Attack

Reverse Engineering Binaries

Aditya K Sood aka 0kn0ck

Difficulty

●●○

This paper describes a Level 2 practical analysis of a window binary. It covers the methodical approach to reverse engineer an executable. The binary can be a console program or GUI based. The point of this talk is to understand a hierarchical layout to reverse an application within specific time limits.

The primary concern is to understand the flow of executing statements in a definitive way so that reversing will be easy. This is only possible if there are specific ways to follow. The techniques will be practically cited. This is undertaken as Real Time dissection of an executable. This article is designed specifically to give hands-on experience in reversing a windows executable. We will reverse engineer different binary structures to prove the ingrained concepts. A number of tools will be used in demonstrating a concept. Each single technique is projected with use of a tool. This helps the user in understanding the core concepts and the usage of different tools.

The reversing of a binary basically revolves around on three parameters. Time is a crucial factor because targets have to be completed in defined constraints of time. Resources are important because it reflects the dependency of a binary on other objects of system. The final point is the Functionality of code. It encompasses the flow and direction of the statements. So the overall approach is to walk along the triangular edges for analysis. The practical analysis of a binary is structured around the paradigm shown below: see Figure 1.

All the versatility of an executable primarily works on these benchmarks. The basic fundamental in reversing an executable is to check the characteristics of that window executable. We will examine a binary called afind.exe, designed for proving reverse engineering concepts. Through this a user will understand the points to look for in a binary and type of technique to be applied.

What you will learn...

- The user will learn a practical way to dissect executables
- New techniques of analyzing executables by reversing the parameters
- Framing of reverse engineering as a process
- Hand held knowledge of active debugging and disassembling

What you should know...

- The user should have basic skills of reverse engineering
- Good understanding of Windows Executable
- Intermediate knowledge of debugging

Facts Regarding Binaries:

- The first fact regarding binaries is the Association of Events. It covers the executable behavior of a binary. This is summed up as the working effect on the system. It is only possible if an executable has an inter-facial paradigm with the base system. Due to this certain events occurred in a system that changes the state when a binary is executed. This process is termed as *Event Association*.
- The second fact comprises of the Algorithmic view. This means whether an executable is using a certain algorithm or its working is independent. The term independent is used because there are a number of binaries that only use easy functions with any interdependency among code objects. This process is called *Scrutinizing Algorithmic Flow*. The algorithms can be directly applicable or multi-staged. The directly applicable algorithms have directed flow. This means the algorithm functionality is totally driven in a single pattern. On the other side, multi-step working is undertaken and cross referenced checks are performed during the implementation of an algorithm.
- The third fact relates to extracting the overall information by looking at the front end of a binary. This process is termed as *Front End Checking*. It is useful in analyzing GUI-based programs and helps the reverse engineer to understand the working functionality on front end objects. This technique is general but very useful when one is scratching any executable on the system.
- The fourth fact is summed up as the compression of an executable. This means whether an .exe file is compressed or packed with the help of a packer. So it is absolutely crucial to have information on that packer. After that,

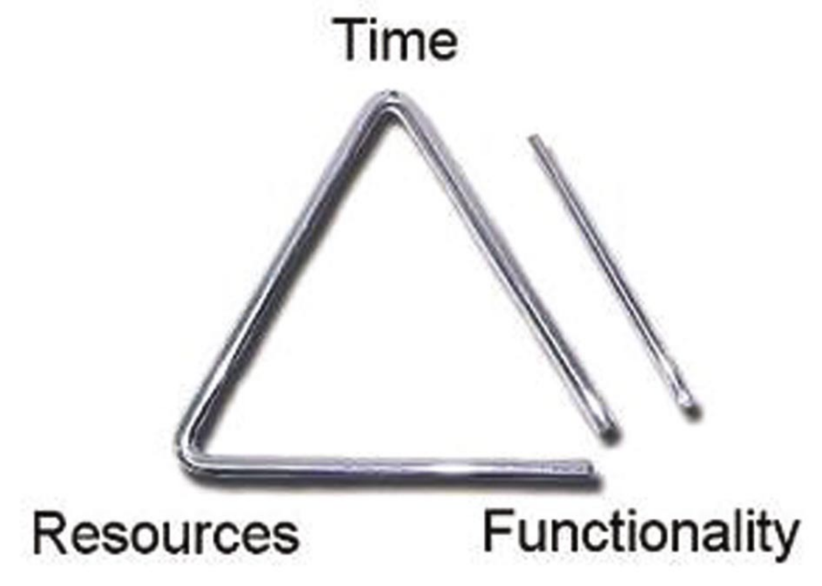
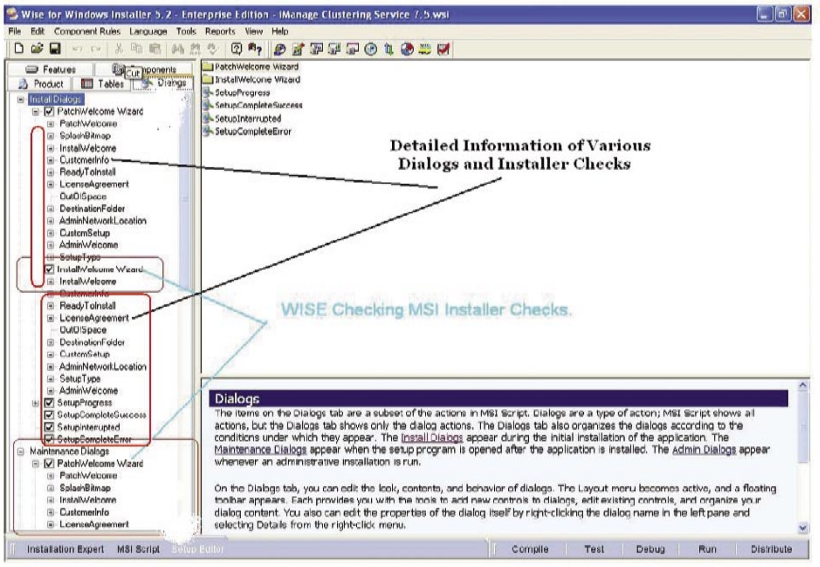


