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 ComCERT[1] (Poland), S-CURE[2] (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.

[1] Dawid Osojca, Paweł Weżgowiec and Tomasz Chlebowski
[2] Don Stikvoort and Michael Potter

Table of Contents

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[3] 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!

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

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 – PuTTY[4]. Since this code is not malicious, you don't need to worry about accidentally executing it.

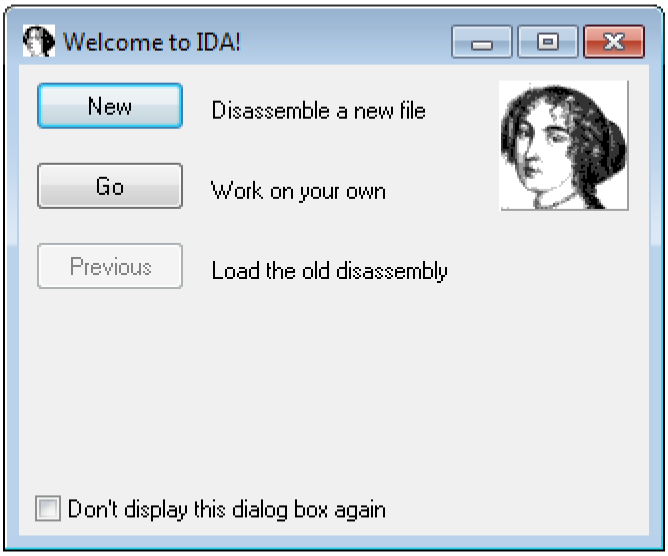
2.1 Opening and closing samples

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

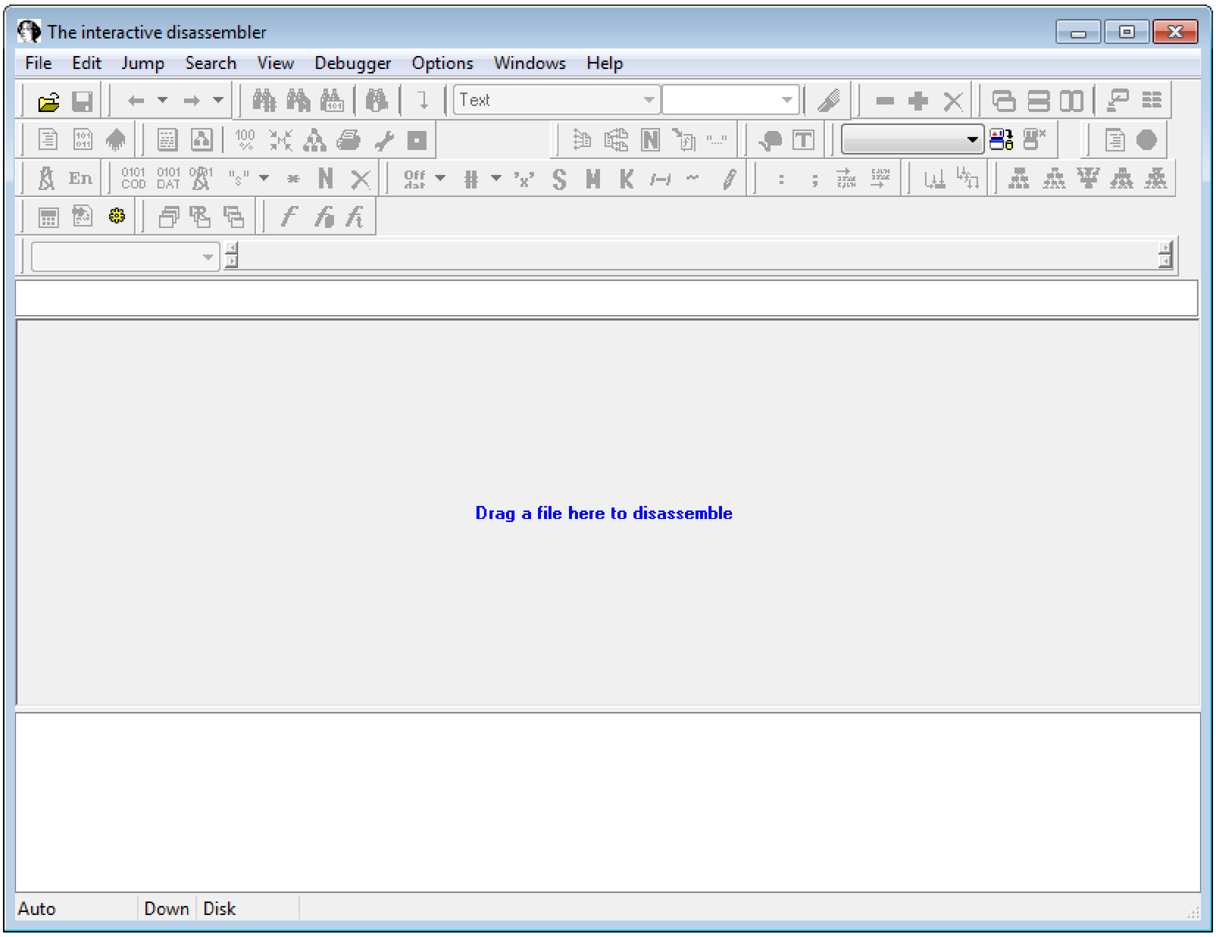
At the beginning of the session you will be presented with the *About* window. Just click *Ok.*

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.

[4] PuTTY: A Free Telnet/SSH Client http://www.chiark.greenend.org.uk/~sgtatham/putty/ (last accessed 11.09.2015)

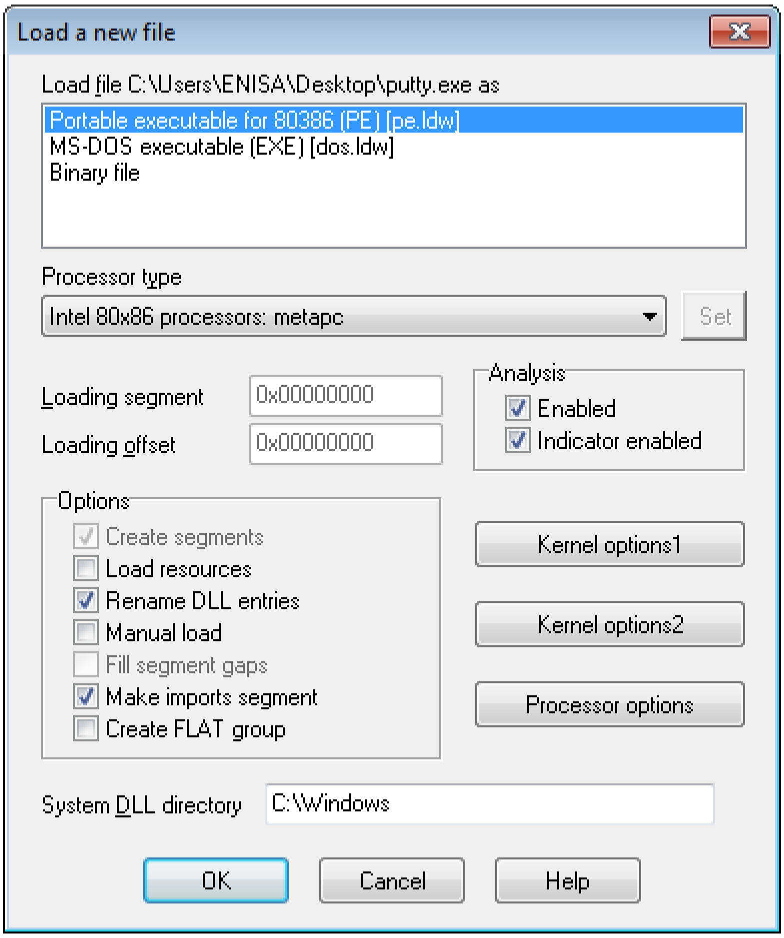


You will be now presented with the main IDA Free workspace window.



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.

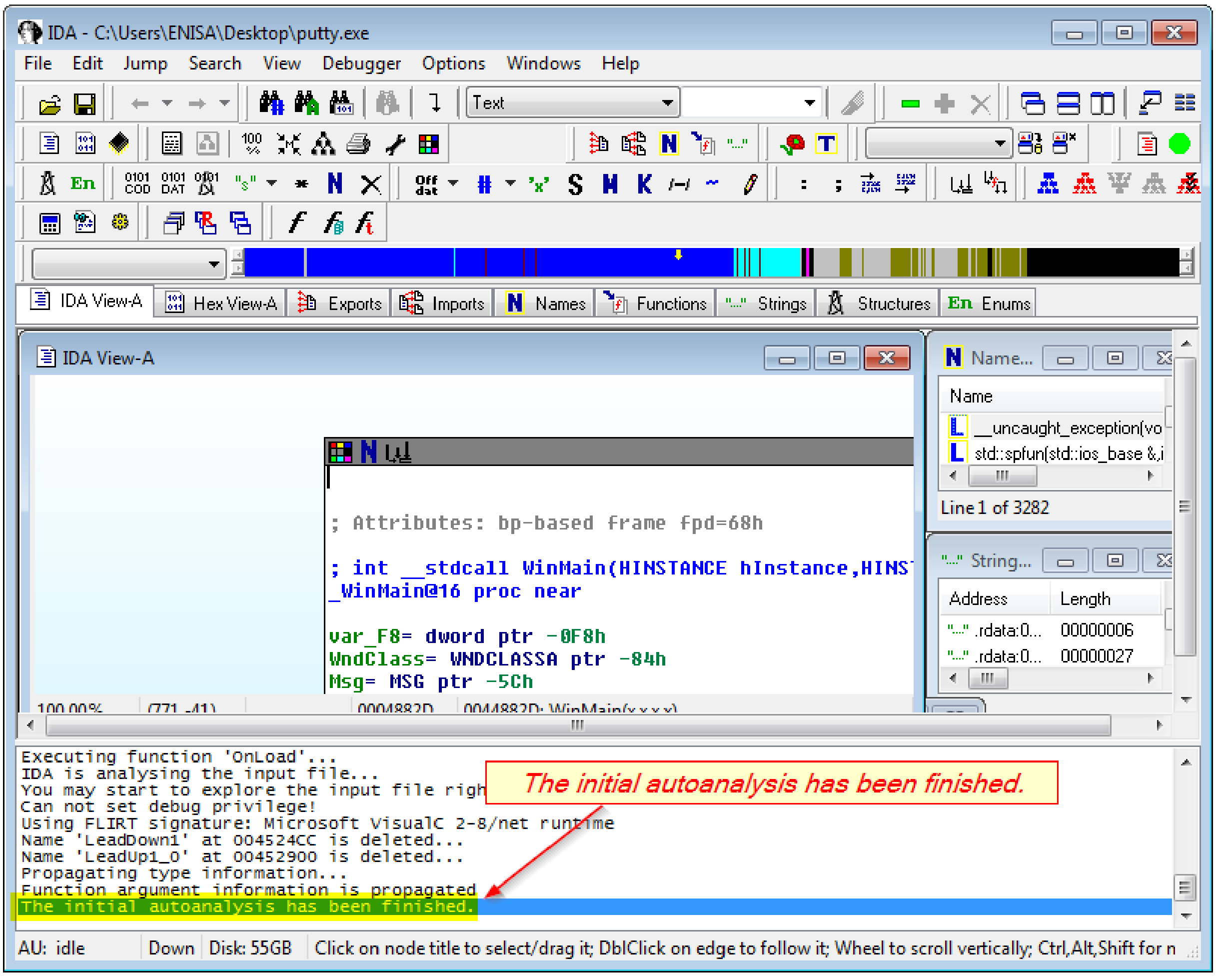
If you had chosen to load putty.exe as a *Binary file,* IDA would have loaded file contents at given memory address (specified with *Loading offset* parameter) without doing extensive analysis. For example it wouldn't try to read PE headers nor recognize the import address table (IAT) or check entry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

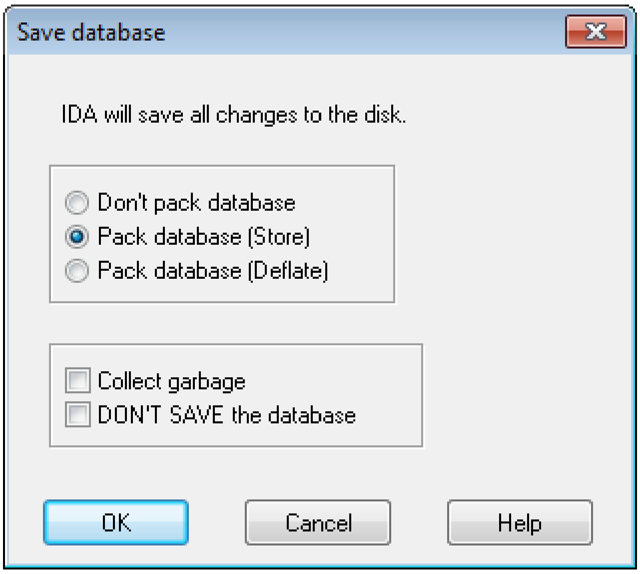
In this exercise, leave all default options set as shown on the screenshot and press *Ok* button.

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.).

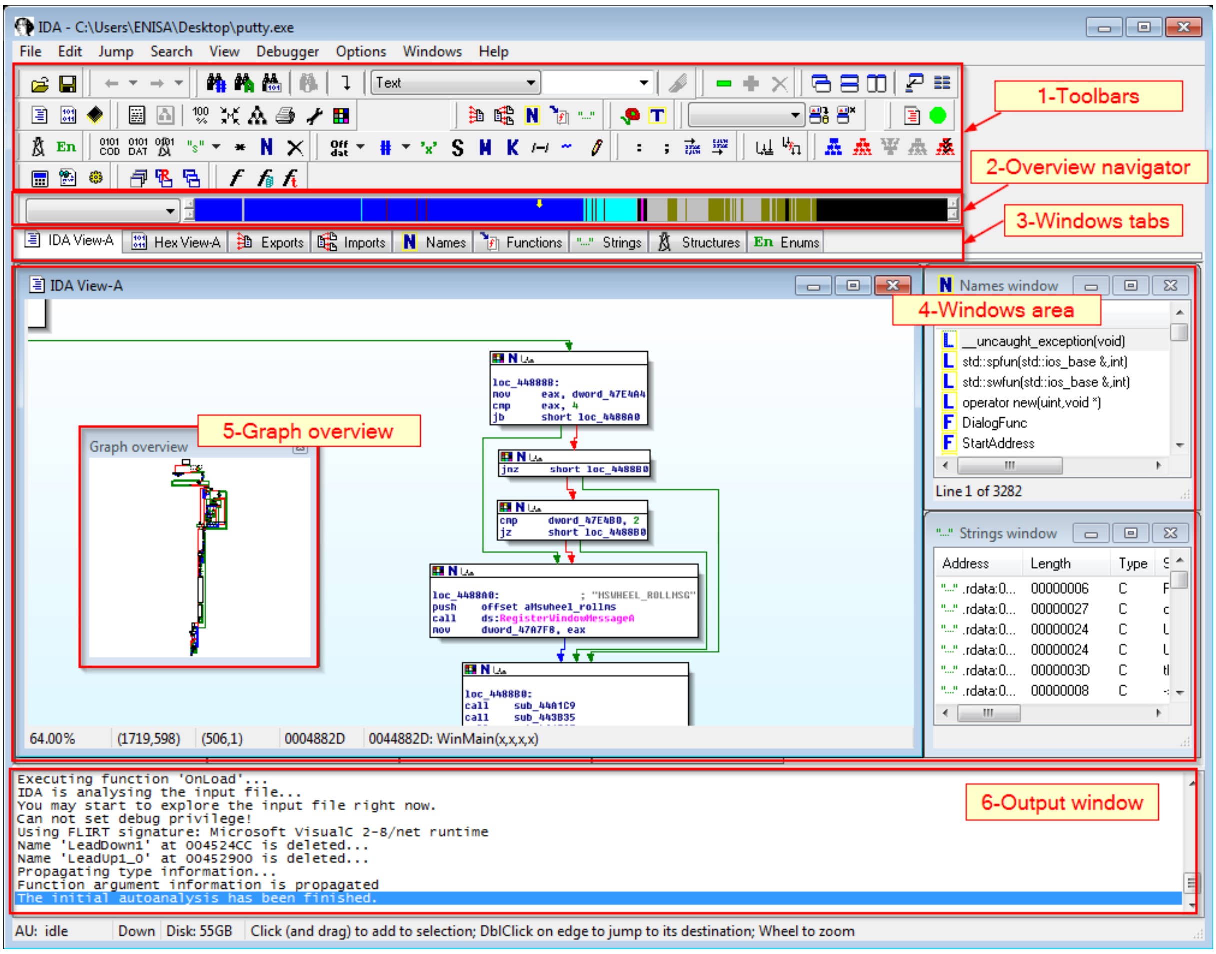
When finishing the analysis by either quitting IDA Pro or selecting *File->Close*, IDA will ask whether to pack database files (*Pack database (store)* - recommended) or leave unpacked files. You can also choose to finish analysis without saving any results (*DON'T SAVE the database* option).



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

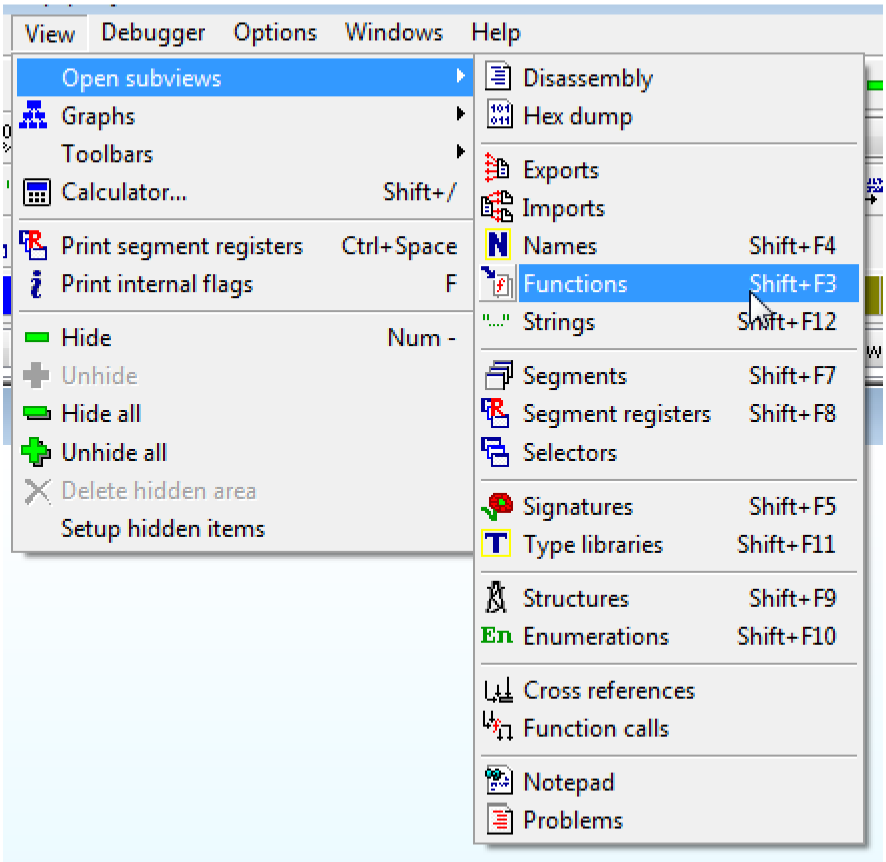
First, load putty.exe as described in the previous step (or open a saved session). After IDA finishes its analysis, you are presented with the default IDA workspace consisting of various windows and other elements. At a first glance IDA interface may look quite complicated but it will become much clearer when you get to know it better.



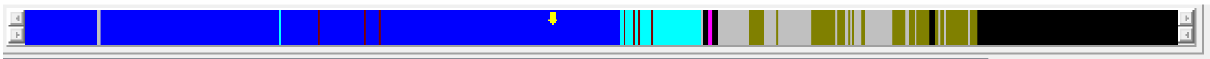
The central part of the workspace is occupied by the *Windows area* (4). IDA uses multiple windows to present various types of information about the disassembled binary. Among the most frequently used windows are:

- *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

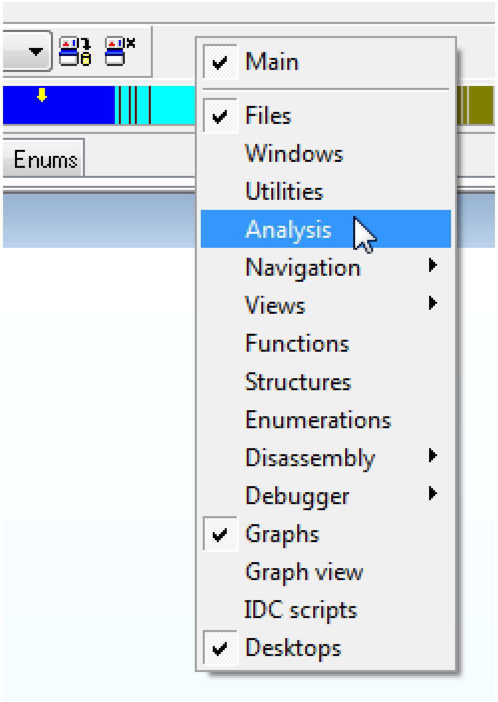
Take some time to switch between the different data views (windows) and check what type of data is presented in each of them.

- *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.

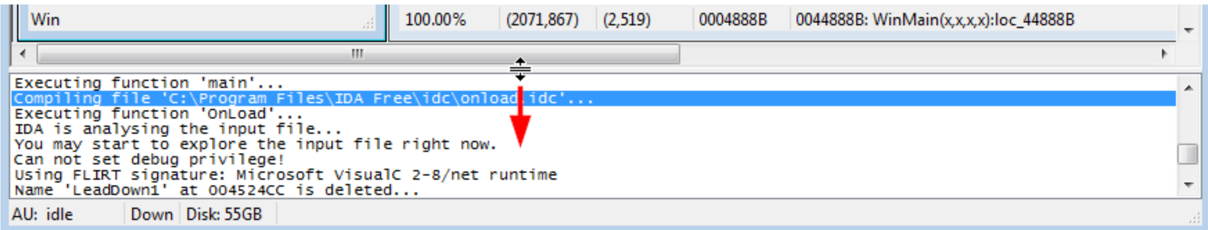
Let's get rid of some of the toolbars first (toolbar functions can be accessed through menus or shortcuts). Right click on the toolbars (1) and uncheck unnecessary toolbars in the context menu.



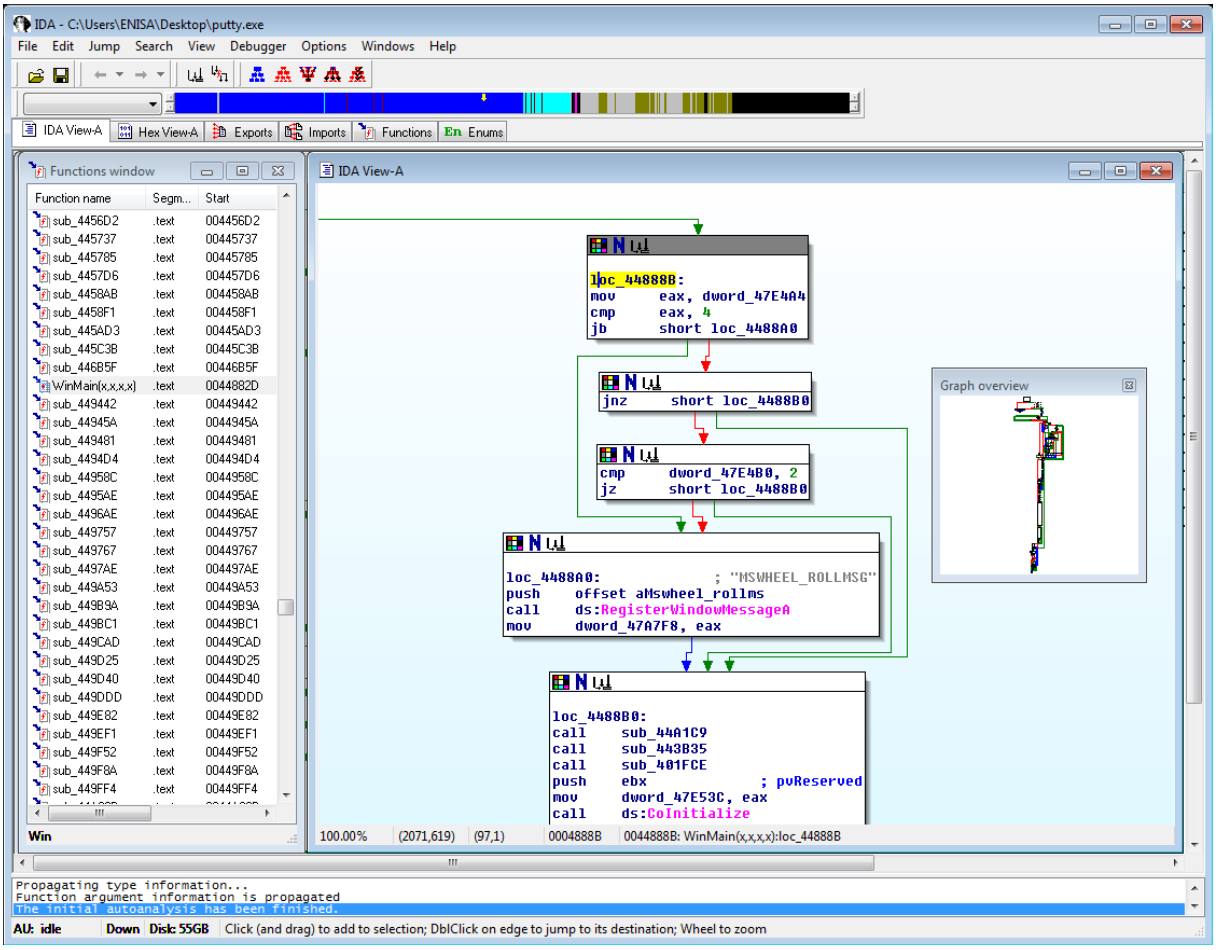
It is up to you which of the toolbars you want to use. You can even decide to remove all toolbars. In the example below we display the following toolbars:

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

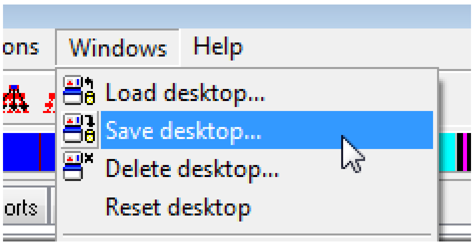
It is also worth resizing output window (6), which is rarely used during analysis.

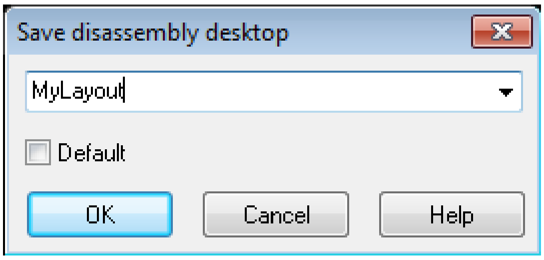


Next, rearrange all the windows and toolbars to give IDA a cleaner look. Since the functions window and disassembly window will be very frequently used, it is good to have them on top. Moreover, it is also good to maximize IDA window if you haven't done so already.



When you are satisfied with the layout, save it using *Windows->Save desktop* option.





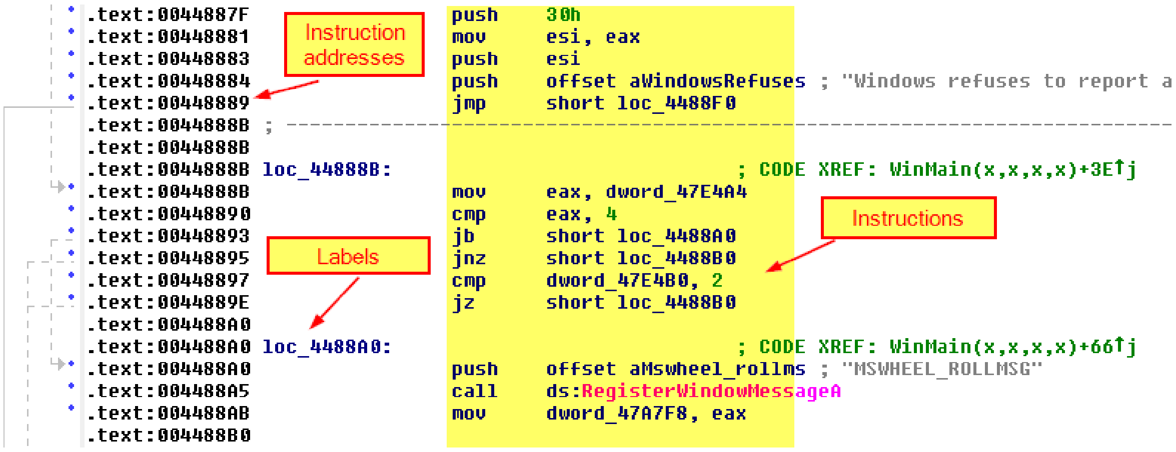
Now whenever you start a new analysis or your layout gets messed up you can quickly restore it using *Windows->Load desktop* option.

2.4 Disassembly view

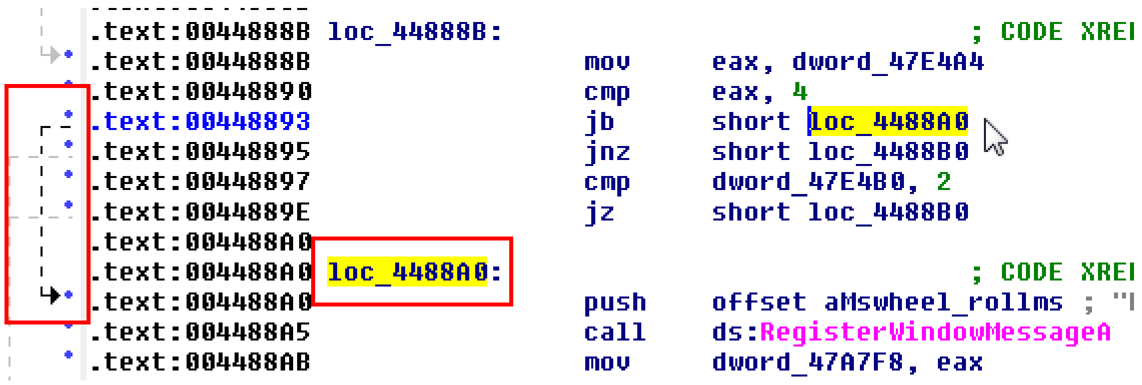
Central to IDA Pro is the assembly view (*IDA View-A*). In the assembly view, IDA presents disassembled code along with all recognized functions.

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.

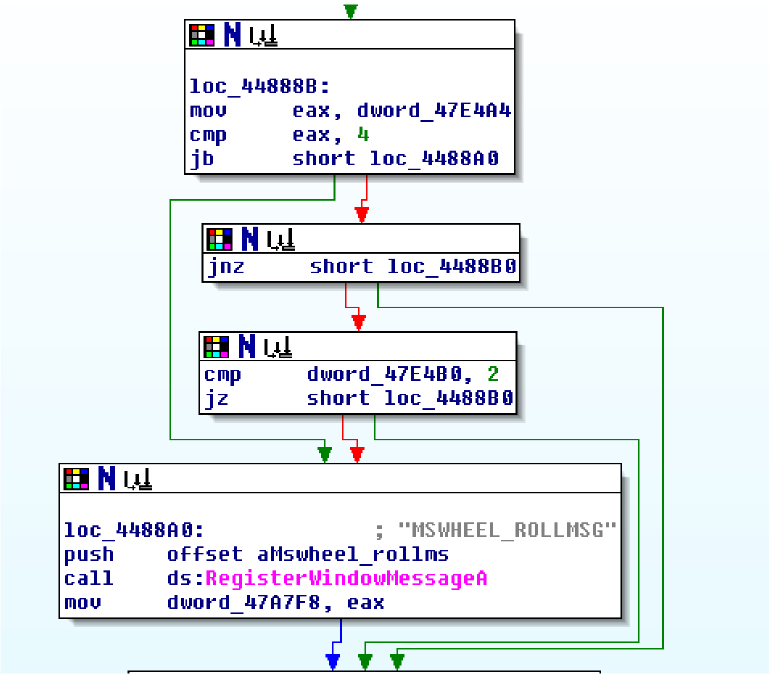
In text view, you can see a linear listing of all disassembled instructions. Text view is useful when you want to analyse parts of the code that IDA hasn't recognized as proper functions.



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.



