Global Information Assurance Certification Paper

Basic Reverse Engineering with Immunity Debugger

GIAC (GSEC) Gold Certification

Author: Roberto Nardella, roberto.nardella@fastwebnet.it

Advisor: Richard Carbone

Accepted: May 7, 2016

Abstract

Reverse Engineering is an intriguing art, but also one of the most difficult topics in Security and Malware Analysis. Skilled reverse engineers have an in-depth knowledge of Assembly language, of processor architectures and a great familiarity with the most important debuggers. However, there is a lot of information that can be gathered with an even essential knowledge of debuggers and Assembler. This paper shows some very basic, but very useful, reverse engineering steps carried out with a great debugger, Immunity Debugger.

1. Introduction

The intention of this technical paper is not to enter into the technical specifics of Assembly language, which is a skill that it is strongly required in order to perform in-depth reverse engineering. Instead, it is show basic and easy steps that are possible to perform with Immunity Debugger and that can return many useful indications at the same time. For a good understanding of this paper, a basic knowledge of the C programming language for Windows (WinAPI) is also required.

The example code snippets used in this paper are meant to resemble, as much as possible, some "real life" code snippets, emulating the actions that best part of malware do, like downloading files from the internet on the targeted machine, renaming extensions, writing files on disk, add some persistence mechanisms by modifying or creating registry keys.

1.2 Main assumptions

All the source codes shown in this paper have been compiled with "Pelles C" free C compiler. The settings used for the compilation of the executables are shown in Appendix A of this paper. All source code has been compiled under an x86 32-bit environment.

Although the purpose of this paper is to show the potential of basic debugging the technical specifics of Assembly language are out of scope, although a description of the main Intel processor registers cannot be avoided. To keep the difficulty of reading this paper to a minimum, the following two tables briefly describe the purpose of the main x8086 general purpose registers and pointers registers (also called "index registers"). A table describing the Segment Registers is deliberately avoided, as no such registers will be cited in the examples looked at in this paper. The initial "E" in the acronyms below (EAX, EBX, etc.) stands for "Extended" or "Enhanced."

Roberto Nardella, Roberto.nardella@fastwebnet.it

Table 1: General Purpose Registers	
(name)	**(description)**
EAX	**Accumulator Register. This register is generally used to store temporary data, like the return value of a function, or used to contain values to be used in mathematical operations.**
EBX	**Base Register. This register has no specific purpose: is it used to store temporary data, it can be also used for indexed addressing.**
ECX	**Counter Register. The Counter register is used as a loop counter.**
EDX	**Data Register. It is used for input/output operations and as an additional general-purpose register.**

Table 2: Pointer Registers	
(name)	**(description)**
ESI	**Source Index. This register is used for operations on arrays and strings, generally in "reading mode."**
EDI	**Destination Index. Same as per ESI, but generally in "write mode."**
EIP	**Instruction Pointer. This is a read only register, containing the address of the instruction that will be executed next.**
EBP	**Base pointer. Contains parameter variables passed to a subroutine, and used also for passing arguments to data structures.**
ESP	**Stack pointer. Contains the address of the top of the memory stack.**

1.3 Immunity Debugger Environment

Immunity Debugger is a great and free, debugger. The basics of Immunity Debugger are explained in a very clear and useful article from Igor Novkovic [1]. Although already explained in his article, it is worth reminding, what the four main Immunity Debugger panes are, and what information do they contain, once an executable is opened or a process is attached.

Shown in Figure 1 is the debugger interface, broken into labelled panes for easy reading.

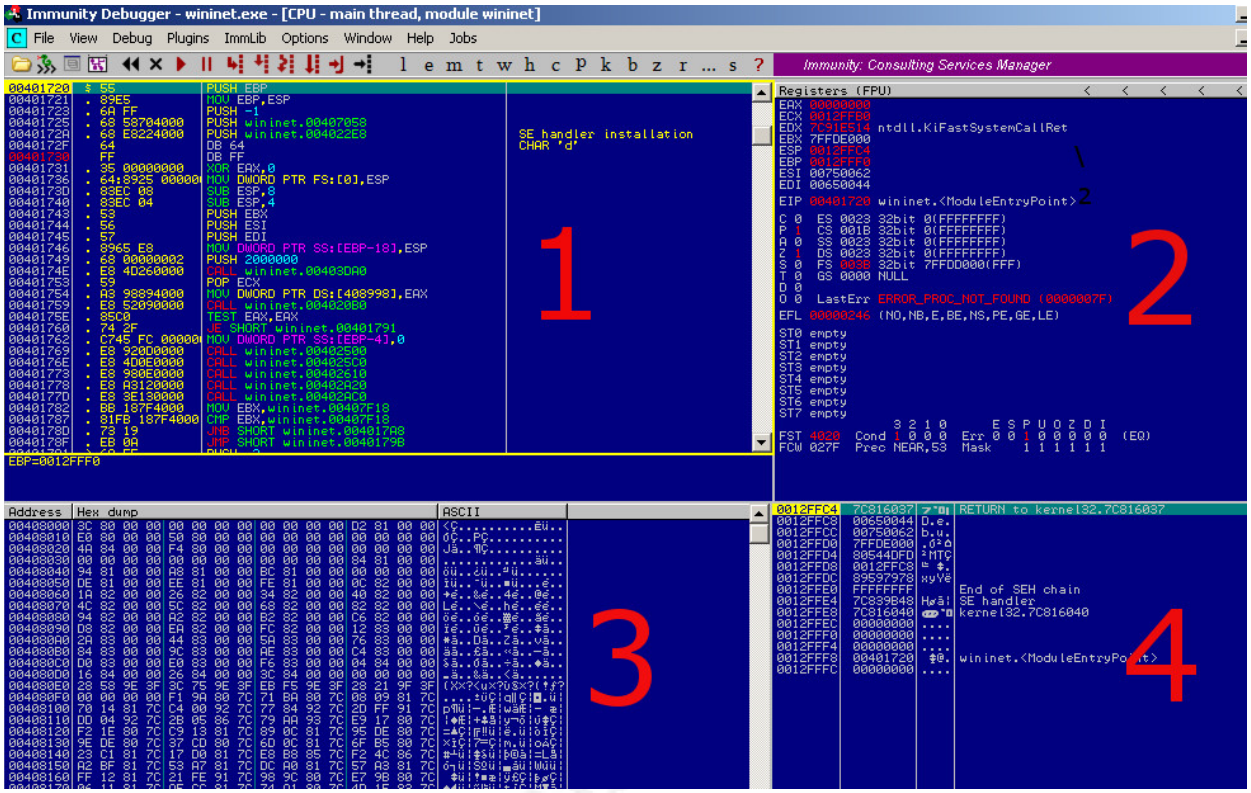


Figure 1: the Immunity Debugger user interface

- **Pane 1**: this pane will contain the assembly instructions (third column from the left), along with the original machine code (second column), and the offset for each instruction (first column). The last column (first from the right) will contain comments added by Immunity Debugger itself, or comments added by the user during the analysis;

- **Pane 2:** Pane nr. 2 contains the CPU registries pane. Some of these registers have been already briefly described, and will be mentioned during some of the examples shown below;

- **Pane 3:** this pane contains the hex dump of the executable under analysis;

- **Pane 4:** this pane contains the Memory stack view (offset and content). During the analysis of the executable or of the process, the content of this pane changes dynamically, as soon as new elements are pushed onto the stack, or removed from it.

Roberto Nardella, Roberto.nardella@fastwebnet.it

Amongst all the icons available on the menu bar, immediately under the drop down menu of the main Immunity Debugger window, three are worth a quick explanation (Figure 2):

- Display graph
- Step into
- Step over

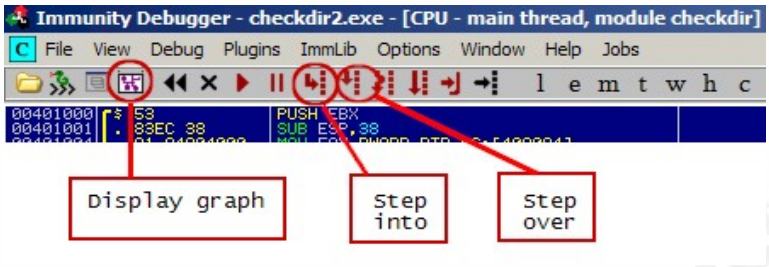


Figure 2: useful functionalities of Immunity Debugger

The first one (Display Graph) will draw a graph of the program flow, as shown in Figure 3. The logic of the program, depending on a certain condition (a new value assigned to a variable, an event, the result of a comparison and so on), may decide to execute a block of instructions instead of another, and this is obviously quite clear to understand if this flow is displayed in a graph. Different from other debuggers, the flow chart displayed by Immunity Debugger will not show the program flow of the entire program, as other debuggers may do, but just at a more contained level of recursion.

In other words, as we will see in Example nr.2, choosing a function as a starting point can be important to generate a useful flow chart for the examination. In Figure 3, for example, we can recognize some "Jump" Assembler instructions, like "JMP" (Jump), "JE" (Jump if equal), JA (Jump if above):