Figure 1. *Elements involved in the capture process*

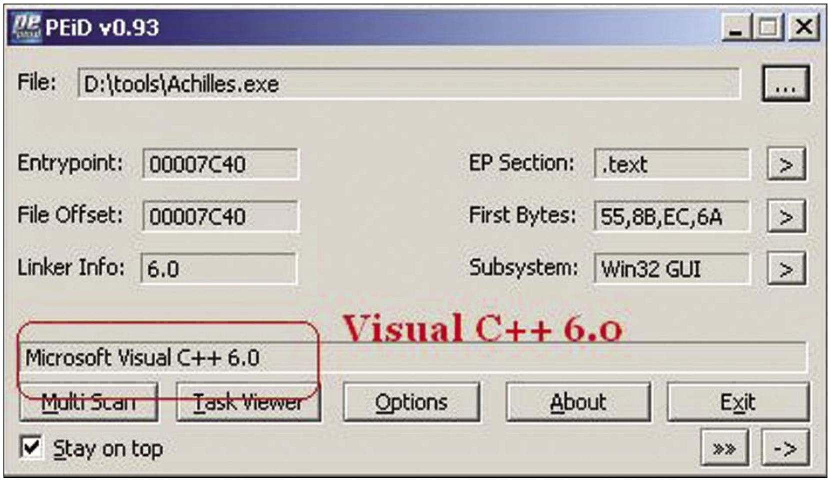

Wise for Windows Installer 5.2 - Enterprise Edition - iManage Clustering Service 7.5.wsi
File Edit Component Rules Language Tools Reports View Help
Features Components Product Tables Dialogs
Install Dialogs
PatchWelcome Wizard
InstallWelcome Wizard
SetupProgress
SetupCompleteSuccess
SetupInterrupted
SetupCompleteError
Detailed Information of Various Dialogs and Installer Checks
WISE Checking MSI Installer Checks.
Installation Expert MSI Script Setup Editor
Compile Test Debug Run Distribute

Dialogs

The items on the Dialogs tab are a subset of the actions in MSI Script. Dialogs are a type of action; MSI Script shows all actions, but the Dialogs tab shows only the dialog actions. The Dialogs tab also organizes the dialogs according to the conditions under which they appear. The Install Dialogs appear during the initial installation of the application. The Maintenance Dialogs appear when the setup program is opened after the application is installed. The Admin Dialogs appear whenever an administrative installation is run.

On the Dialogs tab, you can edit the look, contents, and behavior of dialogs. The Layout menu becomes active, and a floating toolbar appears. Each provides you with the tools to add new controls to dialogs, edit existing controls, and organize your dialog content. You also can edit the properties of the dialog itself by right-clicking the dialog name in the left pane and selecting Details from the right-click menu.