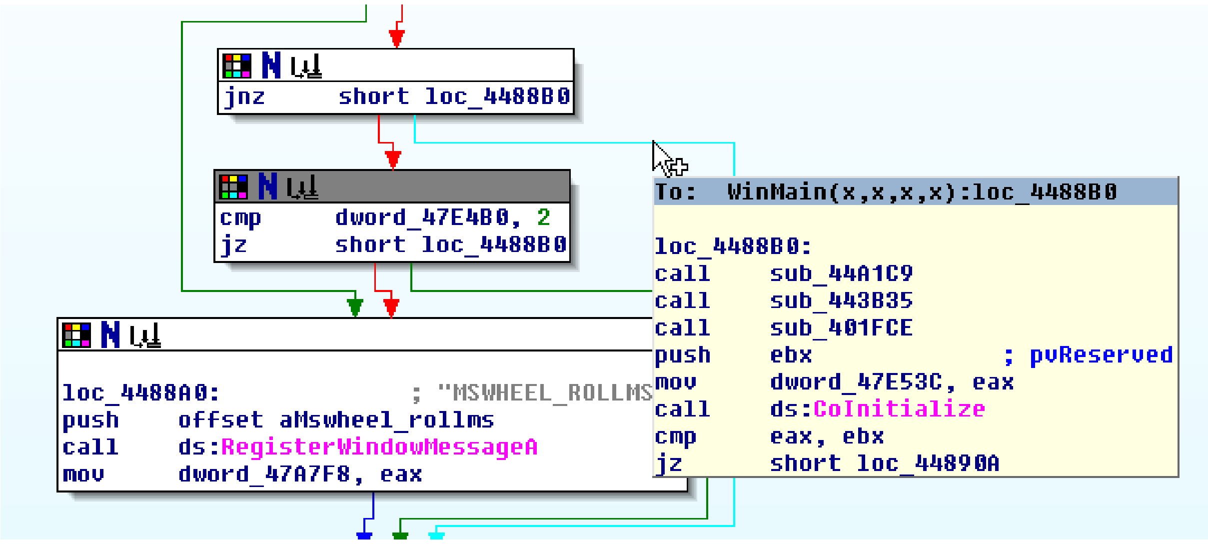
The second type of assembly view is graph view. In the graph view, as the name suggests, IDA presents disassembled code in the form of a graph, where nodes are represented by blocks of disassembled code and lines are branches and unconditional jumps. For each recognized function, IDA creates a separate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



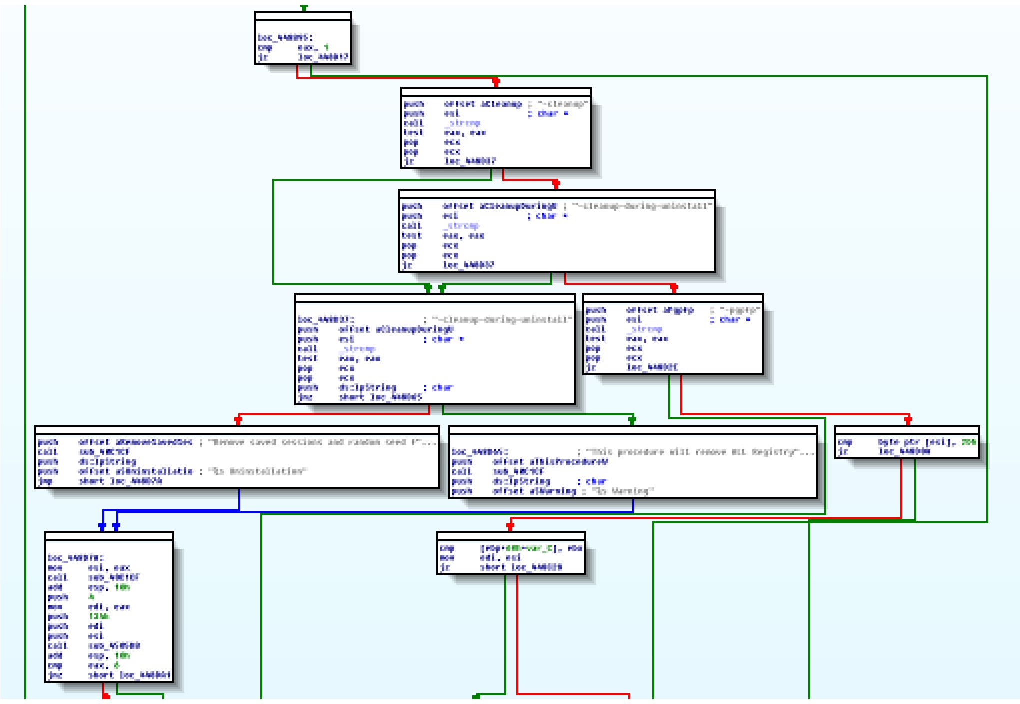
Different colours of the lines are used to indicate different types of code transitions:

- 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.

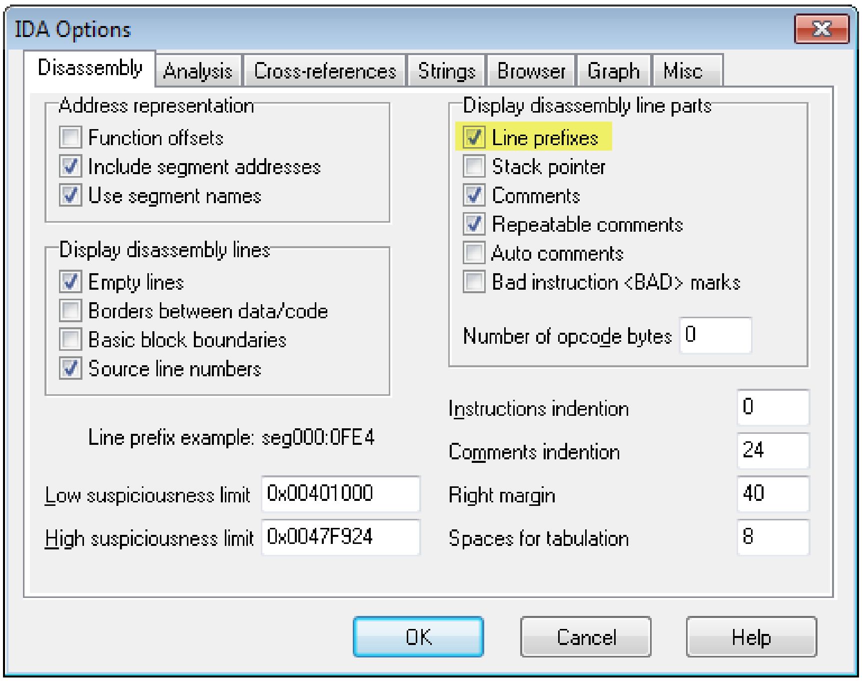


Another very useful feature of IDA is its highlighting capability. You can click on almost any name (register, operation, variable, comment, etc.) and IDA will highlight every other occurrence of this name. For example, you can highlight push/pop operations to track registry changes or highlight a particular registry to track which instructions are changing it.

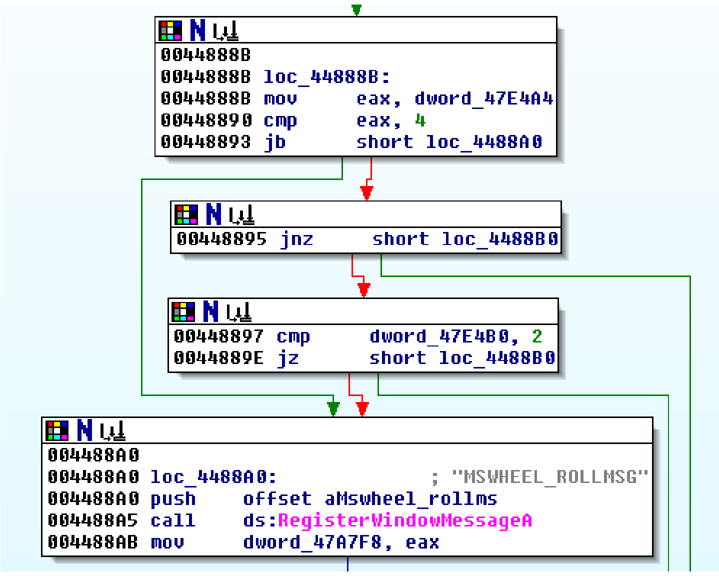
```
loc_44890A:                             ; CODE XREF: WinMain(x,x,x,x)+A0↑j
                                        ; WinMain(x,x,x,x)+A5↑j
                mov     eax, ds:dword_45D4FC
                push    eax
                mov     [ebp+68h+var_C], ebx
                mov     [ebp+68h+nHeight], ebx
                mov     dword_47E534, eax
                call    sub_40F207
                cmp     eax, ebx
                pop     ecx
                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)+FC↑j
                push    ebx
                push    73h
                push    dword_47E53C
                call    sub_4025A5
                push    dword_47E53C
                push    ebx
                call    sub_411A96
                mov     edi, [ebp+68h+nCount]
```

```
call    ds:CoInitialize
cmp     eax, ebx
jz      short loc_44890A
cmp     eax, 1
jz      short loc_44890A
push    ds:lpString      ; char
push    offset aSFatalError ; "%s Fatal Error"
```

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.



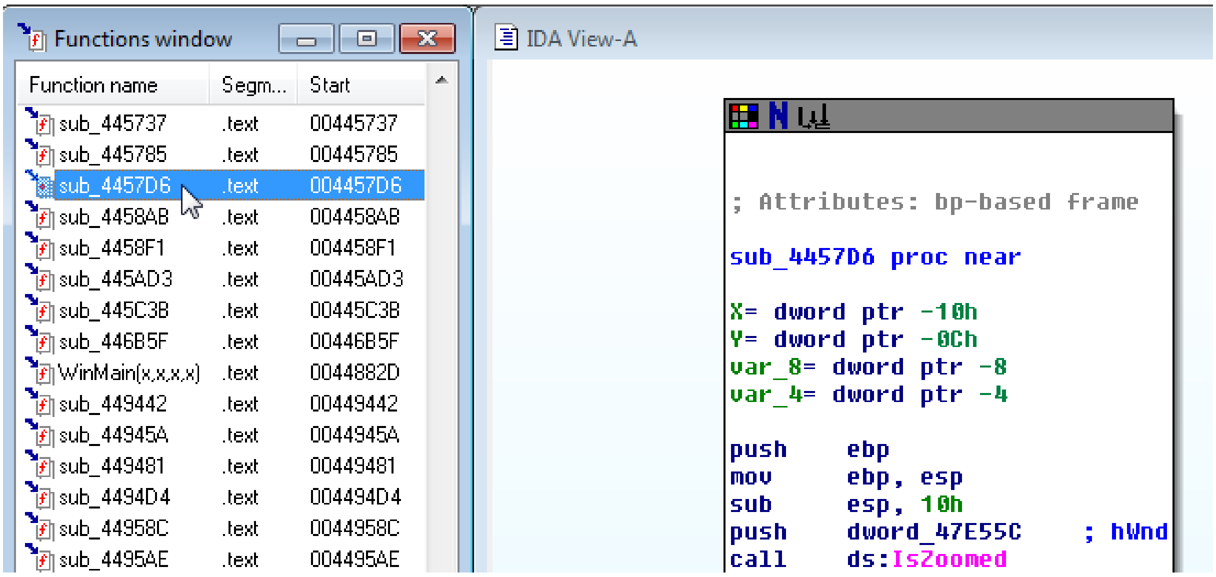
At the end, it is worth mentioning that if IDA doesn't recognize part of the code as a proper function, graph view will be unavailable. You can recognize this situation when instruction addresses in text view are red and it is impossible to switch to graph view. You will see how to deal with this situation later.

```
.text:00430E95                 jnb     loc_430FF2
.text:00430E9B                 mov     [esi+ebx+20h], cl
.text:00430E9F                 inc     dword ptr [eax]
.text:00430EA1                 jmp     short loc_430E75
.text:00430EA3 ; ---------------------------------------------------------------------------
.text:00430EA3
.text:00430EA3 loc_430EA3:                             ; CODE XREF: .text:00430E79↑j
.text:00430EA3                 mov     [ebx+10h], edx
.text:00430EA6                 jmp     loc_431012
.text:00430EAB ; ---------------------------------------------------------------------------
.text:00430EAB
.text:00430EAB loc_430EAB:                             ; CODE XREF: .text:00430E89↑j
.text:00430EAB                 cmp     dword ptr [ebx+4030h], 2Eh
.text:00430EB2                 jb      loc_430FEB
.text:00430EB8                 push    2Eh
```

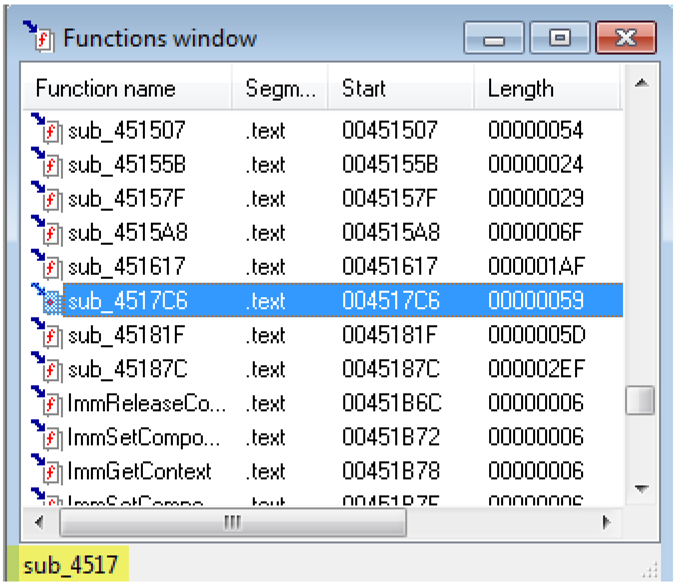
2.5 Basic navigation

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.

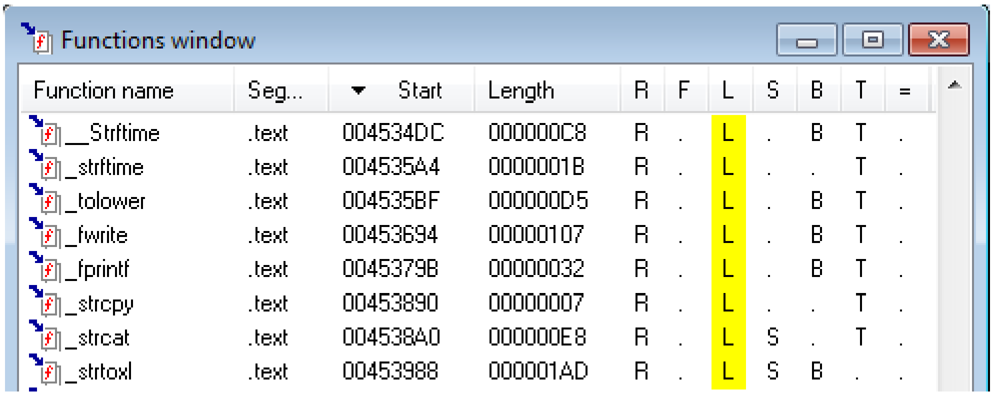


Moreover, if the functions list is long you can click the functions window and start typing a function name. At the bottom of the window, you can observe the characters you have typed and if a function with a given name exists, it will be selected automatically.

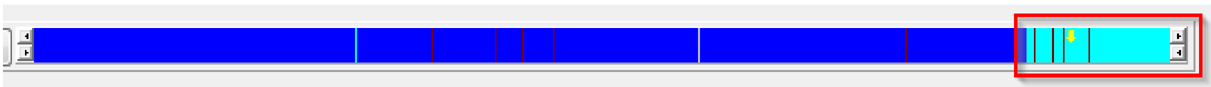


As you may have noticed, some of the functions in the functions list are named differently than *sub_XXXXXX*. Examples of such functions are *_fwrite, _strcat, _sscanf, etc.* With a few exceptions those are library functions statically linked to the binary during compilation.

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 *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)
[7] IDA F.L.I.R.T. Technology: In-Depth https://www.hex-rays.com/products/ida/tech/flirt/in_depth.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.

```
0044882D push    ebp
0044882E lea     ebp, [esp-68h]
00448832 sub     esp, 84h
00448838 mov     eax, [ebp+68h+dwMilliseconds]
0044883B push    ebx
0044883C xor     ebx, ebx
0044883E push    esi
0044883F mov     hInstance, eax
00448844 mov     dword_47E55C, ebx
0044884A mov     dword_47E558, 5
00448854 call    sub_44B2C5
00448859 call    ds:InitCommonControls
0044885F call    sub_441535
00448864 call    sub_44AE44
00448869 test    eax, eax
0044886B jnz     short loc_44888B
```

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.

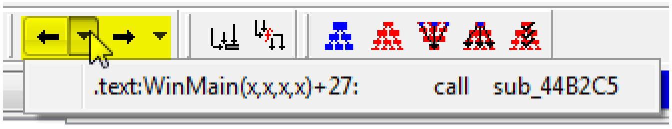
In the above example, we see three calls to local functions (*sub_44B2C5, sub_441535, sub_44AE44*) and one call to WinAPI function *InitCommonControls*. To quickly navigate to *sub_44B2C5*, double click its name.

```
0044B2C5
0044B2C5
0044B2C5
0044B2C5 sub_44B2C5 proc near
0044B2C5 push    ebx                ; char
0044B2C6 push    offset aWs2_32_dll ; "ws2_32.dll"
0044B2CB call    sub_44AE6D
0044B2D0 xor     ebx, ebx
0044B2D2 cmp     eax, ebx
0044B2D4 pop     ecx
0044B2D5 mov     hModule, eax
0044B2DA mov     dword_47E0B8, eax
0044B2DF jnz     short loc_44B305
```

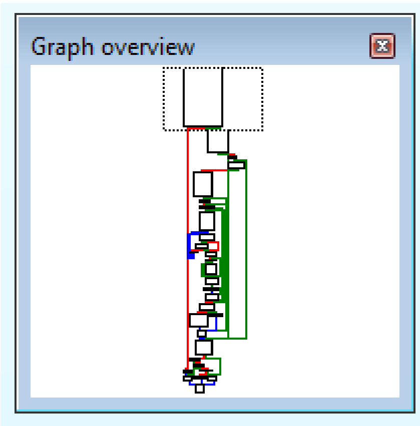
In a similar way, you can also click on data offsets to move to the location of the data in memory. For example, double click on *aWs2_32_dll,* a name given by IDA to the string *"ws2_32.dll"* defined in memory in section *.rdata* at the address *0x473EF0*.

```
.rdata:00473EE3                 align 4
.rdata:00473EE4 aWsock32_dll    db 'wsock32.dll',0      ; DATA XREF: sub_44B2C5+1C↑o
.rdata:00473EF0 aWs2_32_dll     db 'ws2_32.dll',0       ; DATA XREF: sub_44B2C5+1↑o
.rdata:00473EFB                 align 4
```

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:

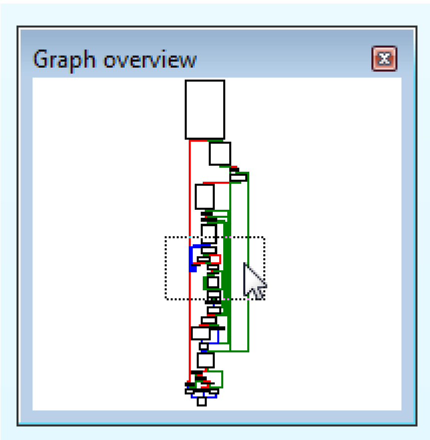


When dealing with large and complicated functions, it is useful to use the small *Graph overview* window to navigate within the code of a function. The *Graph overview* window should be present whenever disassembly view is active and its current mode is graph view. If you accidentally close *Graph overview*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

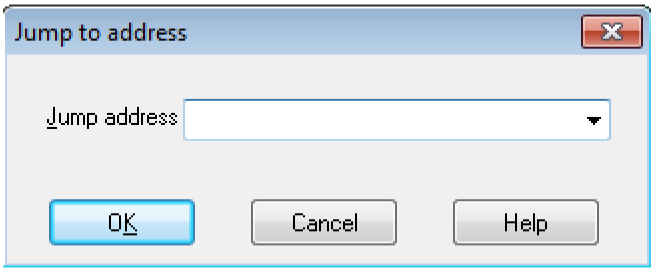


If the function graph is too big to fit your current disassembly view size, your current position will be marked with a small dotted rectangle within the *Graph overview* window. This rectangle will change size whenever you zoom in or out of the function graph.

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

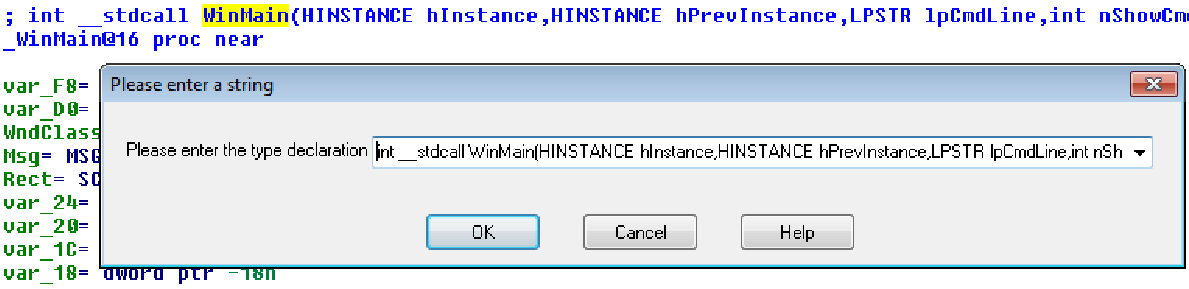
When loading a new binary sample, IDA performs an extensive auto analysis. During this process, IDA tries to find all the functions defined in assembly code as well as determine their arguments, variables or calling convention. Each detected function, whether it is a normal function or a library function,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

The *WinMain* function provides a good example of IDA's analysis capabilities:

```
; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame fpd=68h     4

; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
_WinMain@16     proc near               ; CODE XREF: start+186↓p
                                                                  1
var_F8          = dword ptr -0F8h
var_D0          = dword ptr -0D0h
WndClass        = WNDCLASSA ptr -84h
Msg             = MSG ptr -5Ch
Rect            = SCROLLINFO ptr -40h
var_24          = dword ptr -24h
var_20          = dword ptr -20h          2
var_1C          = dword ptr -1Ch
var_18          = dword ptr -18h
var_14          = dword ptr -14h    5
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
nHeight         = dword ptr -4
dwMilliseconds  = dword ptr  8
dwExStyle       = dword ptr  0Ch          3
nCount          = dword ptr  10h
nCmdShow        = dword ptr  14h

                push    ebp
                lea     ebp, [esp+var_D0]
                sub     esp, 132
                mov     eax, [ebp+68h+dwMilliseconds]
                push    ebx
```

Each function begins with a function prototype header (1). In this example, IDA recognized the function prototype, function calling convention (*stdcall*) and arguments types (*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 *Edit->Functions->Set function type…* from the menu.



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 *GlobalAlloc()*[8], IDA might name it "*hMem*". If IDA is unsuccessful with naming variables, it will give them ordinary names such as arg_0, arg_4, etc., for arguments and 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.

arg_8	ebp+10
arg_4	ebp+C
arg_0	ebp+8
ret. addr.	ebp+4
ebp	ebp
var_4	ebp-4
var_8	ebp-8
var_C	ebp-C

Additionally, if you double click on any of the variable names, IDA will open a *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.

[8] Allocates specific number of bytes from the process heap and returns handle to the allocated memory object.

```
Stack frame
-0000008A                    db ? ; undefined
-00000089                    db ? ; undefined
-00000088                    db ? ; undefined
-00000087                    db ? ; undefined
-00000086                    db ? ; undefined
-00000085                    db ? ; undefined
-00000084 WndClass           WNDCLASSA ?
-0000005C Msg                MSG ?
-00000040 var_40             SCROLLINFO ?
-00000024 var_24             dd ?
-00000020 var_20             dd ?
-0000001C var_1C             dd ?
-00000018 var_18             dd ?
-00000014 var_14             dd ?
-00000010 var_10             dd ?
-0000000C var_C              dd ?
-00000008 var_8              dd ?
-00000004 nHeight            dd ?
+00000000  s                 db 4 dup(?)
+00000004  r                 db 4 dup(?)
+00000008 dwMilliseconds     dd ?
+0000000C dwExStyle          dd ?
+00000010 nCount             dd ?
+00000014 nCmdShow           dd ?
+00000018
+00000018 ; end of stack variables
SP++00000070
```

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[9]. 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_4= dword ptr -4
0042FCAD arg_0= dword ptr  8
0042FCAD
0042FCAD push    ebp
0042FCAE mov     ebp, esp
0042FCB0 push    ecx
0042FCB1 push    ebx
0042FCB2 lea     eax, [ebp+var_4]
0042FCB5 push    eax
0042FCB6 mov     eax, [ebp+arg_0]
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*.

[9] All About EBP http://practicalmalwareanalysis.com/2012/04/03/all-about-ebp/ (last accessed 11.09.2015)

```
004048BB push    5
004048BD push    [esp+60h+var_C]
004048C1 call    sub_408227
004048C6 add     esp, 48h
004048C9 cmp     [esp+1Ch+arg_4], ebp
004048CD mov     ebx, offset aApply ; "Apply"
004048D2 jnz     short loc_4048D9
```

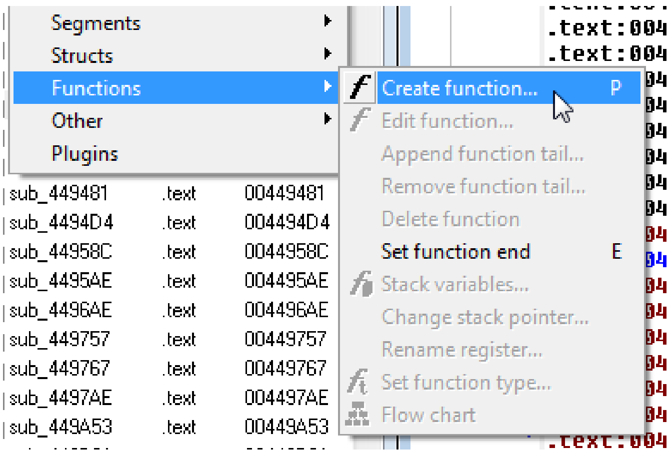
```
-0000000C
-0000000C var_C          dd ?
-00000008 var_8          dd ?
-00000004 var_4          dd ?
+00000000  r             db 4 dup(?)
+00000004 arg_0          dd ?
+00000008 arg_4          dd ?
+0000000C arg_8          dd ?
+00000010 arg_C          dd ?
+00000014
+00000014 ; end of stack variables
```

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*:

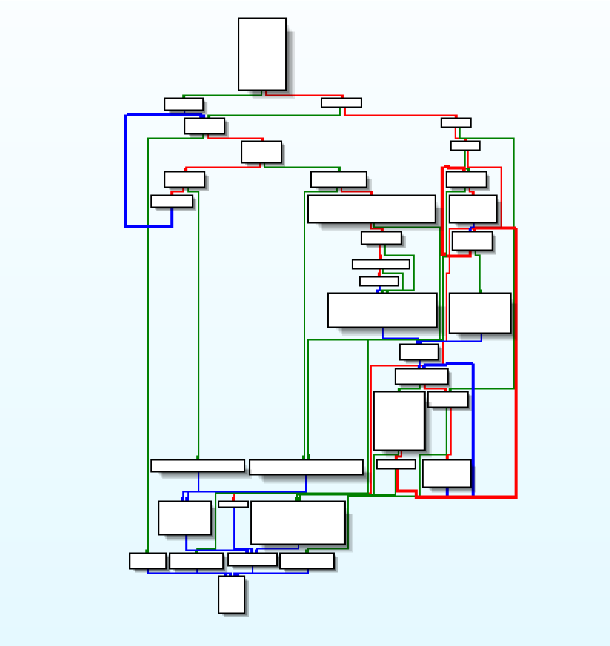
```
.text:00430E36                  leave
.text:00430E37                  retn
.text:00430E37 sub_43043C       endp
.text:00430E37
.text:00430E38 ; ---------------------------------------------
.text:00430E38                  push    ebp
.text:00430E39                  mov     ebp, esp
.text:00430E3B                  push    ebx
.text:00430E3C                  mov     ebx, [ebp+8]
.text:00430E3F                  mov     eax, [ebx+10h]
.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…*



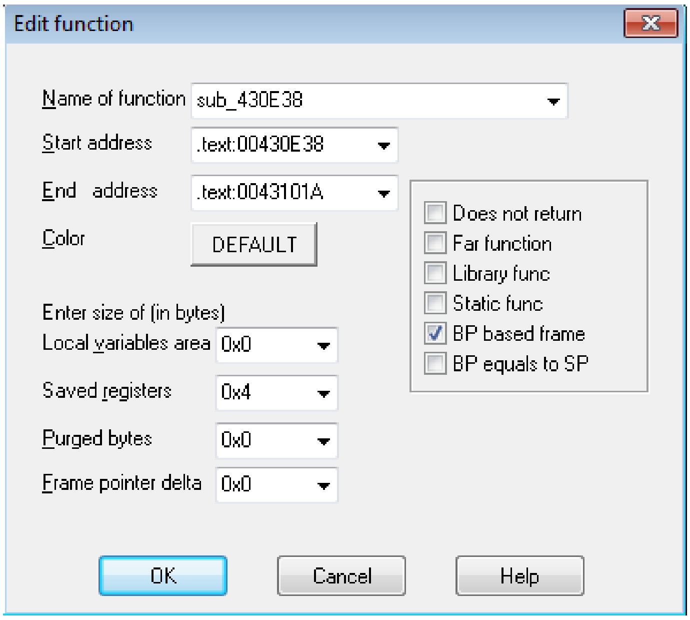
IDA should now recognize this part of the code as a proper function and you should be able to switch to the graph view.

```
.text:00430E38 ; Attributes: bp-based frame
.text:00430E38
.text:00430E38 sub_430E38      proc near
.text:00430E38
.text:00430E38 arg_0           = dword ptr  8
.text:00430E38 arg_8           = dword ptr  10h
.text:00430E38 arg_C           = dword ptr  14h
.text:00430E38
.text:00430E38                 push    ebp
.text:00430E39                 mov     ebp, esp
.text:00430E3B                 push    ebx
.text:00430E3C                 mov     ebx, [ebp+arg_0]
.text:00430E3F                 mov     eax, [ebx+10h]
```



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.

