Advanced artefact analysis
Advanced dynamic analysis

HANDBOOK, DOCUMENT FOR TEACHERS

OCTOBER 2015
About ENISA

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

Authors
This document was created by Yonas Leguesse, Christos Sidiropoulos, Kaarel Jõgi and Lauri Palkmets in consultation with ComCERT¹ (Poland), S-CURE² (The Netherlands) and DFN-CERT Services (Germany).

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

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

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

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

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

Copyright Notice
© European Union Agency for Network and Information Security (ENISA), 2015
Reproduction is authorised provided the source is acknowledged.

¹ Dawid Osojca, Paweł Weżgowiec and Tomasz Chlebowski
² Don Stikvoort and Michael Potter
# Table of Contents

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

The aim of this training is to present methods and techniques of dynamic artefact analysis with the use of OllyDbg\(^3\) debugger package.

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.

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.

The training is performed using the Microsoft Windows operating system.

## Targeted Audience

CSIRT staff involved with the technical analysis of incidents, especially those dealing with the sample examination and malware analysis. Prior knowledge of assembly language and operating systems internals is highly recommended.

## Total Duration

8-10 hours

## Frequency

Once for each team member.

---

1. Training introduction

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

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

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

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

---

4 OllyDbg [http://www.ollydbg.de/] (last accessed 11.09.2015)
5 Dyre: Emerging threat on financial fraud landscape
2. Introduction to OllyDbg

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

You will learn:

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

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

2.1 OllyDbg interface

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

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

---

6OllyDbg http://www.ollydbg.de/ (last accessed 11.09.2015)
8Building artefact handling and analysis environment https://www.enisa.europa.eu/activities/cert/training/training-resources/technical-operational#building (last accessed 11.09.2015)
Now you should see the OllyDbg interface.

There are two ways to start a debugging process. Firstly, you can attach to the already running process. To do this, choose File->Attach and then choose the process of your interest. After attaching to the running process, 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. This is the window you will use most often during an analysis. It consists of five separate subpanels: disassembly view, registers, information panel, memory dump and stack panel.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Disassembly</th>
<th>Registers</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV EAX, DWORD PTR DS:[ESI+8]</td>
<td>0856 88</td>
<td>MOV EAX, DWORD PTR DS:[47E:4E]</td>
<td>EIP</td>
<td>putty.&lt;ModuleEntryPoint&gt;</td>
</tr>
<tr>
<td>MOV EAX, DWORD PTR DS:[ESI+16]</td>
<td>0864 90</td>
<td>MOV EAX, DWORD PTR DS:[47E:4E]</td>
<td>EIP</td>
<td>putty.&lt;ModuleEntryPoint&gt;</td>
</tr>
<tr>
<td>MOV EAX, DWORD PTR DS:[ESI+24]</td>
<td>0872 9C</td>
<td>MOV EAX, DWORD PTR DS:[47E:4E]</td>
<td>EIP</td>
<td>putty.&lt;ModuleEntryPoint&gt;</td>
</tr>
</tbody>
</table>

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

```
Registers (CPU) -
EBX: 77381162 kernel32.BaseThreadInitThunk
EDX: 00000000
EDX: 09454000
ESP: 001F2F39
EIP: 09454000
C 0 ES 0023 32bi: 01FFFFFFF
D 0 DS 0023 32bi: 01FFFFFFF
A 0 SS 0023 32bi: 01FFFFFFF
S 0 FS 0038 32bi: 7FDD0000
T 0 DS 0000 NULL
O 0 O LastErr ERROR_SUCCESS 00000000
EFL: 00000245 (NO,HB,E,EB,SH,PE,OE,LE)
```

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

```
RI: 00000000
EBX: 77381162 (kernel32.BaseThreadInitThunk)
```

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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0047AB000</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
<tr>
<td>0047B181</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
<tr>
<td>0047B250</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
<tr>
<td>0047B250</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
<tr>
<td>0047B250</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
<tr>
<td>0047B250</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
<tr>
<td>0047B250</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
<tr>
<td>0047B250</td>
<td>00 00 00 00</td>
<td>...</td>
</tr>
</tbody>
</table>

Besides the hexadecimal, you can choose other data representation formats by right-clicking on the memory dump panel and choosing required data representation from the context menu.
Take some time to check other data representations. At the end, restore the default format: Hex->Hex/ASCII (16 bytes).

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

Besides the CPU window, OllyDbg offers few other windows used for different purposes. All windows can be accessed with windows buttons on the toolbar or View menu.
The more frequently used windows are: **Executable modules, Memory map, Threads, Handles, Call stack, Breakpoints.**

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

For example, right-click on **msvcrt** and choose **View names** to be presented with a list of all names defined in the **msvcrt** library (imports and exports).
Memory map window presents the memory structure with all allocated memory regions in the address space of the debugged process. It is useful to track memory allocation operations done by the malicious code. Similarly as in the previous example you can right-click on any memory region to access the context menu with additional operations (dumping memory, searching memory, changing access rights, freeing memory, etc.).

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.

