDCLASSEX;
```

Address of the *WndClassEx* structure is put as the first argument onto the stack. Follow it in the dump.



Now read the address of the Window procedure from the hex dump remembering that addresses are written in the little-endian notation. In this case the Window procedure address is *0x02100210*.

Address	Hex dump	ASCII
0012FA08	30 00 00 00 03 00 00 00 10 02 10 02 00 00 00 00	0...♥...>B>B...
0012FA18	1E 00 00 00 00 00 40 00 07 00 01 00 11 00 01 00	▲.....@..@.↓.@.
0012FA28	10 00 00 00 00 00 00 00 20 C8 40 00 00 00 00 00	▶.....L@.....
0012FA38	01 03 01 03 01 03 01 03 01 03 01 03 01 03 01 03	0*000*000*000*00
0012FA48	01 03 01 03 01 03 01 03 ED 02 10 02 10 02 10 02	0*000*000*000*00

Next try to follow this address in the assembly window. If you land in the existing code section, put a breakpoint at this address. To go back to the current location you can follow the EIP register.

Next scroll down until a call to *EnumDisplayMonitors* function. Put a breakpoint on this call and resume the execution (F9) or alternatively select this location and use *Run to selection* (F4).

00401941	FF7424 18	PUSH DWORD PTR SS:[ESP+18]	
00401945	53	PUSH EBX	
00401946	53	PUSH EBX	
00401947	FF15 4C714000	CALL DWORD PTR DS:[&USER32.EnumDisplayMonitors]	USER32.EnumDisplayMonitors
0040194D	EB 02	JMP SHORT voiyhabs.00401951	
0040194F	33C0	XOR EAX,EAX	
00401951	8B8C24 7C050000	MOV ECX, DWORD PTR SS:[ESP+57C]	voiyhabs.00404234

```

BOOL EnumDisplayMonitors (
    _In_   HDC          hdc,
    _In_   LPCRECT     lpRectClip,
    _In_   MONITORENUMPROC lpfnEnum,
    _In_   LPARAM      dwData
);

```

Check the address of the enumeration procedure (*lpfnEnum*) on the stack (the third argument, in this example *0xEDE688*). Notice that this address points to the memory range of the previously created heap. This suggests that the unpacking stub is likely located there.

0012F9A0	• 00000000	
0012F9A4	• 00000000	
0012F9A8	• 00EDE688	lpfnEnum
0012F9AC	• 00000000	
0012F9B0	• 7C910208	ntdll.7C910208
0012F9B4	• 00000000	

Put a breakpoint on the enumeration procedure (in the assembly window go to the address of the enumeration procedure – *lpfnEnum* and toggle a breakpoint on this address). Resume the execution (F9).

```

00EDE686 | 0900 | OR DWORD PTR DS:[EAX],EAX
00EDE688 | B8 E8F91200 | MOV EAX,12F9E8
00EDE68D | E9 76140000 | JMP 00EDFB08
00EDE692 | CC | INT3
00EDE693 | CC | INT3

```

The execution should break inside the enumeration procedure. Step over (F8) two times to follow the jump.

```

00EDFB06 | CC | INT3
00EDFB07 | CC | INT3
00EDFB08 | 55 | PUSH EBP
00EDFB09 | 8BEC | MOV EBP,ESP
00EDFB0B | 83EC 38 | SUB ESP,38
00EDFB0E | 8945 F4 | MOV DWORD PTR SS:[EBP-C],EAX
00EDFB11 | E8 42EEFFFF | CALL 00EDE958
00EDFB16 | E8 0DFFFFFF | CALL 00EDFAF8
00EDFB1B | 8945 EC | MOV DWORD PTR SS:[EBP-14],EAX
00EDFB1E | 8B45 EC | MOV EAX,DWORD PTR SS:[EBP-14]

```

Scroll down and put a breakpoint on the suspicious *CALL EAX* just before the function return (*RETN*). As mentioned in the introduction, single calls to a register just before a function return might indicate calls to the OEP or some other important part of the code. It is also worth noting that there isn't much going on after this call. This means that the jump to the OEP is likely taking place in this call.

```

00EDFBD9 | FF75 DC | PUSH DWORD PTR SS:[EBP-24]
00EDFBD0 | FF75 E4 | PUSH DWORD PTR SS:[EBP-1C]
00EDFBE0 | 8B45 FC | MOV EAX,DWORD PTR SS:[EBP-4]
00EDFBE3 | FFD0 | CALL EAX
00EDFBE5 | 8BE5 | MOV ESP,EBP
00EDFBE7 | 5D | POP EBP
00EDFBE8 | C3 | RETN

```

Resume the execution (F9). When you hit the breakpoint on *CALL EAX* step into (F7) the call. You should land in another unpacking stub function.

```

00EE16B8 | 0000 | ADD BYTE PTR DS:[EAX],AL
00EE16BA | 0000 | ADD BYTE PTR DS:[EAX],AL
00EE16BC | B8 00000000 | MOV EAX,0
00EE16C1 | E9 26220000 | JMP 00EE38EC
00EE16C6 | CC | INT3
00EE16C7 | CC | INT3

```

Step over (F8) two times.

```

00EE38EA | CC | INT3
00EE38EB | CC | INT3
00EE38EC | 55 | PUSH EBP
00EE38ED | 8BEC | MOV EBP,ESP
00EE38EF | 83EC 44 | SUB ESP,44
00EE38F2 | 8945 FC | MOV DWORD PTR SS:[EBP-4],EAX
00EE38F5 | E8 A2E0FFFF | CALL 00EE199C
00EE38FA | 837D FC 00 | CMP DWORD PTR SS:[EBP-4],0

```

Now scroll down to a group of three calls just before the function return. At this point you already know that after the return (to the previous routine) there isn't much going on in the code. This means that the jump to OEP is likely taking place in one of this calls. Put a breakpoint on the first call and resume the execution (F9).

```

00EE3A31 | 50 | PUSH EAX
00EE3A32 | 6A 00 | PUSH 0
00EE3A34 | 6A 00 | PUSH 0
00EE3A36 | E8 11E2FFFF | CALL 00EE1C4C
00EE3A38 | 6A 00 | PUSH 0
00EE3A3D | E8 DAE1FFFF | CALL 00EE1C1C
00EE3A42 | E8 75DCFFFF | CALL 00EE16BC
00EE3A47 | 8BE5 | MOV ESP,EBP
00EE3A49 | 5D | POP EBP
00EE3A4A | C2 1000 | RETN 10
00EE3A4D | CC | INT3
00EE3A4E | CC | INT3

```

After reaching the breakpoint step into the first function call (F7). You will see several PUSH instructions followed by a call instruction.

55	PUSH EBP	
8BEC	MOV EBP,ESP	
83EC 0C	SUB ESP,0C	
E8 C5FCFFFF	CALL 01971824	
8945 F8	MOV DWORD PTR SS:[EBP-8],EAX	
8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]	
8B48 40	MOV ECX,DWORD PTR DS:[EAX+40]	
894D F4	MOV DWORD PTR SS:[EBP-C],ECX	
FF75 1C	PUSH DWORD PTR SS:[EBP+1C]	
FF75 18	PUSH DWORD PTR SS:[EBP+18]	
FF75 14	PUSH DWORD PTR SS:[EBP+14]	
FF75 10	PUSH DWORD PTR SS:[EBP+10]	
FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
FF75 08	PUSH DWORD PTR SS:[EBP+8]	
FF55 F4	CALL DWORD PTR SS:[EBP-C]	voiyhabs.00400000
8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]	
8BE5	MOV ESP,EBP	

When you step over (F8) to this call instruction you will see this is a call to *CreateThread* function.

55	PUSH EBP	
8BEC	MOV EBP,ESP	
83EC 0C	SUB ESP,0C	
E8 C5FCFFFF	CALL 01971824	
8945 F8	MOV DWORD PTR SS:[EBP-8],EAX	
8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]	
8B48 40	MOV ECX,DWORD PTR DS:[EAX+40]	kernel32.CreateThread
894D F4	MOV DWORD PTR SS:[EBP-C],ECX	kernel32.CreateThread
FF75 1C	PUSH DWORD PTR SS:[EBP+1C]	
FF75 18	PUSH DWORD PTR SS:[EBP+18]	
FF75 14	PUSH DWORD PTR SS:[EBP+14]	
FF75 10	PUSH DWORD PTR SS:[EBP+10]	
FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
FF75 08	PUSH DWORD PTR SS:[EBP+8]	
FF55 F4	CALL DWORD PTR SS:[EBP-C]	kernel32.CreateThread
8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]	
8BE5	MOV ESP,EBP	