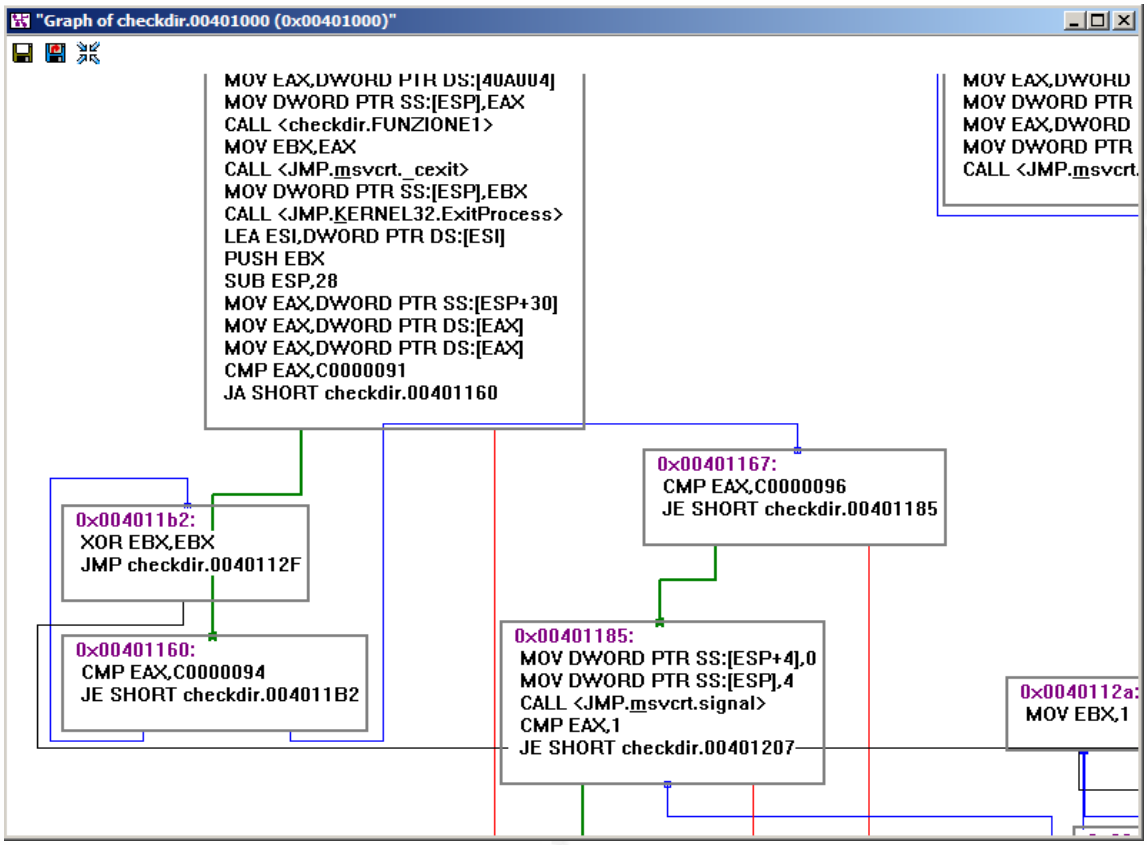


Figure 3: sample of flow chart generated by Immunity Debugger

For further details about Assembler jump instructions, it may be worthwhile to check the following Jump instruction quick reference [2].

The other two functionalities, “Step into” and “Step over,” are particularly useful to examine the program by executing one assembly instruction at a time: the difference is that, when a “CALL” instruction is met, the “Step into” functionality will make the cursor jump to the location where the called function is. The “Step over” functionality, instead, will execute the called function but will leave the cursor where it is.

When a Jump instruction is met, both “Step Into” and “Step over” will behave in the same way, and the cursor will just move to the specified offset.

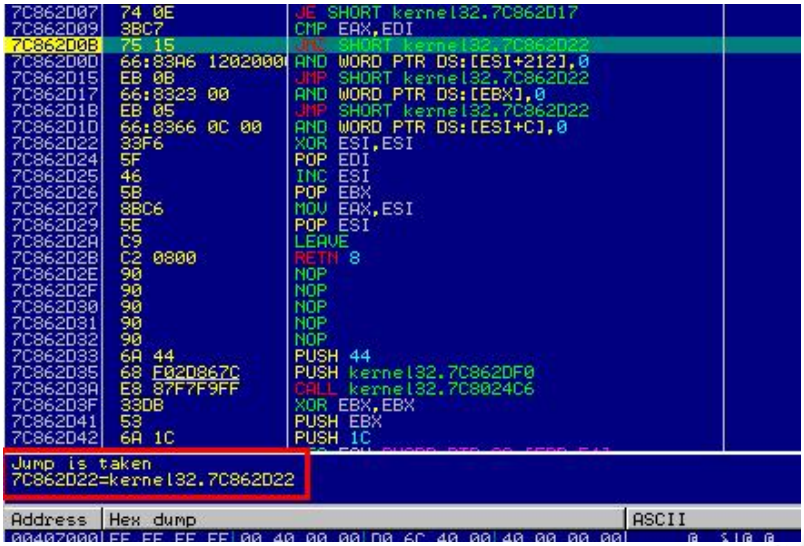


Figure 4: evaluation of a Jump instruction

The cursor (the green line) visible in Figure 4 is now placed at offset 0x7C862D0B, on a JNZ ("Jump if not zero") instruction. Immunity Debugger, in the small sub-pane immediately below the assembly instructions pane ("Pane 1"), is suggesting us that the Jump instruction will be taken (see the red box in Figure 4), because the instruction immediately preceding JNZ (CMP EAX, EDI) meets the requirement of the jump instruction. By clicking either "Step into" or "Step Over," the cursor will move to offset 0x7C862D22, which is still visible in Figure 4 (instruction: XOR ESI, ESI).

In Figure 5, the "green line" cursor is on a CALL instruction, that it is not yet executed. The pane below hints that this instruction will call a block of instructions that is located at offset 0x77C0537C:

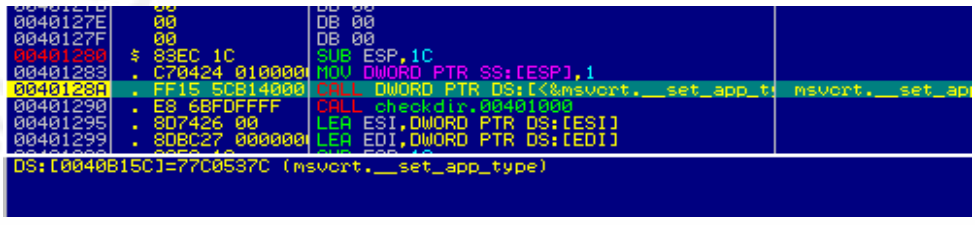


Figure 5: "green line" cursor of Immunity Debugger

By clicking "Step into," the cursor will then move to offset 0x77C0537C, as seen for the jump instructions. By clicking "Step over," the cursor will remain as it is, but all the instructions between offset 0x77C0537C and the next RETN instructions will be

executed, and all the values for all the CPU registries will be updated regularly. After the RETN instruction is executed, the cursor will go on following the program execution.

Another two quick things that it is worth mentioning about the Immunity Debugger environment are that functions are highlighted automatically with yellow brackets, as shown below in Figure 6 (exactly in between the offsets and the machine language instructions):

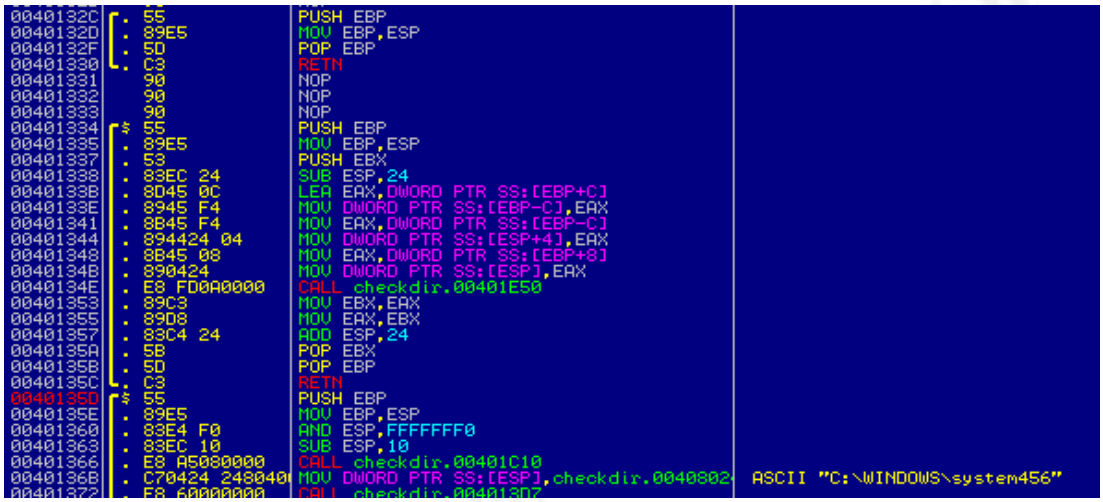


Figure 6: functions highlighted by Immunity Debugger

As we see, functions in Assembler usually start with a "PUSH EBP – MOV EBP, ESP" instructions, and end with a RETN instruction. Immunity Debugger automatically locates these instructions and highlights them with a yellow bracket.

The last detail that it is worth showing is how Immunity Debugger can show "loops" (like "For" loops or "While" loops). According to the structure and flow of the loop itself, the equivalent Assembler code may remarkably vary but the simplest one is shown in Figure 7:

Figure 7: loop highlighted by the user interface

Loop start at offset 0x004010CE, and is indicated by an "arrow" symbol (or "greater than" bracket), ">".

Roberto Nardella, Roberto.nardella@fastwebnet.it

The Assembler instruction on the line marked with a ">" symbol (the first line in Figure 7) is an "INC" ("increment") opcode, incrementing a value stored in an EAX register ("Accumulator"). Once this opcode is executed, the next assembler instruction is a "CMP" ("compare") instruction, comparing a "zero" to the value obtained by calculating the expression shown at offset 0x004010CF. At the very last line, at offset 0x00400D3, the "JNZ" ("jump if not zero") Assembler instruction will jump to the specified offset 0x004010CE if the value obtained from the expression calculation is not zero, as also indicated by the little "arrow symbol" pointing upwards. Those two just described arrows are a graphical indicator of a loop.

2. Example nr.1

The code snippet below is a simple piece of C code that:

- Renames a "notepad.txt" file, placed into a "Temp" folder, and renames it into a "notepad.txt" file;
- If the renaming operation is successful, then "notepad.exe" file is run.

Old malwares (Unitrix, to mention one [3]) used to be downloaded in the impacted machine in another file format (say, for example, jpeg) and then renamed into .EXE, with the intent of circumventing a possible rule that can detect the suspicious download of certain .EXE files. New malwares (ransomwares, like Cryptowall [4], etc.) also rename the extensions of the documents impacted by the malware (Word, Excel and so on), after they are encrypted. So, although a very basic example, it can be considered as a "real life" example. The example code is very easy to understand, and it is the following:

```
#include <windows.h>
#include <stdio.h>

int main(void){

if (MoveFile ( "c:\\temp\\notepad.txt", "c:\\temp\\notepad.exe" )) {
   ShellExecute( NULL, "open", "c:\\temp\\notepad.exe", "", NULL, SW_SHOW );
} else {
  //  printf("Errato %d\n",GetLastError());
                        return 0;
                }
           }
```

Figure 8: C code of Example nr. 1

"MoveFile" and "ShellExecute" are two functions from the Windows API and, as such, they both require the "windows.h" header file [5]. "MoveFile" [6] is used to move or rename files (including extensions): "ShellExecute" [7] is used to perform several operations on a file or directory (opening the directory, searching into it, running a file and so on).

