Advanced artefact analysis
Advanced static 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 6
2. Introduction to IDA Pro 7
  2.1 Opening and closing samples 7
  2.2 IDA Pro interface 11
  2.3 Exercise 13
  2.4 Disassembly view 16
  2.5 Basic navigation 20
  2.6 Exercise 24
  2.7 Functions 24
  2.8 Enhancing assembly code 29
  2.9 Exercise 42
  2.10 Exercise 43
  2.11 Summary 43
3. Recognizing important functions 44
  3.1 Using call graphs 44
  3.2 Exercise 50
  3.3 Using cross references 54
  3.4 Exercise 63
  3.5 Summary 63
4. Functions analysis 64
  4.1 Analysis of network function 64
  4.2 Analysis of WinMain 78
  4.3 Analysis of thread function 84
  4.4 Exercise 94
  4.5 Summary 95
5. Anti-disassembly techniques 96
  5.1 Linear sweep vs. recursive disassemblers 96
  5.2 Anti-disassembly techniques 98
5.3 Analysis of anti-disassembly techniques

5.3.1 Analysis of a call to loc_40101A

5.3.2 Analysis of a call to loc_401045

5.3.3 Analysis of a call to sub_401065

5.3.4 Analysis of a call to sub_4010B2

5.3.5 Analysis of a call to sub_40116D

5.4 Exercise

6. Training summary

Appendix A: Answers to exercises

Exercise 2.3

Exercise 2.6

Exercise 2.9

Exercise 4.4

Exercise 5.4

Exercise 6.4
## Main Objective
The main goal of this training is to teach the participants all aspects of a static artefact analysis. During the first part they are taught how to approach the disassembly of binary code, recognize basic programming language structures and navigate through the disassembled code. This part is conducted with non-malicious binary code for safety reasons.

Second part of the exercise focuses on characteristic patterns in assembly code that can be found in popular artefacts. The participants will learn to quickly recognize these common patterns which adds to the effectiveness of their further work.

Eventually, the instructor guides the class through real-world samples of known threats while gradually increasing level of their complexity.

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

## Total Duration
8-12 hours

## Frequency
Once for each team member
1. Training introduction

In this training, students will learn the fundamentals of advanced static analysis. During the training, students will have an opportunity to disassemble live malware samples with the help of IDA Free® disassembler to determine their functionality and gain additional knowledge of how malicious code works.

During the first part of the training, students will be introduced to the IDA disassembler, which is currently most widely used disassembler. They will learn how to navigate through the code, use different views and functions, as well as how to enhance and comment disassembled code. Next, students will learn how to find key parts in the code and how to analyse disassembled functions. Finally, they will learn basic anti-disassembly techniques.

After the training, students will have learned:

- How to effectively use IDA to disassemble malicious code
- How to customize IDA workspace
- How to create call graphs and use them to find important functions
- How to use cross references
- How to analyse disassembled functions
- How to recognize some anti-disassembly techniques

Students should be familiar with the material presented during the first part of the training “Introduction to Advanced Artifact Analysis” before starting this exercise, as it contains key knowledge required through the whole course. At this point, students should be already familiar with x86 assembly language and principles of malicious artefact analysis. Students should also have knowledge about Microsoft Windows system internals. Prior completion of second part “Advanced dynamic analysis” training is also advisable.

In this training you will be using real malware samples. Since only static analysis will be performed and samples won’t be executed, it is not necessary to restore a clean snapshot after each exercise. However, in case you accidentally execute a malware sample, you should perform all analyses in an isolated environment. As a matter of principle: execute caution when dealing with malware samples at all times!

---

2. Introduction to IDA Pro

During the first part of the training, you will learn how to use IDA Free disassembler, which is a powerful tool allowing an analyst to effectively analyse disassembled code. In this training you will examine the binary of the popular SSH client – PuTTY⁴. Since this code is not malicious, you don’t need to worry about accidentally executing it.

2.1 Opening and closing samples

Copy putty.exe sample to the Desktop and start IDA Free disassembler.

At the beginning of the session you will be presented with the About window. Just click Ok.

![About](image)

In the next window you will be asked whether to disassemble a new file or just start IDA. Click Go button. You can also check “Don’t display this dialog box again” option to prevent IDA from displaying this dialog each time.

---

You will be now presented with the main IDA Free workspace window.

Open **putty.exe** file by choosing **File->Open...** or dragging putty.exe binary onto the disassembler window.

Now you will be presented with the **Load a new file** window. In this window, the analyst can choose various options regarding how IDA should open and analyse selected sample.
When opening a new sample, IDA tries to recognize sample’s file format and properly set default options. At the top of the window there is a list with file formats recognized by IDA. Here you can see that IDA correctly recognized putty.exe as a Portable executable for 80386 file. However, IDA still gives you the chance to load putty.exe as a MS-DOS executable or plain binary file.

If you had chosen to load putty.exe as a Binary file, IDA would have loaded file contents at given memory address (specified with Loading offset parameter) without doing extensive analysis. For example it wouldn’t try to read PE headers nor recognize the import address table (IAT) or check entry point address.

The next option is a drop-down list with processor types. Since assembly code for various processors differs you may choose here what processor type IDA Pro should use when disassembling binary.

Below, there are various other options telling IDA how it should analyse binary. In most cases when analysing typical Portable Executable (PE) binaries you can leave the default options selected. Click on each of the “options” buttons to see the parameters of analysis that IDA Free offers.

In this exercise, leave all default options set as shown on the screenshot and press Ok button.

Now IDA will start disassembling and perform an initial (background) analysis process. It might take several seconds or even a few minutes for larger and more complex binaries. When the analysis is finished you will see an appropriate message in the message log box at the bottom of the window.
Now take a look at the directory where `putty.exe` is located. You should notice four new files: `putty.id0`, `putty.id1`, `putty.nam` and `putty.til`. Those are database files where IDA stores runtime information about current analysis (disassembled code, comments, labels, etc.).

When finishing the analysis by either quitting IDA Pro or selecting `File->Close`, IDA will ask whether to pack database files (`Pack database (store)` - recommended) or leave unpacked files. You can also choose to finish analysis without saving any results (`DON'T SAVE the database` option).
If you choose to pack the database, a single putty.idb file is created instead of four database files. To continue the analysis later just open this file in IDA. If you are restoring clean snapshots of the virtual machine, remember to preserve .idb files to not lose the results of your work.

2.2 IDA Pro interface
First, load putty.exe as described in the previous step (or open a saved session). After IDA finishes its analysis, you are presented with the default IDA workspace consisting of various windows and other elements. At a first glance IDA interface may look quite complicated but it will become much clearer when you get to know it better.
The central part of the workspace is occupied by the *Windows area* (4). IDA uses multiple windows to present various types of information about the disassembled binary. Among the most frequently used windows are:

- *IDA View-A* – window with disassembled code
- *Hex View-A* – hex view of disassembled binary
- *Imports* – functions imported in Import Address Table
- *Functions* – list of local functions recognized by IDA in disassembled code
- *Strings* – list of strings found in executable

To switch between windows you can use *Windows tabs* (3). If you accidentally close any of the windows you can bring it back using the *View->Open sub views* menu or a corresponding shortcut key.
Right above the window tabs there is an *Overview navigator* (2) panel. This panel is used to present your current location in the disassembled code/hex view within the address space of the loaded sample.

Switch to *Hex View*-A window and scroll up and down to observe how it changes your current position (pointed by the yellow arrow). Note that different colours are used to indicate different types of data at given address (e.g. dark blue means regular function)\(^5\).

The last three elements of the IDA workspace are: *toolbars area* (1) – to quickly access certain IDA functions, *graph overview* (5) – to quickly navigate disassembled code and the *output window* (6) – to present various information outputted by IDA.

### 2.3 Exercise

*Take some time to switch between the different data views (windows) and check what type of data is presented in each of them.*

- Name a few functions imported by PuTTY executable.
- What sections are present within executable?
- What do strings tell you about this binary?

---

\(^5\) Full colours legend can be checked in Options->Colors...->Navigation band menu.
One of the problems with the default layout of the IDA Free is that rarely used functions occupy too much space while most frequently used ones (disassembly window and functions window) have too little space left. We will now customize the default layout to use available space more effectively. Additionally it always helps to perform an analysis on a bigger screen whenever possible.

Let's get rid of some of the toolbars first (toolbar functions can be accessed through menus or shortcuts). Right click on the toolbars (1) and uncheck unnecessary toolbars in the context menu.

It is up to you which of the toolbars you want to use. You can even decide to remove all toolbars. In the example below we display the following toolbars:

- Main
- Files
- Navigation -> Jumps
- Navigation -> Navigation
- Navigation -> Graph overview
- Disassembly -> Cross references
- Graphs

It is also worth resizing output window (6), which is rarely used during analysis.
Next, rearrange all the windows and toolbars to give IDA a cleaner look. Since the functions window and disassembly window will be very frequently used, it is good to have them on top. Moreover, it is also good to maximize IDA window if you haven’t done so already.

When you are satisfied with the layout, save it using Windows->Save desktop option.
Now whenever you start a new analysis or your layout gets messed up you can quickly restore it using Windows- >Load desktop option.

2.4 Disassembly view
Central to IDA Pro is the assembly view (IDA View-A). In the assembly view, IDA presents disassembled code along with all recognized functions.

There are two types of the assembly view: text view and graph view. To switch between the text and graph views, click on the assembly view (IDA View-A) and press the spacebar.

In text view, you can see a linear listing of all disassembled instructions. Text view is useful when you want to analyse parts of the code that IDA hasn’t recognized as proper functions.

Notice the dashed and solid lines on the left side of the text view. They are used to indicate conditional and unconditional jumps, respectively. If you click on jump destination, IDA will highlight destination label as well as a corresponding arrow.

The second type of assembly view is graph view. In the graph view, as the name suggests, IDA presents disassembled code in the form of a graph, where nodes are represented by blocks of disassembled code and lines are branches and unconditional jumps. For each recognized function, IDA creates a separate graph; that is, each graph represents only a single function. Graph view is useful to quickly figure out the execution flow of a function.
Different colours of the lines are used to indicate different types of code transitions:

- Green – preceding jump is taken
- Red – jump is not taken
- Blue – normal branches (unconditional jump or just transition to the next instruction)

You can also hover the cursor over branches. IDA will show a small hint window with a code snippet about where a branch is leading. This is useful if a branch leads to a location outside the current screen.

Sometimes you will want to get a higher level grasp of the code flow in the function. In such a situation, it is useful to zoom out the graph view with Ctrl + Scroll button.
Another very useful feature of IDA is its highlighting capability. You can click on almost any name (register, operation, variable, comment, etc.) and IDA will highlight every other occurrence of this name. For example, you can highlight push/pop operations to track registry changes or highlight a particular registry to track which instructions are changing it.

```plaintext
loc_44898A:
  ; CODE XREF: WinMain(x,x,x,x)+A01j
  ; WinMain(x,x,x,x)+A51j
  mov    eax, ds:dword_45D4FC
  push   eax
  mov    [ebp+68+var_C], ebx
  mov    [ebp+68+nHeight], ebx
  mov    dword_47E534, eax
  call   sub_40F207
  cmp    ecx, ebx
  pop    eax, ebx
  mov    dword_47E540, ebx
  jz     short loc_448933
  mov    eax, [eax+48h]
  mov    dword_47E540, eax

loc_448933:
  ; CODE XREF: WinMain(x,x,x,x)+FC1j
  push   ebx
  push   73h
  push   dword_47E53C
  call   sub_4025A5
  push   dword_47E53C
  push   ebx
  call   sub_411096
  mov    edi, [ebp+68+nCount]
```
Advanced artefact analysis
Advanced static analysis

By default when viewing code in graph view, IDA doesn’t show instruction addresses. If you would like to see instruction addresses while staying in graph view choose Options->General... and select Line prefixes option.

Now when viewing code in graph view, you will also see instruction addresses. For convenience you will use this in the rest of the document so you could always easily navigate to the part of the code pointed by the screenshot.
At the end, it is worth mentioning that if IDA doesn’t recognize part of the code as a proper function, graph view will be unavailable. You can recognize this situation when instruction addresses in text view are red and it is impossible to switch to graph view. You will see how to deal with this situation later.

2.5 Basic navigation
When reverse engineering a disassembled binary, you will spend most of your time trying to figure out which code parts are important and what each function is doing. Thus it is crucial to learn how to navigate through the code effectively and quickly.

One of the easiest ways to navigate through code is to use the functions window. Just find an interesting function name and double click it to move to this function instantaneously. For example, go to the sub_4457D6 function.

Moreover, if the functions list is long you can click the functions window and start typing a function name. At the bottom of the window, you can observe the characters you have typed and if a function with a given name exists, it will be selected automatically.
As you may have noticed, some of the functions in the functions list are named differently than \textit{sub\_XXXXXX}. Examples of such functions are \_fwrite, \_strcat, \_sscanf, etc. With a few exceptions those are library functions statically linked to the binary during compilation.

If you resize the functions window, such functions will be marked with capital L in sixth column\(^6\).

Moreover if you take a look at the \textit{overview navigator} bar, library functions are marked with cyan colour.

Statically linked functions are pretty much indistinguishable from normal code. To distinguish them, IDA uses a special FLIRT engine\(^7\), which uses the signatures of functions from popular and well-known libraries. More advanced users can try to extend FLIRT with their own signatures; however, this topic is not covered in this training.

\(^6\) To check meaning of other columns refer to https://www.hex-rays.com/products/ida/support/idadoc/586.shtml (last accessed 11.09.2015)

Go back to the WinMain function and look at the group of four calls at the beginning of the routine.

There are four types of calls you will see most frequently in disassembled code:

- Calls to local routines (e.g. `call sub_XXXXXX`)
- Calls to the address stored in memory (e.g. `call dword_XXXXXX`)
- Calls to location pointed by register or local variable (e.g. `call eax`)
- Calls to WinAPI or other library functions (e.g. `call ds:CreateProcessA`)

The most troublesome are usually calls to addresses stored in memory and calls to locations pointed by register. This is because determining the destination address of such a call usually requires more detailed code inspection and good code understanding.

In the above example, we see three calls to local functions (`sub_44B2C5, sub_441535, sub_44AE44`) and one call to WinAPI function `InitCommonControls`. To quickly navigate to `sub_44B2C5`, double click its name.