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

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

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

Threads window shows all threads of the current process. If the process has more than one thread, double-clicking on the thread would switch the context to this thread.
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 if 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. This is useful for checking to which function the current instruction belongs to and from where this function was called. Note that the screenshot below presents the Call stack window during process execution, after two calls were made. If you open the Call stack window while at the entry point, it will be empty.

One of the useful OllyDbg features is highlighting elements that have changed. A good example of this is highlighting newly allocated memory blocks or newly loaded modules.

To present this, 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`. This way OllyDbg will load an additional module in the address space of the currently debugged process. Loading extra modules is sometimes useful in more advanced debugging when you want to load the DLLs with your custom code.

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

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

### 2.2 Basic debugging and code navigation

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

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

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

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

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

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

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

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

Other useful debug operations are:

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

Debugging actions can be also accessed through the toolbar at the top of OllyDbg.
If you want to quickly pre-view the execution flow of a program (find loops, check which jumps are taken, etc.) you might decide to use the instruction trace or instruction animation functions. Both functions come in two forms: Trace into/Trace over and Animate into/Animate over.

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

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

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

Animate into function usually takes some time until the program finishes execution. To stop it, use Pause (F12) function.
Next restart the sample again and choose **Debug->Set condition**.

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

For example set the following two conditions.

- **EIP is outside the range**
  - 00400000
  - 00500000

- **Command is one of**
  - `call eax`
  - `call edi`

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

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

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

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

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

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

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

Whenever you see some call or jump instruction you can follow it (without executing) by clicking on this instruction and pressing <Enter>. In this example follow a call to putty.004545A0. You should land at the function body.

You can do the same with jump instructions.

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

---

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

Click on disassembly view and press Ctrl+G.

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

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

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

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

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

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

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

For example, start clicking on registers values. If a register points to the existing address in the address space of the current program, there should be the following options: Follow in Disassembler and Follow in Dump.
If the register does not contain a valid address, these options won’t be available. Additionally if a register points to the location on the stack (like in case of ESP register) there will be an option *Follow in Stack*.

You can do the same with values stored on stack.
2.3 Breakpoints

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

There are four types of breakpoints in OllyDbg:\(^{10}\):

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

Software breakpoints work by inserting an INT 3\(^{11}\) instruction in the place of the Instruction on which the breakpoint is set. When the instruction is about to be executed, the interrupt is raised and the debugger steps in. The entire process is transparent to the user.

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

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

```plaintext
004542CF L C3 RETN
004542D0 6A 60 PUSH 60
004542D2 68 07041700 PUSH 0x07041700
004542D4 6A 44468000 PUSH 0x44468000
004542D6 80C4 MOV EAX,EBX
004542D8 E8 FFFFFFFF CALL 0x7FFFDFFFF
```

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

```plaintext
004542CF L C3 RETN
004542D0 6A 60 PUSH 60
004542D2 68 07041700 PUSH 0x07041700
004542D4 6A 44468000 PUSH 0x44468000
004542D6 80C4 MOV EAX,EBX
004542D8 E8 FFFFFFFF CALL 0x7FFFDFFFF
```

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

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

\(^{10}\) [http://www.ollydbg.de/Help/i_Breakpoints.htm](http://www.ollydbg.de/Help/i_Breakpoints.htm) (last accessed 11.09.2015)

\(^{11}\) The INT 3 instruction is defined for use by debuggers to temporarily replace an instruction in a running program in order to set a breakpoint. [https://en.wikipedia.org/wiki/int_(x86_instruction)](https://en.wikipedia.org/wiki/int_(x86_instruction)) (last accessed 11.09.2015)
You can also use this window to remove or temporarily disable chosen breakpoints.

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

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

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

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

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

Next in the PuTTY window, click the About button and then the Visit Web Site button.
Now go back to OllyDbg. OllyDbg should break on a call to\texttt{ShellExecuteA} (on the previously set breakpoint).

Take a look at the stack view to see arguments passed to \texttt{ShellExecuteA}.

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

You can also open the call stack window (\textit{View}→\textit{Call stack}, Clt+K) to check from where \texttt{ShellExecuteA} function was called.
The second type of breakpoints are hardware breakpoints. In general, instead of changing program instructions in the memory as software breakpoints do, they use special processor registers (debug registers). On the x86 architecture there are four debug registers (DR0-DR3) used to store the linear address of breakpoints. Thus it is possible to set four hardware breakpoints at a time. Additionally, in contrast to software breakpoints, hardware breakpoints can be also used to break on memory read or write operations.

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

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

Now let’s see how to set up hardware breakpoints: 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.

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

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

---


2 Debugger flow control: More on breakpoints (part 2) [http://www.nynaeve.net/?p=81](http://www.nynaeve.net/?p=81) (last accessed 11.09.2015)
Now **Memory Dump** view should be centred on the **0x47E140** address. Select the first 4 bytes (dword) and right-click on them. From the context menu choose **Breakpoint->Hardware, on access->Dword**.

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

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

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

After setting up a hardware breakpoint on **0x47E140**, resume the program execution (F9).
Almost immediately the program should break. As the message in the status bar shows, hardware breakpoint 1 was hit and EIP points to one instruction after 0x47E140 address was accessed.