```

HANDLE WINAPI CreateThread(
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_     SIZE_T dwStackSize,
    _In_     LPTHREAD_START_ROUTINE lpStartAddress,
    _In_opt_ LPVOID lpParameter,
    _In_     DWORD dwCreationFlags,
    _Out_opt_ LPDWORD lpThreadId
);

```

Now take a look at the stack. The thread routine is passed as the third argument. In this example it points to *0xEE38AC*.

0012F86C	00000000	
0012F870	00000000	
0012F874	00EE38AC	lpStartAddress
0012F878	0015CF98	
0012F87C	00000000	
0012F880	00000000	
0012F884	7C8106C7	kernel32.CreateThread
0012F888	00EE1926	
0012F88C	00000000	

Put a breakpoint on the thread function (*lpStartAddress*) and resume the execution (F9).

00EE38AB	CC	INT3	
00EE38AC	8B4424 04	MOV EAX,DWORD PTR SS:[ESP+4]	
00EE38B0	83F8 00	CMP EAX,0	
00EE38B3	75 0B	JNZ SHORT 00EE38C0	
00EE38B5	E8 00000000	CALL 00EE38BA	
00EE38BA	58	POP EAX	kernel132.7C80B713
00EE38BB	83E8 0E	SUB EAX,0E	
00EE38BE	EB 1B	JMP SHORT 00EE38D8	
00EE38C0	8B48 10	MOV ECX,DWORD PTR DS:[EAX+10]	
00EE38C3	51	PUSH ECX	
00EE38C4	8B48 0C	MOV ECX,DWORD PTR DS:[EAX+C]	
00EE38C7	51	PUSH ECX	
00EE38C8	8B48 08	MOV ECX,DWORD PTR DS:[EAX+8]	
00EE38CB	51	PUSH ECX	
00EE38CC	8B48 04	MOV ECX,DWORD PTR DS:[EAX+4]	
00EE38CF	51	PUSH ECX	
00EE38D0	8B08	MOV ECX,DWORD PTR DS:[EAX]	
00EE38D2	64:8B1D 30000000	MOV EBX,DWORD PTR FS:[30]	
00EE38D9	FFD1	CALL ECX	
00EE38DB	C2 0400	RETN 4	
00EE38DE	CC	INT3	

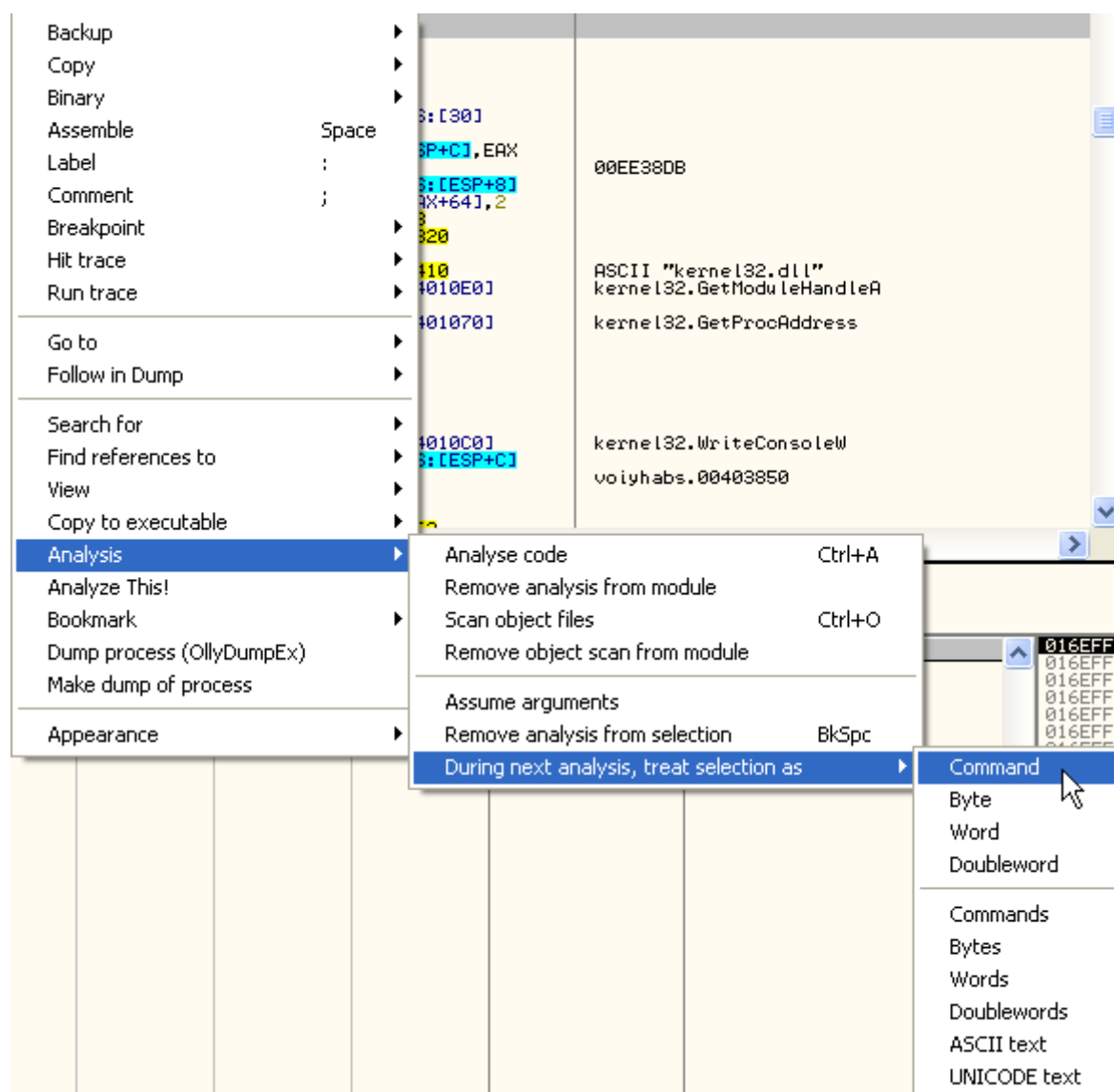
When a breakpoint at the thread function is hit step over (F8) until a call to ECX. As you see ECX points to the memory of the .text section of the original executable (voiyhabs.00403850). This is a good indicator that you are jumping to the OEP. Step into the call (F7).

00EE38D0	8B08	MOV ECX,DWORD PTR DS:[EAX]	voiyhabs.00403850
00EE38D2	64:8B1D 30000000	MOV EBX,DWORD PTR FS:[30]	
00EE38D9	FFD1	CALL ECX	voiyhabs.00403850
00EE38DB	C2 0400	RETN 4	
00EE38DE	CC	INT3	

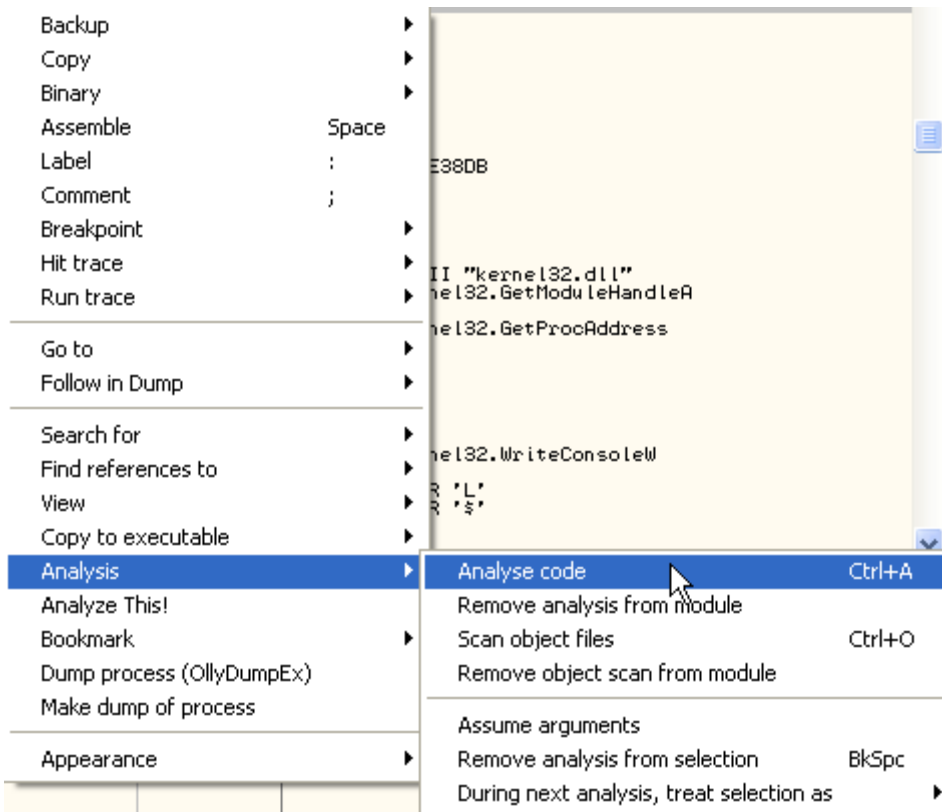
You should land at the OEP with overwritten code which OllyDbg hasn't analysed properly. At this point you could proceed to dump the process and reconstruct the IAT, but to be sure that you are at the OEP you need to tell OllyDbg to interpret the following fragment as code and disassemble it properly.

0040384E	. CC	INT3	
0040384F	? CC	INT3	
00403850	. 55	PUSH EBP	
00403851	? 8BEC	MOV EBP,ESP	
00403853	? 83E4 F8	AND ESP,FFFFFFF8	
00403856	. 83EC 2C	SUB ESP,2C	
00403859	> 56	PUSH ESI	
0040385A	? 50	PUSH EAX	
0040385B	? 64:A1 30000000	MOV EAX,DWORD PTR FS:[30]	
00403861	? 85DB	TEST EBX,EBX	
00403863	? 894424 0C	MOV DWORD PTR SS:[ESP+C],EAX	
00403867	? 58	POP EAX	00EE38DB
00403868	? 8B4424 08	MOV EAX,DWORD PTR SS:[ESP+8]	

Starting at the OEP address select a group of instructions. Next right-click on them and from the context menu choose 'Analysis->During next analysis, treat selection as->Command'.



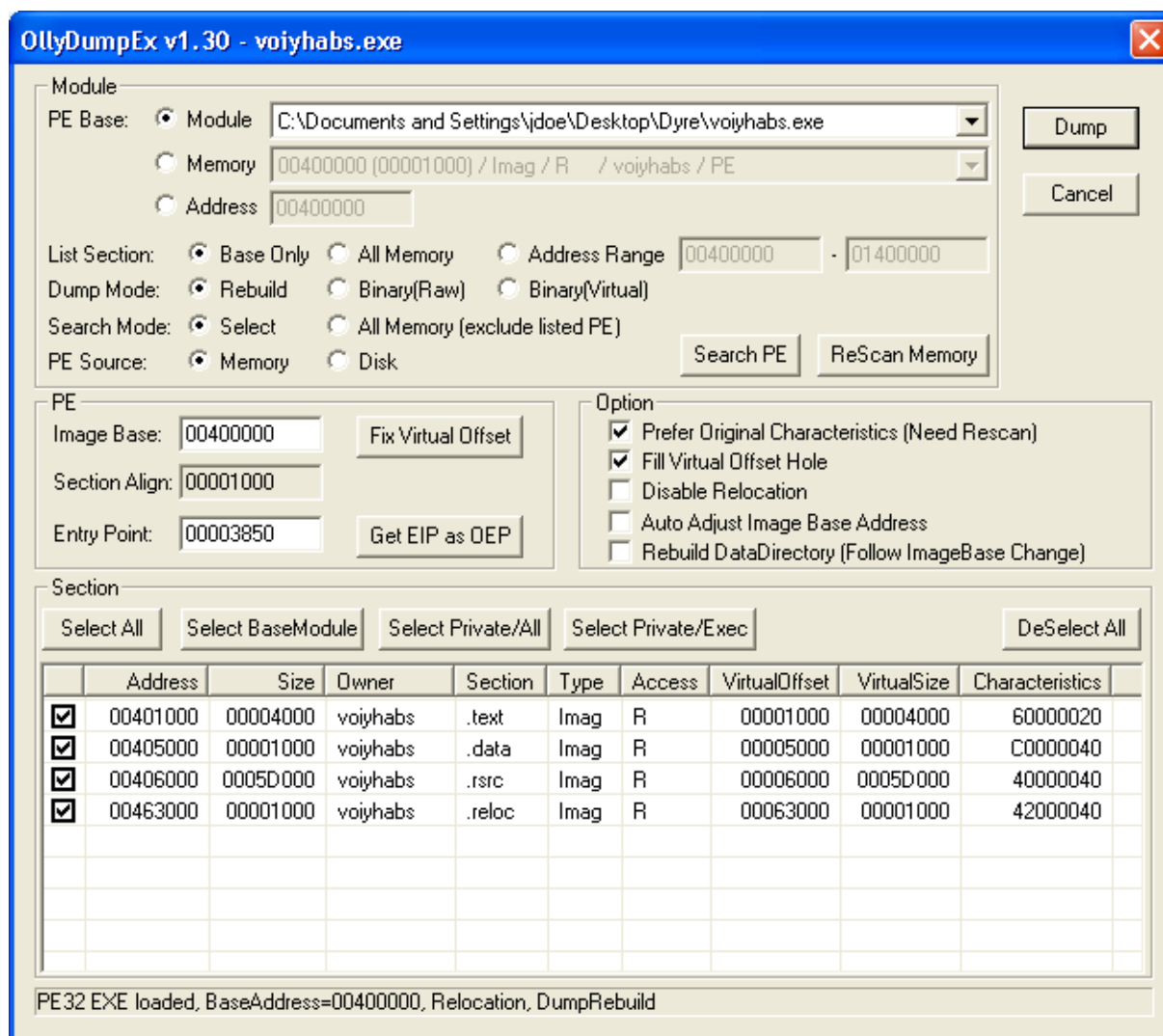
Now click on them once again and from the context menu choose 'Analysis->Analyse code'.



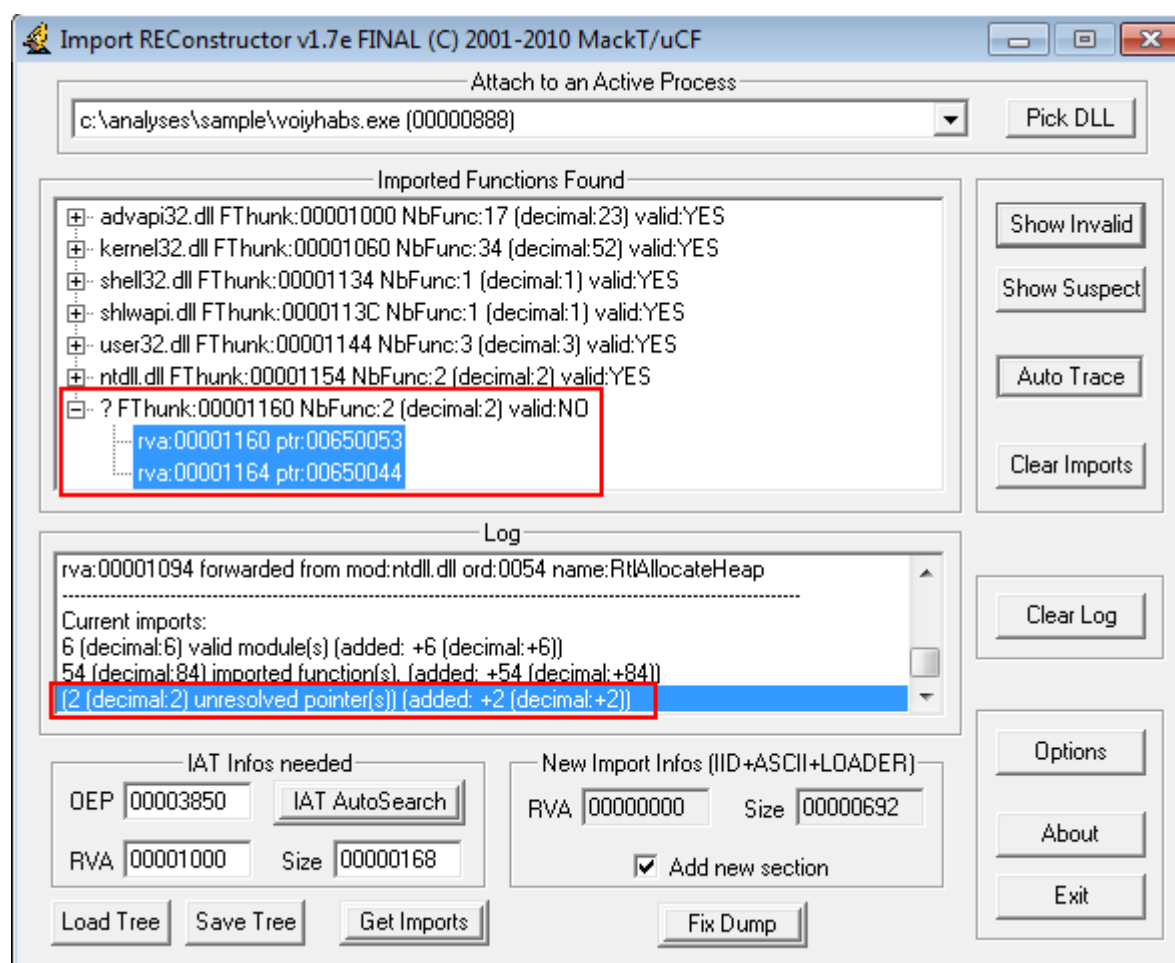
Now you can clearly see that you are most likely at the OEP - typical function prologue with the EBP based stack frame followed by later call to *GetModuleHandleA*. Moreover you just jumped from a code in the allocated memory block to the initial executable section: another indicator that you are at the original entry point.

0040384F	CC	INT3	
00403850	55	PUSH EBP	
00403851	8BEC	MOV EBP,ESP	
00403853	83E4 F8	AND ESP,FFFFFFF8	
00403856	83EC 2C	SUB ESP,2C	
00403859	56	PUSH ESI	
0040385A	50	PUSH EAX	
0040385B	64:A1 30000000	MOV EAX,DWORD PTR FS:[30]	
00403861	85DB	TEST EBX,EBX	
00403863	894424 0C	MOV DWORD PTR SS:[ESP+C],EAX	
00403867	58	POP EAX	00EE38DB
00403868	8B4424 08	MOV EAX,DWORD PTR SS:[ESP+8]	
0040386C	8378 64 02	CMP DWORD PTR DS:[EAX+64],2	
00403870	0F82 E0000000	JB voiyhabs.00403963	
00403876	E8 A5FFFFFF	CALL voiyhabs.00403820	
0040387B	50	PUSH EAX	
0040387C	68 10144000	PUSH voiyhabs.00401410	ASCII "kernel32.dll"
00403881	FF15 E0104000	CALL DWORD PTR DS:[4010E0]	kernel32.GetModuleHandleA

Next you will dump the process image and reconstruct the IAT table. To dump the process use OllyDumpEx plugin (as described previously).



Now try to reconstruct the Import Address Table (RVA of the OEP: *0x3850*) by using *ImpRec*. This time you might see some invalid imports after clicking *Get Imports* and *Show Invalid*. Right-click on each of them and from the context menu choose *Cut thunks(s)*. After all invalid pointers are resolved, use *Fix dump* to fix the dumped executable.



In this exercise you have unpacked the executable of a real malware sample, protected with a more complex packer. You have achieved this by first tracking the memory allocation operations and then following the unpacking stub in a newly allocated heap. It is worth noting that this isn't the only way of unpacking this executable nor is it the quickest method. As goes for any packed executable there are many ways of unpacking code and reaching the OEP.

4. Anti-debugging techniques

4.1 Anti-debugging and anti-analysis techniques

Malware creators usually don't want malware analysts to be able to analyse their code. As a consequence they use various anti-analysis techniques to make analysis as hard as possible. You can distinguish four groups of anti-analysis techniques:

- *Anti-debugging* – detects if the process is being debugged
- *Anti-emulation (anti-VM)* – detects if the process is running in a virtual machine or in some other emulated environment
- *Anti-sandbox* – detects if the process was executed in some well-known sandbox or environment dedicated for malware analysis
- *Anti-disassembly* – makes disassemblers to incorrectly disassemble code

When debugging, most often you would need to cope with the anti-debugging and anti-VM techniques. Whenever malicious code detects that it is being debugged or is running in a virtual machine, it might terminate or run completely other (non-malicious) code instead, to mislead the analyst.

There are plenty of anti-debugging techniques^{29 30}. Most of them can be assigned to one of the following categories:

- *API related techniques* – those techniques use the fact that calls to certain API functions would return different result depending on whether the application is being debugged or not. Examples of such functions are *IsDebuggerPresent* or *OutputDebugString*.
- *Checking flags* – certain flags set by an operating system in process's data structures are different when the process is being debugged. Examples of such flags are *NtGlobalFlag* and *IsDebugged* flag in (PEB).
- *Searching for breakpoints* – it is possible for a process to search for breakpoints in its current address space. This applies for software, hardware and memory breakpoints.
- Searching for processes (or open windows) of popular debuggers and other analysis tools (e.g. *Wireshark*, *Regshot*, *Process Explorer*).
- *Time based checks* – a malicious code can check how much time elapsed between two different parts of the code. If the time delay is too big it will be assumed that the application is being debugged.
- *Self-debugging* – a clever anti-debugging technique in which the malware starts debugging its own processes making the analyst unable to attach a debugger to them. This technique was used by Zero-Access trojan³¹.

²⁹The "Ultimate" Anti-Debugging Reference <http://pferrie.host22.com/papers/antidebug.pdf> (last accessed 11.09.2015)

³⁰Anti-debugging Techniques Cheat Sheet <http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/> (last accessed 11.09.2015)

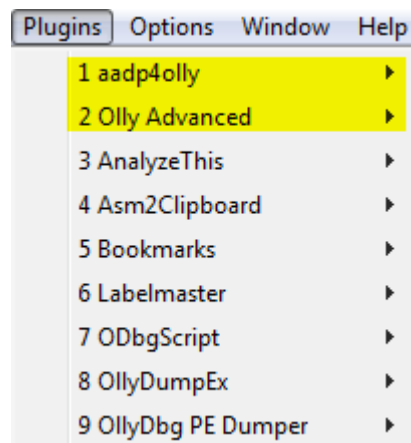
³¹ZeroAccess uses Self-Debugging <https://blog.malwarebytes.org/intelligence/2013/07/zeroaccess-anti-debug-uses-debugger/> (last accessed 11.09.2015)

Except trying to detect the debugger, the malicious code is frequently also trying to detect virtualization environments using a subset of the following techniques³²:

- *Detecting processes related to the virtualization software.* This is one of the most common techniques. A malicious code iterates a process list in search for processes like *VBoxService.exe*, *VBoxTray.exe*.
- *Searching for VM artefacts in the registry and filesystem* (strings referencing *vbox*, *vmware*, *qemu*).
- *Checking the amount of resources available in the system.* For example a malicious code might check the size of the hard disk assuming that most modern computers should have hard disks of size at least 80GB (VMs frequently have smaller disks).
- Detecting what hardware is present in the system (e.g. *VBOX HARDDISK*).
- Certain assembly instructions also behave differently on the virtualized system than on bare-metal.

One of the countermeasures for anti-debugging is to use special plugins for OllyDbg like *aadp4olly*³³ and *Olly Advanced*³⁴.

When using those plugins you need to check which anti-anti-debugging techniques should be used. You can do this by accessing the plugin's options dialog via *Plugins* menu.



In general it is a good idea to use only one plugin for a specific anti-debugging technique as otherwise it might lead to unexpected behaviour. If a certain plugin doesn't work or crashes, try a different one.

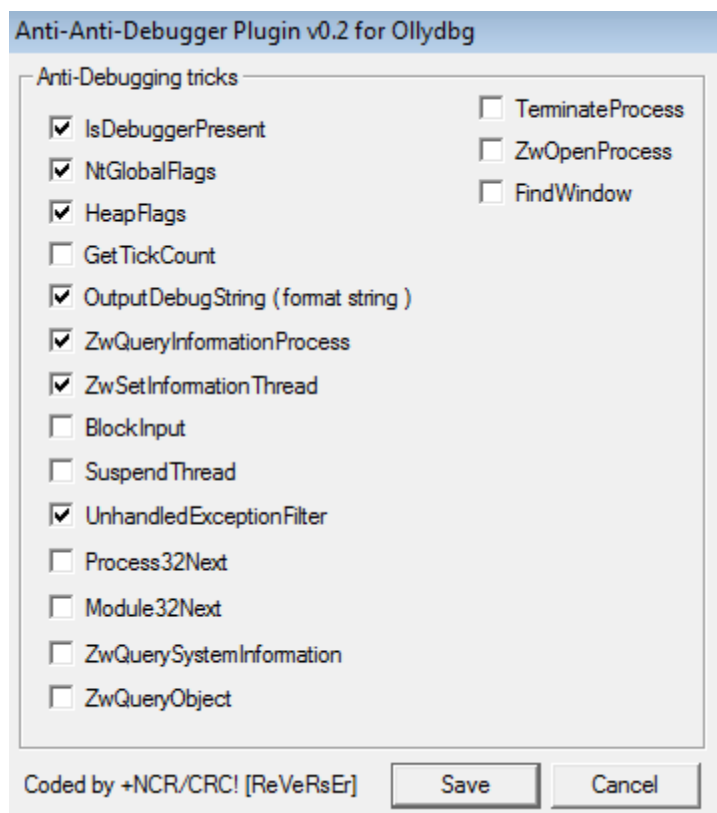
The screenshot below presents the anti-anti-debugging options of *aadp4olly* plugin.

³²On the Cutting Edge: Thwarting Virtual Machine Detection

http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf (last accessed 11.09.2015)

³³*aadp4olly* <https://tuts4you.com/download.php?view.3021> (last accessed 11.09.2015)

³⁴*Olly Advanced* <https://tuts4you.com/download.php?view.75> (last accessed 11.09.2015)



However even when you are using such plugins you need to still be cautious for anti-VM techniques and more sophisticated anti-debugging techniques. One way to check if the malicious code is using anti-VM or anti-debugging techniques is to try and run it freely (under debugger) and then using a behavioural analysis techniques to check if the malware behaves as expected.

In a typical scenario malware would create one or more child processes, install itself somewhere in the system and finally generate some network traffic to communicate with its C&C server. If you don't see such behaviour it might mean that the malware has detected the debugger or VM or just that the malware you are analysing doesn't behave in this way.

4.2 Dyre - basic patching with OllyDbg

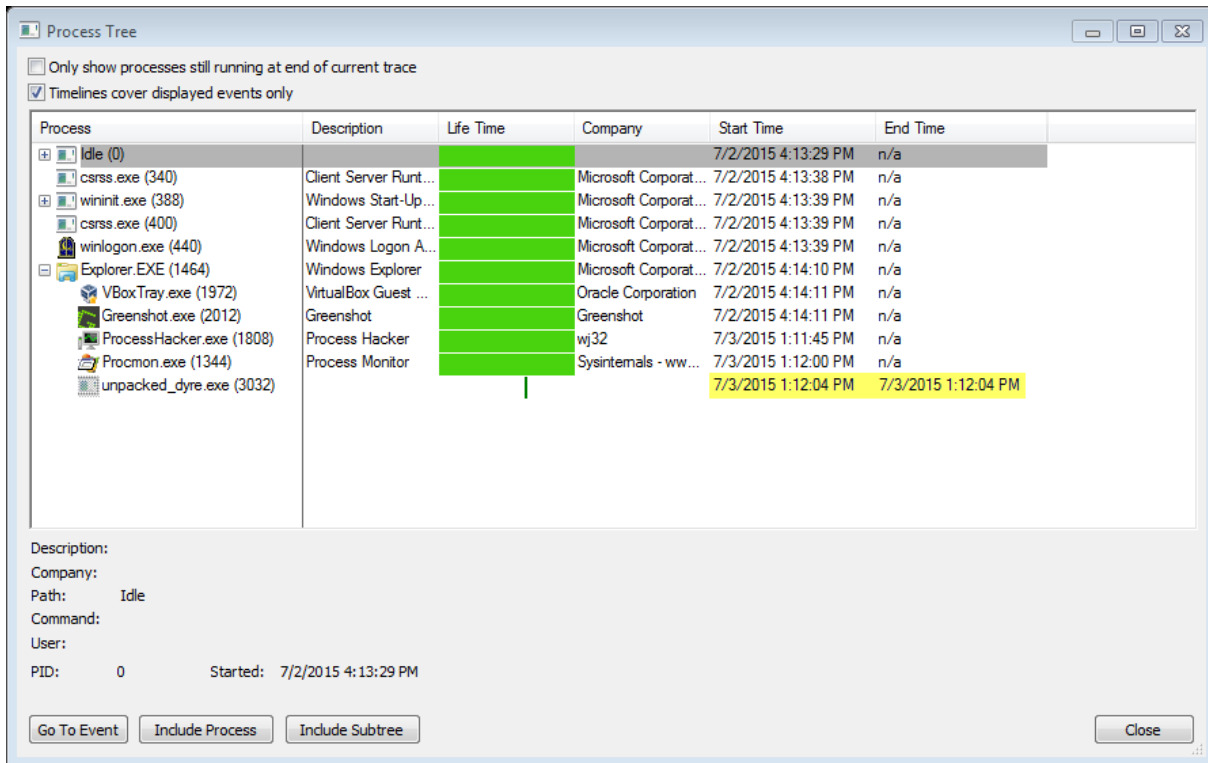
A recent version of the Dyre trojan uses an interesting anti-sandbox mechanism. It checks the number of processor cores visible by the system³⁵ and if it determines that the number of cores is less than two it stops the execution. Since the most modern systems run on multi-core CPUs it won't affect them while it would still prevent the execution on a poorly configured sandbox running on virtual machines with only one CPU attached.

In this exercise you will use the previously unpacked Dyre sample and patch it to allow code execution also on one core system. If you haven't finished the previous exercise or something went wrong, use the `unpacked_dyre.exe` sample provided.

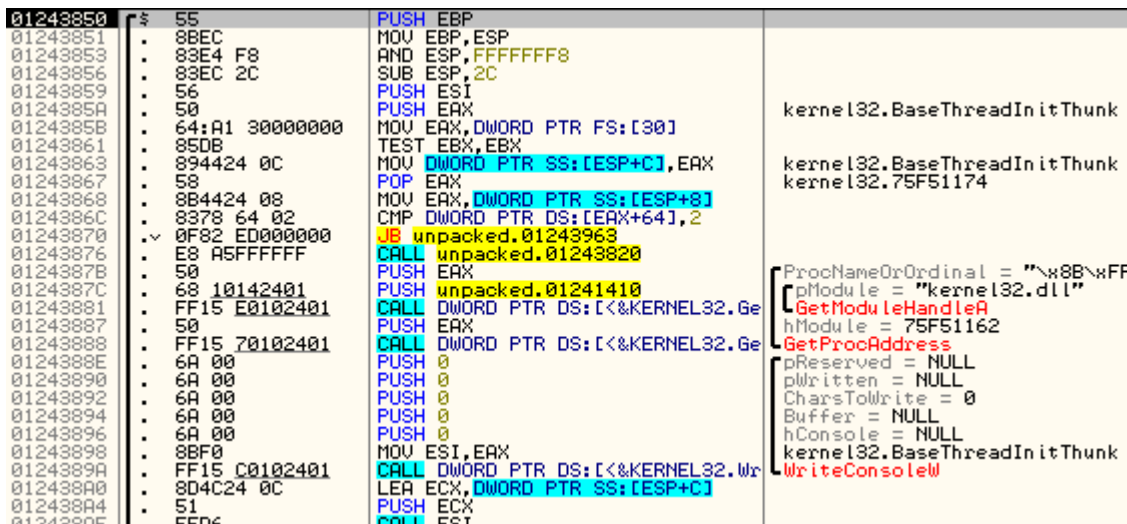
If your virtual machine has only one CPU configured you can start Process Explorer/Process Monitor and try to execute the unpacked Dyre sample. You will observe that the sample quits almost immediately and nothing much

³⁵ Dyre <http://www.seculert.com/blog/2015/04/new-dyre-version-evades-sandboxes.html> (last accessed 11.09.2015)

seems to be happening. The screenshot below shows the Process Tree view as created by Process Monitor tool (*Tools->Process tree*).



Now open the unpacked Dyre in OllyDbg.



If you step over (F8) a few times you will notice that at the first jump instruction (JB) the program is jumping to the *ExitProcess* routine.