```
0044D26D push    edi
0044D26E push    107h             ; uSize
0044D273 lea     eax, [ebp+FileName]
0044D279 push    eax              ; lpBuffer
0044D27A call    ds:GetWindowsDirectoryA
0044D280 lea     eax, [ebp+FileName]
0044D286 push    offset asc_474704 ; "\\*"
0044D28B push    eax              ; char *

0044D2C1 lea     eax, [ebp+FindFileData]
0044D2C7 push    eax              ; lpFindFileData
0044D2C8 push    esi              ; hFindFile
0044D2C9 call    ds:FindNextFileA
0044D2CF test    eax, eax
```

There are also some unknown calls to an address stored in registers:

```
0044D384 push    0F0000000h
0044D389 push    1                constants?
0044D38B push    edi
0044D38C push    edi
0044D38D lea     ecx, [ebp+var_4]
0044D390 push    ecx
0044D391 call    eax              ???
0044D393 test    eax, eax
```

And calls to functions pointed by some global variable:

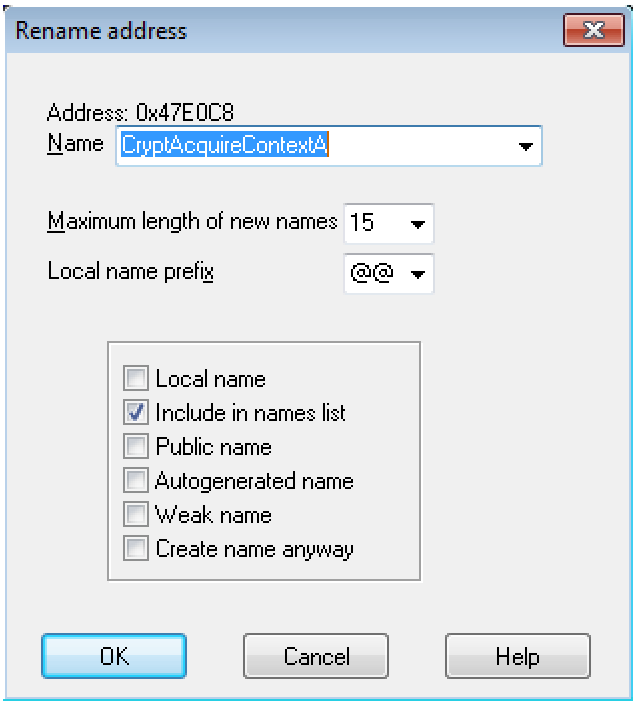
```
0044D397 lea     eax, [ebp+var_28]
0044D39A push    eax
0044D39B push    20h
0044D39D push    [ebp+var_4]  ???
0044D3A0 call    dword_47E0C4
0044D3A6 test    eax, eax
0044D3A8 jz      short loc_44D3B4
```

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 *GetProcAddr*.

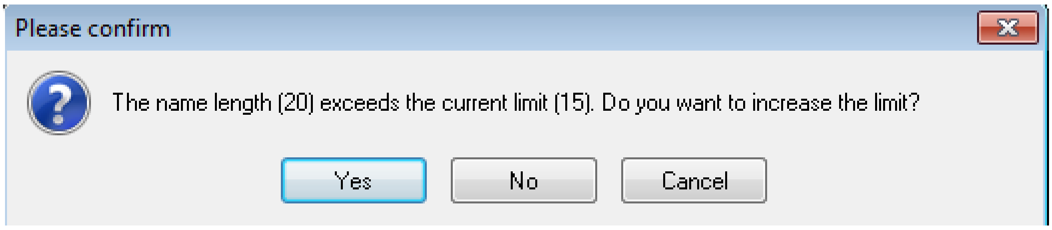
```
0044D311 push    offset aCryptacquireco ; "CryptAcquireContextA"
0044D316 push    eax              ; hModule
0044D317 call    esi ; GetProcAddress
0044D319 mov     dword_47E0C8, eax       saving result
0044D31E mov     eax, dword_47E0D0
0044D323 jmp     short loc_44D32B
```

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 aCryptacquireco ; "CryptAcquireContextA"
0044D316 push    eax             ; hModule
0044D317 call    esi ; GetProcAddress
0044D319 mov     CryptAcquireContextA, eax
0044D31E mov     eax, dword_47E0D0
0044D323 jmp     short loc_44D32B
```

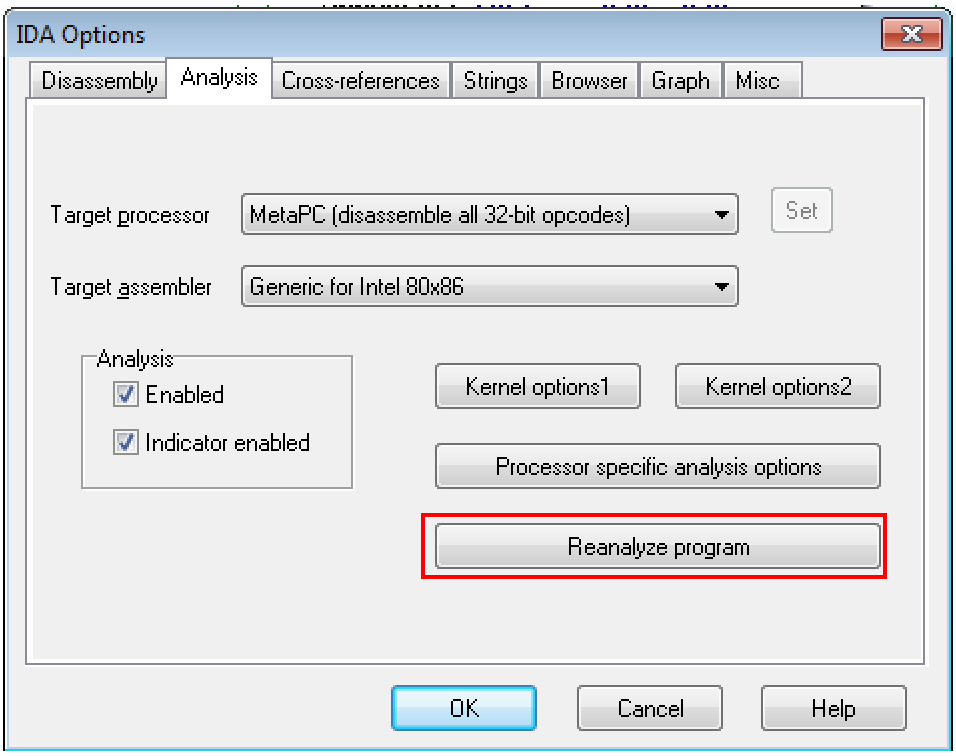
Repeat this step for the remaining two calls to *GetProcAddress* in *sub_44D262* (*CryptGenRandom, CryptReleaseContext*). Make sure that you rename the memory locations exactly the same as the names of the resolved functions.

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?

```
0044D397 lea     eax, [ebp+var_28]
0044D39A push    eax
0044D39B push    20h
0044D39D push    [ebp+var_4]
0044D3A0 call    CryptGenRandom
0044D3A6 test    eax, eax
0044D3A8 jz      short loc_44D3B4
```

```
0044D3B4
0044D3B4 loc_44D3B4:
0044D3B4 push    edi
0044D3B5 push    [ebp+var_4]
0044D3B8 call    CryptReleaseContext
```

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!

```
0044D397 lea     eax, [ebp+var_28]
0044D39A push    eax
0044D39B push    20h
0044D39D push    [ebp+var_4]
0044D3A0 call    CryptGenRandom
0044D3A6 test    eax, eax
0044D3A8 jz      short loc_44D3B4
```

```
0044D397 lea     eax, [ebp+pbBuffer]
0044D39A push    eax              ; pbBuffer
0044D39B push    20h              ; dwLen
0044D39D push    [ebp+hProv]      ; hProv
0044D3A0 call    CryptGenRandom
0044D3A6 test    eax, eax
0044D3A8 jz      short loc_44D3B4
```

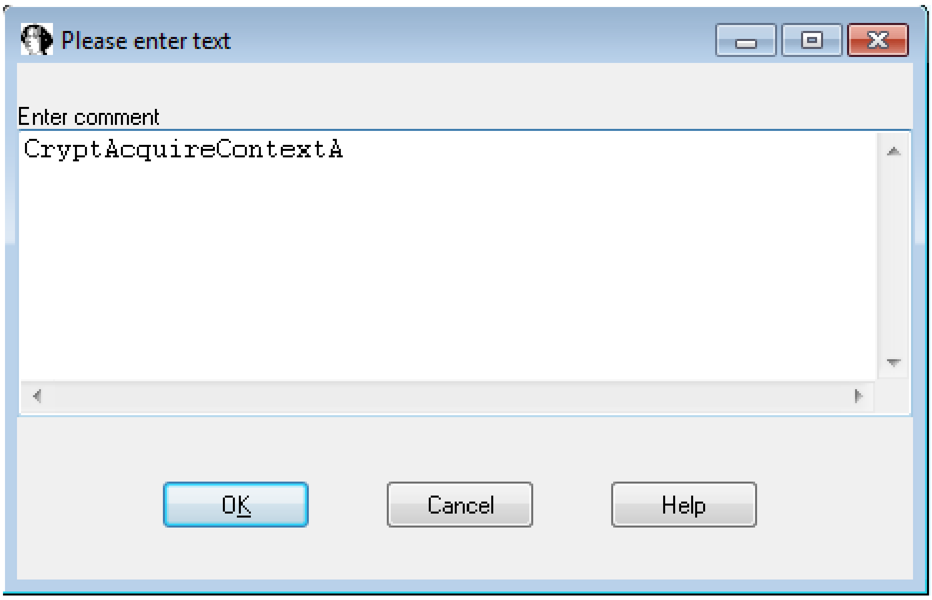
Now scroll to the location *0x44D391* where there is a call to *eax*:

```
0044D384 push    0F0000000h
0044D389 push    1
0044D38B push    edi
0044D38C push    edi
0044D38D lea     ecx, [ebp+hProv]
0044D390 push    ecx
0044D391 call    eax
0044D393 test    eax, eax
0044D395 jz      short loc_44D3BE
```

IDA still doesn’t know where this call is made to, 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*.

```
0044D36B
0044D36B loc_44D36B:
0044D36B mov     eax, CryptAcquireContextA
0044D370 cmp     eax, edi
0044D372 jz      short loc_44D3BE
```

It is good to comment this finding. To add comment click on *call eax* and pres <:> (colon):

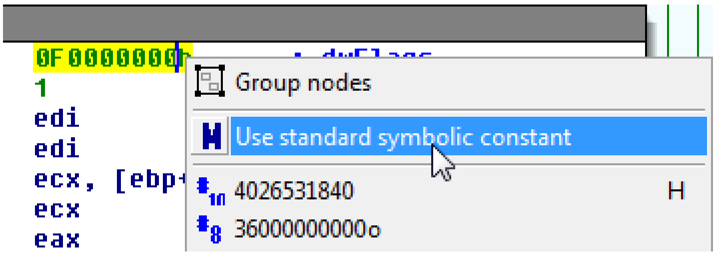


Comment remaining arguments of *CryptAcquireContextA* accordingly to this function prototype[10] to make it look like the following:

```
0044D384 push    0F0000000h       ; dwFlags
0044D389 push    1                ; dwProvType
0044D38B push    edi              ; pszProvider
0044D38C push    edi              ; pszContainer
0044D38D lea     ecx, [ebp+hProv]
0044D390 push    ecx              ; phProv
0044D391 call    eax              ; CryptAcquireContextA
0044D393 test    eax, eax
0044D395 jz      short loc_44D3BE
```

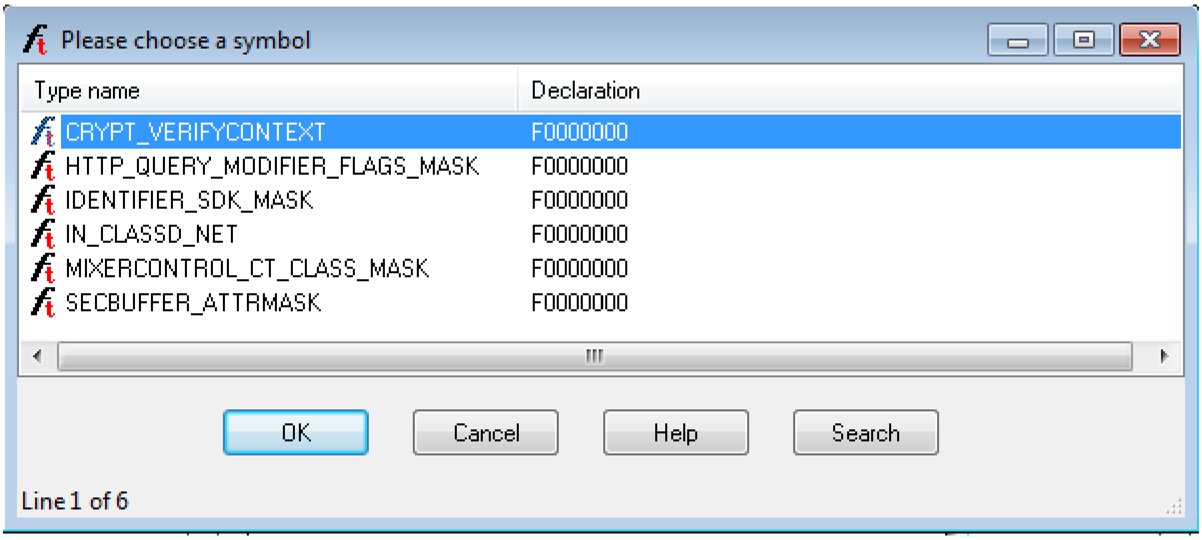
Now you know that *0F0000000h* and *1* are the constants passed to *CryptAcquireContextA* in arguments dwFlags and dwProvType. You can check in function reference[11] that dwFlags takes the constant with the *CRYPT_* prefix while dwProvType takes the constant with the *PROV_* prefix. You can tell IDA to represent those values as a symbolic constant.

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



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

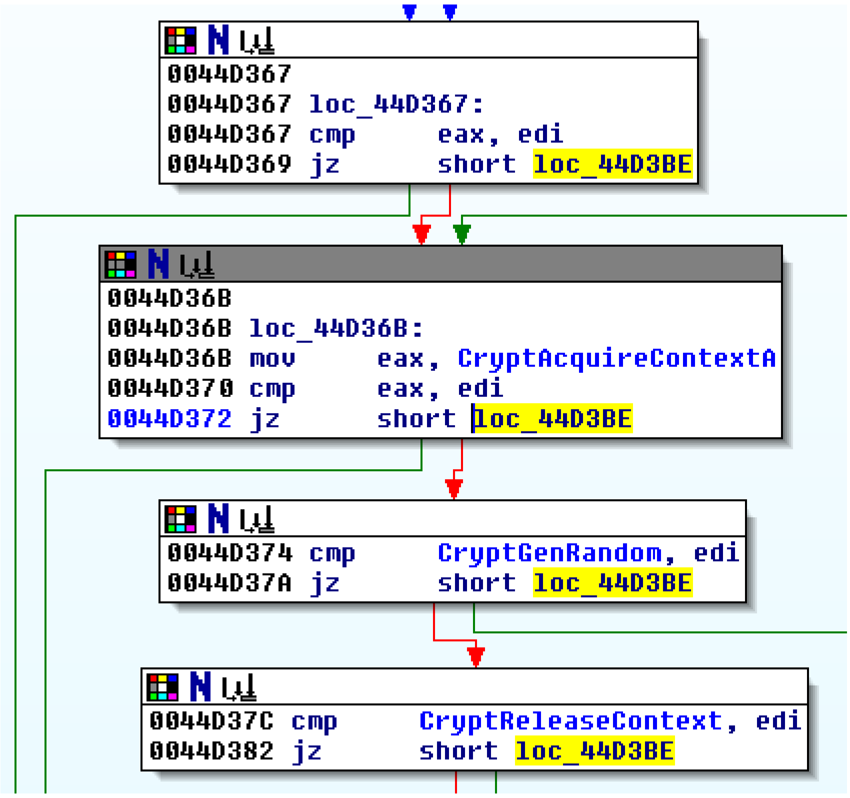
[10] CryptAcquireContext function https://msdn.microsoft.com/en-us/library/windows/desktop/aa379886%28v=vs.85%29.aspx (last accessed 11.09.2015)
[11] CryptAcquireContext function https://msdn.microsoft.com/en-us/library/windows/desktop/aa379886%28v=vs.85%29.aspx (last accessed 11.09.2015)



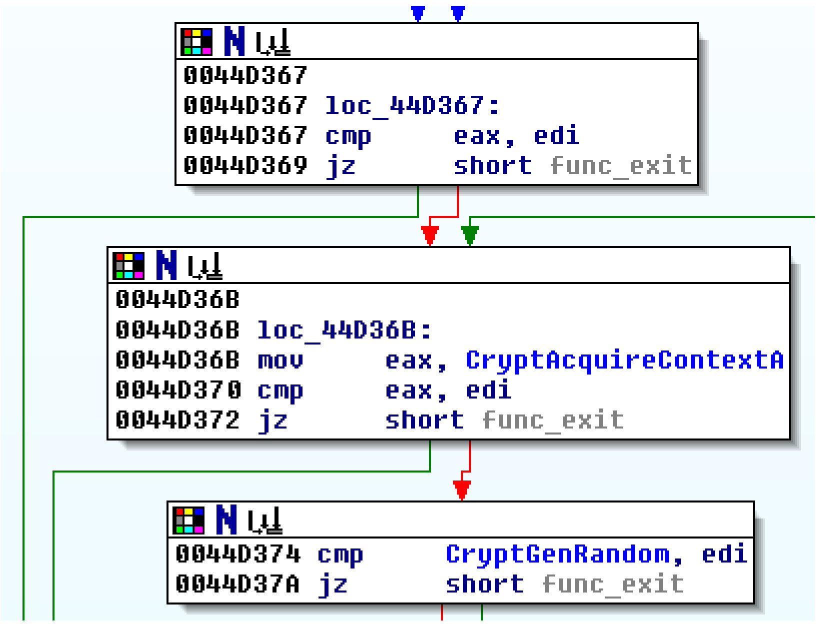
Repeat those steps for *dwProvType*, but this time choosing *PROV_RSA_FULL*. Now code should look like this:

```
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              ; phProv
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*.

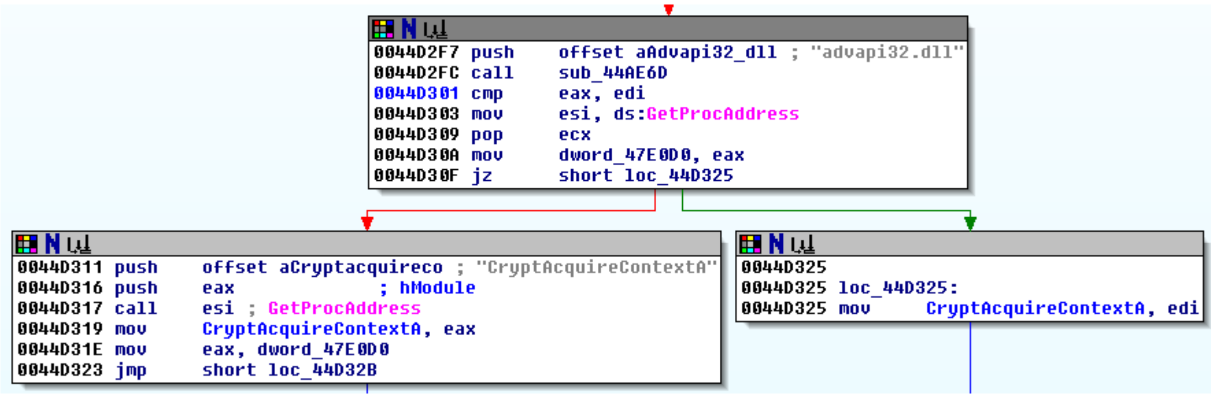


Further inspection shows that *loc_44D3BE* is a location of a function epilogue – probably jumped to if something earlier fails. Rename this location to *func_exit* in the same way as renaming memory location. Now all jumps should look much more clearly:

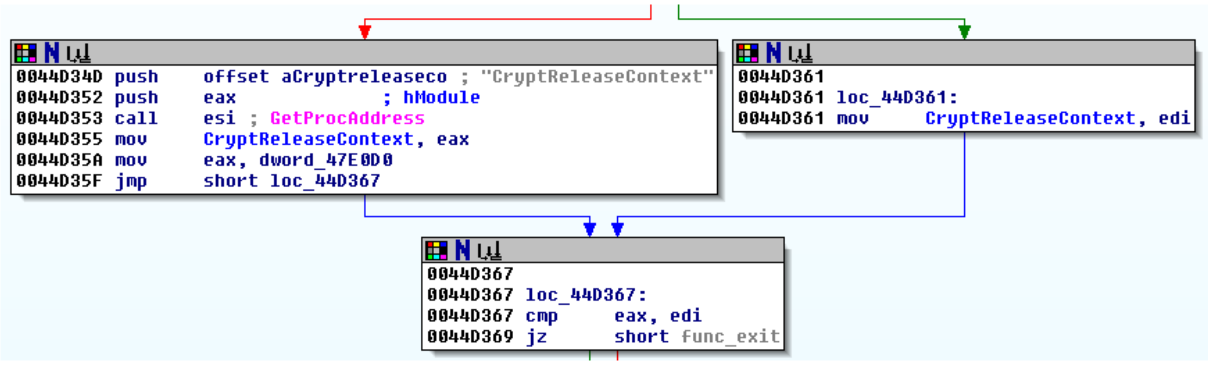


You can rename almost any name used in IDA (function names, arguments, variables, etc.) in the same way.

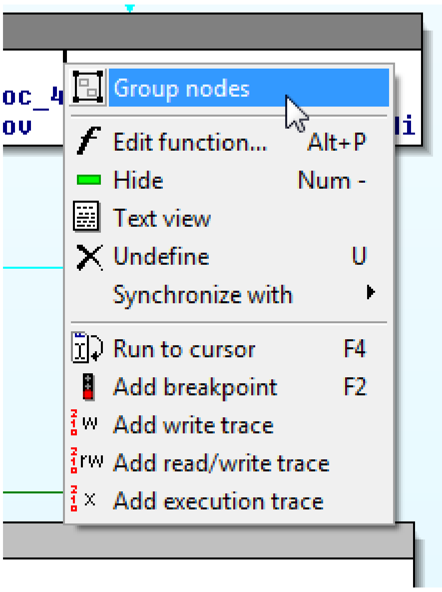
To further simplify function structure, you will now group graph nodes used to resolve crypto functions addresses. To do this, go to the graph node at the address *0x44D2F7* and select graph nodes by clicking on them while holding the <Ctrl> key.



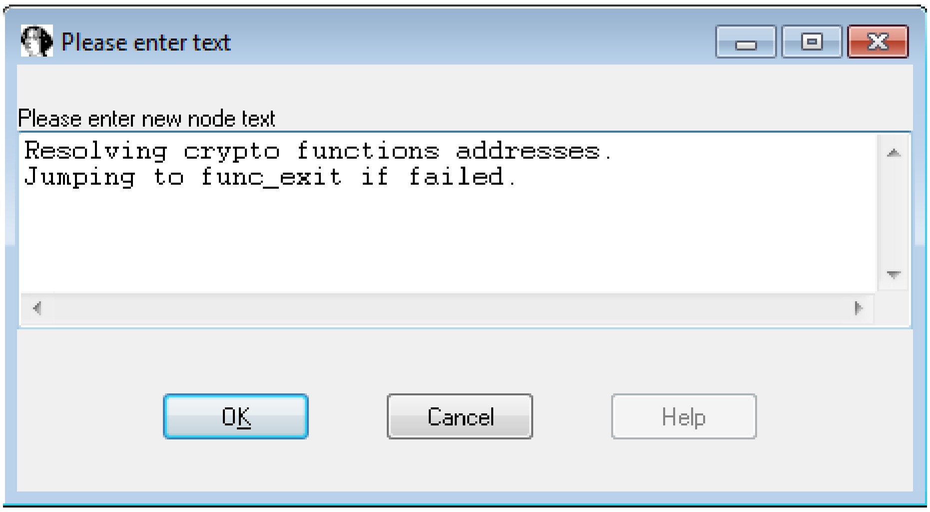
Select all graph nodes starting from *0x44D2F7* up to *0x44D367*.



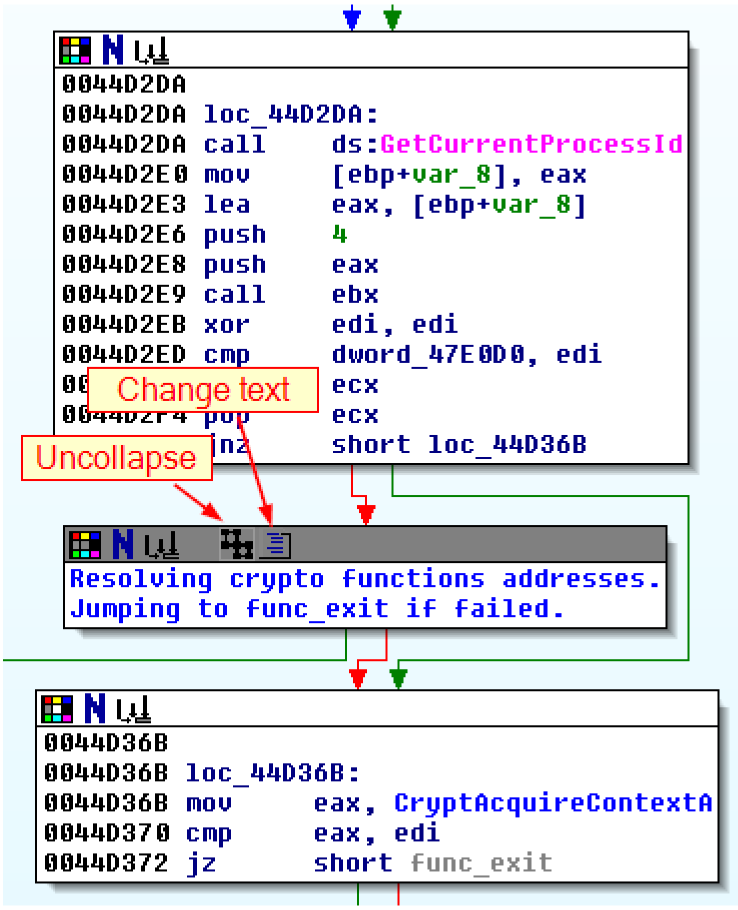
Now right-click on selected nodes and choose *Group nodes*.



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*).

```
0044D2B1
0044D2B1 loc_44D2B1:
0044D2B1 lea     eax, [ebp+FindFileData]
0044D2B7 push    140h
0044D2BC push    eax
0044D2BD call    ebx
0044D2BF pop     ecx
0044D2C0 pop     ecx
0044D2C1 lea     eax, [ebp+FindFileData]
0044D2C7 push    eax             ; lpFindFileData
0044D2C8 push    esi             ; hFindFile
0044D2C9 call    ds:FindNextFileA
0044D2CF test    eax, eax
0044D2D1 jnz     short loc_44D2B1
```

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:

```
0044D2DA call    ds:GetCurrentProcessId
0044D2E0 mov     [ebp+var_8], eax
0044D2E3 lea     eax, [ebp+var_8]
0044D2E6 push    4
0044D2E8 push    eax
0044D2E9 call    ebx
```

```
0044D3AA lea     eax, [ebp+pbBuffer]
0044D3AD push    20h
0044D3AF push    eax
0044D3B0 call    ebx
```

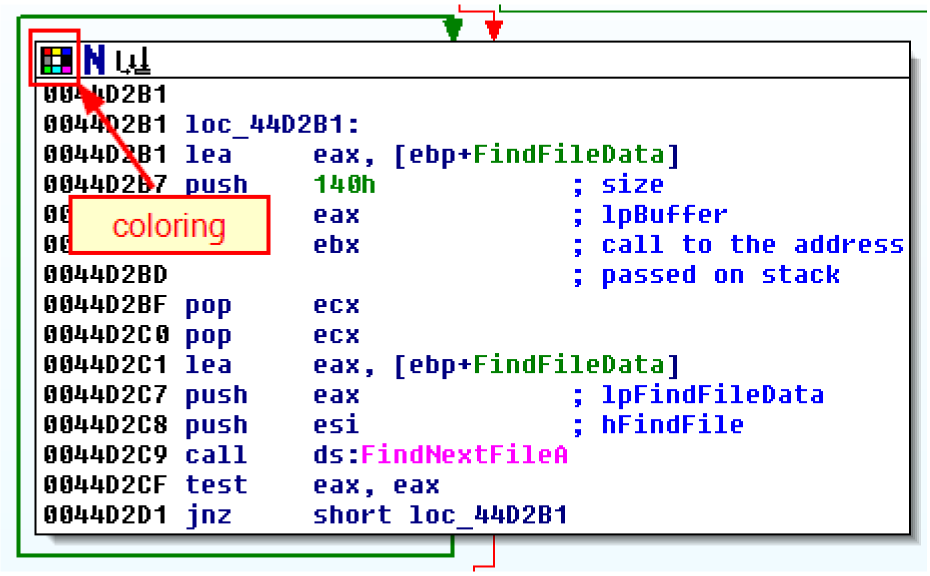
In each case, two arguments are pushed onto the stack – first some address, and the second one seems to be the size of a buffer pointed by the first argument (it is good to comment this!).

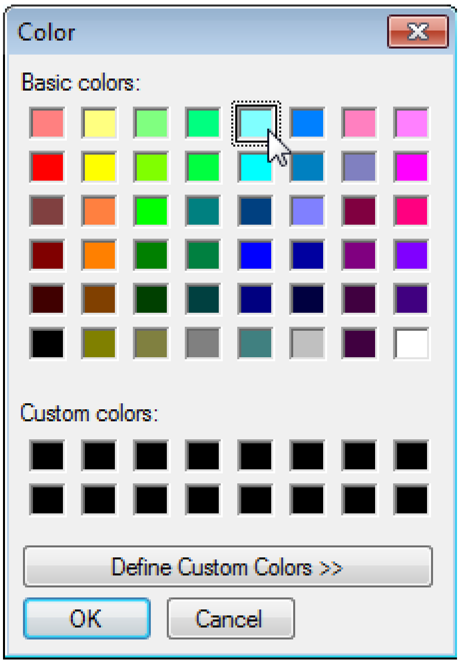
Now if you select only the *ebx* register you will notice that its value is being assigned once at the beginning of the function:

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

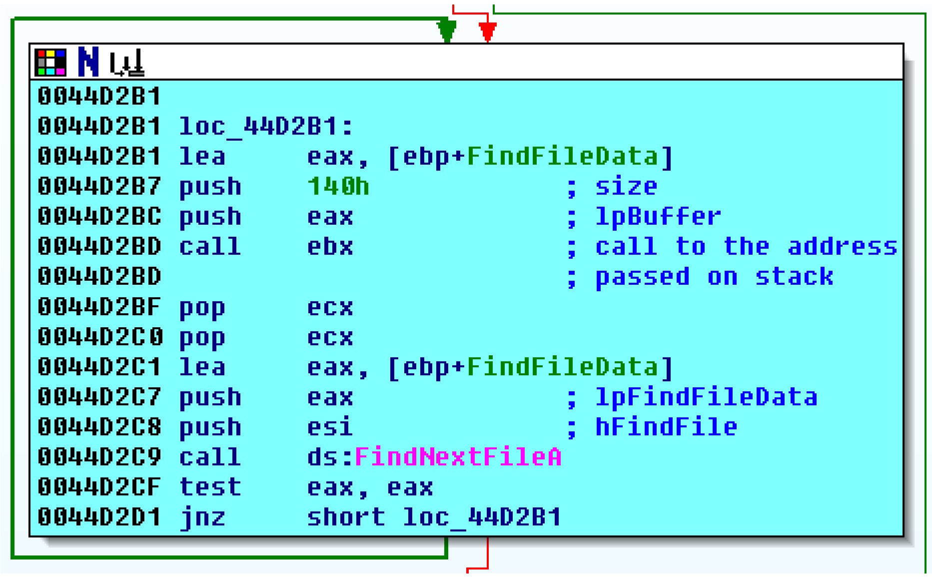
This means that arg_0 is a function pointer and the function pointed by this argument is called three times in our function (you can rename *arg_0* to *func_ptr*). Since this seems to be a significant element, it is good to mark all three graph nodes where such a call takes place.

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:





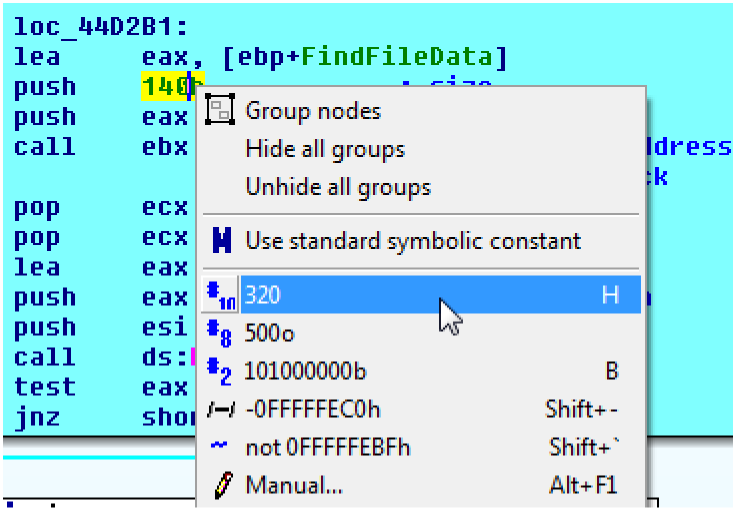
After clicking *Ok* node background should become cyan.



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:

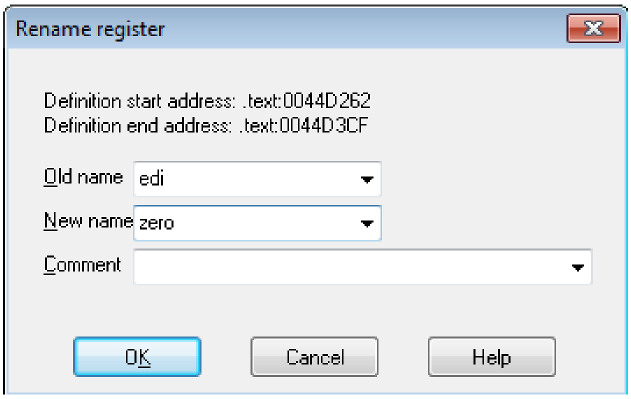
```
0044D2E8 push    eax
0044D2E9 call    ebx
0044D2EB xor     edi, edi        ; zeroing edi
0044D2ED cmp     dword_47E0D0, edi
0044D2F3 pop     ecx
```

```
0044D374 cmp     CryptGenRandom, edi
0044D37A jz      short func_exit
```

```
0044D37C cmp     CryptReleaseContext, edi
0044D382 jz      short func_exit
```

```
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                ; phProv
0044D391 call    eax                ; CryptAcquireContextA
```

To rename a register, click on register and press <N> (rename):



Now the code should look like this:

```
0044D374 cmp     CryptGenRandom, zero
0044D37A jz      short func_exit
```

```
0044D37C cmp     CryptReleaseContext, zero
0044D382 jz      short func_exit
```

2.9 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?*

- *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[12] which is a banking trojan first detected by S21sec company[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.

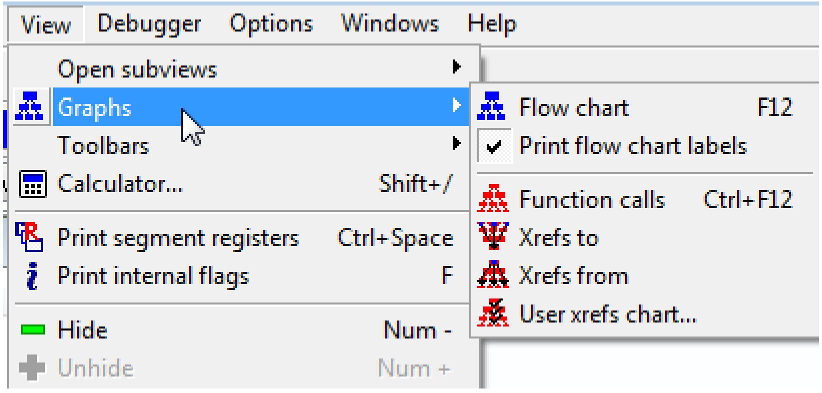
3.1 Using call graphs

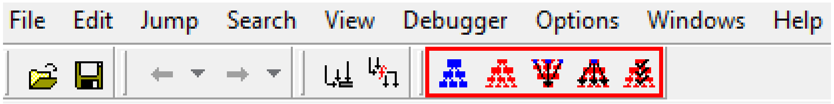
Starting the analysis of a new binary, some of the first questions that comes to mind are what is the execution flow of the code? What local functions are called by what other functions? Are there any API calls? What data variables are referenced in the code? To answer some of those questions, IDA provides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

[12] Sample
400fbcaaac9b50becbe91ea891c25d71 (MD5)
https://malwr.com/analysis/OTRiMDk1ODFkOGVjNDhkMzljYzdiZTUzZDUyYjEwM2M/ (last accessed 11.09.2015)
[13] New banking trojan 'Slave' hitting Polish Banks http://securityblog.s21sec.com/2015/03/new-banker-slave-hitting-polish-banks.html (last accessed 11.09.2015)

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





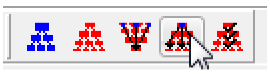
There are four basic call graph types:

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

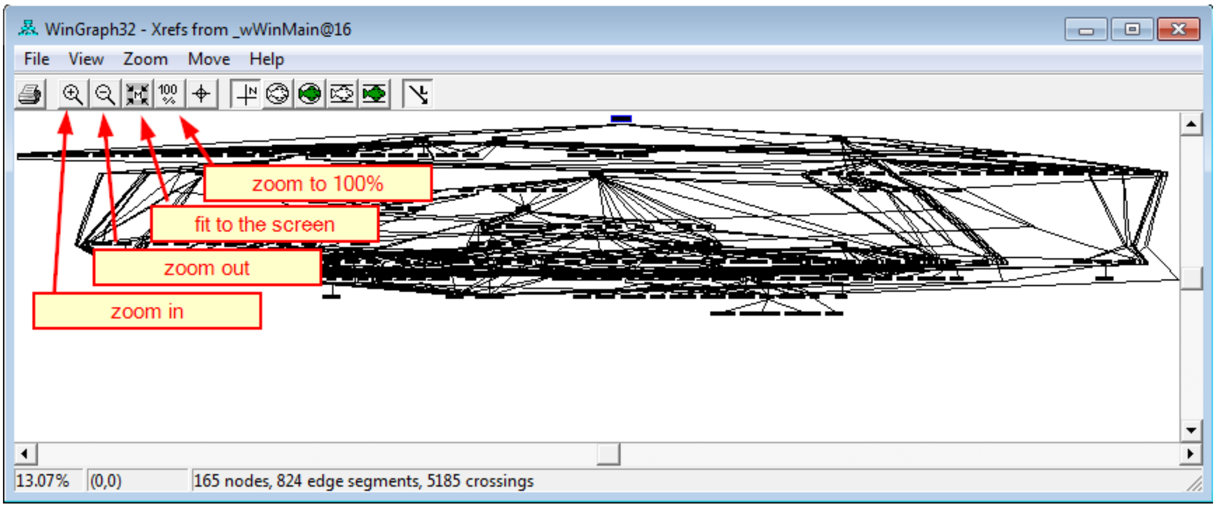
Note that creating *Xrefs to* or *Xrefs from* is possible only if, in disassembly view, the currently selected item is some function name or a named data location (*dword_XXXXXX*).

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.