Figure 2. *Wise in Action*



File: D:\tools\Achilles.exe

Entrypoint: 00007C40

EP Section: .text

File Offset: 00007C40

First Bytes: 55,8B,EC,6A

Linker Info: 6.0

Subsystem: Win32 GUI

Microsoft Visual C++ 6.0

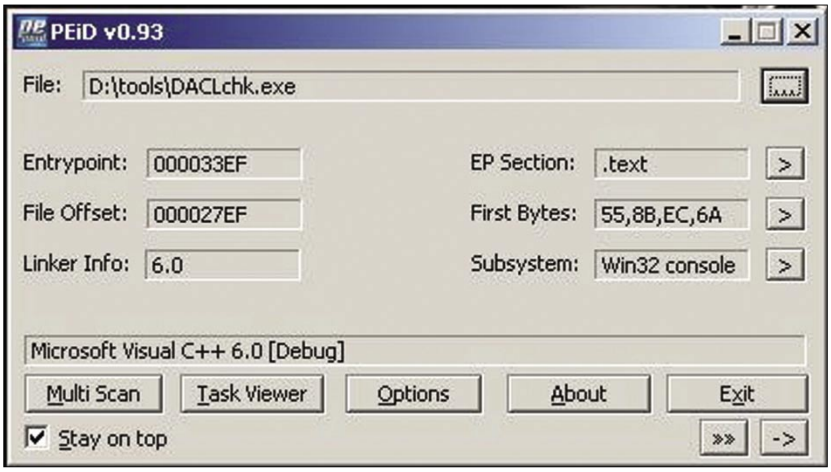
☑ Stay on top

Figure 3. *Executable Achilles is Identified with PEI*

the unpacking procedure should be applied with help of a related unpacker. This whole process of leveraging packer information and unpacking is called as *Sanitizing Binary*. It directly presents the format of an executable prior starting reverse engineering process.

So these four factors should be in a mind of a Reverse Engineer while performing Level 2 analysis.

The basic of reversing a binary starts from analyzing MSI installers. The installers are used when number of binaries are packed collectively which serves the software installation process. It is imperative to undertake the intricacies of windows installer because if the installer service is not properly configured in the system, the software execution may be marginalized. This is because the installer is not able to decompress the files in a right sequential manner there by tempering the dependencies of software. The installer check is always performed by WISE enterprise edition. This software is very reliable in analyzing the cross functionality of objects that are providing software registration mechanism. When you analyze a MSI file in WISE, there are number of dialogs displayed comprising of different functionality structure. These dialogs include license agreement, customer info etc. and get displayed during installation process. The WISE enables you to circumvent the properties of dialogs to some extent and provides control. This enables reverse engineer to test the software installer.



PEiD v0.93

File: D:\tools\DACLchk.exe

Entrypoint: 000033EF

EP Section: .text

File Offset: 000027EF

First Bytes: 55,8B,EC,6A

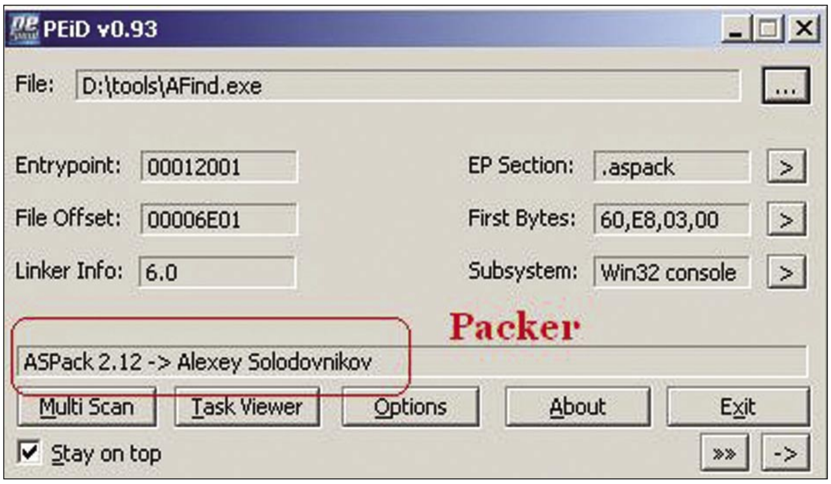
Linker Info: 6.0

Subsystem: Win32 console

Microsoft Visual C++ 6.0 [Debug]