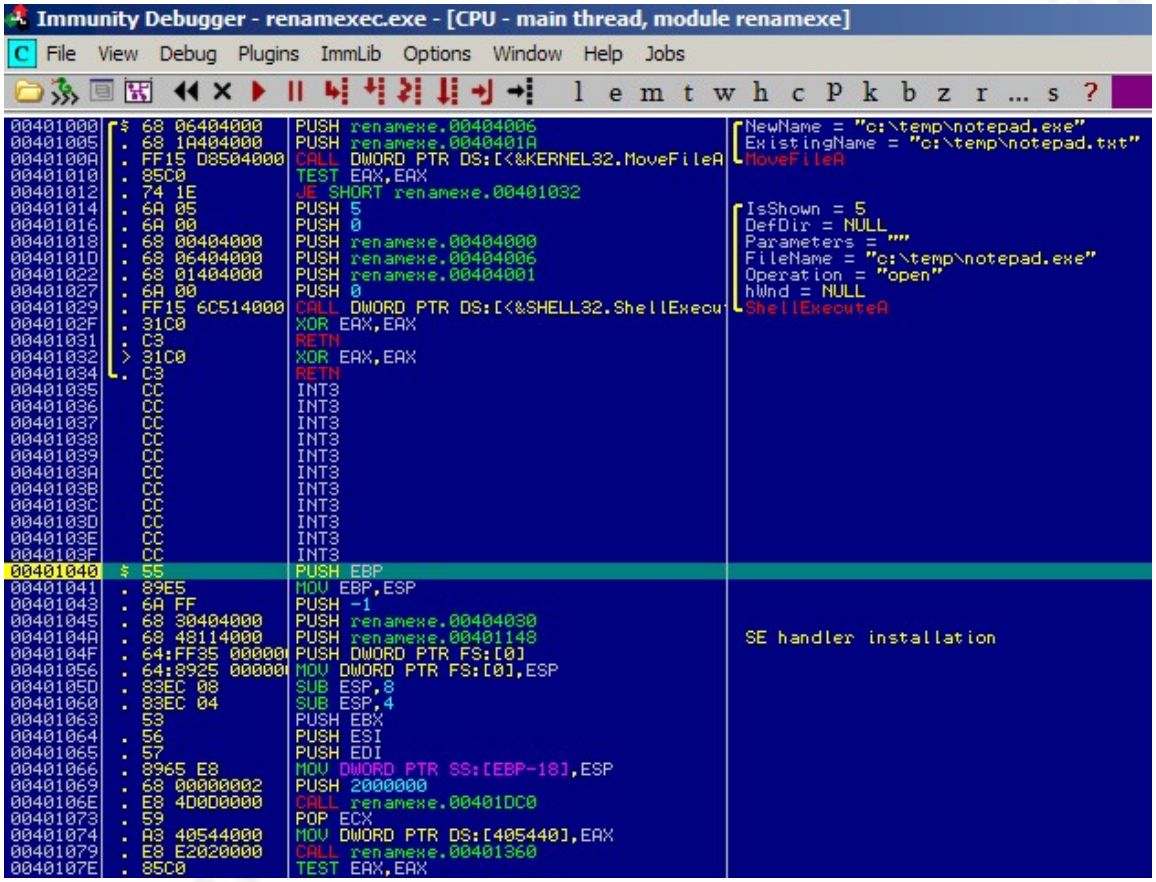


Figure 9: Example nr. 1 opened in Immunity Debugger

Once opened, the first observations of the compiled executable as seen by Immunity Debugger, is shown in Figure 9. Now we will look at specifics.

The highlighted line at offset "0x00401040" is not the Main function of the program, but the entry point for the executable. Before the code created by the programmer is run, an executable does a lot of operations (checking the CPU architecture, setup Exception Handlers, and so on) [8] before the code intended by the programmer is run.

The code represented in the screenshot above derives from an executable compiled with Pelles C. Pelles places (part of) the code created by the programmer at the very top (line

at 0x00401000), so in this case, it is very easy to find what the programmer did. But, other compilers may create completely different Assembler code, so reality is always not obvious.

One way to search for "programmer created" instructions is to look for "referenced text strings," which is a searching feature that is obtainable from the popup menu appearing with a right click on the Assembler pane, as shown in Figure 10:

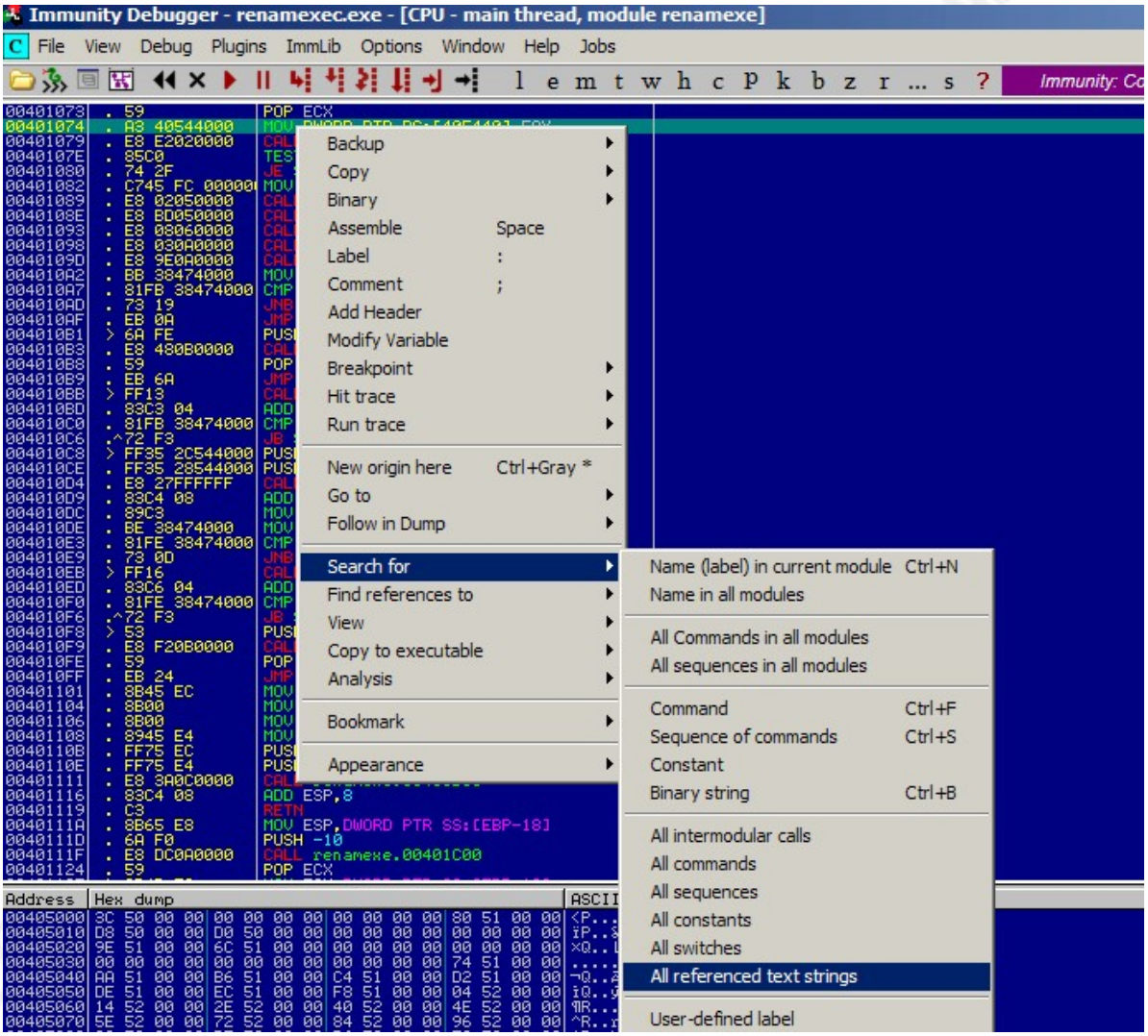


Figure 10: "Search for all referenced text strings" functionality

The result that we would get is a list of all strings that are usually stored in the ".data" section of an executable, and shown in Immunity Debugger in a dedicated pane. Each of the lines containing the extracted strings listed is a hyperlink, leading to the

corresponding Assembly instruction in the main window ("Assembly instructions" pane), using, or manipulating that string somehow.

As shown in Figure 11, some of the strings (in this example: the "ASCII c:\temp\notepad.exe") can be of interest, because usually the executable "notepad.exe" is not in the "temp" directory, nor a text file named "notepad.txt" is present in default windows installations:

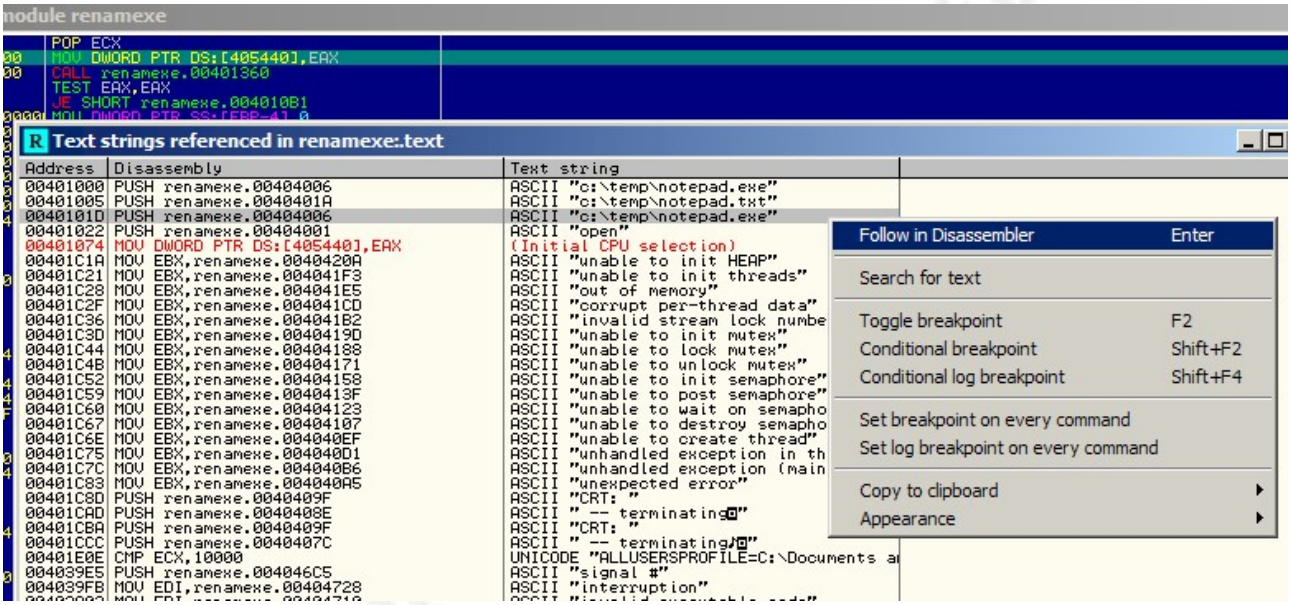


Figure 11: referenced strings in the analyzed executable

Once the selected text string is followed in the disassembler, the "green line" cursor will be placed exactly on the selected string, within the main Immunity Debugger pane:

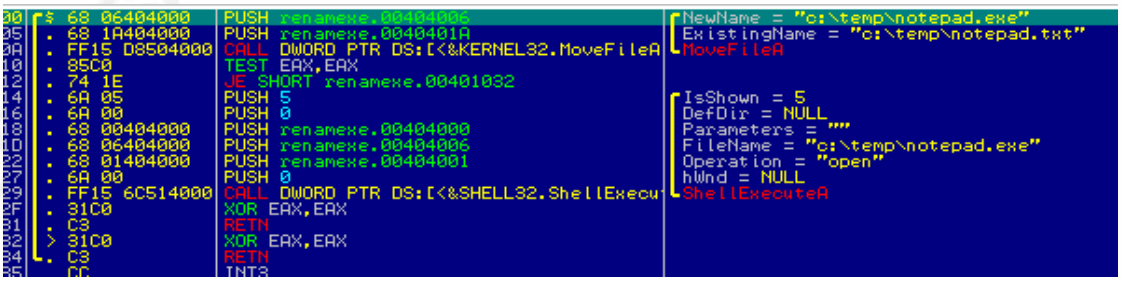


Figure 12: argument passed to MoveFileA

The first example code consists in one unique function (the Main function): from inside Main() , the other two mentioned functions "MoveFile" and "ShellExecute" are called. Apart from grouping these functions with the "yellow brackets" mentioned in the

previous paragraph, it is possible to see that the order in which the arguments are passed to the functions is exactly the opposite order used in the developer created C source code, to pass said arguments to the functions. This is because the calling convention used to compile this executable (and the next ones in this paper) is "_cdecl." See Appendix A for the various compiler options used. Microsoft MSDN explains that the "cdecl" calling convention is a "right to left" argument passing order [9].

In conclusion, from Figure 12, it appears clear that the program in question first executes a "MoveFile" instruction, which renames a "notepad.txt" file into "notepad.exe" (full paths passed as arguments to the MoveFile function). Then, the newly created "notepad.exe" is run through function "ShellExecute."

3. Example nr. 2

The code below is another C code, a bit more complex than Example nr.1, that checks the existence of a directory "C:\\WINDOWS\\system456" and, if present, the program does nothing and quits. If not present, it executes other instructions that create a registry key, write a file and open a socket connection.

In this case, the code is divided into more functions, called from the "Main()" function. In the source code in Figure 13, functions are all separated by a divider, highlighted in green font color for ease of read. Each function, in turn, will call several WinAPI functions, this time invoked not only through the "windows.h" header file, but also from the "winsock.h" [10] (for the socket) and "aclapi.h" [11] header files (since the creation of a registry key requires higher access level, obtained through the usage of the "SECURITY_ATTRIBUTES" data structure).

Looking at the code, it "emulates" actions commonly done by a huge variety of malware. Before running the malicious code, many malware check for the presence of some specific antivirus (that may already know the signature of the malware that is it going to be downloaded), malware detection tools, dissectors and analysis tools by verifying if their specific installation files and folders are present in the impacted machine. If those

files are not present and therefore the malicious code is “safe” to be downloaded and run on the impacted machine, the malware may carry out other actions like, for example, add a persistence mechanism by writing a registry value in common Autorun keys (like, for example: “HKEY_CURRENT_USER \ Microsoft\ Windows \ CurrentVersion \ Run “). Other common actions that malwares may do are downloading files, contacting C&C servers (so, starting sockets, HTTP sessions, etc.) and writing files on the disk.

The code below starts by checking the presence of a “C:\ WINDOWS\ system456”, through function “Controlla.” If the path is present, the code execution will stop. But, since this path does not exist on default Windows installation, it will do the following:

- Write a registry key named “Provadiscrittura” (through function named “scriviregistro”);
- Writes a text file in the C:\ path (through function “scrivifile”);
- Starts a socket to Google, on port 80 (through function “connetti”);

Different from the previous source code (where only two APIs were called, and there is only a single function “Main”), in this second example the code has a main() function and several additional functions, is divided into more functions, so the program flow is more complex and dynamic, especially when seen in the debugger.

```
#include <stdio.h>
#include <windows.h>
#include <strings.h>
#include <stdbool.h>
#include <aclapi.h>
#include <winsock.h>
#pragma comment(lib,"ws2_32.lib")


int controlla(const char *);
void scrivifile(void);
void scriviregistro(void);
void connetti(void);

int valore;
HANDLE hFile;
BOOL bRet = FALSE;
char* bBuffer;
DWORD bytesdascrivere;
DWORD dwWritten;
```

```
//**********************************************************************


int main(void){
   valore = controlla("C:\\WINDOWS\\system456");
            if (valore == 1){
            printf(" valore trovato : %d\n", valore);
   exit(0);
            }
   If (valore == 0) {
           scrivifile();
            scriviregistro();
            connetti();
            exit(0);
            }
            else
            exit(0);
}


//**********************************************************************

int controlla(const char* percorso){
  DWORD var = GetFileAttributesA(percorso);
  if (var == INVALID_FILE_ATTRIBUTES)
    return false;

  if (var & FILE_ATTRIBUTE_DIRECTORY)
    return true;
  return false;
}

//**********************************************************************

void scrivifile(void){
  hFile = CreateFile ("C:\\txtdiprova.txt", GENERIC_WRITE, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);

   bBuffer = "Scrivi qualcosa.";
   bytesdascrivere = (DWORD)strlen(bBuffer);
   bRet = WriteFile (hFile, bBuffer, bytesdascrivere, &dwWritten, NULL);
}

//**********************************************************************

void scriviregistro(void){
           PSECURITY_DESCRIPTOR secdesc = NULL;
           DWORD dwDisposition;
           SECURITY_ATTRIBUTES sa;
            LONG lRes;
            HKEY hkSub = NULL;

   sa.nLength = sizeof(SECURITY_ATTRIBUTES);
   sa.lpSecurityDescriptor = secdesc;
   sa.bInheritHandle = FALSE;

   char cName[] = "Provadiscrittura";
   HKEY hKey = HKEY_CURRENT_USER;
   lRes = RegCreateKeyEx(hKey, cName, 0, "", 0, KEY_ALL_ACCESS, &sa, &hkSub, &dwDisposition);

}
```

Roberto Nardella, Roberto.nardella@fastwebnet.it

```
//*********************************************************************

void connetti(void){

   WSADATA wsa;
   SOCKET s;
   struct sockaddr_in server;
   WSAStartup(MAKEWORD(2,2),&wsa);
   s = socket(AF_INET , SOCK_STREAM , 0 );

   server.sin_addr.s_addr = inet_addr("74.125.235.20");
   server.sin_family = AF_INET;
   server.sin_port = htons( 80 );
   connect(s , (struct sockaddr *)&server , sizeof(server));
          exit(0);
}
```

Figure 13: C code of Example nr. 2

Once opened the compiled executable into Immunity Debugger, the first step carried out this time was to check for all the WinAPI – Function calls done by the program. This was achieved by right clicking on the main pane and selecting "search for – all intermodular calls," as shown in Figure 14.

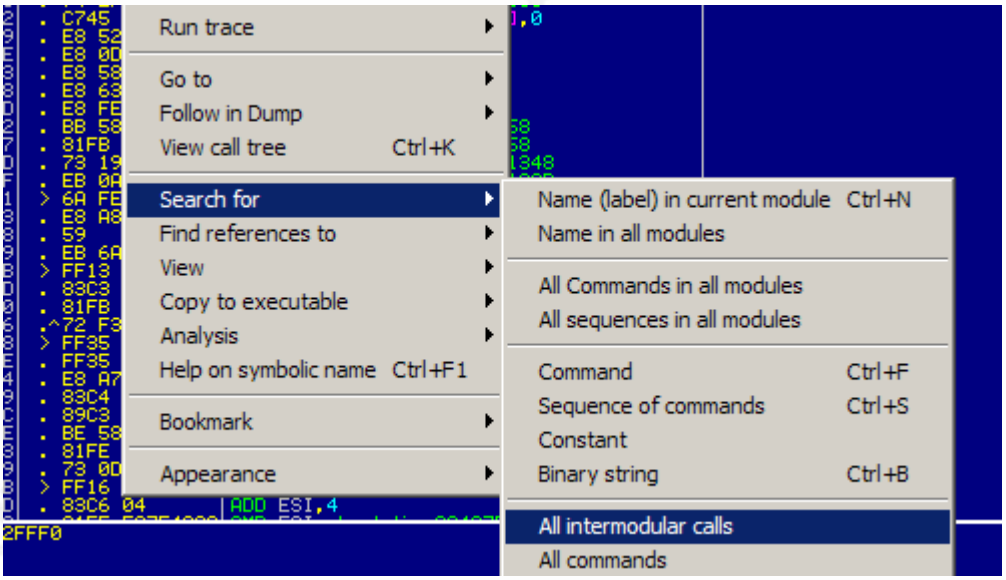


Figure 14: "Search for all intermodular calls" functionality

The obtained result was a list of all API calls for the program. Note that Immunity Debugger indicates the respective DLL for each of the called API. This is shown in Figure 15.

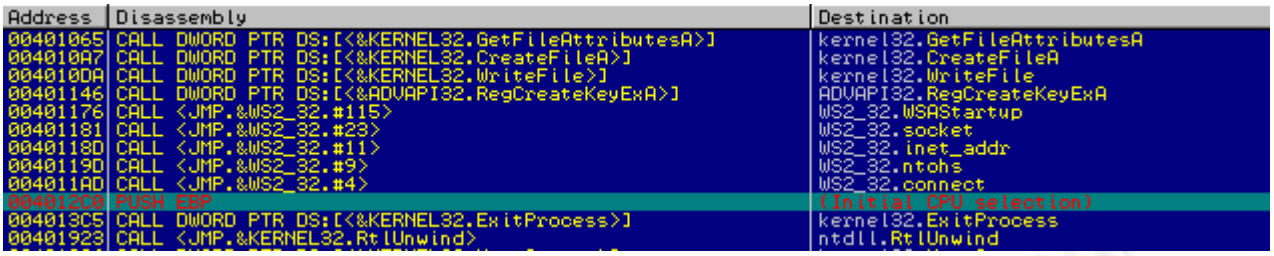


Figure 15: list of DLL and respective called functions

In Figure 15 it is possible to see that, above the program entry point (offset 0x004012C0, highlighted in red) there are some possibly interesting "GetFileAttributesA," "CreateFileA" and "WriteFile" WinAPI, imported from "KERNEL32.DLL." Then, a "RegCreateKeyExA" WinAPI, imported from "ADVAPI32.DLL," and several functions for socket creation (a "Berkeley" socket, in this example), were imported from "WS2_32.DLL." These are already some indications on what the program may do: obtain some info on a file, create a file, write content in a file, create a registry key and start a socket.

