



Advanced artefact analysis

Advanced dynamic analysis

TOOLSET, DOCUMENT FOR STUDENTS

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	14
2.3 Breakpoints	20
2.4 Execution flow manipulation	26
2.5 Plugins	28
2.6 Shortcuts	29
3. Unpacking artifacts	31
3.1 Unpacking UPX packed sample	31
3.2 Unpacking UPX with ESP trick	40
3.3 Unpacking Dyre sample	44
4. Anti-debugging techniques	54
4.1 Anti-debugging and anti-analysis techniques	54
4.2 Dyre - basic patching with OllyDbg	55
5. Process creation and injection	59
5.1 Following child processes of Tinba loader	59
5.1.1 First stage	59
5.1.2 Second stage	68
6. Introduction to scripting	74
6.1 Decoding hidden strings in Tinba	74

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	<p>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.</p>
Total Duration	8-10 hours

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

1. Training introduction

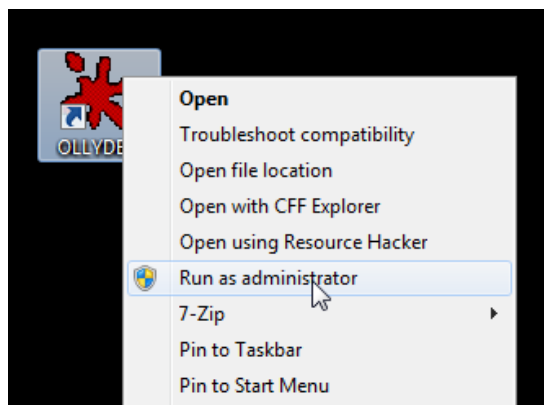
In this training you will learn practical elements of advanced dynamic analysis and debugging of malicious code.

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.

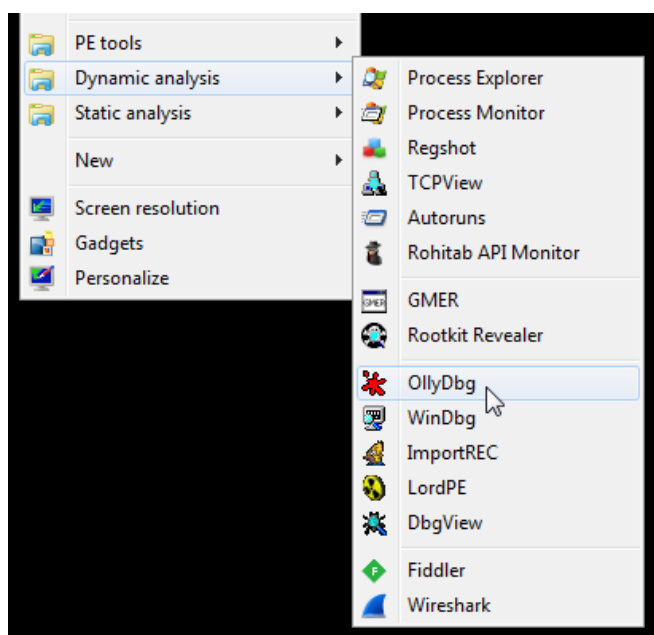
2. Introduction to OllyDbg

2.1 OllyDbg interface

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

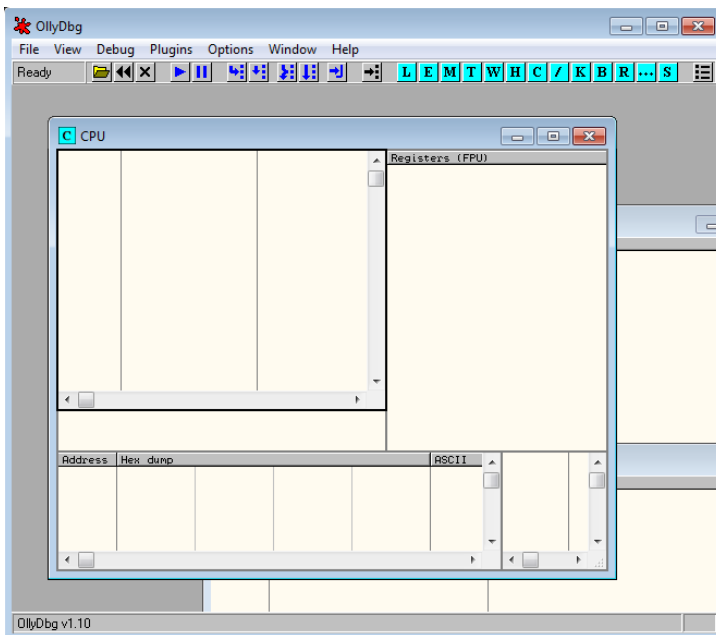


If you are using a Windows virtual machine prepared the same way as in the *Building artefact handling and analysis environment*⁴ training then you can also access OllyDbg using the context menu.

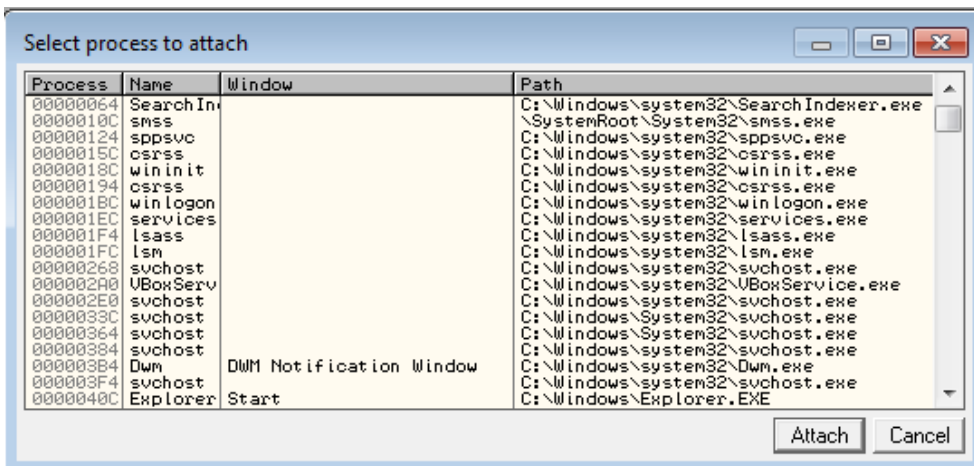


Now you should see the OllyDbg interface.

⁴Building artefact handling and analysis environment <https://www.enisa.europa.eu/activities/cert/training/training-resources/technical-operational#building> (last accessed 11.09.2015)

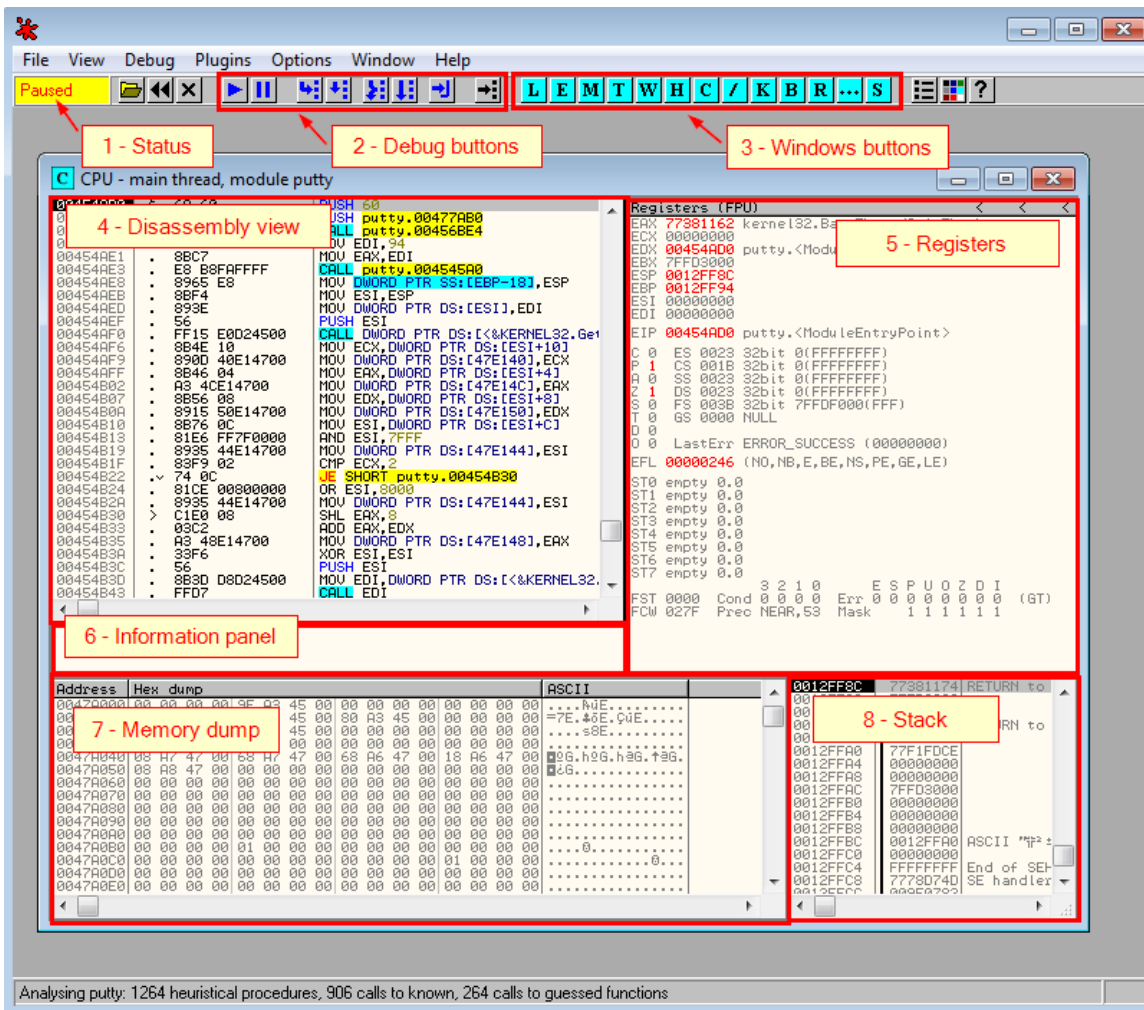


There are two ways to start a debugging process. Firstly, you can attach to the already running process. To do this, choose *File->Attach* and then choose the process of your interest. After attaching to the running process, OllyDbg should automatically break at the `ntdll.DbgBreakPoint` function.



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.

Disassembly view (4) presents a listing with the disassembled code.

00454AEF	. 56	PUSH ESI	GetVersionInformation = 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 00454A00 putty.<ModuleEntryPoint>
EBX 7FFD3000
ESP 0012FF8C
EBP 0012FF94
ESI 00000000
EDI 00000000
EIP 00454A00 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)
```

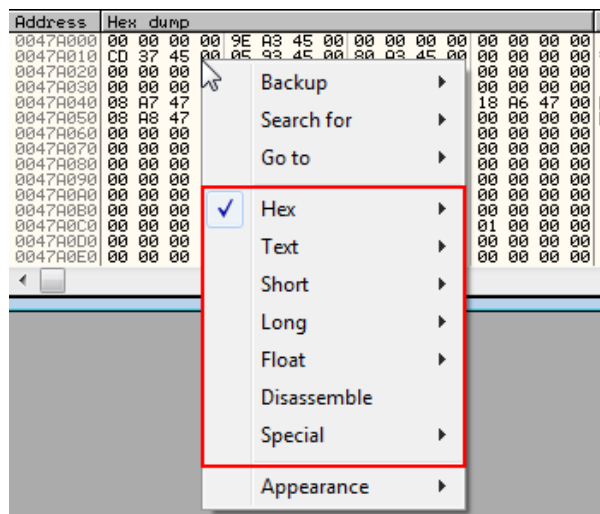
Information panel (6) is used to present additional information about the instruction selected in disassembly view (e.g. operation result, registers values).

```
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 00	...RUE.....
0047A010	CD 37 45 00 05 93 45 00 80 A3 45 00 00 00 00 00	=7E.#6E.CUE....
0047A020	00 00 00 00 73 38 45 00 00 00 00 00 00 00 00 00	...s8E.....
0047A030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0047A040	08 A7 47 00 68 A7 47 00 68 A6 47 00 18 A6 47 00 00	88.h8G.h8G.↑8G.
0047A050	08 A8 47 00 00 00 00 00 00 00 00 00 00 00 00 00	88.h8G.h8G.↑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.



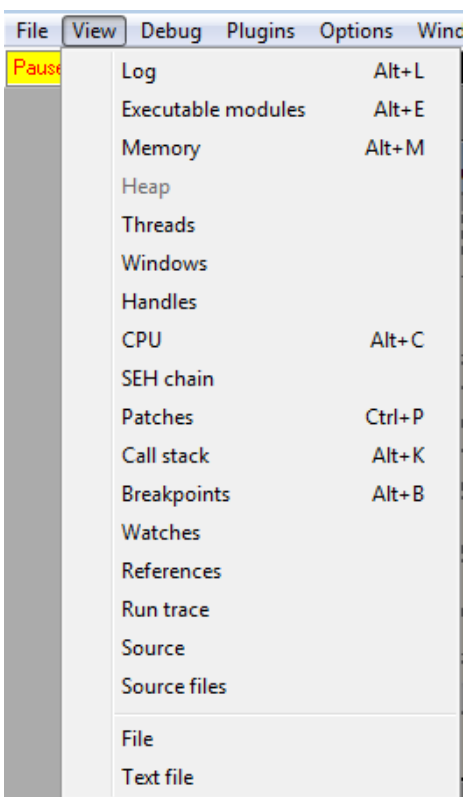
Take some time to check other data representations. At the end, restore the default format: *Hex->Hex/ASCII* (16 bytes).

Finally, *stack panel* (8) presents the stack state of the currently selected thread.

```

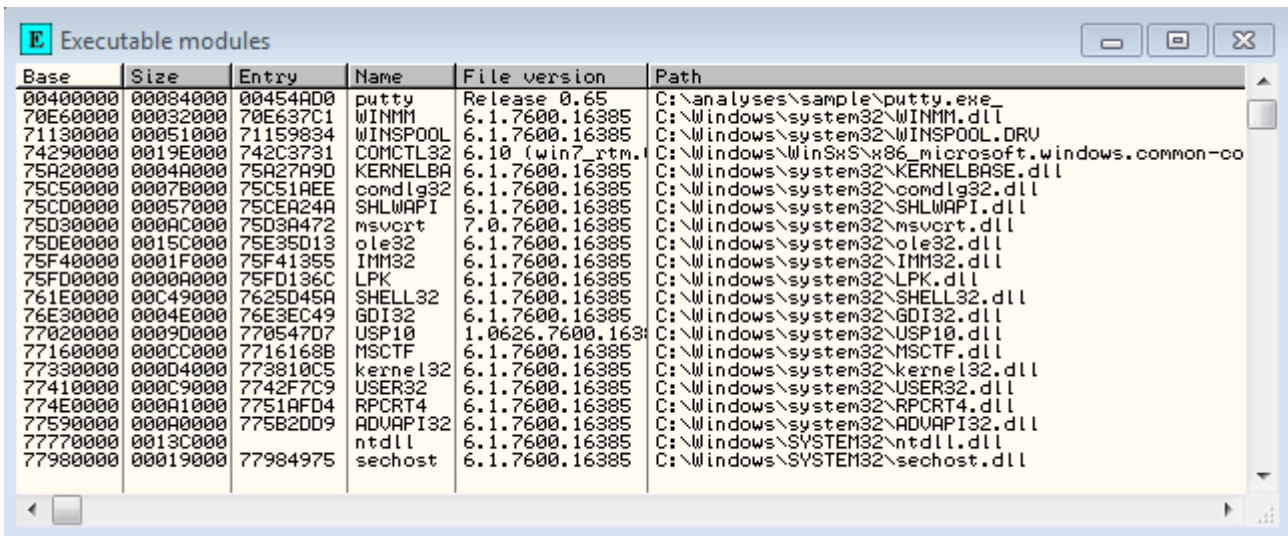
0012FF8C 77381174 RETURN to kernel32.77381174
0012FF90 7FFD3000
0012FF94 0012FFD4
0012FF98 777CB3F5 RETURN to ntdll.777CB3F5
0012FF9C 7FFD3000
0012FFA0 77F1FDCE
0012FFA4 00000000
0012FFA8 00000000
0012FFAC 7FFD3000
0012FFB0 00000000
0012FFB4 00000000
0012FFB8 00000000
0012FFBC 0012FFA0 ASCII "f² ±w"
0012FFC0 00000000
  
```

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.



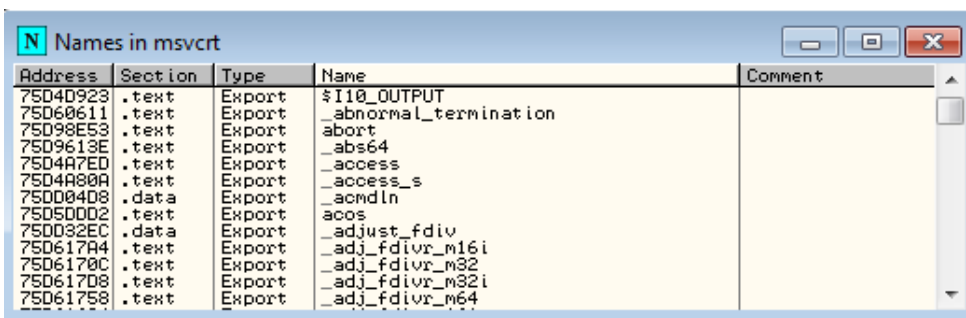
The more frequently used windows are: *Executable modules*, *Memory map*, *Threads*, *Handles*, *Call stack*, *Breakpoints*.

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.



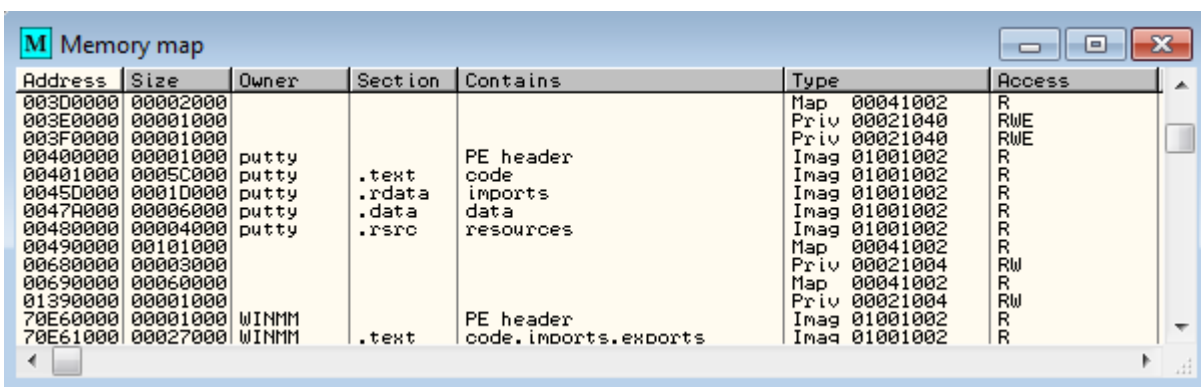
Base	Size	Entry	Name	File version	Path
00400000	00084000	00454A00	putty	Release 0.65	C:\analyses\sample\putty.exe
70E60000	00032000	70E637C1	WINMM	6.1.7600.16385	C:\Windows\system32\WINMM.dll
71130000	00051000	71159834	WINSPOOL	6.1.7600.16385	C:\Windows\system32\WINSPOOL.DRV
74290000	0019E000	742C3731	COMCTL32	6.10 (win7_rtm)	C:\Windows\WinSxS\x86_microsoft.windows.common-co
75A20000	0004A000	75A27A9D	KERNELBA	6.1.7600.16385	C:\Windows\system32\KERNELBASE.dll
75C50000	0007B000	75C51AEE	comdlg32	6.1.7600.16385	C:\Windows\system32\comdlg32.dll
75CD0000	00057000	75CEA24A	SHLWAPI	6.1.7600.16385	C:\Windows\system32\SHLWAPI.dll
75D30000	000AC000	75D3A472	msvcrt	7.0.7600.16385	C:\Windows\system32\msvcrt.dll
75DE0000	0015C000	75E35D13	ole32	6.1.7600.16385	C:\Windows\system32\ole32.dll
75F40000	0001F000	75F41355	IMM32	6.1.7600.16385	C:\Windows\system32\IMM32.dll
75FD0000	0000A000	75FD136C	LPK	6.1.7600.16385	C:\Windows\system32\LPK.dll
761E0000	00C49000	7625D45A	SHELL32	6.1.7600.16385	C:\Windows\system32\SHELL32.dll
76E30000	0004E000	76E3EC49	GDI32	6.1.7600.16385	C:\Windows\system32\GDI32.dll
77020000	0009D000	770547D7	USP10	1.0626.7600.16385	C:\Windows\system32\USP10.dll
77160000	000CC000	7716168B	MSCTF	6.1.7600.16385	C:\Windows\system32\MSCTF.dll
77330000	000D4000	773810C5	kernel32	6.1.7600.16385	C:\Windows\system32\kernel32.dll
77410000	000C9000	7742F7C9	USER32	6.1.7600.16385	C:\Windows\system32\USER32.dll
774E0000	000A1000	7751AFD4	RPCRT4	6.1.7600.16385	C:\Windows\system32\RPCRT4.dll
77590000	000A0000	775B2DD9	ADVAPI32	6.1.7600.16385	C:\Windows\system32\ADVAPI32.dll
77770000	0013C000		ntdll	6.1.7600.16385	C:\Windows\SYSTEM32\ntdll.dll
77980000	00019000	77984975	sechost	6.1.7600.16385	C:\Windows\SYSTEM32\sechost.dll

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).



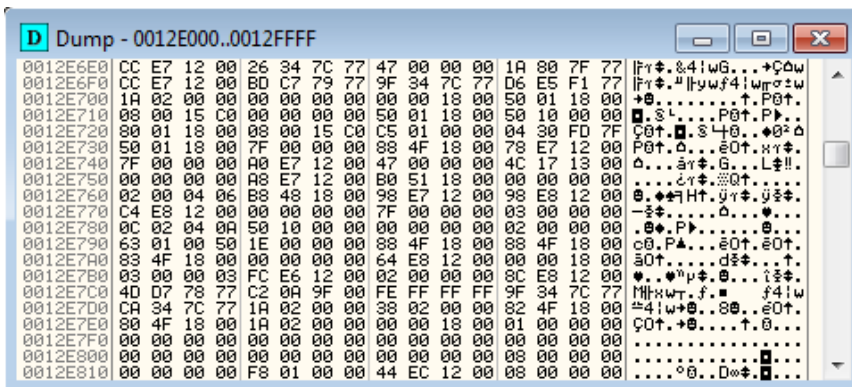
Address	Section	Type	Name	Comment
75040923	.text	Export	\$I10_OUTPUT	
75060611	.text	Export	_abnormal_termination	
75098E53	.text	Export	abort	
7509613E	.text	Export	_abs64	
7504A7ED	.text	Export	_access	
7504A80A	.text	Export	_access_s	
750004D8	.data	Export	_acmdln	
7505DD02	.text	Export	acos	
750032EC	.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.



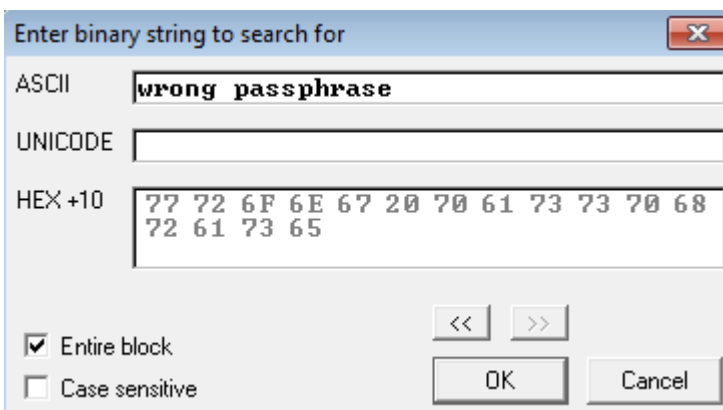
Address	Size	Owner	Section	Contains	Type	Access
003D0000	00002000				Map 00041002	R
003E0000	00001000				Priv 00021040	RWE
003F0000	00001000				Priv 00021040	RWE
00400000	00001000	putty		PE header	Imag 01001002	R
00401000	0005C000	putty	.text	code	Imag 01001002	R
0045D000	0001D000	putty	.rdata	imports	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	00006000				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.

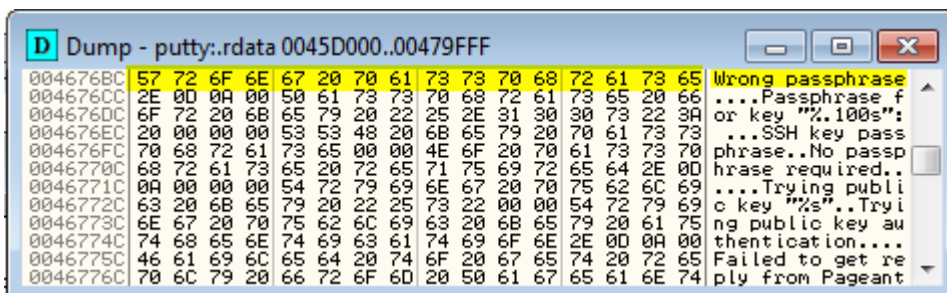


Another operation you might try is searching all memory regions for a particular string or byte pattern. Let's say you know that somewhere in the memory the string *'wrong passphrase'* is present, but you don't know the exact address nor in which memory region is it located.