```

002E3859 | . | 56 | PUSH ESI
002E385A | . | 50 | PUSH EAX
002E385B | . | 64:A1 30000000 | MOV EAX,DWORD PTR FS:[30]
002E3861 | . | 85DB | TEST EBX,EBX
002E3863 | . | 894424 0C | MOV DWORD PTR SS:[ESP+C],EAX
002E3867 | . | 58 | POP EAX
002E3868 | . | 8B4424 08 | MOV EAX,DWORD PTR SS:[ESP+8]
002E386C | . | 8378 64 02 | CMP DWORD PTR DS:[EAX+64],2
002E3870 | . | 0F82 ED000000 | JB unpacked.002E3963
002E3876 | . | E8 A5FFFFFF | CALL unpacked.002E3820
002E387B | . | 50 | PUSH EAX
002E387C | . | 68 10142E00 | PUSH unpacked.002E1410
| | | | ProcNameOrOrdinal = ""
| | | | ProModule = "kernel32.dll"

002E3957 | . | E8 74F8FFFF | CALL unpacked.002E31D0
002E395C | . | EB 05 | JMP SHORT unpacked.002E3963
002E395E | . | E8 80FAFFFF | CALL unpacked.002E33F0
002E3963 | . | 5A 00 | PUSH 0
| | | | ExitCode = 0
| | | | ExitProcess
002E3965 | . | FF15 B8102E00 | CALL DWORD PTR DS:[<&KERNEL32.Ex
002E3968 | . | CC | INT3
002E396C | . | CC | INT3
002E396D | . | CC | INT3
002E396E | . | CC | INT3

```

If you take a closer look at the code just before the jump, you notice that Dyre is checking the number of CPU cores as pointed by the Process Environment Block (PEB)³⁶. The Process Environment Block is a special system structure containing various information about the running process. It is stored in user space memory and pointed to by the FS segment.

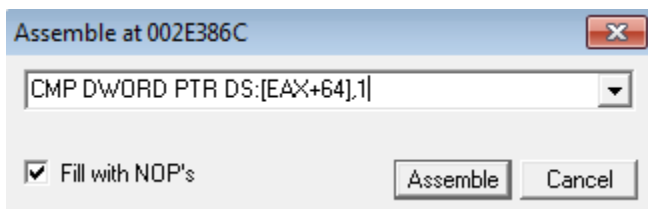
If the value is less than two it terminates the process.

```

002E385B | . | 64:A1 30000000 | MOV EAX,DWORD PTR FS:[30]
002E3861 | . | 85DB | TEST EBX,EBX
002E3863 | . | 894424 0C | MOV DWORD PTR SS:[ESP+C],EAX
002E3867 | . | 58 | POP EAX
002E3868 | . | 8B4424 08 | MOV EAX,DWORD PTR SS:[ESP+8]
002E386C | . | 8378 64 02 | CMP DWORD PTR DS:[EAX+64],2
002E3870 | . | 0F82 ED000000 | JB unpacked.002E3963