In a similar way, you can also click on data offsets to move to the location of the data in memory. For example, double click on `aWs2_32_dll`, a name given by IDA to the string “ws2_32.dll” defined in memory in section .rdata at the address 0x473EF0.
Now to go back to WinMain quickly press the <Esc> key twice. It will move you back to the WinMain routine. Respectively, to move forward, press <Ctrl> + <Enter> and you will be back in sub_44B2C5. You can also use the Jumps toolbar:

When dealing with large and complicated functions, it is useful to use the small Graph overview window to navigate within the code of a function. The Graph overview window should be present whenever disassembly view is active and its current mode is graph view. If you accidentally close Graph overview window, open it using View -> Toolbars -> Navigation -> Graph overview.

If the function graph is too big to fit your current disassembly view size, your current position will be marked with a small dotted rectangle within the Graph overview window. This rectangle will change size whenever you zoom in or out of the function graph.

You can move this rectangle or just click on any part of the Graph overview to move to the chosen part of the function. Now try to inspect function sub_44F102 using only the Graph overview window.
Often, you know the particular memory address that you would like to inspect but you don’t know which function it belongs to. In such situations, you can use Jump to address feature (Jump -> Jump to address... or press <g>).

In this dialog, you can enter any hexadecimal address within the memory range of analysed binary (e.g. 0x440C74) or any name recognized by IDA like a function name or certain label (e.g. sub_40E589, loc_40E5CA).

2.6 Exercise
Take some time to navigate through the various functions of disassembled PuTTY binary.

- Find function sub_4497AE. What API calls are made within this function?
- Go to the address 0x406AFB. To which function does this address belong?
- Go to the address 0x430EAB. Is there anything special about the instructions stored at this address?

2.7 Functions
When loading a new binary sample, IDA performs an extensive auto analysis. During this process, IDA tries to find all the functions defined in assembly code as well as determine their arguments, variables or calling convention. Each detected function, whether it is a normal function or a library function, is listed in functions window.

The WinMain function provides a good example of IDA’s analysis capabilities:
Each function begins with a function prototype header (1). In this example, IDA recognized the function prototype, function calling convention (\textit{stdcall}) and arguments types (\textit{HINSTANCE, HINSTANCE, LPSTR, int}).

However, IDA doesn’t always properly recognize function prototypes. Consequently, if you obtain additional information about the calling convention, arguments or return value during analysis, you can edit the function prototype by clicking on the function name and choosing \textit{Edit->Functions->Set function type...} from the menu.

\begin{verbatim}
; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
_WinMain@16 proc near

Please enter interesting

Please enter the type declaration: int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)

OK Cancel Help

var_F8= 
var_D8= WndClass
◽= nk
Rect= SQ 
var_2A= 
var_28= 
var_1C= 
var_18= ; word ptr 10h
\end{verbatim}

This provides IDA with additional information about the function and help analyse rest of the code.

Below the function header is a list of local variables (2) and function arguments (3). IDA tracks how those variables are used in the code and then tries to suggest their names. For example, if a variable is used only to store result of a call to \textit{GlobalAlloc()}\(^8\), IDA might name it “\textit{hMem}”. If IDA is unsuccessful with naming variables, it will give them ordinary names such as \textit{arg\_0, arg\_4, etc.}, for arguments and \textit{var\_4, var\_8, etc.}, for local variables.

Notice the offsets to the right of the variable names (5). The offsets tell the position of a variable on the stack in reference to the stack frame of the function. This is also how you can distinguish local variables from function arguments. Local variables will always have negative offsets while function arguments will have positive offsets.

\begin{verbatim}
<table>
<thead>
<tr>
<th>variable</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg_8</td>
<td>ebp+10</td>
</tr>
<tr>
<td>arg_4</td>
<td>ebp+C</td>
</tr>
<tr>
<td>arg_0</td>
<td>ebp+8</td>
</tr>
<tr>
<td>ret. addr.</td>
<td>ebp+4</td>
</tr>
<tr>
<td>ebp</td>
<td>ebp</td>
</tr>
<tr>
<td>var_4</td>
<td>ebp-4</td>
</tr>
<tr>
<td>var_8</td>
<td>ebp-8</td>
</tr>
<tr>
<td>var_C</td>
<td>ebp-C</td>
</tr>
</tbody>
</table>
\end{verbatim}

Additionally, if you double click on any of the variable names, IDA will open a \textit{stack frame} window for the current function. Using stack window, you can get a better understanding of how variables and arguments are positioned on the stack. At this point you should also remember that what IDA sees as a group of separate variables might as well be a structure or some array.

\small
\(^8\) Allocates specific number of bytes from the process heap and returns handle to the allocated memory object.
Another important thing to know is how IDA references variables in the function body. This differs depending on whether the function uses an EBP-based stack frame or an ESP-based stack frame. In functions with EBP-based stack frames, all variables are referenced relative to the EBP register. *WinMain* or *sub_42FCAD* are examples of such functions.

```
0042FCAD var_a= dward ptr -4
0042FCAD arg_0= dward ptr 8
0042FCAD
0042FCAD push ebp
0042FCAD mov ebp, esp
0042FCB0 push ecx
0042FCB1 push ebx
0042FCB2 lea eax, [ebp+var_4]
0042FCB5 push eax
0042FCB6 mov eax, [ebp+arg_6]
0042FCB9 xor ebx, ebx
```

You can recognize EBP-based functions by the typical function prologue in which in the first instruction EBP register is pushed onto the stack (*push ebp*).

The second type of functions are those with an ESP-based stack frame. In such functions, the EBP register isn’t preserved and all variables are referenced relative to the ESP register. Example of such a function is *sub_40486C*.

---

In some situations, IDA doesn’t properly recognize functions. Sometimes, this requires correcting the code first – either manually or by a custom script, but sometimes it is enough to tell IDA to create a function at the given address.

Example of a function that IDA did not properly recognize is code at address 0x430E38:

```assembly
.text:00430e3e        leave
.text:00430e3f        ret
.text:00430e37        sub_430433
.text:00430e37        endp
.text:00430e38        ; -----------------------------
.text:00430e38        push ebp
.text:00430e39        mov ebp, esp
.text:00430e39        push ebx
.text:00430e3c        mov ebx, [ebp+8]
.text:00430e3f        mov eax, [ebx+16h]
.text:00430e42        push esi
.text:00430e43        xor esi, esi
.text:00430e45        sub eax, esi
```

Fortunately, this code doesn’t require any changes and is not using any anti-disassembly techniques. To create a function, click on the first instruction (push ebp) and choose Edit->Functions->Create function...
IDA should now recognize this part of the code as a proper function and you should be able to switch to the graph view.

Unfortunately, this won’t always work – especially if malware is using anti-disassembly techniques. In such case you may do analysis using only text view mode or try to correct code manually.

Additionally, if you believe a function was wrongly recognized, you can click on the function’s name in the code and choose Edit->Functions->Edit function... to change various function parameters like the function’s start or end address. To get more information about those parameters refer to IDA help file. Moreover, if for some reason you would like to delete a function, just click on its name in the code and choose Edit->Functions->Delete function.
2.8 Enhancing assembly code

When analysing disassembled code, it is important to document all of your findings properly. This will gradually make the code easier to understand and track its execution flow. It will be also helpful if you decide to return to the analysis later or share your results with someone else.

Fortunately IDA offers a lot of means to document code and improve its readability, such as:

- Editing numbers format and using symbolic constants
- Renaming functions, variables, names
- Adding comments
- Changing graph node colour
- Grouping one or several nodes

To show how to use the features that can improve assembly readability, go to the function sub_44D262 (0x44D262). This function takes one unknown argument (arg_0) and uses a few variables, two of them IDA named FileName and FindFileData.

```
0044D262 sub_44D262 proc near
0044D262
0044D262 FIndFileData= _WIN32_FIND_DATAA ptr -270h
0044D262 FileName= byte ptr -130h
0044D262 var_28= dword ptr -28h
0044D262 var_8= dword ptr -8
0044D262 var_4= dword ptr -4
0044D262 arg_0= dword ptr 8
0044D262
```

In the function body you will see a few API calls to functions such as GetWindowsDirectoryA, FindFirstFileA, FindNextFileA, GetProcAddress, etc.
There are also some unknown calls to an address stored in registers:

```
004D260 push    edi
004D26E push    107h ; uSize
004D273 lea    eax, [ebp+FileName]
004D279 push    eax ; lpBuffer
004D27A call    ds:GetWindowsDirectoryA
004D280 lea    eax, [ebp+FileName]
004D286 push    offset asc_4747D4 ; "\\"
004D288 push    eax ; char *
004D2C1 lea    eax, [ebp+FindFileData]
004D2C7 push    eax ; lpFindFileData
004D2C8 push    esi ; hFindFile
004D2C9 call    ds:FindNextFileA
004D2CF test    eax, eax
```

And calls to functions pointed by some global variable:

```
004D384 push    0F000000h
004D389 push    1
004D38E push    edi
004D393 push    edi
004D39D lea    ecx, [ebp+var_4]
004D39F push    ecx
004D3A1 call    eax
004D3A3 test    eax, eax
```

Such calls make analysis more difficult because you don’t know where those calls are leading to.

To start improving code readability, first look at the graph nodes with calls to `GetProcAddress`.

```
In total, there three such calls in `sub_44D262`. You can read the name of the function being resolved from the value pushed onto stack (`CryptAcquireContextA`). After the call to `GetProcAddress`, the result is saved to the memory location pointed by `dword_47E0C8`.

You can rename this memory location by clicking on `dword_47E0C8` and pressing <n> key. Rename it to `CryptAcquireContextA`.

```
```
After pressing Ok you will be informed that name exceeds 15 characters. Ignore this warning and click Yes.

Now the code should look like this:

```
0044D311 push offset aCryptAcquireContext : "CryptAcquireContextA"
0044D316 push eax ; hModule
0044D317 call esi ; GetProcAddress
0044D319 mov CryptAcquireContextA, eax
0044D31E mov eax, dword_47E0D0
0044D323 jmp short loc_4AD32B
```

Repeat this step for the remaining two calls to GetProcAddress in sub_44D262 (CryptGenRandom, CryptReleaseContext). Make sure that you rename the memory locations exactly the same as the names of the resolved functions.

Next, scroll down to the location where the calls to the functions pointed by memory address (call dword_XXXXXX) were previously. Notice how they changed?
Now that IDA knows a little more about what functions are called at those locations, let it reanalyse the code. To do this, go to the IDA Options dialog (menu Options->General...), switch to Analysis tab and click Reanalyze program. Wait for IDA to finish the analysis and close the IDA Options dialog. Notice how IDA has now added additional comments and renamed some variables!
Now scroll to the location 0x44D391 where there is a call to eax:

IDA still doesn’t know where this call is made, but if you highlight eax register and take a look a few blocks above, you will notice that eax is assigned with the pointer to CryptAcquireContextA.

It is good to comment this finding. To add comment click on call eax and press <colon> (colon):
Comment remaining arguments of CryptAcquireContextA accordingly to this function prototype to make it look like the following:

```
004D384 push 0F000000h ; dwFlags
004D389 push 1 ; dwProvType
004D38B push edi ; pszProvider
004D38C push edi ; pszContainer
004D38D lea ecx, [ebp+phProv]
004D390 push ecx ; phProv
004D391 call eax ; CryptAcquireContextA
004D393 test eax, eax
004D395 jz short loc_4AD3BE
```

Now you know that 0F0000000h and 1 are the constants passed to CryptAcquireContextA in arguments dwFlags and dwProvType. You can check in function reference that dwFlags takes the constant with the CRYPT_ prefix while dwProvType takes the constant with the PROV_ prefix. You can tell IDA to represent those values as a symbolic constant.

To use symbolic constant representation, right-click on 0F0000000h and choose “Use standard symbolic constant”.

```
0F000000h = CRYPT_VERIFYCONTEXT
```

In the next window IDA will display all known standard symbolic constants whose value equals to 0F0000000h. Choose constant with CRYPT_ prefix – CRYPT_VERIFYCONTEXT.

---

Repeat those steps for `dwProvType`, but this time choosing `PROV_RSA_FULL`. Now code should look like this:

```
0044D384 push CRYPT_VERIFYCONTEXT ; dwFlags
0044D389 push PROV_RSA_FULL ; dwProvType
0044D38B push edi ; pszProvider
0044D38C push edi ; pszContainer
0044D38D lea ecx, [ebp+hProv]
0044D390 push ecx ; pszProv
0044D391 call eax ; CryptAcquireContextA
0044D393 test eax, eax
0044D395 jz short loc_44D3BE
```

Now scroll up to the address 0x44D367. Here you can see a group of nodes making jump to the same location – `loc_44D3BE`. 
Further inspection shows that `loc_44D3BE` is a location of a function epilogue – probably jumped to if something earlier fails. Rename this location to `func_exit` in the same way as renaming memory location. Now all jumps should look much more clearly:

You can rename almost any name used in IDA (function names, arguments, variables, etc.) in the same way.
To further simplify function structure, you will now group graph nodes used to resolve crypto functions addresses. To do this, go to the graph node at the address 0x44D2F7 and select graph nodes by clicking on them while holding the <Ctrl> key.

Select all graph nodes starting from 0x44D2F7 up to 0x44D367.

Now right-click on selected nodes and choose Group nodes.

In the next window write short description of what grouped nodes are used to.
After clicking Ok all previously selected nodes should be replaced with the single node. To edit node group text or temporarily un-collapse group, use pair of new buttons on the node group header.

Now go to the location loc_44D2B1 (0x44D2B1).
Take a look at the call ebx instruction. If you select call ebx, you will notice that very similar calls are made in two other locations in the function:

```
0044D20A call ds:GetCurrentProcessId
0044D2E0 mov [ebp+var_8], eax
0044D2E3 lea eax, [ebp+var_8]
0044D2E6 push 4
0044D2E8 push eax
0044D2E9 call ebx
0044D3A0 push 20h
0044D3A9 push eax
0044D3B0 call ebx
```

In each case, two arguments are pushed onto the stack – first some address, and the second one seems to be the size of a buffer pointed by the first argument (it is good to comment this!).

Now if you select only the ebx register you will notice that its value is being assigned once at the beginning of the function:

```
0044D2A1 call ds:FindFirstFileA
0044D2A7 mov ebx, [ebp+arg_0]
0044D2AA mov esi, eax
```

This means that arg_0 is a function pointer and the function pointed by this argument is called three times in our function (you can rename arg_0 to func_ptr). Since this seems to be a significant element, it is good to mark all three graph nodes where such a call takes place.

To mark a graph node you will use the node colouring feature. Go back to loc_44D2B1 and click the icon of the colour palette in the left upper corner:
After clicking Ok node background should become cyan.
Repeat this step for the two remaining graph nodes where a call to `ebx` takes place.

Node colouring is a useful feature that can be used to mark graph nodes that we have already analysed or those that are for some reason significant.

One more thing you can do with IDA to improve code readability is to change how IDA presents numerical values. By default any numerical value is presented as hexadecimal. Sometimes you would like to view it as a decimal, binary or even custom defined constant. To change value format you can right-click on it and choose more suitable format.

Additionally in some rare situations it might be also helpful to change the name of some registers. For example, if in a given function some register is frequently used for only one purpose—e.g. storing some pointer or constant value—it might be good to change its name. This change would only apply to the current function.

An example of such register in `sub_44D262` is `edi`. The register is first zeroed (`xor edi, edi`) and then used in rest of the function only to compare other values to zero, or push zero onto the stack:
To rename a register, click on register and press <N> (rename):

Now the code should look like this:

Exercise
- Find where variable var_8 is used and rename it.
- Try to rename remaining locations: loc_44D2B1, loc_44D2DA, loc_44D36B, loc_44D3B4. What names would you suggest for them?
Advanced artefact analysis

Advanced static analysis

- Group three graph nodes checking if functions CryptAcquireContextA, CryptGenRandom and CryptReleaseContext were resolved correctly (0x44D36B, 0x44D374, 0x44D37C).
- Has the code readability of the function improved?
- Can you guess what function sub_44D262 might be used for?

2.10 Exercise
Take time to get familiar with IDA Pro and disassembled code. Make sure you know how to perform all presented operations and how to navigate through a code. Don’t hesitate to use functions not covered in this section. If something goes wrong you can always reload the sample.

2.11 Summary
In this exercise you have learned how to use IDA to analyse disassembled code. First you have learnt how to customize the IDA workspace and then how to navigate through code. Basic function structure and function types were also introduced. Finally you saw how to enhance disassembled code by adding comments, changing names and using colouring functions to improve code readability.
3. Recognizing important functions

A problem with analysing complex malware samples is that disassembled code is often quite overwhelming and consists of many functions. Usually not all of those functions are important. Some of them perform only trivial tasks or we just want to focus on one particular malware functionality. In this exercise you will learn how to find which functions might be important and which ones you should try to analyse first.

Always begin by thinking what the goal of your analysis is. Do you want to learn about general malware functionality or just want to obtain information about one particular function? Depending on the answer, you should narrow your search.

When starting the analysis of a new binary, one approach is to analyse the main routine and to try following its execution flow. As long as such analysis might give us valuable information about the sample itself this is worth trying, but it can also be quite a tedious task – especially when functions you are looking for are not directly called from the main routine.

Fortunately there are three basic techniques which can help us to find interesting functions:

a) Using call graphs
b) Following cross references to strings and imported functions
c) Learning functions addresses during dynamic analysis

The first two techniques will be presented in the following exercises. In the last technique you will need to apply techniques learnt during the second part of the training – Advanced dynamic analysis – to pinpoint where in the code the interesting malware function is located (for an example, check the address of the code responsible for communication with the C&C server) and then start analysis of this code in IDA. This technique is not covered in the exercise.

In this exercise, you will use sample of the Slave trojan\(^\text{12}\) which is a banking trojan first detected by S21sec company\(^\text{13}\). Before continuing, please load slave.exe sample in IDA and wait until the initial auto analysis completes. Because you will be now analysing a live malware sample, remember to take all necessary precautions.

3.1 Using call graphs

Starting the analysis of a new binary, some of the first questions that comes to mind are what is the execution flow of the code? What local functions are called by what other functions? Are there any API calls? What data variables are referenced in the code? To answer some of those questions, IDA provides us with its graphing capability.

Call graphs are graphical representations of all recognized function calls in the code. They use an external application wingraph32 to present function calls in the form of a directed graph in which nodes represent functions or data locations and lines are calls or references to data.

---

\(^\text{12}\) Sample 400fbcaaac9b50becbe91ea891c25d71 (MD5) https://malwr.com/analysis/OTRiMDk1ODFkOGVjNDhkMzljYzdiZTUzY2EwM2M/ (last accessed 11.09.2015)

To access the call graph functionality use menu View->Graphs or use the Graphs toolbar.

There are four basic call graph types:

- Function calls
- Xrefs to
- Xrefs from
- User xrefs chart...

Note that creating Xrefs to or Xrefs from is possible only if, in disassembly view, the currently selected item is some function name or a named data location (dword_XXXXXX).

Start by clicking on wWinMain function in the slave.exe sample and then choose to create Xrefs from call graph. Note that you need to click on actual function (as on the picture below) and not on function name in function prototype.

```plaintext
:    stdcall wWinMain(x, x, x, x)
    wWinMain proc near
    push    esi
    push    edi
    call    sub_402800
    mov     esi, ds:CreateMutexW
```

Now you should see WinGraph32 window with newly created call graph for wWinMain function. This Xrefs from graph presents all functions called from wWinMain routine (local functions, library functions as well API functions).
Depending on the code complexity and size of your screen such graph might be more or less readable. For more complex malware or malware using many linked libraries such graph might be barely readable.

To navigate the graph, use left mouse button. To zoom in or zoom out, use the toolbar buttons as shown on the screen above.

Now zoom in (or zoom to 100%) to notice the different colours of the graph nodes. Black nodes represent local functions while pink nodes represent API calls. There might be also cyan nodes and white nodes representing functions recognized by IDA as library functions and named data locations, respectively.

So far, you have been analysing what functions from the \texttt{WinMain} were called. What if you want to check what functions call \texttt{WinMain}? You can use the Xrefs to call graph. Click on \texttt{WinMain} and choose Xrefs to graph.
Without much of a surprise, we see that `wWinMain` was called from `___tmainCRTStartup` routine. To get a little more complex example, create Xrefs to graph for `sub_404330`.
Xrefs to graphs might be also used to check what functions are referencing particular memory location. As an example go to the \texttt{wWinMain} function, click on \texttt{dword\_438120} and choose to create the Xrefs to graph.

\begin{verbatim}
; __stdcall wWinMain(x, x, x, x)
 wWinMain@16 proc near
push esi
push edi
call sub\_402860
mov esi, ds:CreateMutexW
push 0 ; lpName
push 0 ; bInitialOwner
push 0 ; lpMutexAttributes
mov dword\_438120, 0
call esi ; CreateMutexW
mov edi, ds:time
push 0 ; time_t *
mov hWnd, eax
\end{verbatim}

You should see all functions referencing this memory location. This may prove to be useful if you know that at memory location is stored some important variable (e.g. flag telling whether virtual machine was detected) and you want to see which functions are checking that variable.

The third type of graphs are user defined graphs. In contrast to Xrefs to and Xrefs from graphs, when creating a user defined graph you can specify additional parameters for how this graph should look. To create this graph for \texttt{wWinMain} select \texttt{wWinMain} and choose User xrefs chart....
In the new window, you can specify additional graph parameters. You can hover the cursor over any parameter to get a hint what this parameter changes. The most frequently used group of parameters are Starting direction and Recursion depth. Using Recursion depth you can limit the number of graph nodes followed from the current location. This might be useful when dealing with more complex code.

As an example, create a graph for wWinMain presenting only references from this function and limiting the graph to recursion depth 2.
Is newly created graph clearer and easier to follow?

3.2 Exercise

Take a few minutes to experiment with the other options of user defined graphs. Create a few graphs for functions other than wWinMain.

The last graph type – Function calls, presents a graph of function calls for all recognized functions. This usually would be quite a complex graph, but you can use it to detect if there are any functions in the code not called from the main routine. This might be caused by various circumstances, such as external functions (exported in Export Table), functions that are called indirectly and IDA failed to recognize them or functions being injected to some other process.
Now that you know how to create various call graphs and what they are used for, how can you recognize important function calls?

A good starting point is to create an Xrefs from graph for the wWinMain function (or any other function recognized by IDA as a main function). Depending on the code complexity, you might decide to limit recursion depth. Zoom in the graph and start looking for two types of functions:

a) Functions calling groups of similar APIs. Based on what API calls are made, you can often deduce the purpose of such a function, for example a function calling registry-related APIs might be an installation routine, while a function calling network-related APIs might be used to communicate with a C&C server.

b) Functions that call many local functions. This might indicate that some important program logic takes place inside such a function. It may not always be true, but it is usually worth the time to inspect such functions. You may also note which functions are called by many other (often unrelated) functions. Such functions usually complete some trivial task and analysing them first might help you understand rest of the code.

As an example you will now analyse call graph of wWinMain function\(^\text{14}\).

First, notice the top group of three functions (1): sub_406410, sub_406120, sub_401890. At this point you can already suspect that those are important functions because they are called directly from the wWinMain and they are calling a lot of APIs. Unfortunately due to the structure of the graph it is hard to tell which API is called by which function. To deal with this problem, create a call graph of wWinMain with recursion depth equal to 2.

\(^{14}\) This graph might be slightly different, but if using the same IDA version its general structure should be very similar.
Then take a look at **sub_401B90**. We can see that this function is iterating through the process list (calls to \textit{Process32FirstW}, \textit{Process32NextW}, etc.). This might mean that this function is looking for a specific process to inject some code into it or it is using some anti-analysis techniques (e.g. trying to detect AV processes).

Next, look at **sub_406410**. It calls APIs such as \textit{RegSetValueExW}, \textit{Create DirectoryW}, \textit{CreateFileW}, \textit{MoveFileExW}. It likely indicates that this is an installation procedure. You should inspect it if you want to know how the malware installs itself in the system.

Then take a look at **sub_406120**. It enumerates the registry (\textit{RegEnumValueW}) and checks some module path (\textit{GetModuleFileNameW}). It is hard to tell what its purpose is, but it is likely still worth inspecting.

Now go back to the general graph (\textit{wWinMain}) and take a look at function **sub_402050** (2). Among the other APIs it is also calling \textit{CreateRemoteThread} and \textit{WriteProcessMemory}. This tells us that this function is most likely injecting some code to other processes (you can also notice that \textit{sub_402050} was first called from already checked \textit{sub_401B90} which was an iterating process list).

Next, take a look at function **sub_405760** (3) which is calling many other functions. This might suggest that some important program logic is taking place inside this function.
If you look closer at the rest of the graph you notice several other potentially interesting functions like `sub_4027E0` performing some file system operations (DeleteFileW, WriteFile, SetFileAttributesW, CreateFileW) or `sub_406CA0` doing some threads operations (ResumeThread, SuspendThread, OpenThread, ...).

The next thing you might consider doing would be to create separate call graphs for functions such as the previously noticed `sub_405760`. However at this point it seems that the most important functions that should be analysed first are:

- `wWinMain` – main routine
- `sub_401B90` – iterating process list
- `sub_406410` – installation routine
- `sub_406120` – possible registry enumeration
- `sub_402050` – process injection routine
- `sub_405760` – calling many other subroutines

One more thing you might do would be to create a call graph for all functions (Function calls graph) and as previously described, check if there are any functions not called directly from `wWinMain`. If there are any, you might repeat the steps described above for each function not called directly from `wWinMain`. 
3.3 Using cross references

One of the very useful features of IDA are cross references (short: *xrefs*). During initial autoanalysis, for each named object – whether it is a function, string, variable or memory location – IDA tracks all locations where this object is referenced. Where an object reference is any assembly instruction referencing to the object, reading its value, writing to the object, pushing object’s address onto the stack or calling object (if object is a function). Using cross references you can learn at what addresses a given function was called, where a string was used or a certain variable was written to. The call graphs used in the previous exercise were constructed by IDA based on cross references.

To use cross references, go to the place where a given object is defined (not referenced), click on the object name and press <X> (or select View->Open subviews->Cross references).

As an example, go to *wWinMain* function.

```assembly
00405660                      ;declspec dllcall wWinMain(x, x, x, x)
00405660                      ; wWinMain@16 proc near
00405660     push   esi
00405661     push   edi
00405662     call   sub_402960
00405667     mov   esi, ds:CreateMutexW
0040566d     push   0       ; lpName
00405670     push   0       ; bInitialOwner
00405673     push   0       ; lpMutexAttributes
00405676     mov   dword_438120, 0
00405679     call   esi; CreateMutexW
0040567f     mov   edi, ds:time
00405685     push   0       ; time_t *
```

To check where the global variable *dword_438120* is used double click it to go to the memory location where this data variable is defined.

```
data:0040381c ; HANDLE hHandle
.data:0040381c |hHandle       dd ?          ; data XREF: sub_4025a0+76+Tr
.data:0040381c  ; sub 402540+270+Tr ...
.data:0040381c  ; sub 402540+82+Tr ...
.data:0040381c  ; sub 402540+81+Tr ...
.data:0040381c  ; sub 402300+14+Tr ...
data:0040381c  ; sub 402300+14+Tr ...
```

Notice that on the right, IDA already tells you about two cross references to this variable. However to get a better view and list of all cross references it is best to select variable and press <Ctrl+X> to open *Cross references* dialog.
By default the Cross references list consists of four columns. The first column (Direction) tells you whether the cross reference to the object occurred before or after the object (in regard to the memory address). The second column (Type) tells the cross reference type (r – read operation, w – write operation, o – operation on the object’s address e.g. pushing it onto the stack). The third column (Address) gives the exact address at which the cross reference occurred. Notice how the addresses are presented: <func_name>+<offset>, where the first part is a function name in which the cross reference occurs and the second part is an offset to the location within this function. Finally in the last column (Text) there is an assembly operation referencing the object.

You can also immediately jump to any cross reference by double clicking it. For example, jump to the cross reference at the address sub_402540+C5 (if you then want to go back, simply press <Esc>).

At this address, you see that the data address is moved onto the stack (assigned to local variable var_4).

Now you will use cross references to find important functions. You can do this by first following cross references to imported functions and secondly by following cross references to strings found by IDA. By following cross references to API functions you are basically doing the same as when analysing call graphs in previous exercise. However since call graphs are not always easy to read, this method also makes sure that you haven’t missed anything. Moreover if you are only interested in specific APIs, it is easier to find them by directly following cross references than to look for them on the call graph.

First, switch to imports view. If the window is not already, open it by choosing View -> Open subviews -> Imports. To make searching easier, sort imported functions by name by clicking on the Name column.
Let's say you want to find which function is injecting code to other processes. To do this, first find the `WriteProcessMemory` function on the imports list and double click it.

Next click on the function name and open the Xrefs dialog.
There is only one function calling WriteProcessMemory twice – sub_402050. Note that this is the same function you already found during call graphs analysis.

When looking at the imports list one thing that stands out is a complete lack of network related functions. It is rather uncommon for a malware to not communicate with any servers. This suggests such functions might be loaded dynamically at runtime. Let’s check it by following cross references to GetProcAddress function.

As suspected, there are quite a lot calls to GetProcAddress. In total there are 10 different functions calling GetProcAddress:

- sub_401B30 – 1 call
- sub_401B50 – 1 call
Now go to any cross reference in `sub_402860` (or just go to this function), and take a look at calls to `GetProcAddress`:

```
00402939  loc h02939:  ; "InternetOpenA"
0040293D  push offset alInternetopena
0040293F  push edi    ; hModule
00402941  call esi    ; GetProcAddress
00402947  push offset alInternetconnect ; "InternetConnectA"
0040294B  push edi    ; hModule
0040294F  mov dword 438104, eax
00402951  call esi    ; GetProcAddress
00402955  push offset alHttpopenrequest ; "HttpOpenRequestA"
00402959  push edi    ; hModule
0040295B  mov dword 43810C, eax
0040295D  call esi    ; GetProcAddress
00402961  mov dword 438114, eax
00402963  push offset alHttpopenrequest ; "HttpSendRequestA"
00402967  push edi    ; hModule
00402969  mov dword 438118, eax
0040296B  call esi    ; GetProcAddress
0040296D  push offset alInternetreadfile ; "InternetReadFile"
00402971  push edi    ; hModule
00402973  mov dword 438110, eax
00402975  call esi    ; GetProcAddress
00402977  push offset alInternetclose ; "InternetCloseHandle"
0040297B  push edi    ; hModule
0040297D  mov dword 438118, eax
00402981  call esi    ; GetProcAddress
00402983  pop edi
00402985  pop esi
00402987  mov dword 438110, eax
```

Six network-related functions are dynamically loaded at runtime and their addresses saved in memory:

- `InternetOpenA` -> `dword_438104`
- `InternetConnectA` -> `dword_43810C`
- `HttpOpenRequestA` -> `dword_438114`
- `HttpSendRequestA` -> `dword_438108`
- `InternetReadFile` -> `dword_438118`
- `InternetCloseHandle` -> `dword_438110`

Now follow cross references to `dword_438108` to check where `HttpSendRequestA` function is called:

```
.data:00438104  dword_438104 dd ?
.data:00438108  dword_438108 dd ?
.data:00438110  dword_438110 dd ?
.data:00438114  dword_438114 dd ?
.data:00438118  dword_438118 dd ?
.data:00438120  dword_438120 dd ?
.data:00438124  dword_438124 dd ?
.data:00438128  dword_438128 dd ?
.data:00438130  dword_438130 dd ?
.data:00438134  dword_438134 dd ?
.data:00438138  dword_438138 dd ?
.data:00438140  dword_438140 dd ?
.data:00438144  dword_438144 dd ?
.data:00438148  dword_438148 dd ?
.data:00438150  dword_438150 dd ?
```
You see that there is one call to `HttpSendRequestA` in `sub_402300`. Follow this cross reference to land in a function which is evidently used to communicate with some C&C server. This function was missed by us before because in this function the only meaningful API calls are to network functions loaded dynamically at runtime.

At this point (depending on what you want to find) you could continue analysis of cross references to other functions from imports list.

A second way of finding important functions using cross references is to follow cross references to strings found by IDA. You follow cross references to strings in a similar manner to following cross references to imported functions. First you open the strings list, then you look for any strings that stand out and check where those strings are referenced in the code.

First, switch to strings view. If strings view is not open, choose View -> Open subviews -> Strings.
In the strings window, you see a few interesting strings. There is some domain name: `www.bizzanalytics.com`. Double click on this string and follow cross references to it:

```
.rdata:00411140  unicode 0, <__NDTLL_CORE__>, 0
.rdata:0041115E  align 10h
.rdata:004116B0 bwww_bizzanalyt db 'www.bizzanalytics.com', 0 ; DATA XREF: sub_402300+45To
.rdata:004116C0 db 0
.rdata:00411760 db 0
```

You see there are two cross references, first one leads to `sub_402300` – function you have already found to communicate with a C&C server and the second one is a string offset written in memory. At this point it is hard to tell what it is used for.

```
.rdata:00410320  dd Offset Name       ; "NDTLL_CORE__"
.rdata:00410324  dd offset ainfo_php?key="$/info.php?key=QDEwnWJUozTqt1ARgajoKm88B"
.rdata:00410328  dd offset alwww_bizzanalyt ; "www.bizzanalytics.com"
.rdata:00410330  dd offset astartup ; "\);
.rdata:00410334  dd offset awininet.dll ; "wininet.dll"
.rdata:00410338  dd offset asc_http20 ; "\n"n"
.rdata:0041033C  dd offset aPost ; "POST"
.rdata:0041033E  dd offset aSacceptEncoding ; "\nAccept-Encoding:"
.rdata:00410348  dd offset aStransferEncod ; "Transfer-Encoding: chunked\\n"
```
Now go back to the strings window and notice the strings named \textit{PR\_Write}, \textit{PR\_Read}, and \textit{PR\_Close}, which are names of functions from the NSPR library used for network communication\textsuperscript{15}. This library is used for example by Mozilla Firefox web browser. This is typical for modern malware performing so-called MitB (\textit{Main-in-the-browser}) attacks by hooking network-related functions in a web browser and injecting malicious code into the content of some websites (usually financial) or stealing user credentials\textsuperscript{16} \textsuperscript{17} \textsuperscript{18}.

Let's examine where those strings are referenced.

\begin{verbatim}
.rdata:0011558 ; char aPr_write[]
.rdata:0011558 aPr_write db 'PR\_Write',0 ; DATA XREF: sub A05390+17Efo
.rdata:0011558
.rdata:0011558 ; char aPr_read[]
.rdata:0011558 aPr_read db 'PR\_Read',0 ; DATA XREF: sub A05390+1A7fo
.rdata:0011558
.rdata:0011558 ; char aPr_close[]
.rdata:0011558 aPr_close db 'PR\_Close',0 ; DATA XREF: sub A05390+1D0fo
.rdata:0011558
.rdata:0011575 ; align 4
\end{verbatim}

\begin{footnotesize}
\textsuperscript{18} Firefox FormGrabber https://redkiing.wordpress.com/2012/04/30/firefox-formgrabber-iii-code-injection/ (last accessed 11.09.2015)
\end{footnotesize}
All three of these strings are referenced in two different functions: sub_405390 and sub_405760. If you jump to either of those two functions and examine it, you will see references to strings like “HttpQueryInfoA”, “InternetReadFile”, “InternetReadFileExA”, “InternetQueryDataAvailable” and “InternetCloseHandle” which are network functions used in Internet Explorer web browser. This confirms our suspicion that malware is likely performing MitB attack.

It should be noted that this is not a complete analysis of cross references to strings or to imported functions. However at this point you should already have idea how to use cross references to find important or interesting functions.

Using cross references to strings and imported functions, you have confirmed a few findings from the previous exercise and found three more suspicious functions:

- sub_402300 – function likely used for communication with C&C server
- sub_405390, sub_405760 – functions probably used to set up hooks in web browser
3.4 Exercise
Save the results of your current work and open a new sample dexter.exe which is a sample of Dexter malware targeting POS systems\textsuperscript{19}. Using techniques presented in this exercise try to pinpoint important functions in disassembled code.

- Find network related functions.
- Find the installation routine.
- Find the function performing RAM scraping (reading memory of other processes).
- Find the process injection routine.
- Are there any other potentially interesting or suspicious functions?

This exercise might be conducted in a small groups. After the assigned time passes, each group should present their findings. Are findings of each group similar?

3.5 Summary
In this exercise you have learnt how to recognize important functions in disassembled code. To do this you first used call graphs to track execution flow and then you followed cross references to strings and imported functions. This way, you were able to find groups of suspicious functions such as an installation routine, process injection routine or a function likely used to communication with a C&C server. All functions that were found are also good starting points for further analysis.

However you should remember that the approach presented in this exercise might not always work or could be quite difficult to apply. The first problem are samples that obfuscate their execution flow or that load all API functions dynamically. You will see examples of such code in later exercises. The second problem might be samples that use many statically linked libraries not recognized by IDA. In this case, you might have difficulties recognizing what parts of the code are part of main malware code and what parts are just some library functions.

Finally, if you are looking for important functions, it is a good practice to rename each suspicious function you find. This way it will be easier to follow which functions you have already visited and which ones you haven’t. If you rename any functions or add comments to the code, remember to save results of your work.

4. Functions analysis

In the previous exercise you found a group of suspicious functions. The next step is to analyse those functions in order to better understand their functionality and what they are used for. In this exercise, you will learn the basic principles of function analysis: how to start analysis, what to look for and how to understand a function’s role.

In general when analysing a function you want to answer three questions:

1. What are the function’s arguments?
2. Is the function returning anything?
3. What is the role of the function? To perform some operation on arguments? To perform some memory operations? Execute other tasks?

Full function analysis strongly depends on function complexity. There are simple functions, performing only a single or a few tasks, which are usually fairly easy to analyse. There are also very complex functions, performing a lot of operations and using many variables or complex data structures, analysis of which is usually quite demanding and takes a long time. Moreover if a function is calling other local functions you would often need to analyse them first in order to understand their role in the context of our function. Fortunately a full function analysis is usually not necessary. In many cases, a quick assessment of a function without fully understanding details of its operation should be enough.

When starting an analysis of a function it might be helpful to answer the following questions (not necessarily in this order):

- Are there any API calls in the function? If yes, what are they used for?
- Are there any calls to other local functions? What are they doing?
- Are there any xrefs to the analysed function? From which other functions is the function called? Are there any arguments pushed onto the stack when the function is called? Is their type known (e.g. some handle, buffer address, decimal value, etc.)?
- What is the function calling convention?
- How many arguments is the function using? How are they used in the code?
- Are there any local (stack) variables used? How are they used in the code?
- Are there any global variables used in the function? How are they used in the code?
- Is the function ending (no endless loop)? Is it returning any value?
- Are there any loops or switch statements in the function? Is there only one execution path?
- Are there any strings referenced in the function?

You will now proceed to analyse chosen functions from the Slave Trojan. When analysing a function remember to always document your findings as presented in the *Enhancing assembly code* exercise.

4.1 Analysis of network function

You will start the analysis with the subroutine that you suspect communicates with the C&C server.

First go to `sub_402300` (or `0x402300` address). At first glance this function doesn’t seem to be very complicated. Only a few blocks of code and one loop.
For convenience (if you haven’t done it already) rename sub_402300 to f_CnC_func. If you later decide this is inappropriate you will rename it something else.

```
00402300 f_CnC_func proc near
00402300 var_1018- dword ptr -1018h
00402300 var_18= dword ptr -18h
00402300 var_14= dword ptr -14h
00402300 var_10= dword ptr -10h
00402300 var_C= dword ptr -0Ch
00402300 var_8= dword ptr -8
00402300 var_y= dword ptr -4
00402300 push ebp
00402301 mov ebp, esp
00402303 mov eax, 1018h
00402308 call __alloca_probe
```

To check what functions are called within f_CnC_func you need to first deal with calls to global variables:

```
00402319 mov [ebp+var_14], 0
00402320 push 0
00402322 mov [ebp+var_C], edi
00402325 call dword_408194
00402328 mov ebx, eax
0040232d mov [ebp+var_10], ebx
00402330 test ebx, ebx
```
Fortunately you already know where those variables are set (please refer to the previous exercise). Using cross references go to the place where value of `dword_438104` is set (or just jump (G) to 0x402939):

```
00402939  loc 402939:   ; "InternetOpenA"
00402939  push offset aInternetopena
0040293E  push edi          ; hModule
00402943  call esi ; GetProcAddress
00402947  push offset aInternetconnect ; "InternetConnectA"
0040294E  push edi          ; hModule
00402953  call esi ; GetProcAddress
00402957  push offset aHttpopenrequest ; "HttpOpenRequestA"
0040295E  push edi          ; hModule
00402963  call esi ; GetProcAddress
00402967  push offset aHttpsendrequest ; "HttpSendRequestA"
0040296E  push edi          ; hModule
00402973  call esi ; GetProcAddress
00402977  push offset aInternetreadfile ; "InternetReadFile"
0040297D  push edi          ; hModule
00402983  call esi ; GetProcAddress
00402989  push edi          ; hModule
0040298E  call esi ; GetProcAddress
00402994  push edi          ; hModule
0040299A  pop edi
0040299C  pop esi
0040299E  pop dword_438110, eax
```

Rename all global variables used to store addresses of network related functions (make sure you don’t change the order or make a typo):
Advanced artefact analysis
Advanced static analysis

Now go back to f_CnC_func and reanalyse code (Options->General->Analysis->Reanalyse program). IDA should add additional comments:

Now you can check what functions are called within f_CnC_func. A convenient way to do this is to use Function calls sub view which will also present where f_CnC_func is called from.

While staying in f_CnC_func, choose View->Open subviews->Function calls.

20 If at some point you notice that your disassembly is lacking some comments (except the ones added manually) in comparison to the screenshots in this document you can try repeating this step. Also make sure that you properly renamed global variables containing pointers to API functions.
In the upper part of the window, there is a list of locations where \textit{f\_CnC\_func} was called. In the lower part of the window there is a list of all calls made within \textit{f\_CnC\_func}. You can double click on any of those calls to be moved to the calling instruction.

Short analysis of this list tells us three important things. Firstly, \textit{there are no other API calls except calls to network related functions} (and a few memory allocation functions from C standard library). Secondly, \textit{there are no calls to other local functions}. Thirdly, \textit{f\_CnC\_func} is called only once (in \textit{sub\_402540} function).

Knowing this plus the fact that \textit{f\_CnC\_func} is rather simple and short function you can assume that that \textit{f\_CnC\_func} is most likely used only to communicate with C&C server and is not doing any analysis of received data.

Consequently what should you be now interested is:

- What are \textit{f\_CnC\_func} arguments?
- Is \textit{f\_CnC\_func} returning anything?
- Is there any data sent to C&C server? How?
- Is there any data received from C&C server? What is happening to this data?

Let’s start by analysing if there are any function arguments:
IDA recognized this function as a function with bp-based stack frame. There are a few stack variables used in the function but it seems there aren’t any arguments. Are there?

Just to be sure go to the place where \texttt{f\_CnC\_func} is called from following the address 0x40256C that you got from the function calls window.

You are now at the beginning of the \texttt{sub\_402540}. It seems there are no push instructions before a call to \texttt{f\_CnC\_func}. However notice that \texttt{ecx} register is assigned with the address of \texttt{var\_8} variable, which is later also initialized to zero.

Notice also how \texttt{eax} register is tested after a call to \texttt{f\_CnC\_func} and if it equals to zero \texttt{sub\_402540} returns. This suggests that \texttt{f\_CnC\_func} is returning some value in \texttt{eax} register and it should be nonzero on success.

Now go back to \texttt{f\_CnC\_func} to check if \texttt{ecx} register is used for anything.
Yes, you were right. Value of ecx is assigned to edi register. This means that \textit{f\_CnC\_func} is either using the \textit{fastcall} calling convention or you might be dealing with object-oriented programming and ecx is used to pass this pointer to a member function (\textit{thiscall} calling convention). If you analyse other functions in the code you will notice that arguments to some other functions are passed in ecx and edx registers. This means this is likely \textit{fastcall} function and ecx is used to pass pointer to variable or some data structure.

Notice that later the edi register is assigned to \textit{var\_C}. Rename \textit{var\_C} to \textit{this}.

Now go to the last block of \textit{f\_CnC\_func} (loc\_40246A):

\begin{verbatim}
0040245A loc_40245A: ; hInternet
0040245A push ebx
0040245B call InternetCloseHandle
00402471 mov eax, [ebp+var_14]
00402474 pop edi
00402475 pop ebx
00402476 mov esp, ebp
00402478 pop ebp
00402479 ret
00402479 sub_4012300 endp
00402479
\end{verbatim}

Notice that the \textit{eax} register is assigned with the value of the \textit{var\_14} variable. This means that the \textit{var\_14} variable is used to store the return value. Rename \textit{var\_14} to \textit{retval}. For convenience it is also good to rename label \textit{loc\_40246A} to something like \textit{func\_exit}:
At this point you know that the f_CnC_func is taking a single argument (passed in ecx) and is returning some value in the eax register. Now you will analyse how communication with the C&C server is taking place and what happens to the received data.

Go to beginning of the function.