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).

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

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

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

You can set memory breakpoints in a similar manner as hardware breakpoints by selecting some data in Memory Dump view and then choosing Breakpoint->Memory.
The second way of creating a memory breakpoint is using *Memory map* window.

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

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

If you check *Call stack* window you will see that the breakpoint was hit after a call to *CreateDialogParamA* from which *FindResourceExA* was called.
To remove a memory breakpoint, go to the Memory map window, right-click on the memory region on which the memory breakpoint was set and select Remove memory breakpoint.

### 2.4 Execution flow manipulation

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

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

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

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

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

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

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

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.

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

This time however there are two options: **Modify** and **Edit**. The difference between them is that **Modify** treats stack values as numbers while **Edit** treats stack values as group of bytes.
Besides modifying registers and data in the memory, it is also possible to change instructions that are executed. To achieve this just select the instruction you want to modify and press <space>.

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

2.5 Plugins

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

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

- Collaborative RCE Tool Library\(^{14}\)
- Tuts 4 You\(^{15}\)
- OpenRCE.org\(^{16}\)

\(^{15}\) OllyDbg 1.xx Plugins https://tuts4you.com/download.php?list.9 (last accessed 11.09.2015)
After downloading a plugin, unpack it and copy the plugin’s .dll library to the OllyDbg’s plugin directory (e.g. `c:\tools\Portable version\Olly_110\Plugins`). The exact location of the plugins directory can be checked in the Options->Appearance->Directories menu.

After plugin installation, restart OllyDbg. If the plugin is working, it should be available through the plugins menu.
Note that plugins created for OllyDbg v1.10 are not compatible with OllyDbg 2.xx and vice versa.

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

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

### 2.6 Shortcuts

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

**Debugging:**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>SHORTCUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>F9</td>
</tr>
<tr>
<td>Pause</td>
<td>F12</td>
</tr>
<tr>
<td>Restart debugged app</td>
<td>Ctrl+F2</td>
</tr>
<tr>
<td>Close debugged app</td>
<td>Alt+F2</td>
</tr>
<tr>
<td>Step into</td>
<td>F7</td>
</tr>
<tr>
<td>Step over</td>
<td>F8</td>
</tr>
<tr>
<td>Execute till return</td>
<td>Ctrl+F9</td>
</tr>
<tr>
<td>Execute till user code</td>
<td>Alt+F9</td>
</tr>
<tr>
<td>Pass exception to the program</td>
<td>Shift+F7/F8/F9</td>
</tr>
</tbody>
</table>
### Advanced artefact analysis

**October 2015**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animate into</td>
<td>Ctrl+F7</td>
</tr>
<tr>
<td>Animate over</td>
<td>Ctrl+F8</td>
</tr>
<tr>
<td>Trace into</td>
<td>Ctrl+F11</td>
</tr>
<tr>
<td>Trace over</td>
<td>Ctrl+F12</td>
</tr>
</tbody>
</table>

**Windows and views:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU window</td>
<td>Alt+C</td>
</tr>
<tr>
<td>Memory map</td>
<td>Alt+M</td>
</tr>
<tr>
<td>Executable modules</td>
<td>Alt+E</td>
</tr>
<tr>
<td>Call stack</td>
<td>Alt+K</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>Alt+B</td>
</tr>
</tbody>
</table>

**Other operations:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Follow jump/call</td>
<td>Enter</td>
</tr>
<tr>
<td>Assembly instruction</td>
<td>Space</td>
</tr>
<tr>
<td>Edit memory</td>
<td>Ctrl+E</td>
</tr>
<tr>
<td>Add comment</td>
<td>; (semicolon)</td>
</tr>
<tr>
<td>Add label</td>
<td>: (colon)</td>
</tr>
<tr>
<td>Insert bookmark X</td>
<td>Alt+Shift+0..9</td>
</tr>
<tr>
<td>Go to bookmark X</td>
<td>Alt+0..9</td>
</tr>
</tbody>
</table>
3. Unpacking artefacts

3.1 Packers and protectors

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

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

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

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

![Diagram showing unpacking process](image)

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

When doing a malware analysis other than a behavioural or automatic analysis, it will be necessary to first unpack the malware sample. Otherwise you won’t be able to analyse the original code. To detect if the sample was packed and what packer was used, you might use tools such as PEiD\(^{17}\) or ExeInfo PE\(^{18}\). If the packer used to pack the sample is well known, these tools should return its name. You can then search for an automatic unpacking tool for this particular packer\(^ {19}\). Remember to always use unpacking tools in isolated environments.

### 3.1.2 Unpacking steps

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

There are three stages of unpacking a binary file:

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

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

After the OEP has been found, you need to dump the memory of the unpacked image of the original executable. Sometimes, the packer might utilize anti-dumping techniques\(^{20}\): these are however not part of this training.

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

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

### 3.1.3 Finding the OEP

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

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

---


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

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

### 3.2 Unpacking UPX packed sample

UPX\(^{21}\) is a fairly simple and commonly used packer. Samples that have been packed with UPX can be easily unpacked using publicly available tools, but during this exercise it will be shown how to manually unpack a UPX packed sample (putty.exe), to show the general concept of unpacking artefacts manually.