```
; __stdcall wWinMain(x, x, x, x)
_wWinMain@16 proc near
push    esi
push    edi
call    sub_402860
mov     esi, ds:CreateMutexW
```



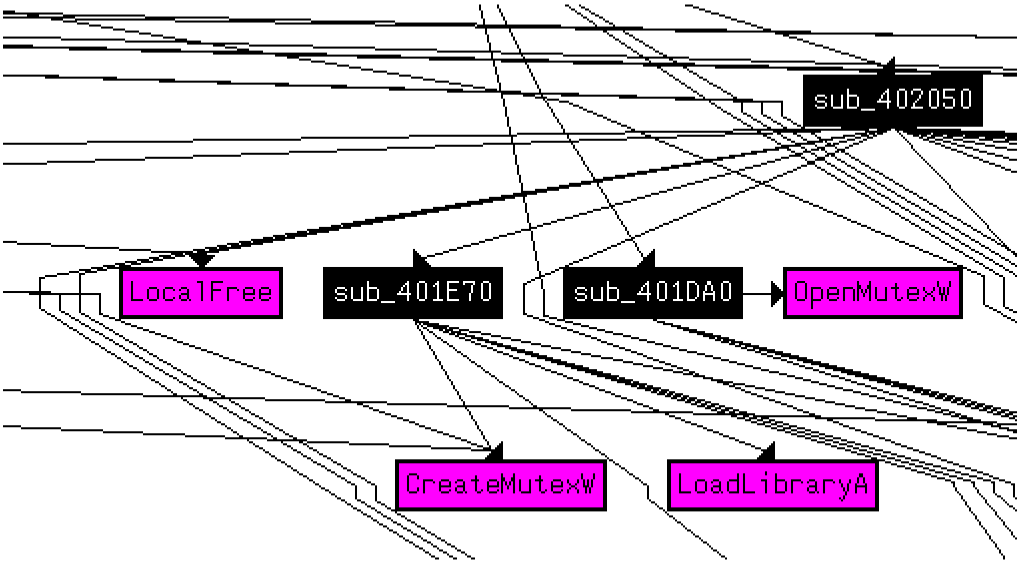
Now you should see *WinGraph32* window with newly created call graph for *wWinMain* function. This *Xrefs from* graph presents all functions called from *wWinMain* routine (local functions, library functions as well API functions).



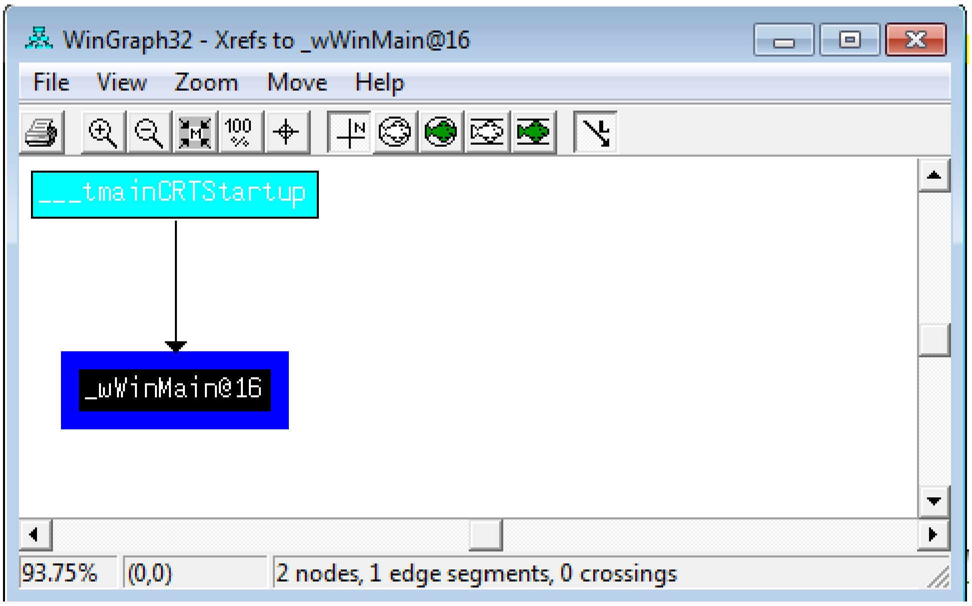
Depending on the code complexity and size of your screen such graph might be more or less readable. For more complex malware or malware using many linked libraries such graph might be barely readable.

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

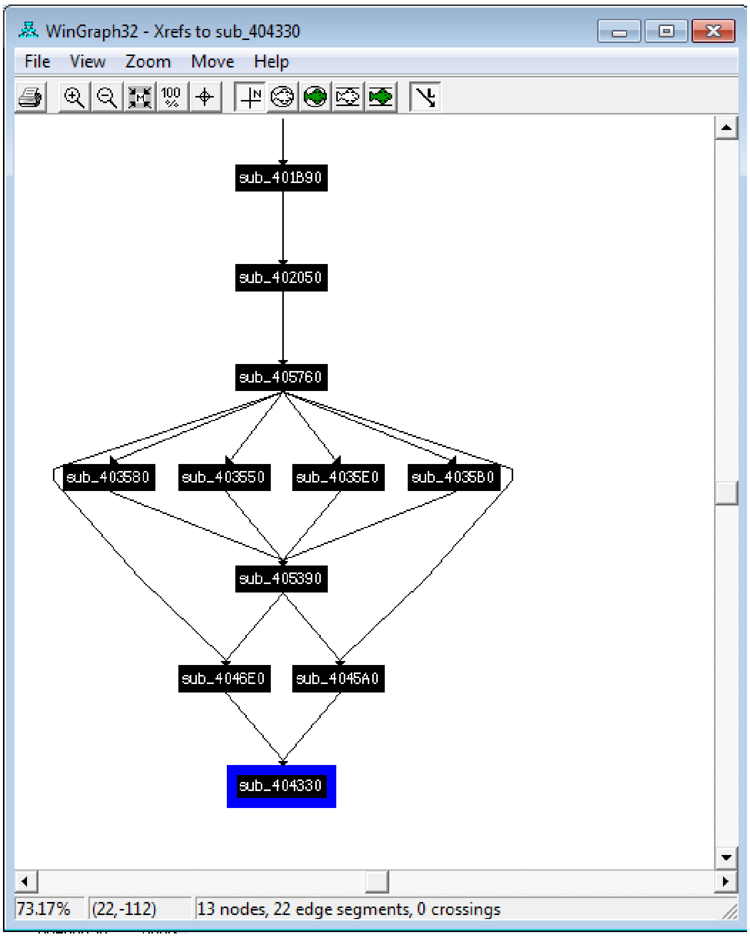
Now zoom in (or zoom to 100%) to notice the different colours of the graph nodes. Black nodes represent local functions while pink nodes represent API calls. There might be also cyan nodes and white nodes representing functions recognized by IDA as library functions and named data locations, respectively.



So far, you have been analysing what functions from the *wWinMain* were called. What if you want to check what functions call *wWinMain*? You can use the *Xrefs to* call graph. Click on *wWinMain* and choose *Xrefs to* graph.



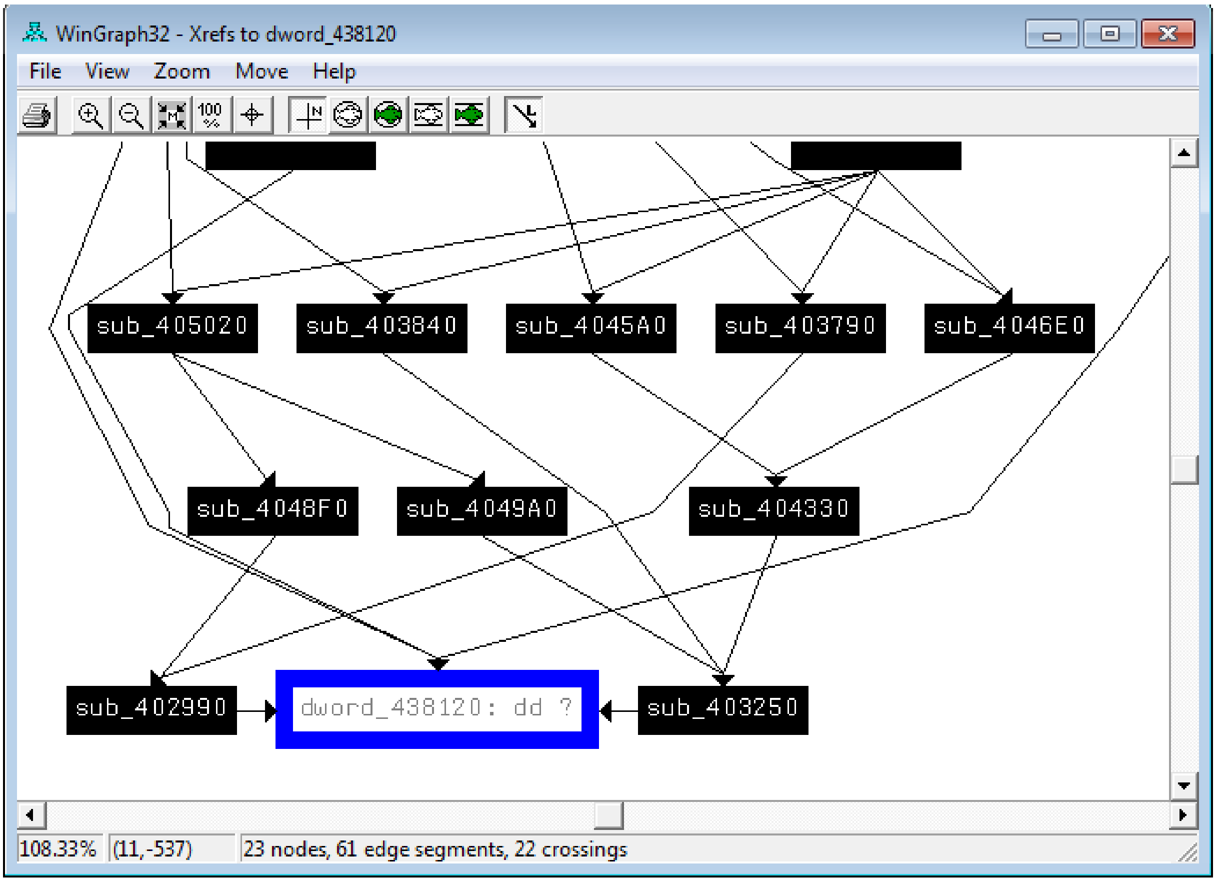
Without much of a surprise, we see that *wWinMain* was called from *___tmainCRTStartup* routine. To get a little more complex example, create *Xrefs to* graph for *sub_404330*.



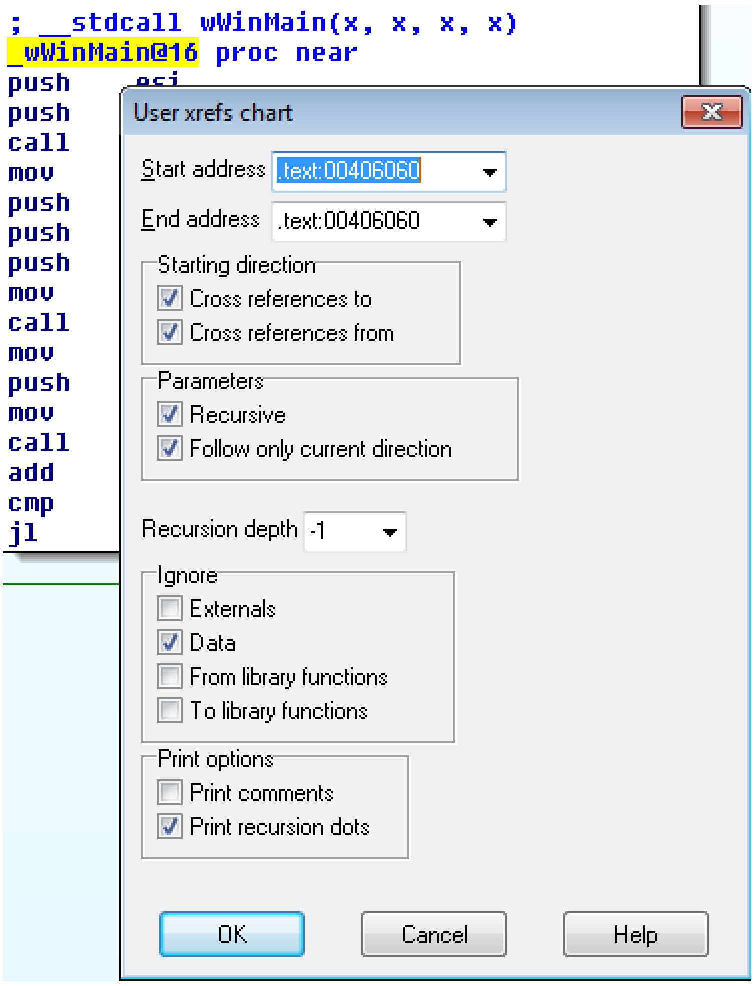
Xrefs to graphs might be also used to check what functions are referencing particular memory location. As an example go to the *wWinMain* function, click on *dword_438120* and choose to create the *Xrefs to* graph.

```
; __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     hHandle, eax
```

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.

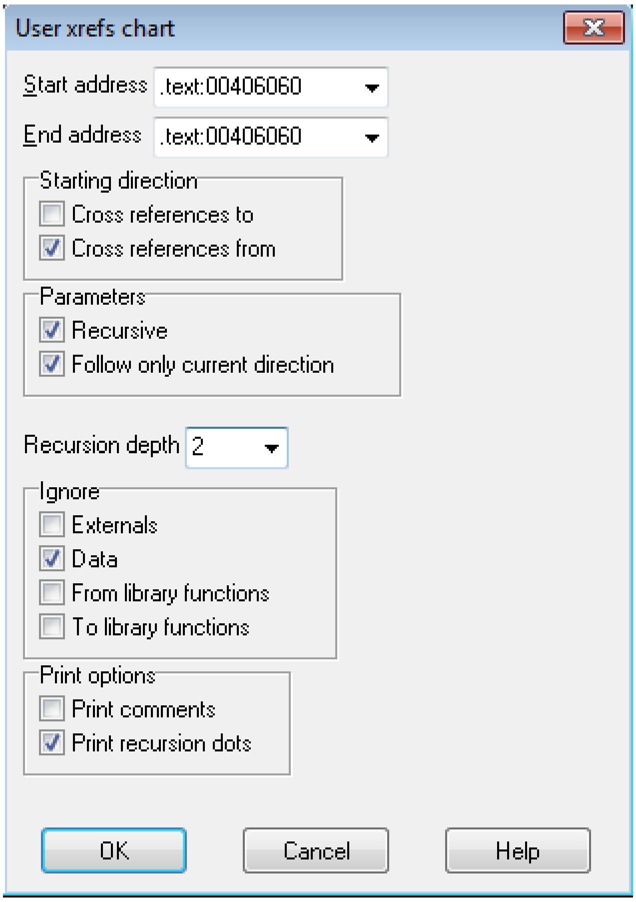


The third type of graphs are user defined graphs. In contrast to *Xrefs to* and *Xrefs from* graphs, when creating a user defined graph you can specify additional parameters for how this graph should look. To create this graph for *wWinMain* select *wWinMain* and choose *User xrefs chart…*.

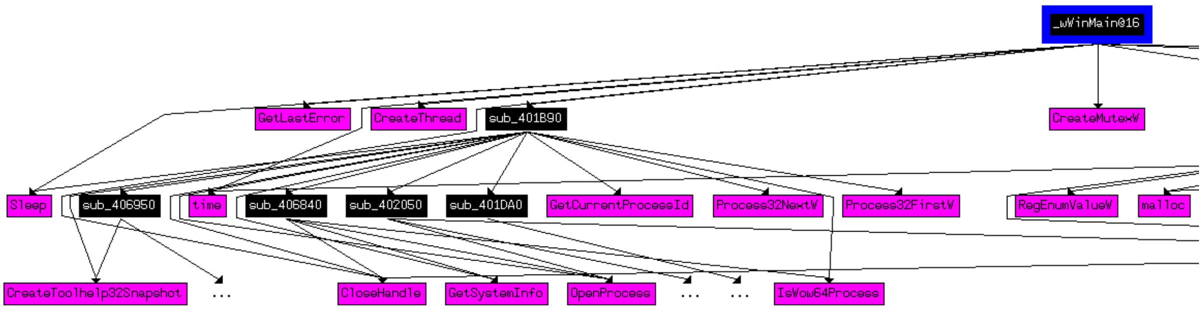


In the new window, you can specify additional graph parameters. You can hover the cursor over any parameter to get a hint what this parameter changes. The most frequently used group of parameters are *Starting direction* and *Recursion depth.* Using *Recursion depth* you can limit the number of graph 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.



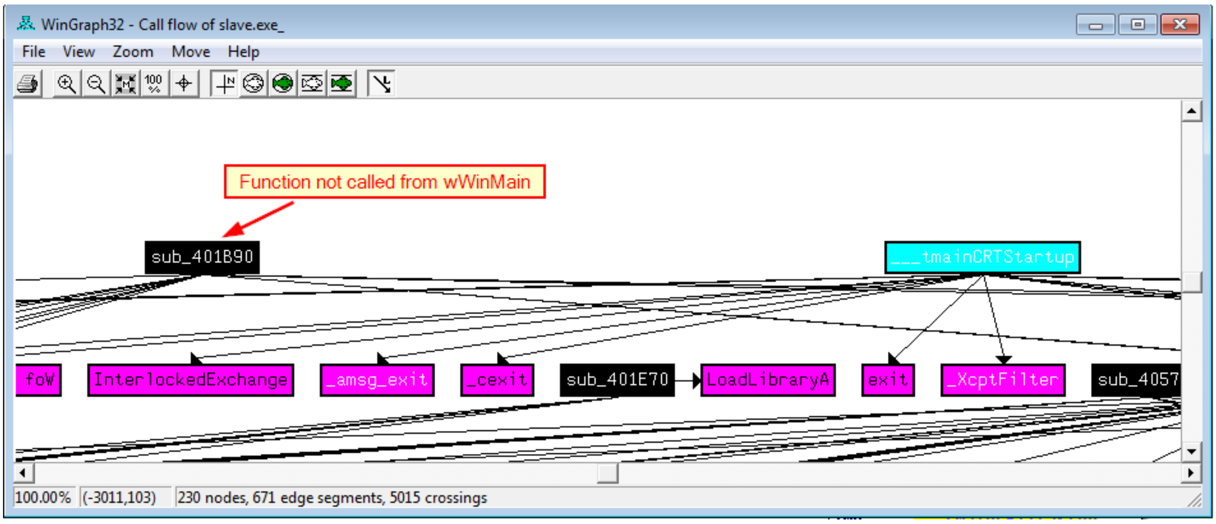
Is newly created graph clearer and easier to follow?



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.



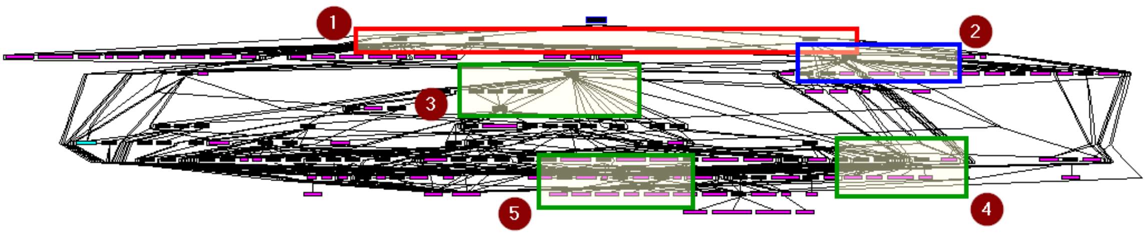
Now that you know how to create various call graphs and what they are used for, how can you recognize important function calls?

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.

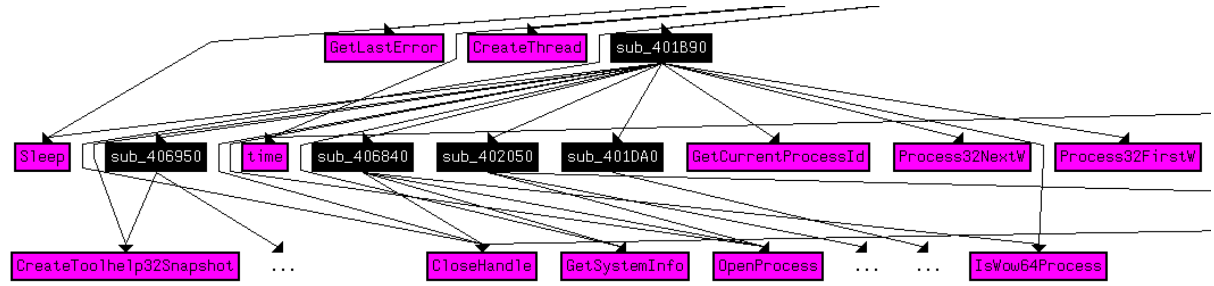
As an example you will now analyse call graph of *wWinMain* function[14].



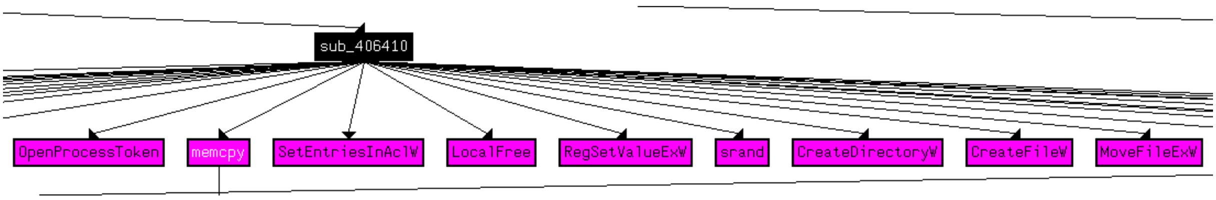
First, notice the top group of three functions (1): *sub_406410, sub_406120, sub_401B90*. At this point you can already suspect that those are important functions because they are called directly from the *wWinMain* and they are calling a lot of APIs. Unfortunately due to the structure of the graph it 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.

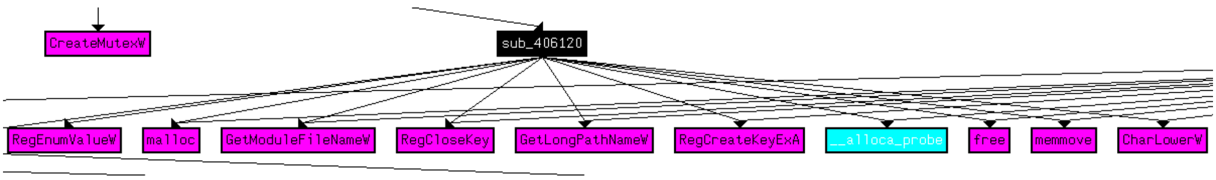
Then take a look at ***sub_401B90***. We can see that this function is iterating through the process list (calls to *Process32FirstW, Process32NextW,* etc.). This might mean that this function is looking for a specific process to inject some code into it or it is using some anti-analysis techniques (e.g. trying to detect AV processes).



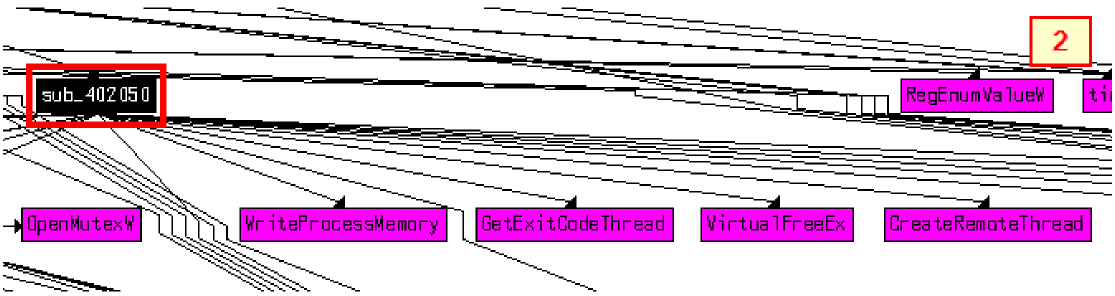
Next, look at ***sub_406410***. It calls APIs such us *RegSetValueExW, CreateDirectoryW, CreateFileW, MoveFileExW*. It likely indicates that this is an installation procedure. You should inspect it if you want to know how the malware installs itself in the system.



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

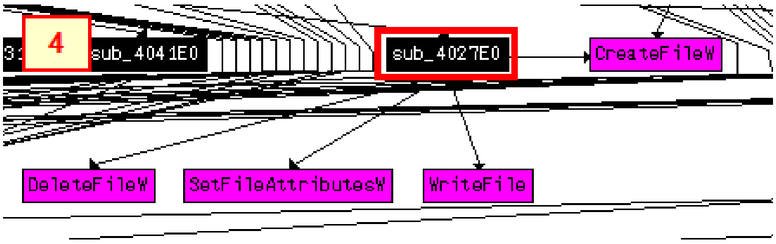


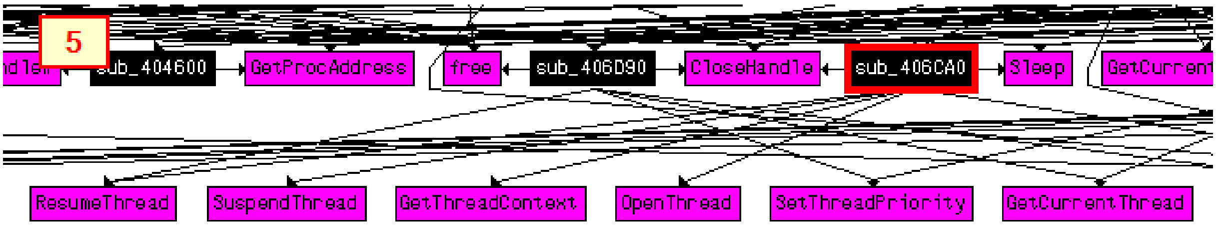
Now go back to the general graph (*wWinMain*) and take a look at function ***sub_402050*** (2). Among the other APIs it is also calling *CreateRemoteThread* and *WriteProcessMemory*. This tells us that this function is most likely injecting some code to other processes (you can also notice that *sub_402050*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

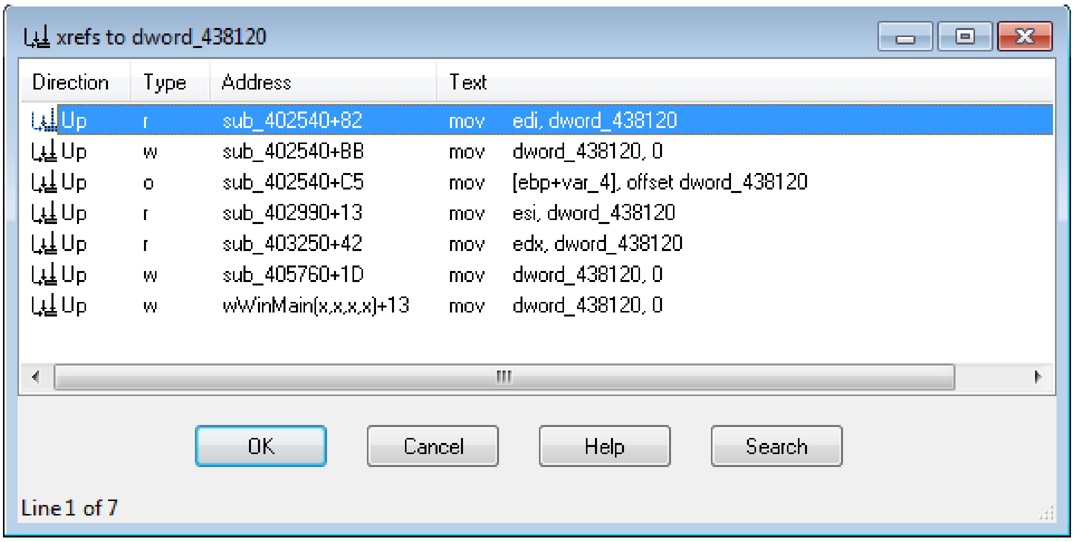
As an example, go to *wWinMain* function.

```
00406060 ; __stdcall wWinMain(x, x, x, x)
00406060 _wWinMain@16 proc near
00406060 push    esi
00406061 push    edi
00406062 call    sub_402860
00406067 mov     esi, ds:CreateMutexW
0040606D push    0               ; lpName
0040606F push    0               ; bInitialOwner
00406071 push    0               ; lpMutexAttributes
00406073 mov     dword_438120, 0
0040607D call    esi ; CreateMutexW
0040607F mov     edi, ds:time
00406085 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.

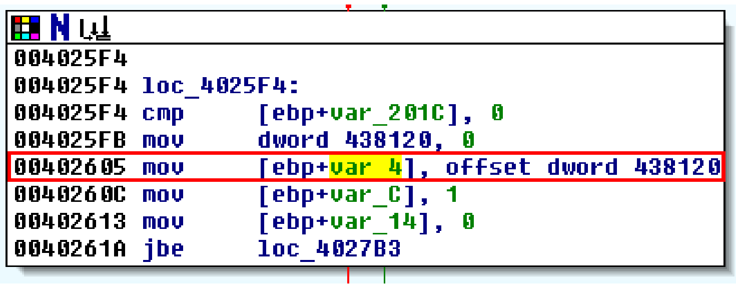
```
.data:0043811C ; HANDLE hHandle
.data:0043811C hHandle         dd ?                    ; DATA XREF: sub_402540+76↑r
.data:0043811C                                         ; sub_402540+27B↑r ...
.data:00438120 dword_438120    dd ?                    ; DATA XREF: sub_402540+82↑r
.data:00438120                                         ; sub_402540+BB↑w ...
.data:00438124 dword_438124    dd ?                    ; DATA XREF: sub_402300+92↑w
.data:00438124                                         ; sub_402300+14F↑w ...
```

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 consist 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.