```assembly
0040230F push 0 ; dwFlags
00402310 push 0 ; lpszProxyBypass
00402315 push 0 ; lpszProxy
00402317 mov edi, ecx
00402319 mov [ebp+retval], 0
00402320 push 0 ; lpszAccessType
00402322 mov [ebp+this], edi
00402325 call InternetOpenA
00402328 mov ebx, eax
0040232a mov [ebp+var_10], ebx
00402330 test ebx, ebx
00402332 jz func_exit
```

Notice how the initial return value (retval) is set to zero. Then there is a call to InternetOpenA with all parameters set to zero. According to MSDN documentation\(^{21}\) this function initializes use of the WinInet functions and returns the hInternet handle. You see that this handle is assigned to var_10 and if it is zero then there is a jump to func_exit.

For clarity rename var_10 to hInternet.

```assembly
00402319 mov [ebp+retval], 0
00402320 push 0 ; lpszAgent
00402322 mov [ebp+this], edi
00402325 call InternetOpenA
00402328 mov ebx, eax
0040232a mov [ebp+hInternet], ebx
00402330 test ebx, ebx
00402332 jz func_exit
```

If InternetOpenA succeeds in the next step malware calls InternetConnectA to initiate connection with the destination server.

What’s important here is that connection is made to hardcoded hostname – www.bizzanalytics.com on standard HTTP port – 80/tcp (50h). Result of a call to InternetConnectA (connection handle) is then saved to var_18.

For clarity, rename variables and add symbolic constants. For 0x40233D, right click and select symbolic constant -> use standard symbolic constant from the list select "INTERNET_SERVICE_HTTP”. For 0x402343 switch to decimal by clicking on it and use shortcut key Shift+H. Also rename var_18 to hConnect.

In the next step, the malware is opening an HTTP request using HttpOpenRequestA.

Here you see that the HTTP request (GET) is made to the similarly hardcoded info.php with some hardcoded key as a GET variable. To get full key value hover mouse cursor over szObjectName or double click it.
You can also see that there are some flags (dwFlags) passed to HttpOpenRequestA. Unfortunately, IDA fails if a variable is a sum of more than one flag (symbolic constants).

Finally, a new request handle is temporarily saved to the ebx register.

Next the malware is sending an HTTP request.

Nothing special is happening here. There are no extra headers and there is no POST data (lpOptional). Notice that request handle (hRequest) is saved to global variable dword_438124. Rename it to CnC_hRequest and check the xrefs to it.
Notice that there are some references to this variable outside of the \texttt{f\_CnC\_func}. Renaming this variable might help us in later analysis.

Next if sending HTTP requests doesn’t fail (\texttt{eax} will be nonzero on fail), the malware starts reading data received from the server (\texttt{InternetReadFile}). You will now analyse what happens to the received data, where it is being saved and if it is being processed anyhow (for example xor’ed).

Now take a look at the next three code blocks (0x4023A6, 0x4023D3, 0x4023DA):

In the first block there is a single call to \texttt{InternetReadFile}.

Then there is a loop over block [2] and [3] with an additional call to \texttt{InternetReadFile} in block [3]:

This is a popular scheme of downloading any data from the Internet. Malware first tries to download first part of the server response (in block [1]) and if any data is received it continues calling \texttt{InternetReadFile} (in block [3]) until it fails or number of received bytes is zero – meaning that there is no more data to be received.
Now let’s analyse block [1] in more detail.

At the beginning of this block there is a call to `malloc` allocating a memory block with size of 1 byte.

```assembly
004023A6 xor esi, esi
004023A8 push 1 ; size_t
004023A0 mov [ebp+var_8], esi
004023A4 call ds:malloc
004023B3 add esp, 4
004023B6 mov [edi], eax
```

Notice the address of the newly allocated memory block is saved to the variable pointed by the edi register. But what is the edi register? Highlight it and search where in the code its value was last set:

```assembly
00402317 mov edi, ecx
00402319 mov [ebp+ret], 0
00402320 push 0 : lpszAgent
00402322 mov [ebp+this], edi
00402325 call InternetOpenA
```

So it looks like `edi` still contains a variable pointer passed to this function as an argument and an address of allocated memory is saved to this variable.

Going back to block [1], notice that some variable (`var_8`) is initialized to zero. Highlight `var_8` and check where else in the code this variable is used:

```assembly
004023E0 push [ebp+var_8] ; size_t
004023E2 mov esi, [esi]
004023EE push esi ; void *
004023EF push edi ; void *
004023F0 call memcpy
004023F5 push esi ; void *
004023F6 call ds:Free
```

You see that `var_8` is used a few times in block [3]. First in conjunction with `memcpy` function to specify a number of bytes to be copied and later a number of received bytes is added to `var_8`. This means that `var_8` is used to store number of received bytes. Knowing all of this you can comment appropriately beginning of the block [1]:

```assembly
004023A6 xor esi, esi ; esi <- 0
004023A8 push 1 ; size_t
004023AA mov [ebp+recu_len], esi ; recu_len <- 0
004023A0 call ds:malloc ; allocating 1 byte of memory
004023B3 add esp, 4
004023B6 mov [edi], eax ; *this <- eax (memptr)
```

In the second half of block [1] there is a call to `InternetReadFile`:
Here you see that received data is saved to a Buffer variable which is a memory buffer declared on the stack with the size of 4096 bytes (1000h). Moreover the number of received bytes will be saved to the dwNumberOfBytesRead variable.

By taking a look at the stack you can also notice that you have already identified all local variables.

Now go to block [2] – the first block of the receive loop.

As you see in block [2] there is a check if the number of received bytes in the last call to InternetReadFile is nonzero. If it is zero you jump out of the loop to loc_402442.

Now let’s proceed with the analysis to block [3]. To make analysis easier, there are already some comments added in the pictures below.
The first thing that happens in block [3] is allocation of a new memory block of size equal to length of data received so far (recv_len) plus the length of the newly received data plus one. Then the data from previously allocated memory block (memptr_old) is copied to the beginning of new memory block. After this, the old memory block is freed.

```
004029f6 call ds:free ; free memptr_old  
004029fc mov eax, [ebp+this]  
00402a00 push [ebp+dwNumberOfBytesRead] ; n (num of bytes to copy)  
00402a05 mov esi, [ebp+recv_len] ; esil <= recv_len  
00402a0b mov [eax], edi ; *this <= memptr_new (updating memptr)  
00402a17 lea eax, [ebp+Buffer]  
00402a1b push eax  ; src (Buffer)  
00402a21 lea eax, [edi+esi]  
00402a25 push eax ; dst (memptr_new)  
00402a29 call memcpy ; Copy newly received data from Buffer  
00402a2d mov [edi+esi], eax ; to the end of memptr_new
```

In the next part, the newly received data from the buffer on the stack is copied to the end of the newly allocated memory block (just after previously copied data).

```
00402a2f call memcpy  
00402a33 mov esi, [ebp+dwNumberOfBytesRead] ; esi <= recv_len + dwNumberOfBytesRead  
00402a37 lea eax, [ebp+dwNumberOfBytesRead]  
00402a3b add esp, 28h  
00402a43 mov [ebp+recv_len], esi ; recv_len <= recv_len+dwNumberOfBytesRead  
00402a47 mov [ebp+retval], 1 ; Received some data: set retval to 1  
00402a4b push eax ; lpdwNumberOfBytesRead  
00402a4f push 4006 ; dwNumberOfBytesToRead  
00402a53 lea eax, [ebp+Buffer]  
00402a57 push eax ; lBuffer  
00402a5b push ebx ; NFile  
00402a63 call InternetReadFile  
00402a67 test eax, eax  
00402a6b jnz loc_402442
```

Finally variable recv_len is updated with new length of received data and InternetReadFile is called again. Notice that retval variable is set to 1.

As already mentioned, the loop will execute until InternetReadFile fails or the number of received bytes is zero:

```
00402a6d mov eax, [ebp+dwNumberOfBytesRead]  
00402a73 test eax, eax  
00402a77 jz short loc_402442
```

Next, the block after the loop is loc_402442 in which last byte of allocated memory is zeroed.

```
00402a79 loc_402442: ; edi <- this  
00402a7d mov edi, [ebp+this]  
00402a81 loc_402445:  
00402a85 mov eax, [edi]  
00402a89 mov byte ptr [esi]+eax, 0 ; Zeroing last allocated byte.  
00402a8d mov [esi]+eax, 0  ; eax <= memptr  
00402a91 mov byte ptr [esi+recv_len], edi  ; esi - recv_len
```
After this the only thing that happens is the closing all opened handles:

```
0040246A  func_exit:    ; hInternet
0040246A  push    ebx
00402468  call     InternetCloseHandle
00402471  mov      eax, [ebp+retval]
00402474  pop      edi
00402475  pop      ebx
00402476  mov      esp, ebp
00402478  pop      ebp
00402479  retn
00402479  sub_402479 endp
```

Finally in `func_exit` the `eax` register is assigned with the value of `retval` variable and function returns.

At this point, detailed function analysis is done. However, remember that detailed function analysis is not always necessary. Sometimes it is enough just to do quick assessment what the function is doing. It is important to set a goal before beginning analysis.

What you have learnt about `f_CnC_func`:

- Returns 1 if any data was received
- Connection is made to the hardcoded URL
- No POST data is sent in the request to the C&C server
- There is no processing of received data. Function is used solely to download some data from the server.
- Received data is saved to a newly allocated memory block. A pointer to this memory is saved to the variable, passed as a function argument.

### 4.2 Analysis of WinMain

Now you will perform an analysis of `wWinMain` function located at address `0x406060`.

Taking general look at this function, it looks rather short.
It also seems that `WinMain` is not using any local variables nor referencing any of its arguments.

Because this function is rather simple, you will analyse it block by block.

For convenience, first go to the last block of the function (loc_40610F) and rename it as `func_exit`:

Now take a look at the first block of the function:
A couple of things take place here. First, you see a call to the `sub_402860` function (line 03). If you take a quick look at this function you will see it is used to dynamically load a few API functions:

```
00402895 push esi
00402896 push edi
00402897 call sub_402860
```

Rename `sub_402860` to `f_Initialize_APIs`.

Then at lines 04-07 and 09 the program is creating an unnamed mutex. The handle to this mutex is then saved to the global variable `hHandle` at line 12. Rename this variable to `hUnnamedMutex`.

Additionally at line 11 some global variable (`dword_438120`) is initialized to zero. You don’t know yet what this variable will be used for in the code but it is good to give it a temporary name, for example `var_main_zero`. If you later see reference to this variable you will immediately know it was first set to zero in the `wWinMain` function.
Finally at lines 10-14, `time()` function is called. The `time()` function returns system time represented as a number of seconds elapsed since January 1, 1970. Then, the result value is compared to variable `dword_43F40` (line 15) and if it is lower, the function quits.

```
00406680 call edi; time
00406682 add esp, 4
00406685 cmp eax, dword_437E40
00406687 jl short Func_exit
```

What is the value of `dword_437E40`? If you check xrefs to it, you will see that this variable seems never to be initialized:

![xrefs to dword_437E40](Image)

However the virtual address `0x43F40` is located in an uninitialized part of the `data` section of `slave.exe` and according to PE-COFF specification\(^\text{22}\) this memory is automatically initialized to zero.

“... SizeOfRawData - The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. **If this is less than VirtualSize, the remainder of the section is zero-filled. ...**”

Moreover since it is logical to compare `time()` result to zero (value -1 is returned on error) we can safely assume this is what is taking place here.

To sum up, the first block program loads a few API functions, creates an unnamed mutex, initializes some variables and checks system time.

---

The next code block is quite interesting.

If the \texttt{time()} result is greater or equal to zero, then the same result is compared to value 0x551B3500 (1427846400). This value is Unix timestamp representation of the date 01 April 2015, 12:00am (UTC). If the \texttt{time()} result is greater than this value, then main function quits. This means that the malware won’t run after this date.

In the next two code blocks, the malware tries to create a named mutex \texttt{"__NTDLL_CORE__"} and checks if it succeeds. If \texttt{CreateMutexW} returns \texttt{INVALID_HANDLE_VALUE} (0xFFFFFFFF) or \texttt{GetLastError} returns \texttt{ERROR_ALREADY_EXISTS} (0xB7) then the function quits. Creation of a named mutex is a typical malware technique to prevent running two or more instances of the same malware on the same system.
In the next two code blocks, the program calls two functions: _sub_406120 and _sub_406410. None of those functions seem to take any arguments and the second function is called only if the first one returns value zero (eax).

In one of the previous exercises, you already found that _sub_406410 is probably installation routine. Indeed if you take a look into it, there are calls to API functions such as: CreateDirectoryW, CreateFileW, MoveFileExW, RegSetValueExW, as well as references to strings such as “Software\Microsoft\Windows\CurrentVersion\Run”. Rename this function to _f_ InstallRoutine.

```
004067F3 push 0 ; lpClass
004067F5 push 0 ; Reserved
004067F7 push offset Subkey ; "Software\Microsoft\Windows\CurrentVersion"
004067FC push [esp+400h+Subkey] ; hKey
00406800 call ds:RegCreate; char Subkey[]
00406806 mov ebx, [esp+400h+Subkey db 'Software\Microsoft\Windows\CurrentVersion'...
0040680A lea eax, [esi+00h+Version\Run\0
0040680D push eax ; cbData
0040680E lea eax, [esp+408h+Data]
00406812 push eax ; lpData
```

At this point you still don’t know what the purpose of the first routine _sub_406120 is. However, knowing that if this function returns a value other than zero, the installation routine won’t execute, you can suspect that _sub_406120 might be checking if the malware was already installed.