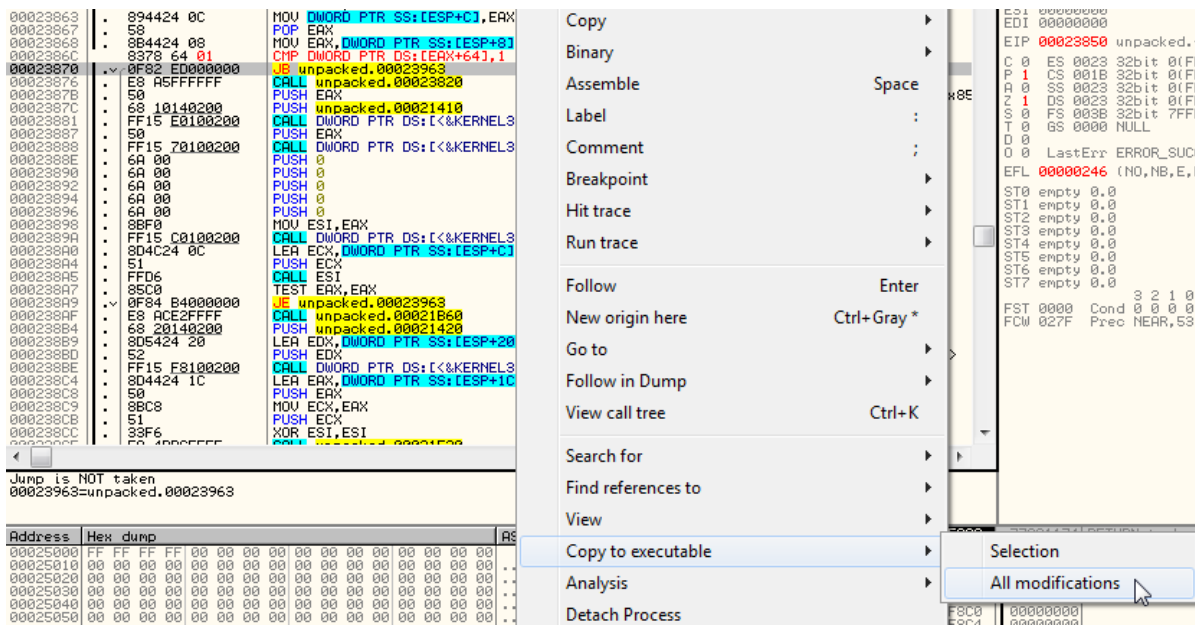
```

To patch this behaviour click on *CMP* instruction and press space (or select *Assemble* from context menu). Replace value '2' with '1'.

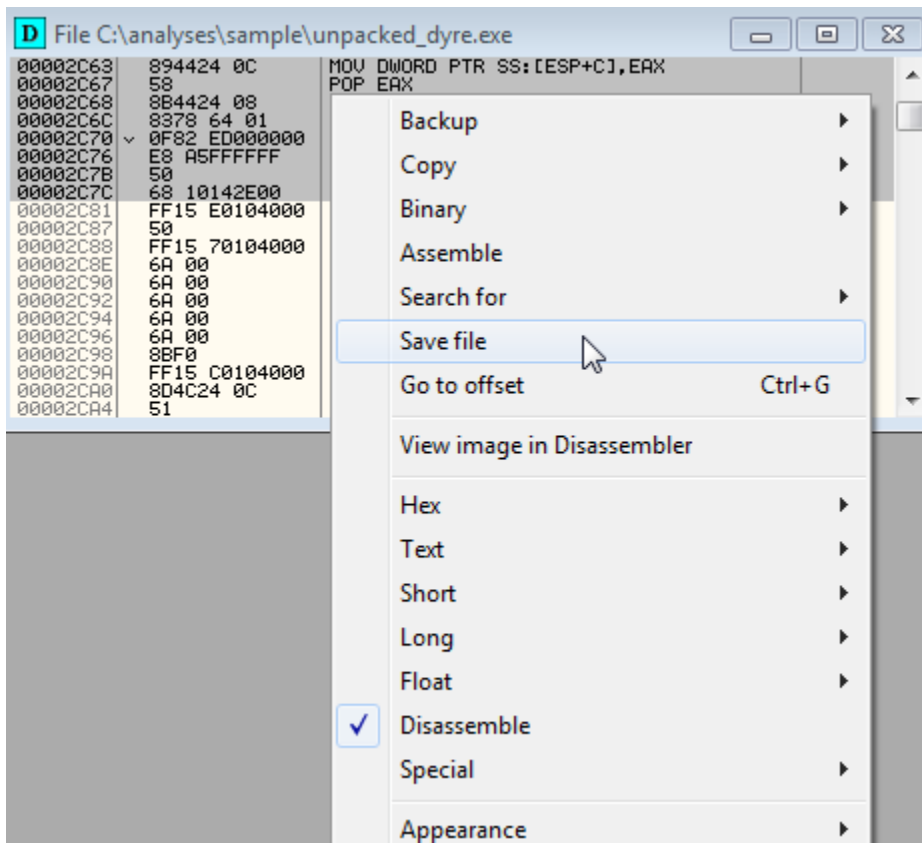


Select modified commands and from the context menu choose *Copy to executable -> All modifications* and then *Copy All* in the dialog window.

³⁶PEB-Process-Environment-Block <http://www.aldeid.com/wiki/PEB-Process-Environment-Block> (last accessed 11.09.2015)



In the new window from the context menu choose *Save file* and save the patched executable.



Now try running the patched executable while observing its behaviour in Process Explorer or Process Monitor (process tree).

Explorer.EXE (1036)	Windows Explorer	Microsoft Corporat...	7/3/2015 5:05:48 PM	n/a
VBoxTray.exe (1876)	VirtualBox Guest ...	Oracle Corporation	7/3/2015 5:05:49 PM	n/a
Greenshot.exe (1900)	Greenshot	Greenshot	7/3/2015 5:05:49 PM	n/a
OLLYDBG.EXE (1648)	OllyDbg. 32-bit an...		7/3/2015 5:17:07 PM	n/a
unpacked_dyre.exe (2576)			7/3/2015 5:18:08 PM	n/a
procexp.exe (2660)	Sysinternals Proce...	Sysinternals - ww...	7/3/2015 5:20:49 PM	n/a
Procmon.exe (3652)	Process Monitor	Sysinternals - ww...	7/3/2015 5:21:06 PM	n/a
unpacked_dyre_fix.exe (3492)			7/3/2015 5:21:15 PM	7/3/2015 5:21:16 PM
WYTIByatnlkEqDW.exe (3936)			7/3/2015 5:21:16 PM	7/3/2015 5:21:26 PM

If everything is done correctly you should see that the Dyre process is creating a new child process which uses significantly more time. This means that you have successfully patched the Dyre executable and likely no other anti-analysis check is preventing it from running anymore.

5. Process creation and injection

5.1 Process injection and process hollowing

Modern malware is frequently using some form of code injection into other processes. Whichever technique is used, the goal is almost always the same: to disguise the malicious code and to make the analysis more difficult.

Process replacement – also known as process hollowing³⁷, is a technique in which the process image in the memory is replaced with a new image containing malicious code. Usually the new process is created in suspended state using some legitimate binary (e.g. `explorer.exe` or `notepad.exe`). Then its memory is overwritten with the malicious code, a new entry point is set and the process is resumed. This way the user will see only well-known processes and if a malicious process is found it won't point to the initial malicious file but to the legitimate binary from which it was created.

A typical process hollowing scheme is executed as follow:

1. Creation of a new process in the suspended state (*CreateProcess*, *CreateProcessInternal*)
2. Unmapping a new process's image from the memory (*NtUnmapViewOfSection*)
3. Allocating memory in the new process (*VirtualAllocEx*)
4. Writing malicious code to the newly allocated memory (*WriteProcessMemory*)
5. Setting a new entry point address of the main thread in the hollowed process (*GetThreadContext*, *SetThreadContext*)
6. Resuming the main thread of the hollowed process (*ResumeThread*)

Process injection - in this technique the malicious code is injected into an already running process as a separate thread. This way, while the original process is still doing its work, the malicious code will be running at the same time in a separate thread. Most often the malware injects its code into the `explorer.exe` process, which is always running in Windows systems and will hardly ever be terminated by the user.

There are a couple of methods³⁸ how to inject code into other processes. One of the most frequently used methods is using the *WriteProcessMemory* and *CreateRemoteThread* functions:

1. Malicious code iterates over the process list to find a process to which it will be injected (*CreateToolhelp32Snapshot*, *Process32First*, *Process32Next*)
2. Opening a destination process handle (*OpenProcess*)
3. Allocating memory in the selected process address space (*VirtualAllocEx*)
4. Writing malicious code to the newly allocated memory (*WriteProcessMemory*)
5. Creating a remote thread in the chosen process (*CreateRemoteThread*)

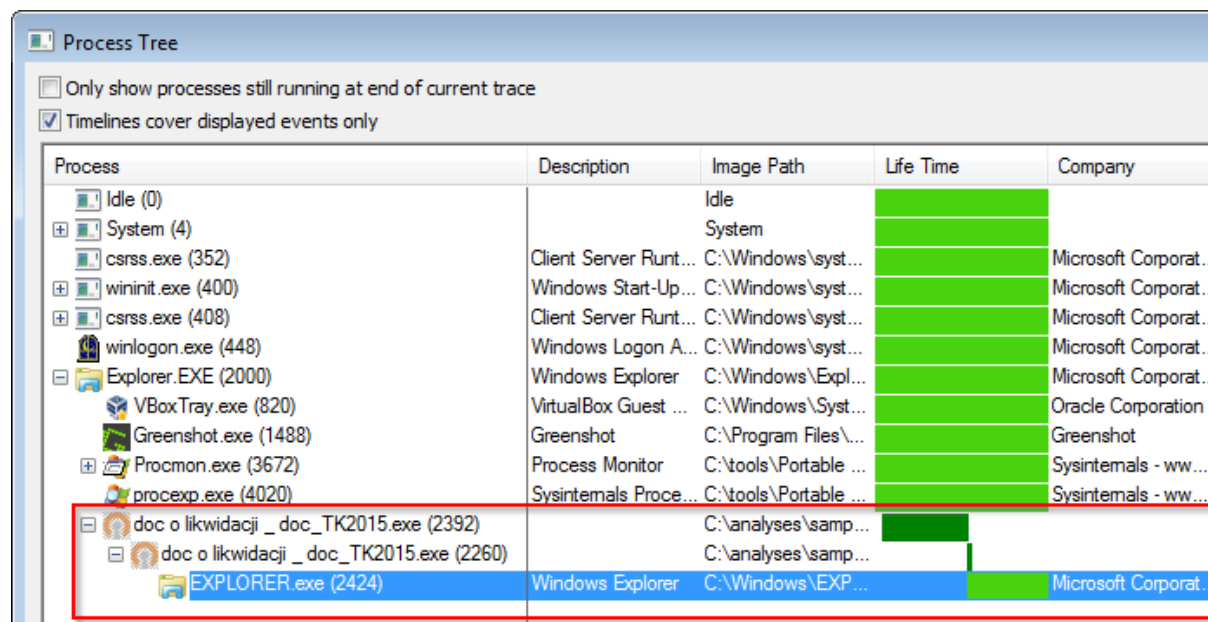
5.2 Following child processes of Tinba banking trojan

In this exercise you will follow child (hollowed) processes created by the Tinba loader till you reach the main Tinba payload. The sample to be analysed is a file "`doc o likwidacji _ doc_TK2015.exe`" which was sent to users during the malware campaign in 2015.

³⁷Process Hollowing <http://www.autosectools.com/process-hollowing.pdf> (last accessed 11.09.2015)

³⁸Three Ways to Inject Your Code into Another Process <http://www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Proces> (last accessed 11.09.2015)

From the results of the behavioural analysis it is known that the Tinba loader is creating two child processes shortly after execution. The first one is a copy of the original loader executable while the second one is the *EXPLORER.exe* process. Please note that in contrast to some other malware, this Tinba variant isn't injecting code into the existing *explorer.exe* instance but is creating a new instance of said process.

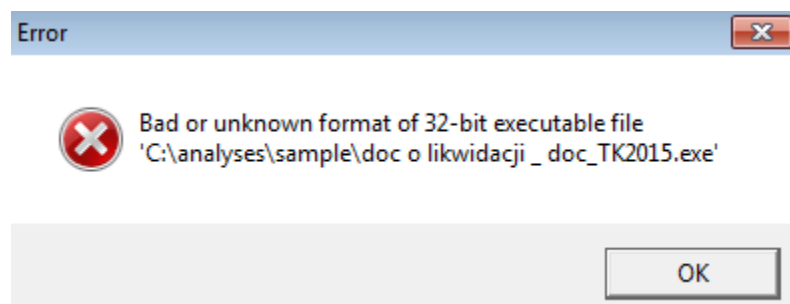


Process	Description	Image Path	Life Time	Company
Idle (0)	Idle			
System (4)	System			
csrss.exe (352)	Client Server Runt...	C:\Windows\sys...		Microsoft Corporat...
wininit.exe (400)	Windows Start-Up...	C:\Windows\sys...		Microsoft Corporat...
csrss.exe (408)	Client Server Runt...	C:\Windows\sys...		Microsoft Corporat...
winlogon.exe (448)	Windows Logon A...	C:\Windows\sys...		Microsoft Corporat...
Explorer.EXE (2000)	Windows Explorer	C:\Windows\Expl...		Microsoft Corporat...
VBoxTray.exe (820)	VirtualBox Guest ...	C:\Windows\Syst...		Oracle Corporation
Greenshot.exe (1488)	Greenshot	C:\Program Files\...		Greenshot
Procmon.exe (3672)	Process Monitor	C:\tools\Portable ...		Sysintemals - ww...
procexp.exe (4020)	Sysintemals Proce...	C:\tools\Portable ...		Sysintemals - ww...
doc o likwidacji _ doc_TK2015.exe (2392)		C:\analyses\samp...		
doc o likwidacji _ doc_TK2015.exe (2260)		C:\analyses\samp...		
EXPLORER.exe (2424)	Windows Explorer	C:\Windows\EXP...		Microsoft Corporat...

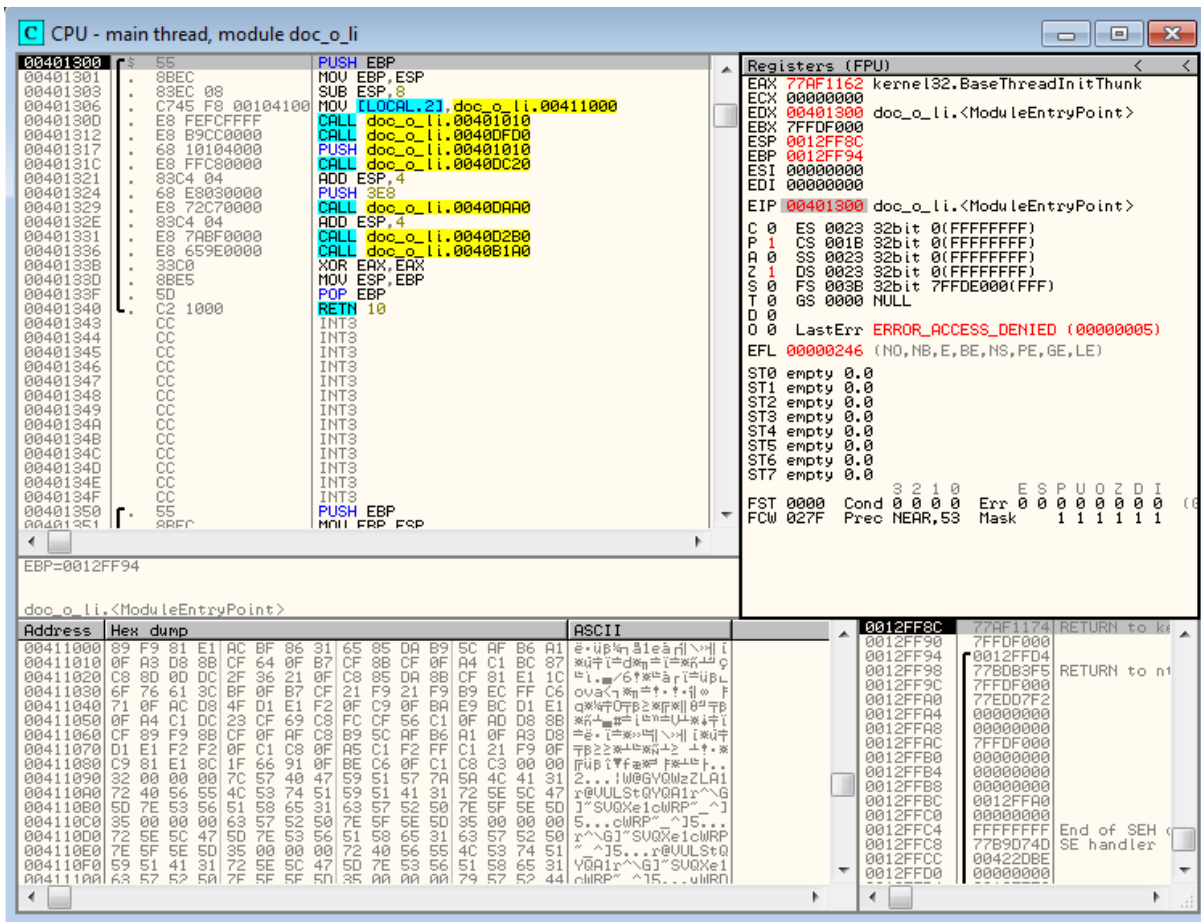
5.2.1 First stage

In the first stage Tinba loader follows the typical process hollowing scheme. That is, it first creates a new process from its own executable (suspended). Then it unmaps memory sections and creates new sections with the unpacked code. Following it sets a new entry point address and resumes the process.

First open OllyDbg and load the malware sample "*doc o likwidacji _ doc_TK2015.exe*". Ignore the error about bad format of the executable.



Now you should land in the entry point at *0x401300*.



The screenshot shows a debugger window titled "CPU - main thread, module doc_o_li". The main window is divided into several panes:

- Assembly View:** Shows assembly instructions starting from address 00401300. Key instructions include:
 - 00401300: 55 PUSH EBP
 - 00401301: 8BEC MOV EBP, ESP
 - 00401302: 83EC 08 SUB ESP, 8
 - 00401306: C745 F8 00104100 MOV LOCAL:23, doc_o_li.00411000
 - 0040130D: E8 FEFCFFFF CALL doc_o_li.00401010
 - 00401312: E8 B9CC0000 CALL doc_o_li.00401010
 - 00401317: 68 10104000 PUSH doc_o_li.00401010
 - 0040131C: E8 FFC80000 CALL doc_o_li.00401020
 - 00401321: 83C4 04 ADD ESP, 4
 - 00401324: 68 E8030000 PUSH 3E8
 - 00401329: E8 72C70000 CALL doc_o_li.0040DAA0
 - 0040132E: 83C4 04 ADD ESP, 4
 - 00401331: E8 7ABF0000 CALL doc_o_li.0040D2B0
 - 00401336: E8 659E0000 CALL doc_o_li.0040B1A0
 - 0040133B: 33C0 XOR EAX, EAX
 - 0040133D: 8BE5 MOV ESP, EBP
 - 0040133F: 5D POP EBP
 - 00401340: C2 1000 RETN 10
- Registers (FPU):** Shows the state of CPU registers. EAX is 77AF1162 (kernel32.BaseThreadInitThunk), EDX is 00401300 (doc_o_li.<ModuleEntryPoint>), and EIP is 00401300 (doc_o_li.<ModuleEntryPoint>). The LastErr register shows ERROR_ACCESS_DENIED (00000005).
- Memory Dump:** Shows a hex dump of memory starting at address 00411000. The dump contains various characters and symbols, including "e-u0k31e0d\|i" and "r@UULSt0VAlr^G".

Insert breakpoints on the following functions:

- CreateProcessW
- SetThreadContext
- WriteProcessMemory
- ResumeThread

Resume the process execution (F9). After a short while you should land at the *CreateProcessW* breakpoint. As you can see either in the stack window or the call stack window (Alt+K) a new process is created in suspended state (*CREATE_SUSPENDED*) and is created using the original executable image.

Step over (F8) a few times till you go past return and land back in the loader code. You should land on the instruction **TEST EAX, EAX**.

0045EF1E	8B45 0C	MOV EAX, DWORD PTR SS:[EBP+C]	doc_o_li.0041006C
0045EF21	50	PUSH EAX	
0045EF22	8B45 08	MOV EAX, DWORD PTR SS:[EBP+8]	
0045EF25	50	PUSH EAX	
0045EF26	FF55 C8	CALL DWORD PTR SS:[EBP-38]	kernel32.CreateProcessW
0045EF29	85C0	TEST EAX, EAX	
0045EF2B	0F84 A1030000	JE 0045F2D2	
0045EF31	6A 04	PUSH 4	
0045EF33	68 00100000	PUSH 1000	
0045EF38	68 00020000	PUSH 200	

Scroll down in the assembly code and you should see calls to functions such as *GetThreadContext*, *ReadProcessMemory*, *ZwUnmapViewOfSection*, *VirtualAllocEx*, *WriteProcessMemory*, *SetThreadContext*, *ResumeThread*. This is typical for the process hollowing technique.

Now you could step over the code (F8) and follow how exactly the process hollowing is taking place. Sometimes this would be necessary, especially when the malware uses some anti-debugging techniques or some other nonstandard approaches. In this case it is enough to just follow previously set breakpoints on *SetThreadContext*, *WriteProcessMemory* and *ResumeThread*.

Resume the execution (F9). The execution should break on *WriteProcessMemory*. Take a look at the arguments passed via stack to the *WriteProcessMemory* function.

```

0012FBF0 0045F263 CALL to WriteProcessMemory from 0045F260
0012FBF4 0000003C hProcess = 0000003C (window)
0012FBF8 00400000 Address = 400000
0012FBFC 02130000 Buffer = 02130000
0012FC00 00004600 BytesToWrite = 4600 (17920.)
0012FC04 0012FD60 pBytesWritten = 0012FD60
0012FC08 00000000
0012FC0C 00000000
0012FC10 7FFDF000
0012FC14 00000000
0012FC18 77BD316F RETURN to ntdll.77BD316F from ntdll.77BC6BF0
0012FC1C 00000044
0012FC20 00000000
0012FC24 00000000
0012FC28 00000000
0012FC2C 00000000
0012FC30 00000000

```

You see that the loader overrides 17920 bytes at the address *0x400000* of the previously created child process. If you follow in the dump source buffer (*0x2130000*³⁹) you will see typical PE headers with likely some unpacked code.

Address	Hex dump	ASCII
02130000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZÉ.♦...♦... ..
02130010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	7.....@.....
02130020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130030	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00C.....
02130040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	#\ \$.+.=?@L=?Th
02130050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program cannot
02130060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	be run in DOS
02130070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$.U.....
02130080	50 45 00 00 4C 01 01 00 E0 AE 0D 55 00 00 00 00	PE..L00.αα.U.....
02130090	00 00 00 00 E0 00 0F 01 0B 01 01 47 00 36 00 00	...α.*0000G.6...
021300A0	00 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00▶.....▶...
021300B0	00 00 00 00 00 00 40 00 00 00 10 00 00 00 02 00 00@.....▶.....▶...
021300C0	01 00 00 00 00 00 00 00 05 00 00 00 00 00 00 00	0.....▶.....▶...
021300D0	00 46 00 00 00 02 00 00 1C FB 00 00 02 00 00 00	.F...@...Lr...@...
021300E0	00 10 00 00 00 10 00 00 00 00 01 00 00 00 00 00	▶.....▶.....@.....
021300F0	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00▶.....▶.....
02130100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02130170	00 00 00 00 00 00 00 00 2E 66 6C 61 74 00 00 00flat...
02130180	00 36 00 00 00 10 00 00 00 36 00 00 00 02 00 00	.6.....▶.....6.....@...
02130190	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60
021301A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
021301B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
021301C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
021301D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

At this point you could decide to dump a new PE image to disk for later analysis but skip this step and start debugging the child process.

Resume the execution (F9) two times until you break on *SetThreadContext* function. This function is used by the malware to set a new entry point address of the initial thread of the suspended process (before it will be resumed). Write down the address of the context structure put on the stack (*0x420000 - pContext*) and follow it in the dump.