In this case, Immunity Debugger was able to recognize the 100% of the functions and APIs imported from the "Kernel32.DLL," "ADVAPI32.DLL" and "WS2_32.DLL" library files, since those are fundamental, widely used DLL. So, the name of the involved WinAPIs is automatically detected and returned in either the "Destination" column of the above screenshot, and in the first column from the right of the main pane (in red font color).

For informational purposes, it is useful to know that Immunity Debugger offers the possibility to load the Microsoft Debugging Symbols Table similarly to Microsoft' WinDBG debugger, by choosing the "Debug – Debugging Symbols Options" from the drop down menu at the very top of the GUI [12].

Different from the famous GDB debugger, Immunity Debugger does not have the capability for identifying the Main() function offered by the GDB command "disas main." However, a tutorial named "Finding Main() – Compiler Code vs. Developer Code" [13] shows a way to locate the main function by following the logic of the program itself.

Roberto Nardella, Roberto.nardella@fastwebnet.it

Proceeding to analyze the program, the program's entry point (indicated in the main assembly instruction pane as "Initial CPU selection", and also visible in Figure 15) does not correspond to the program's Main() function, so the first big challenge is identifying the program's starting point, from a programmer's perspective. Thus, differentiating between programmer generated code and compiler generated code may be challenging.

A way that can help understanding the logic of the program is following the procedure explained in the tutorial from Heffner by also combining that modus operandi with the usage of the "LABEL" functionality of Immunity Debugger, and in conjunction with the "Display Graph" one.

To do this, once one of the functions with the "search for all intermodular calls" or "search for all referenced text strings" functionalities is located, we can add an arbitrary name to the function just found by setting up a "label" (right click on the first instruction line of the function, and then choose "Label":

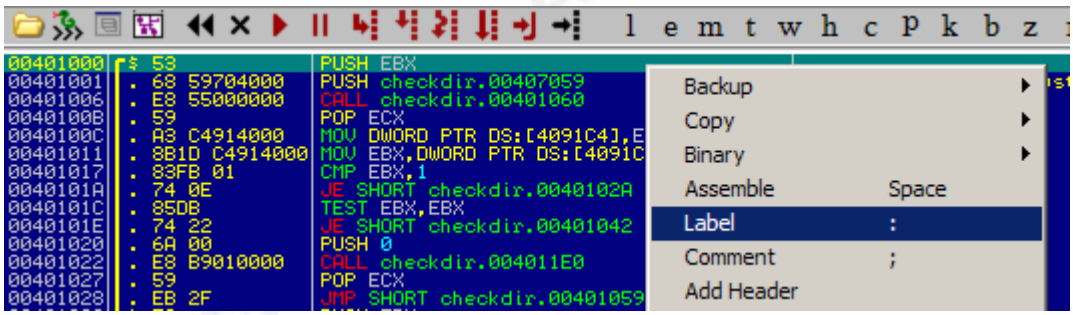


Figure 16: adding a label to an instruction line

For this exercise, the author has chosen "CHECK" as Label name. Then, by repeating this step for every string or intermodular call of interest that the author has located using these methods, the author can label them as "GET FILE ATTRIBUTE", "WRITEFILE", "REGISTRYKEY" and "SOCKET." Then, once activated the "Display graph" functionality, we see that the entry point of the program calls a block of instructions at offset 0x00401302, containing four CALL instructions:

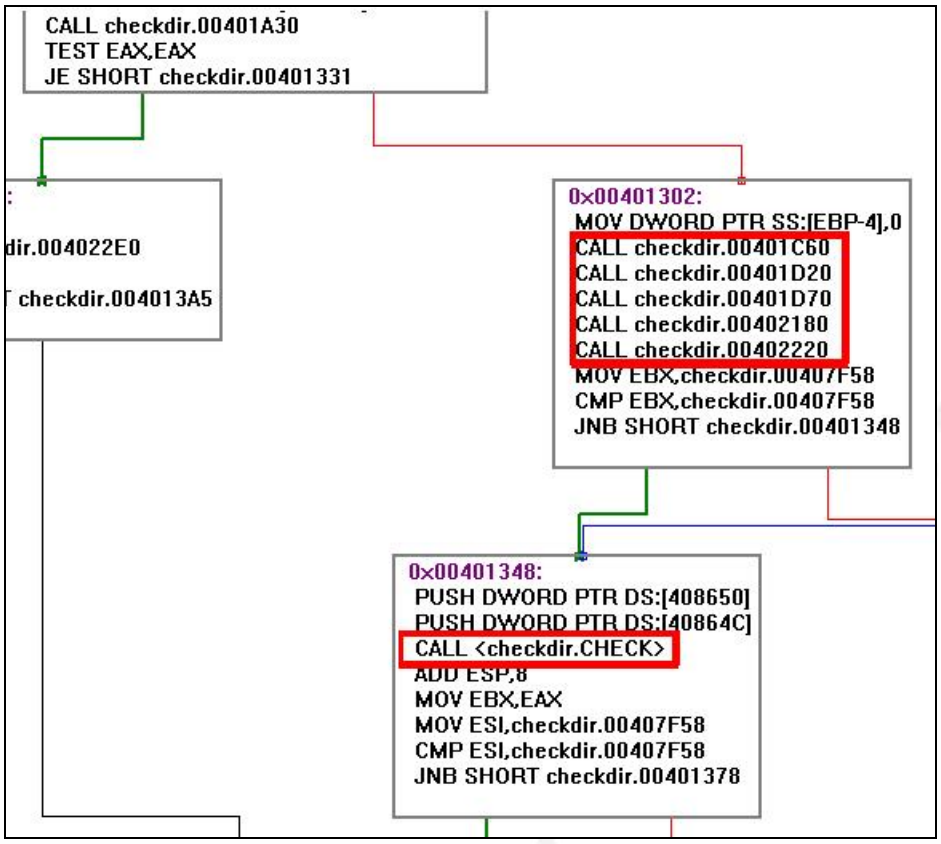


Figure 17: flow chart for Example nr.2, program entry point

By examining the code pointed by the four CALL opcodes, one by one, we can say that none of them is the Main() of the program because:

- 0x00401560 contains the CPUID instruction, which is an Assembly opcode that collects more info on the processor;
- 0x00401D20 contains GetFileSystemAsTime, which Is another API usually called to initialize an executable;
- 0x00401D70 contains opcodes corresponding to APIs GetStartupInfoA, GetFileType, GetStdHandle (same considerations as above);
- 0x00402180 contains a reference to API GetCommandLineA, so it could be a possible equivalent of the “int main(void)” C instruction;

- 0x00402220 contains a reference to the "GetEnvironmentStrings" API, so same considerations as above.

In the flow chart shown in Figure 17, we see that the block just reviewed leads to another block of instructions where there is another call of possible interest (highlighted in a red box as well), we can see that the offset for that instruction block has been replaced with the Label name that we have provided before ("<CALL checkdir.CHECK>"). This makes the flow chart a bit more readable and, at the same time, hinting us that we may be close to the code written by the programmer.

If the "Display graph" functionality is invoked by highlighting the starting instruction line of "checkdir.CHECK" (with the "green line cursor", in the main pane) instead of the program entry point ("Initial CPU Selection"), we will get a different flow chart, starting from the line that we have just highlighted:

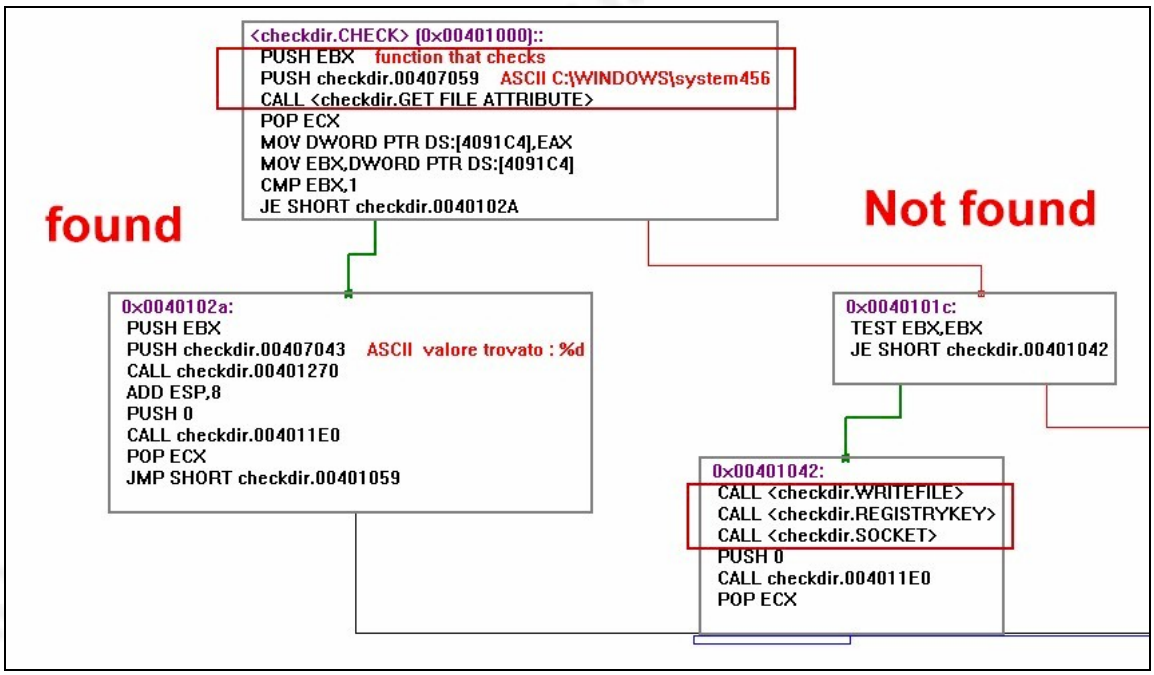


Figure 18: flow chart of Example nr. 2 from "checkdir.CHECK"

This shows a much a clearer logic of the program.

At this stage, it is worth showing additional details concerning how the arguments involved in the "Socket" instruction block are passed to the functions. In the socket data

structure of the example C code, we have specified a Port 80 by using the “htons” ("host to network short") function. But, in the debugger, we see that this parameter is passed to the data structure via a “ntohs” ("network to host short") function, and the argument is a “50”. “50” is the Hex equivalent for the decimal “80,” and the explanation on why this function substitution occurred is explained in another paper [14].