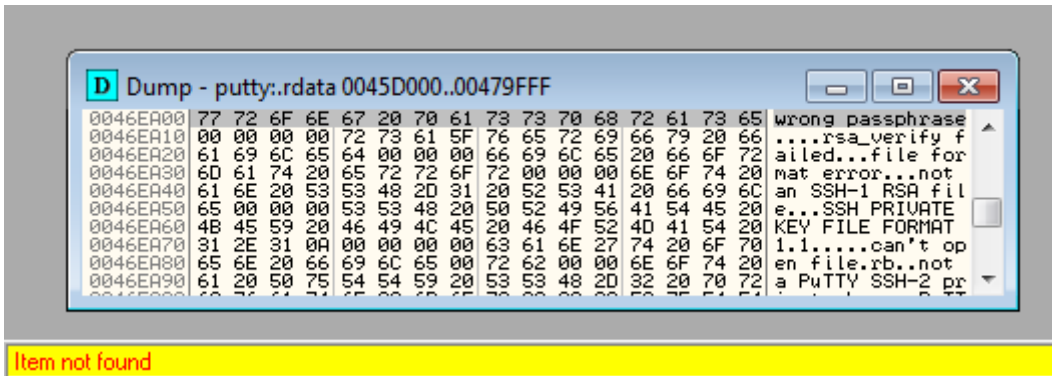
To solve this problem, right-click anywhere in the memory map and choose *Search* (Ctrl+B) from the context menu. In the new window, type *'wrong passphrase'* and click *Ok*.



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

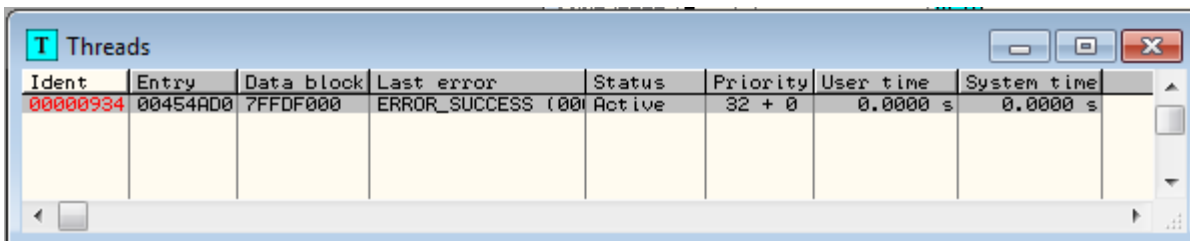


To keep searching for other occurrences of this string in this memory region click on *Dump* window (to make it active) and keep pressing Ctrl+L.

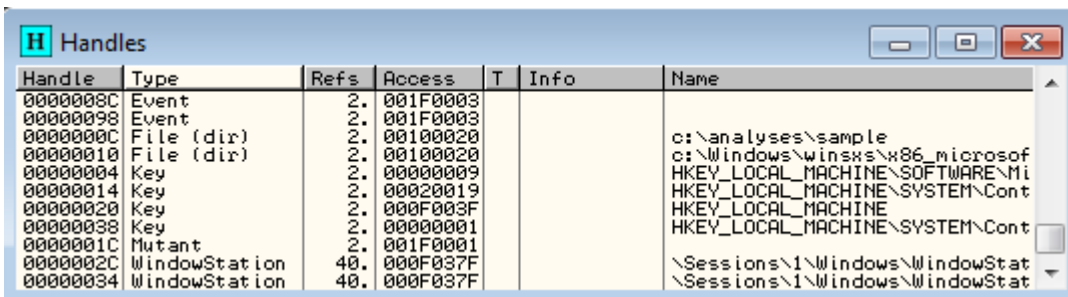


To continue searching for the string in other memory regions go back to *Memory map* window (make it active) and keep pressing Ctrl+L.

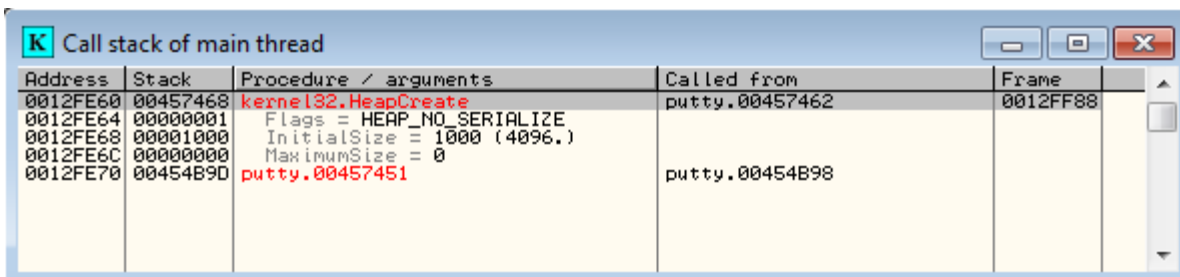
Threads window shows all threads of the current process.



Handles window shows all windows opened by the process handles with an additional information regarding the handle type, value and name. This window may be useful for example you see that some API call is referring to a certain handle and you don't remember what this handle is.

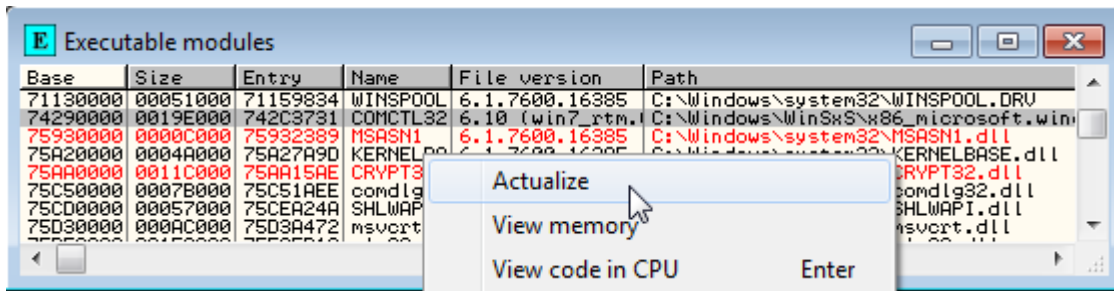


Finally *call stack* window shows all function calls made up to the current instruction in the current thread.

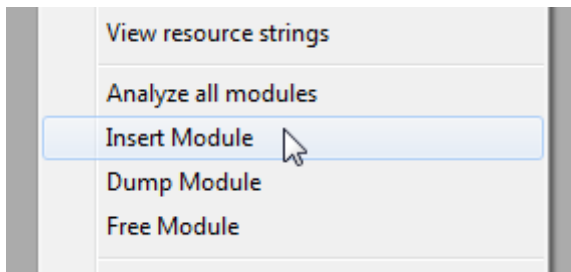


One of the useful OllyDbg features is highlighting elements that have changed.

Open the *Executable modules* window. If there are any red coloured elements in the window, right-click it and choose *Actualize*.

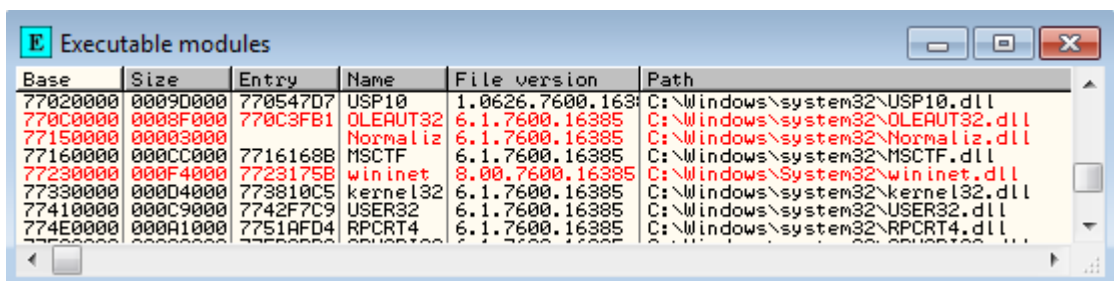


Next, right-click anywhere in the window and choose *Insert module* from the context menu (this operation is available only with Olly Advanced plugin).



In the *Open* dialog, choose *c:\Windows\System32\wininet.dll*.

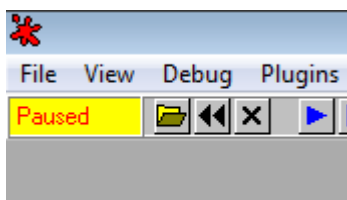
Now all the newly loaded modules should be marked with red font in the *Executable modules* window.



2.2 Basic debugging and code navigation

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

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
00454AD0 . 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

There are two modes of instruction stepping:

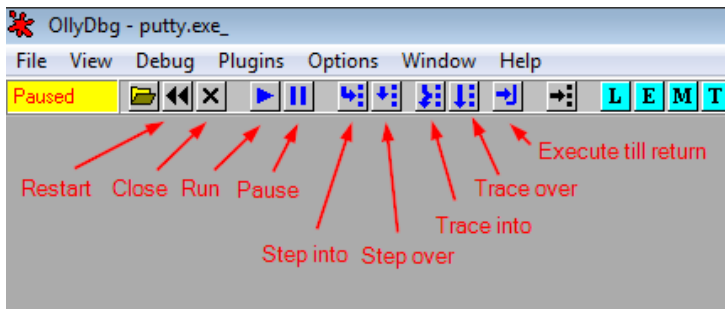
- *Step into* (F7) – executes current instruction and moves program execution to the next instruction. If the current instruction is a function call then the debugger steps into the call and starts stepping over instructions of the called function.
- *Step over* (F8) – behaves the same as *Step into* 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*).

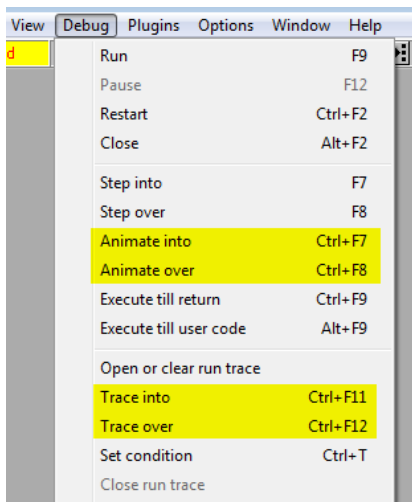
Other useful debug operations are:

- *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.

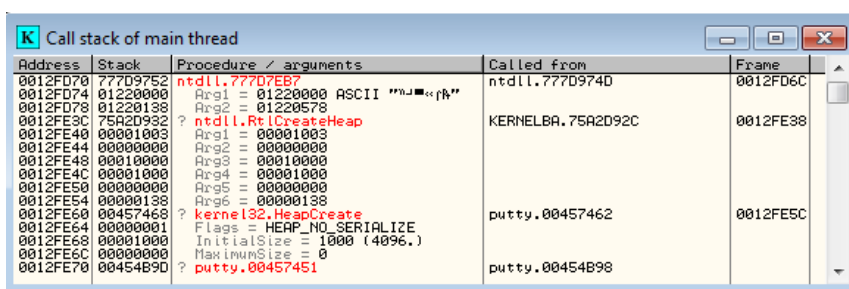


If you want to quickly pre-view the execution flow of a program (find loops, check which jumps are taken, etc.) you might decide to use the instruction trace or instruction animation functions.



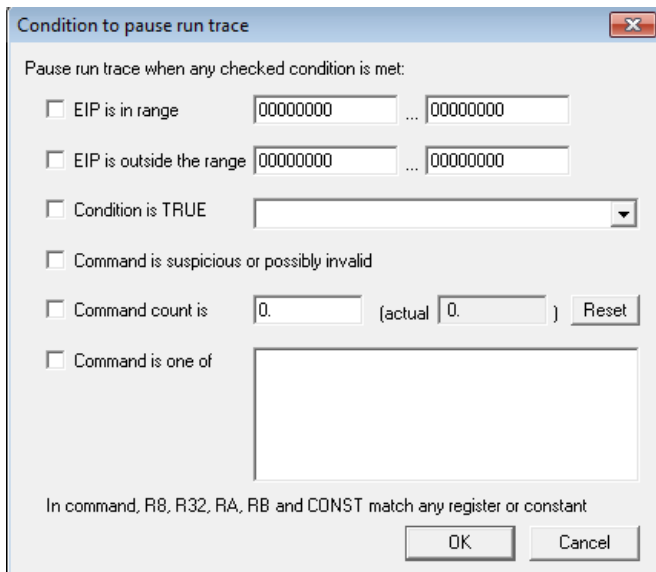
Restart PuTTY sample (*Debug->Reset*) and then choose *Debug -> Animate over* (Ctrl+F8). Observe what happens in the disassembly window.

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.

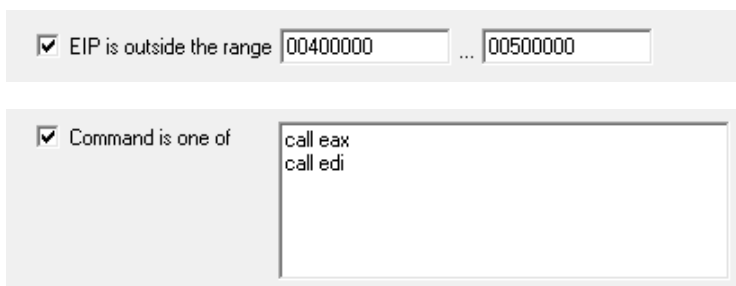


Animate into function usually takes some time until the program finishes execution. To stop it, use *Pause* (F12) function.

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



Set the following two conditions.



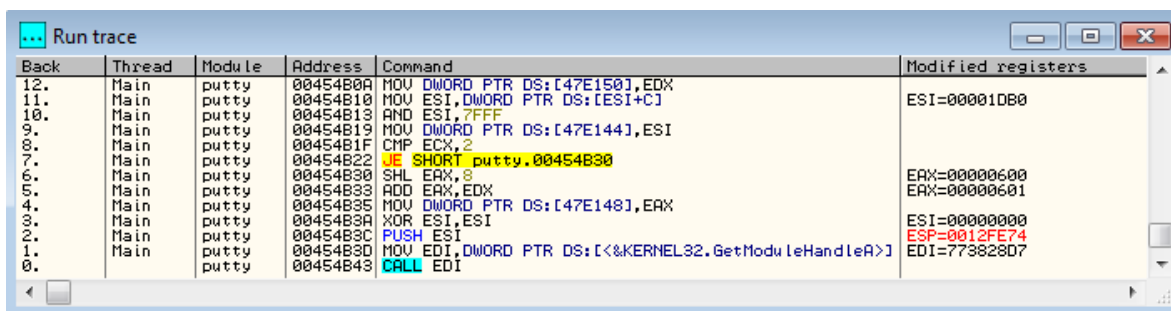
Now open *Run trace* window (*View->Run trace*) and then choose *Debug->Trace over*.

00454B3A	. 39F6	XOR ESI,ESI
00454B3C	. 56	PUSH ESI
00454B3D	. 8B3D D8D24500	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.00454B68
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.



Now take a look at the *Run trace* window.



Back	Thread	Module	Address	Command	Modified registers
12.	Main	putty	00454B0A	MOV DWORD PTR DS:[47E150],EDX	
11.	Main	putty	00454B10	MOV ESI,DWORD PTR DS:[ESI+C]	ESI=000010B0
10.	Main	putty	00454B13	AND ESI,7FFF	
9.	Main	putty	00454B19	MOV DWORD PTR DS:[47E144],ESI	
8.	Main	putty	00454B1F	CMP ECX,2	
7.	Main	putty	00454B22	JE SHORT putty.00454B30	
6.	Main	putty	00454B30	SHL EAX,3	EAX=00000600
5.	Main	putty	00454B33	ADD EAX,EDX	EAX=00000601
4.	Main	putty	00454B35	MOV DWORD PTR DS:[47E148],EAX	
3.	Main	putty	00454B3A	XOR ESI,ESI	ESI=00000000
2.	Main	putty	00454B3C	PUSH ESI	ESP=0012FE74
1.	Main	putty	00454B3D	MOV EDI,DWORD PTR DS:[&KERNEL32.GetModu	EDI=773828D7
0.	Main	putty	00454B43	CALL EDI	

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).

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>.

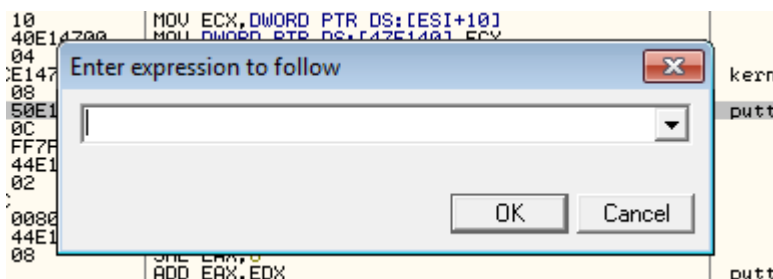
In this example follow a call to *putty.004545A0*.

```

0045459E  CC          INT3
0045459F  CC          INT3
004545A0  $ 3D 00100000    CMP EAX,1000
004545A5  . 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
  
```

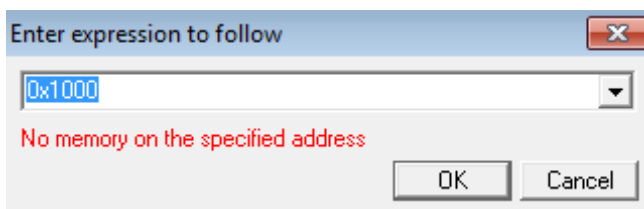
Another way of navigating through the code is using the *Go to expression* feature.

Click on disassembly view and press Ctrl+G.

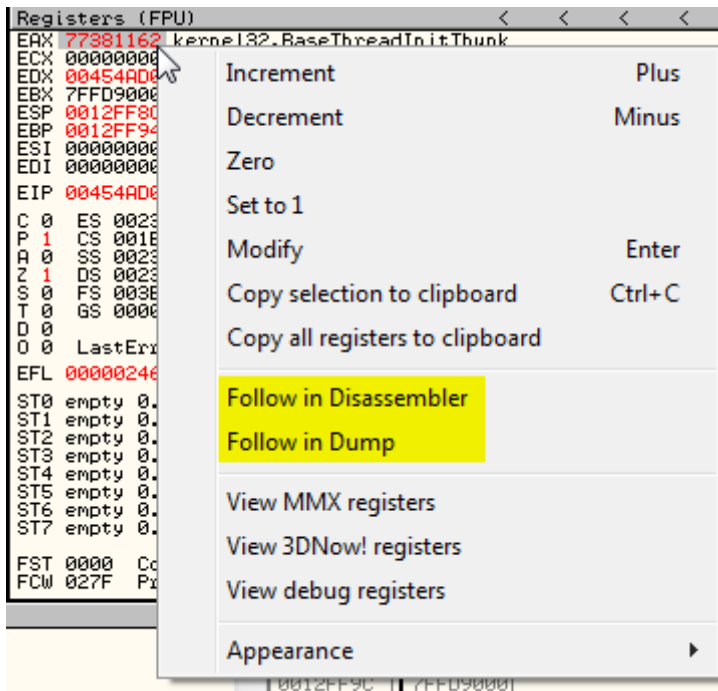


Type *eip* to be moved to the current location in the code (pointed by EIP register).

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.

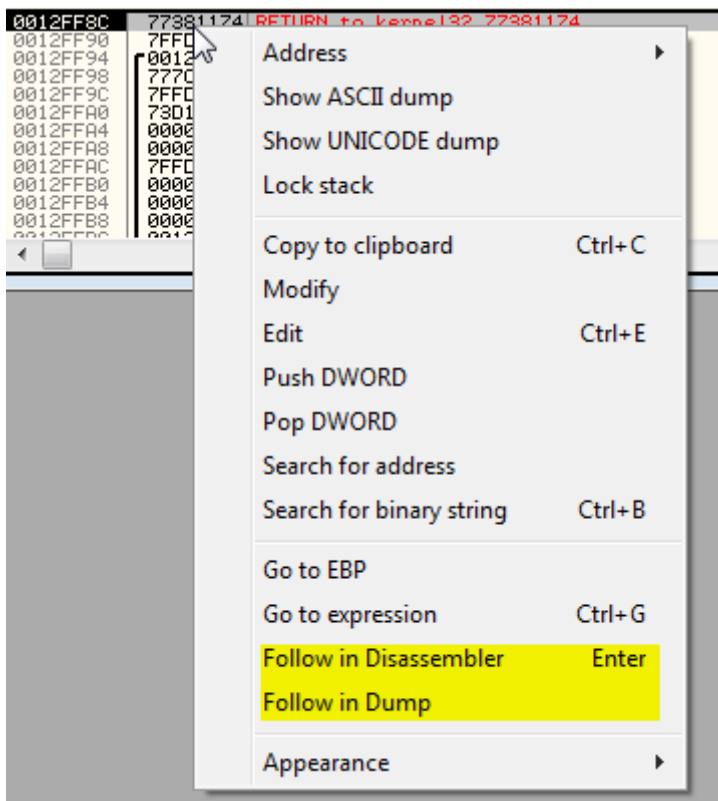


Start clicking on registers values. If a register points to the existing address in the address space of the current program, there should be the following options: *Follow in Disassembler* and *Follow in Dump*.



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

To set a software breakpoint, double-click in the second column next to the instruction or select an instruction and press F2.

```

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>]
  
```

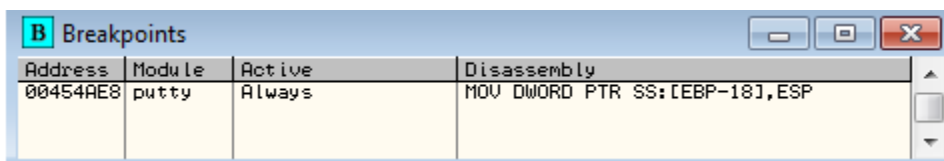
Now press F9 (run) and the program should stop execution on this instruction (before executing it).

```

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>]
  
```

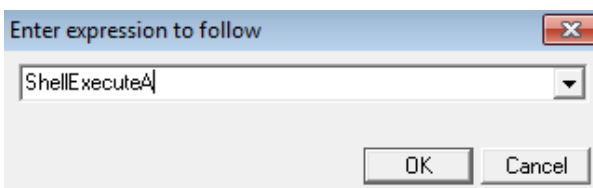
To remove a breakpoint, repeat the same steps as when setting it.