```

0012FBFC 0045F2A4 CALL to SetThreadContext from 0045F2A1
0012FC00 00000038 hThread = 00000038 (window)
0012FC04 00420000 pContext = 00420000
0012FC08 00000000
0012FC0C 00000000
0012FC10 7FFDF000
0012FC14 00000000
0012FC18 77BD316F RETURN to ntdll.77BD316F from ntdll.77BC6BF0

```

³⁹ This address might be different.

The entry point of the newly created process is stored in its EAX register⁴⁰. Its value can be read from the context structure at the address `pContext+0xB0`⁴¹. In this case the entry point address is `0x00401000` (remember about little-endian notation). Write down the address of the entry point, it will be needed later.

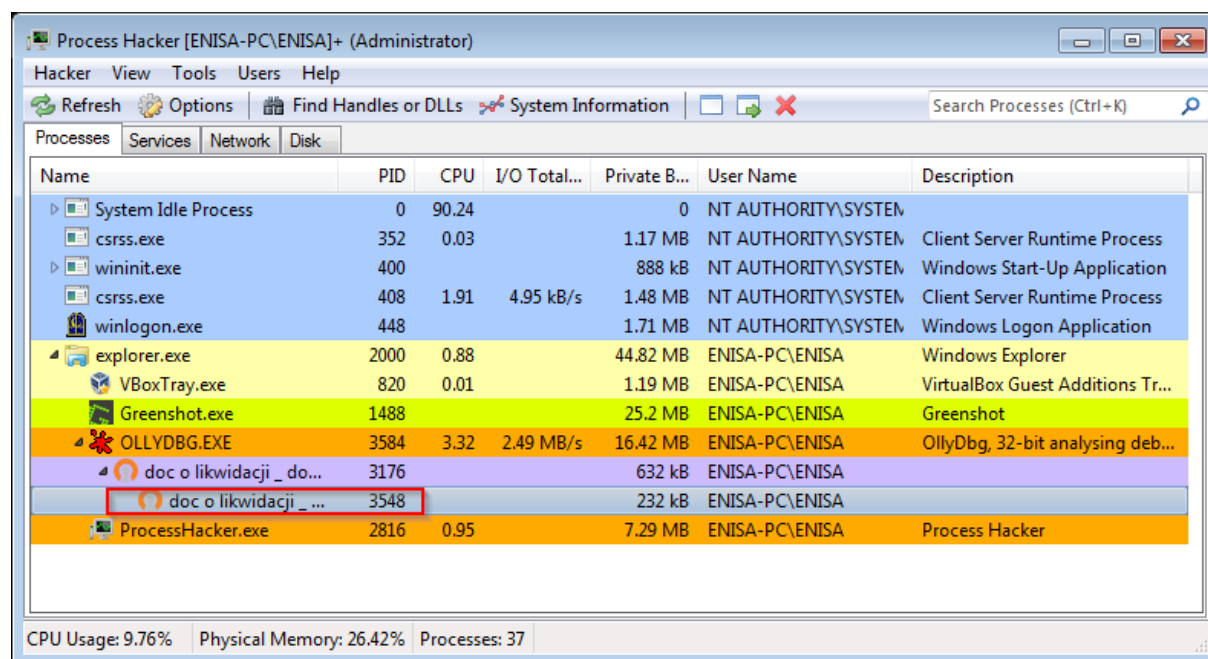
Address	Hex dump	ASCII
00420000	07 00 01 00 00 00 00 00 00 00 00 00 00 00 00	.0.....
00420010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00420090	3B 00 00 00 23 00 00 00 23 00 00 00 00 00 00	...#...#...
004200A0	00 00 00 00 00 F0 FD 7F 00 00 00 00 00 00 00	...=^Δ...
004200B0	00 10 40 00 00 00 00 00 08 64 BC 77 1E 00 00 00	...cµw+...
004200C0	00 02 00 00 F0 FF 12 00 23 00 00 00 00 00 00 00	.0.=+.#.....
004200D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004200E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Resume the execution (F9) until you land at the breakpoint on *ResumeThread*. If you had stepped over this function, the child process would be resumed and you would miss the chance to follow its code. You also can't just attach OllyDbg to the child process because OllyDbg doesn't allow to be attached to suspended processes.

To cope with this problem you will use a simple trick. You will override the first two bytes at the entry point of the child process with `0xEBFE`. This opcode translates to a JMP instruction to itself. This way after resuming the process the initial thread will be stuck in the endless loop giving us a chance to attach OllyDbg to the child process.

To override the child process memory you can use Process Hacker⁴² tool.

Open Process Hacker and find the suspended child process. Right-click on it and open *Properties* window.

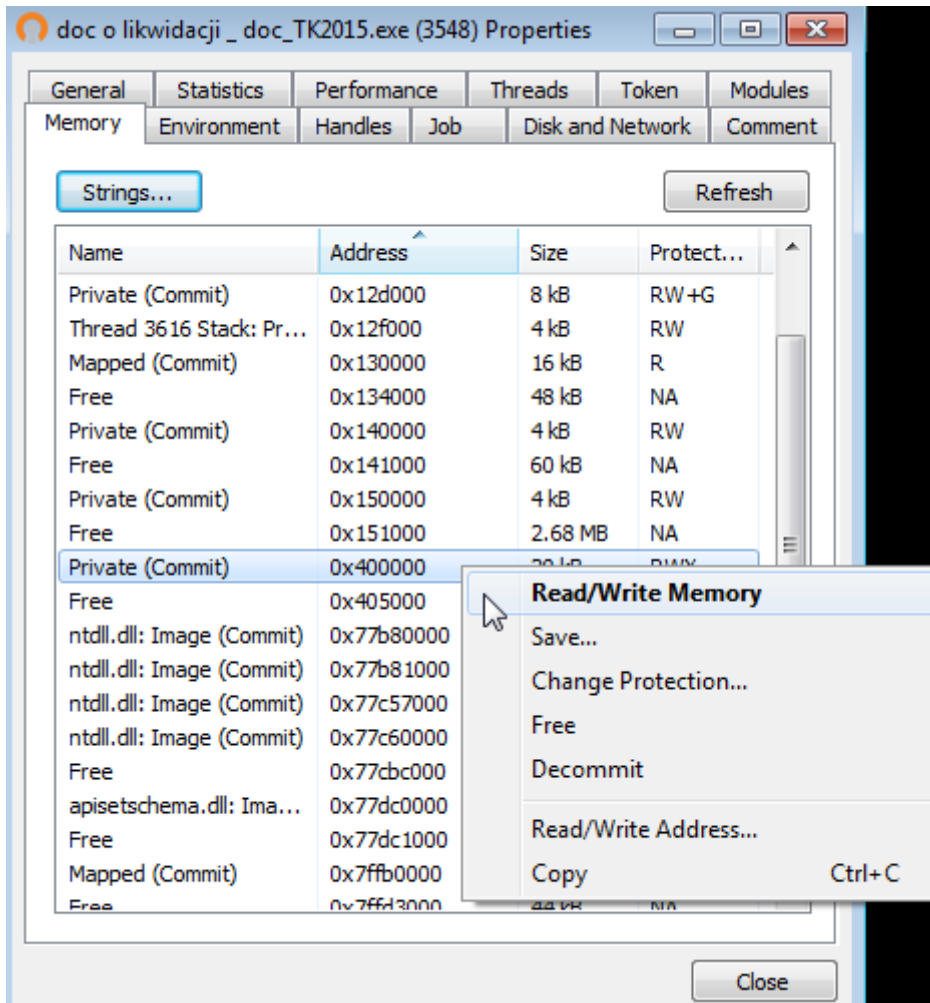


⁴⁰ EAX value in the context of newly created process, don't mistake it with the EAX value in OllyDbg which is the value of EAX register in the context of currently debugged process.

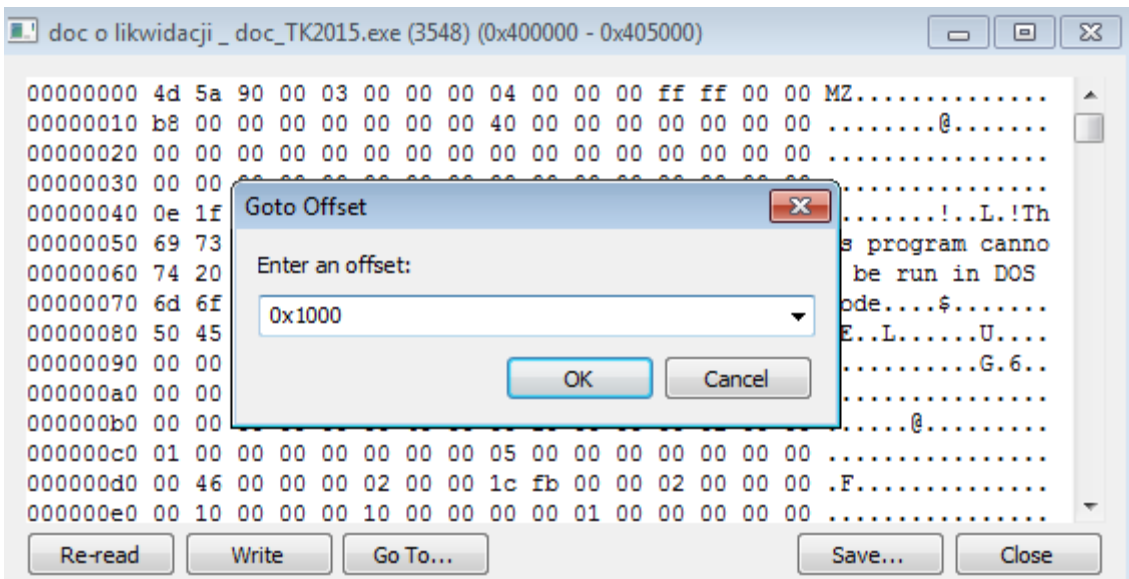
⁴¹ struct CONTEXT http://www.nirsoft.net/kernel_struct/vista/CONTEXT.html (last accessed 11.09.2015)

⁴² Process Hacker <http://processhacker.sourceforge.net/> (last accessed 11.09.2015)

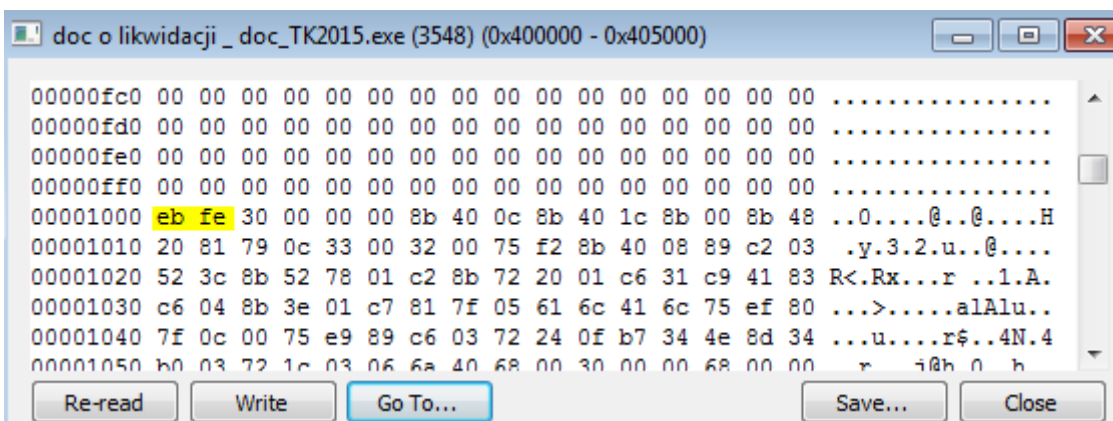
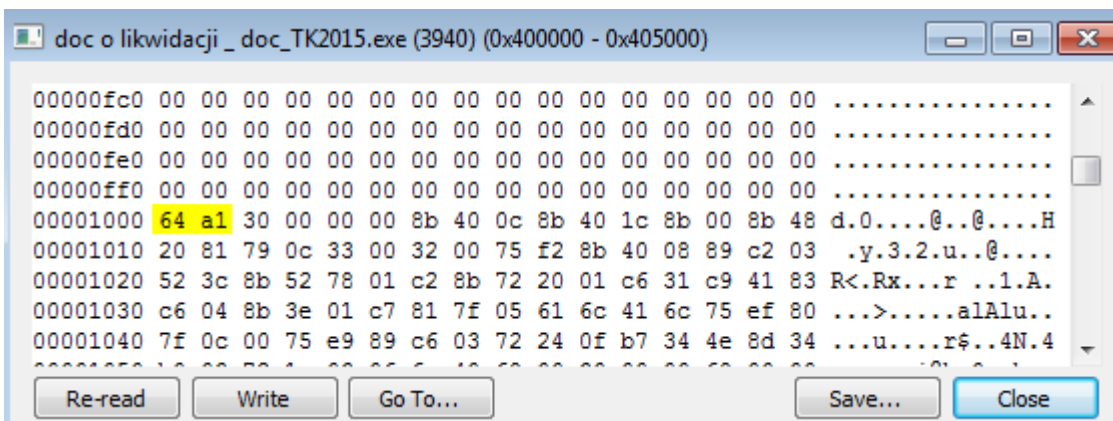
In the *Properties* window switch to *Memory* tab and find a memory block where the entry point is located (0x401000 -> *memory block 0x400000*). Right-click on it and choose *Read/Write Memory* option.



In the new window go to the entry point address at offset 0x1000 (addresses are relative to 0x400000).



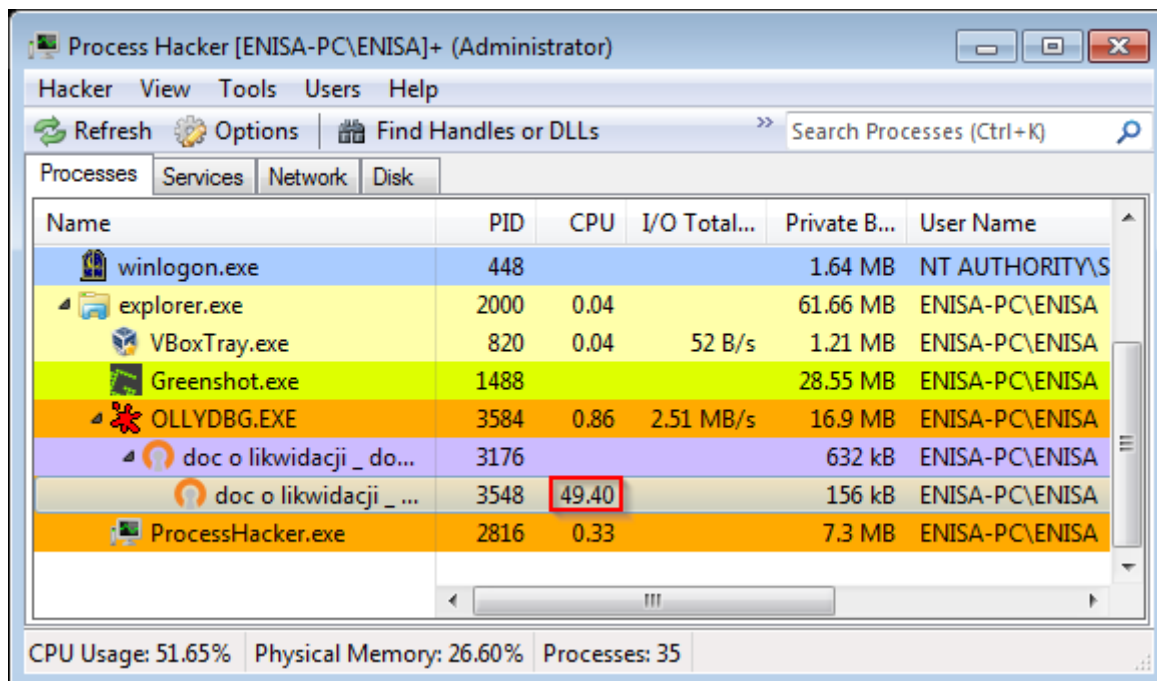
Write down the first two bytes at this offset (*0x64A1*) and override them with (*0xEBFE*). Click *Write* and close the window.



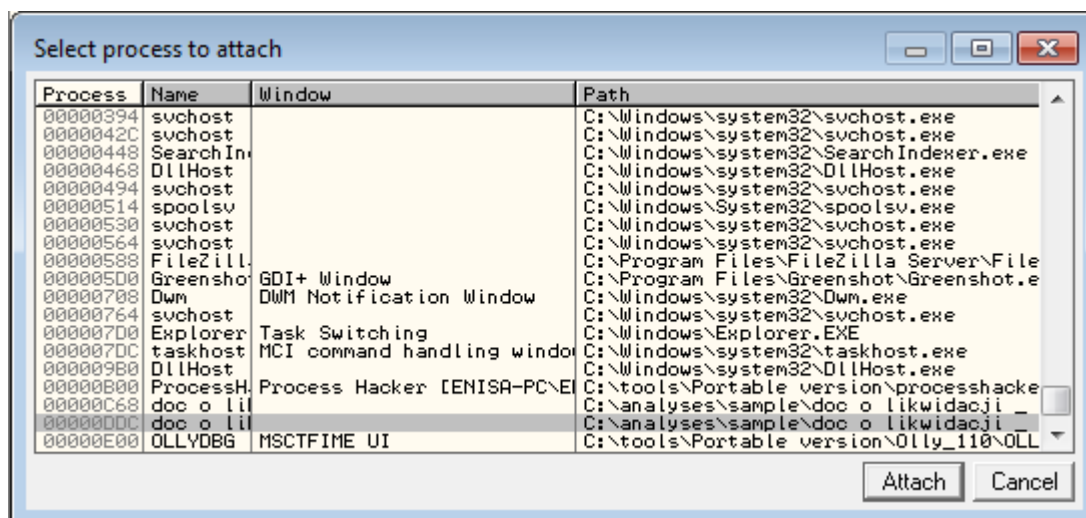
Now switch back to OllyDbg and step over (F8) till return from *ResumeThread* function.

75F8C3C7	90	NOP	
75F8C3C8	90	NOP	
75F8C3C9	8BFF	MOV EDI,EDI	ResumeThread
75F8C3CB	55	PUSH EBP	
75F8C3CC	8BEC	MOV EBP,ESP	
75F8C3CE	8D45 08	LEA EAX,DWORD PTR SS:[EBP+8]	
75F8C3D1	50	PUSH EAX	
75F8C3D2	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
75F8C3D5	FF15 4413F875	CALL DWORD PTR DS:[<&ntdll.NtResumeThread	ntdll.ZwResumeThread
75F8C3D8	85C0	TEST EAX,EAX	
75F8C3DD	0F8C E4CF0100	JL KERNELBA.75FA93C7	
75F8C3E3	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]	
75F8C3E6	5D	POP EBP	0050F2AE
75F8C3E7	C2 0400	RETN 4	
75F8C3EA	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]	

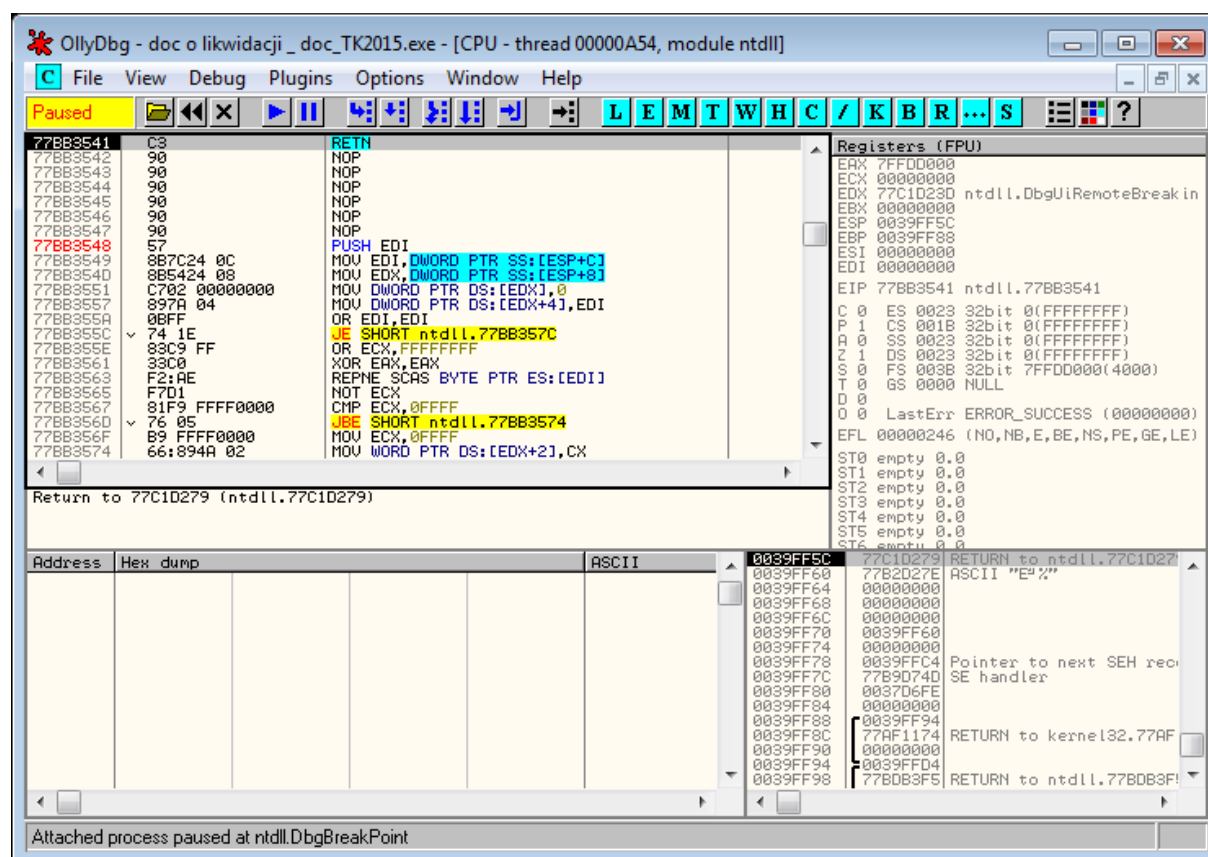
Now you can minimize OllyDbg. In the Process Hacker window you can also notice that the child process was resumed and is now using a considerable amount of CPU time. This is the result of the endless loop you created in this process. Note the process identifier (PID) of the child process (in decimal).



Start a new OllyDbg instance and attach it to the child process (*File->Attach*). Note that OllyDbg presents process PIDs in the hexadecimal format. If you are unsure which of “doc o likwidacji...” entries is the child process, just try attaching to both of them. Since the parent process is already being debugged it will be possible to attach to only one process – the real child process.



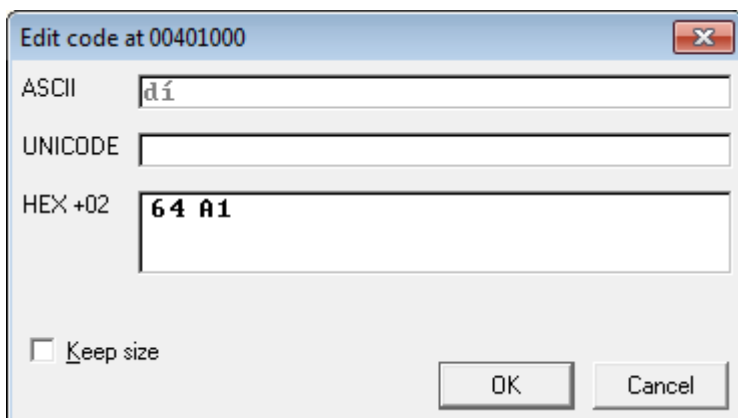
Ignore the error message (the same as previously). You should land somewhere in *ntdll*.



In the assembly window go to the address *0x401000* (EP). You should see the previously injected *0xEBFE* bytes. Put a breakpoint on this instruction and resume the process (F9).

Address	Disassembly	Comment
00401000	EB FE	JMP SHORT doc_o_li.00401000
00401002	3000	XOR BYTE PTR DS:[EAX],AL
00401004	0000	ADD BYTE PTR DS:[EAX],AL
00401006	8B40 0C	MOV EAX,DWORD PTR DS:[EAX+C]
00401009	8B40 1C	MOV EAX,DWORD PTR DS:[EAX+1C]
0040100C	8B00	MOV EAX,DWORD PTR DS:[EAX]
0040100E	8B48 20	MOV ECX,DWORD PTR DS:[EAX+20]
00401011	8179 0C 33003200	CMP DWORD PTR DS:[ECX+C],320033
00401018	75 F2	JNZ SHORT doc_o_li.00401000

After you land at the breakpoint select *JMP* instruction and press Ctrl+E to edit it. Replace *EB FE* bytes with the original *64 A1*.



After confirmation OllyDbg will automatically reanalyse the code, changing it significantly.

Address	Disassembly	Comment
00401000	64:A1 30000000	MOV EAX,DWORD PTR FS:[30]
00401006	8B40 0C	MOV EAX,DWORD PTR DS:[EAX+C]
00401009	8B40 1C	MOV EAX,DWORD PTR DS:[EAX+1C]
0040100C	8B00	MOV EAX,DWORD PTR DS:[EAX]
0040100E	8B48 20	MOV ECX,DWORD PTR DS:[EAX+20]
00401011	8179 0C 33003200	CMP DWORD PTR DS:[ECX+C],320033
00401018	75 F2	JNZ SHORT doc_o_li.00401000
0040101A	8B40 08	MOV EAX,DWORD PTR DS:[EAX+8]
0040101D	89C2	MOV EDX,EAX
0040101F	0352 3C	ADD EDX,DWORD PTR DS:[EDX+3C]
00401022	8B52 78	MOV EDX,DWORD PTR DS:[EDX+78]

Now you are in the entry point of the second stage.

5.2.2 Second stage

Second stage loader creates a new instance of the *EXPLORER.exe* process and injects the malicious code into it. But instead of entirely overriding the *EXPLORER.exe* code it also uses the file mapping mechanism to share a portion of its code with the new process.

First when still paused at the entry point of the second stage, create a snapshot of the virtual machine (name it *'Tinba – second stage'*). In case of anything going wrong you wouldn't need to repeat the entire process.

Next put breakpoints on the following functions:

- *CreateProcessInternalW*
- *GetThreadContext*
- *SetThreadContext*
- *WriteProcessMemory*
- *ResumeThread*

Resume the execution (F9). Shortly you should land at the *CreateProcessInternalW* call. Right-click on the assembly code and select 'Analyze this!' (while using the OllyDbg plugin). Next open the *Call stack* window (*View -> Call stack*, Alt+K).

Address	Stack	Procedure / arguments	Called from	Frame
0012F914	77AFF699	? kernel32.CreateProcessInternalW	kernel32.77AFF694	
0012F9F4	77AA208E	? kernel32.CreateProcessInternalW	kernel32.CreateProcessA+27	
0012FA2C	00410E91	kernel32.CreateProcessA	00410E8B	0012FA28
0012FA30	00000000	ModuleFileName = NULL		
0012FA34	0012FE84	CommandLine = "EXPLORER"		
0012FA38	00000000	pProcessSecurity = NULL		