☑ Stay on top

Figure 4. *Executable DachlChk is Identified with PEID*



PEiD v0.93

File: D:\tools\AFind.exe

Entrypoint: 00012001

EP Section: .aspack

File Offset: 00006E01

First Bytes: 60,E8,03,00

Linker Info: 6.0

Subsystem: Win32 console

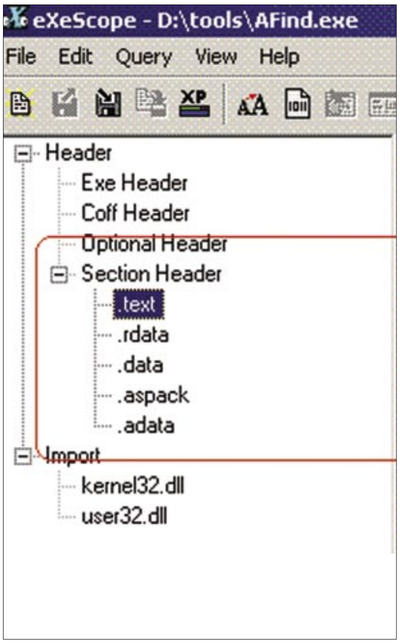
ASPack 2.12 -> Alexey Solodovnikov

☑ Stay on top

Figure 5. *Target AFind.exe is Packed with ASPPack*

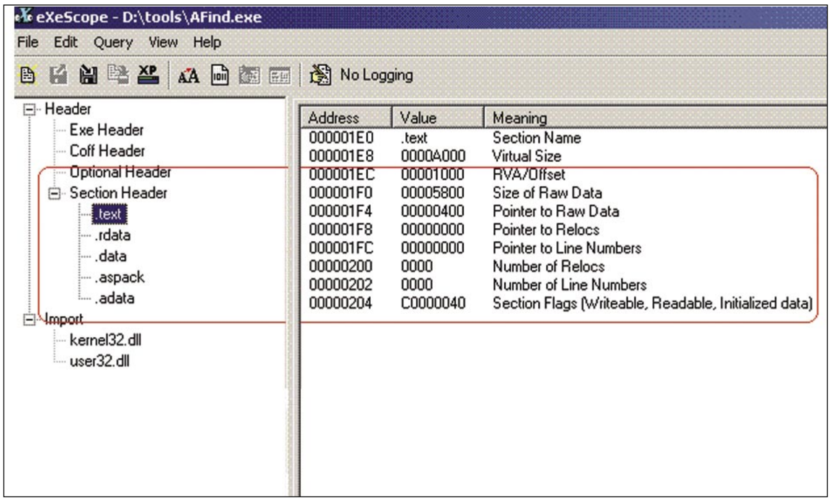
The WISE provide recompilation facility to remake the installer with altered properties. Some installers use CAB file, in that case a new CAB file will be generated after recompilation (Figure 2).

The above presented WISE layout provides much information regarding an installer. All the dialogs are arranged in a hierarchical way in the form of tree. This representation depicts the flow in which these dialogs are going to be executed. One can easily interpret the properties of any dialog. So control and time constraint are marginal in a way WISE provides functionality. One can see Installer Version Wizard entry above under which all major installer modules are defined. The reverse engineer can easily locate the Installer function that provides check. For Example, if a function named as InstallApplication exists one can get to it by looking at the event related to it. The event provides functional specificity of that dialog. Generally InstallApplication takes parameter to true after the registration check is performed. The Reverse Engineer makes that condition to true always by supplying argument as 1. Afterwards, the MSI file is recompiled and the