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



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

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



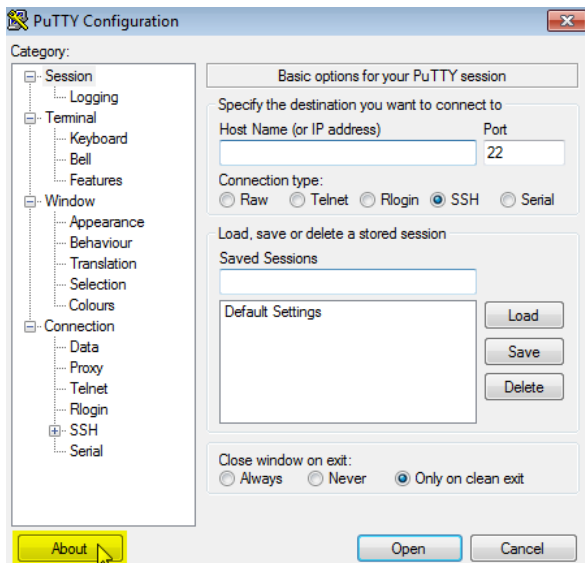
Then set breakpoint on the first instruction of *ShellExecuteA* (the one to which you were moved).

```

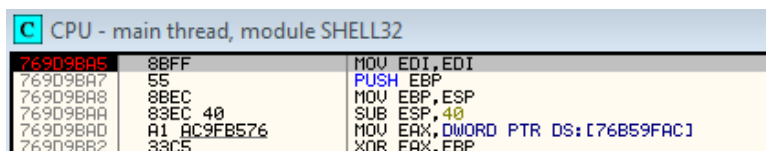
769D9BA5 8BFF MOV EDI,EDI
769D9BA7 55 PUSH EBP
769D9BA8 8BEC MOV EBP,ESP
769D9BAA 83EC 40 SUB ESP,40
769D9BAD A1 AC9FB576 MOV EAX,DWORD PTR DS:[76B59FAC]
769D9BB2 33C5 XOR EAX,EBP
769D9BB4 8945 FC MOV DWORD PTR SS:[EBP-4],EAX
769D9BB7 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
769D9BBA 8B4D 0C MOV ECX,DWORD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

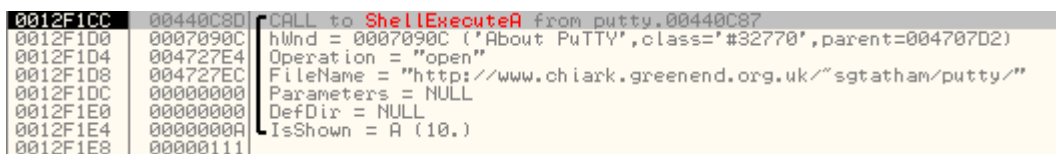


Now go back to OllyDbg.



Breakpoint at SHELL32.ShellExecuteA

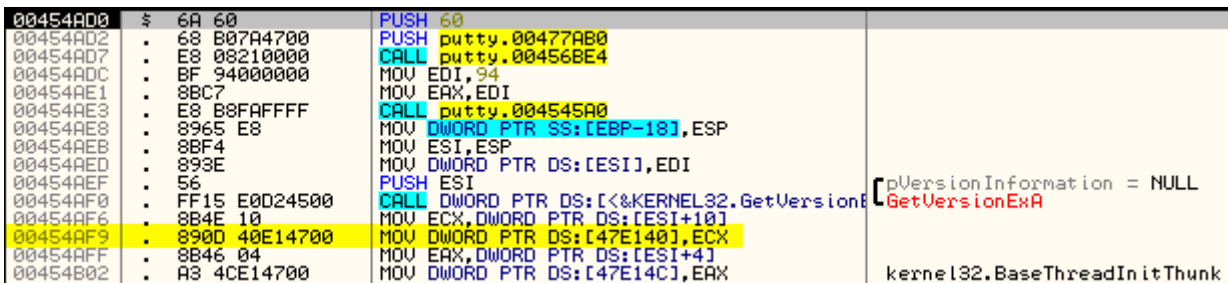
Take a look at the stack view to see arguments passed to *ShellExecuteA*.



Open the call stack window (*View->Call stack, Ctrl+K*) to check from where *ShellExecuteA* function was called.



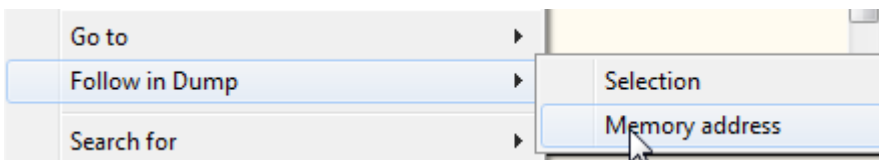
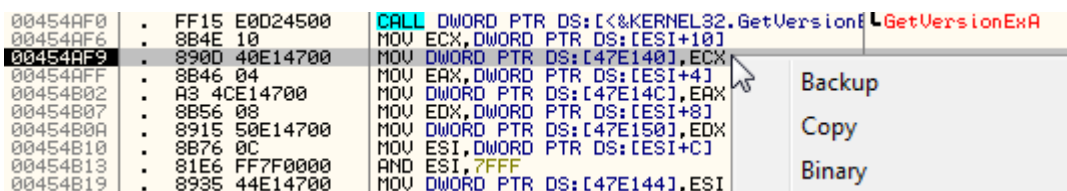
Go to OllyDbg and restart the PuTTY sample.



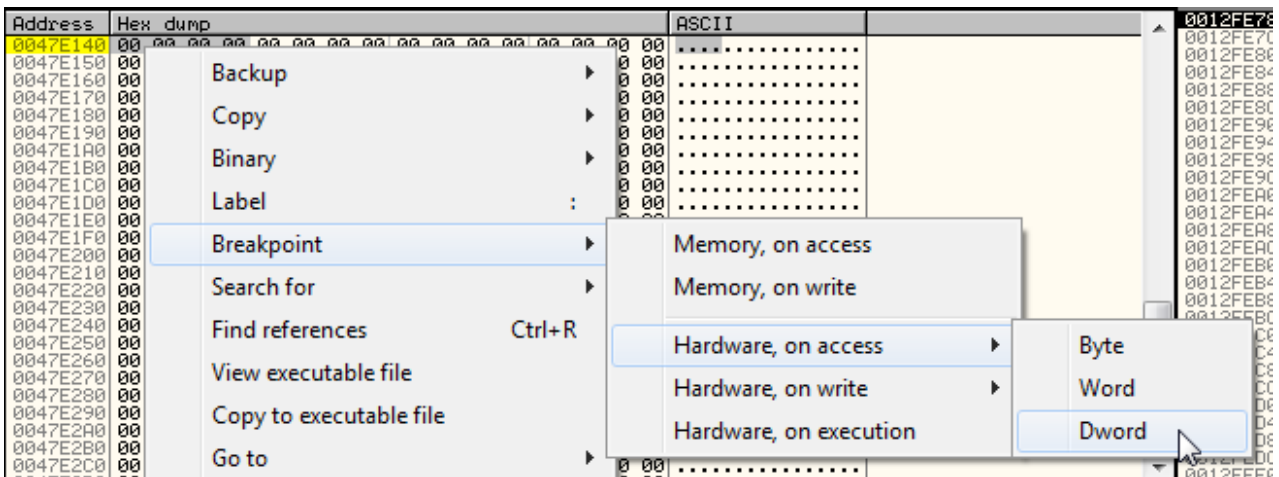
Next, step over until the instruction at 0x454AF9. As you can see some dword value is being written to the memory at the address 0x47E140.

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

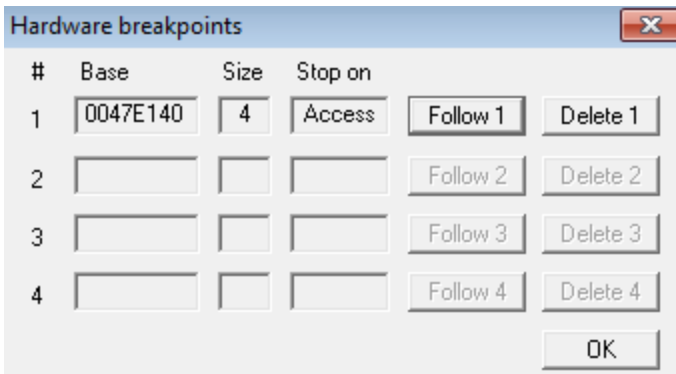
Right-click on this instruction and from the context menu choose *Follow in Dump->Memory address*.



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*.



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



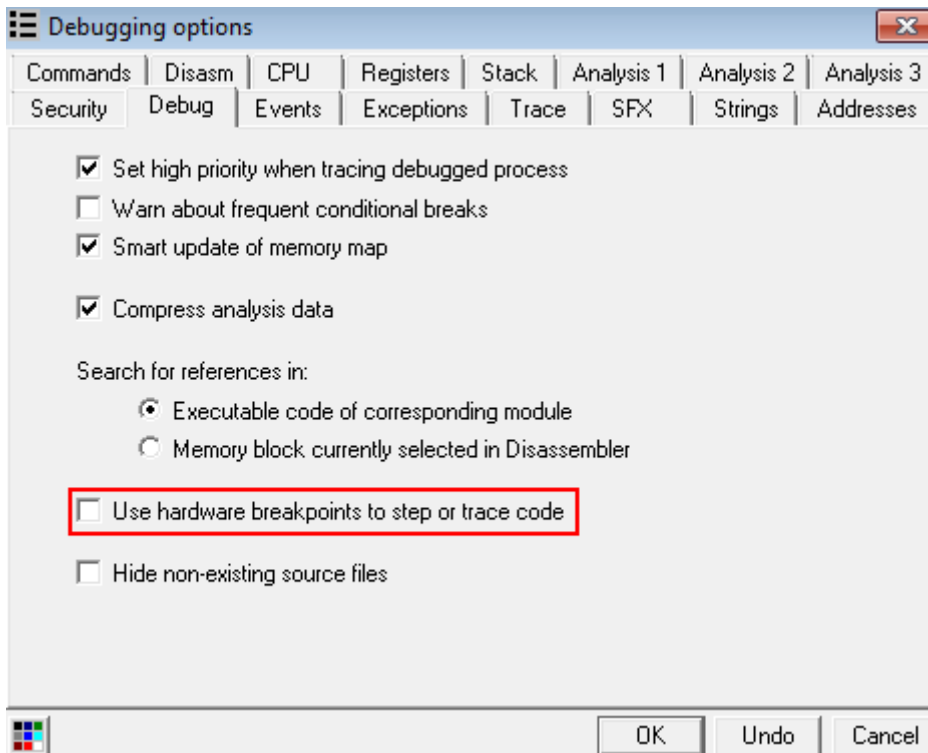
After setting up a hardware breakpoint on *0x47E140*, resume the program execution (F9).

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

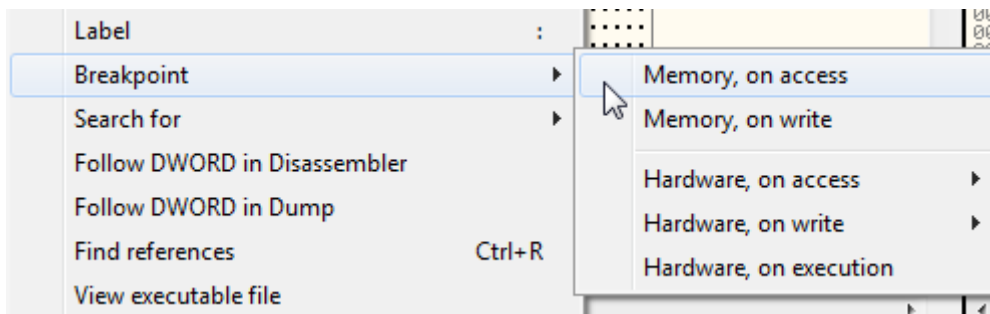
Scroll the disassembly view one line up to see the instruction accessing *0x47E140*.



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

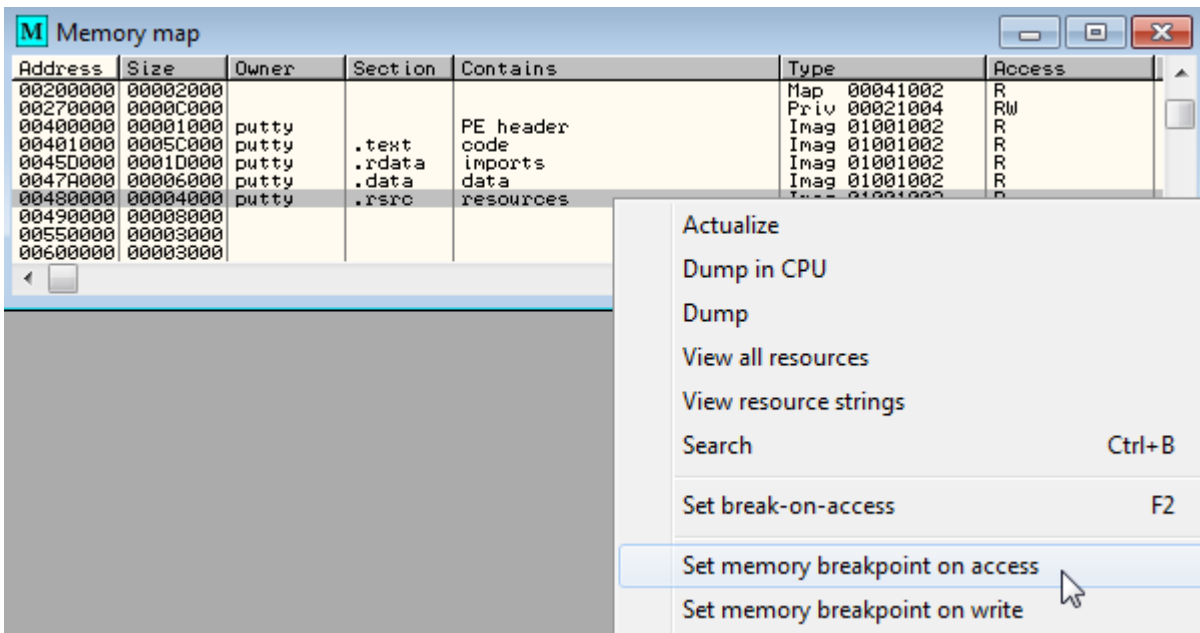


You can set memory breakpoints in a similar manner as hardware breakpoints by selecting some data in *Memory Dump* view and then choosing *Breakpoint->Memory*.

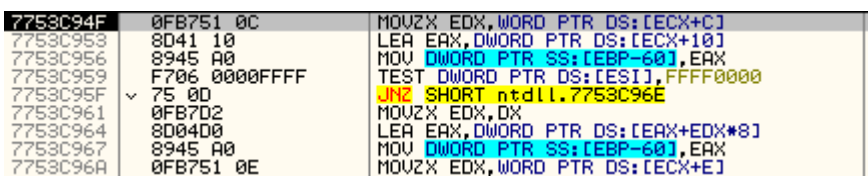


The second way of creating a memory breakpoint is using *Memory map* window.

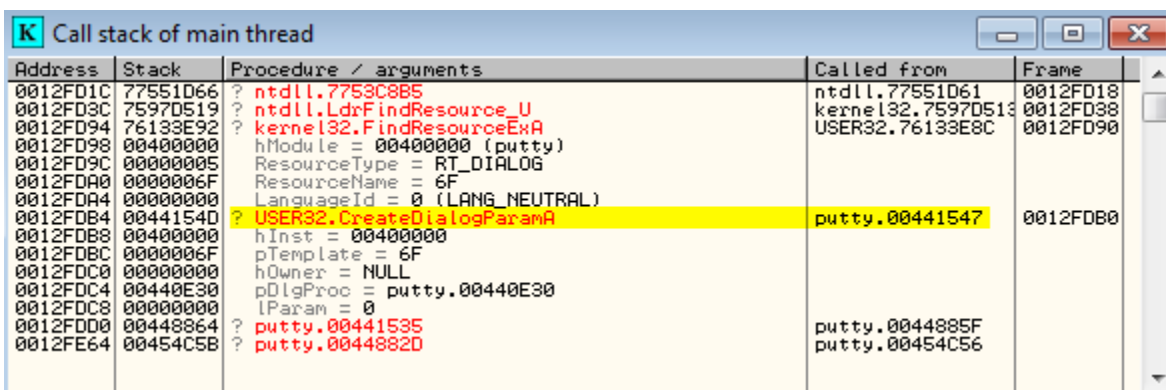
Restart the PuTTY sample and open *Memory map* window. Then find PuTTY's *.resource* section, right-click it and from the context menu, choose *Set memory breakpoint on access*. Now if some code tries to access any data in *.resource* section, the breakpoint would hit.



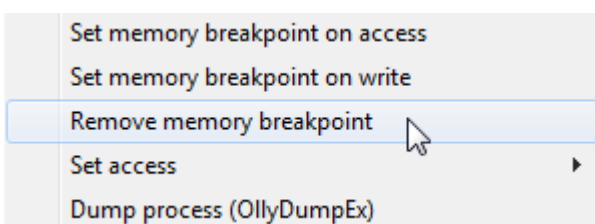
Next, resume the program (F9). The breakpoint should hit someplace in the system code.



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



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

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 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 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.

```

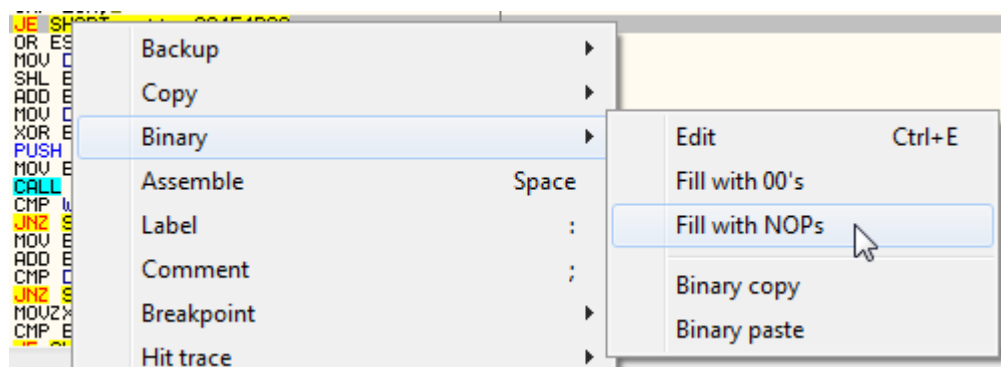
EIP 00454B22 putty.00454B22
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(4000)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_INSUFFICIENT_BUFFER (0000007A)
  
```

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
  
```

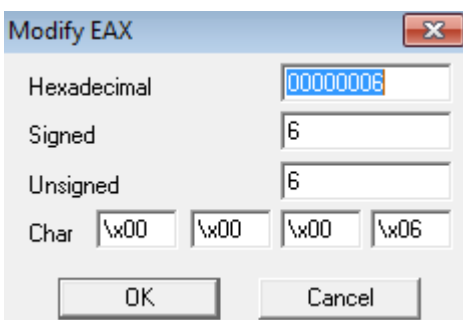
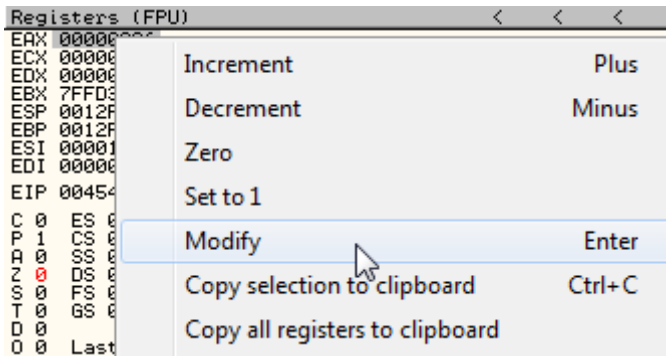
You can also change a jump to never be made by overwriting the jump instruction with NOP instructions. To do this, just right-click on the jump instruction and choose *Binary->Fill with NOPs*.