As an initial step, the packed code needs to be detected, similarly to the steps in the ENISA training material “Artefact analysis fundamentals”\(^{22}\). PEiD\(^{23}\) is used to identify if the sample was packed and what packer was used. In some cases PEiD won’t properly identify if the sample was packed. In such situations, other checks and some manual assessment may be needed (for example checking embedded strings, inspecting IAT table or inspecting list of sections) – as described in “Artefact analysis fundamentals”.

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

---


To confirm the output of PEiD and to identify packing more specifically, CFF Explorer$^{24}$ is used. CFF Explorer is designed to make the PE editing as easy as possible. Beside PE headers viewing and editing CFF Explorer contains an integrated hex editor, simple disassembler and many other useful features. One distinct feature of UPX packed samples are two sections UPX0 and UPX1 within PE file, as seen on the screenshot below under the section headers part. Please note the Virtual Addresses of both sections$^{25}$, they will be referred later in this exercise.

![CFF Explorer screenshot](image)

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”.

![OllyDbg screenshot](image)

---


$^{25}$ Actually CFF Explorer presents the Relative Virtual Addresses (RVA) in the Virtual Address column in the sections list. To get actual Virtual Addresses of the sections, add to them the module base address (here 0x4000000) or use the ‘Address Converter’ feature from the CFF Explorer.
Execution of the executable should be paused at the entry point of putty_upx.exe (0x48CD50) which is located in the UPX1 section.

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

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>ESI1</th>
</tr>
</thead>
<tbody>
<tr>
<td>044D0000</td>
<td>37 FF FF PF FF 50 0C 66 01 0D 34 08 08 08 FF 70 06 06 4C 2C 72</td>
<td></td>
</tr>
</tbody>
</table>
Then follow in the hex dump EDI register to see a clean memory where unpacked code will be stored.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00431200</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00431220</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00431240</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00431260</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00431280</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>004312A0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>004312C0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>004312E0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00431300</td>
<td>36 6A 69 64 61 73 34 34 0B 6A 6E 73 39 34 24 19</td>
<td>U</td>
</tr>
<tr>
<td>00431320</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00431340</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00431360</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00431380</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>004313A0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>004313C0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>004313E0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

Now scroll down over numerous jump instructions until you see three CALL instructions (at the addresses 0x48CE8A, 0x48CE8B, 0x48CEBA). Set breakpoints at those instructions to inspect what functions are called there. Resume execution (F9).

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

Put breakpoint at 0x48CEBF (outside IAT reconstruction loop) and press F9 a few times (5-10). In the stack window you can observe what functions are being loaded.
Take a look where resolved functions addresses are being stored. After a call to GetProcAddress the result is checked if it is non-zero (OR EAX,EAX) and is written to the address pointed by EBX register (MOV DWORD PTR DS:[EBX],EAX).

Follow EBX register in hex dump.

Now remove all the breakpoints except the breakpoint outside the IAT reconstruction loop (0x48CEBF) and resume execution (F9) to let the IAT reconstruction finish. A memory region pointed by EBX should be filled with addresses of resolved functions. It is characteristic for the IAT table that many addresses start with 0x75, 0x76, 0x77. This is because system libraries are usually loaded at such address ranges.

At this point it might be handy to change the hex dump view type to the address view to list all functions names (as hinted by OllyDbg). To do this right-click the hex dump and from the context menu choose Long Address.
Next scroll down the assembly code until the characteristical JMP instruction at 0x48CEFC. Such unconditional jump instructions at the end of the unpacking routine often leads to the OEP thus it is always worth inspecting them (but keep in mind this is not the only way of jumping to the OEP). Put a breakpoint at this instruction and resume the execution (F9).
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.

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

When dumping unpacked putty.exe code use the default OllyDumpEx settings. Save dumped process as dump.exe. Don’t close OllyDbg yet.
Now if you would try to execute `dump.exe` you will see an Application Error. That’s because `dump.exe` still doesn’t have the IAT reconstructed.

To reconstruct IAT you will use the ImpREC\textsuperscript{26} tool. Run ImpREC (as Administrator) and from the scroll down menu at the top of the window choose the `putty_upx.exe` process.

\textsuperscript{26}ImpREC \url{http://www.woodmann.com/collaborative/tools/index.php/ImpREC} (last accessed 10.10.2015)
Next in the “IAT Infos needed” panel enter the RVA address of the OEP (OEP address minus Image Base address, in this case 0x459FE0-0x400000=0x59FE0) and click “IAT AutoSearch”. If the IAT is found you should see the appropriate message box. Otherwise you might need to try and manually enter RVA and Size of the IAT.
Next click “Get Imports”.
Click “Show Invalid” to see if there are any invalid functions. In this case there shouldn’t be any. Click “Fix Dump” and select dump.exe file. If everything goes right you should see a message that dump_.exe was saved successfully (please note underscore in the name of the file name, the originale file wasn’t overwritten).

You can try to run dump_.exe to check if it runs.