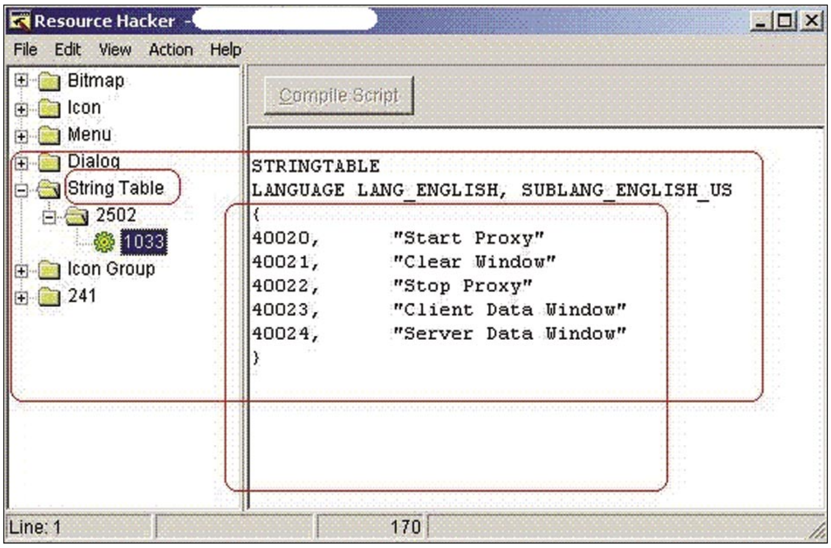
eXeScope - D:\tools\AFind.exe

Figure 6. *Hierarchical View of Headers*



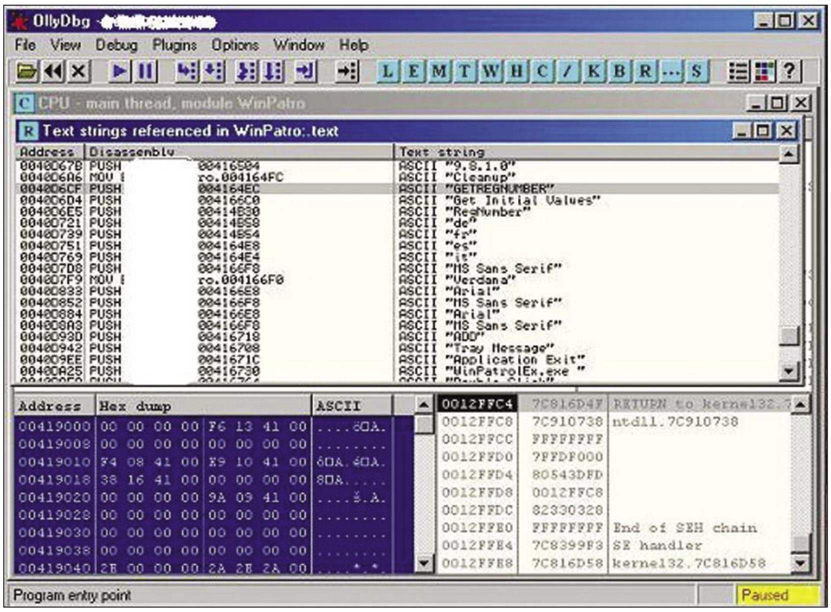
Address	Value	Meaning
000001E0	.text	Section Name
000001E8	0000A000	Virtual Size
000001EC	00001000	RVA/Offset
000001F0	00005800	Size of Raw Data
000001F4	00000400	Pointer to Raw Data
000001F8	00000000	Pointer to Relocs
000001FC	00000000	Pointer to Line Numbers
00000200	0000	Number of Relocs
00000202	0000	Number of Line Numbers
00000204	C0000040	Section Flags (Writeable, Readable, Initialized data)

Figure 7. *Afind.exe is edited with Exescope*



```
STRINGTABLE
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
{
40020,      "Start Proxy"
40021,      "Clear Window"
40022,      "Stop Proxy"
40023,      "Client Data Window"
40024,      "Server Data Window"
}
```

Figure 8. *Resource Hacker in Action*


OllyDbg
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
CPU - main thread, module WinPatro
Text strings referenced in WinPatro:.text
Program entry point
Paused

```
Address  Disassembly                Text string
0040D67B PUSH          00416504     ASCII "9.8.1.0"
0040D6A6 MOV E         ro.004164FC  ASCII "Cleanup"
0040D6CF PUSH          004164EC     ASCII "GETREGNUMBER"
0040D6D4 PUSH          004166C0     ASCII "Get Initial Values"
0040D6E5 PUSH          00414B30     ASCII "RegNumber"
0040D721 PUSH          00414B58     ASCII "de"
0040D739 PUSH          00414B54     ASCII "fr"
0040D751 PUSH          004164E8     ASCII "es"
0040D769 PUSH          004164E4     ASCII "it"
0040D7D8 PUSH          004166F8     ASCII "MS Sans Serif"
0040D7F9 MOV E         ro.004166F0  ASCII "Verdana"
0040D833 PUSH          004166E8     ASCII "Arial"
0040D852 PUSH          004166F8     ASCII "MS Sans Serif"
0040D884 PUSH          004166E8     ASCII "Arial"
0040D8A3 PUSH          004166F8     ASCII "MS Sans Serif"
0040D93D PUSH          00416718     ASCII "ADD"
0040D942 PUSH          00416708     ASCII "Tray Message"
0040D9EE PUSH          0041671C     ASCII "Application Exit"
0040DA25 PUSH          00416730     ASCII "WinPatrolEx.exe "

Address  Hex dump                ASCII
00419000 00 00 00 00 F6 13 41 00 ....ö□A.
00419008 00 00 00 00 00 00 00 00 ........
00419010 F4 08 41 00 E9 10 41 00 ô□A.é□A.
00419018 38 16 41 00 00 00 00 00 8□A.....
00419020 00 00 00 00 9A 09 41 00 ....š.A.
00419028 00 00 00 00 00 00 00 00 ........
00419030 00 00 00 00 00 00 00 00 ........
00419038 00 00 00 00 00 00 00 00 ........
00419040 2E 00 00 00 2A 2E 2A 00 ....*.*.

0012FFC4  7C816D4F  RETURN to kernel32.7
0012FFC8  7C910738  ntdll.7C910738
0012FFCC  FFFFFFFF
0012FFD0  7FFDF000
0012FFD4  80543DFD
0012FFD8  0012FFC8
0012FFDC  82330328
0012FFE0  FFFFFFFF  End of SEH chain
0012FFE4  7C8399F3  SE handler
0012FFE8  7C816D58  kernel32.7C816D58
```

Figure 9. *Traversing Referenced String*

condition is injected in it. It enables the installer to find the condition always true and without performing any extensive checks the software is installed. This process is utilized by the professionals a lot.

But one cannot be sure that every software works on this pattern. This is termed to be PRE-tempering of software installers. It proves beneficial most of the time but cannot be implemented all the time to various software. For that we have to jum p to core of the software instructions. In this the reader is going to encounter the cross checks of registration.

[1] *Analyzing The Curvature of a Binary*: This means gathering information regarding the curvature of an executable. It comprises the language in which it is written and protection mechanism used in it. It is crucial to leverage information based on this information. In this, a Reverse Engineer tries to find the identity of an executable. This technique is called PEID Traversing. It provides information regarding:

- The language in which a specific executable is constructed. It further helps a reverse engineer to understand the semantics of language used and the required inter-modular designing of functions, or the import and export of various functions in modules. See Figure 3.