0012FA3C	00000000	pThreadSecurity = NULL		
0012FA40	00000000	InheritHandles = FALSE		
0012FA44	00000004	CreationFlags = CREATE_SUSPENDED		
0012FA48	00000000	pEnvironment = NULL		
0012FA4C	00000000	CurrentDir = NULL		
0012FA50	0012FA58	pStartupInfo = 0012FA58		
0012FA54	0012FA9C	pProcessInfo = 0012FA9C		
0012FF88	0041001B	00410DE8	00410016	0012FF84

As you can see *CreateProcessInternalW* was indirectly called as a result of a call to *CreateProcessA*. As before, a new process is created in a suspended state, but this time *explorer.exe* is used as a source image for the new process. Such usage of a well-known system process is a typical malware deception mechanism.

Continue the execution (F9) till you land at the *GetThreadContext* breakpoint. In this case the malware uses this function to check the address of an entry point of the *EXPLORER.exe* process.

The screenshot displays the OllyDbg interface with the CPU window showing assembly instructions and the registers window showing the current state of the CPU registers. The registers window shows EAX: 00000001, ECX: 77AFF680, EDX: 00280174, EBP: 0000F000, ESP: 0012FA4C, ESI: 77B556E8, EDI: 004104E3, and EIP: 77B0962F. The disassembly window shows the instruction CALL DWORD PTR DS:[&ntdll.NtGetThreadContextThrea at address 00410E8B.

Note the address of the *pContext* structure (in this example it is *0x12FAAC*) and follow it at the dump. Next step over (F8) till return from *GetThreadContext* and read *EXPLORER.exe* entry point from the *pContext+0xB0* address. Alternatively you can just find *explorer.exe* executable on the disk and check its entry point address with some PE editor (e.g. CFF Explorer). In this situation EP is located at the address *0x4AA8DF*.

Address	Hex dump	ASCII
0012FAAC	07 00 01 00 98 FA 12 00 50 01 28 00 3C FB 12 00	·.0.0.+.P0(.<J+.+
0012FABC	80 01 28 00 DE 33 4D 00 4E 00 00 00 00 FE 8D 77	00(. 3M.N...#w
0012FACC	50 01 28 00 7F 00 00 00 80 41 28 00 00 E0 FD 7F	P0(.0...A(.x20
0012FADC	7F 00 00 00 00 00 00 00 45 00 00 00 CC FA 12 00	0...E...f+.+
0012FAEC	48 26 28 00 98 FA 12 00 00 00 00 00 00 00 00 00	H&(.0.+.+
0012FAFC	04 00 00 04 C8 29 28 00 00 00 00 00 7F BF 8D 77	+.+.)(...0w
0012FB0C	4C FB 12 00 00 00 00 00 7F 00 00 00 3A 33 E3 77	Lr+.+.+.3w
0012FB1C	02 00 00 00 3C FB 12 00 BC 6F 8D 77 20 FC 12 00	0...<J+.0w n+.+
0012FB2C	00 00 00 00 06 00 08 00 20 FC 12 00 00 00 00 00	...+.0. n+.+
0012FB3C	3B 00 00 00 23 00 00 00 23 00 00 00 00 00 00	...#.n+.+
0012FB4C	00 00 00 00 00 F0 FD 7F 00 00 00 00 00 00 00	;...#...#...+
0012FB5C	DF A8 4A 00 00 00 00 00 D8 64 BC 77 1B 00 00 00	■0J...tdw+
0012FB6C	00 02 00 00 FC F9 19 00 23 00 00 02 00 00 00	0...n+.#.0...+
0012FB7C	00 00 00 00 06 00 00 00 00 00 00 00 06 00 00	...+.+.+.+
0012FB8C	46 FC 12 00 D8 23 28 00 06 00 08 02 40 FC 12 00	F n+.+.#(00 n+.+

Resume the execution (F9) till the breakpoint on *WriteProcessMemory*.

Address	Hex dump	Comment
0012FA40	00410F50	CALL to <i>WriteProcessMemory</i> from 00410F4A
0012FA44	0000001C	hProcess = 0000001C (window)
0012FA48	004AA8DF	Address = 4AA8DF
0012FA4C	00411026	Buffer = 00411026
0012FA50	00000100	BytesToWrite = 100 (256.)
0012FA54	00000000	oBytesWritten = NULL
0012FA58	00000044	
0012FA5C	00000000	
0012FA60	00000000	
0012FA64	00000000	
0012FA68	00000000	

Notice that this time only a very small portion of the code (256 bytes) is written to the child process memory (there are also no subsequent calls to *WriteProcessMemory* that could write rest of the malicious code). What's also important is that the code is overridden at the exact address of the previously checked entry point – *0x4AA8DF*. This suggests that the entry point address won't be changed this time (and indeed it isn't).

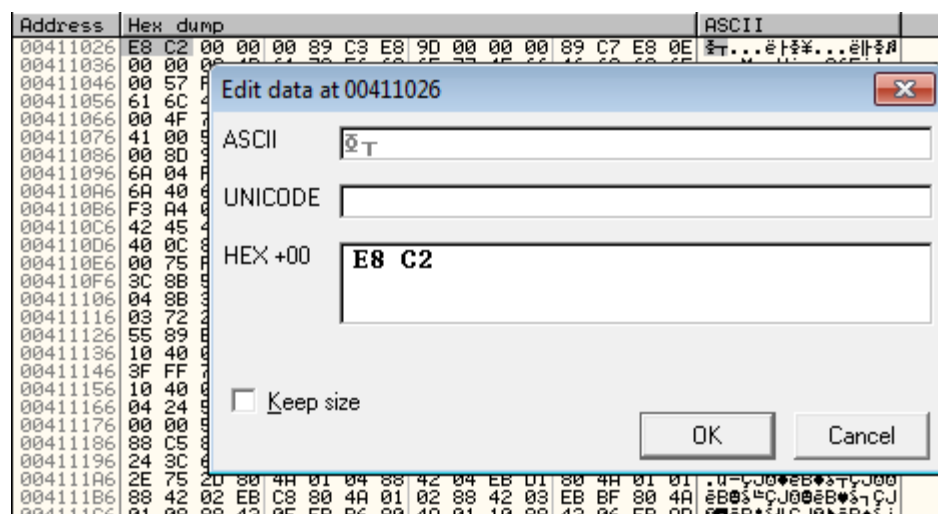
Follow the source buffer in the dump address (in this case *0x411026*).

Address	Hex dump	ASCII
00411026	E8 C2 00 00 00 89 C3 E8 9D 00 00 00 89 C7 E8 0E	0T...0T00...0T00
00411036	00 00 00 4D 61 70 56 69 65 77 4F 66 46 69 6C 65	...MapViewOfFile
00411046	00 57 FF D3 89 C6 E8 0D 00 00 00 56 69 72 74 75	W 0F0...Virtu
00411056	61 6C 41 6C 6C 6F 63 00 57 FF D3 97 E8 11 00 00	alAlloc.W 0004..
00411066	00 4F 70 65 6E 46 69 6C 65 40 61 70 70 69 6E 67	.OpenFileMapping
00411076	41 00 50 FF D3 E8 00 00 00 00 58 81 EB 80 20 40	A.P 00...[000 @
00411086	00 8D 93 C3 20 40 00 88 9B BF 20 40 00 52 6A 00	.i0t 0.i0t 0.Rj.
00411096	6A 04 FF D0 53 6A 00 6A 00 6A 04 50 FF D6 89 C6	J+ 0Sj.jj0P mef
004110A6	6A 40 68 00 30 00 53 6A 00 FF D7 89 C7 89 D9	j0h.0..Sj. 0000
004110B6	F3 A4 05 FD 0F 00 00 50 C3 6E 5F 00 00 41 43 45	0000...Ptn...ACE
004110C6	42 45 43 43 42 4D 45 4D 00 64 A1 30 00 00 00 8B	BECCBMEM.dio...i
004110D6	40 0C 88 40 1C 8E 00 88 48 20 81 79 0C 33 00 32	0.i0i.iH 0y.3.2
004110E6	00 75 F2 88 40 08 C3 E8 D0 FF FF FF 89 C2 03 52	.u0i0000 000R
004110F6	3C 88 52 78 01 C2 88 72 20 01 C6 31 C9 41 83 C6	<IR000r 0f1rA0f
00411106	04 8B 3E 01 C7 81 7F 05 6F 63 41 64 75 EF 89 C6	0i>0H0000Adun0f
00411116	03 72 24 0F B7 34 4E 8D 34 B0 03 72 1C 03 06 C3	000004N00000000
00411126	55 89 E5 8B 7D 0C 83 C7 08 57 FF 75 08 FF 93 24	U00i).000W 00 00
00411136	10 40 00 89 47 F8 31 C0 B9 FF 00 00 00 F2 AE 83	00.000100...0000
00411146	3F FF 75 E2 C9 C2 08 00 E8 9A FF FF FF 89 83 24	? u0r00.00 0000
00411156	10 40 00 E8 71 FF FF FF 50 8D 83 2C 10 40 00 87	00.00 P0a,00.00
00411166	04 24 50 E8 B8 FF FF FF C3 60 89 CE 51 E8 00 00	0000000000000000
00411176	00 00 5D 31 C9 30 C0 89 D7 B1 2C F3 AA AC 88 C1	..J1r000000000000
00411186	88 C5 80 E1 FE 80 E5 E7 3C F0 74 20 80 F9 F2 74	0000000000000000
00411196	24 3C 66 74 29 3C 67 74 2E 80 F9 64 74 05 80 FD	0000000000000000
004111A6	2E 75 2D 80 4A 01 04 88 42 04 EB D1 80 4A 01 01	.u-0J000000000000
004111B6	88 42 02 EB C8 80 4A 01 02 88 42 03 EB BF 80 4A	0000000000000000
004111C6	01 08 88 42 05 FF B6 80 4D 01 10 88 42 06 FF 0D	0000000000000000

Notice names of functions such as *MapViewOfFile* and *OpenFileMapping*. This suggests that the rest of the code will be transferred using the file mapping mechanism.

Knowing that the above buffer will be written to the exact address of an entry point, this time you will do *0xEBFE* track before the memory is written to the child process. Please note that this step should be done before stepping over the *WriteProcessMemory* function.

Select the first two bytes of the source buffer (*E8 C2*) and press Ctrl+E. Replace them with bytes *EB FE*.



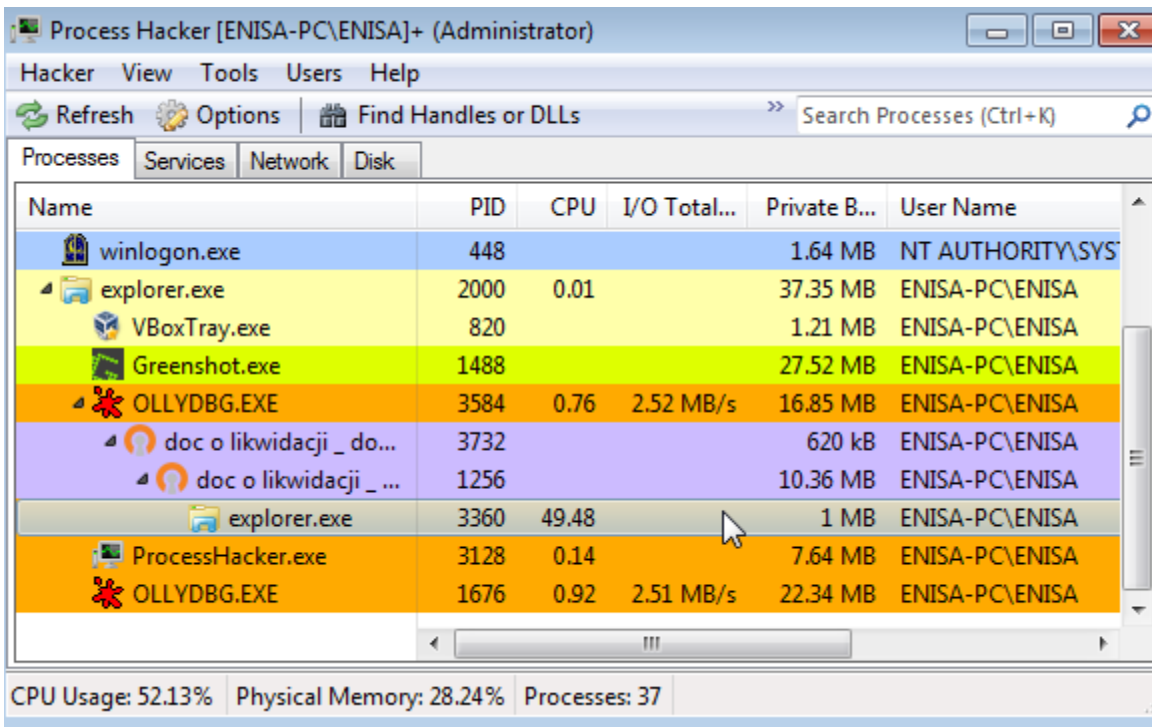
Next step over *WriteProcessMemory* function till the user code. You should land at *TEST EAX, EAX* instruction.

00410F3B	870424	XCHG DWORD PTR SS:[ESP],EAX	
00410F3E	FFB5 08FBFFFF	PUSH DWORD PTR SS:[EBP-428]	
00410F44	FFB5 18FBFFFF	PUSH DWORD PTR SS:[EBP-4E8]	
00410F49	FF93 75134000	CALL DWORD PTR DS:[EBX+401375]	kernel32.WriteProcessMemory
00410F50	85C0	TEST EAX,EAX	
00410F52	0F84 AA000000	JE 00411002	
00410F58	50	PUSH EAX	
00410F59	8D83 C3204000	LEA EAX,DWORD PTR DS:[EBX+4020C3]	
00410F5F	870424	XCHG DWORD PTR SS:[ESP],EAX	
00410F62	FFB5 F4FDFFFF	PUSH DWORD PTR SS:[EBP-20C]	
00410F68	6A 00	PUSH 0	
00410F6A	6A 04	PUSH 4	
00410F6C	6A 00	PUSH 0	
00410F6E	6A FF	PUSH -1	
00410F70	FF93 43104000	CALL DWORD PTR DS:[EBX+401043]	kernel32.CreateFileMappingA
00410F76	85C0	TEST EAX,EAX	
00410F78	0F84 84000000	JE 00411002	
00410F7E	8985 F8FDFFFF	MOV DWORD PTR SS:[EBP-208],EAX	
00410F84	FFB5 F4FDFFFF	PUSH DWORD PTR SS:[EBP-20C]	
00410F8A	6A 00	PUSH 0	
00410F8C	6A 00	PUSH 0	
00410F8E	6A 02	PUSH 2	
00410F90	50	PUSH EAX	