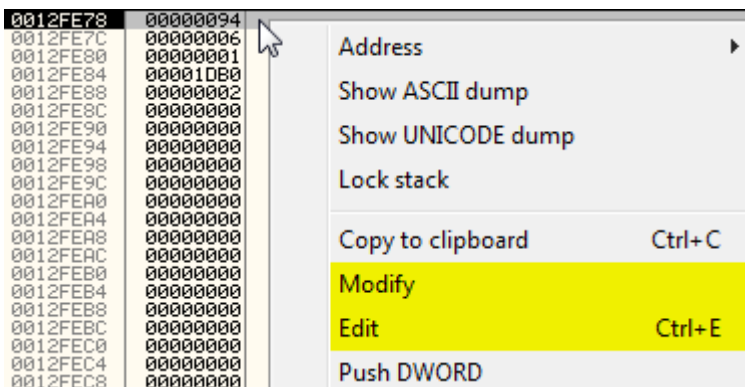
```

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

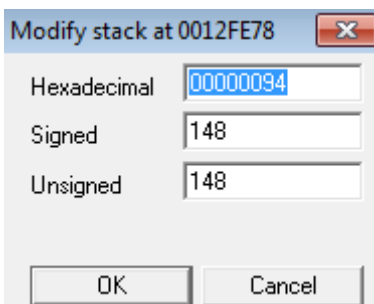
In a similar way as modifying the FLAGS register you can also modify other registers. To do this, right-click on the register value and choose *Modify*.

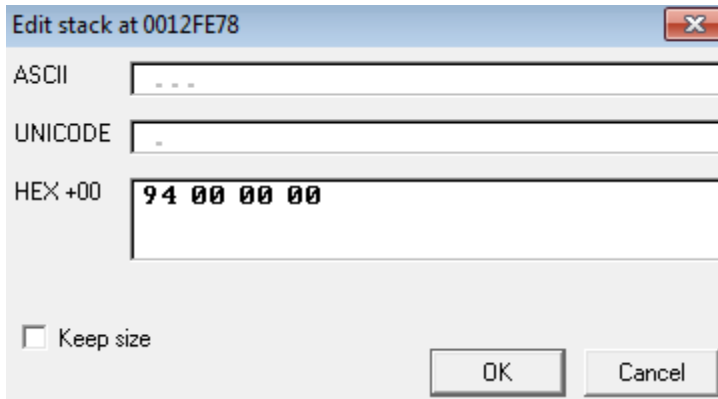


Values on the stack can be modified as well.

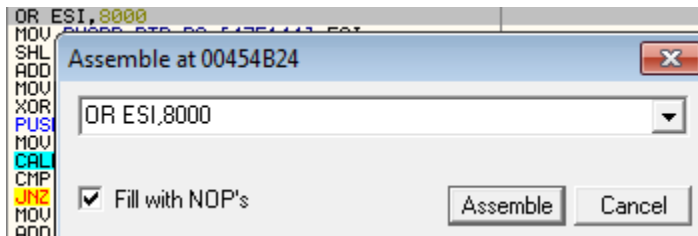


This time however there are two options: *Modify* and *Edit*.



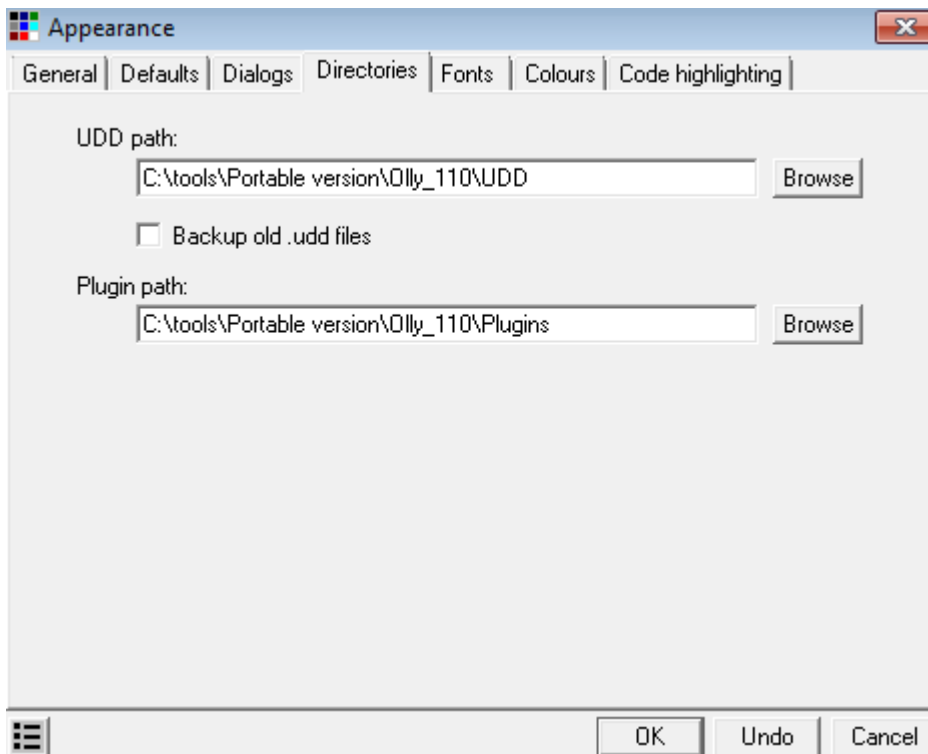


Besides modifying registers and data in the memory, it is also possible to change instructions that are executed. To achieve this just select the instruction you want to modify and press <space>.

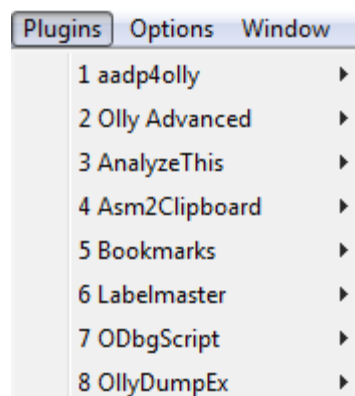


2.5 Plugins

After downloading a plugin, unpack it and copy the plugin's .dll library to the OlyDbg's plugin directory (e.g. *c:\tools\Portable version\Oilly_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:

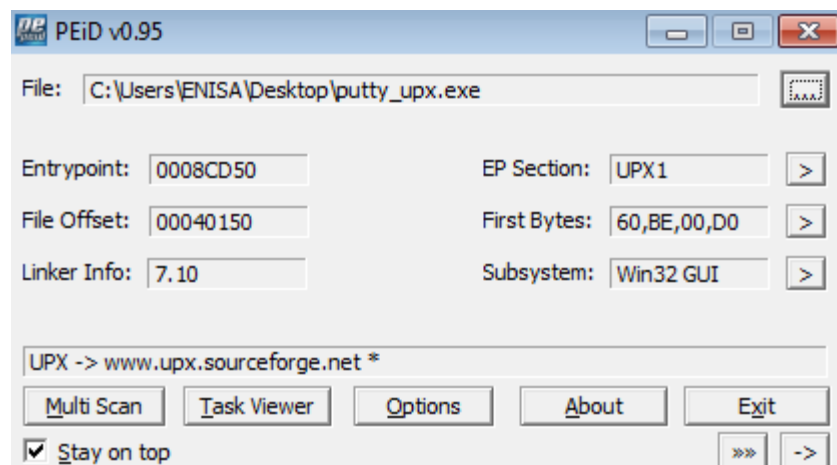
OPERATION	SHORTCUT
Follow jump/call	Enter
Assembly instruction	Space
Edit memory	Ctrl+E
Add comment	; (semicolon)
Add label	: (colon)
Insert bookmark X	Alt+Shift+0..9
Go to bookmark X	Alt+0..9

3. Unpacking artifacts

3.1 Unpacking UPX packed sample

PEiD⁵ is used to identify if the sample was packed and what packer was used.

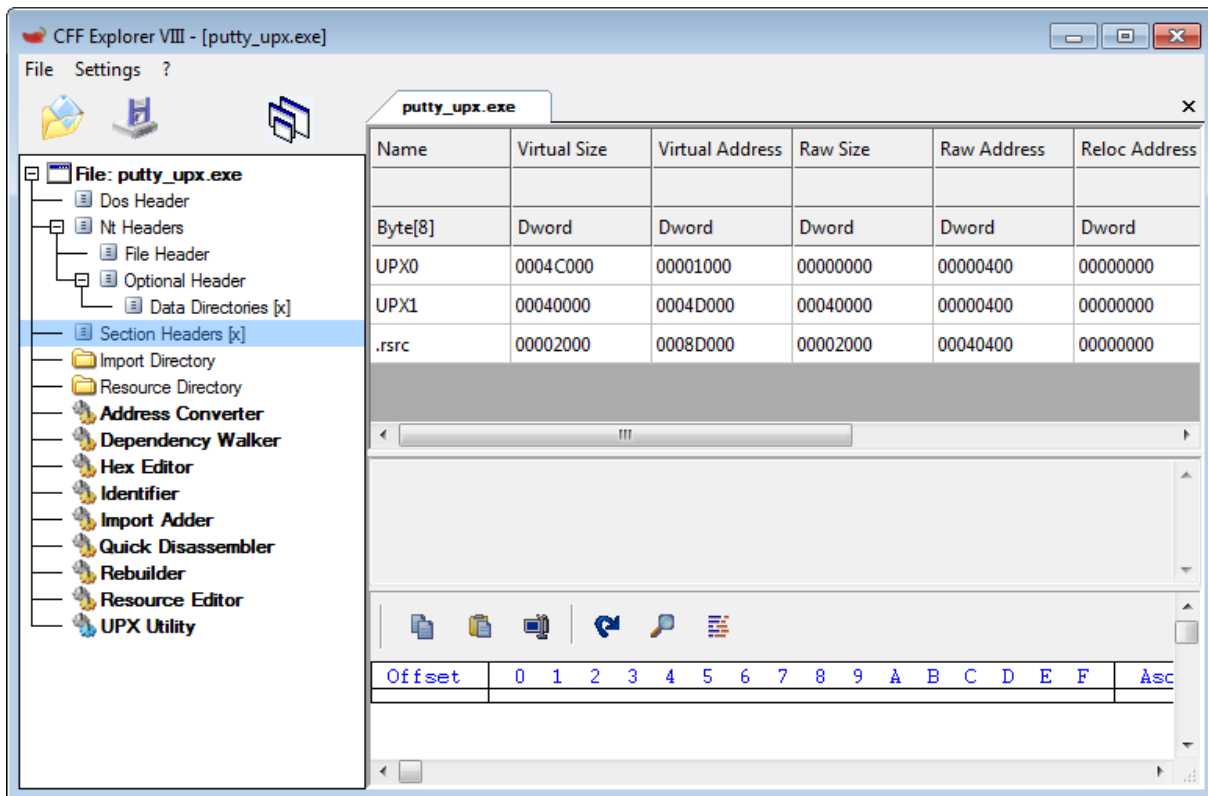
In this case, PEiD reveals that putty_upx.exe sample was packed with UPX as seen in the following screenshot.



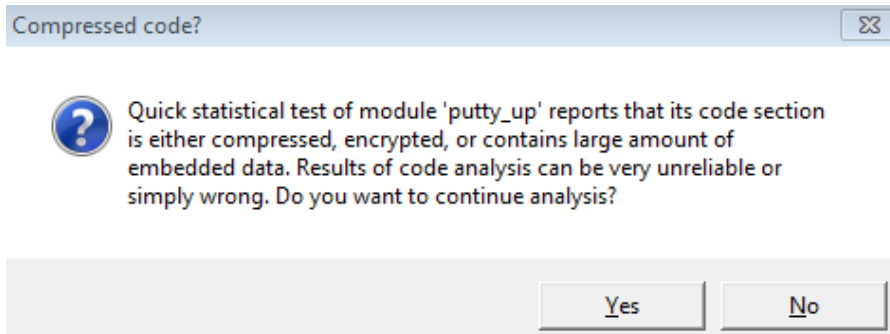
To confirm the output of PEiD and to identify packing more specifically, CFF Explorer⁶ is used.

⁵ PeiD <http://www.aldeid.com/wiki/PEiD> (last accessed 11.09.2015)

⁶ CFF Explorer suite <http://www.ntcore.com/exsuite.php> (last accessed 11.09.2015)



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".



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

Address	Hex dump	ASCII
00401000	00 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 00
00401020	00 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 00
00401040	00 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 00
00401060	00 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 00
00401080	00 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 00
004010A0	00 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 00
004010C0	00 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 00
004010E0	00 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 00
00401100	00 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 00
00401120	00 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 00
00401140	00 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 00
00401160	00 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 00
00401180	00 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 00

Then press and hold for a few seconds the Step Over key (F8).

Address	Hex dump	ASCII
00401000	56 6A 0C 6A 01 E8 34 00 AB EA 8B F0 8B 44 24 10	Uj.j0\$4.%ΩI=ID\$▶
00401010	89 46 04 8B 44 24 14 89 46 08 A1 68 14 48 00 85	eF♦ID\$9eF□lh9H.à
00401020	C0 59 59 74 12 83 30 60 14 48 00 00 75 09 FF 35	4Yvt♣r="9H..u.è
00401030	6C 14 48 00 FF 00 59 A1 64 14 48 00 85 C0 74 04	l9H. 4Yid9H.à4t♣
00401040	89 30 EB 06 89 35 60 14 48 00 89 35 64 14 48 00	è0\$49e0\$"9H.e5d9H
00401050	00 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 00
00401070	00 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 00
00401090	00 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	
0048CEB4	83C3 04	ADD EBX,4	putty_up.0048EDE4
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-100]	

Put breakpoint at *0x48CEBF* (outside IAT reconstruction loop) and press F9 a few times (5-10).

0012FF64	75A10000	kernel32.75A10000
0012FF68	0048A009	ASCII "SetEnvironmentVariable"
0012FF6C	00000000	
0012FF70	00000000	
0012FF74	0012FF94	
0012FF78	0012FF9C	
0012FF64	75A10000	kernel32.75A10000
0012FF68	0048A04C	ASCII "SetEndOfFile"
0012FF6C	00000000	
0012FF70	00000000	

Address	Value	Comment
00461FF0	00000000	
00461FF4	00000000	
00461FF8	00000000	
00461FFC	00000000	
00462000	7598BED4	ADVAPI32.RegCloseKey
00462004	7598BC25	ADVAPI32.RegQueryValueExA
00462008	7597D2ED	ADVAPI32.RegOpenKeyA
0046200C	7597E504	ADVAPI32.GetUserNamesA
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	73E1E09C	COMCTL32.MakeDroplist

Next scroll down the assembly code until the characteristic JMP instruction at *0x48CEFC*. Put a breakpoint at this instruction and resume the execution (F9).

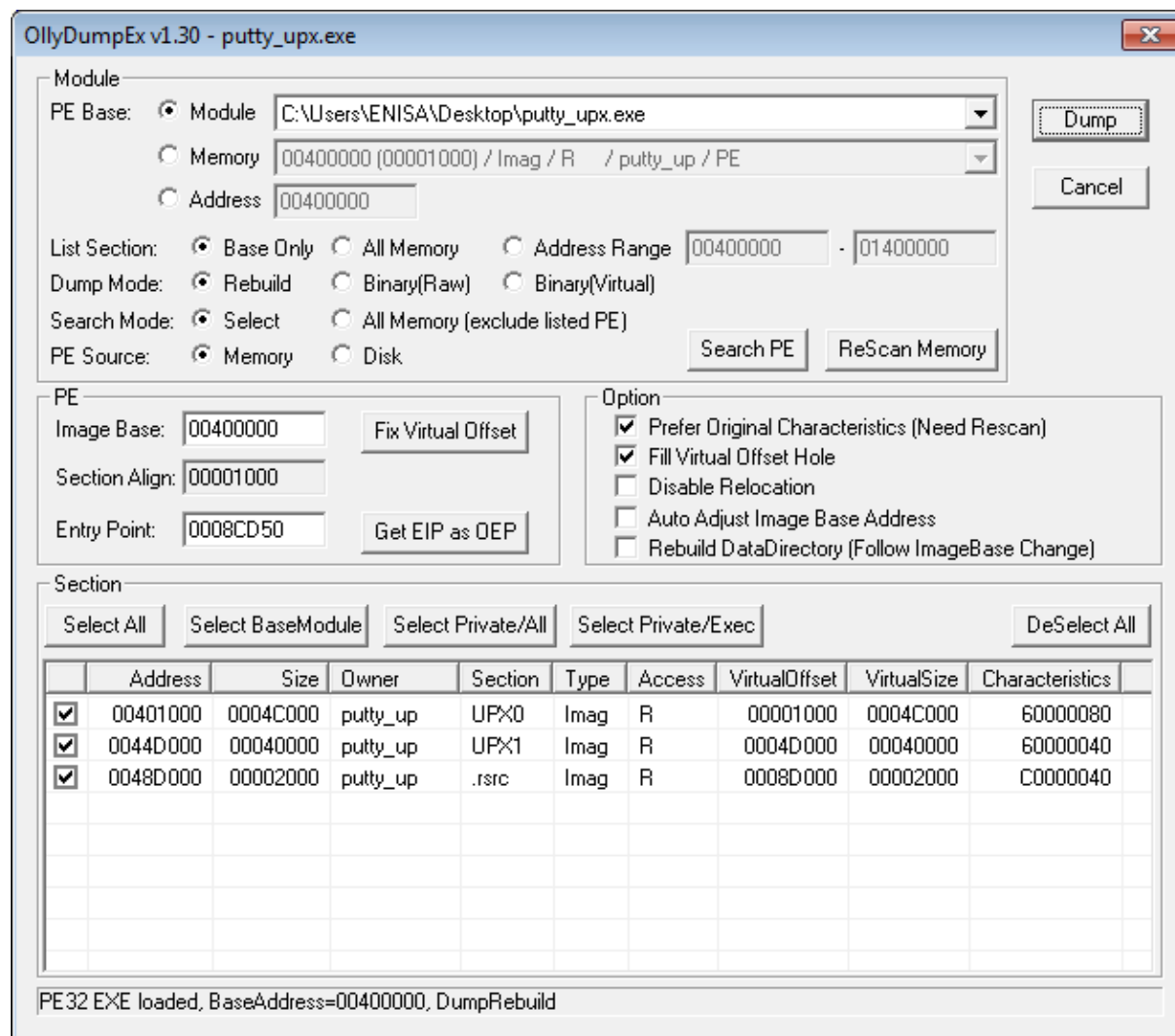
0048CEC8	57	PUSH EDI	
0048CEEB	FFD5	CALL EBX	
0048CEED	58	POP EAX	kernel32.75A61174
0048CEEE	61	POPAD	
0048CEEF	8D4424 80	LEA EAX, DWORD PTR SS:[ESP-80]	
0048CEF3	6A 00	PUSH 0	
0048CEF5	39C4	CMP ESP, EAX	
0048CEF7	75 FA	JNZ SHORT putty_up.0048CEF3	
0048CEF9	83EC 80	SUB ESP, -80	
0048CEFC	E9 DF0FCFF	JMP putty_up.00459FE0	
0048CF01	0000	ADD BYTE PTR DS:[EAX], AL	
0048CF03	0048 00	ADD BYTE PTR DS:[EAX], CL	
0048CF06	0000	ADD BYTE PTR DS:[EAX], AL	
0048CF08	0000	ADD BYTE PTR DS:[EAX], AL	
0048CF0A	0000	ADD BYTE PTR DS:[EAX], AL	
0048CF0C	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.

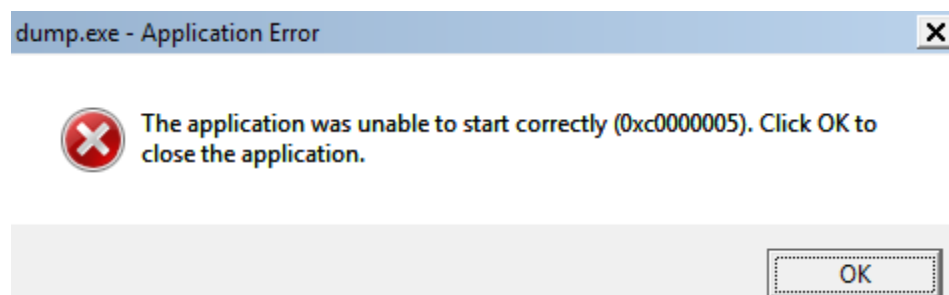
00459FE0	6A 60	PUSH 60	
00459FE2	68 E0DA4700	PUSH putty_up.0047DAE0	
00459FE7	E8 08210000	CALL putty_up.0045C0F4	
00459FEC	BF 94000000	MOV EDI, 94	
00459FF1	8BC7	MOV EAX, EDI	
00459FF3	E8 D8F9FFFF	CALL putty_up.004599D0	
00459FF8	8965 E8	MOV DWORD PTR SS:[EBP-18], ESP	
00459FFB	8BF4	MOV ESI, ESP	
00459FFD	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	

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.

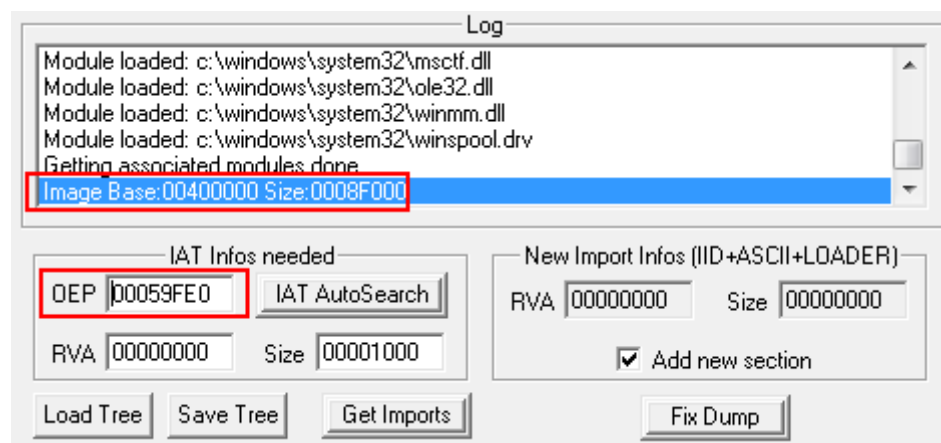
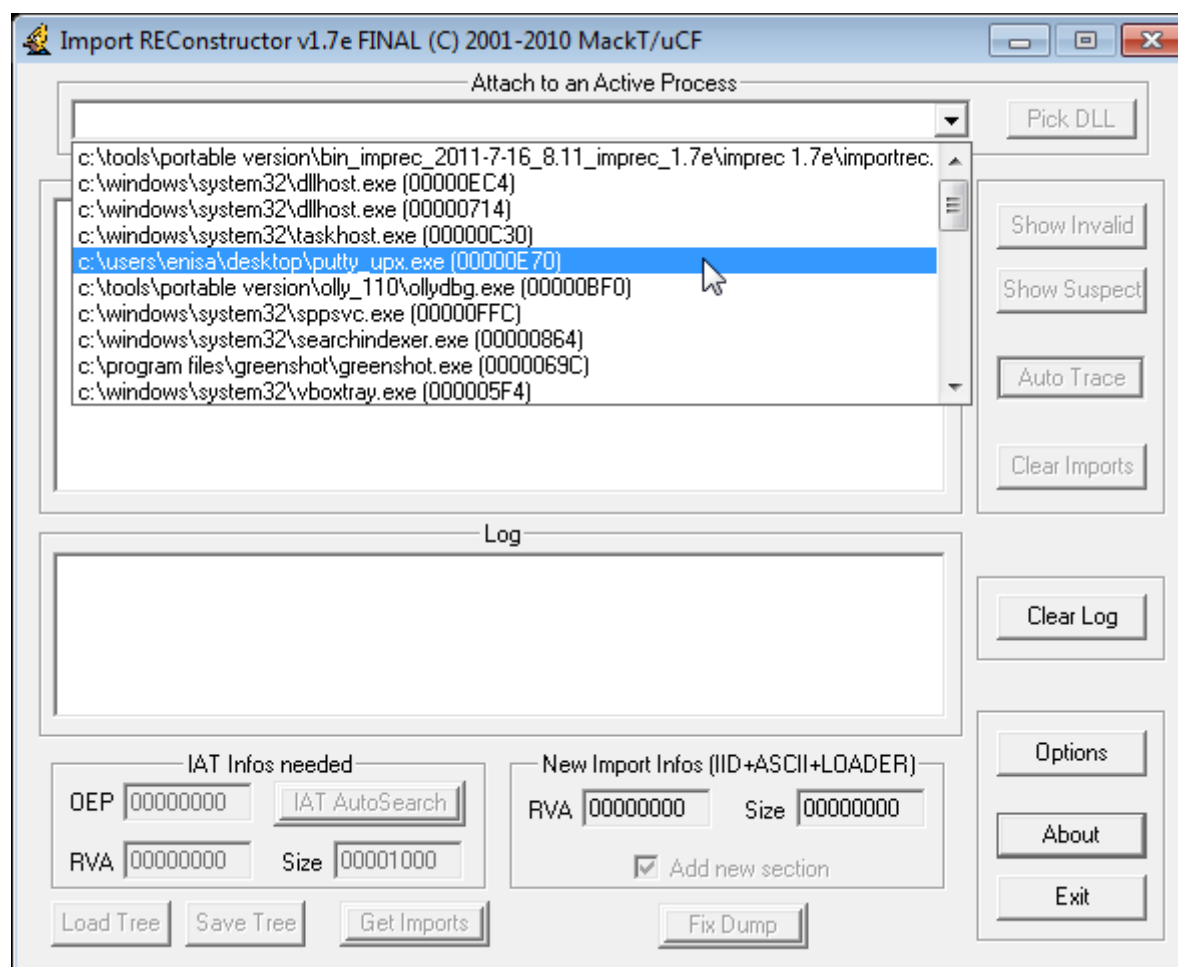
When dumping unpacked putty.exe code use the default OllyDumpEx settings. Save dumped process as dump.exe. Don't close OllyDbg yet.



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.

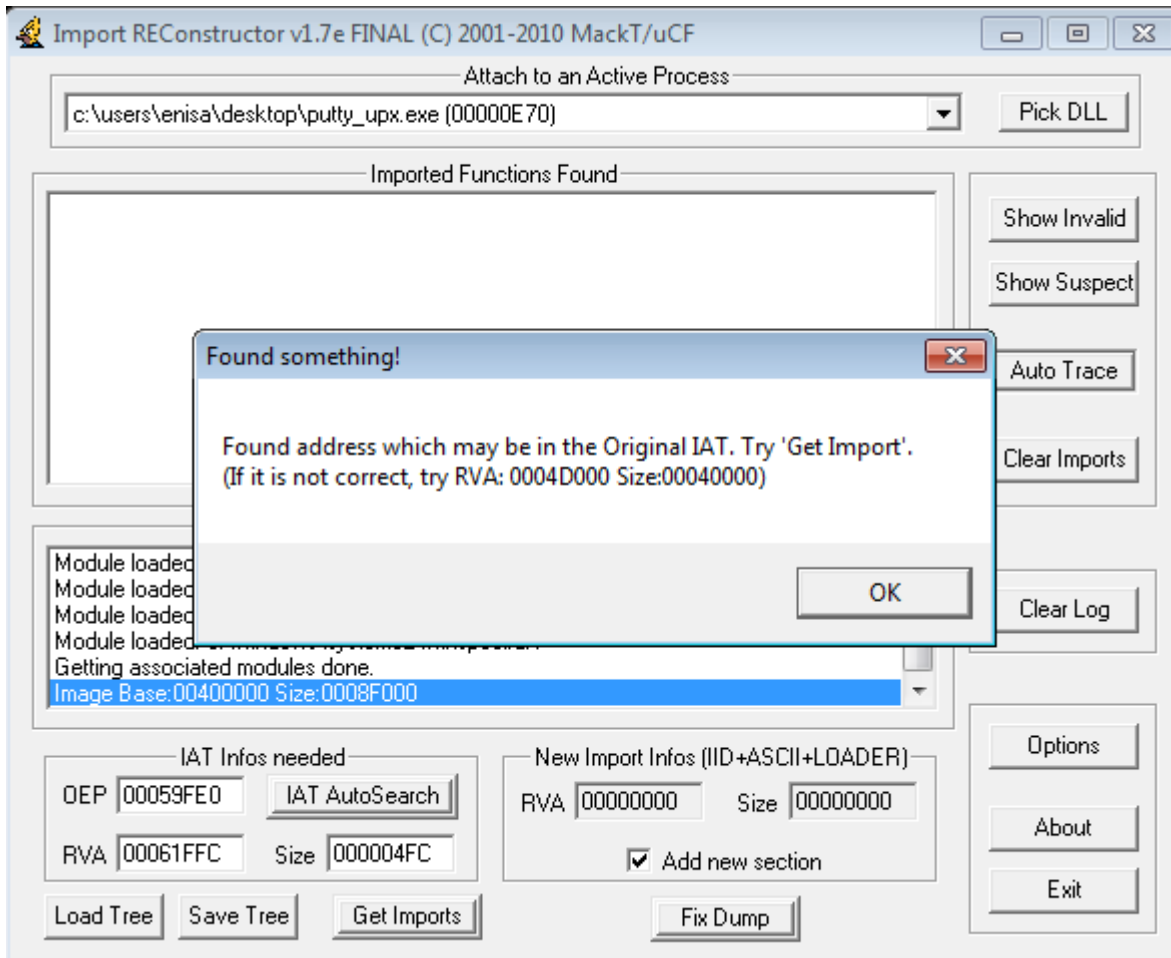


To reconstruct IAT you will use the ImpREC⁷ tool. Run ImpREC (as Administrator) and from the scroll down menu at the top of the window choose the putty_upx.exe process.

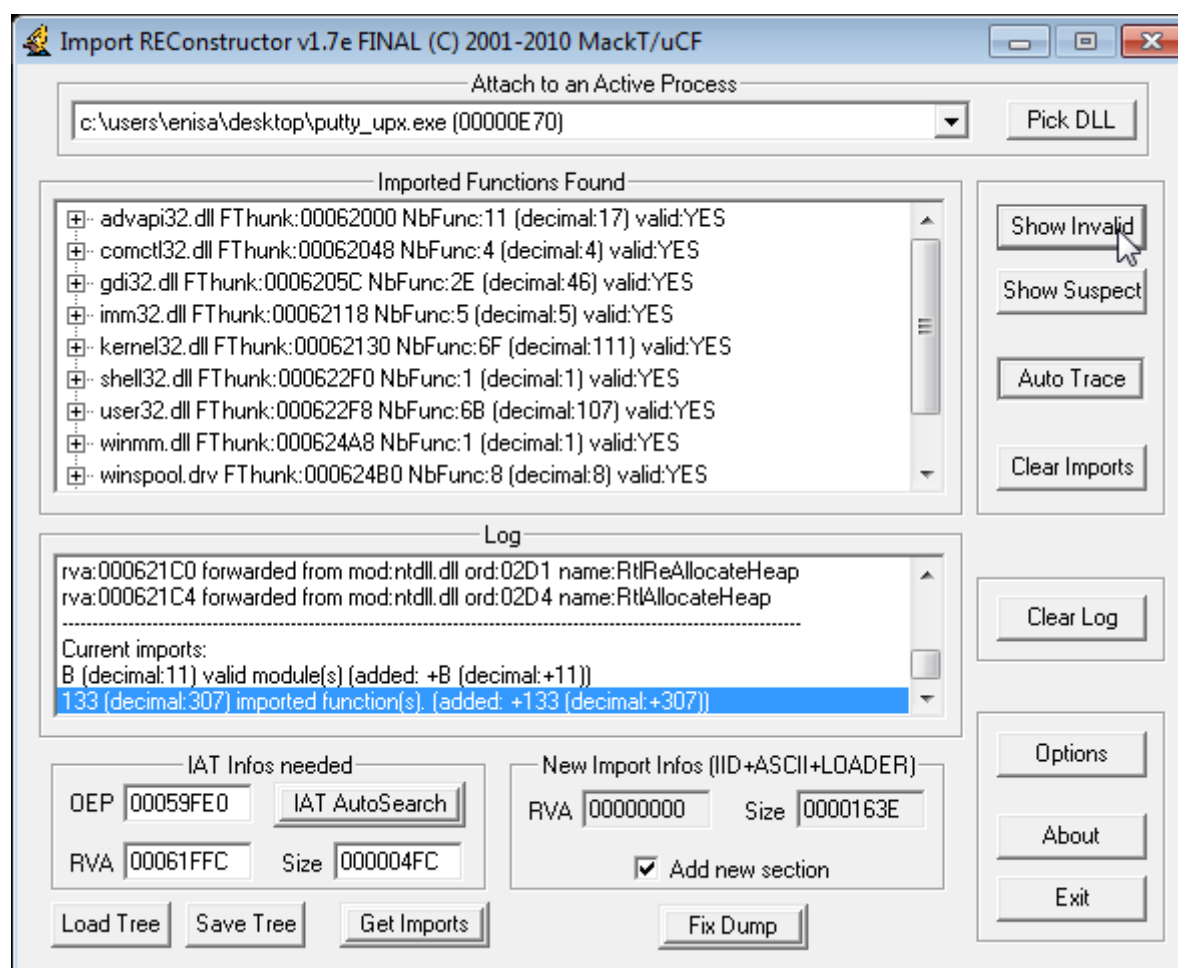


⁷ImpREC <http://www.woodmann.com/collaborative/tools/index.php/ImpREC> (last accessed 10.10.2015)

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”.

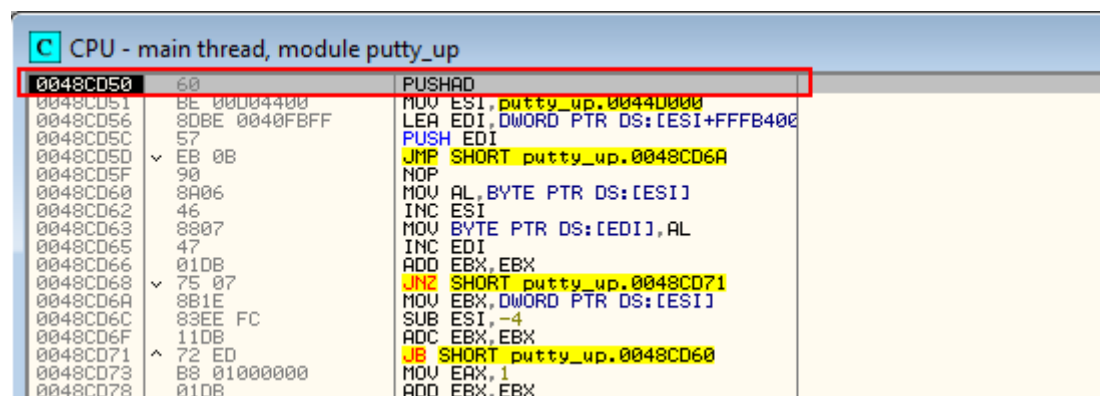


Click “Show Invalid” to see if there are any invalid functions. In this case there shouldn’t be any. Click “Fix Dump” and select *dump.exe* file. If everything goes right you should see a message that *dump.exe* was saved successfully (please note underscore in the name of the file name, the originale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

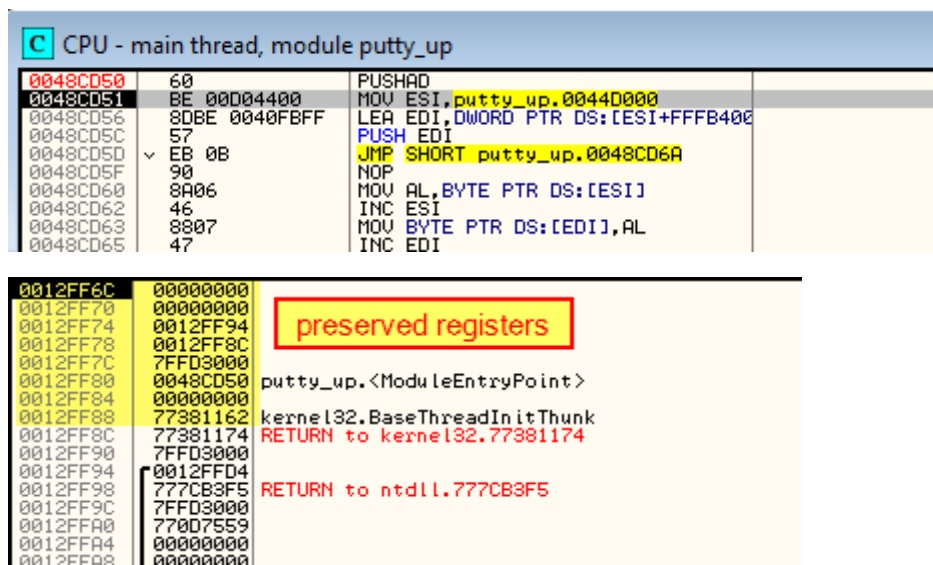
You can try to run *dump.exe* to check if it runs.

3.2 Unpacking UPX with ESP trick

Open *putty_upx.exe* in OllyDbg.



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



The screenshot shows the CPU window for the main thread in the putty_up module. The assembly view shows the following instructions:

```

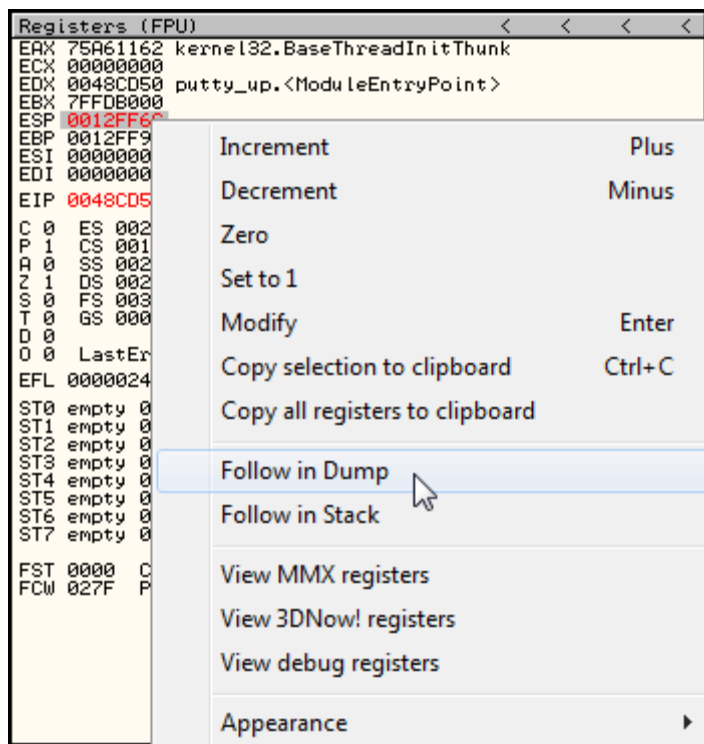
0048CD50 60 PUSHAD
0048CD51 BE 00004400 MOV ESI,putty_up.00440000
0048CD56 8DBE 0040FBFF LEA EDI,DWORD PTR DS:[ESI+FFFB4000]
0048CD5C 57 PUSH EDI
0048CD5D EB 0B JMP SHORT putty_up.0048CD6A
0048CD5F 90 NOP
0048CD60 8A06 MOV AL,BYTE PTR DS:[ESI]
0048CD62 46 INC ESI
0048CD63 8807 MOV BYTE PTR DS:[EDI],AL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

Below the assembly view is a stack dump. A red box highlights the address 0012FF6C, which is labeled as "preserved registers". The stack dump shows the following entries:

```

0012FF6C 00000000
0012FF70 00000000
0012FF74 0012FF94
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.



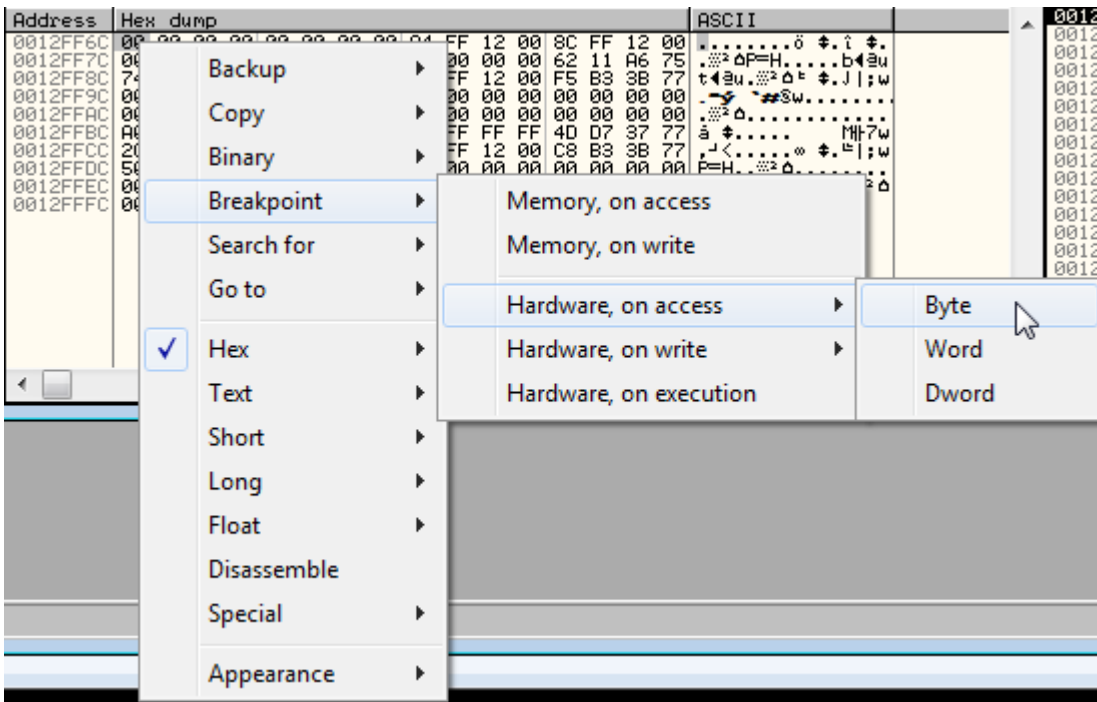
The screenshot shows the Registers (FPU) window. The ESP register is highlighted in red and has a context menu open over it. The context menu options are:

- Increment Plus
- Decrement Minus
- Zero
- Set to 1
- Modify Enter
- Copy selection to clipboard Ctrl+C
- Copy all registers to clipboard
- Follow in Dump (highlighted by the mouse)
- Follow in Stack
- View MMX registers
- View 3DNow! registers
- View debug registers
- Appearance

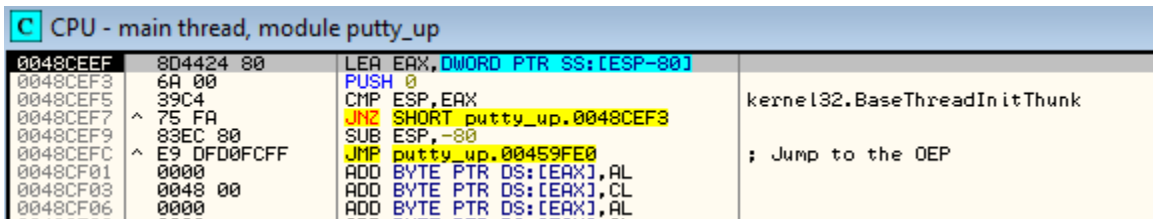
The registers window also shows the following values:

```

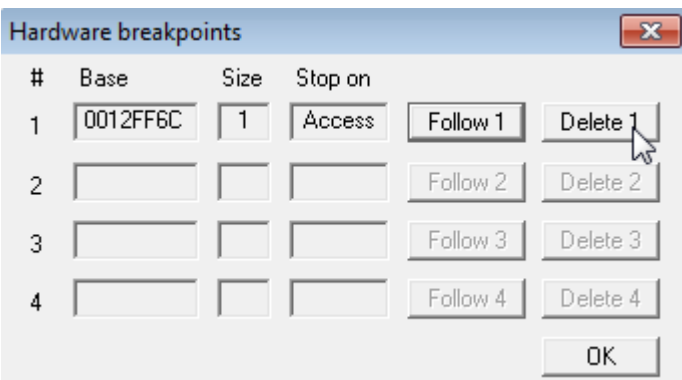
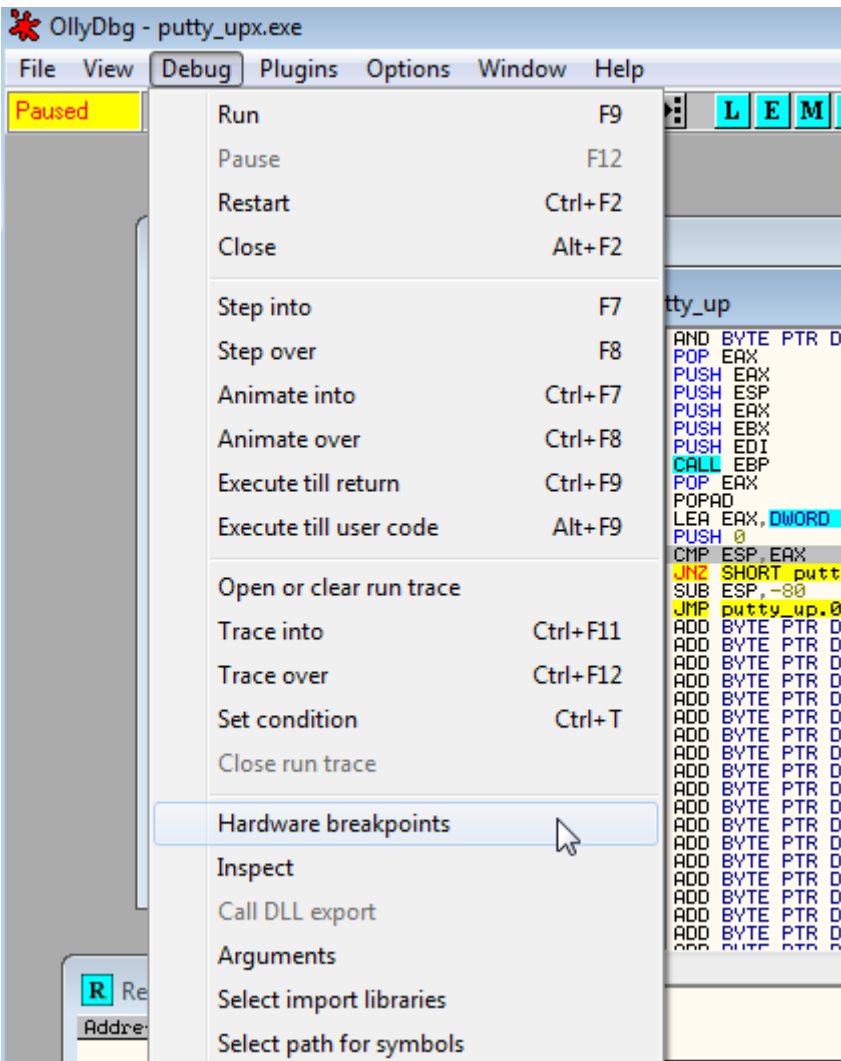
EAX 75A61162 kernel32.BaseThreadInitThunk
ECX 00000000
EDX 0048CD50 putty_up.<ModuleEntryPoint>
EBX 7FFDB000
ESP 0012FF6C
EBP 0012FF99
ESI 00000000
EDI 00000000
EIP 0048CD50
C 0 ES 002
P 1 CS 001
A 0 SS 002
Z 1 DS 002
S 0 FS 003
T 0 GS 000
D 0
O 0 LastEr
EFL 0000024
ST0 empty 0
ST1 empty 0
ST2 empty 0
ST3 empty 0
ST4 empty 0
ST5 empty 0
ST6 empty 0
ST7 empty 0
FST 0000 C
FCW 027F P
  
```



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



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.

3.3 Unpacking Dyre sample

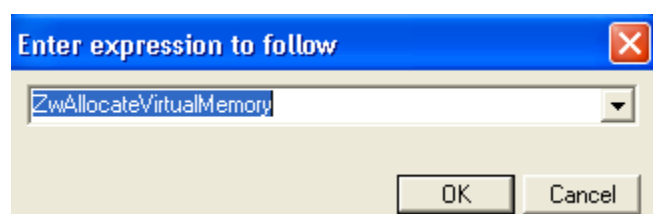
In this exercise the unpacking of the Dyre⁸ malware will be presented. Dyre is a banking trojan and was packed using a more complex packer than UPX. Since it is a live malware sample, run it only in a controlled virtual environment and after the analysis restore a clean snapshot of the virtual machine. 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          ntdll.7C910208
00401B4A | | 55         PUSH EBP
00401B4B | | 8BEC        MOV EBP,ESP
00401B4D | | 81EC 28030000 SUB ESP,328
  
```

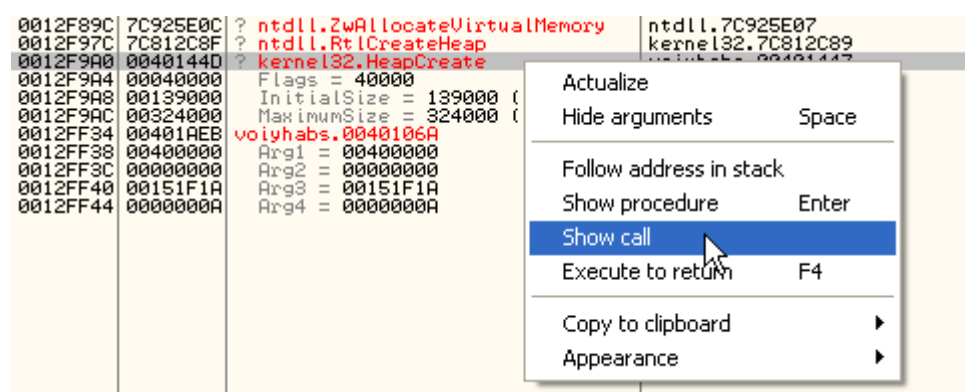
Put a breakpoint on *ZwAllocateVirtualMemory* (as described in the introduction to OllyDbg).



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.

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.0040106A	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.



⁸Dyre: 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)

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
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
00401451 | . 3BC3       CMP  EAX, EBX

```

```

MaximumSize = 324000 (3293184.)
InitialSize = 139000 (1282048.)
Flags = HEAP_CREATE_ENABLE_TRAC
HeapCreate

```

```

Registers (FPU)
EAX 00DE0000
ECX 7C926090 ntdll.7C926090
EDX 7C97B380 ntdll.7C97B380
EBX 00000000
ESP 0012F9B0
EBP 0012FF30
ESI 00139000
EDI 00324000
EIP 0040144D voiyhabs.0040144D

```

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
004018C5 | . 899C24 80000000 MOV  DWORD PTR SS:[ESP+80], EBX
004018CC | . C78424 84000000 MOV  DWORD PTR SS:[ESP+84], voiyhabs
004018D7 | . 899C24 88000000 MOV  DWORD PTR SS:[ESP+88], EBX
004018DE | . FF15 5C714000 CALL DWORD PTR DS:[&USER32.Reg
004018E4 | . 53          PUSH EBX
004018E5 | . 53          PUSH EBX
004018E6 | . 53          PUSH EBX
004018E7 | . 68 E8030000 PUSH 3E8
004018EC | . FF7424 48   PUSH DWORD PTR SS:[ESP+48]
004018F0 | . FF15 58714000 CALL DWORD PTR DS:[&USER32.Crea
004018F6 | . 6A 05       PUSH 5
004018F8 | . 50          PUSH EAX
004018F9 | . FF15 84714000 CALL DWORD PTR DS:[&USER32.Show