Figure 3 depicts an executable that is written in Microsoft Visual C++. The subsystem specified is Win 32 GUI (Graphical User Interface). So the base language is extracted easily. No protection mechanism is used as such in this.

- It provides the state of an executable. The state here corresponds to the Debug and Release build of an executable. This is very important from a reverse engineering point of view. If an executable is found in Debug state, then it is very easy to reverse it and debugging can be performed stringently (Figure 4).

Figure 4 presents a structural view of an executable and showing it is in Debug state. This means that the build type is Debug and the symbols are present in it. The state is clearly mentioned. The subsystem is shown as Console. A simple debugging operation of this executable in Olly Debugger easily dissects it internally.

- It provides an overview of the *Packing Mechanism*. There is a great difference between a protection mechanism of a software and simple executable. The main difference lies in the packing of code. It is easy to compress an already compiled executable with a packer. The packer obfuscates the code in the data and stack segments of an executable and makes it hard to reverse. The ID checking provides information on the packing status and the kind of packer used. A packer is defined as a program that packs an application code based on certain algorithm. It is necessary because unpacking of the executable is required to reverse it further. If this process is not implemented and unpacking is not done then it becomes very hard to disseminate the parameters of an executable. Let's see how to look at the PEID of target executable (Figure 5).

It shows that the executable is packed with ASPack program. In this way a Reverse Engineer is able to find the relative statistics of an executable which enhances the analytical view. It encompasses the properties of an executable.

[2] *Structural Design of a Binary*: This covers the checking of the structural design of the binary that is to be reverse engineered. The understanding of binary structure is necessary and how it is designed (Figure 6).

The process is termed as *PE Editing*. It is composed of reversing a binary with an editor that dissects it on the pattern of a Windows PE executable. As a result of this, an executable is disseminated into required headers, section headers and import /export functions. The header object is divided into Exe Headers, Coff Header, Optional Header and Section Header.

Every single header consists of requisite information of the binary. An editor projects information of a binary in a tree format which is composed of various nodes displaying different objects. The Section Hader is divided into three objects which are *.text*, *.rdata* and *.data*. These objects hold unique information related to the binary. Various import modules depict the kind of functions called from system dynamic link libraries and the cross referencing between them. Let's have a look at *.text* sectional object and the information it presents when the executable is edited.

Listing 1. *Import DLL Summary*

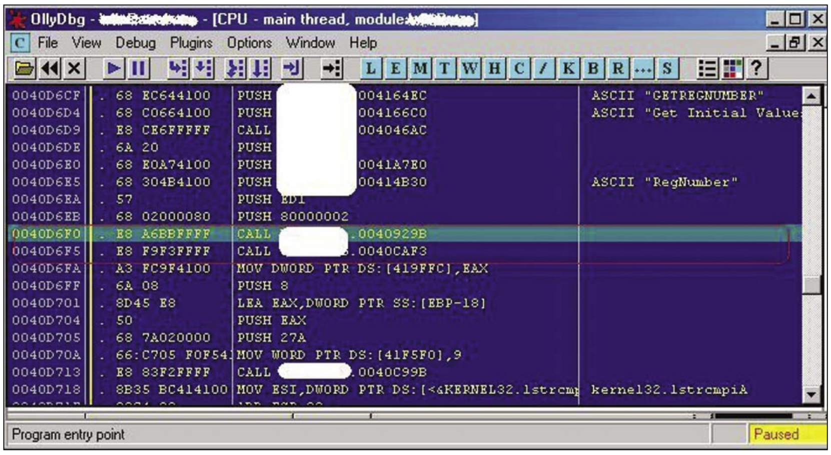
```
Executable modules
Base      Size      Entry     Name       File version      Path
00400000  0003C000  0040E753  Win Patro   9, 8, 1, 0        C:\Program Files\BillP Studios\Afindl\Afindl.exe
10000000  0000D000  100012BE  PATROLPR   1.2.0.0           C:\Program Files\BillP Studios\Afindl\PATROLPRO.DLL
6BD00000  0000D000  6BD01A10   SYNCOR11   1.2.3             C:\WINNT\system32\SYNCOR11.DLL
759B0000  00006000  759B1A6A   LZ32       5.00.2195.6611    C:\WINNT\system32\LZ32.DLL
77570000  00030000  77574164   WINMM      5.00.2161.1       C:\WINNT\system32\WINMM.dll
77820000  00007000  77821334   VERSION    5.00.2195.6623    C:\WINNT\system32\VERSION.dll
77A50000  000F7000  77A52CE2   ole32      5.00.2195.6692    C:\WINNT\system32\ole32.dll
77B50000  00089000  77B56484   COMCTL32   5.81              C:\WINNT\system32\COMCTL32.dll
77C70000  0004A000  77C798A5   SHLWAPI    5.00.3502.6601    C:\WINNT\system32\SHLWAPI.DLL
77D30000  00071000  77D34884   RPCRT4     5.00.2195.6701    C:\WINNT\system32\RPCRT4.DLL
77E10000  00065000  77E311C5   USER32     5.00.2195.6688    C:\WINNT\system32\USER32.DLL
77F40000  0003C000             GDI32      5.00.2195.6660    C:\WINNT\system32\GDI32.DLL
77F80000  0007B000              ntdll      5.00.2195.6685    C:\WINNT\system32\ntdll.dll
782F0000  00248000  782F1FE9  SHELL32    5.00.3700.6705    C:\WINNT\system32\SHELL32.dll
7C2D0000  00062000  7C2D17E4  ADVAPI32   5.00.2195.6710    C:\WINNT\system32\ADVAPI32.DLL
7C4E0000  000B9000  7C4ECE51  KERNEL32   5.00.2195.6688    C:\WINNT\system32\KERNEL32.DLL
```