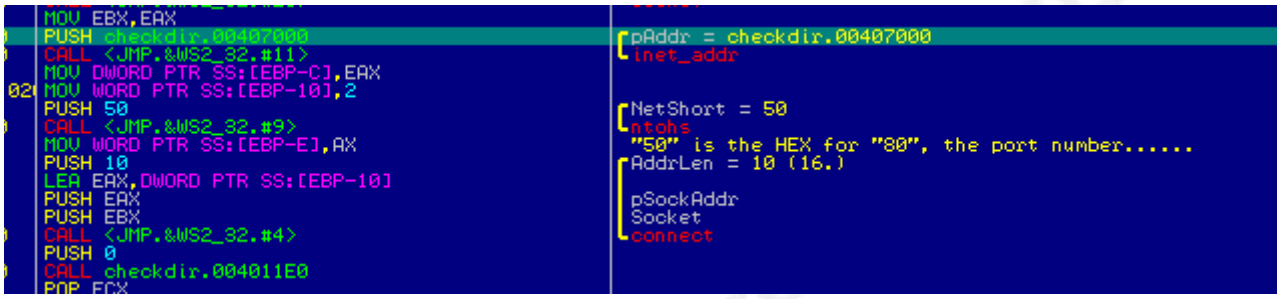


Figure 19: passing arguments to a socket

Therefore, according to the processor running the program, “these functions convert from your native byte order to network byte order and back again.” Another detail highlighted in the screenshot above is that, although the IP address is passed as an “ASCII string” argument, it is not shown in the comment column for unknown reasons, although it is easy to be found by using the already shown “search for all referenced text strings” functionality. However, we can see the offset where this string is located, which is at offset 0x00407000. Immunity Debugger does not show the content of the “.data” section in the four main panes of its GUI: it lists the opcodes containing in the “.text” section of the executable (which, for this program, range from 0x00401000 to 0x00406FFF), and copies the referenced data from the “.data” section exactly in the comment column.

A way to see how this argument is passed, and to see it in clear, is toggling a breakpoint (“F2” key) for each instruction comprised in the “socket” block of instructions. Once the program is run from within the debugger, we can see the IP address appearing in clear in the memory stack pane, at offset 0x0012FDD8, with a clear reference to its offset, 0x00407000:

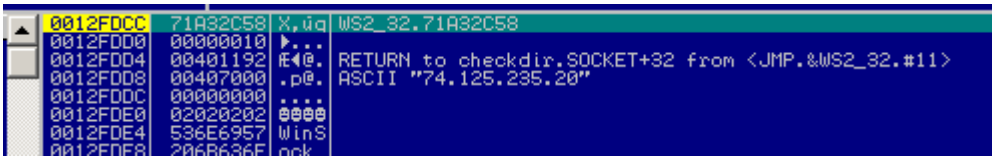


Figure 20: IP Address loaded in memory stack

After that, the IP address is moved to the EAX registry (accumulator), as shown in Figure 21: this registry is a general-purpose register, and in case of need, it may act as a kind of "temporary storage" register.

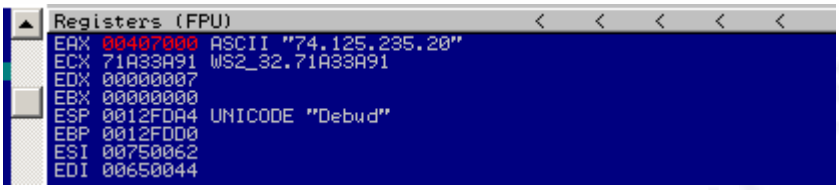


Figure 21: reference to the ASCII string moved in EAX

EAX was used to contain the IP address from the stack because one of the modules (DLL linked to the executable, in this case "WS32_2.DLL") called by the executable was responsible for this operation. In Figure 22, we can first notice three details:

a) the first one is that the title for the main window of the GUI changed into "CPU – Main thread, module WS2_32.DLL";

b) The second one is that the offset range has changed. As we just said, the offset range for the .text section of this program is from 0x00401000 to 0x00406FFF, but in Figure 22 we can see that the current offset is 0x71A32C00;

c) In Figure 22 we can see (highlighted in green by the cursor) that a MOV instruction is copying a value contained in the Stack Segment (SS) , beginning at a location which is equal to the value of the Base Pointer (0x0012FDD0, in this specific case) , incremented by 8 (so, giving 0x0012FDD8).

If we go back to Figure 20 (the memory stack pane), we can see that, in fact, the IP address is located exactly at memory stack offset so, giving 0x0012FDD8.

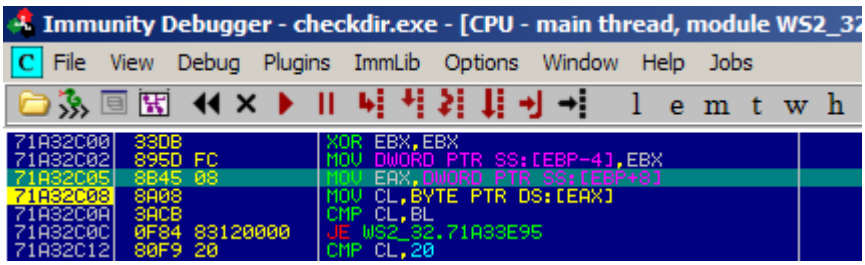


Figure 22: how reference to ASCII string is moved to EAX

After that, just before a socket connection is effectively created, we can see that the EAX registry has been used to pass this IP address to a function "RtlIpv4StringToAddress," which is a function imported from module "NTDLL.DLL" [15]. See Figure 23a for more information.

Figure 23a: arguments passed to "RtlIpv4StringToAddress" function

This function has not been invoked by the code of the programmer, but automatically added by the compiler to the program; it converts a String IPV4 address into an Ipv4 address in binary format. According to the MSDN documentation, this function takes four arguments, and the link to the full documentation for this function is in the References section of this paper.

In Figure 23b (showing the memory stack pane again) it is possible to see the four arguments in question being passed to this function:

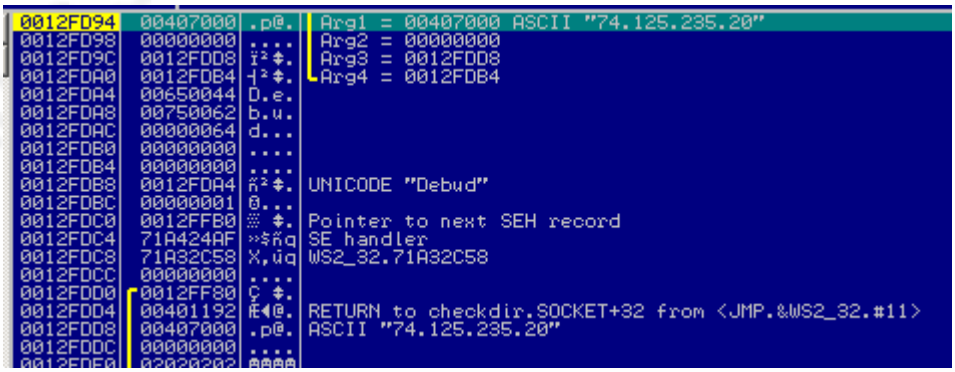


Figure 23b: arguments passed to "RtlIpv4StringToAddress" function

4. Example nr. 3

In Example nr.2 we have seen how a socket is invoked, in order to establish a sort of connection to an external infrastructure: a Berkeley socket has been chosen just for mere demonstration purposes, to show how arguments to socket functions may be passed to the function itself. However, most malware coded to run on Windows systems uses the "Windows native" API and more easy to handle "Wininet" APIs, which are defined in the "wininet.h" header file instead of the Berkeley socket [16].

Moreover, the URL address that a socket may receive as argument may not be "in the clear" as shown in the above example, but could be encoded for various reasons (e.g., to avoid detection or simply to make an eventual analysis a bit more complicated, etc.).

The sample C Code shown below in Figure 24:

a) Stores an encoded URL ("http://www.evil-website.it/main.html") in a variable named "percorso." The URL is ROT13 encoded in a resulting string "uggc122jjj4rivy3jrofvgr4vg2znva4ugzy";

b) Calls function "decodifica," which contains a routine to decode a string from Rot13. Other than decoding, there are some character replacements ("/", ":", "-") to compose the URL in the correct way. The decoded URL is then passed to variable "finale";

c) Calls function "scarica" to download the content of "main.html" by using several APIs from the Wininet library ("InternetOpenURL", "InternetReadFile" etc). The URL "in clear", decoded with function "decodifica", is passed as argument through variable "finale."

```
#include <windows.h>
#include <wininet.h>
#include <stdbool.h>
#include <stdio.h>

LPSTR decodifica(LPSTR);
BOOL scarica (LPSTR);

int main (void) {
        BOOL bRet;
        LPSTR percorso = "uggc122jjj4rivy3jrofvgr4vg2znva4ugzy";
```

```
            LPSTR finale;
            finale = decodifica(percorso);
//          printf("%s\n", finale);
            bRet = scarica (finale);
return 0;
}

//*********************************************************************


BOOL scarica (LPSTR lpszUrl) {
            HINTERNET hInternet;
            HINTERNET hInternetUrl;
            BYTE bBuffer[1024];
            DWORD dwRead;
            BOOL bRet = FALSE;

hInternet = InternetOpen ("MyAgent/1.0", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);

if (hInternet != NULL) {
   hInternetUrl = InternetOpenUrl (hInternet, lpszUrl, NULL, 0, 0, 0);

if (hInternetUrl != NULL) {
            do {
            bRet = InternetReadFile (hInternetUrl, bBuffer, sizeof (bBuffer), &dwRead);
                                    printf(bBuffer);
                                    printf("\n");
            } while (bRet && dwRead == sizeof (bBuffer));

                        InternetCloseHandle (hInternetUrl);
                        }
            InternetCloseHandle (hInternet);
            }
return bRet;
}

//*********************************************************************

LPSTR decodifica(LPSTR input) {
   int cont;
char *buffertemp;
buffertemp = (char*)malloc(sizeof(char) * (strlen(input)+1));

for (cont = 0; cont<strlen(input); cont++) {
              //    else
    buffertemp[cont] = (((input[cont]-97)+13)%26+97);
            if (buffertemp[cont] == 'X') buffertemp[cont] = 58;
            if (buffertemp[cont] == 'Y') buffertemp[cont] = 47;
            if (buffertemp[cont] == 'Z') buffertemp[cont] = 45;
            if (buffertemp[cont] == '[') buffertemp[cont] = 46;

 }
 buffertemp[strlen(input)] = '\0';
            return buffertemp;
}
```

Figure 24: C code for Example nr. 3

To examine this executable, steps already seen in example nr.1 and nr.2 ("search for all referenced text strings" and "search for all intermodular calls") have been followed, and some APIs and functions of possible interest have been located, by also using the "Display graph" function. Instruction blocks have been "delimited" by using labels, comments and by toggling breakpoints. Although this provided with a clearer idea on the actions done by the executable, the problem of the "encoded URL" remains.