### 3.3 Unpacking UPX with the ESP trick

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

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

First, the same as in the previous exercise, open putty_upx.exe in OllyDbg.
Step over `PUSHAD` instruction (Shortcut key F8). Notice how the stack view and ESP register changes, as seen on the following screenshot.

Follow the ESP register in the hex dump and put a hardware breakpoint (on access) on the memory region pointed to by this register.
Next resume the execution (F9) and you should immediately land just before the jump to the OEP.
Remove the hardware breakpoint (Debug -> Hardware breakpoints, Delete).
Now put a breakpoint on the JMP instruction (0x48CEFC), and resume the execution (F9) until you reach it. Step over the JMP instruction (F8) and you should land at the OEP.

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

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

### 3.4 Unpacking a Dyre sample

In this exercise the unpacking of the Dyre\(^\text{27}\) 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.

```
00401920 . C3  RETN
00401920 . EG EC1E0000 DFL voiyhabs.0040312F
00401940 . EX 89FEFFFF JMP voiyhabs.00401901
00401940 . SBFF
00401940 . EE 5E PUSH EBP
00401940 . DD5E SUB ESP, 8EB
00401940 . 8EB 28020000
```

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

\(^{27}\)Dyre: Emerging threat on financial fraud landscape

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

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

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).

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).

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.

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

Address of the WndClassEx structure is put as the first argument onto the stack. Follow it in the dump.
Now read the address of the Window procedure from the hex dump remembering that addresses are written in the little-endian notation. In this case the Window procedure address is 0x02100210.

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

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

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

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

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

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.

Step over (F8) two times.

Now scroll down to a group of three calls just before the function return. At this point you already know that after the return (to the previous routine) there isn’t much going on in the code. This means that the jump to OEP is likely taking place in one of this calls. Put a breakpoint on the first call and resume the execution (F9).
After reaching the breakpoint step into the first function call (F7). You will see several PUSH instructions followed by a call instruction.

<table>
<thead>
<tr>
<th>55</th>
<th>PUSH EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>88EC</td>
<td>MOV EBP, ESP</td>
</tr>
<tr>
<td>08EC</td>
<td>MOV ESP, 0C</td>
</tr>
<tr>
<td>E3</td>
<td>CALL 01571524</td>
</tr>
<tr>
<td>0945 F8</td>
<td>MOV DWORD PTR SS:[EBP-8], 0x38</td>
</tr>
<tr>
<td>0945 F0</td>
<td>MOV EAX, DWORD PTR DS:[EAX+40]</td>
</tr>
<tr>
<td>0945 F0</td>
<td>MOV ECX, DWORD PTR DS:[EAX+48]</td>
</tr>
<tr>
<td>0945 F4</td>
<td>MOV DWORD PTR SS:[EBP-8], ECX</td>
</tr>
<tr>
<td>F75 1C</td>
<td>PUSH DWORD PTR SS:[EBP+1C]</td>
</tr>
<tr>
<td>F75 18</td>
<td>PUSH DWORD PTR SS:[EBP+18]</td>
</tr>
<tr>
<td>F75 14</td>
<td>PUSH DWORD PTR SS:[EBP+14]</td>
</tr>
<tr>
<td>F75 10</td>
<td>PUSH DWORD PTR SS:[EBP+10]</td>
</tr>
<tr>
<td>F75 0C</td>
<td>PUSH DWORD PTR SS:[EBP+0C]</td>
</tr>
<tr>
<td>F75 08</td>
<td>PUSH DWORD PTR SS:[EBP+08]</td>
</tr>
<tr>
<td>0945 FC</td>
<td>MOV DWORD PTR SS:[EBP-4], 0x04000000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>55</th>
<th>PUSH EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>88EC</td>
<td>MOV EBP, ESP</td>
</tr>
<tr>
<td>08EC</td>
<td>MOV ESP, 0C</td>
</tr>
<tr>
<td>E3</td>
<td>CALL 01571524</td>
</tr>
<tr>
<td>0945 F8</td>
<td>MOV DWORD PTR SS:[EBP-8], 0x38</td>
</tr>
<tr>
<td>0945 F0</td>
<td>MOV EAX, DWORD PTR DS:[EAX+40]</td>
</tr>
<tr>
<td>0945 F0</td>
<td>MOV ECX, DWORD PTR DS:[EAX+48]</td>
</tr>
<tr>
<td>0945 F4</td>
<td>MOV DWORD PTR SS:[EBP-8], ECX</td>
</tr>
<tr>
<td>F75 1C</td>
<td>PUSH DWORD PTR SS:[EBP+1C]</td>
</tr>
<tr>
<td>F75 18</td>
<td>PUSH DWORD PTR SS:[EBP+18]</td>
</tr>
<tr>
<td>F75 14</td>
<td>PUSH DWORD PTR SS:[EBP+14]</td>
</tr>
<tr>
<td>F75 10</td>
<td>PUSH DWORD PTR SS:[EBP+10]</td>
</tr>
<tr>
<td>F75 0C</td>
<td>PUSH DWORD PTR SS:[EBP+0C]</td>
</tr>
<tr>
<td>F75 08</td>
<td>PUSH DWORD PTR SS:[EBP+08]</td>
</tr>
<tr>
<td>0945 FC</td>
<td>MOV DWORD PTR SS:[EBP-4], 0x04000000</td>
</tr>
</tbody>
</table>

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`.