00410F91	FF93 8F104000	CALL DWORD PTR DS:[EBX+40108F]	kernel32.MapViewOfFile
00410F97	85C0	TEST EAX,EAX	
00410F99	74 5B	JE SHORT 00410FF6	
00410F9B	8985 FCFDFFFF	MOV DWORD PTR SS:[EBP-204],EAX	
00410FA1	FFB5 F4FDFFFF	PUSH DWORD PTR SS:[EBP-20C]	
00410FA7	50	PUSH EAX	
00410FA8	8D83 24104000	LEA EAX,DWORD PTR DS:[EBX+401024]	
00410FAE	870424	XCHG DWORD PTR SS:[ESP],EAX	
00410FB1	50	PUSH EAX	
00410FB2	E8 24290000	CALL 0041380B	
00410FB7	FFB5 1CFBFFFF	PUSH DWORD PTR SS:[EBP-4E4]	
00410FB0	FF93 18114000	CALL DWORD PTR DS:[EBX+401118]	kernel32.ResumeThread
00410FC3	B9 14000000	MOV ECX,14	
00410FC8	51	PUSH ECX	
00410FC9	68 E8030000	PUSH 3E8	
00410FCE	FF93 40114000	CALL DWORD PTR DS:[EBX+401140]	kernel32.Sleep
00410FD4	50	PUSH EAX	

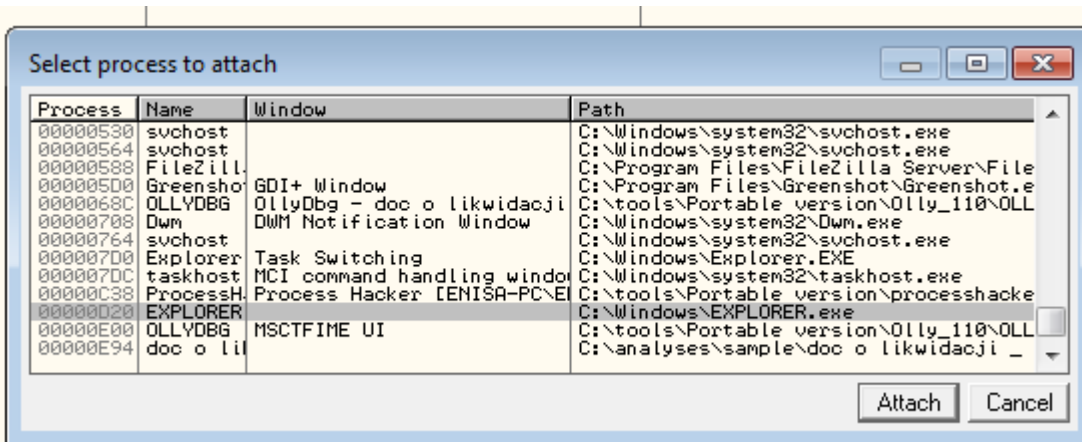
As suspected in the loader there are calls to the *CreateFileMappingA* and the *MapViewOfFile* function, which will be used to share the code with the child process *EXPLORER.exe*. Now you might step over those functions to check their arguments.

Continue the execution (F9) until *ResumeThread* breakpoint. Now since *0xEBFE* trick was already applied you can safely step over (F8) the *ResumeThread* function.

Minimize OllyDbg window and check in Process Hacker if *explorer.exe* process was resumed properly.



Next open a new instance of OllyDbg and attach it to the *EXPLORER.exe* process.



After attaching to *EXPLORER.exe* override *EB EF* bytes at the entry point as described in the previous section (original bytes were *E8 C2*). If you don't remember the address of an entry point you can use *Debug -> Execute till user code* (Alt+K) function.

The screenshot shows a debugger window titled "CPU - main thread, module EXPLORER". The main pane displays assembly instructions with their addresses and hex values. Key instructions include:

- CALL EXPLORER.004AA9A6
- MOV EBX, EAX
- CALL EXPLORER.004AA988
- MOV EDI, EAX
- CALL EXPLORER.004AA900
- DEC EBP
- POPAD
- SHORT EXPLORER.004AA94C
- IMUL ESP, DWORD PTR SS:[EBP+7], 6946664F
- INS BYTE PTR ES:[EDI], DX
- ADD BYTE PTR GS:[EDI-1], DL
- ROR DWORD PTR DS:[ECX+DE8C6], CL
- ADD BYTE PTR DS:[EAX], AL
- PUSH ESI
- IMUL ESI, DWORD PTR DS:[EDX+74], 416C6175
- INS BYTE PTR ES:[EDI], DX
- INS BYTE PTR ES:[EDI], DX
- OUTS DX, DWORD PTR ES:[EDI]
- ARPL WORD PTR DS:[EAX], AX
- PUSH EDI
- CALL EBX
- XCHG EAX, EDI
- CALL EXPLORER.004AA931
- DEC EDI
- SHORT EXPLORER.004AA988
- OUTS DX, BYTE PTR ES:[EDI]
- INC ESI
- IMUL EBP, DWORD PTR SS:[EBP+4D], 69707069
- OUTS DX, BYTE PTR ES:[EDI]
- INC ECX
- ADD BYTE PTR DS:[EAX-1], DL
- SHR EAX, CL
- AND BYTE PTR DS:[EAX], 01

The registers pane on the right shows the state of the FPU registers, with EIP pointing to 004AA8DF. The memory dump pane at the bottom shows hex and ASCII values for addresses starting from 00300000.

Now you are at the entry point of the code injected to *EXPLORER.exe* but this still isn't the main Tinba payload. To reach the payload put a breakpoint on *OpenFileMappingA* and resume the execution (F9).

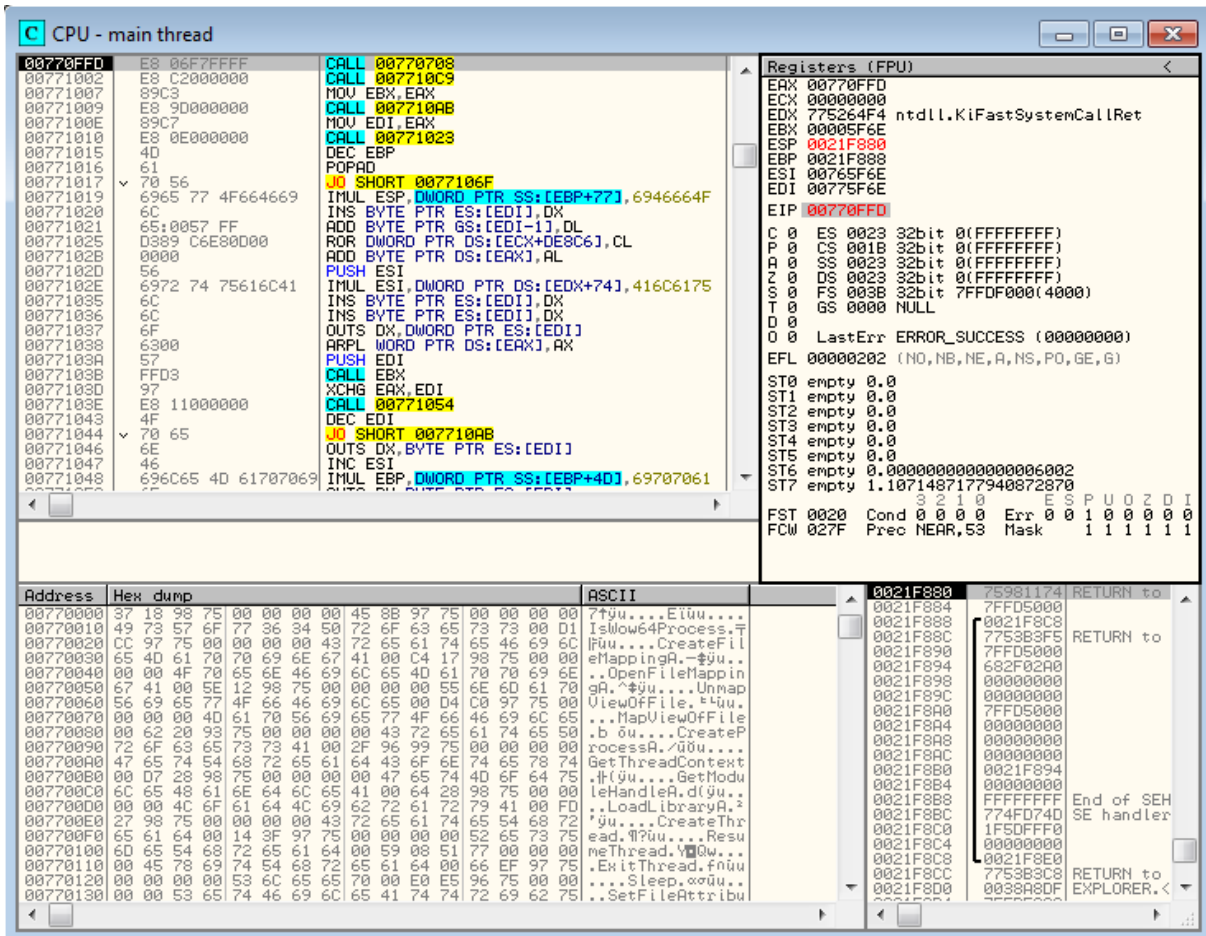
After reaching the *OpenFileMappingA* breakpoint step over (F8) till the user code or choose *Debug->Execute till user code* (Alt+F9). You should land at the *PUSH EBX* instruction.

The screenshot shows a debugger window displaying assembly instructions for the *OpenFileMappingA* function. Key instructions include:

- PUSH 4
- CALL EBX ; call to OpenFileMappingA
- PUSH EBX
- PUSH 0
- PUSH 0
- PUSH 4
- PUSH EAX
- CALL ESI kernel32.MapViewOfFile
- MOV ESI, EAX
- PUSH 40
- PUSH 3000
- PUSH EBX
- PUSH 0
- CALL EDI kernel32.VirtualAlloc
- MOV EDI, EAX
- MOV ECX, EBX
- REP MOVSB BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
- ADD EAX, 0FFD
- PUSH EAX
- RETN

As you see, the malware first opens the file mapping object (*OpenFileMappingA*), then maps the file mapping object into the address space (*MapViewOfFile*), allocates a memory block (*VirtualAlloc*) and finally copies the mapped data to the locally allocated memory block (*REP MOVSB* instruction).

To reach the final payload step over (F8) the return instruction (*RETN*). Don't worry if the address would be different from the one on the screenshot. What's important is that after a return you should see a group of four call instructions.



The screenshot shows a debugger window titled 'CPU - main thread'. It is divided into several panes:

- Instruction List:** Shows assembly instructions with their addresses and hex values. Key instructions include:
 - CALL 00770708
 - CALL 007710C9
 - MOV EBX, EAX
 - CALL 007710AB
 - MOV EDI, EAX
 - CALL 00771023
 - DEC EBP
 - POPAD
 - JN SHORT 0077106F
 - IMUL ESP, DWORD PTR SS:[EBP+77], 6946664F
 - INS BYTE PTR ES:[EDI], DX
 - ADD BYTE PTR GS:[EDI-1], DL
 - ROR DWORD PTR DS:[ECX+DE8C6], CL
 - ADD BYTE PTR DS:[EAX], AL
 - PUSH ESI
 - IMUL ESI, DWORD PTR DS:[EDX+74], 416C6175
 - INS BYTE PTR ES:[EDI], DX
 - INS BYTE PTR ES:[EDI], DX
 - OUTS DX, DWORD PTR ES:[EDI]
 - ARPL WORD PTR DS:[EAX], AX
 - PUSH EDI
 - CALL EBX
 - XCHG EAX, EDI
 - CALL 00771054
 - DEC EDI
 - JN SHORT 007710AB
 - OUTS DX, BYTE PTR ES:[EDI]
 - INC ESI
 - IMUL EBP, DWORD PTR SS:[EBP+40], 69707061
- Registers (FPU):** Shows the state of various registers:
 - EAX: 00770FFD
 - ECX: 00000000
 - EDX: 775264F4 ntdll.KiFastSystemCallRet
 - EBX: 00005F6E
 - ESP: 0021F880
 - EBP: 0021F888
 - ESI: 00765F6E
 - EDI: 00775F6E
 - EIP: 00770FFD
 - Control registers (C, P, A, Z, S, T, D, O) and status registers (EFL, ST0-ST7) are also shown.
- Memory Dump:** Shows a hex dump of memory starting at address 00770000. The ASCII column shows the beginning of a function call: '770u...E!nu...'. The right pane shows a return stack with addresses like 0021F884, 0021F888, etc., and labels like 'RETURN to' and 'End of SEH SE handler'.

Now create a snapshot called 'Tinba'. This snapshot will be used in the later exercises.

6. Introduction to scripting

6.1 Introduction to OllyDbg scripting

When debugging malicious code you sometimes encounter the problem of repetitive and/or tedious tasks. This might be the case when unpacking pieces of code obfuscated with the same packer or performing multiple repetitive actions in some malicious code. One of the solutions to this problem is to automate certain tasks through scripting. In OllyDbg you can do this using the ODbgScript plugin⁴³.

A detailed reference about the scripting language is provided with ODbgScript package in the README.txt file⁴⁴. In general the language is very similar to the assembly language with a few additional commands. Every operation you can do in OllyDbg (except functions provided by other plugins) you can also do in using script.

The list below presents some of the operations you can do with ODbgScript:

- Check and modify registers
- Manipulate the program memory and stack
- Dump memory blocks
- Add breakpoints to the code
- Control program execution (instruction stepping)
- Execute assembly instructions in the context of a debugged program
- Perform arithmetic operations
- Acquire information about instructions and modules
- Search the program memory for specific instructions or patterns

OllyDbg scripts are often used for unpacking binary samples. There are online repositories^{45 46} where you can find scripts dedicated to various packers.

The code below presents an example script which first prints the result of XORing EAX with EDX (without affecting the values in the registers) and then prints the first ten Fibonacci numbers⁴⁷ in a loop.