During an observation of the program execution, at offset 0x004010CE there is an interesting loop, parsing each character of the encoded string (Figure 25).

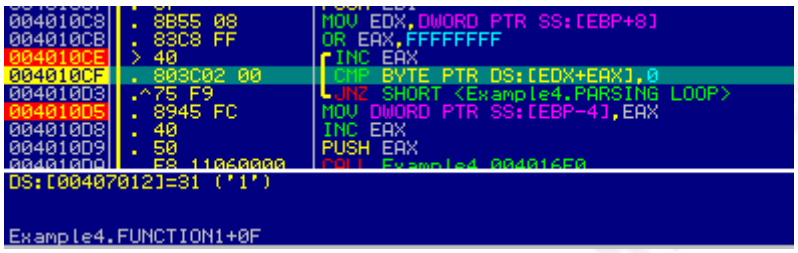


Figure 25: Loop parsing each character of the encoded string

In the subpane below, it is possible to see that the instruction highlighted in green (the "Compare" instruction, CMP" is comparing the current value contained in the Data Segment ("DS"), at offset 0x00407012, with a "zero."

In this particular case, we are seeing the fifth loop, so corresponding to a "1," which is the fifth character of the string "uggc122jjj4rivy3jrofvgr4vg2znva4ugzy."

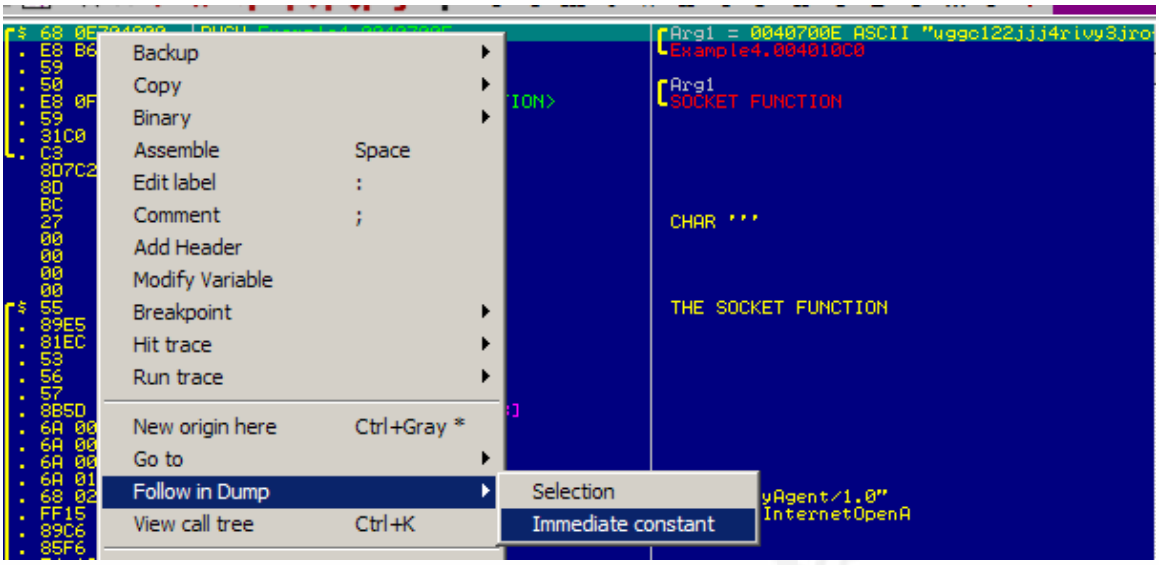


Figure 26: Locating the encoded string in the .data section

The "DS" register is invoked because the string in question is in the .data section of the executable. In order to locate that string in the .data section, the "follow in dump" functionality (right click on the assembly instructions pane) can be of help. "Immediate constant" is the option leading to the string, as the "Selection" option will lead to the "PUSH" instruction (so, in the .text section of the executable). Once "Immediate Constant" is chosen, we can see in the dump that the character "1" is in fact as offset 0x00407012 (highlighted in green, in Figure 27):

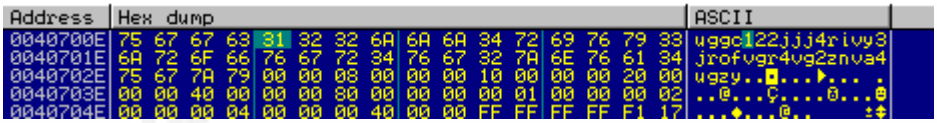


Figure 27: Locating the encoded string in the .data dump

It is then worth observing the instruction "CMP BYTE PTR DS: [EDX+EAX], 0".

The condition here is comparing "null" (represented with a "0") because the "null" character (represented in ANSI C as a '\0') is the string terminator. So, in other words, this instruction is comparing each character of the string to null, if the result is "not zero" (=if is a string character), the loop will continue because of the Jump instruction "JNZ" (jump if not zero).

Why the expression "[EDX+EAX]" was used?

Roberto Nardella, Roberto.nardella@fastwebnet.it

We said that EAX is a kind of temporary container that may contain many different kinds of data. In this case, it is used to store the element number of the array which is currently under examination (the fifth character, because the first element of an array is 0). EDX, which is another general-purpose register, in this case is containing the offset of the initial, first character of the string, which is 0x0070700E. By adding "4" to this offset, we then get 0x00407012. On each loop, EAX is incremented by one. This is visible in Figure 27:

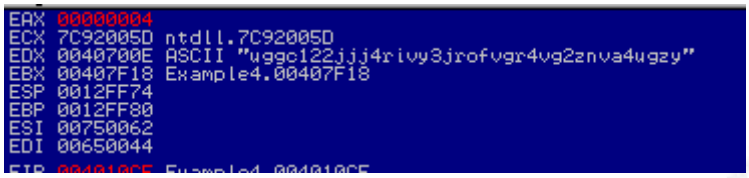


Figure 27: Current values of EAX and EDX

We saw, at table 2, that pointers ESI and EDI are pointer registers used to manipulate arrays and strings, and in particular to contain strings (ESI) and to copy strings in (EDI) after eventual manipulation. In addition, we know that strings are arrays of characters.

After having identified the decoding routine and closely observed it running by using the "step into" functionality, we can see the decoded string "appearing character by character" in the ESI register, whilst keeping F7 pressed (Figure 28):

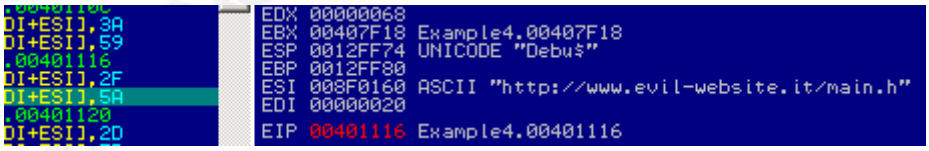


Figure 28: Decoded URL being created in the ESI register

Then, in Figure 29, we can see that the instruction "MOV EAX, ESI" (highlighted in green) copied the decoded URL from ESI to EAX, the temporary container. The reason why the decoded URL was copied to EAX is the same reason already seen in example nr.2 (passing arguments to the socket data structure): after these operations, this URL is processed by the APIs of the WININET.DLL module, and passed to this module as argument by using the EAX register:

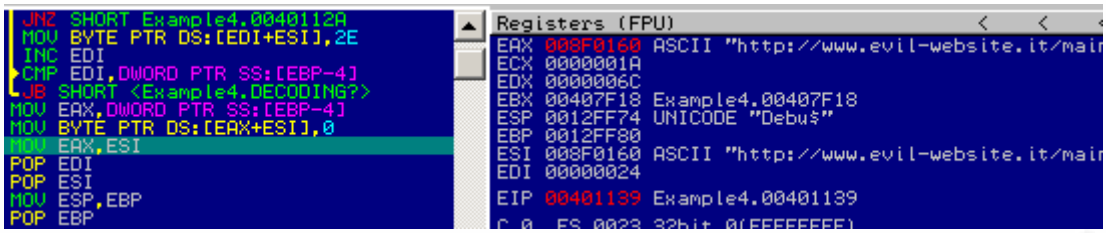


Figure 29: String in the ESI register copied to the EAX register

5. Conclusion

The three shown example codes, although safe to run, tend to simulate some of the actions commonly done by malware with the intent to provide some real world simulation of basic reverse engineering. The scope of this paper is to show a general overview of Immunity Debugger, as well as showing very easy and practice checks that may return useful information on an (unpacked) executable under analysis. Whenever a C code is converted into Assembler with a Debugger, the total number of instruction lines is huge, and examining them all would be a huge (and very often useless) effort, especially during a real malware analysis, where time is of great importance. Therefore, the art of Reverse Engineering consists also in developing the ability of identifying the key actions that the executable (or the DLL, or the attached process) does. Of course, an advanced knowledge of Assembly language and C programming language are highly recommended skills. The case studies shown in this paper represent only the beginning of an amazing but difficult field like reverse engineering, where the learning curve is very steep.