<table>
<thead>
<tr>
<th>0012F90C</th>
<th>00000000</th>
<th>0012F916</th>
<th>0012F91C</th>
<th><code>lpStartAddress</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0012F974</td>
<td>00000000</td>
<td>0012F970</td>
<td>0012F976</td>
<td><code>kernel32.CreateThread</code></td>
</tr>
<tr>
<td>0012F984</td>
<td>00010CFC</td>
<td>0012F980</td>
<td>0012F986</td>
<td><code>kernel32.CreateThread</code></td>
</tr>
<tr>
<td>0012F994</td>
<td>0C810CFC</td>
<td>0012F990</td>
<td>0012F996</td>
<td><code>kernel32.CreateThread</code></td>
</tr>
</tbody>
</table>

Put a breakpoint on the thread function (`lpStartAddress`) and resume the execution (F9).
When a breakpoint at the thread function is hit step over (F8) until a call to ECX. As you see ECX points to the memory of the .text section of the original executable (voiyhabs.00403850). This is a good indicator that you are jumping to the OEP. Step into the call (F7).

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

Starting at the OEP address select a group of instructions. Next right-click on them and from the context menu choose ‘Analysis->During next analysis, treat selection as->Command’.
Now click on them once again and from the context menu choose ‘Analysis->Analyse code’.
Now you can clearly see that you are most likely at the OEP - typical function prologue with the EBP based stack frame followed by later call to `GetModuleHandleA`. Moreover you just jumped from a code in the allocated memory block to the initial executable section: another indicator that you are at the original entry point.

Next you will dump the process image and reconstruct the IAT table. To dump the process use OllyDumpEx plugin (as described previously).
Now try to reconstruct the Import Address Table (RVA of the OEP: 0x3850) by using ImpRec. This time you might see some invalid imports after clicking Get Imports and Show Invalid. Right-click on each of them and from the context menu choose Cut thunks(s). After all invalid pointers are resolved, use Fix dump to fix the dumped executable.
In this exercise you have unpacked the executable of a real malware sample, protected with a more complex packer. You have achieved this by first tracking the memory allocation operations and then following the unpacking stub in a newly allocated heap. It is worth noting that this isn’t the only way of unpacking this executable nor is it the quickest method. As goes for any packed executable there are many ways of unpacking code and reaching the OEP.
4. Anti-debugging techniques

4.1 Anti-debugging and anti-analysis techniques

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

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

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

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

- *API related techniques* – those techniques use the fact that calls to certain API functions would return different result depending on whether the application is being debugged or not. Examples of such functions are *IsDebuggerPresent* or *OutputDebugString*.

- *Checking flags* – certain flags set by an operating system in process’s data structures are different when the process is being debugged. Examples of such flags are *NtGlobalFlag* and *IsDebugged* flag in (PEB).

- *Searching for breakpoints* – it is possible for a process to search for breakpoints in its current address space. This applies for software, hardware and memory breakpoints.

- Searching for processes (or open windows) of popular debuggers and other analysis tools (e.g. *Wireshark*, *Regshot*, *Process Explorer*).

- *Time based checks* – a malicious code can check how much time elapsed between two different parts of the code. If the time delay is too big it will be assumed that the application is being debugged.

- *Self-debugging* – a clever anti-debugging technique in which the malware starts debugging its own processes making the analyst unable to attach a debugger to them. This technique was used by Zero-Access trojan\(^31\).

---


Except trying to detect the debugger, the malicious code is frequently also trying to detect virtualization environments using a subset of the following techniques:\(^{32}\):

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

One of the countermeasures for anti-debugging is to use special plugins for OllyDbg like `aadp4olly`\(^{33}\) and `Olly Advanced`\(^{34}\).

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

<table>
<thead>
<tr>
<th>Plugins</th>
<th>Options</th>
<th>Window</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 aadp4olly</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>2 Olly Advanced</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>3 AnalyzeThis</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>4 Asm2Clipboard</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>5 Bookmarks</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>6 Labelmaster</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>7 CDbgScript</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>8 OllyDumpEx</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
<tr>
<td>9 OllyDbg PE Dumper</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
</tr>
</tbody>
</table>

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

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

---

\(^{32}\)On the Cutting Edge: Thwarting Virtual Machine Detection


\(^{34}\)Olly Advanced https://tuts4you.com/download.php?view.75 (last accessed 11.09.2015)
However even when you are using such plugins you need to still be cautious for anti-VM techniques and more sophisticated anti-debugging techniques. One way to check if the malicious code is using anti-VM or anti-debugging techniques is to try and run it freely (under debugger) and then using a behavioural analysis techniques to check if the malware behaves as expected.

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

### 4.2 Dyre - basic patching with OllyDbg

A recent version of the Dyre trojan uses an interesting anti-sandbox mechanism. It checks the number of processor cores visible by the system\(^3\) and if it determines that the number of cores is less than two it stops the execution. Since the most modern systems run on multi-core CPUs it won’t affect them while it would still prevent the execution on a poorly configured sandbox running on virtual machines with only one CPU attached.