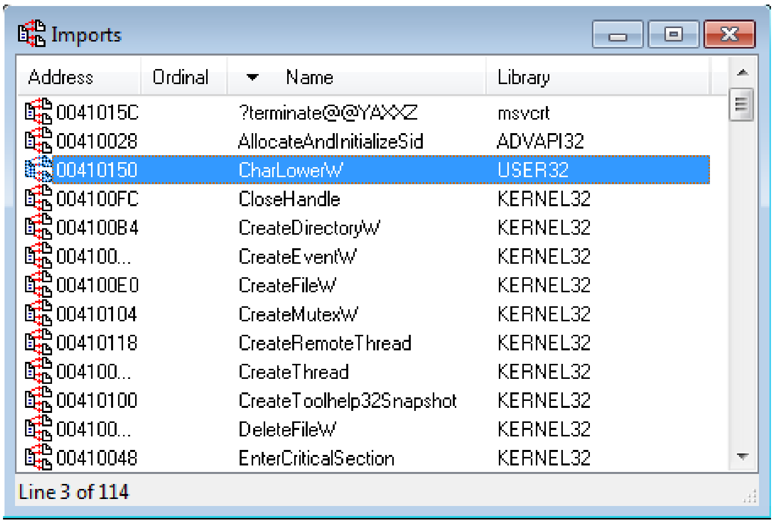
You can also immediately jump to any cross reference by double clicking it. For example, jump to the cross reference at the address *sub_402540+C5* (if you then want to go back, simply press *<Esc>*).



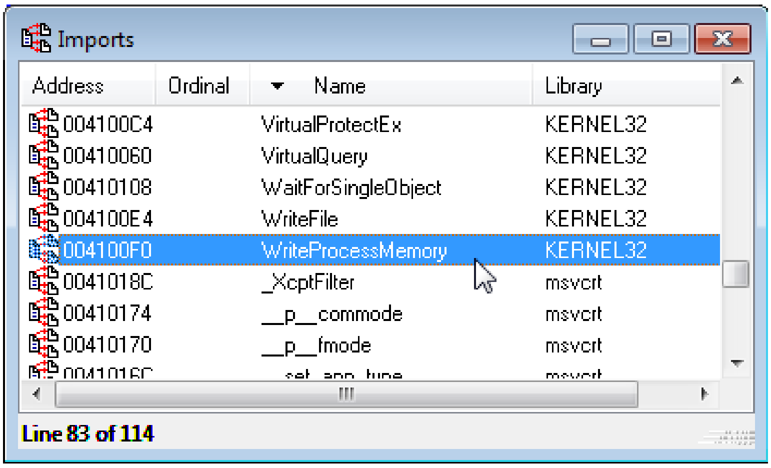
At this address, you see that the data address is moved onto the stack (assigned to local variable *var_4*).

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.

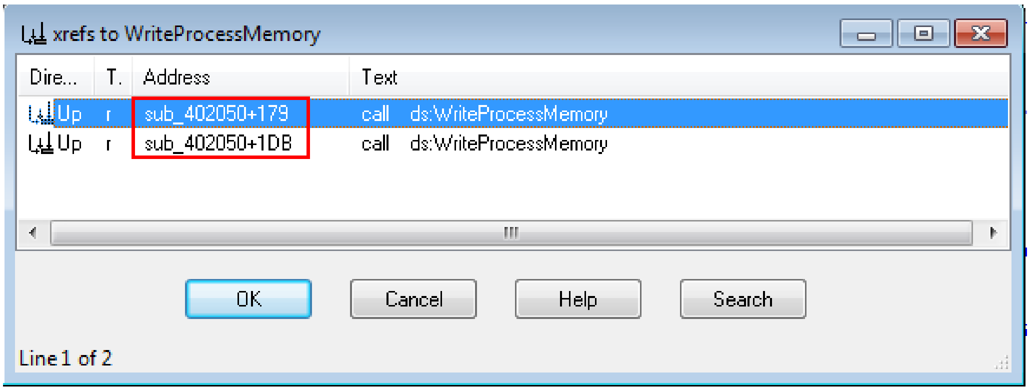


Let’s say you want to find which function is injecting code to other processes. To do this, first find the *WriteProcessMemory* function on the imports list and double click it.




```
.idata:004100EC ; HANDLE __stdcall CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,DWORD d
.idata:004100EC                 extrn CreateThread:dword ; DATA XREF: sub_405760+42↑r
.idata:004100EC                                         ; wWinMain(x,x,x,x)+7A↑r
.idata:004100F0 ; BOOL __stdcall WriteProcessMemory(HANDLE hProcess,LPVOID lpBaseAddress,LPVOID
.idata:004100F0                 extrn WriteProcessMemory:dword ; DATA XREF: sub_402050+179↑r
.idata:004100F0                                         ; sub_402050+1DB↑r
.idata:004100F4 ; HLOCAL __stdcall LocalFree(HLOCAL hMem)
.idata:004100F4                 extrn LocalFree:dword   ; DATA XREF: sub_402050+22E↑r
.idata:004100F4                                         ; sub_406410+356↑r ...
```

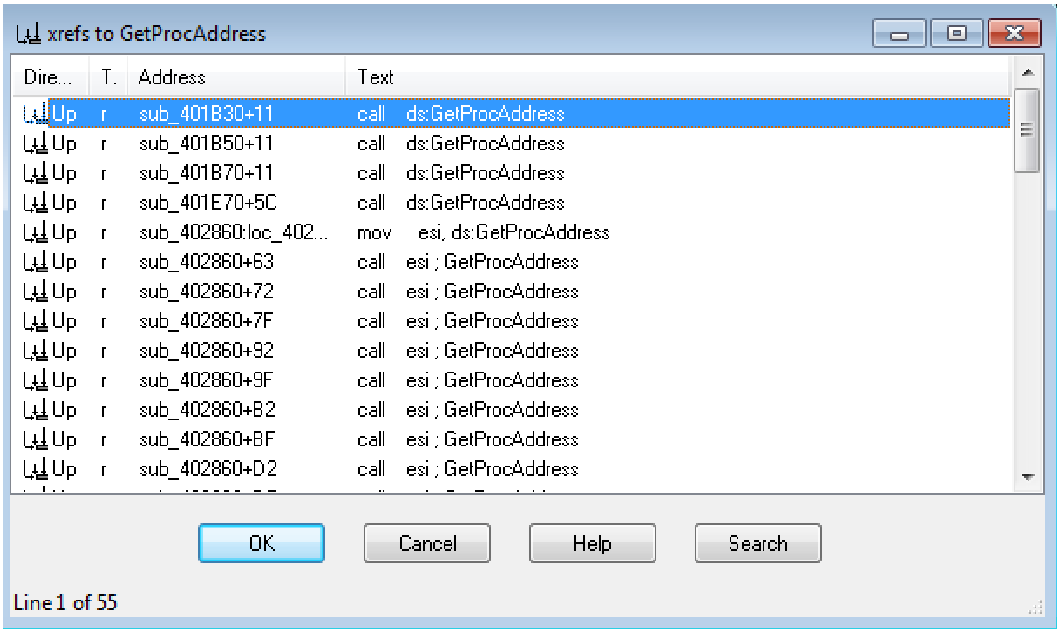
Next click on the function name and open the *Xrefs* dialog.



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.

```
.idata:00410128                 extrn OpenMutexW:dword  ; DATA XREF: sub_401DA0+97↑r
.idata:0041012C ; BOOL __stdcall Process32FirstW(HANDLE hSnapshot,LPPROCESSENTRY32W lppe)
.idata:0041012C                 extrn Process32FirstW:dword ; DATA XREF: sub_401B90+3D↑r
.idata:00410130 ; FARPROC __stdcall GetProcAddress(HMODULE hModule,LPCSTR lpProcName)
.idata:00410130                 extrn GetProcAddress:dword ; DATA XREF: sub_401B30+11↑r
.idata:00410130                                         ; sub_401B50+11↑r ...
.idata:00410134 ; LPVOID __stdcall VirtualAlloc(LPVOID lpAddress,DWORD dwSize,DWORD flAllo
.idata:00410134                 extrn VirtualAlloc:dword ; DATA XREF: sub_402050+58↑r
.idata:00410134                                         ; sub_406B70+BF↑r
```



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*

- *sub_401E70 – 2 calls*
- ***sub_402860 – 15 calls***
- *sub_403120 – 1 call*
- *sub_4041E0 – 1 call*
- *sub_404330 – 2 calls*
- *sub_404600 – 5 calls*
- *sub_405390 – 10 calls*
- *sub_405760 – 17 calls*

Now go to any cross reference in ***sub_402860*** (or just go to this function), and take a look at calls to *GetProcAddress*:

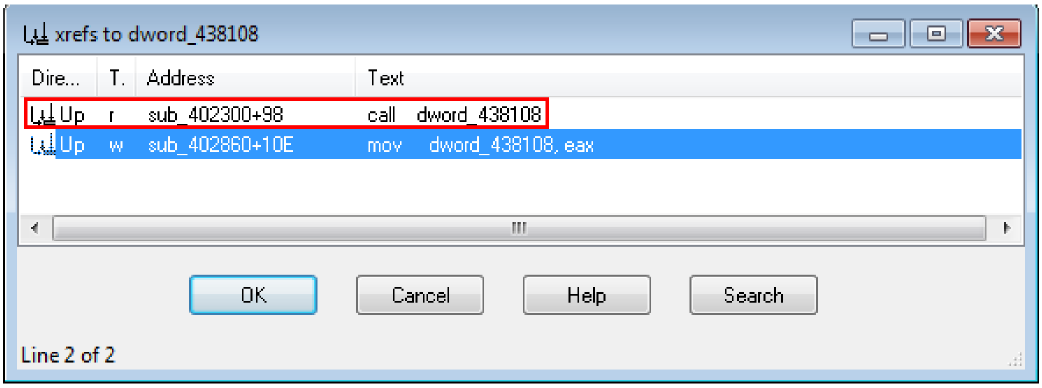
```
00402939 loc_402939:                ; "InternetOpenA"
00402939 push    offset aInternetopena
0040293E push    edi                ; hModule
0040293F call    esi ; GetProcAddress
00402941 push    offset aInternetconnec ; "InternetConnectA"
00402946 push    edi                ; hModule
00402947 mov     dword_438104, eax
0040294C call    esi ; GetProcAddress
0040294E push    offset aHttpopenreques ; "HttpOpenRequestA"
00402953 push    edi                ; hModule
00402954 mov     dword_43810C, eax
00402959 call    esi ; GetProcAddress
0040295B push    offset aHttpsendreques ; "HttpSendRequestA"
00402960 push    edi                ; hModule
00402961 mov     dword_438114, eax
00402966 call    esi ; GetProcAddress
00402968 push    offset aInternetreadfi ; "InternetReadFile"
0040296D push    edi                ; hModule
0040296E mov     dword_438108, eax
00402973 call    esi ; GetProcAddress
00402975 push    offset aInternetcloseh ; "InternetCloseHandle"
0040297A push    edi                ; hModule
0040297B mov     dword_438118, eax
00402980 call    esi ; GetProcAddress
00402982 pop     edi
00402983 pop     esi
00402984 mov     dword_438110, eax
```

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

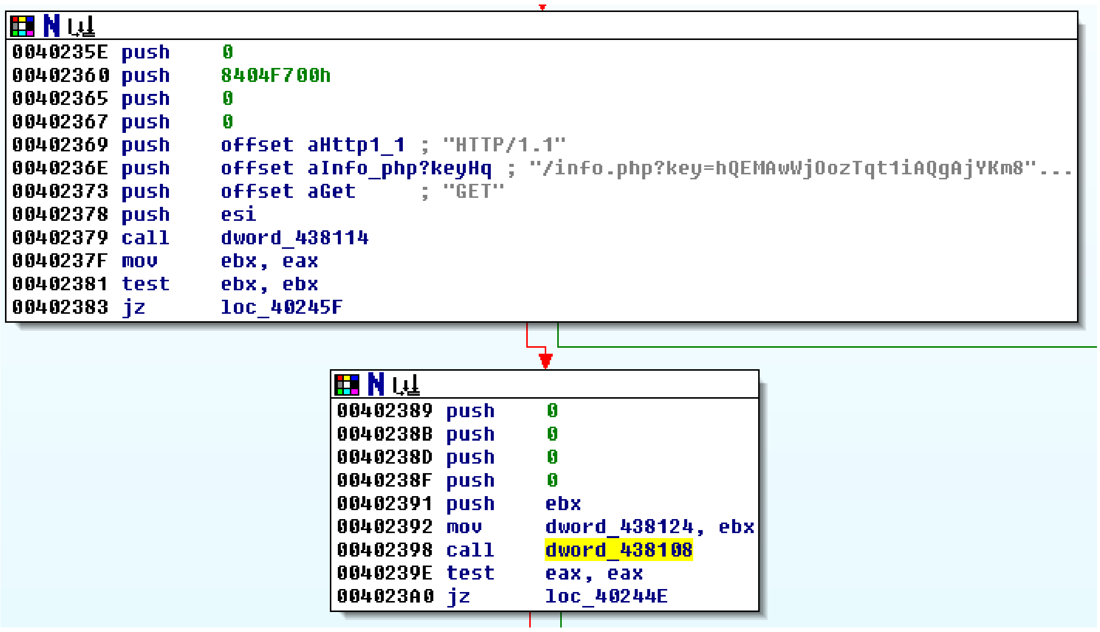
- *InternetOpenA -> dword_438104*
- *InternetConnectA -> dword_43810C*
- *HttpOpenRequestA -> dword_438114*
- *HttpSendRequestA ->* ***dword_438108***
- *InternetReadFile -> dword_438118*
- *InternetCloseHandle -> dword_438110*

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

```
.data:00438104 dword_438104     dd ?                    ; DATA XREF: sub_402300+25↑r
.data:00438104                                          ; sub_402860+E7↑w
.data:00438108 dword_438108     dd ?                    ; DATA XREF: sub_402300+98↑r
.data:00438108                                          ; sub_402860+10E↑w
.data:0043810C dword_43810C     dd ?                    ; DATA XREF: sub_402300+4B↑r
.data:0043810C                                          ; sub_402860+F4↑w
```



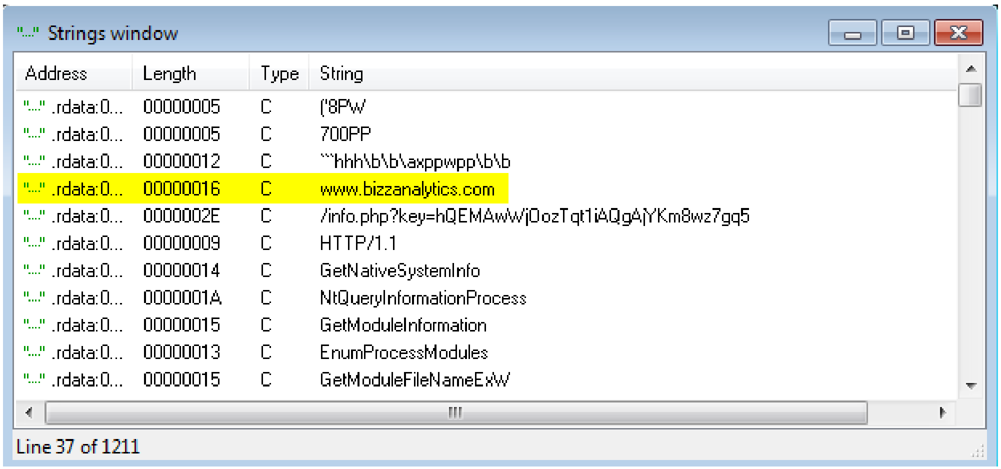
You see that there is one call to *HttpSendRequestA* in ***sub_402300***. Follow this cross reference to land in a function which is evidently used to communicate with some C&C server. This function was missed by us before because in this function the only meaningful API calls are to network functions 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.

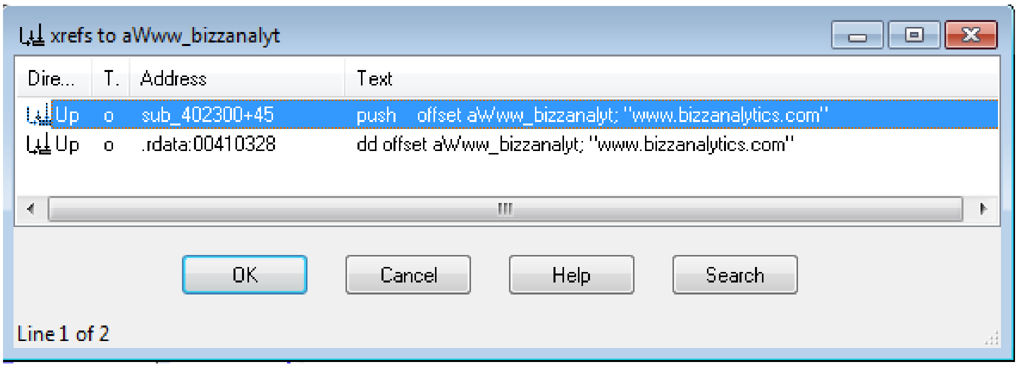
A second way of finding important functions using cross references is to follow cross references to strings found by IDA. You follow cross references to strings in a similar manner to following cross references to imported functions. First you open the strings list, then you look for any strings that 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:

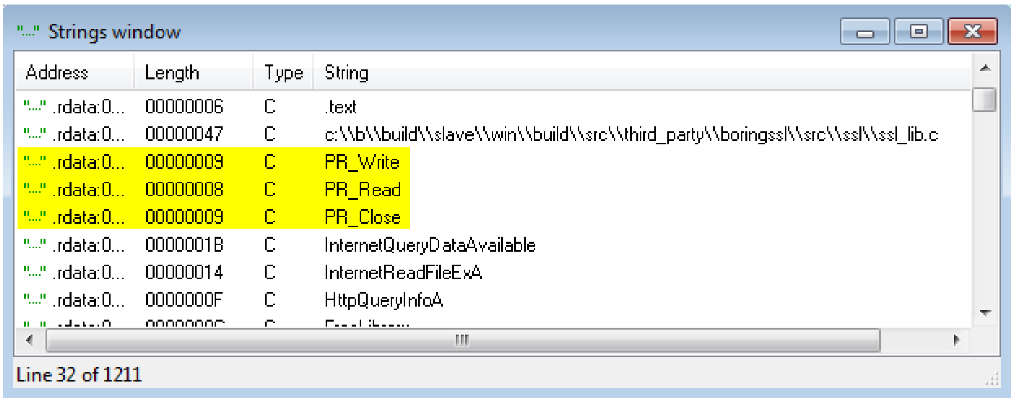
```
.rdata:00411140                 unicode 0, <__NTDLL_CORE__>,0
.rdata:0041115E                 align 10h
.rdata:00411160 aWww_bizzanalyt db 'www.bizzanalytics.com',0 ; DATA XREF: sub_402300+45↑o
.rdata:00411160                                          ; .rdata:00410328↑o
.rdata:00411176                 db    0
.rdata:00411177                 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           ; "__NTDLL_CORE__"
.rdata:00410324                 dd offset aInfo_php?keyHq ; "/info.php?key=hQEMAwWjOozTqt1iAQgAjYKm8"...
.rdata:00410328                 dd offset aWww_bizzanalyt ; "www.bizzanalytics.com"
.rdata:0041032C                 dd offset aGet_0         ; "GET "
.rdata:00410330                 dd offset aWininet_dll   ; "wininet.dll"
.rdata:00410334                 dd offset asc_411420     ; "\r\n"
.rdata:00410338                 dd offset aPost          ; "POST "
.rdata:0041033C                 dd offset aAcceptEncoding ; "\nAccept-Encoding: "
.rdata:00410340                 dd offset aTransferEncodi ; "Transfer-Encoding: chunked\r\n"
```

Now go back to the strings window and notice the strings named *PR_Write, PR_Read*, and *PR_Close*, which are names of functions from the NSPR library used for network communication[15]. This library is used for example by Mozilla Firefox web browser. This is typical for modern malware performing so-called MitB (*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[16] [17] [18].


Strings window

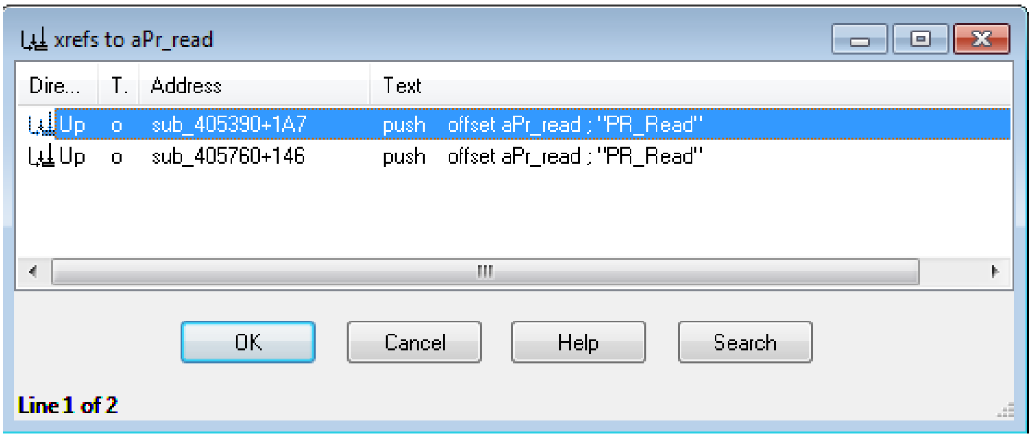
Address	Length	Type	String
.rdata:0...	00000006	C	.text
.rdata:0...	00000047	C	c:\\b\\build\\slave\\win\\build\\src\\third_party\\boringssl\\src\\ssl\\ssl_lib.c
.rdata:0...	00000009	C	PR_Write
.rdata:0...	00000008	C	PR_Read
.rdata:0...	00000009	C	PR_Close
.rdata:0...	0000001B	C	InternetQueryDataAvailable
.rdata:0...	00000014	C	InternetReadFileExA
.rdata:0...	0000000F	C	HttpQueryInfoA

Line 32 of 1211

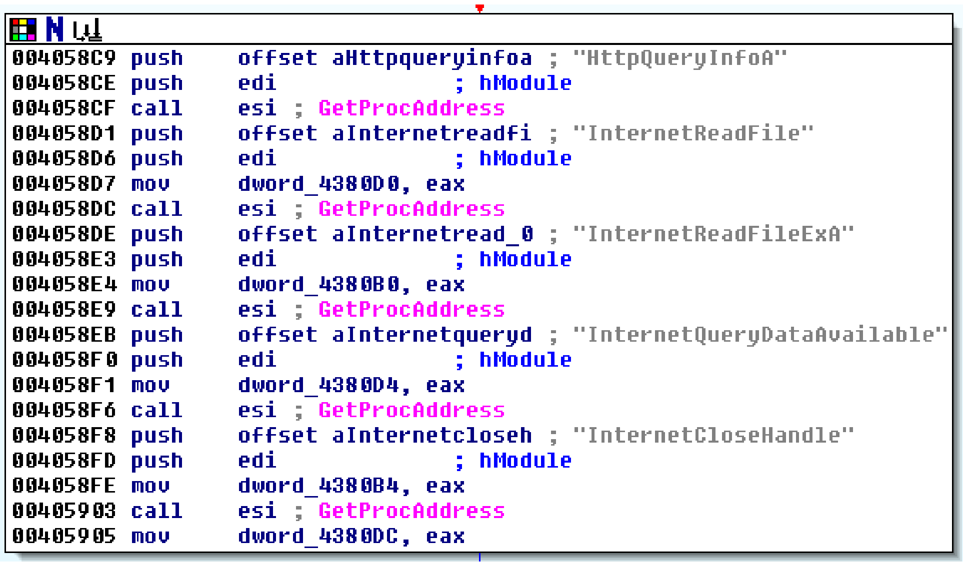
Let's examine where those strings are referenced.

```
.rdata:00411558 ; char aPr_write[]
.rdata:00411558 aPr_write       db 'PR_Write',0         ; DATA XREF: sub_405390+17E↑o
.rdata:00411558                                         ; sub_405760:loc_40589E↑o
.rdata:00411561                 align 4
.rdata:00411564 ; char aPr_read[]
.rdata:00411564 aPr_read        db 'PR_Read',0          ; DATA XREF: sub_405390+1A7↑o
.rdata:00411564                                         ; sub_405760+146↑o
.rdata:0041156C ; char aPr_close[]
.rdata:0041156C aPr_close       db 'PR_Close',0         ; DATA XREF: sub_405390+1D0↑o
.rdata:0041156C                                         ; sub_405760+153↑o
.rdata:00411575                 align 4
```

[15] Netscape Portable Runtime (NSPR) https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR (last accessed 11.09.2015)
[16] Advanced Techniques in Modern Banking Trojans https://www.botconf.eu/wp-content/uploads/2013/12/02-BankingTrojans-ThomasSiebert.pdf (last accessed 11.09.2015)
[17] Analyzing Man-in-the-Browser (MITB) Attacks http://www.sans.org/reading-room/whitepapers/forensics/analyzing-man-in-the-browser-mitb-attacks-35687 (last accessed 11.09.2015)
[18] Firefox FormGrabber https://redkiing.wordpress.com/2012/04/30/firefox-formgrabber-iii-code-injection/ (last accessed 11.09.2015)



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.



```
004058C9 push    offset aHttpqueryinfoa ; "HttpQueryInfoA"
004058CE push    edi             ; hModule
004058CF call    esi ; GetProcAddress
004058D1 push    offset aInternetreadfi ; "InternetReadFile"
004058D6 push    edi             ; hModule
004058D7 mov     dword_4380D0, eax
004058DC call    esi ; GetProcAddress
004058DE push    offset aInternetread_0 ; "InternetReadFileExA"
004058E3 push    edi             ; hModule
004058E4 mov     dword_4380B0, eax
004058E9 call    esi ; GetProcAddress
004058EB push    offset aInternetqueryd ; "InternetQueryDataAvailable"
004058F0 push    edi             ; hModule
004058F1 mov     dword_4380D4, eax
004058F6 call    esi ; GetProcAddress
004058F8 push    offset aInternetcloseh ; "InternetCloseHandle"
004058FD push    edi             ; hModule
004058FE mov     dword_4380B4, eax
00405903 call    esi ; GetProcAddress
00405905 mov     dword_4380DC, eax
```

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[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.

[19] POS malware - a look at Dexter and Decebal http://h30499.www3.hp.com/t5/HP-Security-Research-Blog/POS-malware-a-look-at-Dexter-and-Decebal/ba-p/6654398 (last accessed 11.09.2015)

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):

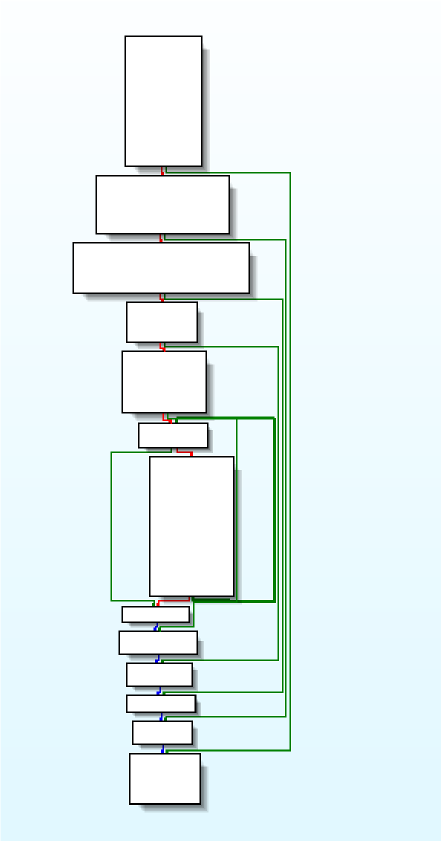
- Are there any API calls in the function? If yes, what are they used for?
- Are there any calls to other local functions? What are they doing?
- Are there any xrefs to the analysed function? From which other functions is the function called? Are there any arguments pushed onto the stack when the function 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
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_4= dword ptr -4
00402300
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_438104
0040232B 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
00402939 loc_402939:                 ; "InternetOpenA"
00402939 push    offset aInternetopena
0040293E push    edi                 ; hModule
0040293F call    esi ; GetProcAddress
00402941 push    offset aInternetconnec ; "InternetConnectA"
00402946 push    edi                 ; hModule
00402947 mov     dword_438104, eax
0040294C call    esi ; GetProcAddress
0040294E push    offset aHttpopenreques ; "HttpOpenRequestA"
00402953 push    edi                 ; hModule
00402954 mov     dword_43810C, eax
00402959 call    esi ; GetProcAddress
0040295B push    offset aHttpsendreques ; "HttpSendRequestA"
00402960 push    edi                 ; hModule
00402961 mov     dword_438114, eax
00402966 call    esi ; GetProcAddress
00402968 push    offset aInternetreadfi ; "InternetReadFile"
0040296D push    edi                 ; hModule
0040296E mov     dword_438108, eax
00402973 call    esi ; GetProcAddress
00402975 push    offset aInternetcloseh ; "InternetCloseHandle"
0040297A push    edi                 ; hModule
0040297B mov     dword_438118, eax
00402980 call    esi ; GetProcAddress
00402982 pop     edi
00402983 pop     esi
00402984 mov     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):

```
00402939 loc_402939:              ; "InternetOpenA"
00402939 push    offset aInternetopena
0040293E push    edi              ; hModule
0040293F call    esi ; GetProcAddress
00402941 push    offset aInternetconnec ; "InternetConnectA"
00402946 push    edi              ; hModule
00402947 mov     InternetOpenA, eax
0040294C call    esi ; GetProcAddress
0040294E push    offset aHttpopenreques ; "HttpOpenRequestA"
00402953 push    edi              ; hModule
00402954 mov     InternetConnectA, eax
00402959 call    esi ; GetProcAddress
0040295B push    offset aHttpsendreques ; "HttpSendRequestA"
00402960 push    edi              ; hModule
00402961 mov     HttpOpenRequestA, eax
00402966 call    esi ; GetProcAddress
00402968 push    offset aInternetreadfi ; "InternetReadFile"
0040296D push    edi              ; hModule
0040296E mov     HttpSendRequestA, eax
00402973 call    esi ; GetProcAddress
00402975 push    offset aInternetcloseh ; "InternetCloseHandle"
0040297A push    edi              ; hModule
0040297B mov     InternetReadFile, eax
00402980 call    esi ; GetProcAddress
00402982 pop     edi
00402983 pop     esi
00402984 mov     InternetCloseHandle, eax
```

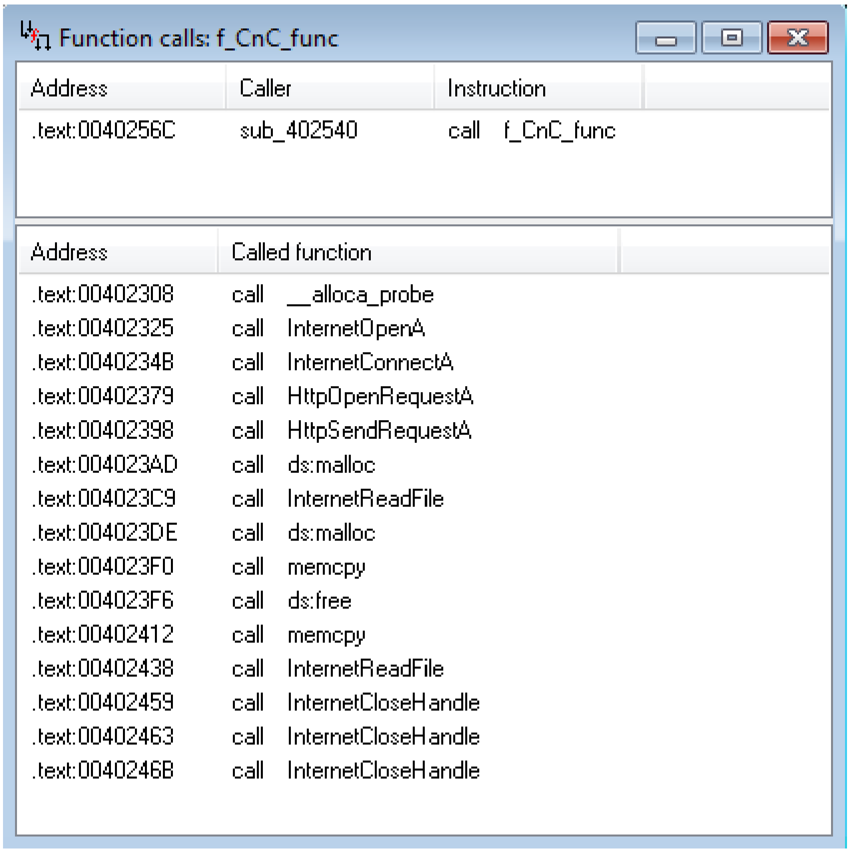
Now go back to *f_CnC_func* and reanalyse code (*Options->General->Analysis->Reanalyse program*). IDA should add additional comments[20]:

```
0040230F push    0                ; dwFlags
00402311 push    0                ; lpszProxyBypass
00402313 push    0                ; lpszProxy
00402315 push    0                ; dwAccessType
00402317 mov     edi, ecx
00402319 mov     [ebp+var_14], 0
00402320 push    0                ; lpszAgent
00402322 mov     [ebp+var_C], edi
00402325 call    InternetOpenA
```

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.