```
00406669 loc sub_4060CB: ; lpThreadId
0040666B push 0
0040666D push 0 ; dwCreationFlags
00406670 push 0 ; lpParameter
00406673 push offset _sub_401B90 ; lpStartAddress
00406676 push 0 ; dwStackSize
00406679 push 0 ; lpThreadAttributes
0040667B call ds:CreateThread
0040667D push eax ; hObject
00406681 call ds:CloseHandle
00406687 mov esi, ds:Sleep
0040668D lea ecx, [ecx+0]
```

In the next block, the program is creating a new thread. The thread routine is set to _sub_401B90. Rename this function to _f_ ThreadFunction.
The next three blocks, create a loop. All the loop does is to check system time and compare it to previously checked date of 01 April 2015. If time is greater than this date, the program quits. Otherwise, the program sleeps one minute (60,000 milliseconds) and repeats checking the date.

4.3 Analysis of thread function

In this exercise you will do an analysis of the thread function \( f\_ThreadFunction - \text{sub}_401B90 \). However, unlike in previous examples, you will do only a quick assessment of this function to get a general knowledge about its functionality.

When you first go to \( f\_ThreadFunction \) in IDA Free, you might notice that IDA highlighted some parts of the code in red. This usually indicates that IDA encountered some problem when disassembling the binary and manual code correction might be needed.

However, in this case, it should be enough to tell IDA to reanalyse the code (Options->General->Analysis->Reanalyze program) and IDA will fix references to local variables:
Starting analysis of a function, we see that the program first checks its own process ID and saves it to the local variable var_264 (rename it to PID):

```assembly
00401B9E call ds:GetCurrentProcessId
00401BA4 mov ecx, eax
00401BA6 mov [esp+20h+PID], eax
```

In the next code block, you see calls to CreateToolhelp32Snapshot and Process32FirstW:

```assembly
00401B63
00401B63 loc_401B63: ; th32ProcessID
00401B63 push 0
00401B65 push 2 ; dwFlags
00401B67 mov [esp+20h+ProcessID], 22Ch
00401B6F call ds:CreateToolhelp32Snapshot
00401BC5 mov edi, eax
00401BC7 lea eax, [esp+20h+pe]
00401BC8 push eax ; lppe
00401BCC push edi ; hSnapshot
00401BDD call ds:Process32FirstW
00401BD3 test eax, eax
00401BD5 jz loc_401B6A
```

This means that the thread function will be iterating over the process list. Indeed, if you take a look at the bigger picture of the function, you will notice that the entire thread function is a big loop, iterating over processes:
Next, go to the block where Process32Next is called and rename the block label to *get_proc_next*:

Now if you take a look at the beginning of the loop (block [1]), you will see that the next process PID is compared to the PID of current process:

If both PIDs are equal, program skips loop iteration and tries to check the next process.
Next, take a look at blocks [2], [3] and [4] to see the references to the process names of three popular web browsers: “firefox.exe”, “iexplore.exe” and “chrome.exe”:

```
004018EE mov ecx, offset aFirefox_exe ; "firefox.exe"
004018F3 lea eax, [esp+280h+pe.szExeFile]
```

```
00401C48 lea ecx, [esp+280h+pe.szExeFile]
```

This means that malware is looking for processes of web browsers and it will probably try to inject into some code.

Next if you take a look at [5] you will also see references to names of DLL libraries (“nspr4.dll”, “nss3.dll”, “chrome.dll”, “wininet.dll”) used by the previously mentioned web browsers:

```
00401C67 loc_401C67: ; "chrome.exe"
00401C67 mov eax, offset chrome_exe
00401C8E lea ecx, [esp+280h+pe.szExeFile]
```

Names of DLLs are passed as a second argument to the sub_406950 (fastcall calling convention). At this point you don’t know what sub_406950 is used for but a quick look at it might suggest it is only used to enumerate DLLs of web browser process to check if given library was loaded (calls to CreateToolhelp32Snapshot, Module32First, Module32Next and portions of the code look like some string comparison).
Next at [6] malware is calling `GetSystemInfo`\(^{23}\) (or `GetNativeSystemInfo`\(^{24}\)) which returns various system information in `SystemInfo` structure (IDA automatically recognized this structure on the stack). Then one of the `SystemInfo` fields (`anonymous_0`) is compared to value 9. But what is the `anonymous_0` field in `SystemInfo` structure? This field is not mentioned in Microsoft documentation\(^{25}\).

```c
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwGmId;
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD_PTR dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO;
```

To check what `anonymous_0` field is, first hover mouse over `SystemInfo`:

![Image of stack declaration of `_SYSTEM_INFO` structure]

Here you can see this is a stack declared structure of type `_SYSTEM_INFO`.


Next go to Structures view (View->Open Subviews->Structures). This view presents all well-known data structures recognized by IDA in disassembled code (it is also possible to create custom data structures).

Next find on the list _SYSTEM_INFO. structure.

To expand the structure declaration, click on _SYSTEM_INFO. name and press ‘+’ on numerical keypad.

Here you can see that anonymous_0 field is the first field in _SYSTEM_INFO structure. This means this is a union containing information about processor architecture (wProcessorArchitecture).
Indeed, value 9 to which anonymous_0 field is compared represents AMD64 processor architecture\textsuperscript{26}. This means that malware was checking if it is running on 64-bit system.

The next block is quite interesting from an educational point of view. It shows that you always need to be cautious when doing analysis because sometimes IDA might disassemble something wrongly (without any warning).

\begin{verbatim}
00401D24 push esi          ; dwProcessId
00401D25 push 0             ; blInheritHandle
00401D27 push 400h          ; dwDesiredAccess
00401D2C call ds:OpenProcess
00401D32 mov esi, eax
00401D34 lea eax, [esp+280h+var_260]
00401D38 push eax
00401D39 push esi
00401D3A call ds:IsWow64Process
00401D40 xor ecx, ecx
00401D42 cmp [esp+280h+var_274], ecx
00401D46 push esi            ; hObject
00401D47 setz cl
00401D4A mov [esp+28Ch+var_274], ecx
00401D4E call ds:CloseHandle
00401D54 mov eax, [esp+288h+var_274]
00401D58 jmp short loc 401D60
\end{verbatim}

This code is executed only if malware determines that it is running on 64-bit system. The call to IsWow64Process suggests that malware checks if web browser process is running under WOW64\textsuperscript{27}.

\textsuperscript{26} SYSTEM_INFO structure https://msdn.microsoft.com/en-us/library/windows/desktop/ms724958%28v=vs.85%29.aspx (last accessed 11.09.2015)

According to Microsoft documentation\(^{28}\), \textit{IsWow64Process} is a stdcall function taking two arguments.

\begin{verbatim}
BOOL WINAPI IsWow64Process(
    _In_  HANDLE hProcess,
    _Out_ PBOOL Wow64Process
);
\end{verbatim}

The second argument (\textit{Wow64Process}) is a pointer to a BOOL variable used to return information whether given process is running under WOW64.

In the code, \textit{Wow64Process} is set to the address of \textit{var}\_\textit{26C} variable (\texttt{lea eax, [esp+280h+var\_26C]}). After a call to \textit{IsWow64Process} we would expect value returned in \textit{var}\_\textit{26C} should be checked. But instead you see references to some other variable (\textit{var}\_\textit{274}) which haven’t been yet initialized or referenced.

One of the possible causes of this problem might be that IDA has a wrongly traced stack pointer. And since the thread function is using an \textit{esp} based stack frame this might cause IDA to wrongly interpret variables. Let’s check how IDA traced a stack pointer.

Choose \textit{Options->General} and check the \textit{Stack pointer} checkbox.

Now you should see in disassembly an additional column with the value of the stack pointer as traced by IDA. Notice that each instruction changing the stack pointer (push, pop, etc.) is changing the value in this column and instructions like mov, xor, add, cmp ... are not changing the stack pointer:

```
00401d24 284 push esi   ; dwProcessId
00401d25 288 push 0   ; bInheritHandle
00401d27 28c push 400h ; dwDesiredAccess
00401d2c 290 call ds:OpenProcess
00401d32 284 mov esi, eax
00401d33 284 lea eax, [esp+280h+var_26c]
00401d39 284 push eax
00401d39 28c call ds:IsWow64Process
00401d40 28c xor ecx, ecx
00401d42 28c cmp [esp+280h+var_274], ecx
00401d46 28c push esi   ; hObject
00401d47 290 setz cl
00401d48 290 nov [esp+28ch+var_274], ecx
00401d4e 290 call ds:CloseHandle
00401d54 28c mov eax, [esp+288h+var_274]
00401d58 28c jmp short loc_401d60
```

Stdcall functions are supposed to clean the stack before return. However for some reason, it looks like IsWow64Process is not cleaning the stack at all (the stack pointer doesn’t change even though the function is taking two arguments).