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

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

---

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

Now open the unpacked Dyre in OllyDbg.

If you step over (F8) a few times you will notice that at the first jump instruction (J8) the program is jumping to the ExitProcess routine.
If you take a closer look at the code just before the jump, you notice that Dyre is checking the number of CPU cores as pointed by the Process Environment Block (PEB). The Process Environment Block is a special system structure containing various information about the running process. It is stored in user space memory and pointed to by the FS segment.

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

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

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

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

A typical process hollowing scheme is executed as follow:

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

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

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

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

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

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

5.2.1 First stage

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

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

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

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

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

Resume the execution (F9). The execution should break on `WriteProcessMemory`. Take a look at the arguments passed via stack to the `WriteProcessMemory` function.
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 (0x21300003) you will see typical PE headers with likely some unpacked code.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x130000</td>
<td>60 59 9E 00 00 00 00 00 00 00 00 00 04 00 00 00 FF FF 00 00</td>
<td>PE Dat...</td>
<td></td>
</tr>
<tr>
<td>0x130010</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x130020</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x130030</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x130040</td>
<td>08 IF 5E 2E 0E 04 09 03 01 04 0C 21 54 6C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x130050</td>
<td>69 73 28 7E 65 34 28 7E 72 66 50 59 34 65 34</td>
<td>is progra...</td>
<td></td>
</tr>
<tr>
<td>0x130060</td>
<td>74 00 62 65 28 72 7E 72 66 50 59 65 6E 2E 65</td>
<td>t be run i...</td>
<td></td>
</tr>
<tr>
<td>0x130070</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x130080</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x130090</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1300A0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1300B0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1300C0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1300D0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Resume the execution (F9) two times until you break on `SetThreadContext` function. This function is used by the malware to set a new entry point address of the initial thread of the suspended process (before it will be resumed). Write down the address of the context structure put on the stack (0x420000 - `pContext`) and follow it in the dump.
The entry point of the newly created process is stored in its EAX register\(^{40}\). Its value can be read from the context structure at the address pContext+0xB0\(^{41}\). In this case the entry point address is 0x00401000 (remember about little-endian notation). Write down the address of the entry point, it will be needed later.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>RST1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00420000</td>
<td>07 00 01 00 00 00 00 00</td>
<td>.0 .............</td>
</tr>
<tr>
<td>00420010</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420020</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420030</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420040</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420050</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420060</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420070</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420080</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00420090</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>004200A0</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>004200B0</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>004200C0</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>004200D0</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>004200E0</td>
<td>00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

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

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

To override the child process memory you can use Process Hacker\(^{42}\) tool.

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

---

\(^{40}\) EAX value in the context of newly created process, don’t mistake it with the EAX value in OllyDbg which is the value of EAX register in the context of currently debugged process.


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

In the new window go to the entry point address at offset 0x1000 (addresses are relative to 0x400000).
Write down the first two bytes at this offset (0x64A1) and override them with (0xEBFE). Click Write and close the window.

Now switch back to OllyDbg and step over (F8) till return from ResumeThread function.
Now you can minimize OllyDbg. In the Process Hacker window you can also notice that the child process was resumed and is now using a considerable amount of CPU time. This is the result of the endless loop you created in this process. Note the process identifier (PID) of the child process (in decimal).

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

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

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

5.2.2 Second stage

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

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

Next put breakpoints on the following functions:

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

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

Continue the execution (F9) till you land at the `GetThreadContext` breakpoint. In this case the malware uses this function to check the address of an entry point of the `EXPLORER.exe` process.
Note the address of the pContext structure (in this example it is 0x12FAAC) and follow it at the dump. Next step over (F8) till return from GetThreadContext and read EXPLORER.exe entry point from the pContext+0xB0 address. Alternatively you can just find explorer.exe executable on the disk and check its entry point address with some PE editor (e.g. CFF Explorer). In this situation EP is located at the address 0x4AA8DF.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012F84C</td>
<td>90 00 00 98 FB 12 00 00 00 23 C7 00 0C FB</td>
<td>.w..P1.0...F.</td>
</tr>
<tr>
<td>0012F850</td>
<td>90 00 10 98 00 00 00 0E 00 00 00 00 00 00 00 00</td>
<td>.w.</td>
</tr>
<tr>
<td>0012F854</td>
<td>90 00 20 99 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>.w.</td>
</tr>
<tr>
<td>0012F858</td>
<td>90 00 30 99 FB 12 00 00 00 00 00 00 00 00 00 00 00</td>
<td>.w.</td>
</tr>
<tr>
<td>0012F85C</td>
<td>90 00 40 99 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>.w.</td>
</tr>
<tr>
<td>0012F860</td>
<td>90 00 50 99 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>.w.</td>
</tr>
<tr>
<td>0012F864</td>
<td>90 00 60 99 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>.w.</td>
</tr>
<tr>
<td>0012F868</td>
<td>90 00 70 99 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>.w.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012F984</td>
<td>00100000</td>
<td>CALL to WriteProcessMemory from 00100048</td>
</tr>
<tr>
<td>0012F988</td>
<td>00100000</td>
<td>hprocess = 0000001C (window)</td>
</tr>
<tr>
<td>0012F98C</td>
<td>00100000</td>
<td>Address = 400000</td>
</tr>
<tr>
<td>0012F990</td>
<td>00000000</td>
<td>BytesToWrite = 1B0 (256.)</td>
</tr>
<tr>
<td>0012F994</td>
<td>00000000</td>
<td>bytesWritten = NULL</td>
</tr>
<tr>
<td>0012F998</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>0012F99C</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>0012FA00</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>0012FA04</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>0012FA08</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>0012FA0C</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