Roberto Nardella, Roberto.nardella@fastwebnet.it

6. Appendix A

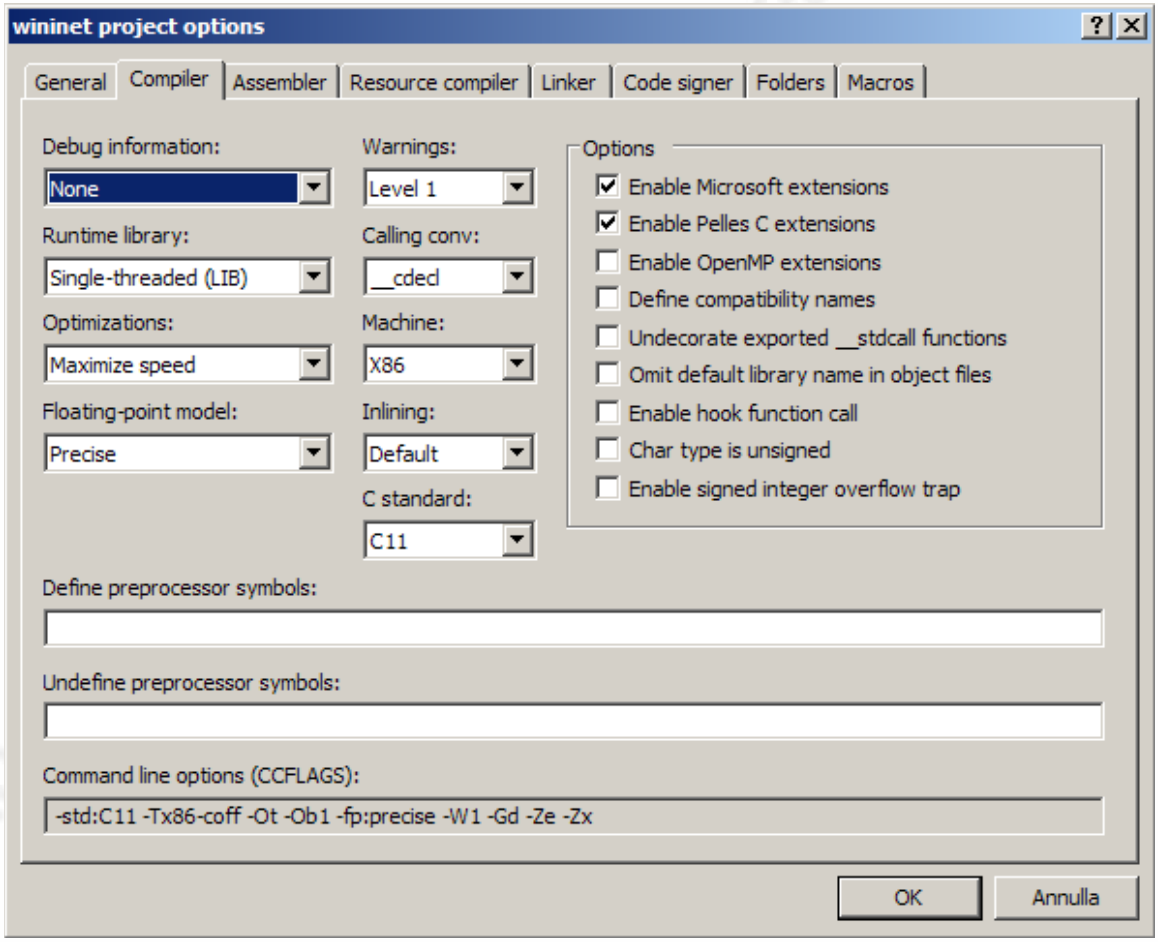
All the source codes shown in the above examples have been compiled with Pelles C Compiler, version 8.00.11, 32-bit edition.

6.1. Pelles Projects options

The Pelles compiler options used are shown in the below screenshots:

6.1.1. Compiler options:

The chosen C standard is "C 2011", and the calling convention is "cdecl":



6.1.2. Linker options:

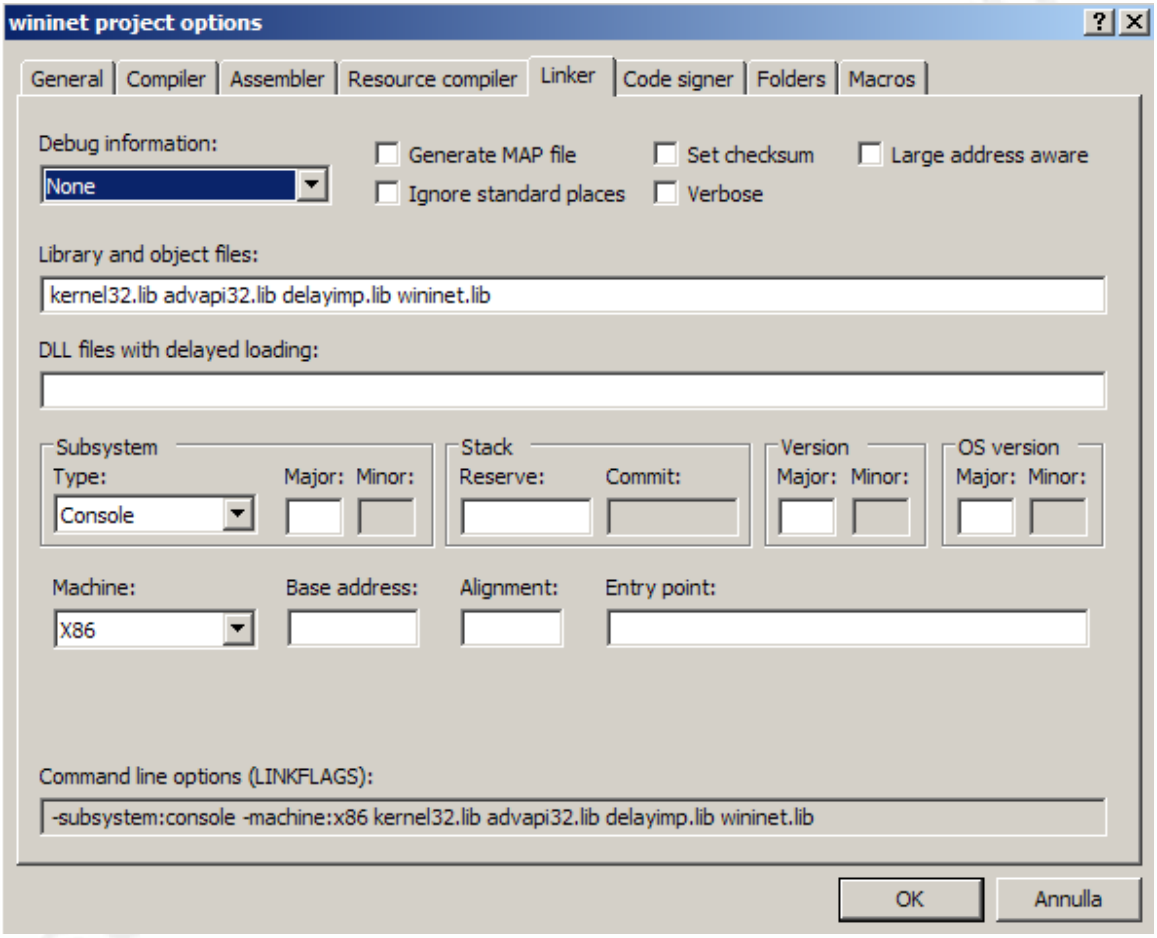
The screenshot below shows the linker settings. In particular , it shows the options for "Library and object files" for the code of "Example nr.3".

Make sure that, in order to recreate the examples correctly, the libraries and object files that have to be chosen for the compilation of the executables are the following:

Example 1: kernel32.lib, advapi32.lib, delayimp.lib, shell32.lib

Example 2: kernel32.lib, advapi32.lib, delayimp.lib, WS32_2.lib

Example 3: kernel32.lib, advapi32.lib, delayimp.lib, wininet.lib



7. References

[1] Novkovic, Igor. *Immunity Debugger basics, part 1*. Blog. Student blog sgros-students.blogspot.ca. May 22, 2014. Retrieved Feb 15th, 2015.

http://sgros-students.blogspot.ca/2014/05/immunity-debugger-basics-part-1.html.

[2] Intel x86 JUMP quick reference, (n.d.), retrieved 04/25/2016, from the Steve Friedl's Unixwiz.Net Tech Tips:

http://unixwiz.net/techtips/x86-jumps.html

[3] Frink, Lyle, *"Unpacking" the Unitrix Malware*, Avast! Blog, Sept. 7th, 2011, Retrieved Feb. 9th, 2016,

https://blog.avast.com/2011/09/07/unpacking-the-unitrix-malware/

[4] *CryptoWall .aaa Extension Ransomware Removal Guide* , Blog. Deletemalware.blogspot.co.uk, Aug. 6th, 2015, Retrieved Mar. 1st, 2016,

http://deletemalware.blogspot.co.uk/2015/08/cryptowall-aaa-extension-ransomware.html

[5] Steane, Andrew M, *Quick introduction to Windows API*, Exeter College, Oxford University and Centre for Quantum Computing , 2009, Retrieved Mar. 2nd, 2016, https://users.physics.ox.ac.uk/~Steane/cpp_help/winapi_intro.htm

[6] MoveFile Function, (n.d.), retrieved 04/25/2016, from Microsoft MSDN:

https://msdn.microsoft.com/it-it/library/windows/desktop/aa365239%28v=vs.85%29.aspx

[7] ShellExecute Function, (n.d.), retrieved 04/25/2016, from Microsoft MSDN:

https://msdn.microsoft.com/en-us/library/windows/desktop/bb762153%28v=vs.85%29.aspx

[8] *Crt0*, Wikipedia, (n.d.), retrieved Mar 4th, 2016,

https://en.wikipedia.org/wiki/Crt0

Roberto Nardella, Roberto.nardella@fastwebnet.it

[9] *Cdecl calling convention*, (n.d.), retrieved 04/25/2016, from Microsoft MSDN:

https://msdn.microsoft.com/en-us/library/zkwh89ks.aspx

[10] *Creating a Basic Winsock Application*, (n.d.), retrieved 04/25/2016, from Microsoft MSDN:

https://msdn.microsoft.com/it-it/library/windows/desktop/ms737629(v=vs.85).aspx

[11] *Taking Object Ownership in C++,* retrieved 04/25/2016, from Microsoft MSDN:

https://msdn.microsoft.com/it-it/library/windows/desktop/aa379620(v=vs.85).aspx

[12] *Load IE Symbols in Immunity Debugger,* (May 28th, 2015), retrieved 04/25/2016, from ReverseEngineering – Stackexchange.com:

http://reverseengineering.stackexchange.com/questions/9006/load-ie-symbols-in-immunity-debugger

[13] *Finding Main() – Compiler Code vs. Developer Code*, (October 18th, 2007), retrieved 04/15/2016, from the Ethical Hacker Network:

https://www.ethicalhacker.net/columns/heffner/intro-to-reverse-engineering-part-2#findingmain

[14] *Htons() function description*, (n.d.), retrieved 04/25/2016, from the Beej's Guide to Network Programming:

http://beej.us/guide/bgnet/output/html/multipage/htonsman.html

[15] *Wininet Reference*, (n.d.), retrieved 04/25/2016, from Microsoft MSDN,

https://msdn.microsoft.com/en-us/library/windows/desktop/aa385483(v=vs.85).aspx

[16] RtlIpv4StringToAddress Function, (n.d.), retrieved 04/25/2016, from Microsoft MSDN:

https://msdn.microsoft.com/it-it/library/windows/desktop/aa814458%28v=vs.85%29.aspx

8. Bibliography

Petzold, Charles (2011), *Programming Windows – 5th Edition, The Definitive Guide to programming Windows API*, Microsoft Press;

Eilam, Eldad (2005), *Reversing – Secrets of Reverse Engineering*, Wiley Publishing;

M. Sikorski, A. Honig (2012), *Practical Malware Analysis*, No Starch Press;