```

```

pWndClassEx = 0012FA08
ASCII "MY_EXCLUSIVE_CLASS"
RegisterClassExA
lParam = 0
pDlgProc = NULL
hOwner = NULL
pTemplate = 3E8
hInst = 02100210
CreateDialogParamA
ShowState = SW_SHOW
hWnd = 0012FA08
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.

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*).

00EDFBD8	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 until a group of three calls just before function return. 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 0AE1FFFF	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	

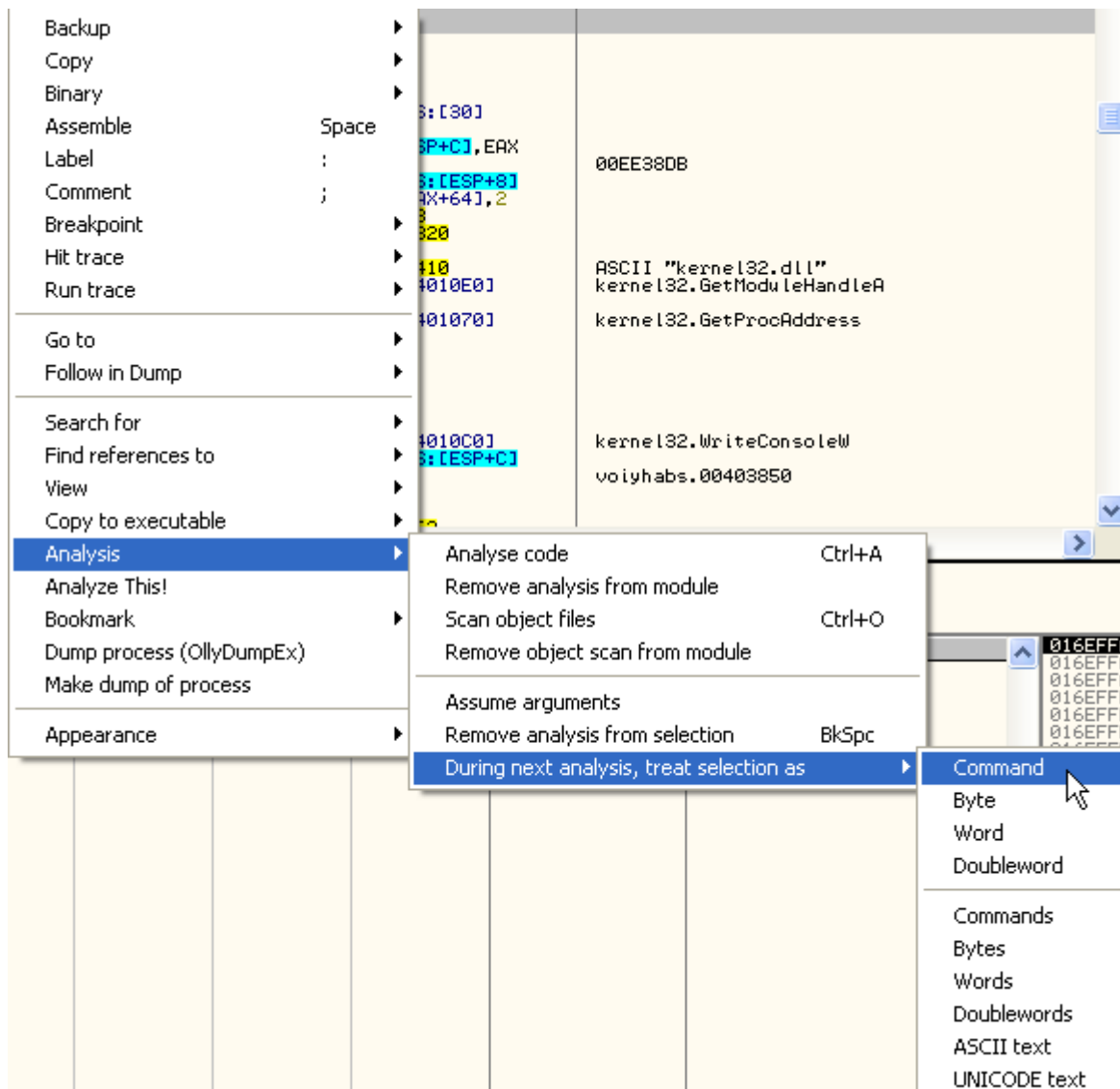
When a breakpoint at the thread function is hit step over (F8) until a call to ECX. Step into the call (F7).

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	

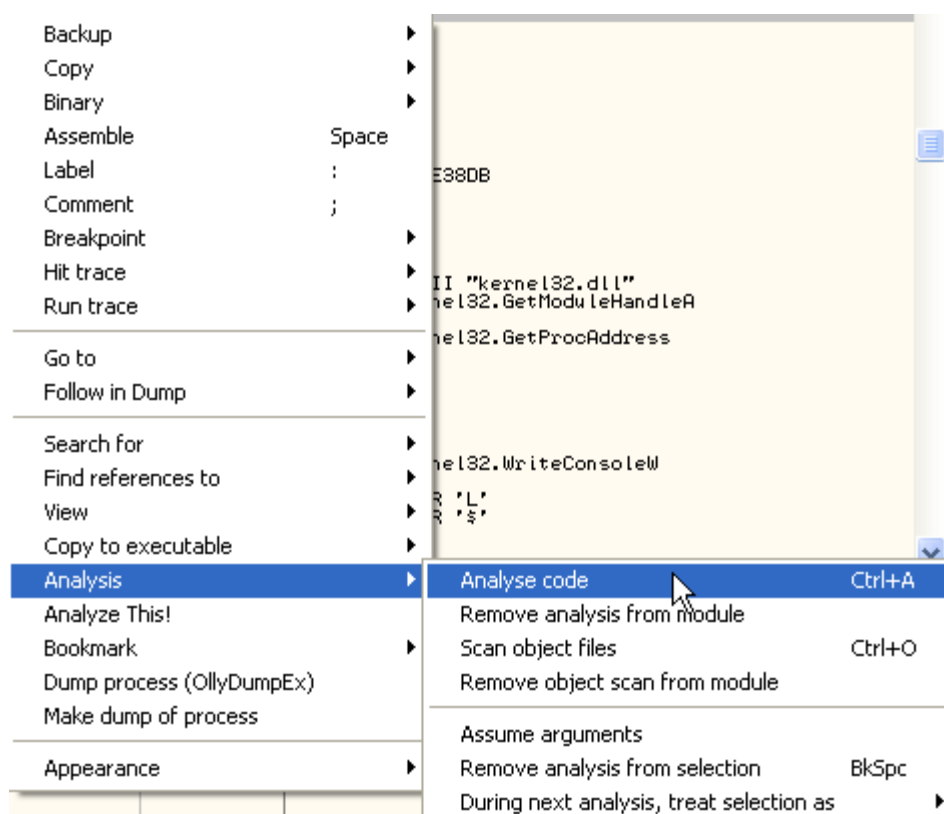
You should land at the OEP with overwritten code which OllyDbg hasn't analysed properly.

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'.

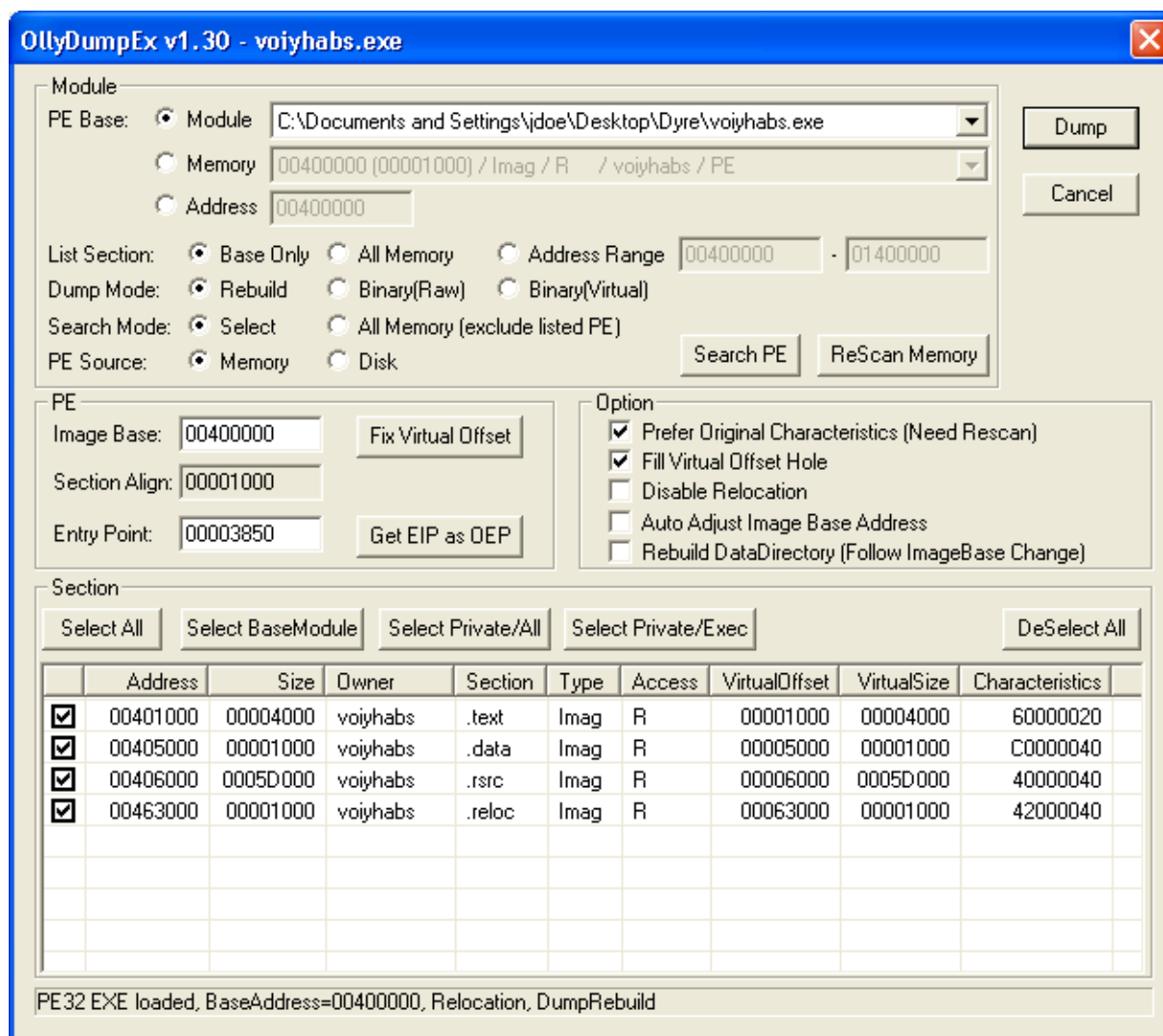


Now click on them once again and from the context menu choose 'Analysis->Analyse code'.

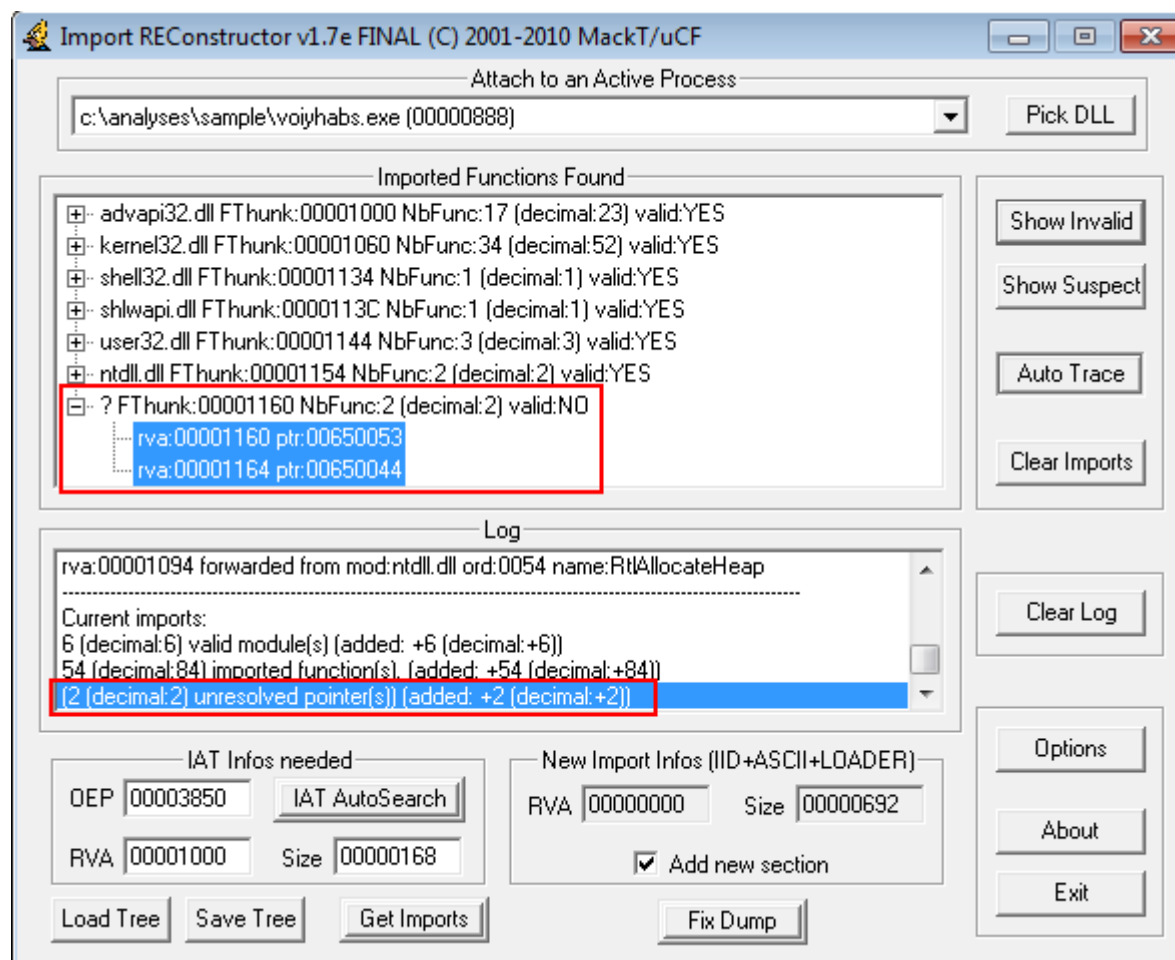


0040384F	CC	INT3	
00403850	55	PUSH EBP	
00403851	8BEC	MOV EBP,ESP	
00403853	83E4 F8	AND ESP,FFFFFFF8	
00403856	83EC 2C	SUB ESP,2C	
00403859	56	PUSH ESI	
0040385A	50	PUSH EAX	
0040385B	64:A1 30000000	MOV EAX,DWORD PTR FS:[30]	
00403861	85DB	TEST EBX,EBX	
00403863	894424 0C	MOV DWORD PTR SS:[ESP+C],EAX	
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 ED000000	JB voiyhabs.00403963	
00403876	E8 A5FFFFFF	CALL voiyhabs.00403820	
0040387B	50	PUSH EAX	
0040387C	68 10140000	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 thanks(s)*. After all invalid pointers are resolved, use *Fix dump* to fix the dumped executable.



The screenshot shows the 'Import REConstructor' application interface. At the top, it is titled 'Import REConstructor v1.7e FINAL (C) 2001-2010 MackT/uCF'. Below the title bar, there is a section for 'Attach to an Active Process' with a dropdown menu showing 'c:\analyses\sample\woiyhabs.exe (00000888)' and a 'Pick DLL' button.

The main area is divided into several sections:

- Imported Functions Found:** A list of imported functions. The last entry is highlighted with a red box: '? FTThunk:00001160 NbFunc:2 (decimal:2) valid:NO'. Below it, two pointers are listed: 'rva:00001160 ptr:00650053' and 'rva:00001164 ptr:00650044'.
- Log:** A text area showing log entries. The last entry is highlighted with a red box: '[2 (decimal:2) unresolved pointer(s)] (added: +2 (decimal:+2))'.
- IAT Infos needed:** Fields for OEP (00003850), RVA (00001000), and Size (00000168). There is an 'IAT AutoSearch' button.
- New Import Infos (IID+ASCII+LOADER):** Fields for RVA (00000000) and Size (00000692). There is a checked 'Add new section' checkbox.

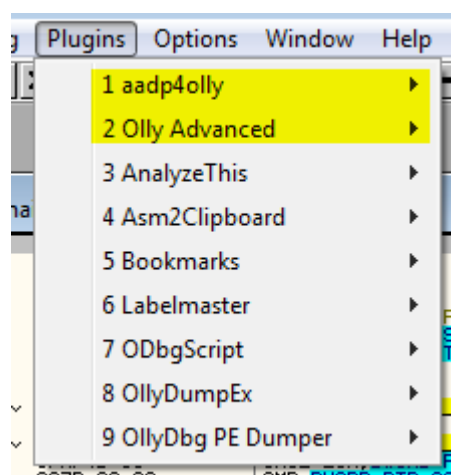
On the right side, there are several buttons: 'Show Invalid', 'Show Suspect', 'Auto Trace', 'Clear Imports', 'Clear Log', 'Options', 'About', and 'Exit'. At the bottom, there are buttons for 'Load Tree', 'Save Tree', 'Get Imports', and 'Fix Dump'.

4. Anti-debugging techniques

4.1 Anti-debugging and anti-analysis techniques

One of the countermeasures for anti-debugging is to use special plugins for OllyDbg like *aadp4olly*⁹ and *Olly Advanced*¹⁰.

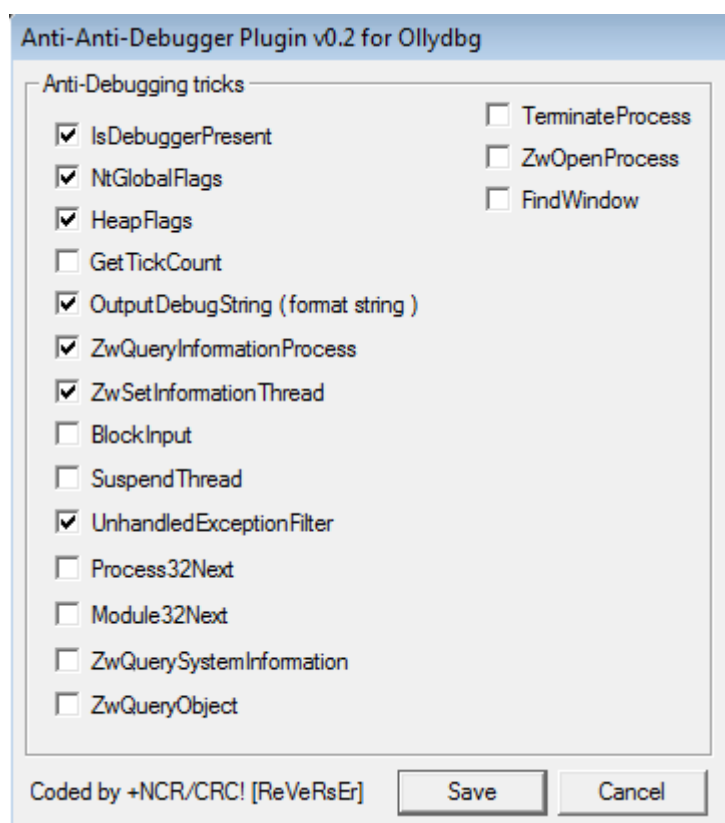
When using those plugins you need to check which anti-anti-debugging techniques should be used. You can do this by accessing the plugin's options dialog via *Plugins* menu.



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

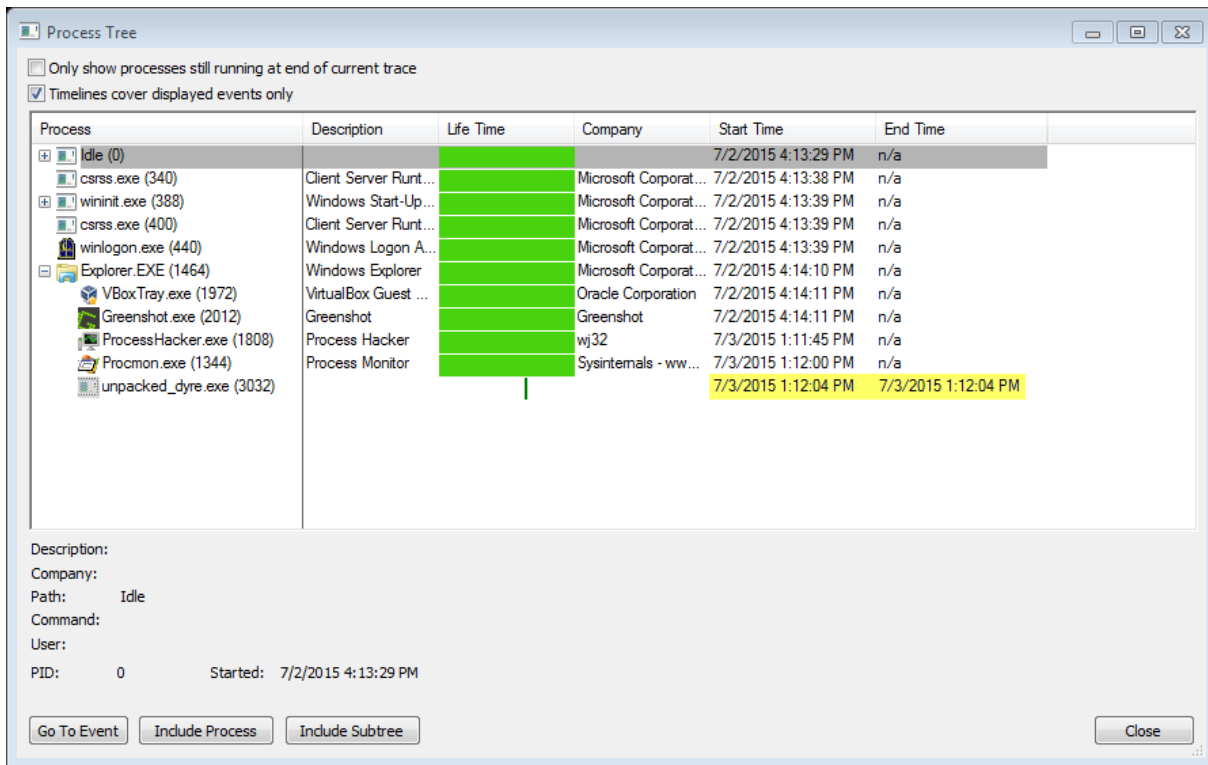
⁹aadp4olly <https://tuts4you.com/download.php?view.3021> (last accessed 11.09.2015)

¹⁰Olly Advanced <https://tuts4you.com/download.php?view.75> (last accessed 11.09.2015)



4.2 Dyre - basic patching with OllyDbg

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 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.

```

01243850 | $ 55          PUSH EBP
01243851 | . 8BEC        MOV EBP,ESP
01243853 | . 83E4 F8     AND ESP,FFFFFFF8
01243856 | . 83EC 2C     SUB ESP,2C
01243859 | . 56          PUSH ESI
0124385A | . 50          PUSH EAX
0124385B | . 64:A1 30000000 MOV EAX,DWORD PTR FS:[30]
01243861 | . 85DB        TEST EBX,EBX
01243863 | . 894424 0C   MOV DWORD PTR SS:[ESP+C],EAX
01243867 | . 58          POP EAX
01243868 | . 8B4424 08   MOV EAX,DWORD PTR SS:[ESP+8]
0124386C | . 8378 64 02  CMP DWORD PTR DS:[EAX+64],2
01243870 | . 0F82 ED000000 JB unpacked.01243963
01243876 | . E8 A5FFFFFF CALL unpacked.01243820
0124387B | . 50          PUSH EAX
0124387C | . 68 10142401 PUSH unpacked.01241410
01243881 | . FF15 E0102401 CALL DWORD PTR DS:[<&KERNEL32.Ge
01243887 | . 50          PUSH EAX
01243888 | . FF15 70102401 CALL DWORD PTR DS:[<&KERNEL32.Ge
0124388E | . 6A 00       PUSH 0
01243890 | . 6A 00       PUSH 0
01243892 | . 6A 00       PUSH 0
01243894 | . 6A 00       PUSH 0
01243896 | . 6A 00       PUSH 0
01243898 | . 8BF0        MOV ESI,EAX
0124389A | . FF15 C0102401 CALL DWORD PTR DS:[<&KERNEL32.Wr
012438A0 | . 8D4C24 0C   LEA ECX,DWORD PTR SS:[ESP+C]
012438A4 | . 51          PUSH ECX
012438A6 | . FF76        CALL EB6
  
```

```

kernel32.BaseThreadInitThunk
kernel32.BaseThreadInitThunk
kernel32.75F51174
ProcNameOrOrdinal = "\x8B\xFF"
pModule = "kernel32.dll"
GetModuleHandleA
hModule = 75F51162
GetProcAddress
pReserved = NULL
pWritten = NULL
CharsToWrite = 0
Buffer = NULL
hConsole = NULL
kernel32.BaseThreadInitThunk
WriteConsoleW
  
```

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 = ""
pModule = "kernel32.dll"
  
```



```

002E3957 | . | E8 74F8FFFF | CALL unpacked.002E31D0
002E395C | . | EB 05 | JMP SHORT unpacked.002E3963
002E395E | . | E8 80FAFFFF | CALL unpacked.002E33F0
002E3963 | . | 6A 00 | PUSH 0
002E3965 | . | FF15 88102E00 | CALL DWORD PTR DS:[<&KERNEL32.Ex
002E3968 | . | CC | INT3
002E396C | . | CC | INT3
002E396D | . | CC | INT3
002E396E | . | CC | INT3
  
```

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

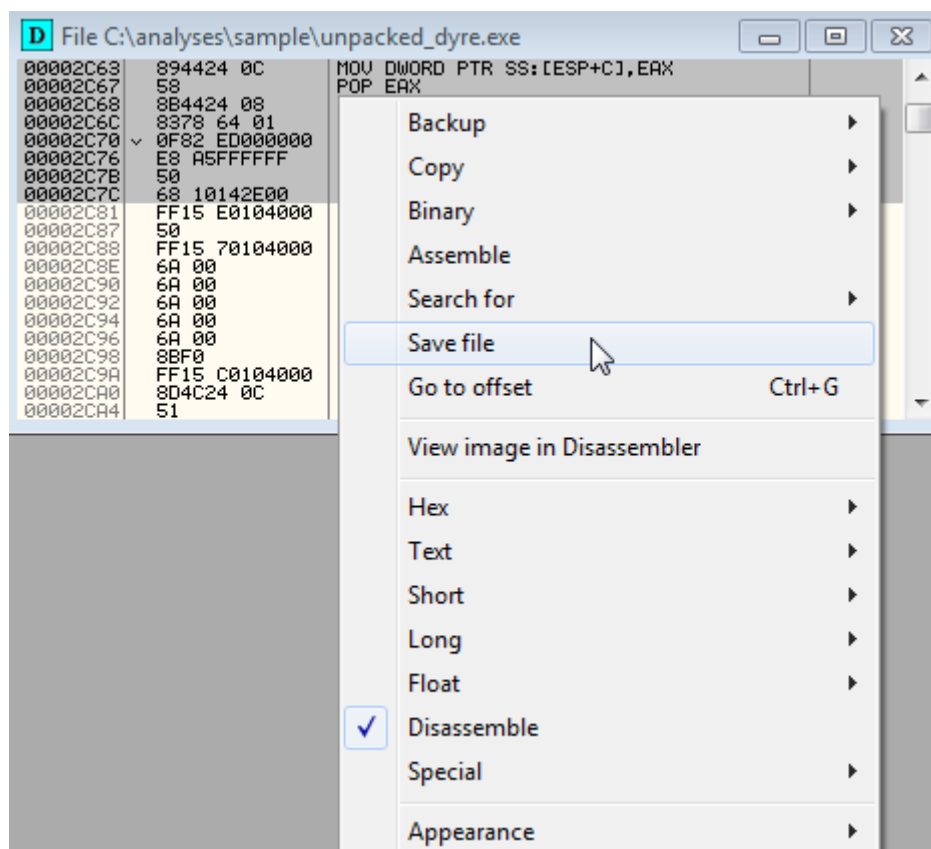
```

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.

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



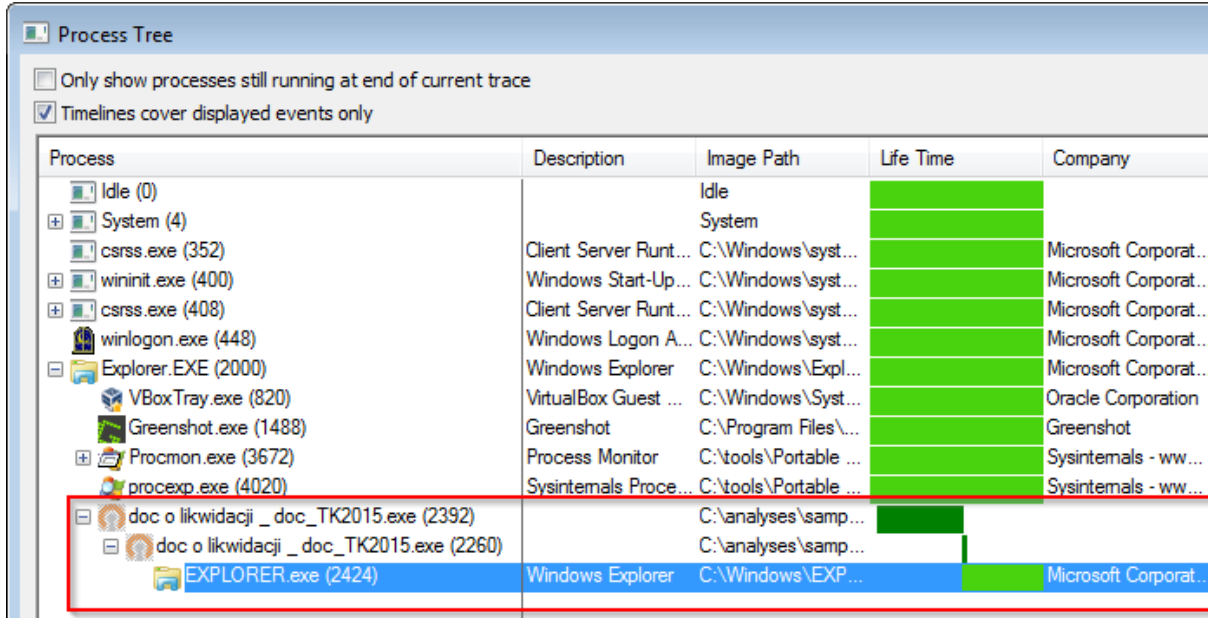
Now try running the patched executable while observing its behaviour in Process Explorer or Process Monitor (process tree).

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
proccxp.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.

5. Process creation and injection

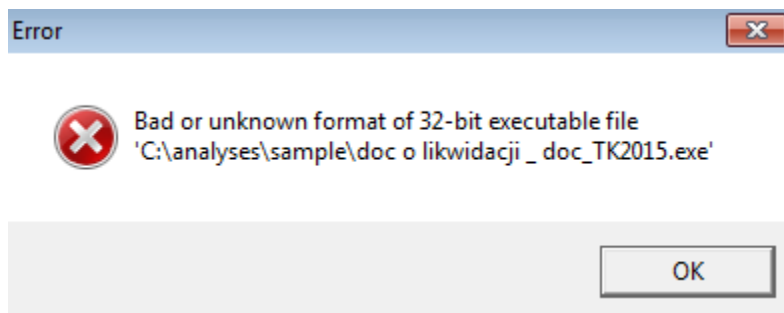
5.1 Following child processes of Tinba loader



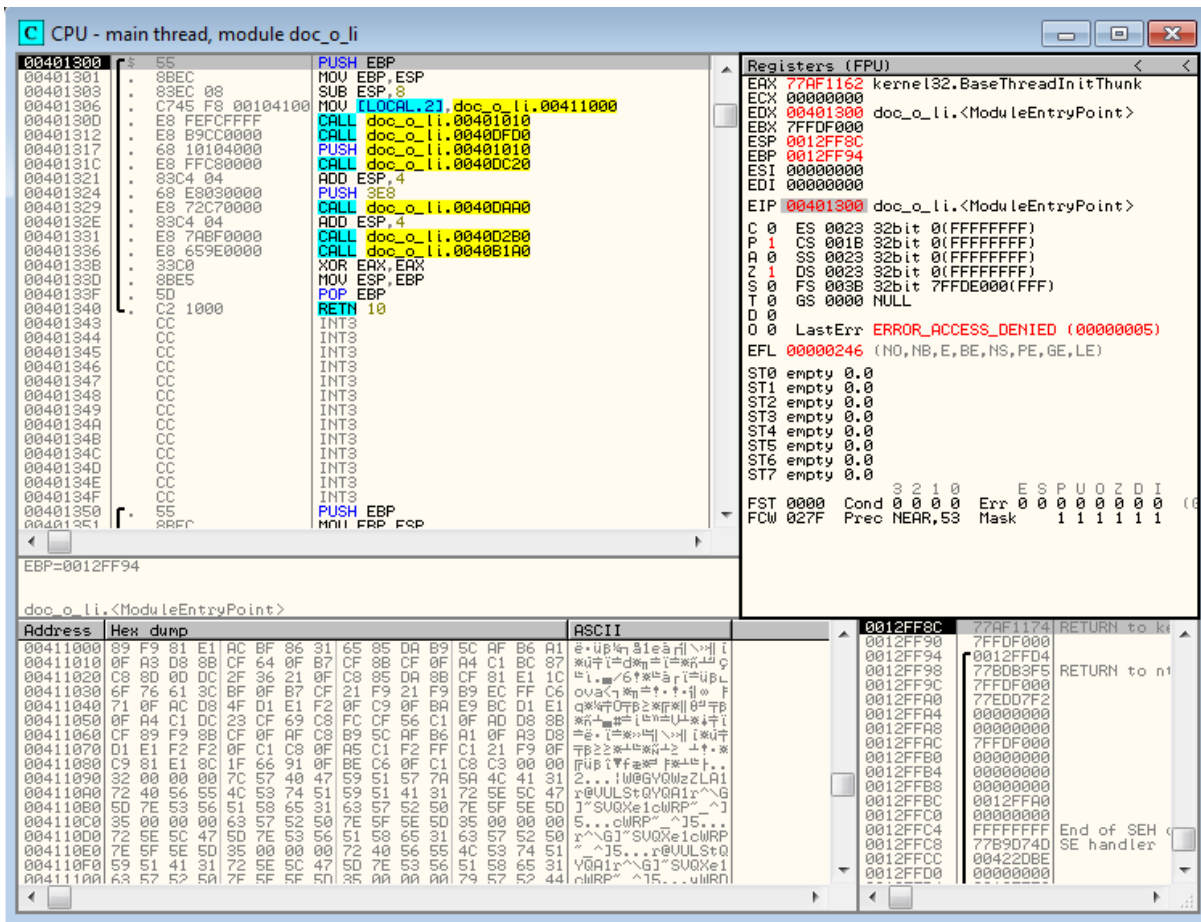
Process	Description	Image Path	Life Time	Company
Idle (0)	Idle			
System (4)	System			
csrss.exe (352)	Client Server Runt...	C:\Windows\sys...		Microsoft Corporat...
wininit.exe (400)	Windows Start-Up...	C:\Windows\sys...		Microsoft Corporat...
csrss.exe (408)	Client Server Runt...	C:\Windows\sys...		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...

5.1.1 First stage

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 instructions at memory addresses from 00401300 to 00401351. Key instructions include `PUSH EBP`, `MOV EBP, ESP`, `CALL doc_o_li.00401010`, `CALL doc_o_li.00400F00`, `CALL doc_o_li.00400C20`, `CALL doc_o_li.00400AA0`, `CALL doc_o_li.004002B0`, `CALL doc_o_li.00400B1A0`, `XOR EAX, EAX`, `MOV ESP, EBP`, `POP EBP`, and `RETN 10`.
- Registers (FPU):** Shows the state of 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)`.
- Hex Dump:** Shows memory addresses from 00411000 to 0041110A with their corresponding hex values and ASCII representations.

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.

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.77BC6BFD
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`¹¹) 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 00C.....
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	PE A.+.=?@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	t be run in DOS
02130070	6D 6F 64 65 2E 00 0A 24 00 00 00 00 00 00 00 00	mode...\$.
02130080	50 45 00 00 4C 01 01 00 E0 AE 0D 55 00 00 00 00	PE..L00.××.U....
02130090	00 00 00 00 E0 00 0F 01 0B 01 01 47 00 36 00 00×.0000G.6..
021300A0	00 00 00 00 00 00 00 00 00 00 10 00 00 00 10 00>.....>
021300B0	00 00 00 00 00 00 00 40 00 00 10 00 00 00 02 00e.▶.....@.
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 00 20 00 00
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

Resume the execution (F9) two times until you break on `SetThreadContext` function. 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.77BC6BFD

```

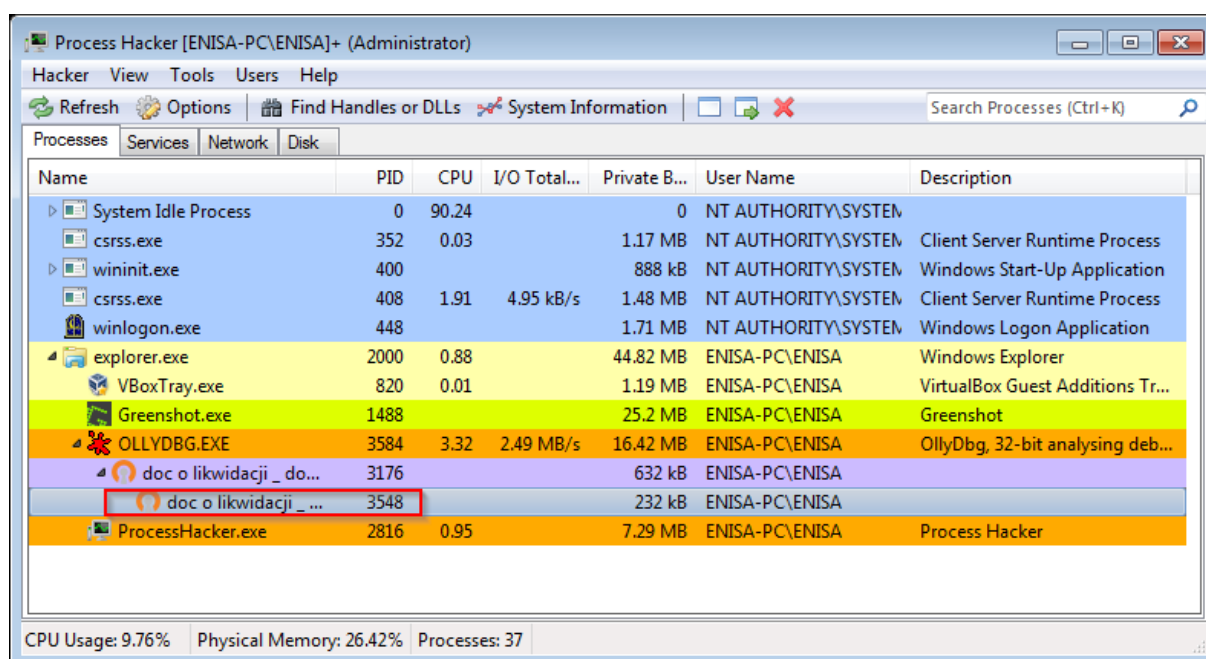
The entry point of the newly created process is stored in its EAX register¹². Its value can be read from the context structure at the address `pContext+0xB0`¹³. In this case the entry point address is `0x00401000` (remember about little-endian notation). Write down the address of the entry point, it will be needed later.