Knowing that the above buffer will be written to the exact address of an entry point, this time you will do 0xEBFE trick before the memory is written to the child process. Please note that this step should be done before stepping over the WriteProcessMemory function.
Select the first two bytes of the source buffer \((E8\ C2)\) and press Ctrl+E. Replace them with bytes \(EB\ FE\).

Next step over \texttt{WriteProcessMemory} function till the user code. You should land at \texttt{TEST EAX, EAX} instruction.

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

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

Minimize OllyDbg window and check in Process Hacker if \textit{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.
Now you are at the entry point of the code injected to EXPLORER.exe but this still isn’t the main Tinba payload. To reach the payload put a breakpoint on `OpenFileMappingA` and resume the execution (F9).

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

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

To reach the final payload step over (F8) the return instruction (RETN). Don’t worry if the address would be different from the one on the screenshot. What’s important is that after a return you should see a group of four call instructions.
Now create a snapshot called 'Tinba'. This snapshot will be used in the later exercises.
6. Introduction to scripting

6.1 Introduction to OllyDbg scripting

When debugging malicious code you sometimes encounter the problem of repetitive and/or tedious tasks. This might be the case when unpacking pieces of code obfuscated with the same packer or performing multiple repetitive actions in some malicious code. One of the solutions to this problem is to automate certain tasks through scripting. In OllyDbg you can do this using the ODbgScript plugin\(^43\).

A detailed reference about the scripting language is provided with ODbgScript package in the README.txt file\(^44\). In general the language is very similar to the assembly language with a few additional commands. Every operation you can do in OllyDbg (except functions provided by other plugins) you can also do in using script.

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

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

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

The code below presents an example script which first prints the result of XORing EAX with EDX (without affecting the values in the registers) and then prints the first ten Fibonacci numbers\(^47\) in a loop.

```plaintext
; printing result of EAX^EDX
var result
mov result, eax
xor result, edx
log result

; fibonacci(10)
var i, n, k
mov i, 3
mov n, 1
```

---


mov k, 1
log "1: 1"
log "2: 1"

fibonacci_loop:
xchg n, k
add k, n
eval "(i): {k}"
log $RESULT,""
add i, 1
cmp i, 10.
jbe fibonacci_loop

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

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

### 6.2 Decoding hidden strings in Tinba

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

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

Next step into the seventh call instruction (F7).
Take a look at the first call instruction. You should notice two interesting things about it. Firstly, the call is jumping into a middle of an instruction (there is no disassembled instruction at 0x7717B7). Secondly, instructions after this call don’t make much sense.

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

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

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

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0021F724</td>
<td>14 F8 21 00 07 EE 53 77 0E 60 53 77 0E EC 53 77 0E 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>EOL 21 00 07 EE 53 77 0E 60 53 77 0E EC 53 77 0E 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0021F729</td>
<td>00 00 55 75 15 92 00 00 64 F8 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>55 75 15 92 00 00 64 F8 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

And step over (F8) a call.

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

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

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

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>INSTRUCTION CODE</th>
<th>INSTRUCTION</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>50</td>
<td>PUSH EAX</td>
<td>pushing dst. address (for decoded data)</td>
</tr>
<tr>
<td>0x1</td>
<td>87 44 24 04</td>
<td>XCHG DWORD PTR [ESP+4], EAX</td>
<td></td>
</tr>
<tr>
<td>0x5</td>
<td>6A ?</td>
<td>PUSH &lt;n&gt;</td>
<td>pushing data length</td>
</tr>
<tr>
<td>0x7</td>
<td>8B ? 00 00 00</td>
<td>CALL (0xB+n)</td>
<td>pushing src. address (5) onto the stack</td>
</tr>
<tr>
<td>0xB+n</td>
<td>&lt;variable length encoded data&gt;</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

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

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

The algorithm would be as follow:

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

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

```javascript
var base
var labels

; checking memory base of Tinba payload
gmemi 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
```
Commands used in this script were:

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

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

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

Then open ODbgScript **Script Window** and **Log Window**.
Next load `script.osc` in Script Window by right-clicking it and choosing Load Script->Open.

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

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

Now that you know all encoded strings you can do typical string analysis to guess some of Tinba’s functionality. For example on the strings list you can find strings such as `data_before`, `data_end`, `data_inject`, `data_after` which tell that Tinba is using webinjects technique known from other banking trojans.
Each printed line has the following message format:

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

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

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

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

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