```
; printing result of EAX^EDX
var result

mov result, eax
xor result, edx
log result

; fibonacci(10)
var i, n, k

mov i, 3
mov n, 1
```

⁴³ ODBGScript <http://sourceforge.net/projects/odbgscript/> (last accessed 11.09.2015)

⁴⁴ ODBGScript <http://sourceforge.net/projects/odbgscript/files/English%20Version/README.txt/view> (last accessed 11.09.2015)

⁴⁵ OllyDbg OllyScripts http://www.openrce.org/downloads/browse/OllyDbg_OllyScripts (last accessed 11.09.2015)

⁴⁶ OllyScript - Scripts <https://tuts4you.com/download.php?list.53> (last accessed 11.09.2015)

⁴⁷ Fibonacci number https://en.wikipedia.org/wiki/Fibonacci_number (last accessed 11.09.2015)

```

mov k, 1

log "1: 1"
log "2: 1"

fibonacci_loop:
  xchg n, k
  add k, n

  eval "{i}: {k}"
  log $RESULT, ""

  add i, 1
  cmp i, 10.
  jbe fibonacci_loop

```

This script is mostly self-explanatory. Variables are declared using the *var* keyword and can be used to store numbers or strings. *\$RESULT* is a special variable used to store a result of previously executed command. All numbers used in the script are by default treated as hexadecimal numbers. To use a decimal number you must add a dot suffix to the number (for example *10.* == A, *11.* == B).

How to execute ODbgScripts in OllDbg will be presented in the next exercise in which you will also learn how to use scripting to automatically decode all hidden strings in the previously analyzed Tinba sample.

6.2 Decoding hidden strings in Tinba

This exercise starts where the previous exercise ended. If necessary, restore the snapshot named *Tinba* created when you reached the main Tinba payload.

00770FFD	E8 06F7FFFF	CALL 00770708	
00771002	E8 C2000000	CALL 007710C9	
00771007	89C3	MOV EBX, EAX	
00771009	E8 9D000000	CALL 007710AB	
0077100E	89C7	MOV EDI, EAX	
00771010	E8 0E000000	CALL 00771023	
00771015	4D	DEC EBP	
00771016	61	POPAD	
00771017	70 56	J0 SHORT 0077106F	
00771019	6965 77 4F664669	IMUL ESP, DWORD PTR SS:[EBP+77], 6946664F	
00771020	6C	INS BYTE PTR ES:[EDI], DX	I/O command

Step into (F7), the first call instruction (actually this call would never return, everything important is taking place inside this call).

00770708	31C0	XOR EAX, EAX	
0077070A	40	INC EAX	
0077070B	90	NOP	
0077070C	75 28	JNZ SHORT 00770736 // always taken	
0077070E	48	DEC EAX	
0077070F	83EC 08	SUB ESP, 8	
00770712	E8 18130000	CALL 00771A2F	
00770717	E8 99190000	CALL 007720B5	
0077071C	48	DEC EAX	
0077071D	83C4 08	ADD ESP, 8	
00770720	48	DEC EAX	
00770721	C7C1 88130000	MOV ECX, 1388	
00770727	FF15 EFF9FFFF	CALL DWORD PTR DS:[FFFFFF9F]	
0077072D	48	DEC EAX	
0077072E	31C9	XOR ECX, ECX	
00770730	FF15 D3F9FFFF	CALL DWORD PTR DS:[FFFFFF9D]	
00770736	E8 00000000	CALL 0077073B	
0077073B	5B	POP EBX	00771002
0077073C	81EB 5F174000	SUB EBX, 40175F	
00770742	E8 E3090000	CALL 00771120	
00770747	E8 49100000	CALL 00771295 ← Step into	
0077074C	31C0	XOR EAX, EAX	

Next step into the seventh call instruction (F7).

```

00771795 55          PUSH EBP
00771796 89E5       MOV EBP,ESP
00771798 81EC 00010000 SUB ESP,100
0077179E 50         PUSH EAX
0077179F 8085 00FFFFFF LEA EAX,DWORD PTR SS:[EBP-100]
007717A5 50         PUSH EAX
007717A6 874424 04    XCHG DWORD PTR SS:[ESP+4],EAX
007717AA 6A 06     PUSH 6
007717AC E8 06000000 CALL 007717B7
007717B1 98       CWDE
007717B2 61       POPAD
007717B3 10DB     ADC BL,BL
007717B5 9A 35E84221 0000 CALL FAR 0000:2142E835
007717BC FF93 EE104000 CALL DWORD PTR DS:[EBX+4010EE]
007717C2 50       PUSH EAX
007717C3 8083 EC264000 LEA EAX,DWORD PTR DS:[EBX+4026EC]

```

Take a look at the first call instruction. You should notice two interesting things about it. Firstly, the call is jumping into a middle of an instruction (there is no disassembled instruction at `0x7717B7`). Secondly, instructions after this call don't make much sense.

What you see here is an anti-disassembly technique used by this Tinba variant. To see how it works step into this call (F7).

You should land at another call instruction followed by a second call to *LoadLibraryA*.

```

007717B7 E8 42210000 CALL 007738FE
007717BC FF93 EE104000 CALL DWORD PTR DS:[EBX+4010EE] kernel32.LoadLibraryA
007717C2 50       PUSH EAX
007717C3 8083 EC264000 LEA EAX,DWORD PTR DS:[EBX+4026EC]
007717C9 870424   XCHG DWORD PTR SS:[ESP],EAX
007717CC 50       PUSH EAX
007717CD E8 30F9FFFF CALL 00771102

```

If you had scrolled up in disassembly window the code would desynchronize.

```

007717AA 6A 06     PUSH 6
007717AC E8 06000000 CALL 007717B7
007717B1 98       CWDE
007717B2 61       POPAD
007717B3 10DB     ADC BL,BL
007717B5 9A 35E84221 0000 CALL FAR 0000:2142E835
007717BC FF93 EE104000 CALL DWORD PTR DS:[EBX+4010EE]
007717C2 50       PUSH EAX
007717C3 8083 EC264000 LEA EAX,DWORD PTR DS:[EBX+4026EC]
007717C9 870424   XCHG DWORD PTR SS:[ESP],EAX

```

What happened here is that the call that you stepped into was only used to push onto the stack address pointing to the data right after the call instruction (return address).

```

0021F764 007717B1 RETURN to 007717B1 from 007717B7
0021F768 00000006
0021F76C 0021F774 ← arg1
0021F770 0021F774 ← arg2
0021F774 0021F814 ASC "eSw" ← arg3
0021F778 7753EEC7 RET :dll.7753EEC7 from ntdll
0021F77C 77536D0E RET :dll.77536D0E from ntdll
0021F780 7753EC50 RETURN to ntdll.7753EC50 from ntdll

```

This call would actually never return and the address pushed onto the stack would be used as a first argument for the next called function.

```

007717B7 E8 42210000 CALL 007738FE
007717BC FF93 EE104000 CALL DWORD PTR DS:[EBX+4010EE] kernel32.LoadLibraryA
007717C2 50       PUSH EAX
007717C3 8083 EC264000 LEA EAX,DWORD PTR DS:[EBX+4026EC]
007717C9 870424   XCHG DWORD PTR SS:[ESP],EAX
007717CC 50       PUSH EAX
007717CD E8 30F9FFFF CALL 00771102

```

The function called in the next instruction takes three arguments (*arg1-arg3*). This functions is used to decrypt *arg2* number of bytes stored at the address pointed by *arg1* and save decrypted data to the address pointed by *arg3*. This isn't presented in this document but you can check this by yourself by stepping into this function.

Now follow in dump *arg3*.

Address	Hex dump	ASCII
0021F774	14 F8 21 00 C7 EE 53 77 0E 6D 53 77 50 EC 53 77	! eSw@mSwPwSw
0021F784	7C 02 2F 68 B3 04 77 00 E8 56 9E 75 FF FF 00 00	!@/h @w.3Uku ..
0021F794	00 00 93 75 15 A9 00 00 64 F8 21 00 20 A9 38 00	..du3r..d°!. r8.

And step over (F8) a call.

007717B7	E8 42210000	CALL 007738FE	
007717BC	FF93 EE104000	CALL DWORD PTR DS:[EBX+4010EE]	kernel32.LoadLibraryA
007717C2	50	PUSH EAX	
007717C3	8D83 EC264000	LEA EAX, DWORD PTR DS:[EBX+4026EC]	
007717C9	870424	XCHG DWORD PTR SS:[ESP], EAX	
007717CC	50	PUSH EAX	
007717CD	E8 30F9FFFF	CALL 00771102	

Take a look at the memory dump. A memory at the address pointed by *arg3* was overwritten with a decrypted text string. Now this string will be used as an argument for a *LoadLibraryA* call.

Address	Hex dump	ASCII
0021F774	4E 54 44 4C 4C 00 53 77 0E 6D 53 77 50 EC 53 77	NTDLL.Sw@mSwPwSw
0021F784	7C 02 2F 68 B3 04 77 00 E8 56 9E 75 FF FF 00 00	!@/h @w.3Uku ..
0021F794	00 00 93 75 15 A9 00 00 64 F8 21 00 20 A9 38 00	..du3r..d°!. r8.

What this mean is that Tinba stores encoded strings in-between normal assembly instructions. To decode such a string it uses the call instruction to push the address of encrypted data onto the stack and then calls the decoding routine.

This technique is used in several places of Tinba code and it always uses the same scheme:

OFFSET	INSTRUCTION CODE	INSTRUCTION	COMMENT
0x0	50	PUSH EAX	pushing dst. address (for decoded data)
0x1	87 44 24 04	XCHG DWORD PTR [ESP+4], EAX	
0x5	6A ?	PUSH <n>	pushing data length
0x7	E8 ? 00 00 00	CALL (0xB+n)	pushing src. address (5) onto the stack
0xB	<variable length encoded data>	-	
0xB+n	E8 ? ? ? ?	CALL <decode_func>	calling decode function

Question marks in the instruction code column represent a single byte with a variable value.

If you would like to find all encoded strings and decode them at once you can use OllyScript to automate this task.

The algorithm would be as follow:

1. Allocate memory for decoded data <dst>
2. Find next byte pattern "50 87 44 24 04 64 ?".
3. If pattern not found -> STOP.
4. Get encrypted data length <n> (push instruction operand)
5. Get encrypted data address <src>
6. Get decoding routine address
7. Call decoding routine in context of debugged process – decode(<src>, <n>, <dst>)
8. Output decoded string (<dst>).

9. Jump to step 2.

To use OllyScript create script.osc file with the following code:

```
var base
var labels

; checking memory base of Tinba payload
gmeme eip, MEMORYBASE
mov base, $RESULT

; allocating memory for results
alloc 1000
mov labels, $RESULT

; printing header information
eval "Memory base: 0x{base}"
log "-----"
log "Searching for encoded strings."
log $RESULT, ""
log "-----"

search_loop:
; searching for byte pattern
find base, #50874424046A??#
cmp $RESULT, 0
je end_loop

mov base, $RESULT
mov push_addr, base+5.
mov call_addr, base+7.
mov data_addr, base+12.

; finding data length
gopi push_addr, 1, DATA
mov len, $RESULT

; finding decode routine address
gci call_addr, DESTINATION
gci $RESULT, DESTINATION
mov decode_addr, $RESULT

; executing decode routine
exec
    pushad
    push {labels}
    push {len}
    push {data_addr}
    call {decode_addr}
    popad
ende

gstr labels
mov string, $RESULT
fill labels, len, 0
```



```

; printing result
eval "{data_addr} ({len} bytes) -> {string}"
log $RESULT, ""

add base, 7
jmp search_loop

end_loop:
log "-----"
free labels
pause

```

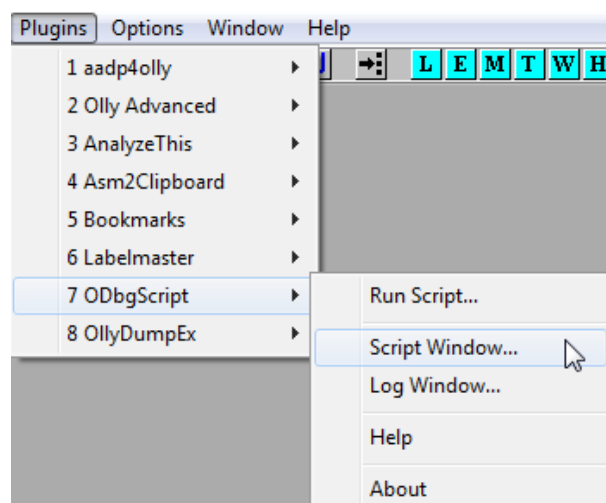
Commands used in this script were:

- **alloc** {size} – allocates {size} bytes of memory and returns address in \$RESULT
- **eval** {expression} – evaluates string expression with variables, returns string in \$RESULT
- **exec, ende** – executes assembly instructions between exec and ende in context of the debugged process
- **fill** {addr}, {len}, {value} – fills {len} bytes at address {addr} with specified {value}
- **find** {addr}, {pattern} – searches memory for {pattern} starting at address {addr}
- **free** {addr} – frees allocated memory at the address {addr}
- **gci** {addr}, DESTINATION – gets destination address of jump/call/return instruction
- **gmeme** {addr}, MEMORYBASE – gets base address of memory block to which {addr} belongs
- **gopi** {addr}, {n}, DATA – gets value of {n}th operand for instruction at address {addr}
- **gstr** {addr} – reads null terminated string from memory at specified address {addr}
- **je, jmp** – standard jump instructions
- **log** {str} – outputs provided string {str} in Script Log Window
- **mov** {dest}, {src} – standard mov instruction

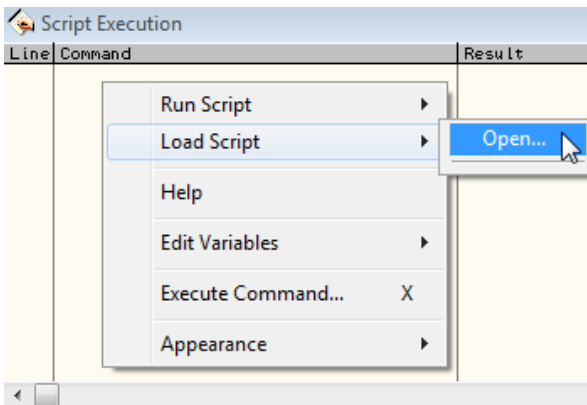
To get more detailed information about each command, refer to ODbgScript reference in the README.txt file.

To use this script first make sure that the EIP register points to the Tinba payload (for example you haven't followed in any API call).

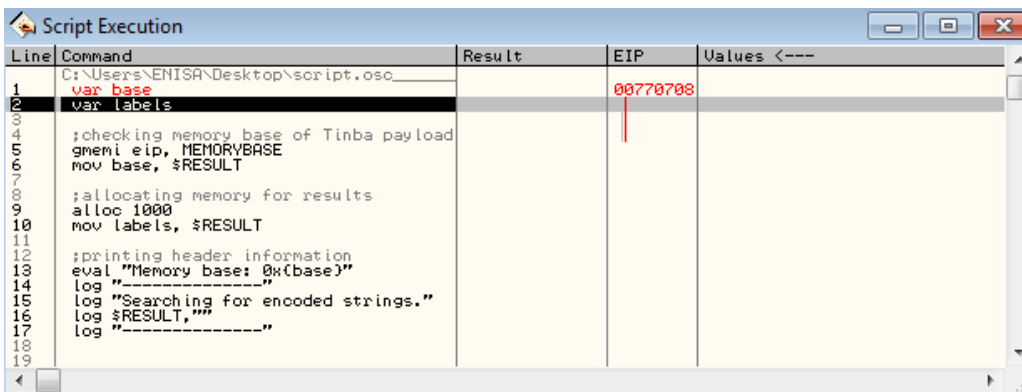
Then open ODbgScript *Script Window* and *Log Window*.



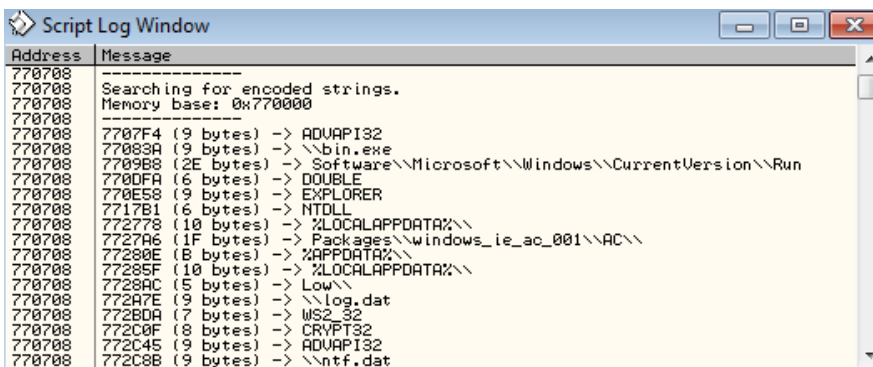
Next load *script.osc* in *Script Window* by right-clicking it and choosing *Load Script->Open*.



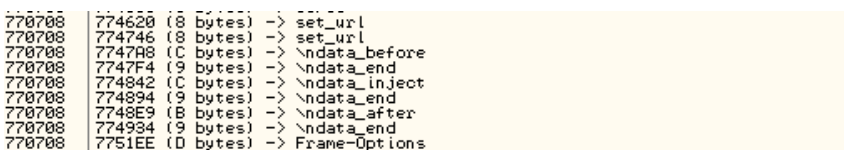
When the script is loaded press <space> to resume script execution or right-click on script and from the context menu choose *Resume*.



At the same time take a look at *Script Log Window* where the decoded strings should be printed.



Now that you know all encoded strings you can do typical string analysis to guess some of Tinba's functionality. For example on the strings list you can find strings such as *data_before*, *data_end*, *data_inject*, *data_after* which tell that Tinba is using webinjects technique known from other banking trojans.



Each printed line has the following message format:

```
{address} ({data_length}) -> {decoded_string}
```

Where {address} is an address where decoding instructions were found. This means that you can use printed messages to localize at what part of the code each string was used.

Additionally you could create a more advanced script, which would not only decode strings but also rewrite the Tinba code in such a way that it would reference to already decoded strings instead of decoding them at runtime.

7. Summary

In this training you have learnt the principles of malicious code debugging. Debugging usually requires a lot of patience and thinking outside the box. Various anti-debugging and anti-analysis techniques make this process much harder, but at the same time, debugging is often the quickest and easiest way of finding how a given sample really works.

When debugging, there are usually multiple ways of achieving the same goal: to unpack a binary sample, to check what its functions are or how it operates. The real skill is in how to achieve those goals in the quickest possible way without spending too much time on the analysis. This can be learnt only through regularly analysing malware samples, because that's when you learn different code patterns and get a better understanding of the system internals.



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