Listing 2. *Disassembled View*

```
0040D6CF  |. 68 EC644100    PUSH Afind.004164EC               ;  ASCII "GETREGNUMBER"
0040D6D4  |. 68 C0664100    PUSH Afind.004166C0               ;  ASCII "Get Initial Values"
0040D6D9  |. E8 CE6FFFFF    CALL Afind.004046AC
0040D6DE  |. 6A 20          PUSH 20
0040D6E0  |. 68 E0A74100    PUSH Afind.0041A7E0
0040D6E5  |. 68 304B4100    PUSH Afind.00414B30               ;  ASCII "RegNumber"
0040D6EA  |. 57             PUSH EDI
0040D6EB  |. 68 02000080    PUSH 80000002
```

Figure 7 presents the information extracted from the *.text* object. It is comprised of the Relative Virtual Address Offset, Relocation Pointers, Section flags, etc. In this way editing a binary is considered a good approach to reversing a binary.

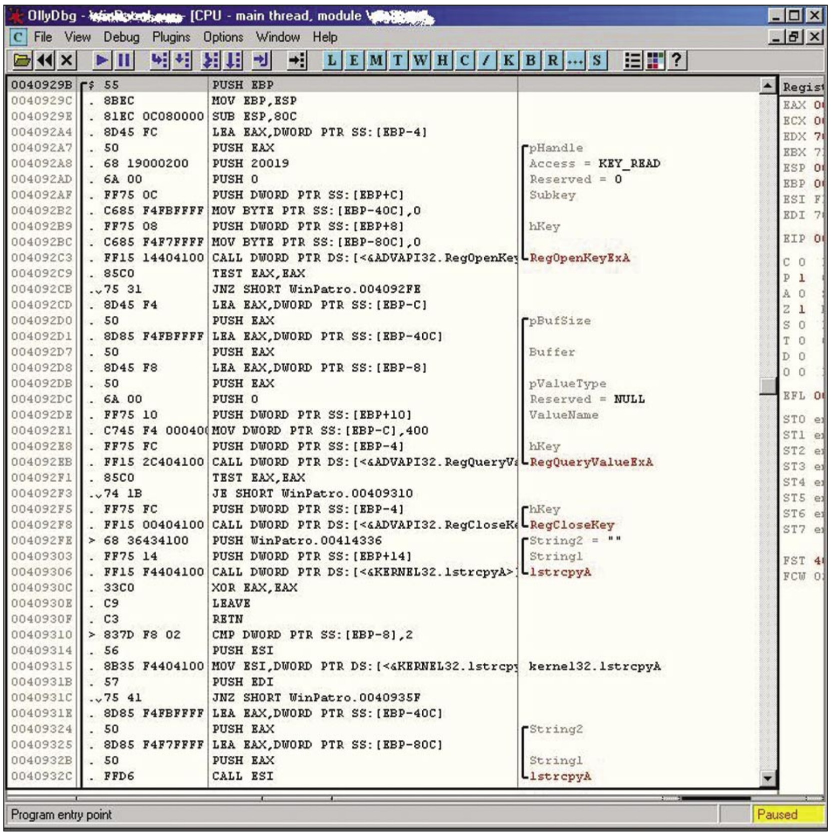
[3] *Hacking Binary Resources*: This technique comes in handy when a Reverse Engineer is analyzing a GUI based binary. As we know, any GUI application is compiled with a number of system resources such as icons, menus, drop boxes, bitmaps, string tables, dialog boxes, etc. The resources adhere to certain functions that are called directly when the resource is initialized. It depends on the binary and the way it is written. It is essential to edit a binary based on the resources used in it. The binary is reversed on the standard benchmarks. The process is called *Stripping Binary Resources*. In this process the kind of resources used in the building of a binary is extracted with the help of Resource Hacker. This tool is flexible and practically applicable in viewing the resources used in a simulating a binary as Figure 8 shows.