¹¹ This address might be different.

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

Resume the execution (F9) until you land at the breakpoint on *ResumeThread*.

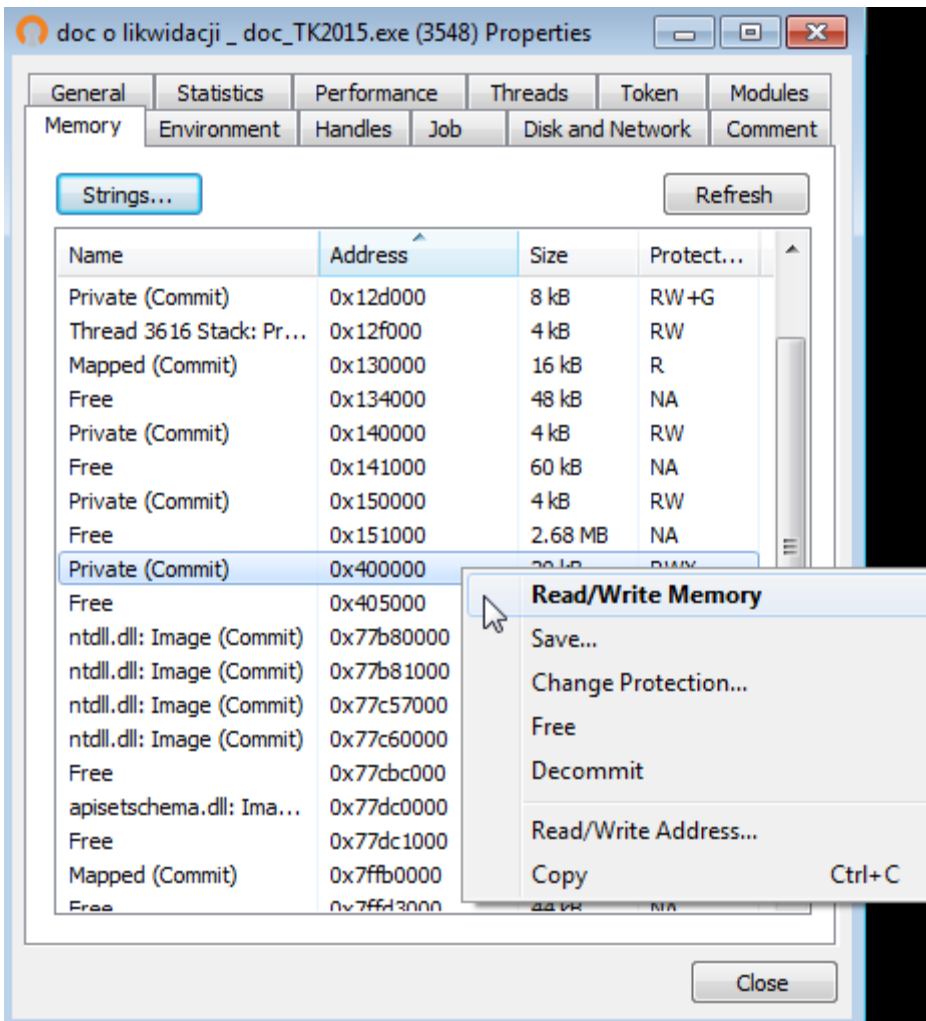
Open Process Hacker and find the suspended child process. Right-click on it and open *Properties* window.



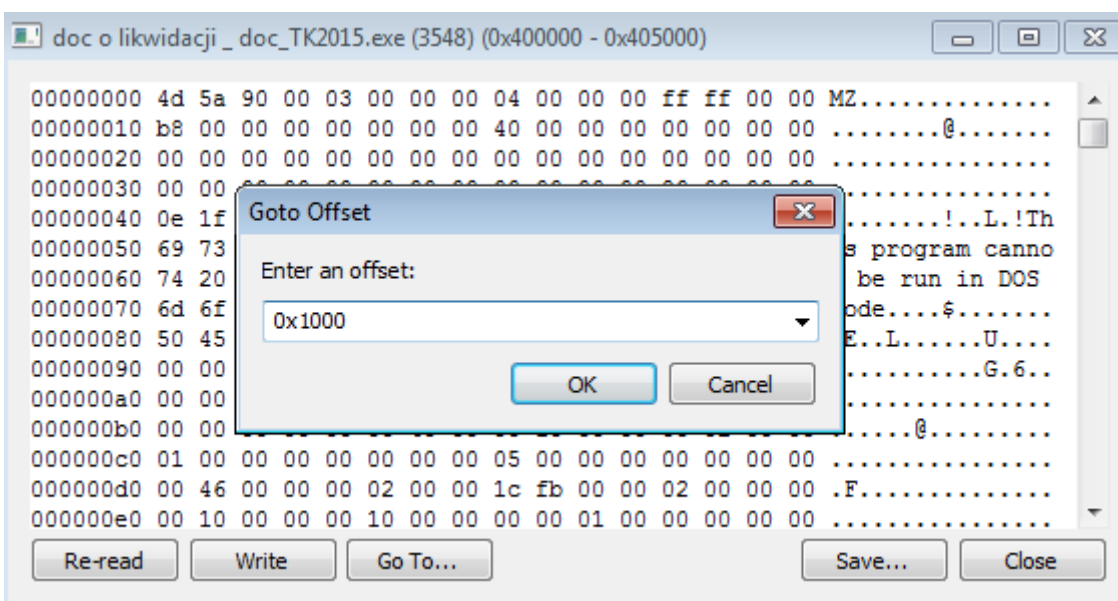
In the *Properties* window switch to *Memory* tab and find a memory block where the entry point is located (0x401000 -> memory block 0x400000). Right-click on it and choose *Read/Write Memory* option.

¹² 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.

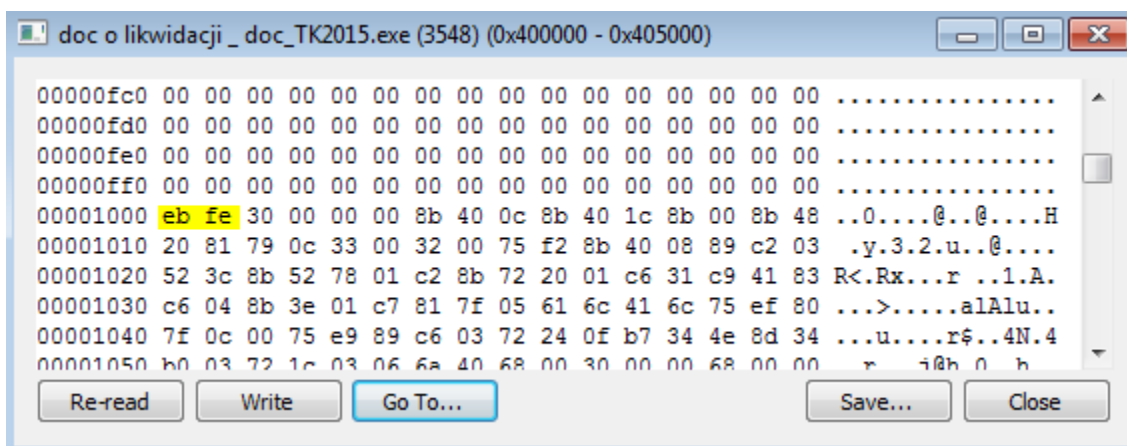
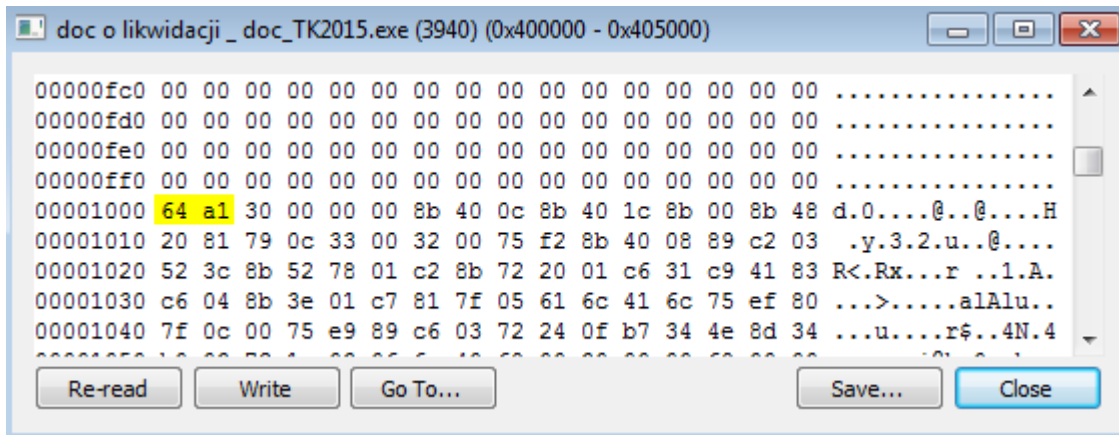
¹³struct CONTEXT http://www.nirsoft.net/kernel_struct/vista/CONTEXT.html (last accessed 11.09.2015)



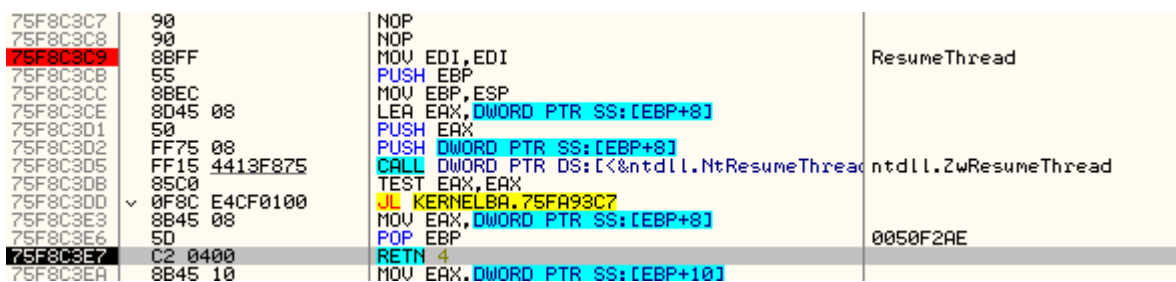
In the new window go to the entry point address at offset *0x1000* (addresses are relative to *0x400000*).



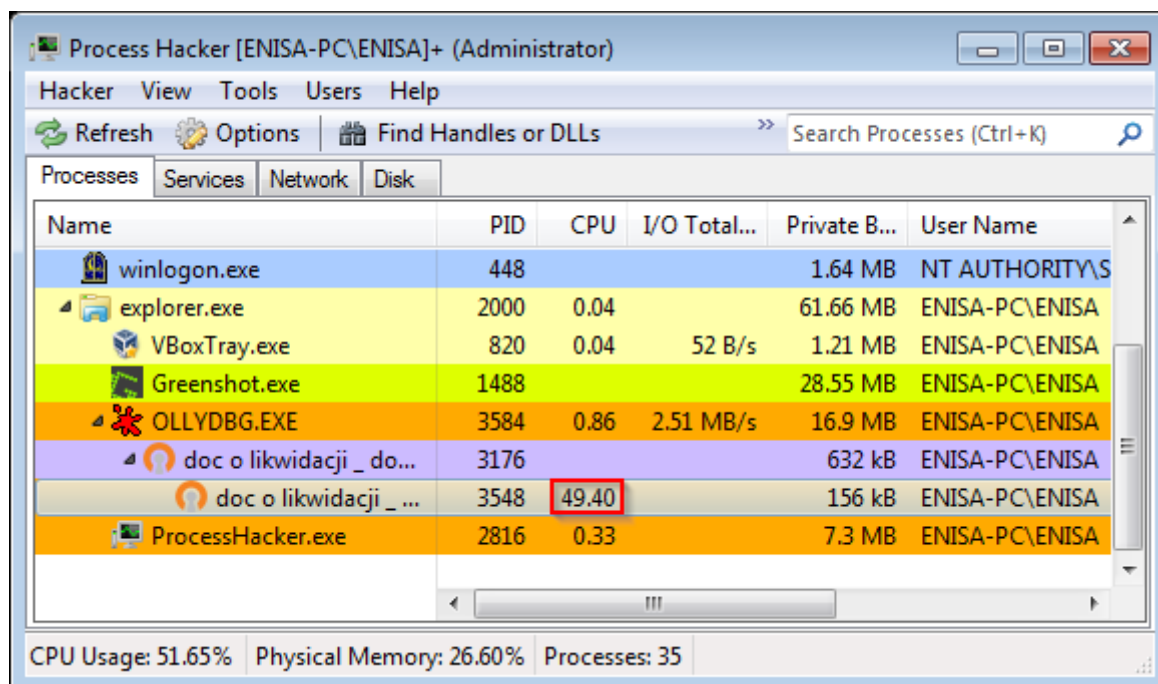
Write down the first two bytes at this offset (0x64A1) and override them with (0xEBFE). Click *Write* and close the window.



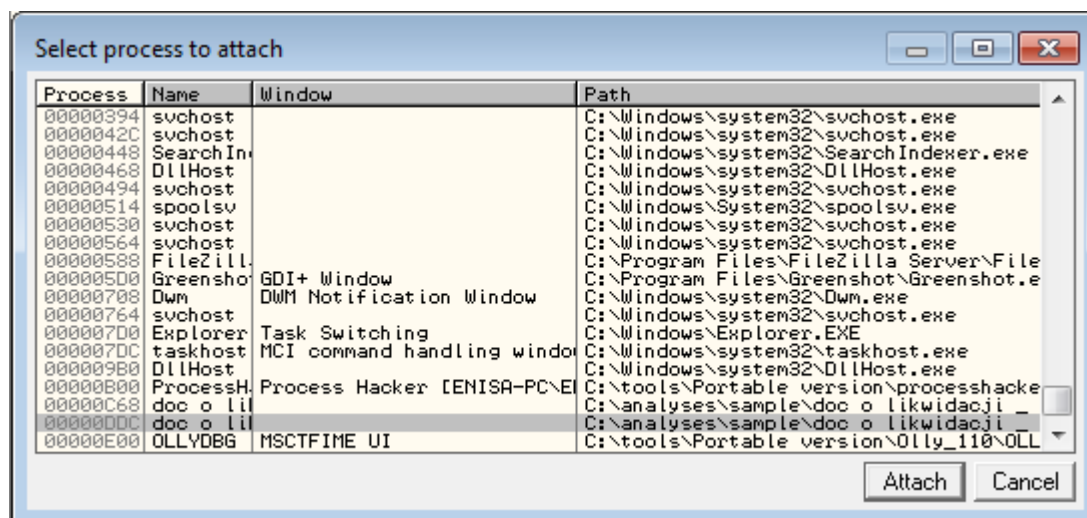
Now switch back to OllyDbg and step over (F8) till return from *ResumeThread* function.



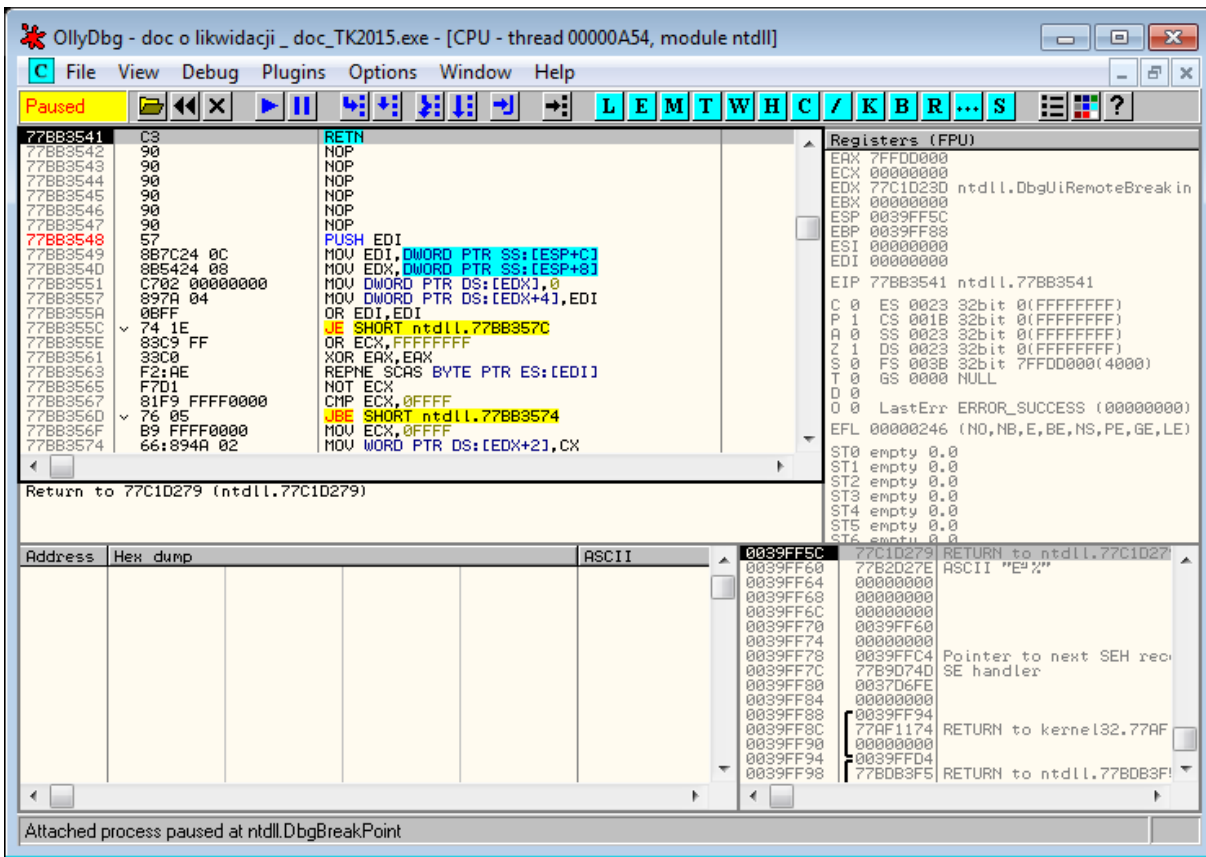
Now you can minimize OllyDbg. In the Process Hacker window you can also notice that the child process was resumed and is now using a considerable amount of CPU time. This is the result of the endless loop you created in this process. Note the process identifier (PID) of the child process (in decimal).



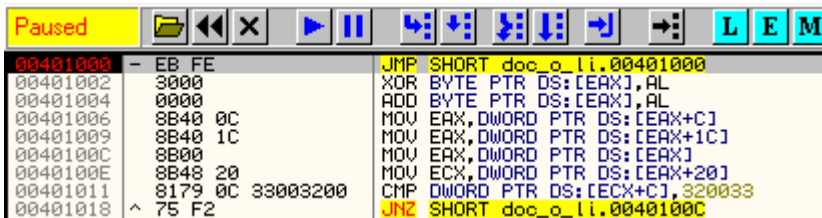
Start a new OllyDbg instance and attach it to the child process (*File->Attach*).



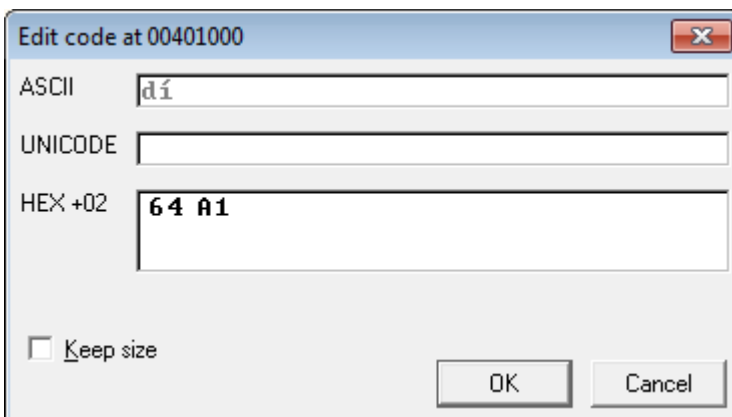
Ignore the error message (the same as previously). You should land somewhere in *ntdll*.



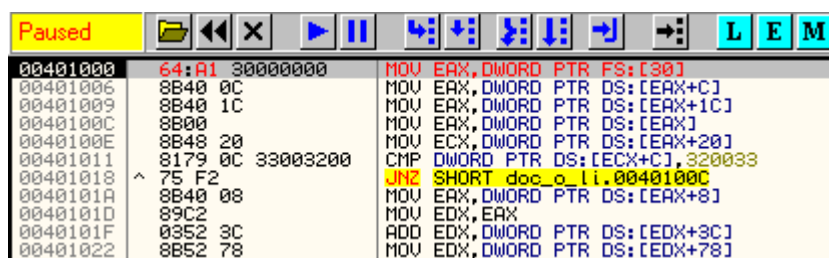
In the assembly window go to the address *0x401000* (EP). You should see the previously injected *0xEBFE* bytes. Put a breakpoint on this instruction and resume the process (F9).



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.



```

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 UNZ SHORT doc_o.ll.0040100C
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]
  
```

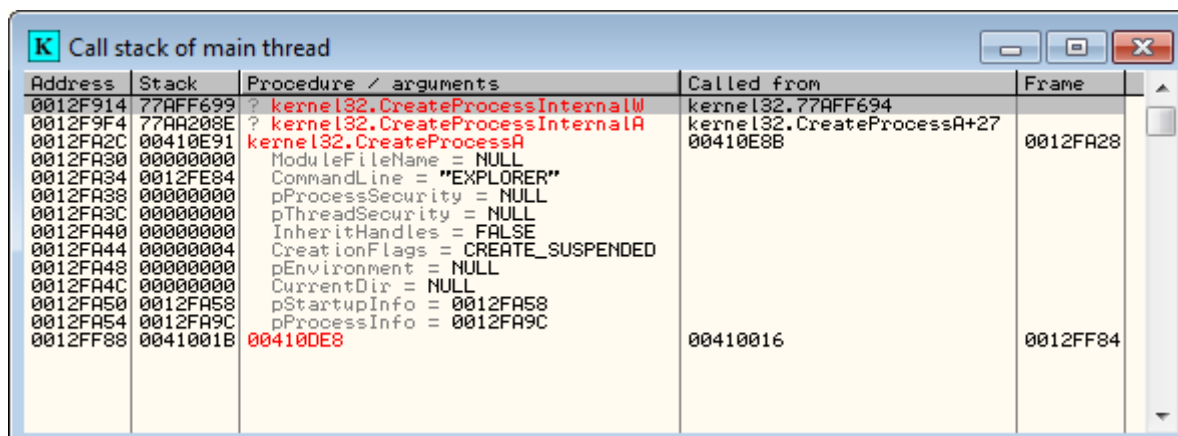
5.1.2 Second stage

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.CreateProcessInternalA	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

Continue the execution (F9) till you land at the *GetThreadContext* breakpoint.