Function calls: f_CnC_func

Address	Caller	Instruction
.text:0040256C	sub_402540	call f_CnC_func

Address	Called function
.text:00402308	call __alloca_probe
.text:00402325	call InternetOpenA
.text:0040234B	call InternetConnectA
.text:00402379	call HttpOpenRequestA
.text:00402398	call HttpSendRequestA
.text:004023AD	call ds:malloc
.text:004023C9	call InternetReadFile
.text:004023DE	call ds:malloc
.text:004023F0	call memcpy
.text:004023F6	call ds:free
.text:00402412	call memcpy
.text:00402438	call InternetReadFile
.text:00402459	call InternetCloseHandle
.text:00402463	call InternetCloseHandle
.text:0040246B	call InternetCloseHandle

In the upper part of the window, there is a list of locations where *f_CnC_func* was called. In the lower part of the window there is a list of all calls made within *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, there are no other API calls except calls to network related functions (and a few memory allocation functions from C standard library). Secondly, there are no calls to other local functions. Thirdly, *f_CnC_func* is called only once (in *sub_402540* function).

Knowing this plus the fact that *f_CnC_func* is rather simple and short function you can assume that that *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:

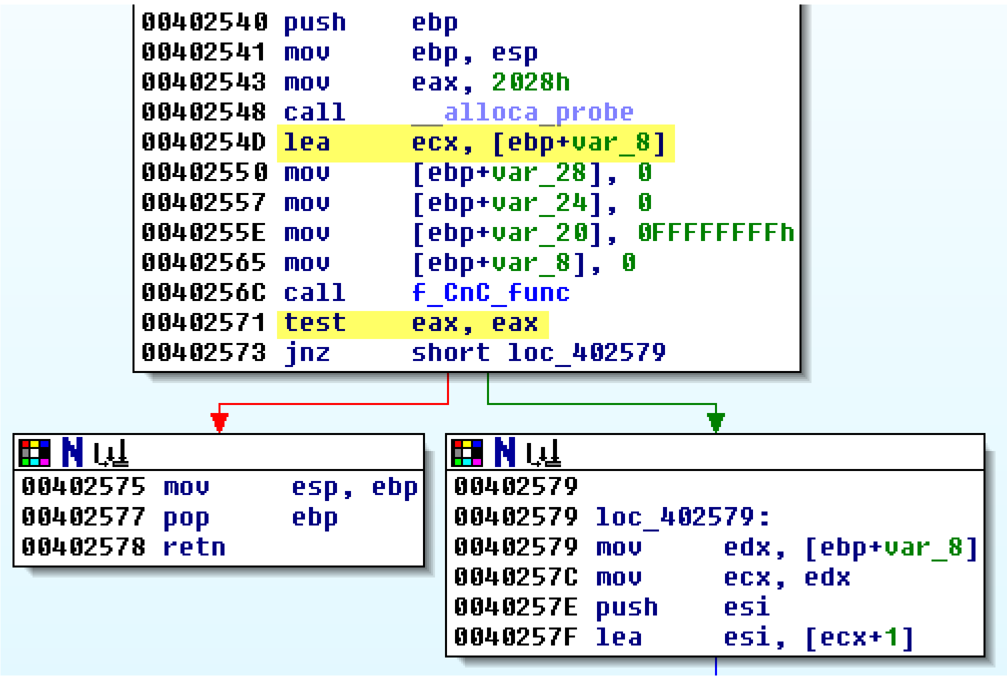
- What are *f_CnC_func* arguments?
- Is *f_CnC_func* returning anything?
- Is there any data sent to C&C server? How?
- Is there any data received from C&C server? What is happening to this data?

Let's start by analysing if there are any function arguments:

```
00402300 ; Attributes: bp-based frame
00402300
00402300 f_CnC_func proc near
00402300
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_4= dword ptr -4
00402300
00402300 push    ebp
00402301 mov     ebp, esp
00402303 mov     eax, 1018h
00402308 call    __alloca_probe
```

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 *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 *sub_402540*. It seems there are no push instructions before a call to *f_CnC_func*. However notice that *ecx* register is assigned with the address of *var_8* variable, which is later also initialized to zero.

Notice also how *eax* register is tested after a call to *f_CnC_func* and if it equals to zero *sub_402540* returns. This suggests that *f_CnC_func* is returning some value in *eax* register and it should be nonzero on success.

Now go back to *f_CnC_func* to check if *ecx* register is used for anything.

```
00402300 push    ebp
00402301 mov     ebp, esp
00402303 mov     eax, 1018h
00402308 call    __alloca_probe
0040230D push    ebx
0040230E push    edi
0040230F push    0               ; dwFlags
00402311 push    0               ; lpszProxyBypass
00402313 push    0               ; lpszProxy
00402315 push    0               ; dwAccessType
00402317 mov     edi, ecx
00402319 mov     [ebp+var_14], 0
00402320 push    0               ; lpszAgent
00402322 mov     [ebp+var_C], edi
00402325 call    InternetOpenA
```

Yes, you were right. Value of *ecx* is assigned to *edi* register. This means that *f_CnC_func* is either using the *fastcall* calling convention or you might be dealing with object-oriented programming and *ecx* is used to pass *this* pointer to a member function (*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 *fastcall* function and *ecx* is used to pass pointer to variable or some data structure.

Notice that later the *edi* register is assigned to *var_C*. Rename *var_C* to *this*.

```
00402317 mov     edi, ecx
00402319 mov     [ebp+var_14], 0
00402320 push    0               ; lpszAgent
00402322 mov     [ebp+this], edi
00402325 call    InternetOpenA
```

Now go to the last block of *f_CnC_func* (loc_40246A):

```
0040246A
0040246A loc_40246A:             ; hInternet
0040246A push    ebx
0040246B call    InternetCloseHandle
00402471 mov     eax, [ebp+var_14]
00402474 pop     edi
00402475 pop     ebx
00402476 mov     esp, ebp
00402478 pop     ebp
00402479 retn
00402479 sub_402300 endp
00402479
```

Notice that the *eax* register is assigned with the value of the *var_14* variable. This means that the *var_14* variable is used to store the return value. Rename *var_14* to *retval*. For convenience it is also good to rename label *loc_40246A* to something like *func_exit*:

```
0040246A
0040246A func_exit:                    ; hInternet
0040246A push     ebx
0040246B call     InternetCloseHandle
00402471 mov      eax, [ebp+retval]
00402474 pop      edi
00402475 pop      ebx
00402476 mov      esp, ebp
00402478 pop      ebp
00402479 retn
00402479 sub_402300 endp
00402479
```

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.

```
0040230F push    0                  ; dwFlags
00402311 push    0                  ; lpszProxyBypass
00402313 push    0                  ; lpszProxy
00402315 push    0                  ; dwAccessType
00402317 mov     edi, ecx
00402319 mov     [ebp+retval], 0
00402320 push    0                  ; lpszAgent
00402322 mov     [ebp+this], edi
00402325 call    InternetOpenA
0040232B mov     ebx, eax
0040232D 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*.

```
00402319 mov     [ebp+retval], 0
00402320 push    0                 ; lpszAgent
00402322 mov     [ebp+this], edi
00402325 call    InternetOpenA
0040232B mov     ebx, eax
0040232D 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.

[21] InternetOpen function https://msdn.microsoft.com/en-us/library/windows/desktop/aa385096%28v=vs.85%29.aspx (last accessed 11.09.2015)

```
00402338 push    esi
00402339 push    0               ; dwContext
0040233B push    0               ; dwFlags
0040233D push    3               ; dwService
0040233F push    0               ; lpszPassword
00402341 push    0               ; lpszUserName
00402343 push    50h             ; nServerPort
00402345 push    offset szServerName ; "www.bizzanalytics.com"
0040234A push    ebx             ; hInternet
0040234B call    InternetConnectA
00402351 mov     esi, eax
00402353 mov     [ebp+var_18], esi
00402356 test    esi, esi
00402358 jz      loc_402462
```

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*.

```
00402338 push    esi
00402339 push    0               ; dwContext
0040233B push    0               ; dwFlags
0040233D push    INTERNET_SERVICE_HTTP ; dwService
0040233F push    0               ; lpszPassword
00402341 push    0               ; lpszUserName
00402343 push    80              ; nServerPort
00402345 push    offset szServerName ; "www.bizzanalytics.com"
0040234A push    ebx             ; hInternet
0040234B call    InternetConnectA
00402351 mov     esi, eax
00402353 mov     [ebp+hConnect], esi
00402356 test    esi, esi
00402358 jz      loc_402462
```

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

```
0040235E push    0               ; dwContext
00402360 push    8404F700h       ; dwFlags
00402365 push    0               ; lplpszAcceptTypes
00402367 push    0               ; lpszReferrer
00402369 push    offset szVersion ; "HTTP/1.1"
0040236E push    offset szObjectName ; "/info.php?key=hQEMAwWjOozTqt1iAQgAjYKm8"...
00402373 push    offset szVerb   ; "GET"
00402378 push    esi             ; hConnect
00402379 call    HttpOpenRequestA
0040237F mov     ebx, eax
00402381 test    ebx, ebx        ; ebx <- hRequest
00402383 jz      loc_40245F
```

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.

```
.rdata:00411197                 db    0
.rdata:00411198 ; char szObjectName[]
.rdata:00411198 szObjectName    db '/info.php?key=hQEMAwWjOozTqt1iAQgAjYKm8wz7gq5',0
.rdata:00411198                                         ; DATA XREF: f_CnC_func+6E↑o
.rdata:00411198                                         ; .rdata:00410324↑o
.rdata:004111C6                 db    0
```

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.

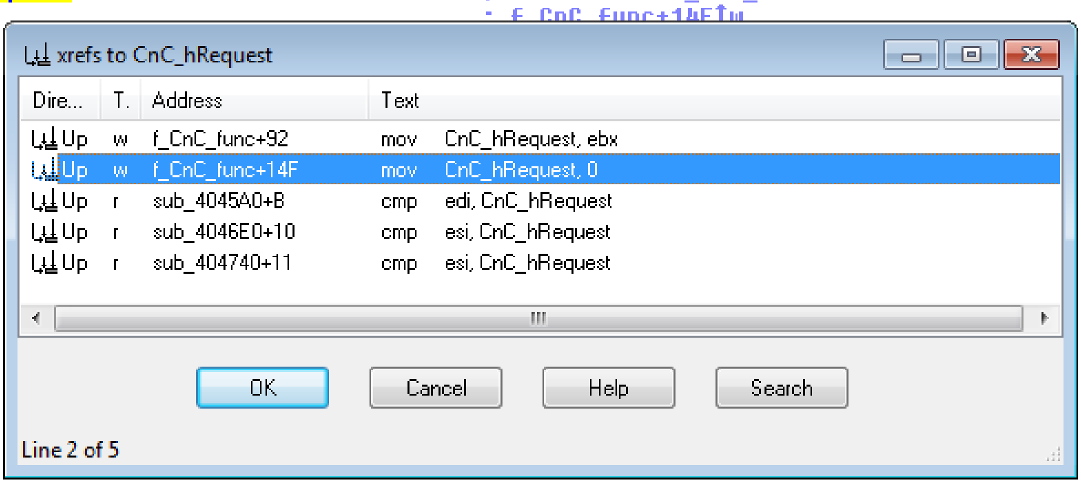
Next the malware is sending an HTTP request.

```
00402389 push    0                  ; dwOptionalLength
0040238B push    0                  ; lpOptional
0040238D push    0                  ; dwHeadersLength
0040238F push    0                  ; lpszHeaders
00402391 push    ebx                ; hRequest
00402392 mov     dword_438124, ebx
00402398 call    HttpSendRequestA
0040239E test    eax, eax
004023A0 jz      loc_40244E
```

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.

```
00402389 push    0                  ; dwOptionalLength
0040238B push    0                  ; lpOptional
0040238D push    0                  ; dwHeadersLength
0040238F push    0                  ; lpszHeaders
00402391 push    ebx                ; hRequest
00402392 mov     CnC_hRequest, ebx
00402398 call    HttpSendRequestA
0040239E test    eax, eax
004023A0 jz      loc_40244E
```

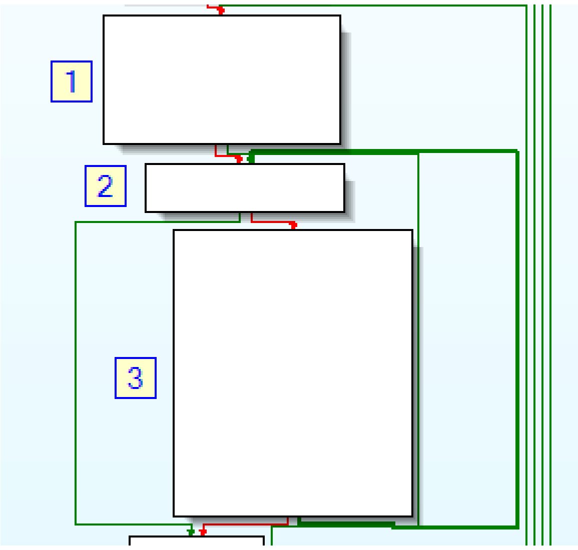
```
CnC_hRequest    dd ?                    ; DATA XREF: f_CnC_func+92↑w

_data
```



Notice that there are some references to this variable outside of the *f_CnC_func*. Renaming this variable might help us in later analysis.

Next if sending HTTP requests doesn't fail (*eax* will be nonzero on fail), the malware starts reading data received from the server (*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 *InternetReadFile*.

```
1
004023A6 xor     esi, esi
004023A8 push    1                ; size_t
004023AA mov     [ebp+var_8], esi
004023AD call    ds:malloc
004023B3 add     esp, 4
004023B6 mov     [edi], eax
004023B8 lea     eax, [ebp+dwNumberOfBytesRead]
004023BB push    eax              ; lpdwNumberOfBytesRead
004023BC push    1000h            ; dwNumberOfBytesToRead
004023C1 lea     eax, [ebp+Buffer]
004023C7 push    eax              ; lpBuffer
004023C8 push    ebx              ; hFile
004023C9 call    InternetReadFile
004023CF test    eax, eax
004023D1 jz      short loc_402445
```

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

```
0040242A push    eax              ; lpdwNumberOfBytesRead
0040242B push    1000h            ; dwNumberOfBytesToRead
00402430 lea     eax, [ebp+Buffer]
00402436 push    eax              ; lpBuffer
00402437 push    ebx              ; hFile
00402438 call    InternetReadFile
0040243E test    eax, eax
00402440 jnz     short loc_4023D3
```

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 *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.

```
004023A6 xor     esi, esi
004023A8 push    1               ; size_t
004023AA mov     [ebp+var_8], esi
004023AD 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:

```
00402317 mov     edi, ecx
00402319 mov     [ebp+retval], 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:

```
004023E9 push    [ebp+var_8]     ; size_t
004023EC mov     esi, [esi]
004023EE push    esi             ; void *
004023EF push    edi             ; void *
004023F0 call    memcpy
004023F5 push    esi             ; void *
004023F6 call    ds:free

004023FF push    [ebp+dwNumberOfBytesRead] ; size_t
00402402 mov     esi, [ebp+var_8]
00402405 mov     [eax], edi
00402407 lea     eax, [ebp+Buffer]
0040240D push    eax             ; void *
0040240E lea     eax, [edi+esi]
00402411 push    eax             ; void *
00402412 call    memcpy
00402417 add     esi, [ebp+dwNumberOfBytesRead]
0040241A lea     eax, [ebp+dwNumberOfBytesRead]
0040241D add     esp, 20h
00402420 mov     [ebp+var_8], esi
```

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]:

```
004023A6 xor     esi, esi        ; esi <- 0
004023A8 push    1               ; size_t
004023AA mov     [ebp+recv_len], esi ; recv_len <- 0
004023AD 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*:

```
004023B6 mov     [edi], eax       ; *this <- eax (memptr)
004023B8 lea     eax, [ebp+dwNumberOfBytesRead]
004023BB push    eax              ; lpdwNumberOfBytesRead
004023BC push    1000h            ; dwNumberOfBytesToRead
004023C1 lea     eax, [ebp+Buffer]
004023C7 push    eax              ; lpBuffer
004023C8 push    ebx              ; hFile
004023C9 call    InternetReadFile
004023CF test    eax, eax
004023D1 jz      short loc_402445
```

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.

```
00402300 f_CnC_func proc near
00402300
00402300 Buffer= dword ptr -1018h
00402300 hConnect= dword ptr -18h
00402300 retval= dword ptr -14h
00402300 hInternet= dword ptr -10h
00402300 this= dword ptr -0Ch
00402300 recv_len= dword ptr -8
00402300 dwNumberOfBytesRead= dword ptr -4
00402300
```

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.

2

```
004023D3
004023D3 loop:
004023D3 mov     eax, [ebp+dwNumberOfBytesRead]
004023D6 test    eax, eax
004023D8 jz      short loc_402442
```

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.

3

```
004023DA inc     eax              ; eax <- dwNumberOfBytesRead+1
004023DB add     eax, esi         ; eax <- eax+recv_len
004023DD push    eax
004023DE call    ds:malloc        ; Allocating new memory block of size:
004023DE                          ; recv_len+dwNumberOfBytesRead+1
004023E4 mov     esi, [ebp+this]
004023E7 mov     edi, eax         ; edi <- memptr_new
004023E9 push    [ebp+recv_len]   ; n (num of bytes to copy)
004023EC mov     esi, [esi]       ; esi <- *this (memptr_old)
004023EE push    esi              ; src
004023EF push    edi              ; dest
004023F0 call    memcpy           ; Copy recv_len bytes from
004023F0                          ; memptr_old to memptr_new
004023F5 push    esi              ; void *
004023F6 call    ds:free          ; free memptr_old
```

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.

3

```
004023F6 call    ds:free          ; free memptr_old
004023FC mov     eax, [ebp+this]
004023FF push    [ebp+dwNumberOfBytesRead] ; n (num of bytes to copy)
00402402 mov     esi, [ebp+recv_len] ; esi <- recv_len
00402405 mov     [eax], edi       ; *this <- memptr_new (updating memptr)
00402407 lea     eax, [ebp+Buffer]
0040240D push    eax              ; src (Buffer)
0040240E lea     eax, [edi+esi]
00402411 push    eax              ; dst (memptr_new)
00402412 call    memcpy           ; Copy newly received data from Buffer
00402412                          ; 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).

3

```
00402412 call    memcpy           ; Copy newly received data from Buffer
00402412                          ; to the end of memptr_new
00402417 add     esi, [ebp+dwNumberOfBytesRead] ; esi <- recv_len + dwNumberOfBytesRead
0040241A lea     eax, [ebp+dwNumberOfBytesRead]
0040241D add     esp, 20h
00402420 mov     [ebp+recv_len], esi ; recv_len <- recv_len+dwNumberOfBytesRead
00402423 mov     [ebp+retval], 1 ; Received some data: set retval to 1
0040242A push    eax              ; lpdwNumberOfBytesRead
0040242B push    4096             ; dwNumberOfBytesToRead
00402430 lea     eax, [ebp+Buffer]
00402436 push    eax              ; lpBuffer
00402437 push    ebx              ; hFile
00402438 call    InternetReadFile
0040243E test    eax, eax
00402440 jnz     short loop
```

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:

2

```
004023D3
004023D3 loop:
004023D3 mov     eax, [ebp+dwNumberOfBytesRead]
004023D6 test    eax, eax
004023D8 jz      short loc_402442
```

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

```
00402442
00402442 loc_402442:              ; edi <- this
00402442 mov     edi, [ebp+this]
```

```
00402445
00402445 loc_402445:
00402445 mov     eax, [edi]
00402447 mov     byte ptr [esi+eax], 0 ; Zeroing last allocated byte.
00402447                          ; eax - memptr
00402447                          ; esi - recv_len
0040244B mov     esi, [ebp+hConnect]
```

After this the only thing that happens is the closing all opened handles:

```
0040244E
0040244E loc_40244E:                ; hInternet
0040244E push    ebx
0040244F mov     CnC_hRequest, 0
00402459 call    InternetCloseHandle
```

```
00402462
00402462 loc_402462:                ; hInternet
00402462 push    esi
00402463 call    InternetCloseHandle
00402469 pop     esi
```

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

```
0040246A
0040246A func_exit:                 ; hInternet
0040246A push    ebx
0040246B call    InternetCloseHandle
00402471 mov     eax, [ebp+retval]
00402474 pop     edi
00402475 pop     ebx
00402476 mov     esp, ebp
00402478 pop     ebp
00402479 retn
00402479 sub_402300 endp
00402479
```

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.

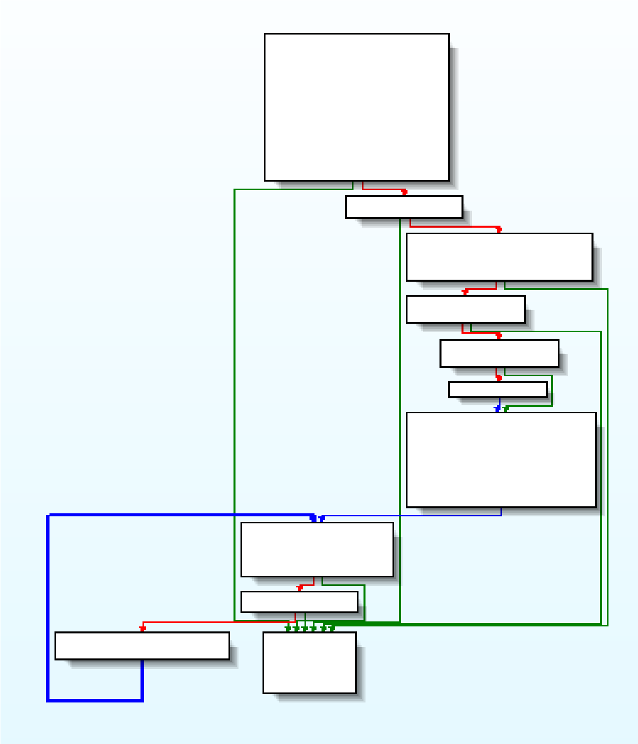
What you have learnt about *f_CnC_func*:

- 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 *wWinMain* is not using any local variables nor referencing any of its arguments.

```
00406060
00406060
00406060
00406060 ; __stdcall wWinMain(x, x, x, x)
00406060 _wWinMain@16 proc near
00406060 push    esi
00406061 push    edi
00406062 call    sub_402860
00406067 mov     esi, ds:CreateMutexW
0040606D push    0                  ; lpName
0040606F push    0                  ; bInitialOwner
00406071 push    0                  ; lpMutexAttributes
00406073 mov     dword_438120, 0
0040607D call    esi ; CreateMutexW
```

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*:

```
0040610F
0040610F func_exit:
0040610F pop     edi
00406110 xor     eax, eax
00406112 pop     esi
00406113 retn    10h
00406113 _wWinMain@16 endp
00406113
```

Now take a look at the first block of the function:

```
   00406060
   00406060
   00406060
   00406060 ; __stdcall wWinMain(x, x, x, x)
   00406060 _wWinMain@16 proc near
01 00406060 push    esi
02 00406061 push    edi
03 00406062 call    sub_402860
04 00406067 mov     esi, ds:CreateMutexW
05 0040606D push    0                 ; lpName
06 0040606F push    0                 ; bInitialOwner
07 00406071 push    0                 ; lpMutexAttributes
08 00406073 mov     dword_438120, 0
09 0040607D call    esi ; CreateMutexW
10 0040607F mov     edi, ds:time
11 00406085 push    0                 ; time_t *
12 00406087 mov     hHandle, eax
13 0040608C call    edi ; time
14 0040608E add     esp, 4
15 00406091 cmp     eax, dword_437E40
16 00406097 jl      short func_exit
```

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:

```
004028B5
004028B5 loc_4028B5:
004028B5 mov     esi, ds:GetProcAddress
004028BB push    offset ProcName ; "GetNativeSystemInfo"
004028C0 push    [ebp+hModule]    ; hModule
004028C3 call    esi ; GetProcAddress
004028C5 push    offset aNtqueryinforma ; "NtQueryInformationProcess"
004028CA push    [ebp+var_8]      ; hModule
004028CD mov     dword_4380F4, eax
004028D2 call    esi ; GetProcAddress
004028D4 push    offset aGetmoduleinfor ; "GetModuleInformation"
004028D9 push    ebx              ; hModule
004028DA mov     dword_4380F0, eax
004028DF call    esi ; GetProcAddress
004028E1 mov     dword_438100, eax
004028E6 test    eax, eax
004028E8 jnz     short loc_4028F9
```

Rename *sub_402860* to *f_Initialize_APIs*.

```
00406060 push    esi
00406061 push    edi
00406062 call    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*.

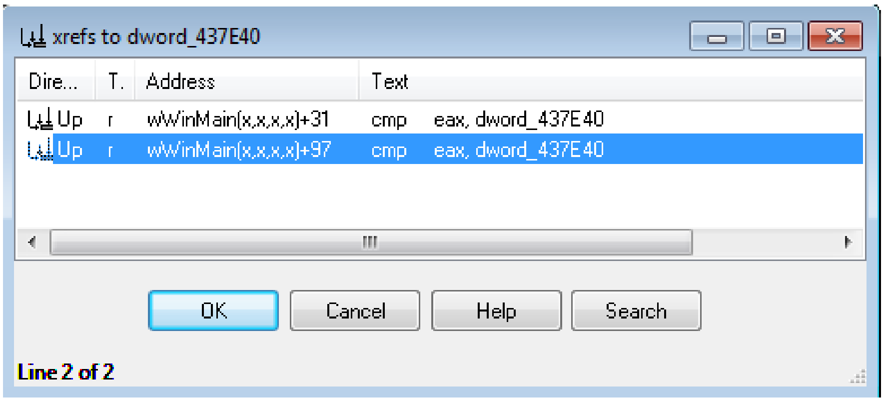
Additionally at line 11 some global variable (*dword_438120*) is initialized to zero. You don't know yet what this variable will be used for in the code but it is good to give it a temporary name, for example *var_main_zero*. If you later see reference to this variable you will immediately know it was first set to zero in the *wWinMain* function.

```
0040606D push    0               ; lpName
0040606F push    0               ; bInitialOwner
00406071 push    0               ; lpMutexAttributes
00406073 mov     var_main_zero, 0
0040607D call    esi ; CreateMutexW
0040607F mov     edi, ds:time
00406085 push    0               ; time_t *
00406087 mov     hUnnamedMutex, eax
```

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_437E40* (line 15) and if it is lower, the function quits.

```
0040608C call    edi ; time
0040608E add     esp, 4
00406091 cmp     eax, dword_437E40
00406097 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:



However the virtual address *0x437E40* is located in an uninitialized part of the *data* section of slave.exe and according to PE-COFF specification[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.

[22] Microsoft PE and COFF Specification https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx (last accessed 11.09.2015)

```
00406060 ; __stdcall wWinMain(x, x, x, x)
00406060 _wWinMain@16 proc near
00406060 push    esi
00406061 push    edi
00406062 call    f_Initialize_APIs ; loading API functions
00406067 mov     esi, ds:CreateMutexW
0040606D push    0                ; lpName
0040606F push    0                ; bInitialOwner
00406071 push    0                ; lpMutexAttributes
00406073 mov     var_main_zero, 0
0040607D call    esi ; CreateMutexW ; creation of unnamed mutex
0040607F mov     edi, ds:time
00406085 push    0                ; time_t *
00406087 mov     hUnnamedMutex, eax
0040608C call    edi ; time
0040608E add     esp, 4
00406091 cmp     eax, zero        ; comparing time() result to zero
00406097 jl      short func_exit
```

The next code block is quite interesting.

```
00406099 cmp     eax, 551B3500h
0040609E jg      short func_exit
```

If the *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 *time()* result is greater than this value, then main function quits. This means that the malware won't run after this date.

```
004060A0 push    offset Name      ; "__NTDLL_CORE__"
004060A5 push    0                ; bInitialOwner
004060A7 push    0                ; lpMutexAttributes
004060A9 call    esi ; CreateMutexW
004060AB cmp     eax, 0FFFFFFFFh
004060AE jz      short func_exit

004060B0 call    ds:GetLastError
004060B6 cmp     eax, 0B7h
004060BB jz      short func_exit
```

In the next two code blocks, the malware tries to create a named mutex *"__NTDLL_CORE__"* and checks if it succeeds. If *CreateMutexW* returns *INVALID_HANDLE_VALUE* (*0xFFFFFFFF*) or *GetLastError* returns *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.

```
004060BD call    sub_406120
004060C2 test    eax, eax
004060C4 jnz     short loc_4060CB

004060C6 call    sub_406410
```

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*).

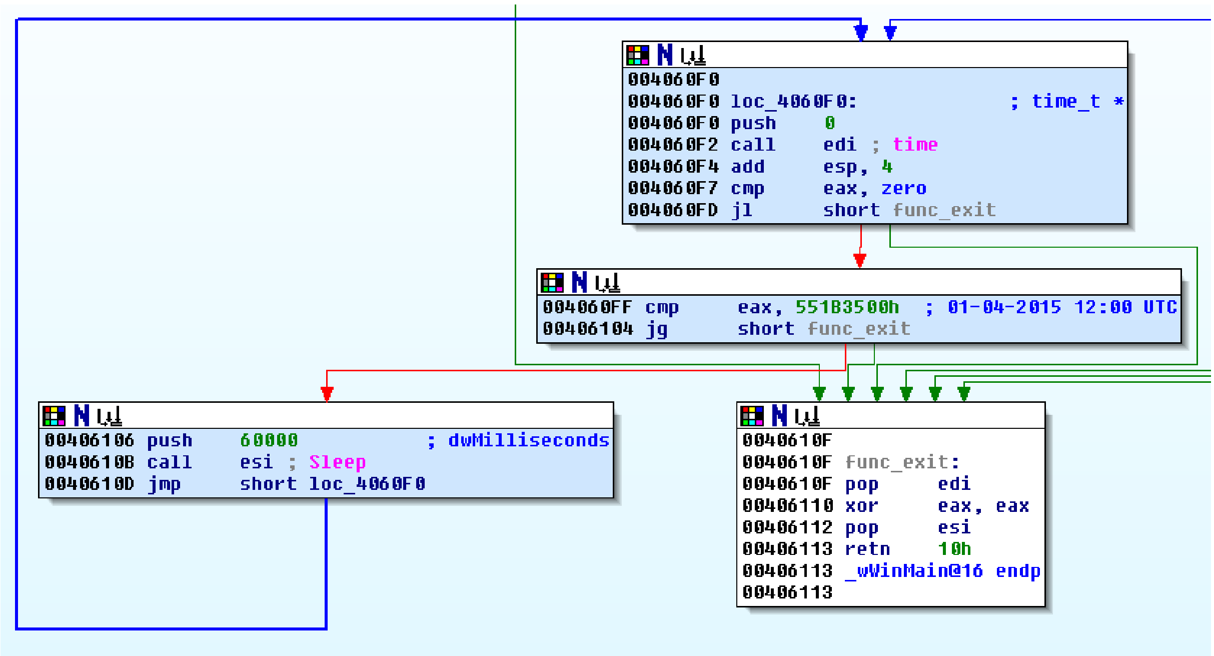
In one of the previous exercises, you already found that *sub_406410* is probably installation routine. Indeed if you take a look into it, there are calls to API functions such as: *CreateDirectoryW*, *CreateFileW*, *MoveFileExW*, *RegSetValueExW*, as well as references to strings such as *"Software\Microsoft\Windows\CurrentVersion\Run"*. Rename this function to *f_InstallRoutine*.

```
004067F3 push    0                   ; lpClass
004067F5 push    0                   ; Reserved
004067F7 push    offset SubKey       ; "Software\\Microsoft\\Windows\\CurrentVersi"...
004067FC push    [esp+40A4h+hKey]    ; hKey
00406800 call    ds:RegCreat ; char SubKey[]
00406806 mov     ebx, [esp+4 SubKey db 'Software\Microsoft\Windows\CurrentV'
0040680A lea     eax, [esi+e db 'ersion\Run',0
0040680D push    eax                 ; cbData
0040680E lea     eax, [esp+4088h+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.

```
004060CB
004060CB loc_4060CB:                 ; lpThreadId
004060CB push    0
004060CD push    0                   ; dwCreationFlags
004060CF push    0                   ; lpParameter
004060D1 push    offset sub_401B90   ; lpStartAddress
004060D6 push    0                   ; dwStackSize
004060D8 push    0                   ; lpThreadAttributes
004060DA call    ds:CreateThread
004060E0 push    eax                 ; hObject
004060E1 call    ds:CloseHandle
004060E7 mov     esi, ds:Sleep
004060ED 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*.