```
00401d38 284 push eax
00401d39 288 push esi
00401d3a 28c call ds:IsWow64Process
00401d40 28c xor ecx, ecx
00401d42 28c cmp [esp+280h+var_274], ecx
```

To see the reason for this, hover mouse over IsWow64Process.
Looks like IDA Free doesn’t know what the proper prototype of `IsWow64Process` and thus IDA didn’t know how many arguments this function is taking nor how it affects the stack pointer. Consequently, IDA assumed that the call to this function is not changing the stack pointer at all.

You can correct this by either manually editing the prototype of the `IsWow64Process` or manually changing how the call instruction is affecting the stack pointer. To demonstrate, let’s use the second method.

Click on the call to `IsWow64Process` and choose `Edit->Functions->Change stack pointer...` (Alt+K). Next enter value `0x8` (because function is taking two DWORD sized arguments):

Now IDA should correctly reference all variables making code much clearer. Notice what was previously referenced as `var_274` is now `var_26C`:

The correction of a stack pointer might be necessary for calls to dynamically computed addresses when IDA doesn’t know what function is called or how it affects stack.

Going back to the thread function analysis, take a look at block [7] where the single function `sub_402050` is called just before loop end.
This function takes a single argument (process ID) and from the call graph for this function, you will see it calls APIs such as WriteProcessMemory or CreateRemoteThread. This means this function is used to inject code into the browser process.

Finally code at [8] is executed after Process32NextW returns FALSE (zero). The code sleeps for 3 seconds and then repeats an enumeration of the entire process list (second loop).

To sum up, you have just done a quick analysis of the thread function. During this analysis you weren’t going into details of what each instruction is doing, but rather you were trying to get a general understanding of the function.

What you have learnt is that the thread function endlessly iterates over the process list in search of the processes of popular web browsers (Mozilla Firefox, Google Chrome and Internet Explorer) to inject some code to such a process in sub_402050. What you haven’t checked is how detection of 64-bit process affects code injection. You have also skipped a call to sub_401DA0 which is a function using mutexes to prevent injection of code twice to the same process.

Additionally you have also learnt how to fix a corrupted stack pointer and how to view data structures recognized by IDA.

4.4 Exercise
Open the dexter.exe sample (the same as in the previous exercise) and try to analyse the following functions:

- sub_401E70 – what this function is used for? How does it return a result?
- sub_402620 – what are the function arguments and how are they used?
- sub_4022B0 – what is this function used for?

For each function do only a quick assessment in order to get general understanding of the function and its role. No detailed analysis is necessary.
4.5 **Summary**

In this exercise you have learnt how to approach to function analysis in disassembled code. When starting to analyse a function it is always good to ask a few standard questions such as what arguments is this function using, what APIs are called and so on. Answering those question might give you valuable information about the function’s purpose. You have also learned that thorough function analysis is not always necessary. In many cases, just a quick assessment could be enough to get a general understanding of the function.
5. Anti-disassembly techniques

As presented in previous exercises, static analysis tools and techniques can teach you a lot of things about malicious code: how it operates, what are its functions, how it installs in the system or how it communicates with a C&C server. Of course this is usually contrary to the intentions of malware creators who would often want us to be unable to analyse code of their creations. Consequently creators of more complex malware often use various anti-disassembly techniques which aim to make analysis of disassembled code much harder.

In this exercise you will learn some of the more popular anti-disassembly techniques. Note that since those techniques affect disassembled code they are usually also a problem during dynamic analysis in which a debugger needs to disassemble code as well.

5.1 Linear sweep vs. recursive disassemblers

To understand anti-disassembly techniques you need to first learn a little more about disassemblers. In general there are two types of disassemblers: linear sweep and recursive disassemblers.

One of the problems with disassembling binary code is code synchronization - that is to tell where each instruction starts and how to distinguish data from executable code. The fact that x86 instructions have variable length doesn’t make this task easier.

For example take a look at hexdump of some executable.

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000A0</td>
<td>83</td>
<td>0D</td>
<td>84</td>
<td>80</td>
<td>43</td>
<td>00</td>
<td>FF</td>
<td>59</td>
<td>53</td>
<td>FF</td>
<td>15</td>
<td>70</td>
<td>01</td>
<td>41</td>
<td>00</td>
<td>0B</td>
</tr>
<tr>
<td>0000000E0</td>
<td>0D</td>
<td>38</td>
<td>7E</td>
<td>43</td>
<td>00</td>
<td>89</td>
<td>08</td>
<td>FF</td>
<td>15</td>
<td>74</td>
<td>01</td>
<td>41</td>
<td>00</td>
<td>8B</td>
<td>0D</td>
<td>34</td>
</tr>
<tr>
<td>0000000C0</td>
<td>7E</td>
<td>43</td>
<td>00</td>
<td>89</td>
<td>08</td>
<td>E8</td>
<td>BS</td>
<td>6A</td>
<td>00</td>
<td>00</td>
<td>83</td>
<td>3D</td>
<td>28</td>
<td>30</td>
<td>41</td>
<td>00</td>
</tr>
<tr>
<td>0000000D0</td>
<td>0D</td>
<td>75</td>
<td>0C</td>
<td>68</td>
<td>80</td>
<td>80</td>
<td>40</td>
<td>00</td>
<td>FF</td>
<td>15</td>
<td>78</td>
<td>01</td>
<td>41</td>
<td>00</td>
<td>59</td>
<td>E8</td>
</tr>
<tr>
<td>0000009E0</td>
<td>8E</td>
<td>03</td>
<td>00</td>
<td>00</td>
<td>33</td>
<td>C0</td>
<td>C3</td>
<td>EE</td>
<td>34</td>
<td>04</td>
<td>00</td>
<td>00</td>
<td>EE</td>
<td>58</td>
<td>FD</td>
<td>FF</td>
</tr>
<tr>
<td>0000009F0</td>
<td>FF</td>
<td>8B</td>
<td>FF</td>
<td>55</td>
<td>8B</td>
<td>EC</td>
<td>81</td>
<td>EC</td>
<td>28</td>
<td>03</td>
<td>00</td>
<td>00</td>
<td>A3</td>
<td>10</td>
<td>7C</td>
<td>43</td>
</tr>
<tr>
<td>000001000</td>
<td>00</td>
<td>89</td>
<td>0D</td>
<td>0C</td>
<td>7C</td>
<td>43</td>
<td>00</td>
<td>89</td>
<td>15</td>
<td>08</td>
<td>7C</td>
<td>43</td>
<td>00</td>
<td>89</td>
<td>1D</td>
<td>04</td>
</tr>
<tr>
<td>000001100</td>
<td>7C</td>
<td>43</td>
<td>00</td>
<td>89</td>
<td>35</td>
<td>00</td>
<td>7C</td>
<td>43</td>
<td>00</td>
<td>89</td>
<td>3D</td>
<td>FC</td>
<td>7B</td>
<td>43</td>
<td>00</td>
<td>66</td>
</tr>
<tr>
<td>000001200</td>
<td>8C</td>
<td>15</td>
<td>28</td>
<td>7C</td>
<td>43</td>
<td>00</td>
<td>66</td>
<td>8C</td>
<td>0D</td>
<td>1C</td>
<td>7C</td>
<td>43</td>
<td>00</td>
<td>66</td>
<td>8C</td>
<td>1D</td>
</tr>
<tr>
<td>000001300</td>
<td>08</td>
<td>7B</td>
<td>43</td>
<td>00</td>
<td>66</td>
<td>8C</td>
<td>05</td>
<td>F4</td>
<td>7B</td>
<td>43</td>
<td>00</td>
<td>66</td>
<td>8C</td>
<td>25</td>
<td>F0</td>
<td>7B</td>
</tr>
<tr>
<td>000001400</td>
<td>43</td>
<td>00</td>
<td>66</td>
<td>8C</td>
<td>2D</td>
<td>EC</td>
<td>7B</td>
<td>43</td>
<td>00</td>
<td>9C</td>
<td>0F</td>
<td>05</td>
<td>20</td>
<td>7C</td>
<td>43</td>
<td>00</td>
</tr>
</tbody>
</table>

Highlighted bytes represent consecutive assembly instructions:

- **E8 34 04 00 00**: call 0x401a20
- **E9 58 FD FF**: jmp 0x401349
- **8B FF**: mov edi, edi

But if you start analysis, for example, at the offset changed by two bytes this would produce completely different assembly code.
Red frames mark previously disassembled instructions while highlighted bytes mark new instructions after disassembling with changed offset.

04 00: add al, 0x0
00 E9: add cl, ch
58: pop eax
FD: std
FF: db 0xFF (incorrect)
FF 8B FF 55 8B EC: dec dword [ebx-0x1374aa01]

The difference between a linear sweep and recursive disassembler is how a disassembler follows consecutive instructions. A linear sweep disassembler tries to disassemble all the code in a code section of an executable. The beginning of a new instruction is always marked with the end of a previous instruction and it doesn’t depend on the instruction type. That is, if there were some bytes injected between instructions, the disassembler would try to interpret them as another instruction.

For example:

In this example, a linear disassembler would try to disassemble bytes 6D 73 67... as an instruction instead of interpreting it as text string. Resulting disassembly would look as follows:

Notice that the first two instructions (push, jmp) are disassembled properly but the rest of the code is completely different.

(Examples of linear disassemblers are WinDbg and disassembler, included in the CFF Explorer.)
Unlike linear disassemblers, recursive disassemblers currently consider disassembled instructions. If the instruction is changing execution flow (jump, call or return instruction) a disassembler tries to adequately interpret this and add the destination address to a list of locations to disassemble. For example if an instruction is an unconditional jump then a disassembler might try to analyse the code at the address where the jump is leading to instead of analysing bytes right after the jump instruction.

However, recursive disassemblers aren’t perfect and there are situations which might cause them problems. One of their drawbacks is that if a part of the code is never directly referenced (neither called nor jumped to), the disassembler might never try to analyse it. Secondly, a recursive algorithm might also not work well if a disassembler doesn’t know the destination address of the call or jump – for example if this address is dynamically computed.

(Examples of recursive disassemblers are IDA and OllyDbg.)

5.2 Anti-disassembly techniques

Anti-disassembly techniques are techniques which try to mislead a disassembler by creating code desynchronization or by affecting program execution flow in some nonstandard way. As a result disassembled code usually becomes incomplete or contains garbage instructions (junk code).

Though they are not strictly anti-disassembly techniques in this category, you can also add techniques which are not trying to directly affect the disassembling process but rather try to make disassembled code more complex and less clear, making static analysis more difficult. Examples of such techniques would be inserting junk instructions or dynamic loading of API functions.

Below there is a short summary of common anti-disassembly techniques:

- **Inserting garbage bytes.**
  This technique works by inserting random bytes in chosen parts of the code. The intention is to make a disassembler interpret those bytes as a normal code, what would then lead to incorrect disassembly. This technique is usually used in conjunction with some other technique.

- **Return address manipulation.**
  This is one of several execution flow manipulation techniques. It works by changing the return address of the current function. This way, while a disassembler is expecting a function to return to the address after a call, the instruction the function would return to is in a completely different part of the code.

- **Middle instruction jump.**
  In this technique one instruction (e.g. push, mov) is used to hide another instruction.

- **Always taken jumps.**
  This technique works by using conditional jumps for which the condition will be always met. Since disassembler will likely not know this, it will try to disassemble bytes following this instruction.

- **Indirect calls based on runtime value.**
  If the jump or call is made to the dynamically computed address/offset then a recursive disassembler won’t know which address should be analysed next. Additionally, if this is a call instruction, a disassembler won’t know calling convention of the destination function and how a called function is changing the stack pointer.

- **Structured Exception Handling (SEH).**
  Structured Exception Handling (SEH) is a mechanism normally used to handle exceptions in programs. It can be also used to obscure execution flow by first installing an exception handler routine and then triggering an exception in some part of the code. As a consequence, program execution will be switched to the exception handler routine.
• **Inserting junk code.**
  This technique works by inserting instructions in the code that have no direct effect on execution and doesn’t change program result. The only aim of this technique is to make disassembled code less clear and harder to analyse (it is usually difficult for the analyst to distinguish real instructions from the junk code).

• **Dynamic API loading.**
  Based on what API functions the malware is calling, you can try to predict its functionality and also recognize the important parts of the code. To make such analysis harder, malicious code frequently dynamically loads important API functions so that they are not present by default in the import address table.

In general, to deal with anti-disassembly techniques it is necessary to have a deep understanding of the analysed code and also know what kind of anti-disassembly techniques you can encounter. In some cases anti-disassembly techniques can be handled manually, usually by following some specific address and forcing it to be interpreted as a code. In other cases anti-disassembly techniques might be so extensive that the only solution is to create some scripts or use dynamic analysis techniques.

### 5.3 Analysis of anti-disassembly techniques

In this exercise you will analyse a specially prepared binary file (non-malicious) which is using various anti-disassembly techniques.

First start by opening antidisasm.exe in IDA:

```
00401000 00401000 public start
00401000 start proc near
00401000 call loc_40101A
00401005 call loc_401045
0040100a call sub_401065
0040100f call sub_401082
00401014 call sub_40116d
00401019 retn
00401019 start endp
00401019
```

You can see here a group of calls to various functions. Each function is using different anti-disassembly techniques and then returns some value in the eax register. The task is to tell what value is returned by each function using only static analysis techniques.

#### 5.3.1 Analysis of a call to loc_40101A

First go to function at 0x40101A.
IDA hasn’t recognized this code as a proper function. Indeed, it seems there is no return from this function because after a call to EAX there is some junk code and loc_401045 is the beginning of the next function.

Notice that at the beginning of loc_40101A there is a strange call (call $+5).

This is very characteristic call – call to the next instruction (0x401022). What it does is pushing onto the stack return address (0x401022) which is then loaded into eax (pop eax). That is by executing pop eax you read the virtual memory address of this exact instruction (0x401022).

Then you add 10h to eax value and call to the address of the newly computed eax value.

At this point you know that the eax value is 0x401032 (0x401022+0x10). Unfortunately this leads us right into the middle of the junk code and it seems there is no instruction at this address.

By now it should be obvious that junk code is likely a result of some code desynchronization. IDA didn’t know what address was called when calling eax and as a result just tried to disassemble next instruction.

To correct this, first select all junk code and then right click it and choose undefined (or press <U>):
Next click on the byte at the address 0x401032 and press <C> to convert it to code. Notice also the string “Fantastic!” right after a call to eax.

Now the code should be much clearer. You can also read return value of loc_40101A which is 0x1337.
To sum up, in this function you have seen two anti-disassembly techniques. First there was an indirect call to dynamically computed address. IDA didn’t know what address was called and thus it just tried to disassemble next instruction which happened to be inline embedded string (second technique). This resulted in creation of junk code instead of valid assembly instructions.

5.3.2 Analysis of a call to loc_401045
The second function which you will analyse is the function at loc_401045.

At first glance even though IDA hasn’t recognized this code as a normal function you can see here a typical function prologue and epilogue with a return instruction. You can also highlight the eax register to check where its value is set.

It seems that eax is first set to 0x11EB and then increased by 0x1000. However what should catch our attention is the jump instruction (jz) which seems to lead to the middle of an instruction. Notice also the red coloured cross reference – suggesting that something is wrong here.
Before we start analysing where this jump leads, let's check if and on what condition it will be taken. The last instruction sets a zero flag before the jump is xor eax, eax which is zeroing eax register and always sets the zero flag. This means that the jump will be always taken.

Since the jump leads to the middle of an instruction, select this instruction and convert it to data (use Undefine or press <U>).

```
; CODE XREF: .flat:00040104A
.push 1158h
.pop  eax
jz short near ptr loc_40104A+1
add    eax, 1600h
```

IDA will likely undefine more code than you intended, but this isn’t a problem since you already know the jz destination address (0x40104B) and where the original jz instruction was located (0x401050).

```
XOR    eax, eax
```

```
; CODE XREF: .flat:000401057
mov    esp, ebp
```

Now select the byte at 0x40104B and press <C> to define code. Do the same with the byte at 0x401050 (jz instruction). After this, you should see code similar to this one:
This means that in the middle of the push instruction was hidden another jump instruction.

As you see the hidden jump is again leading us into the middle of an instruction at 0x40105D (to the address 0x40105E). But this time it looks like a normal assembly desynchronization.

To proceed, go to the undefined instruction at 0x40105D and create code at the address 0x40105E. After those operations code should look as follow:

```
add eax, 1000h
```

Now you can clearly see return value set to 0x4096. Notice that after retn instruction a few garbage bytes were added to prevent IDA from properly disassembling instructions where the eax value is being set.
The screenshot below shows the execution flow of a routine before making any changes to it:

```
.loc_to:                     ; CODE XREF: start+5T
    push    ebp
    mov     ebp, esp
    xor     eax, eax
.loc_from0:                 ; CODE XREF: .flat:0040185j
    push    11C0h
    pop     eax
    jz      short near ptr loc_40104A+1
    add     eax, 1000h
.loc_to:                     ; CODE XREF: .flat:00401057:
    mov     esp, ebp
    pop     ebp
    retn
.loc_to:                     ; CODE XREF: .flat:0040105B:
    adc     esi, [edi]
.loc_to:                     ; CODE XREF: .flat:0040105D:
    adc     [eax+4096h], bh
.loc_to:                     ; CODE XREF: .flat:00401063:
    jmp     short loc_401057
```

To sum up, in this routine you have seen a few anti-disassembly techniques. The most notable one is the jump into the middle of another instruction. In this scenario, a push instruction was used to conceal another jump instruction. You have also seen usage of a conditional jump that is always taken as well as the use of garbage bytes to desynchronize disassembled code.

5.3.3 Analysis of a call to sub_401065

The next call is made to `sub_401065`. This time, IDA recognized this code as a normal function:

```
0041065 ; Attributes: bp-based frame
0041065
0041065 sub_401065 proc near
0041065 push    ebp
0041066 mov     ebp, esp
0041068 xor     eax, eax
004106A push    1000h
004106C call    sub_40107D
0041074 add     eax, 1000h
0041079 mov     esp, ebp
004107B pop     ebp
004107C retn
004107C sub_401065 endp
```

What you see here is that the `eax` register is first zeroed, then some function `sub_40107D` is called (with argument 0x1000) and finally you add 0x1000 to `eax`. The question is whether `sub_40107D` changes `eax` to return some value.

Let’s take a look at `sub_40107D`:
It looks like the only thing this function is doing with eax is first loading arg_0 value (0x1000) and then adding another 0x1000. Thus after the function returns, eax should have value 0x2000. Does it mean that return value of sub_401065 is 0x3000 (0x2000+0x1000)?

As you might have suspected, it is not that easy. Take a look what happens just before sub_40107D returns:

First load to edx the stack address of the first argument and then subtract 4 bytes from edx. What does the address stored in edx point to now? Remember stack frame structure:

```
arg_0     ebp+8
ret. addr. ebp+4
ebp       ebp
```

After subtraction, edx points to the return address stored on the stack. Then, in the third line, we add 0x2B to the return address value. This means that return address of the function was changed and sub_40107D will now return to a different place of the code.

To check where the function will now return go back to the sub_401065:
The original return address should be \texttt{0x401074}. But you know it was increased by \texttt{0x2B}. This means that function \texttt{sub\_40107D} will return to the address \texttt{0x40109F} (\texttt{0x401074+0x2B}). Switch from graph view to the text view and search for this address.

Not surprisingly you see some junk code stored at this location. Undefine (\texttt{<U>}) this code and then create new code (\texttt{<C>}) starting at the address \texttt{0x40109F}.

You have just found final \texttt{eax} value which is \texttt{0xC0DE}!

To sum up, in this section, you have seen a quite popular anti-disassembly technique which is return address replacement. Malicious code trying to deceive the disassembler replaces return address in call to a certain function so that it would point to a completely different part of the code than the disassembler expects.

5.3.4 Analysis of a call to \texttt{sub\_4010B2}

Now you will analyse a call to subroutine \texttt{sub\_4010B2}. 
If you go to this function you will see a long disassembled code with many operations on the \texttt{eax} register. However if you take a closer look at the code you might notice groups of instructions that are not doing anything (some of them might change some flags but this is not relevant in this example).
This is a little simplified version of a technique, in which blocks of junk instructions having no effect on the program execution and only making manual analysis harder are injected into real code.

The only way of dealing with such code is to try to look for any repeated pattern of junk code in disassembly. If you notice such pattern you might try to eliminate it by writing script which would overwrite junk code with NOP instructions or highlight it with some colour. However writing scripts in IDA is not a part of this course.

If you analyse the code a little more, you will notice that only three instructions have an effect on the final eax value:

```
0x4010B2 push ebp
0x4010B3 mov ebp, esp
0x4010B5 xor eax, eax
0x4010B7 push eax
0x4010B8 mov eax, 40000h
0x4010BD add eax, 1430BE3h

0x4010F4 push eax
0x4010F5 push 2000h
0x4010F8 push ecx
0x4010FB add esp, 12
0x4010FC mov eax, 1000h
0x401103 push ecx
0x401104 push edx
0x401105 mov ecx, 52Ah
0x40110A add ecx, 7

0x401110 pop eax
0x401111 pop eax
0x401112 inc edx
0x401113 dec edx
0x401114 add eax, 50h
0x401119 push eax
0x40111A mov eax, 100h
0x40111F add eax, 0C8h
0x401124 pop eax
```

This means that the final eax value will be 0x1500.

5.3.5 Analysis of a call to sub_40116D
The last call which you will analyse is a call to sub_40116D:
In this routine, the `eax` register is seemingly set to `0xEBFE` value. However you should immediately notice the instruction `mov fs:0, esp` which tells us that a new Structured Exception Handler (SEH) is being installed.

Information about all exception handlers is stored in the list of EXCEPTION_REGISTRATION structures:

```
_EXCEPTION_REGISTRATION struc
    prev      dd      ?
    handler   dd      ?
_EXCEPTION_REGISTRATION ends
```

This structure consists of two fields. The first field (`prev`) is a pointer to the next EXCEPTION REGISTRATION structure while the second field (`handler`) is a pointer to exception handler function.

The pointer to the first EXCEPTION_REGISTRATION structure (list head) is always stored in the first DWORD value of the Thread Information Block (TIB). On the Win32 platform, the TIB address is stored in FS register, thus by executing `mov fs:0, esp`, you are setting the first exception handler to the EXCEPTION_REGISTRATION structure created on the stack.

```
00401173 push 15232A1h    ; SEH handler
00401178 push large dword ptr fs:0 ; SEH prev
0040117F mov large fs:0, esp
```

In the case of `sub_40116D`, the stack would look as follows (after SEH installation):

---

29 To get more information about SEH refer to [https://www.microsoft.com/msj/0197/exception/exception.aspx](https://www.microsoft.com/msj/0197/exception/exception.aspx) (last accessed 11.09.2015)
The next question should be whether any exception is triggered in this function? Yes, take a look at the ecx register: First, it is zeroed and then the program tries to write a DWORD value to the address pointed by this register. However, because ecx points to unallocated address 0x00000000 this will cause an exception (STATUS_ACCESS_VIOLATION – 0xC0000005) and program execution would be switched to the installed exception handler.

But what is the address of the exception handler routine? In this example you see that the value 0x15232A1 is being pushed onto stack as an exception handler. But this is not a valid address of any function. Indeed, notice the xor instruction xoring the exception handler address on the stack with value 0x1122300. This means that the real exception handler address is:

0x15232A1 xor 0x1122300 = 0x4011A1

To calculate xor value you can use IDA calculator (View -> Calculator):

Now switch from graph view to text view and search for an address 0x4011A1:
Repeat steps from previous exercises to convert data at 0x4011A1 to code:

```asm
.toggle 0x4011A1 loc_40119B: ; CODE XREF: .toggle 0x4011AD j
.toggle 0x4011B8 pop ebp
.toggle 0x4011C3 ret
.toggle 0x4011C9 sub_40116D endp
.toggle 0x4011C9 ; -----------------------------
.toggle 0x4011C9 db 65h ; e
.toggle 0x4011C9 db 60h ; h
.toggle 0x4011C9 db 6Ch ; l
.toggle 0x4011C9 db 66h ; o
.toggle 0x4011C9 ; -----------------------------
.toggle 0x4011C9 mov eax, 512h
.toggle 0x4011C9 mov esp, [esp+8]
.toggle 0x4011C9 add esp, 8
.toggle 0x4011C9 jmp short loc_40119B
.toggle 0x4011C9 ; -----------------------------
.toggle 0x4011C9 db 0
```

What you see here is that `eax` is assigned with the value `0x512`. Other instructions just restore stack pointer and jumps to the end of `sub_40116D`.

To sum up what you have seen in this subroutine was a usage of Structured Exception Handling (SEH) to change the execution flow of the program. SEH is commonly used as both an anti-disassembly and an anti-debugging technique. Additionally, the address of the exception handler routine was obscured with a xor operation.

### 5.4 Exercise

After completing the analysis of all anti-disassembly techniques in the sample, try to repeat this exercise but using OllyDbg instead. This executable is not performing any malicious actions so you don’t need to worry about accidentally executing it. When debugging in OllyDbg, try to follow execution using `Step into (F7)` function instead of stepping over analysed functions.

- How does disassembled code in OllyDbg differ from the code initially disassembled by IDA?
- Was analysis easier in OllyDbg or IDA?
6. Training summary

In this training, students had the opportunity to learn various aspects of advanced static analysis using IDA Free. First they learnt how to use IDA and what features it offers. Then they learnt how to find significant parts in disassembled code and how to analyse functions. Finally, students reviewed common anti-disassembly techniques and how to deal with them. Some of the more advanced features of IDA like scripting, creating plugins or F.L.I.R.T. signatures were not covered in this document because they require more advanced training and some features are not available in the free version of IDA.
Appendix A: Answers to exercises

Exercise 2.3

*Name a few functions imported by PuTTY executable.*

Click View->Open subviews->Imports:

![Imports table]

What sections are present within executable?

Click View->Open subviews->Segments:

![Segments table]

Sections: .text, .idata, .rdata, .data.

This can be also checked using other tools (e.g. CFF Explorer).

*What do strings tell you about this binary?*

Click View->Open subviews->Strings

There are many descriptive strings in the binary. In general, strings give away that you are analyzing PuTTY, a network application using many different protocols and cryptographic functions.

- There are many strings hinting to “PuTTY” name and PuTTY version.
- There are many strings with names of network protocols, e.g. ssh, telnet, rlogin.
- There are strings pointing to cryptographic functions (AES, Blowfish, 3DES) suggesting that executable is using some form of cryptography.
- There are various caption messages suggesting PuTTY functionality, e.g. “Options controlling proxy usage”.

---

114
There are many error messages also suggesting PuTTY capabilities.

```
"." rdata:0046C84  00000024 C Proxy error: Unexpected proxy error
"." rdata:0046C8A  00000053 C Proxy error: Server chose username/password authentication but we didn't c...
"." rdata:0046CFC  00000034 C Proxy error: We don't support GSSAPI authentication
"." rdata:0046D30  0000003E C Proxy error: SOCKS proxy returned unrecognised address format
"." rdata:0046D70  00000021 C Unrecognised SOCKS error code %d
"." rdata:0046752  00000025 C Using CryptoCard authentication %s%
"." rdata:004675A  0000001E C SSH CryptoCard authentication
"." rdata:004675C  0000001E C Received CryptoCard challenge
"." rdata:004675E  0000002D C CryptoCard challenge packet was badly formed
"." rdata:0046EF8  0000000D C HMAC-SHA-256
"." rdata:0046EFS  0000000E C hmac-sha2-256
"." rdata:0046EFA  00000008 C SHA-256
```

**Exercise 2.6**

*Find function sub_4497AE. What API calls are made within this function?*

Called API functions:

- RegOpenKeyA
- RegQueryValueEx
- RegCloseKey
- LoadLibraryA
- GetProcAddress

**Go to the address 0x406AFB. To which function does this address belong?**

Function sub_40486C.

**Go to the address 0x430EAB. Is there anything special about the instructions stored at this address?**

At this address there is code which is not part of any function. Probably some function wasn’t recognized by IDA as a proper function.

```
.text:00430EAB  loc_430EAB: ; CODE XREF: .text:00430E97J
.text:00430E8B cmp  dword ptr [ebx+4030h], 2Eh
.text:00430EB2 jb loc_430FEB
.text:00430EB6 push 2Eh
.text:00430EB8 lea  esi, [ebx+20h]
.text:00430EBA push  offset aSshconnection_0 ; "SSHCONNECTION@putty.p
```

**Exercise 2.9**

*Find where variable var_8 is used and rename it.*

cur_process_id – this variable is used to store ID of the current process.
Try to rename remaining locations: loc_44D2B1, loc_44D2DA, loc_44D36B, loc_44D3B4. What names would you suggest for them?

loc_44D2B1 – file_loop, file_iteration, ...

loc_44D2DA – get_curr_process_id, pid_check, ...

loc_44D36B – check_cryptacquire_success, cryptoacquire_check, ...

loc_44D3B4 – release_crypt_context, crypt_release, ...

Group three graph nodes checking if functions CryptAcquireContextA, CryptGenRandom and CryptReleaseContext were resolved correctly (0x44D36B, 0x44D374, 0x44D37C).

Can you guess what function sub_44D262 might be used for?

Function takes one argument – function pointer (ebx). Then it gathers information about file names (FindNextFileA), current process ID (GetCurrentProcessId) and also generates block of random data (CryptGenRandom). After each of those calls some data is received (file names, process ID and block of random data). Then this data is passed always to the same function (ebx).
Because non-uniform and random data is passed multiple times to the same function this suggests that this function is likely used as some random data pool collector.

To confirm this guess you would need to analyze where sub_44D262 was called from. There are also two additional function calls in func_exit block which should be likely inspected first.

```assembly
0044D3BE func_exit:
0044D3BE push ebx
0044D3BF call sub_44F63E
0044D3C4 pop ecx
0044D3C5 call sub_44D0C8
0044D3CA pop zero
0044D3CB pop esi
0044D3CC pop ebx
0044D3CD leave
0044D3CE retn
0044D3CE sub_44D262 endp
```

**Exercise 4.4**

*Find network related functions.*

`sub_402710` – calls to functions such as InternetOpenA, InternetConnectA, HttpSendRequestA. There are also references to strings such as “http://%s%s”, “/test/gateway.php” or “193.107.17.126”.

*Find installation routine.*

`sub_402EC0` - called from main, there are calls to CopyFileW, RegSetValueExW, DeleteFileW. It also references strings such as “Software\Microsoft\Windows\CurrentVersion\Run”.

*Find function performing RAM scraping (reading memory of other processes).*


*Find process injection routine.*

`sub_403550` – calls to CreateRemoteThread.

`sub_403370` – calls to WriteProcessMemory (called from `sub_403550`).

*Are there any other potentially interesting or suspicious functions?*

`sub_401E70` – references strings with different operating systems names.

`sub_4022B0` – references strings such as “&spec=", “&query=", “&ver=" which looks like some HTTP GET request parameters.

`sub_4045B0` – references strings such as “update-“, “checkin:”, “scanin”.

`start (0x4036B0)` – start routine.

**Exercise 5.4**

`sub_401E70` – what is this function used for? How does it return result?

Function is used for OS identification. String containing operating system name is copied to memory buffer passed to this function as an argument.
Advanced artefact analysis
Advanced static analysis

sub_402620 – what are function arguments and how are they used?
Function takes three arguments (renamed on the screenshot for clarity):

All three arguments were recognized by IDA as string pointers.

IpString2 (second argument) is processed in calls to sub_4017C0 and sub_401830 and result is copied to the allocated buffer (lpMem). You might decide to analyze both calls to learn how they affect value of IpString2.

Short before sub_402620 returns, there are two string concatenation operations. First lpString1 is concatenated to lpString3. Then lpMem buffer is concatenated to IpString3.

mov     eax, [ebp+IpString1]
push    eax                      ; lpString1
mov     ecx, [ebp+IpString9]     ; IpString3
push    ecx                      ; IpString3
call    ds:istrcatA              ; concatenate lpString1 to lpString3
mov     edx, [ebp+lpMem]         ; lpMem
push    edx                      ; lpMem
mov     eax, [ebp+IpString9]     ; IpString3
push    eax                      ; IpString3
call    ds:istrcatA              ; concatenate lpMem to lpString3
Based on this short analysis you can tell that function takes three string pointer arguments (arg1..arg3). Then performs following operation written in pseudocode:

\[ \text{arg3} += \text{arg1} + f(\text{arg2}) \]

Where \( f() \) is function somehow processing second string argument.

**sub_4022B0 – what is this function used for?**

In this function there are calls to functions like GetUserNameA, GetComputerNameA, sub_401E70 (which you should already know that is returning the name of the operating system). There are also references to strings such as “&spec=”, “&query=”, “&ver=”, “32 Bit”, “64 Bit”.

Function itself is called from sub_402710 which, as it was already found out, is a function used to communicate with C&C server.

This suggests that this function is used to construct string with parameters to HTTP GET request containing various information about infected system. You can do more detailed analysis to check all parameters in constructed GET request.

**Exercise 6.4**

In this exercise it should be enough to debug using only Step into (F7) and read return value from EAX register just before function end.

In this exercise for a few times you will hit part of the disassembly which wouldn’t be recognized by OllyDbg as an assembly code:

```
00400100  56  48 89 5A  00 00 00  5B  89 05  00 00 00  5D
00400105  8B  DD  8B  00  8B  01  8B  02  8B  03  EC  DC  00  0F
```

To fix this select group of bytes starting at the current EIP location (black square), right click on the selection and from the context menu choose: Analysis->During next analysis, treat selection as->Command.

This should fix the problem:
Special attention is only required in last function (0x40116D) which uses Structured Exception Handlers (SEH) to hide some code.

When you hit the instruction at which exception occurs (at 0x40118E) OllyDbg would stop and inform you at status bar that access violation exception has occurred:

Open SEH View (View->SEH Chain) to check if there are any extra exception handlers:

You can see that there is one exception handler defined in current module. Select it and press F2 to put breakpoint on its address. Answer ‘Yes’ in suspicious breakpoint dialog.
Then press Shift+F9 to resume execution and pass exception handling to the program. You should immediately land at exception handling code:

```
004011B0: 08 120E3000 MOV EAX,512
004011B4: 88 4424 88 MOV EFLAGS PTR SS:[ESP+8]
004011B8: 88 44 88 ADD ESP,8
004011BC: EB EC JMP SHORT antidisa.0040119B
```

Tell OllyDbg to treat those instructions as a normal code (Analysis->During next analysis, treat selection as->Command) and continue instruction stepping.