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	...
00411036	00 00 00 40 61 70 56 69 65 77 4F 66 46 69 6C 65	...MapViewOfFile
00411046	00 57 FF 03 89 C6 E8 00 00 00 00 00 56 69 72 74 75	...VirtualAlloc
00411056	61 6C 41 6C 6C 6F 63 00 57 FF 03 97 E8 11 00 00	VirtualAllocEx
00411066	00 4F 70 65 6E 46 69 6C 65 40 61 70 70 69 6E 67	OpenFileMapping
00411076	41 00 50 FF 03 E8 00 00 00 00 5B 81 EB 80 20 40	API
00411086	00 8D 93 C3 20 40 00 8B 9B BF 20 40 00 52 6A 00	...
00411096	6A 04 FF 00 53 6A 00 6A 00 6A 04 50 FF 06 89 C6	...
004110A6	6A 40 68 00 30 00 00 53 6A 00 FF 07 89 C7 89 D9	...
004110B6	F3 A4 05 FD 0F 00 00 50 C3 6E 5F 00 00 41 43 45	...
004110C6	42 45 43 43 42 40 45 40 00 64 A1 30 00 00 8B	BECCBMEM.dio...
004110D6	40 0C 88 40 1C 88 00 8B 48 20 81 79 0C 33 00 32	...
004110E6	00 75 F2 8B 40 08 C3 E8 DD FF FF FF 89 C2 03 52	...
004110F6	3C 8B 52 78 01 C2 8B 72 20 01 C6 31 C9 41 83 C6	...
00411106	04 8B 3E 01 C7 81 7F 05 6F 63 41 64 75 EF 89 C6	...
00411116	03 72 24 0F B7 34 4E 8D 34 B0 03 72 1C 03 06 C3	...
00411126	55 89 E5 8B 7D 0C 83 C7 03 57 FF 75 08 FF 93 24	...
00411136	10 40 00 89 47 F8 31 C0 89 FF 00 00 00 F2 AE 83	...
00411146	3F FF 75 E2 C9 C2 08 00 E8 9A FF FF FF 89 83 24	...
00411156	10 40 00 E8 71 FF FF FF 50 8D 83 2C 10 40 00 87	...
00411166	04 24 50 E8 B8 FF FF FF C3 60 89 CE 51 E8 00 00	...
00411176	00 00 5D 31 C9 30 C0 89 D7 B1 2C F3 AA AC 88 C1	...
00411186	88 C5 80 E1 FE 80 E5 E7 3C F0 74 20 80 F9 F2 74	...
00411196	24 3C 66 74 29 3C 67 74 2E 80 F9 64 74 05 80 FD	...
004111A6	2E 75 2D 80 4A 01 04 88 42 04 EB D1 80 4A 01 01	...
004111B6	88 42 02 EB C8 80 4A 01 02 88 42 03 EB BF 80 4A	...
004111C6	01 02 88 42 05 FR B6 80 40 01 10 88 42 06 FR 00	...

Notice names of functions such as *MapViewOfFile* and *OpenFileMapping*.

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

Address	Hex dump	ASCII
00411026	E8 C2 00 00 00 89 C3 E8 9D 00 00 00 89 C7 E8 0E	...
00411036	00 00 00 40 61 70 56 69 65 77 4F 66 46 69 6C 65	...MapViewOfFile
00411046	00 57 FF 03 89 C6 E8 00 00 00 00 00 56 69 72 74 75	...VirtualAlloc
00411056	61 6C 41 6C 6C 6F 63 00 57 FF 03 97 E8 11 00 00	VirtualAllocEx
00411066	00 4F 70 65 6E 46 69 6C 65 40 61 70 70 69 6E 67	OpenFileMapping
00411076	41 00 50 FF 03 E8 00 00 00 00 5B 81 EB 80 20 40	API
00411086	00 8D 93 C3 20 40 00 8B 9B BF 20 40 00 52 6A 00	...
00411096	6A 04 FF 00 53 6A 00 6A 00 6A 04 50 FF 06 89 C6	...
004110A6	6A 40 68 00 30 00 00 53 6A 00 FF 07 89 C7 89 D9	...
004110B6	F3 A4 05 FD 0F 00 00 50 C3 6E 5F 00 00 41 43 45	...
004110C6	42 45 43 43 42 40 45 40 00 64 A1 30 00 00 8B	BECCBMEM.dio...
004110D6	40 0C 88 40 1C 88 00 8B 48 20 81 79 0C 33 00 32	...
004110E6	00 75 F2 8B 40 08 C3 E8 DD FF FF FF 89 C2 03 52	...
004110F6	3C 8B 52 78 01 C2 8B 72 20 01 C6 31 C9 41 83 C6	...
00411106	04 8B 3E 01 C7 81 7F 05 6F 63 41 64 75 EF 89 C6	...
00411116	03 72 24 0F B7 34 4E 8D 34 B0 03 72 1C 03 06 C3	...
00411126	55 89 E5 8B 7D 0C 83 C7 03 57 FF 75 08 FF 93 24	...
00411136	10 40 00 89 47 F8 31 C0 89 FF 00 00 00 F2 AE 83	...
00411146	3F FF 75 E2 C9 C2 08 00 E8 9A FF FF FF 89 83 24	...
00411156	10 40 00 E8 71 FF FF FF 50 8D 83 2C 10 40 00 87	...
00411166	04 24 50 E8 B8 FF FF FF C3 60 89 CE 51 E8 00 00	...
00411176	00 00 5D 31 C9 30 C0 89 D7 B1 2C F3 AA AC 88 C1	...
00411186	88 C5 80 E1 FE 80 E5 E7 3C F0 74 20 80 F9 F2 74	...
00411196	24 3C 66 74 29 3C 67 74 2E 80 F9 64 74 05 80 FD	...
004111A6	2E 75 2D 80 4A 01 04 88 42 04 EB D1 80 4A 01 01	...
004111B6	88 42 02 EB C8 80 4A 01 02 88 42 03 EB BF 80 4A	...
004111C6	01 02 88 42 05 FR B6 80 40 01 10 88 42 06 FR 00	...

Address: 00411026

ASCII: 0T

UNICODE:

HEX +00: E8 C2

Keep size

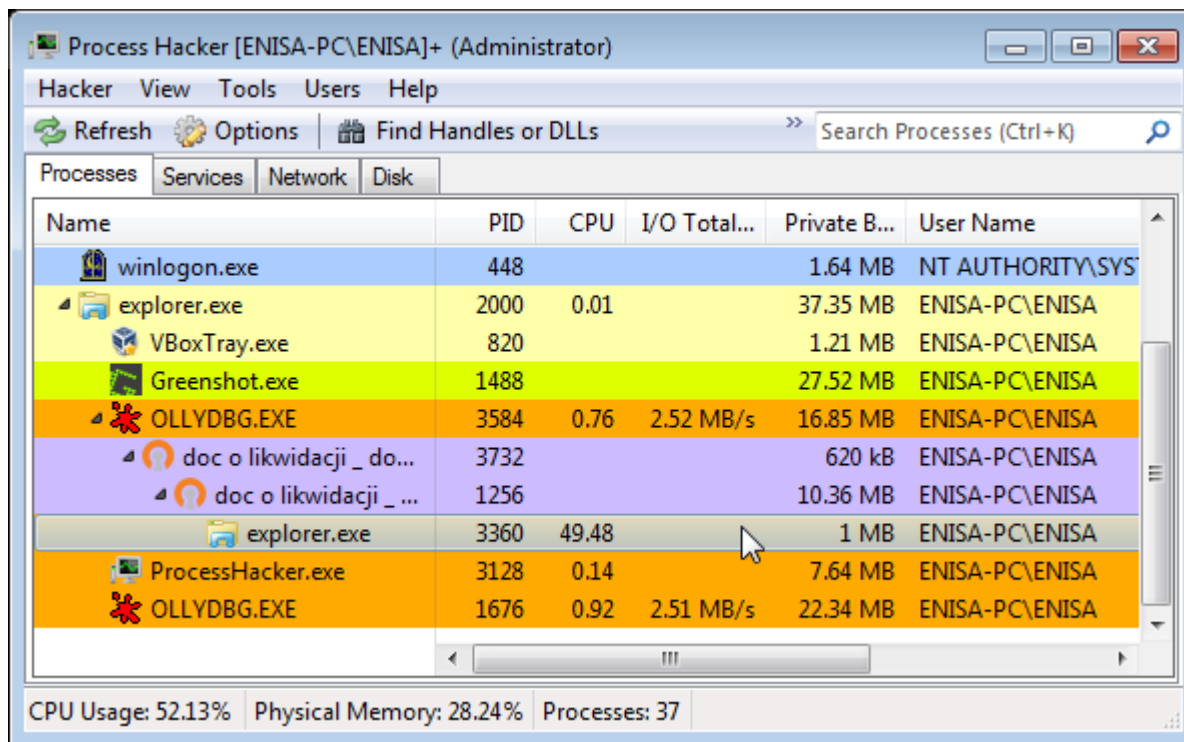
OK Cancel

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

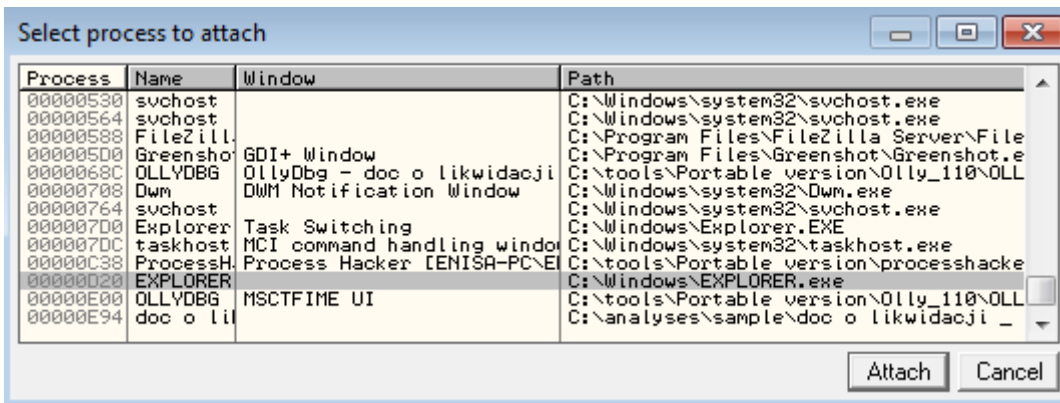
00410F3B	870424	XCHG DWORD PTR SS:[ESP],EAX	
00410F3E	FFB5 08FBFFFF	PUSH DWORD PTR SS:[EBP-428]	
00410F44	FFB5 18FBFFFF	PUSH DWORD PTR SS:[EBP-4E8]	
00410F49	FF93 75134000	CALL DWORD PTR DS:[EBX+401375]	kernel32.WriteProcessMemory
00410F50	85C0	TEST EAX,EAX	
00410F52	0F84 AA000000	JE 00411002	
00410F58	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 004138DB	
00410FB7	FFB5 1CFBFFFF	PUSH DWORD PTR SS:[EBP-4E4]	
00410FB8	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	

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

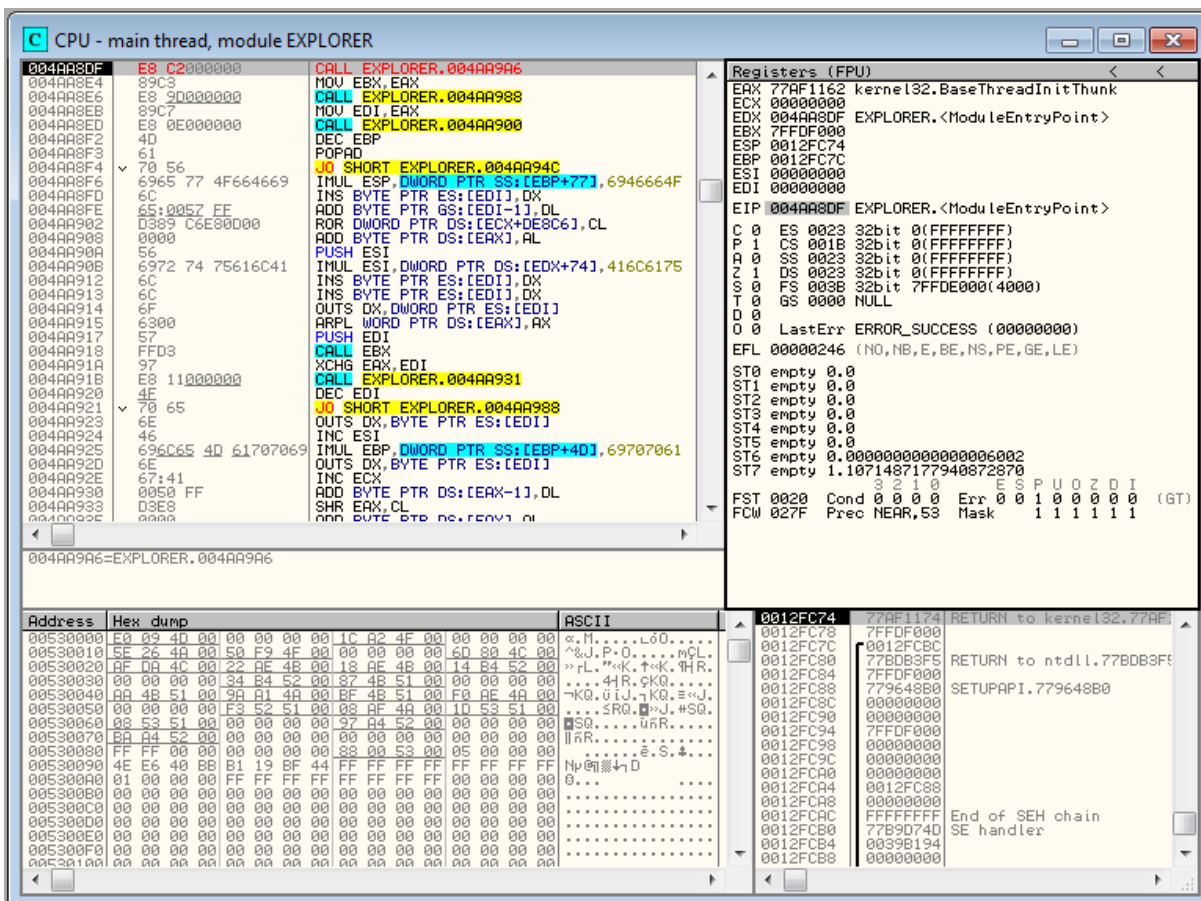
Minimize OllyDbg window and check in Process Hacker if *explorer.exe* process was resumed properly.



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.

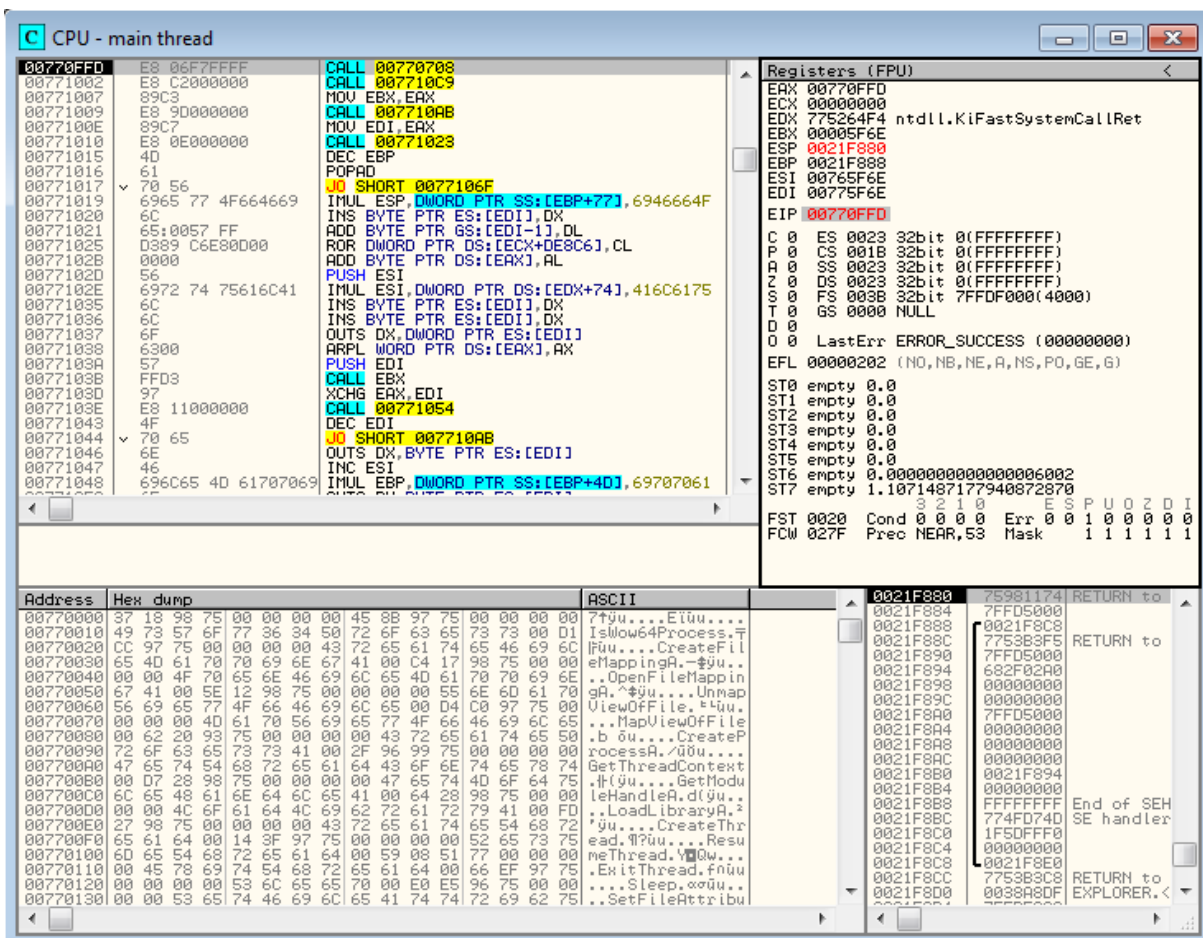


To reach the payload put a breakpoint on *OpenFileMappingA* and resume the execution (F9).

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.

004AA94F	6A 04	PUSH 4	
004AA951	FFD0	CALL EAX	; call to OpenFileMappingA
004AA953	53	PUSH EBX	
004AA954	6A 00	PUSH 0	
004AA956	6A 00	PUSH 0	
004AA958	6A 04	PUSH 4	
004AA95A	50	PUSH EAX	
004AA95B	FFD6	CALL ESI	kernel32.MapViewOfFile
004AA95D	89C6	MOV ESI,EAX	
004AA95F	6A 40	PUSH 40	
004AA961	68 00300000	PUSH 3000	
004AA966	53	PUSH EBX	
004AA967	6A 00	PUSH 0	
004AA969	FFD7	CALL EDI	kernel32.VirtualAlloc
004AA96B	89C7	MOV EDI,EAX	
004AA96D	89D9	MOV ECX,EBX	
004AA96F	F3:04	REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]	
004AA971	05 FD0F0000	ADD EAX,0FFD	
004AA976	50	PUSH EAX	
004AA977	C3	RETN	

To reach the final payload step over (F8) the return instruction (RETN).



The screenshot shows a debugger window titled 'CPU - main thread'. The main window displays assembly instructions with their addresses and hex values. The registers window on the right shows the state of various registers, with EAX at 00770FFD. The memory dump window at the bottom shows hex and ASCII data starting at address 00770000.

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

6. Introduction to scripting

6.1 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

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:[FFFFFF9EF]	
0077072D	48	DEC EAX	
0077072E	31C9	XOR ECX,ECX	
00770730	FF15 D3F9FFFF	CALL DWORD PTR DS:[FFFFFF9D3]	
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 00771795 ← 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	8D85 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	8D83 EC264000	LEA EAX,DWORD PTR DS:[EBX+4026EC]	

Take a look at the first call instruction. 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	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	

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 000 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

```

```

0021F764 007717B1 RETURN to 007717B1 from 007717B7
0021F768 00000006
0021F76C 0021F774
0021F770 0021F774
0021F774 0021F814 ABC ?eSw**
0021F778 7753EEC7 RET :dll.7753EEC7 from ntd
0021F77C 77536D0E RET :dll.77536D0E from ntd
0021F780 7753EC50 RETURN to ntdll.7753EC50 from ntd

```

```

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

```

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	!0!. =Sw#mSwPwSw
0021F784	7C 02 2F 68 B3 04 77 00 E8 56 9E 75 FF FF 00 00	!0/h 0w.\$Uku ..
0021F794	00 00 93 75 15 A9 00 00 64 F8 21 00 20 A9 38 00	..0u3r..d0!. 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 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

```

Take a look at the memory dump.

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	!0/h 0w.\$Uku ..
0021F794	00 00 93 75 15 A9 00 00 64 F8 21 00 20 A9 38 00	..0u3r..d0!. r8.

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

```

var base
var labels

; checking memory base of Tinba payload
gmemei 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

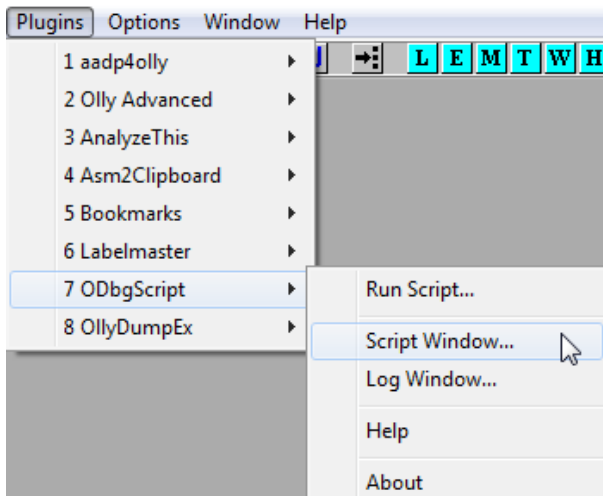
; printing result
eval "{data_addr} ({len} bytes) -> {string}"
log $RESULT,""

add base,7
jmp search_loop

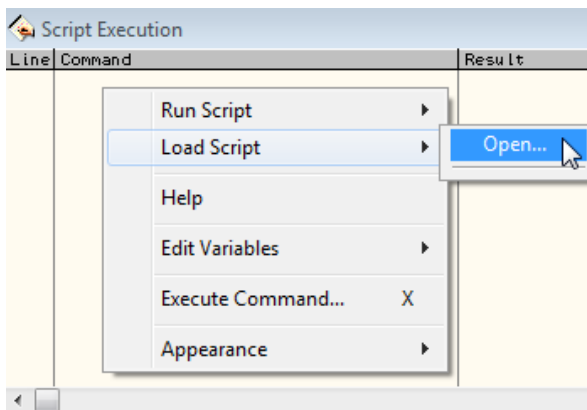
end_loop:
log "-----"
free labels
pause
```

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).

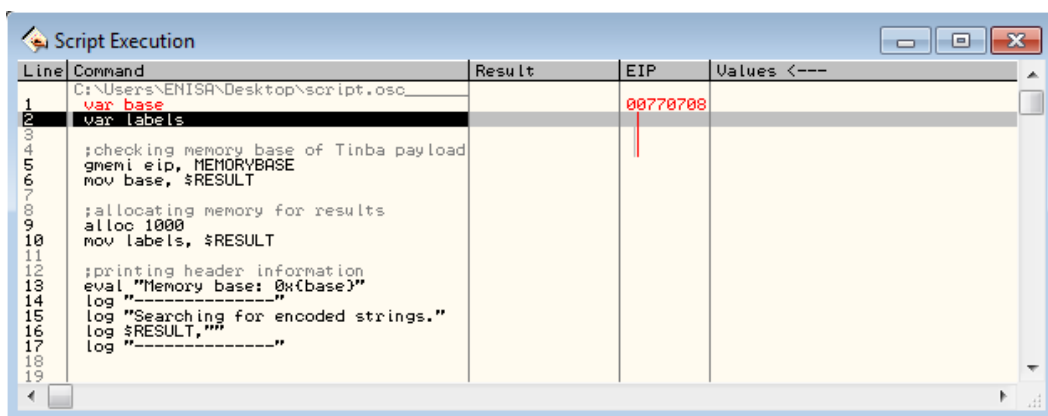
Then open ODbgScript *Script Window* and *Log Window*.



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.

```

Script Log Window
-----
Address  Message
-----
770708  Searching for encoded strings.
770708  Memory base: 0x770000
-----
770708  7707F4 (9 bytes) -> ADVAPI32
770708  77083A (9 bytes) -> \\bin.exe
770708  7709B8 (2E bytes) -> Software\Microsoft\Windows\CurrentVersion\Run
770708  770DFA (6 bytes) -> DOUBLE
770708  770E58 (9 bytes) -> EXPLORER
770708  7717B1 (6 bytes) -> NTDLL
770708  772778 (10 bytes) -> %LOCALAPPDATA%\
770708  7727A6 (1F bytes) -> Packages\windows_ie_ac_001\AC\
770708  77280E (8 bytes) -> %APPDATA%\
770708  77285F (10 bytes) -> %LOCALAPPDATA%\
770708  7728AC (5 bytes) -> Low\
770708  772A7E (9 bytes) -> \\log.dat
770708  772BDA (7 bytes) -> WS2_32
770708  772C0F (8 bytes) -> CRYPT32
770708  772C45 (9 bytes) -> ADVAPI32
770708  772C8B (9 bytes) -> \ntf.dat

```

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.

```

770708  774620 (8 bytes) -> set_url
770708  774746 (8 bytes) -> set_url
770708  7747A8 (C bytes) -> \ndata_before
770708  7747F4 (9 bytes) -> \ndata_end
770708  774842 (C bytes) -> \ndata_inject
770708  774894 (9 bytes) -> \ndata_end
770708  7748E9 (8 bytes) -> \ndata_after
770708  774934 (9 bytes) -> \ndata_end
770708  7751EE (0 bytes) -> Frame-Options

```

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.



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



Catalogue Number (IF APPLICABLE)



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

ISBN: xxxxxxxx
DOI: xx.xxxx/xxxxx
REMOVE If not applicable