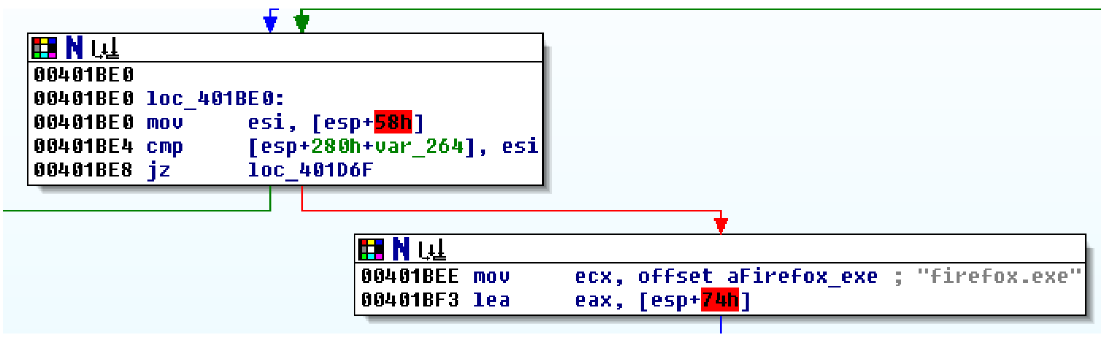


The next three blocks, create a loop. All the loop does is to check system time and compare it to previously checked date of 01 April 2015. If time is greater than this date, the program quits. Otherwise, the program sleeps one minute (60,000 milliseconds) and repeats checking the date.

4.3 Analysis of thread function

In this exercise you will do an analysis of the thread function (*f_ThreadFunction - 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:

```
00401BE0
00401BE0 loc_401BE0:
00401BE0 mov     esi, [esp+280h+pe.th32ProcessID]
00401BE4 cmp     [esp+280h+var_264], esi
00401BE8 jz      loc_401D6F
```

```
00401BEE mov     ecx, offset aFirefox_exe ; "firefox.exe"
00401BF3 lea     eax, [esp+280h+pe.szExeFile]
```

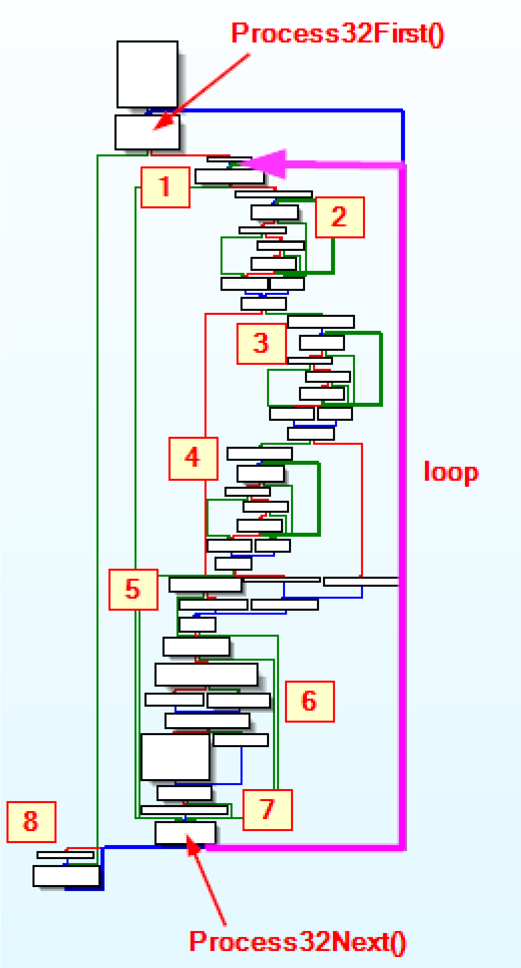
Starting analysis of a function, we see that the program first checks its own process ID and saves it to the local variable var_264 (rename it to PID):

```
00401B9E call    ds:GetCurrentProcessId
00401BA4 mov     ecx, eax
00401BA6 mov     [esp+280h+PID], eax
```

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

```
00401BB3
00401BB3 loc_401BB3:                 ; th32ProcessID
00401BB3 push    0
00401BB5 push    2                   ; dwFlags
00401BB7 mov     [esp+288h+pe.dwSize], 22Ch
00401BBF call    ds:CreateToolhelp32Snapshot
00401BC5 mov     edi, eax
00401BC7 lea     eax, [esp+280h+pe]
00401BCB push    eax                 ; lppe
00401BCC push    edi                 ; hSnapshot
00401BCD call    ds:Process32FirstW
00401BD3 test    eax, eax
00401BD5 jz      loc_401D8A
```

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*:

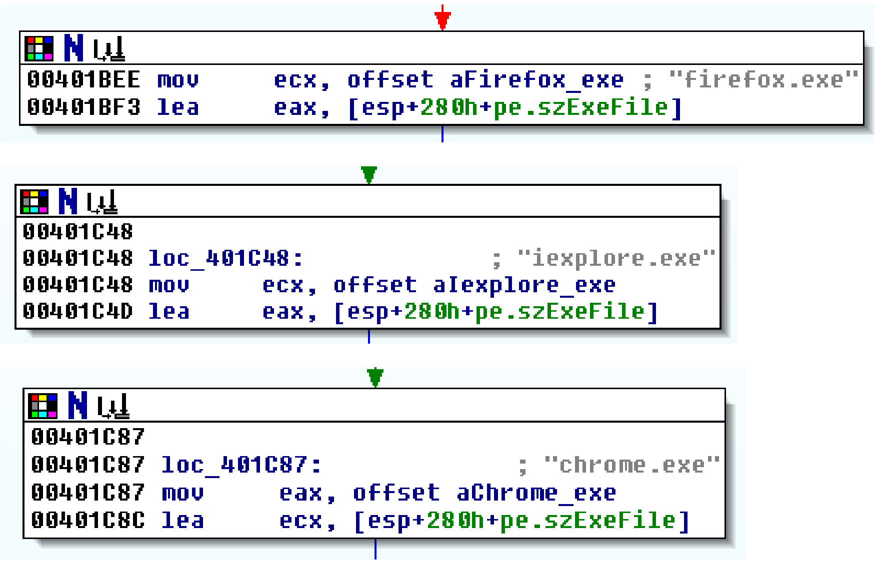
```
00401D6F
00401D6F get_proc_next:
00401D6F lea     eax, [esp+280h+pe]
00401D73 push    eax              ; lppe
00401D74 push    edi              ; hSnapshot
00401D75 call    ds:Process32NextW
00401D7B test    eax, eax
00401D7D jnz     loc_401BE0
```

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:

```
00401BE0
00401BE0 loc_401BE0:
00401BE0 mov     esi, [esp+280h+pe.th32ProcessID]
00401BE4 cmp     [esp+280h+PID], esi
00401BE8 jz      get_proc_next
```

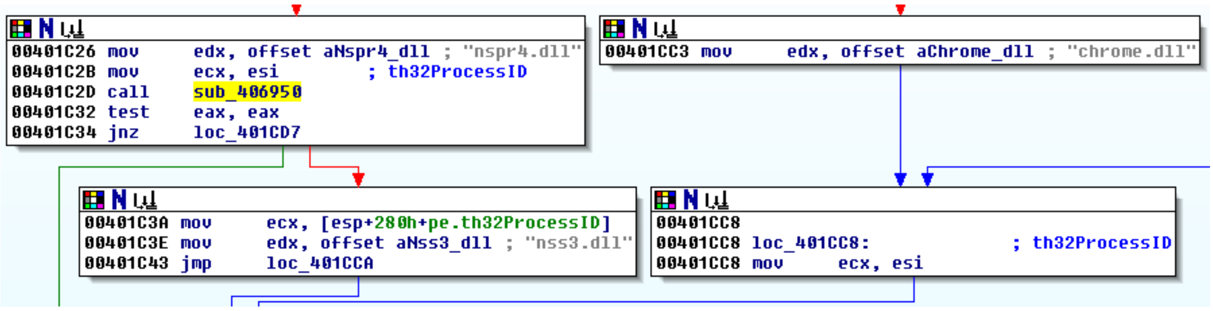
If both PIDs are equal, program skips loop iteration and tries to check the next process.

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*":

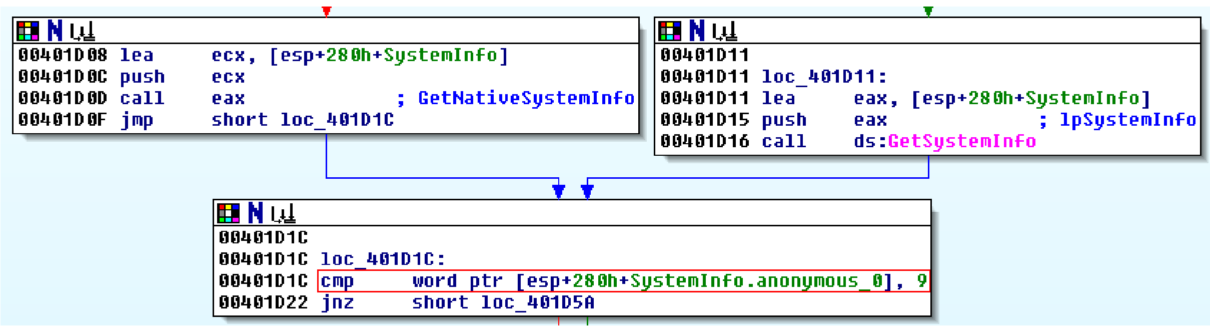


This means that malware is looking for processes of web browsers and it will probably try to inject into some code.

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:



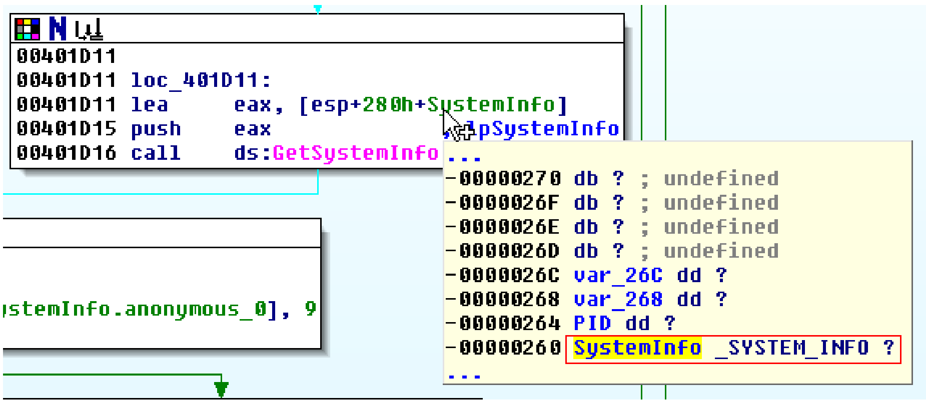
Names of DLLs are passed as a second argument to the *sub_406950* (*fastcall* calling convention). At this point you don't know what *sub_406950* is used for but a quick look at it might suggest it is only used to enumerate DLLs of web browser process to check if given library was loaded (calls to *CreateToolhelp32Snapshot*, *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].

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD  dwOemId;
    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*:

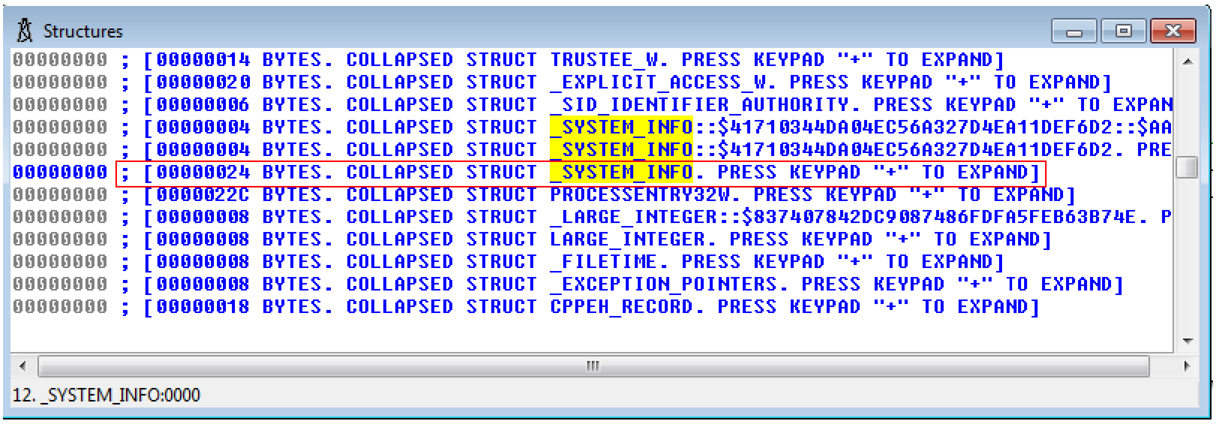


Here you can see this is a stack declared structure of *type _SYSTEM_INFO*.

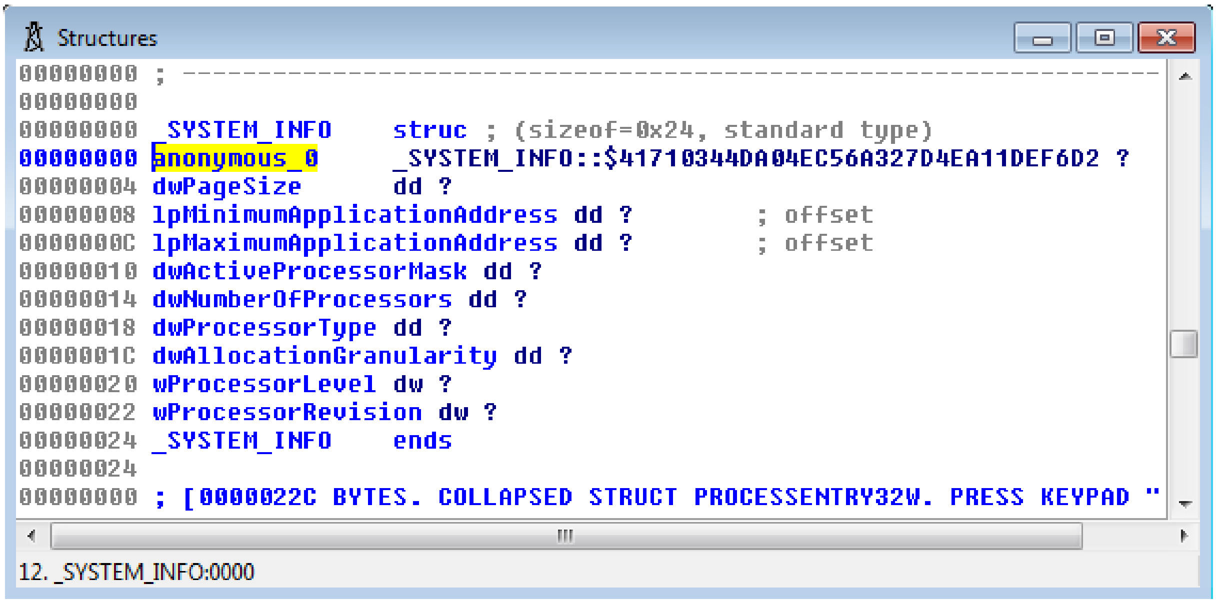
[23] GetSystemInfo function https://msdn.microsoft.com/en-us/library/windows/desktop/ms724381%28v=vs.85%29.aspx (last accessed 11.09.2015)
[24] GetNativeSystemInfo function https://msdn.microsoft.com/en-us/library/windows/desktop/ms724340%28v=vs.85%29.aspx (last accessed 11.09.2015)
[25] SYSTEM_INFO structure https://msdn.microsoft.com/en-us/library/windows/desktop/ms724958%28v=vs.85%29.aspx (last accessed 11.09.2015)

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*).

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD  dwOemId;
    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;
```

Indeed, value 9 to which *anonymous_0* field is compared represents AMD64 processor architecture[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).

```
00401D24 push    esi                    ; dwProcessId
00401D25 push    0                      ; bInheritHandle
00401D27 push    400h                   ; dwDesiredAccess
00401D2C call    ds:OpenProcess
00401D32 mov     esi, eax
00401D34 lea     eax, [esp+280h+var_26C]
00401D38 push    eax
00401D39 push    esi
00401D3A call    ds:IsWow64Process
00401D40 xor     ecx, ecx
00401D42 cmp     [esp+288h+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
```

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[27].

[26] SYSTEM_INFO structure https://msdn.microsoft.com/en-us/library/windows/desktop/ms724958%28v=vs.85%29.aspx (last accessed 11.09.2015)
[27] Windows subsystem allowing 32-bit applications running on 64-bit Windows system (https://msdn.microsoft.com/en-us/library/windows/desktop/aa384249%28v=vs.85%29.aspx) (last accessed 11.09.2015)

According to Microsoft documentation[28], *IsWow64Process* is a stdcall function taking two arguments.

```
BOOL WINAPI IsWow64Process(
  _In_  HANDLE hProcess,
  _Out_ PBOOL  Wow64Process
);
```

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

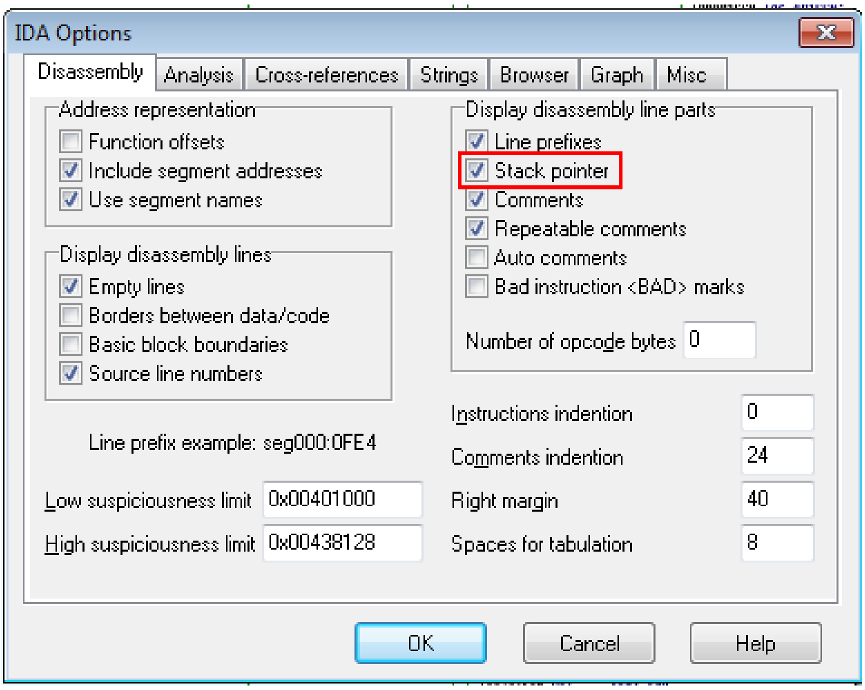
In the code, *Wow64Process* is set to the address of *var_26C* variable (*lea eax, [esp+280h+var_26C]*). After a call to *IsWow64Process* we would expect value returned in *var_26C* should be checked. But instead you see references to some other variable (*var_274*) which haven't been yet initialized or referenced.

```
00401D24 push    esi                  ; dwProcessId
00401D25 push    0                    ; bInheritHandle
00401D27 push    400h                 ; dwDesiredAccess
00401D2C call    ds:OpenProcess
00401D32 mov     esi, eax
00401D34 lea     eax, [esp+280h+var_26C]
00401D38 push    eax
00401D39 push    esi
00401D3A call    ds:IsWow64Process
00401D40 xor     ecx, ecx
00401D42 cmp     [esp+288h+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
```

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 *esp* based stack frame this might cause IDA to wrongly interpret variables. Let's check how IDA traced a stack pointer.

Choose *Options->General* and check the *Stack pointer* checkbox.

[28] IsWow64Process function https://msdn.microsoft.com/en-us/library/windows/desktop/ms684139%28v=vs.85%29.aspx (last accessed 11.09.2015)



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
00401D34 284 lea     eax, [esp+280h+var_26C]
00401D38 284 push    eax
00401D39 288 push    esi
00401D3A 28C call    ds:IsWow64Process
00401D40 28C xor     ecx, ecx
00401D42 28C cmp     [esp+288h+var_274], ecx
00401D46 28C push    esi             ; hObject
00401D47 290 setz    cl
00401D4A 290 mov     [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+288h+var_274], ecx
```

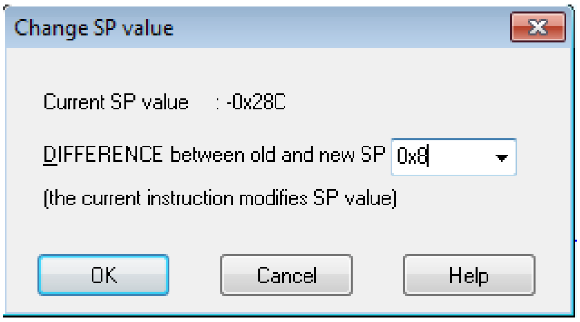
To see the reason for this, hover mouse over *IsWow64Process*.

```
00401D38 284 push    eax
00401D39 288 push    esi
00401D3A 28C call    ds:IsWow64Process
00401D40 28C xor     ecx, ecx
extrn IsWow64Process:dword
00401D46 28C push    esi              ; hObject
00401D47 290 setz    cl
```

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):

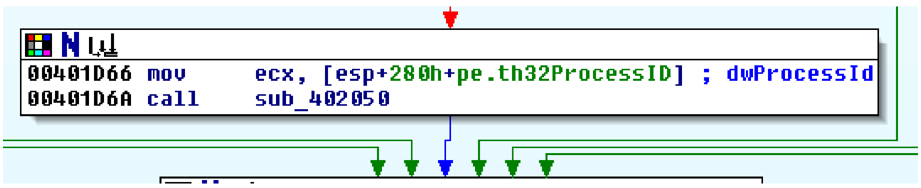


Now IDA should correctly reference all variables making code much clearer. Notice what was previously referenced as *var_274* is now *var_26C*:

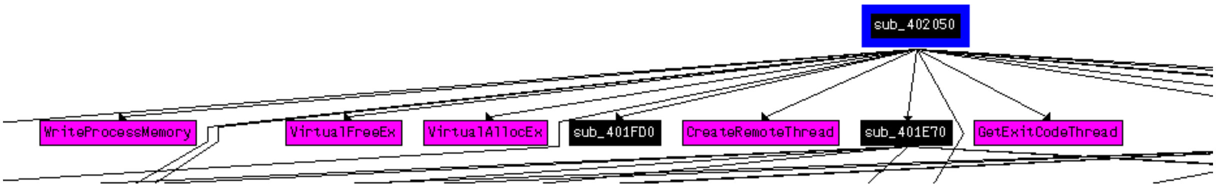
```
00401D32 284 mov     esi, eax
00401D34 284 lea     eax, [esp+280h+var_26C]
00401D38 284 push    eax
00401D39 288 push    esi
00401D3A 28C call    ds:IsWow64Process
00401D40 284 xor     ecx, ecx
00401D42 284 cmp     [esp+280h+var_26C], ecx
00401D46 284 push    esi              ; hObject
00401D47 288 setz    cl
00401D4A 288 mov     [esp+284h+var_26C], ecx
00401D4E 288 call    ds:CloseHandle
00401D54 284 mov     eax, [esp+280h+var_26C]
00401D58 284 jmp     short loc_401D60
```

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.

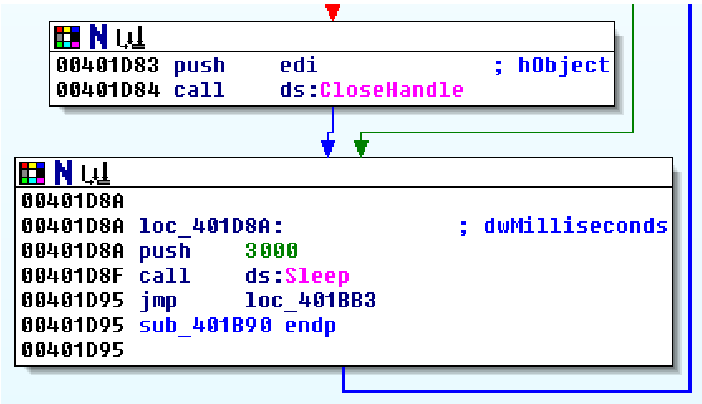
Going back to the thread function analysis, take a look at block [7] where the single function *sub_402050* is called just before loop end.



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.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000009A0	83	0D	84	80	43	00	FF	59	59	FF	15	70	01	41	00	8B
000009B0	0D	38	7E	43	00	89	08	FF	15	74	01	41	00	8B	0D	34
000009C0	7E	43	00	89	08	E8	B6	6A	00	00	83	3D	28	30	41	00
000009D0	00	75	0C	68	80	80	40	00	FF	15	78	01	41	00	59	E8
000009E0	8E	03	00	00	33	C0	C3	E8	34	04	00	00	E9	58	FD	FF
000009F0	FF	8B	FF	55	8B	EC	81	EC	28	03	00	00	A3	10	7C	43
00000A00	00	89	0D	0C	7C	43	00	89	15	08	7C	43	00	89	1D	04
00000A10	7C	43	00	89	35	00	7C	43	00	89	3D	FC	7B	43	00	66
00000A20	8C	15	28	7C	43	00	66	8C	0D	1C	7C	43	00	66	8C	1D
00000A30	F8	7B	43	00	66	8C	05	F4	7B	43	00	66	8C	25	F0	7B
00000A40	43	00	66	8C	2D	EC	7B	43	00	9C	8F	05	20	7C	43	00

Highlighted bytes represent consecutive assembly instructions:

E8 34 04 00 00: **call 0x401a20**
E9 58 FD FF 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.

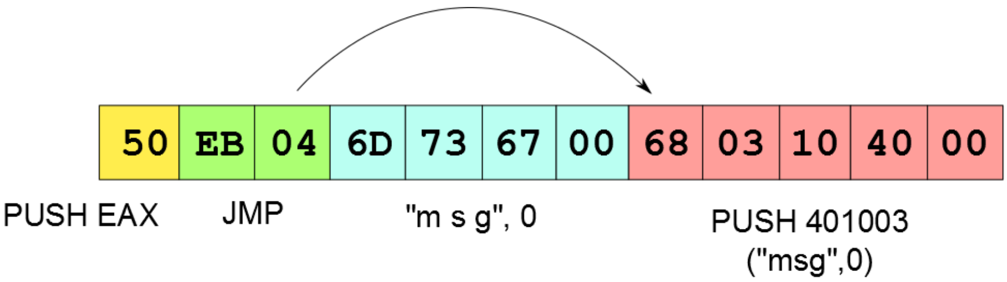
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000009A0	83	0D	84	80	43	00	FF	59	59	FF	15	70	01	41	00	8B
000009B0	0D	38	7E	43	00	89	08	FF	15	74	01	41	00	8B	0D	34
000009C0	7E	43	00	89	08	E8	B6	6A	00	00	83	3D	28	30	41	00
000009D0	00	75	0C	68	80	80	40	00	FF	15	78	01	41	00	59	E8
000009E0	8E	03	00	00	33	C0	C3	E8	34	04	00	00	E9	58	FD	FF
000009F0	FF	8B	FF	55	8B	EC	81	EC	28	03	00	00	A3	10	7C	43
00000A00	00	89	0D	0C	7C	43	00	89	15	08	7C	43	00	89	1D	04
00000A10	7C	43	00	89	35	00	7C	43	00	89	3D	FC	7B	43	00	66
00000A20	8C	15	28	7C	43	00	66	8C	0D	1C	7C	43	00	66	8C	1D
00000A30	F8	7B	43	00	66	8C	05	F4	7B	43	00	66	8C	25	F0	7B
00000A40	43	00	66	8C	2D	EC	7B	43	00	9C	8F	05	20	7C	43	00

Red frames mark previously disassembled instructions while highlighted bytes mark new instructions after disassembling with changed offset.

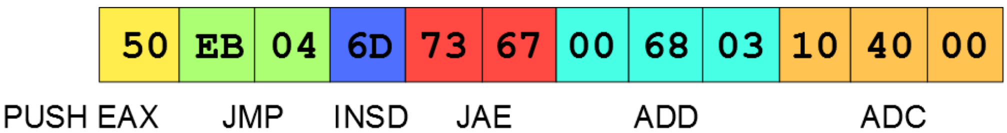
04 00: **add al, 0x0**
00 E9: **add cl, ch**
58: **pop eax**
FD: **std**
FF: **db 0xFF** (incorrect)
FF 8B FF 55 8B EC: **dec dword [ebx-0x1374aa01]**

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_4010B2
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.

```
.flat:0040101A
.flat:0040101A loc_40101A:                             ; CODE XREF: start↑p
.flat:0040101A                 push    ebp
.flat:0040101B                 mov     ebp, esp
.flat:0040101D                 call    $+5
.flat:00401022                 pop     eax
.flat:00401023                 add     eax, 10h
.flat:00401026                 call    eax
.flat:00401028                 inc     esi
.flat:00401029                 popa                          Junk code
.flat:0040102A                 outsb
.flat:0040102B                 jz      short near ptr loc_40108C+2
.flat:0040102D                 jnb     short near ptr loc_4010A1+2
.flat:0040102F                 imul    esp, [ebx+21h], 1337B8h
.flat:00401036                 add     [ecx+0C35DECh], cl
.flat:00401036 ; ---------------------------------------------------------------------------
.flat:0040103C                 dd 2 dup(0)
.flat:00401044                 db 0
.flat:00401045 ; ---------------------------------------------------------------------------
.flat:00401045
.flat:00401045 loc_401045:                             ; CODE XREF: start+5↑p
.flat:00401045                 push    ebp
.flat:00401046                 mov     ebp, esp
.flat:00401048                 xor     eax, eax
```

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*).

```
.flat:0040101D                 call    $+5
.flat:00401022                 pop     eax
```

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.

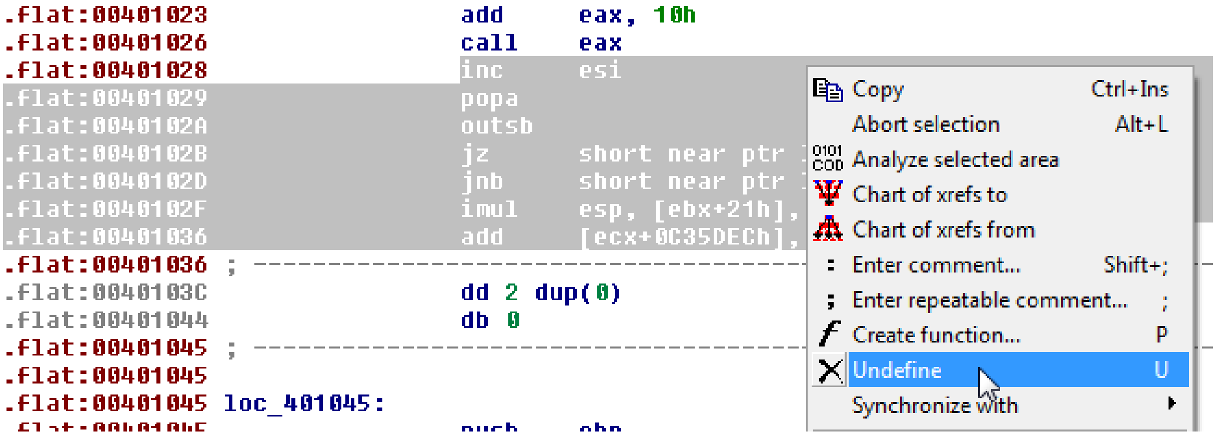
```
.flat:00401023                 add     eax, 10h
.flat:00401026                 call    eax
```

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.

```
.flat:00401028                 inc     esi
.flat:00401029                 popa                          Junk code
.flat:0040102A                 outsb
.flat:0040102B                 jz      short near ptr loc_40108C+2
.flat:0040102D                 jnb     short near ptr loc_4010A1+2
.flat:0040102F                 imul    esp, [ebx+21h], 1337B8h
.flat:00401036                 add     [ecx+0C35DECh], cl
```

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.

```
.flat:00401023                 add     eax, 10h
.flat:00401026                 call    eax
.flat:00401026 ; ---------------------------------------
.flat:00401028                 db  46h ; F
.flat:00401029                 db  61h ; a
.flat:0040102A                 db  6Eh ; n
.flat:0040102B                 db  74h ; t
.flat:0040102C                 db  61h ; a
.flat:0040102D                 db  73h ; s
.flat:0040102E                 db  74h ; t
.flat:0040102F                 db  69h ; i
.flat:00401030                 db  63h ; c
.flat:00401031                 db  21h ; !
.flat:00401032                 db 0B8h ; +
.flat:00401033                 db  37h ; 7
.flat:00401034                 db  13h
.flat:00401035                 db    0
.flat:00401036                 db    0
.flat:00401037                 db  89h ; ë
.flat:00401038                 db 0ECh ; 8
.flat:00401039                 db  5Dh ; ]
.flat:0040103A                 db 0C3h ; +
.flat:0040103B                 db    0
.flat:0040103C                 dd 2 dup(0)
.flat:00401044                 db 0
```

Now the code should be much clearer. You can also read return value of *loc_40101A* which is *0x1337*.

```
.flat:0040102F                 db  69h ; i
.flat:00401030                 db  63h ; c
.flat:00401031                 db  21h ; !
.flat:00401032 ; ------------------------------------
.flat:00401032                 mov     eax, 1337h
.flat:00401037                 mov     esp, ebp
.flat:00401039                 pop     ebp
.flat:0040103A                 retn
.flat:0040103A ; ------------------------------------
.flat:0040103B                 db    0
.flat:0040103C                 dd 2 dup(0)
.flat:00401044                 db 0
```

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*.

```
.flat:00401045 loc_401045:                           ; CODE XREF: start+5↑p
.flat:00401045                 push    ebp
.flat:00401046                 mov     ebp, esp
.flat:00401048                 xor     eax, eax
.flat:0040104A
.flat:0040104A loc_40104A:                           ; CODE XREF: .flat:00401050↓j
.flat:0040104A                 push    11EBh
.flat:0040104F                 pop     eax
.flat:00401050                 jz      short near ptr loc_40104A+1
.flat:00401052                 add     eax, 1000h
.flat:00401057
.flat:00401057 loc_401057:                           ; CODE XREF: .flat:00401063↓j
.flat:00401057                 mov     esp, ebp
.flat:00401059                 pop     ebp
.flat:0040105A                 retn
.flat:0040105B ; ------------------------------------------------------------
.flat:0040105B                 adc     esi, [edi]
.flat:0040105D                 adc     [eax+4096h], bh
.flat:00401063                 jmp     short loc_401057
```

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.

```
.flat:0040104A loc_40104A:                           ; CODE XREF: .flat:00401050↓j
.flat:0040104A                 push    11EBh
.flat:0040104F                 pop     eax
.flat:00401050                 jz      short near ptr loc_40104A+1
.flat:00401052                 add     eax, 1000h
```

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>).

```
.flat:0040104A loc_40104A:                              ; CODE XREF: .flat:00401050↓j
.flat:0040104A                 push    11EBh
.flat:0040104F                 pop     eax
.flat:00401050                 jz      short near ptr loc_40104A+1
.flat:00401052                 add     eax, 1000h
```

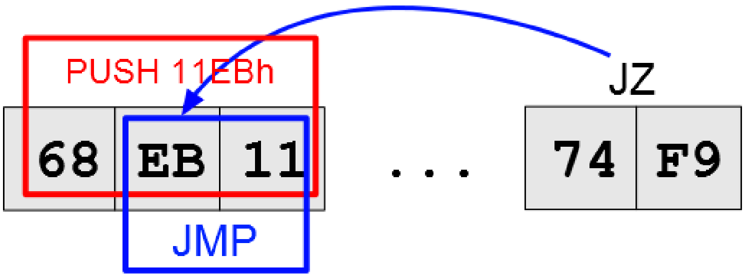
IDA will likely undefine more code than you intended, but this isn't a problem since you already know the *jz* destination address (*0x40104B*) and where the original *jz* instruction was located (*0x401050*).

```
.flat:00401048                 xor     eax, eax
.flat:00401048 ; ---------------------------------------------------------------------------
.flat:0040104A                 db  68h ; h
.flat:0040104B                 db 0EBh ; d          jz destination
.flat:0040104C                 db  11h
.flat:0040104D                 db    0
.flat:0040104E                 db    0
.flat:0040104F                 db  58h ; X
.flat:00401050                 db  74h ; t          undefined jz instruction
.flat:00401051                 db 0F9h ; ·
.flat:00401052                 db    5
.flat:00401053                 db    0
.flat:00401054                 db  10h
.flat:00401055                 db    0
.flat:00401056                 db    0
.flat:00401057 ; ---------------------------------------------------------------------------
.flat:00401057
.flat:00401057 loc_401057:                             ; CODE XREF: .flat:00401063↓j
.flat:00401057                 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:

```
.flat:00401048 ; ---------------------------------------------------------------------------
.flat:0040104A                 db  68h ; h
.flat:0040104B ; ---------------------------------------------------------------------------
.flat:0040104B
.flat:0040104B loc_40104B:                             ; CODE XREF: .flat:00401050↓j
.flat:0040104B                 jmp     short near ptr loc_40105D+1
.flat:0040104B ; ---------------------------------------------------------------------------
.flat:0040104D                 db    0
.flat:0040104E                 db    0
.flat:0040104F                 db  58h ; X
.flat:00401050 ; ---------------------------------------------------------------------------
.flat:00401050                 jz      short loc_40104B
.flat:00401052                 add     eax, 1000h
.flat:00401057
.flat:00401057 loc_401057:                             ; CODE XREF: .flat:00401063↓j
.flat:00401057                 mov     esp, ebp
.flat:00401059                 pop     ebp
.flat:0040105A                 retn
.flat:0040105B ; ---------------------------------------------------------------------------
.flat:0040105B                 adc     esi, [edi]
.flat:0040105D
.flat:0040105D loc_40105D:                             ; CODE XREF: .flat:loc_40104B↑j
.flat:0040105D                 adc     [eax+4096h], bh
.flat:00401063                 jmp     short loc_401057
```

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:

```
.flat:00401052                 add     eax, 1000h
.flat:00401057
.flat:00401057 loc_401057:                             ; CODE XREF: .flat:00401063↓j
.flat:00401057                 mov     esp, ebp
.flat:00401059                 pop     ebp
.flat:0040105A                 retn
.flat:0040105B ; ---------------------------------------------------------------------------
.flat:0040105B                 adc     esi, [edi]          garbage bytes
.flat:0040105B ; ---------------------------------------------------------------------------
.flat:0040105D                 db  10h
.flat:0040105E ; ---------------------------------------------------------------------------
.flat:0040105E
.flat:0040105E loc_40105E:                             ; CODE XREF: .flat:loc_40104B↑j
.flat:0040105E                 mov     eax, 4096h
.flat:00401063                 jmp     short loc_401057
```

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:

```
.flat:00401045 loc_401045:                             ; CODE XREF: start+5↑p
.flat:00401045                 push    ebp
.flat:00401046                 mov     ebp, esp
.flat:00401048                 xor     eax, eax
.flat:0040104A
.flat:0040104A loc_40104A:                             ; CODE XREF: .flat:00401050↓j
.flat:0040104A                 push    11EBh
.flat:0040104F            1    pop     eax
.flat:00401050                 jz      short near ptr loc_40104A+1
.  never executed              add     eax, 1000h
.flat:00401057
.flat:00401057 loc_401057:                             ; CODE XREF: .flat:00401063↓j
.flat:00401057                 mov     esp, ebp
.flat:00401059                 pop     ebp                              2
.flat:0040105A                 retn
.flat:0040105B ; ---------3-------------------------------------------------------
.flat:0040105B                 adc     esi, [edi]
.flat:0040105D                 adc     [eax+4096h], bh
.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:

```
00401065
00401065
00401065 ; Attributes: bp-based frame
00401065
00401065 sub_401065 proc near
00401065 push    ebp
00401066 mov     ebp, esp
00401068 xor     eax, eax
0040106A push    1000h
0040106F call    sub_40107D
00401074 add     eax, 1000h
00401079 mov     esp, ebp
0040107B pop     ebp
0040107C retn
0040107C sub_401065 endp
0040107C
```

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*:

```
0040107D
0040107D
0040107D ; Attributes: bp-based frame
0040107D
0040107D sub_40107D proc near
0040107D
0040107D arg_0= dword ptr  8
0040107D
0040107D push    ebp
0040107E mov     ebp, esp
00401080 mov     eax, [ebp+arg_0]
00401083 add     eax, 1000h
00401088 lea     edx, [ebp+arg_0]
0040108B sub     edx, 4
```

```
0040108E
0040108E loc_40108E:
0040108E add     dword ptr [edx], 2Bh
00401094 mov     esp, ebp
00401096 pop     ebp
00401097 retn    4
00401097 sub_40107D endp
00401097
```

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:

```
00401083 add     eax, 1000h
00401088 lea     edx, [ebp+arg_0]
0040108B sub     edx, 4
```

```
0040108E
0040108E loc_40108E:
0040108E add     dword ptr [edx], 2Bh
00401094 mov     esp, ebp
00401096 pop     ebp
00401097 retn    4
```

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*:

```
00401068 xor     eax, eax
0040106A push    1000h
0040106F call    sub_40107D
00401074 add     eax, 1000h
00401079 mov     esp, ebp
0040107B pop     ebp
```

The original return address should be *0x401074*. But you know it was increased by *0x2B*. This means that function *sub_40107D* will return to the address ***0x40109F*** (*0x401074+0x2B*). Switch from graph view to the text view and search for this address.

```
.flat:00401097                  retn    4
.flat:00401097 sub_40107D       endp
.flat:00401097
.flat:0040109A ; ---------------------------------------------------------------------
.flat:0040109A                  push    ebp                                  junk code
.flat:0040109B                  mov     ebp, esp
.flat:0040109D                  xchg    ah, [esi+0C0DEB8h]
.flat:004010A3
.flat:004010A3 loc_4010A3:                              ; CODE XREF: .flat:0040102D↑j
.flat:004010A3                  add     [ecx+0C35DECh], cl
.flat:004010A3 ; ---------------------------------------------------------------------
.flat:004010A9                  db 3 dup(0)
.flat:004010AC                  dd 0
.flat:004010B0                  db 2 dup(0)
.flat:004010B2
```

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

```
.flat:0040109D                  db  86h ; å
.flat:0040109E                  db 0A6h ; ª
.flat:0040109F ; ---------------------------------------------------------------------
.flat:0040109F                  mov     eax, 0C0DEh      ; CODE XREF: .flat:0040102D↑j
.flat:004010A4                  mov     esp, ebp
.flat:004010A6                  pop     ebp
.flat:004010A7                  retn
.flat:004010A7 ; ---------------------------------------------------------------------
.flat:004010A8                  db    0
```

You have just found final *eax* value which is *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 sub_4010B2

Now you will analyse a call to subroutine *sub_4010B2*.

```
004010B2
004010B2
004010B2 ; Attributes: bp-based frame
004010B2
004010B2 sub_4010B2 proc near
004010B2 push     ebp
004010B3 mov      ebp, esp
004010B5 xor      eax, eax
004010B7 push     eax
004010B8 mov      eax, 40000h
004010BD add      eax, 143ABE3h
004010C2 pop      eax
004010C3 push     ecx
004010C4 push     edx
004010C5 mov      ecx, 52Ah
004010CA add      ecx, 7
004010CD xchg     ecx, edx
004010CF xor      ecx, edx
```

…

```
00401150 pop      eax
00401151 push     eax
00401152 mov      eax, 128h
00401157 add      eax, 2710h
0040115C pop      eax
0040115D push     eax
0040115E mov      eax, 699h
00401163 add      eax, 0EA60h
00401168 pop      eax
00401169 mov      esp, ebp
0040116B pop      ebp
0040116C retn
0040116C sub_4010B2 endp
0040116C
```

If you go to this function you will see a long disassembled code with many operations on the *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).

```
004010B7 push     eax
004010B8 mov      eax, 40000h
004010BD add      eax, 143ABE3h
004010C2 pop      eax

004010C3 push     ecx
004010C4 push     edx
004010C5 mov      ecx, 52Ah
004010CA add      ecx, 7
004010CD xchg     ecx, edx
004010CF xor      ecx, edx
004010D1 pop      ecx
004010D2 pop      edx
004010D3 xchg     ecx, edx

004010D5 inc      ecx
004010D6 dec      ecx

004010F4 push     eax
004010F5 push     2000h
004010FA push     ecx
004010FB add      esp, 12
```

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:

```
004010B2 push    ebp
004010B3 mov     ebp, esp
004010B5 xor     eax, eax
004010B7 push    eax
004010B8 mov     eax, 40000h
004010BD add     eax, 143ABE3h
         . . .
004010F4 push    eax
004010F5 push    2000h
004010FA push    ecx
004010FB add     esp, 12
004010FE mov     eax, 1000h
00401103 push    ecx
00401104 push    edx
00401105 mov     ecx, 52Ah
0040110A add     ecx, 7

00401130 pop     eax
00401131 pop     eax
00401132 inc     edx
00401133 dec     edx
00401134 add     eax, 500h
00401139 push    eax
0040113A mov     eax, 100h
0040113F add     eax, 0C8h
00401144 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*:

```
00401160
0040116D
0040116D ; Attributes: bp-based frame
0040116D
0040116D sub_40116D proc near
0040116D
0040116D var_4= dword ptr -4
0040116D
0040116D push    ebp
0040116E mov     ebp, esp
00401170 and     ecx, 0
00401173 push    15232A1h
00401178 push    large dword ptr fs:0
0040117F mov     large fs:0, esp
00401186 xor     [esp+8+var_4], 1122300h
0040118E mov     dword ptr [ecx], 0
00401194 mov     eax, 0EBFEh
00401199 mov     esp, ebp
0040119B pop     ebp
0040119C retn
0040119C sub_40116D endp
0040119C
```

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[29].

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

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

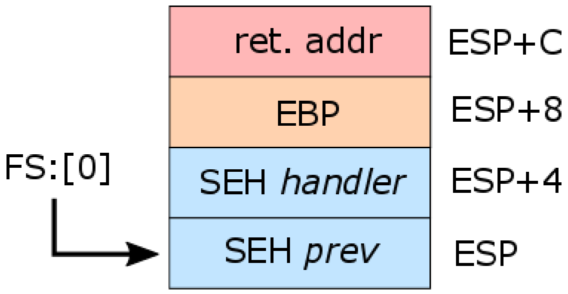
This structure consists of two fields. The first field (*prev*) is a pointer to the next EXCEPTION REGISTRATION structure while the second field (*handler*) is a pointer to exception handler function.

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 (last accessed 11.09.2015)



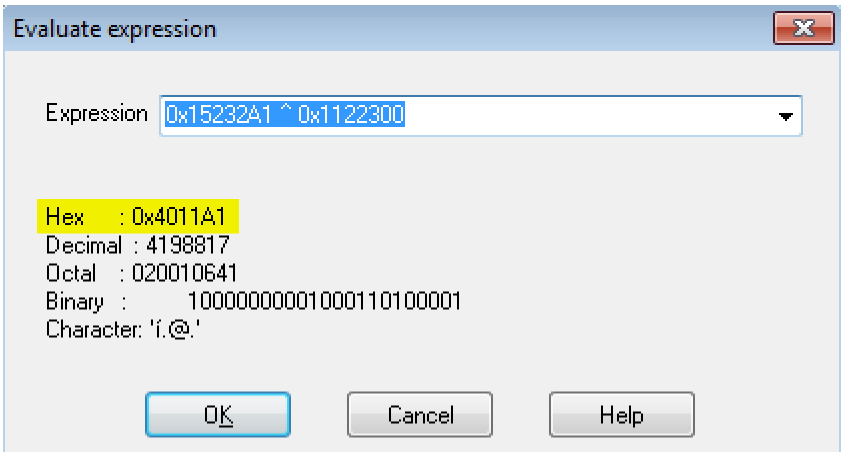
The next question should be whether any exception is triggered in this function? Yes, take a look at the *ecx* register: First, it is zeroed and then the program tries to write a DWORD value to the address pointed by this register. However, because *ecx* points to unallocated address *0x00000000* this will cause an exception (STATUS_ACCESS_VIOLATION – 0xC0000005) and program execution would be switched to the installed exception handler.

```
00401170 and     ecx, 0
00401173 push    15232A1h
00401178 push    large dword ptr fs:0
0040117F mov     large fs:0, esp
00401186 xor     dword ptr [esp+4], 1122300h
0040118E mov     dword ptr [ecx], 0
00401194 mov     eax, 0EBFEh
```

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*:

```
.flat:0040119C sub_40116D      endp
.flat:0040119C
.flat:0040119C ; ---------------------------------------------------------------------------
.flat:0040119D                 db 65h, 68h, 6Ch
.flat:004011A0                 dd 512B86Fh, 648B0000h, 0C4830824h, 0ECEB08h, 14h dup(0)
.flat:004011A0 _flat           ends
.flat:004011A0
```

Repeat steps from previous exercises to convert data at *0x4011A1* to code:

```
.flat:0040119B loc_40119B:                              ; CODE XREF: .flat:004011AD↓j
.flat:0040119B                 pop     ebp
.flat:0040119C                 retn
.flat:0040119C sub_40116D      endp
.flat:0040119C
.flat:0040119C ; ---------------------------------------------------------------------
.flat:0040119D                 db  65h ; e
.flat:0040119E                 db  68h ; h
.flat:0040119F                 db  6Ch ; l
.flat:004011A0                 db  6Fh ; o
.flat:004011A1 ; ---------------------------------------------------------------------
.flat:004011A1                 mov     eax, 512h
.flat:004011A6                 mov     esp, [esp+8]
.flat:004011AA                 add     esp, 8
.flat:004011AD                 jmp     short loc_40119B
.flat:004011AD ; ---------------------------------------------------------------------
.flat:004011AF                 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

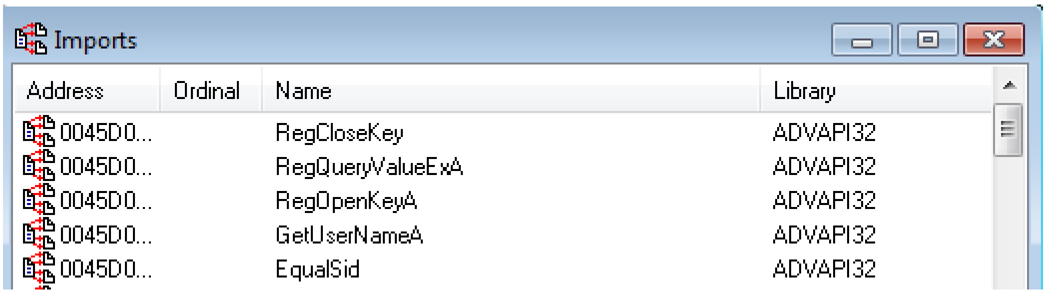
In this training, students had the opportunity to learn various aspects of advanced static analysis using IDA Free. First they learnt how to use IDA and what features it offers. Then they learnt how to find significant parts in disassembled code and how to analyse functions. Finally, students reviewed 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.

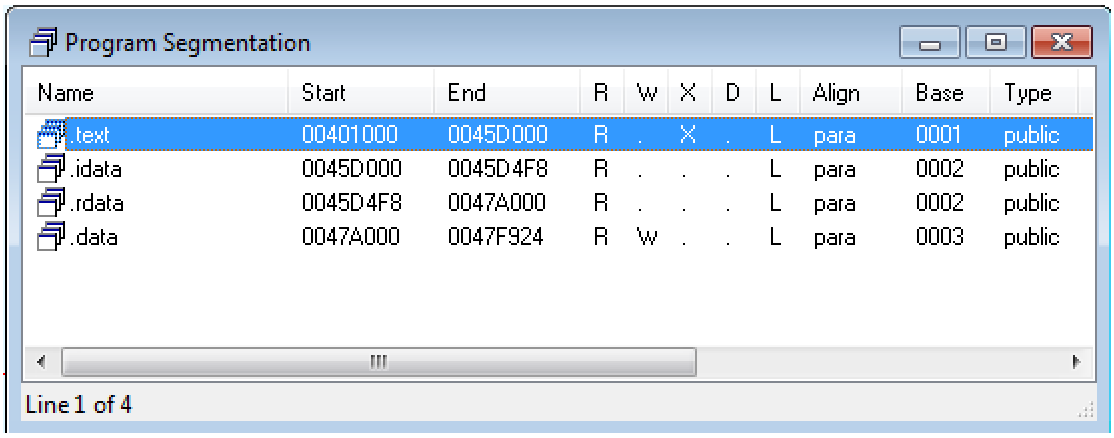
Click *View->Open subviews->Imports*:



Address	Ordinal	Name	Library
0045D0...		RegCloseKey	ADVAPI32
0045D0...		RegQueryValueExA	ADVAPI32
0045D0...		RegOpenKeyA	ADVAPI32
0045D0...		GetUserNameA	ADVAPI32
0045D0...		EqualSid	ADVAPI32

What sections are present within executable?

Click *View->Open subviews->Segments*:



Name	Start	End	R	W	X	D	L	Align	Base	Type
.text	00401000	0045D000	R	.	X	.	L	para	0001	public
.idata	0045D000	0045D4F8	R	.	.	.	L	para	0002	public
.rdata	0045D4F8	0047A000	R	.	.	.	L	para	0002	public
.data	0047A000	0047F924	R	W	.	.	L	para	0003	public

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”.

- There are many error messages also suggesting PuTTY capabilities.

```
"..." .rdata:00463C84    00000024    C    Proxy error: Unexpected proxy error
"..." .rdata:00463CA8    00000053    C    Proxy error: Server chose username/password authentication but we didn't o...
"..." .rdata:00463CFC    00000034    C    Proxy error: We don't support GSSAPI authentication
"..." .rdata:00463D30    0000003E    C    Proxy error: SOCKS proxy returned unrecognised address format
"..." .rdata:00463D70    00000021    C    Unrecognised SOCKS error code %d

"..." .rdata:00467578    00000025    C    Using CryptoCard authentication.%s%s
"..." .rdata:004675A0    0000001E    C    SSH CryptoCard authentication
"..." .rdata:004675C0    0000001E    C    Received CryptoCard challenge
"..." .rdata:004675E0    0000002D    C    CryptoCard challenge packet was badly formed

"..." .rdata:0046EF84    0000000D    C    HMAC-SHA-256
"..." .rdata:0046EF94    0000000E    C    hmac-sha2-256
"..." .rdata:0046EFA4    00000008    C    SHA-256

"..." .data:0047A7B8     0000000D    C    Release 0.65
"..." .data:0047A7C8     00000013    C    PuTTY-Release-0.65
```

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:00430E89↑j
.text:00430EAB          cmp     dword ptr [ebx+4030h], 2Eh
.text:00430EB2          jb      loc_430FEB
.text:00430EB8          push    2Eh
.text:00430EBA          lea     esi, [ebx+20h]
.text:00430EBD          push    offset aSshconnectio_0 ; "SSHCONNECTION@putty.p
.text:00430EC2          push    esi
```

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.

```
0044D2DA call    ds:GetCurrentProcessId
0044D2E0 mov     [ebp+var_8], eax
0044D2E3 lea     eax, [ebp+var_8]
```

Try to rename remaining locations: loc_44D2B1, loc_44D2DA, loc_44D36B, loc_44D3B4. What names would you suggest for them?

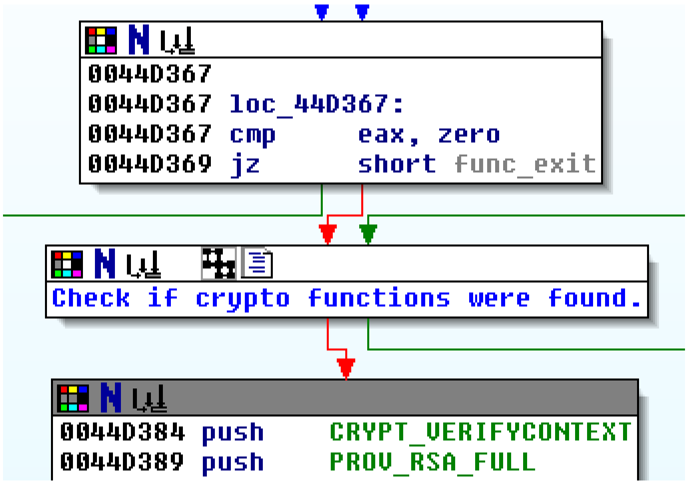
loc_44D2B1 – file_loop, file_iteration, …

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 dada). Then this data is passed always to the same function (ebx).

```
0044D2DA call    ds:GetCurrentProcessId
0044D2E0 mov     [ebp+var_8], eax
0044D2E3 lea     eax, [ebp+var_8]
0044D2E6 push    4                 ; var_8 size (DWORD)
0044D2E8 push    eax               ; ptr to var_8 containing process id
0044D2E9 call    ebx               ; call to func_ptr
```

```
0044D3AA lea     eax, [ebp+pbBuffer]
0044D3AD push    32                ; random data block size
0044D3AF push    eax               ; ptr to random data block (pbBuffer)
0044D3B0 call    ebx               ; func_ptr
```

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.

```
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).

sub_403BD0 – calls to ReadProcessMemory, CreateProcess32Snapshot, Process32First, Process32Next.

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".

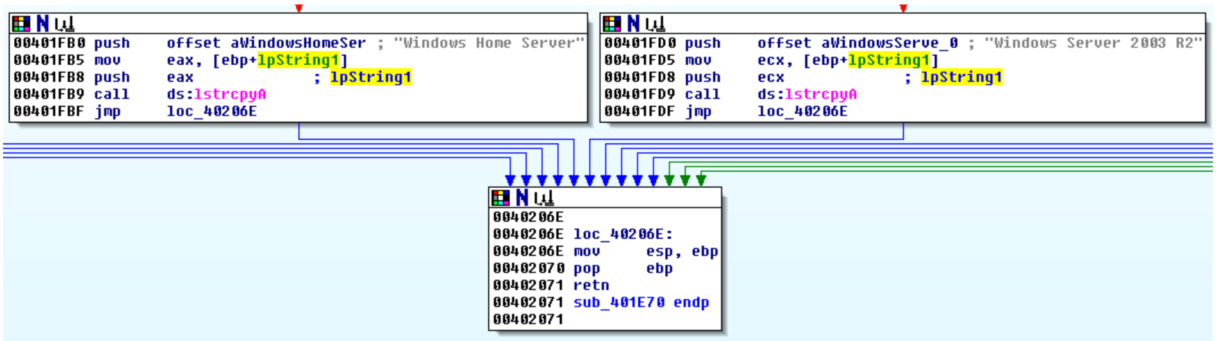
start (0x4036B0) – start routine.

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.

```
00401E70 ; int __cdecl sub_401E70(LPSTR lpString1)
00401E70 sub_401E70 proc near
00401E70
00401E70 SystemInfo= _SYSTEM_INFO ptr -0CCh
00401E70 VersionInformation= _OSVERSIONINFOA ptr -0A8h
00401E70 var_10= word ptr -10h
00401E70 var_E= byte ptr -0Eh
00401E70 var_4= dword ptr -4
00401E70 lpString1= dword ptr  8
00401E70
```



sub_402620 – what are function arguments and how are they used?

Function takes three arguments (renamed on the screenshot for clarity):

```
00402620 ; int __cdecl sub_402620(LPCSTR lpString1,LPCSTR lpString2,LPSTR lpString3)
00402620 sub_402620 proc near
00402620
00402620 lpMem= dword ptr -8
00402620 var_4= dword ptr -4
00402620 lpString1= dword ptr  8
00402620 lpString2= dword ptr  0Ch
00402620 lpString3= dword ptr  10h
00402620
```

All three arguments were recognized by IDA as string pointers.

lpString2 (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 lpString2.

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

```
00402686 mov     eax, [ebp+lpString1]
00402689 push    eax              ; lpString1
0040268A mov     ecx, [ebp+lpString3]
0040268D push    ecx              ; lpString3
0040268E call    ds:lstrcatA      ; concatenate lpString1 to lpString3
00402694 mov     edx, [ebp+lpMem]
00402697 push    edx              ; lpMem
00402698 mov     eax, [ebp+lpString3]
0040269B push    eax              ; lpString3
0040269C call    ds:lstrcatA      ; 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:

arg3 += arg1 + f(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".

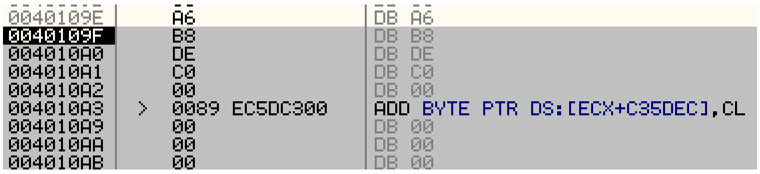
Function itself is called from sub_402710 which, as it was already found out, is a function used to communicate with C&C server.

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.

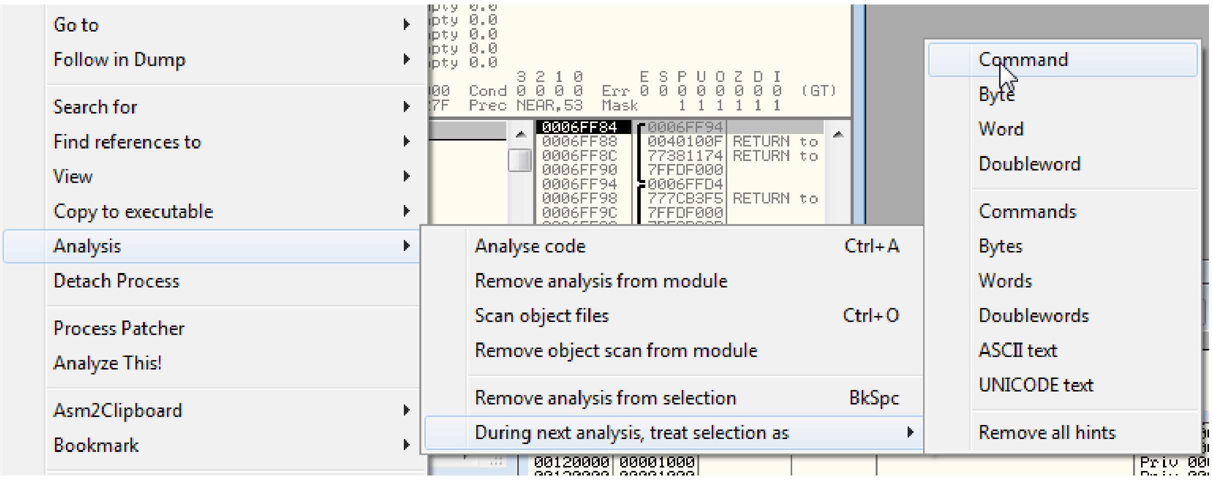
Exercise 6.4

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:



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:

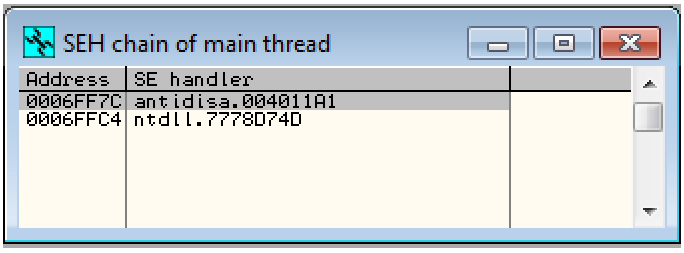
```
0040109E |    A6              |DB A6
0040109F      B8 DEC00000      MOV EAX,0C0DE
004010A4      89EC             MOV ESP,EBP
004010A6      5D               POP EBP
004010A7      C3               RETN
004010A8      0000             ADD BYTE PTR DS:[EAX],AL
004010AA      0000             ADD BYTE PTR DS:[EAX],AL
004010AC |  ? 0000             ADD BYTE PTR DS:[EAX],AL
```

Special attention is only required in last function (*0x40116D*) which uses Structured Exception Handlers (SEH) to hide some code.

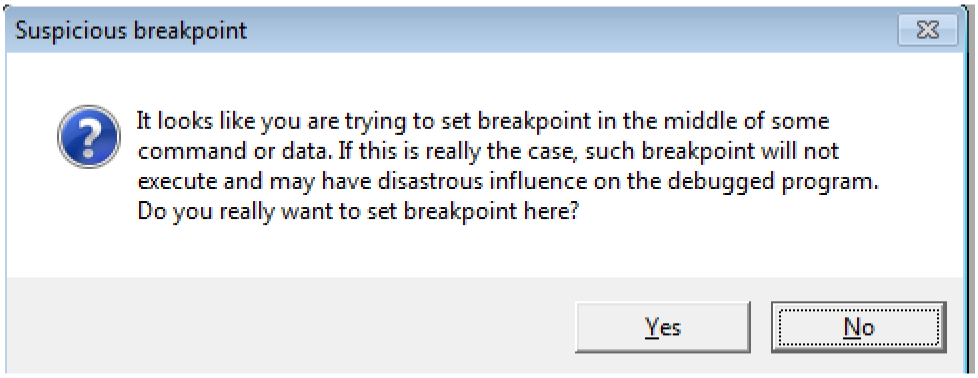
When you hit the instruction at which exception occurs (*at 0x40118E*) OllyDbg would stop and inform you at status bar that access violation exception has occurred:

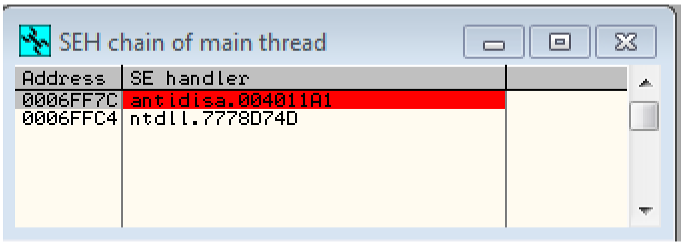
```
0040117F  .  64:8925 00000000  MOV DWORD PTR FS:[0],ESP
00401186  .  817424 04 002312  XOR DWORD PTR SS:[ESP+4],1122300
0040118E  .  C701 00000000     MOV DWORD PTR DS:[ECX],0
00401194  .  B8 FEEB0000       MOV EAX,0EBFE
00401199  .  89EC              MOV ESP,EBP
0040119B  >  5D                POP EBP
0040119C  .  C3                RETN
```

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:

```
004011A1   ?  B8 12050000    MOV EAX,512
004011A6   ?  8B6424 08      MOV ESP,DWORD PTR SS:[ESP+8]
004011AA   .  83C4 08        ADD ESP,8
004011AD   .^ 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.

ENISA
European Union Agency for Network
and Information Security
Science and Technology Park of Crete (ITE)
Vassilika Vouton, 700 13, Heraklion, Greece

Athens Office
1 Vass. Sofias & Meg. Alexandrou
Marousi 151 24, Athens, Greece

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