OllyDbg - [redacted] - [CPU - main thread, module [redacted]]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?

```
0040D6CF  . 68 EC644100    PUSH [redacted]004164EC       ASCII "GETREGNUMBER"
0040D6D4  . 68 C0664100    PUSH         004166C0         ASCII "Get Initial Value
0040D6D9  . E8 CE6FFFFF    CALL         004046AC
0040D6DE  . 6A 20          PUSH
0040D6E0  . 68 E0A74100    PUSH         0041A7E0
0040D6E5  . 68 304B4100    PUSH         00414B30         ASCII "RegNumber"
0040D6EA  . 57             PUSH EDI
0040D6EB  . 68 02000080    PUSH 80000002
0040D6F0  . E8 A6BBFFFF    CALL [redacted].0040929B
0040D6F5  . E8 F9F3FFFF    CALL [redacted].0040CAF3
0040D6FA  . A3 FC9F4100    MOV DWORD PTR DS:[419FFC],EAX
0040D6FF  . 6A 08          PUSH 8
0040D701  . 8D45 E8        LEA EAX,DWORD PTR SS:[EBP-18]
0040D704  . 50             PUSH EAX
0040D705  . 68 7A020000    PUSH 27A
0040D70A  . 66:C705 F0F54] MOV WORD PTR DS:[41F5F0],9
0040D713  . E8 83F2FFFF    CALL [redacted].0040C99B
0040D718  . 8B35 BC414100  MOV ESI,DWORD PTR DS:[<&KERNEL32.lstrcm kernel32.lstrcmpiA
```

Program entry point Paused

Figure 10. *Checking Function Callings*


OllyDbg - [redacted] [CPU - main thread, module [redacted]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?
Regist
EAX
ECX
EDX
EBX
ESP
EBP
ESI
EDI
EIP
C 0
P 1
A 0
Z 1
S 0
T 0
D 0
O 0
EFL
ST0
ST1
ST2
ST3
ST4
ST5
ST6
ST7
FST
FCW

```
0040929B r$ 55             PUSH EBP
0040929C  . 8BEC           MOV EBP,ESP
0040929E  . 81EC 0C080000  SUB ESP,80C
004092A4  . 8D45 FC        LEA EAX,DWORD PTR SS:[EBP-4]
004092A7  . 50             PUSH EAX                                 rpHandle
004092A8  . 68 19000200    PUSH 20019                               |Access = KEY_READ
004092AD  . 6A 00          PUSH 0                                   |Reserved = 0
004092AF  . FF75 0C        PUSH DWORD PTR SS:[EBP+C]                |Subkey
004092B2  . C685 F4FBFFFF  MOV BYTE PTR SS:[EBP-40C],0              |
004092B9  . FF75 08        PUSH DWORD PTR SS:[EBP+8]                |hKey
004092BC  . C685 F4F7FFFF  MOV BYTE PTR SS:[EBP-80C],0              |
004092C3  . FF15 14404100  CALL DWORD PTR DS:[<&ADVAPI32.RegOpenKey LRegOpenKeyExA
004092C9  . 85C0           TEST EAX,EAX
004092CB  .v75 31          JNZ SHORT WinPatro.004092FE
004092CD  . 8D45 F4        LEA EAX,DWORD PTR SS:[EBP-C]
004092D0  . 50             PUSH EAX                                 rpBufSize
004092D1  . 8D85 F4FBFFFF  LEA EAX,DWORD PTR SS:[EBP-40C]           |
004092D7  . 50             PUSH EAX                                 |Buffer
004092D8  . 8D45 F8        LEA EAX,DWORD PTR SS:[EBP-8]             |
004092DB  . 50             PUSH EAX                                 |pValueType
004092DC  . 6A 00          PUSH 0                                   |Reserved = NULL
004092DE  . FF75 10        PUSH DWORD PTR SS:[EBP+10]               |ValueName
004092E1  . C745 F4 000400 MOV DWORD PTR SS:[EBP-C],400             |
004092E8  . FF75 FC        PUSH DWORD PTR SS:[EBP-4]                |hKey
004092EB  . FF15 2C404100  CALL DWORD PTR DS:[<&ADVAPI32.RegQueryVa LRegQueryValueExA
004092F1  . 85C0           TEST EAX,EAX
004092F3  .v74 1B          JE SHORT WinPatro.00409310
004092F5  . FF75 FC        PUSH DWORD PTR SS:[EBP-4]                rhKey
004092F8  . FF15 00404100  CALL DWORD PTR DS:[<&ADVAPI32.RegCloseKe LRegCloseKey
004092FE  > 68 36434100    PUSH WinPatro.00414336                   rString2 = ""
00409303  . FF75 14        PUSH DWORD PTR SS:[EBP+14]               |String1
00409306  . FF15 F4404100  CALL DWORD PTR DS:[<&KERNEL32.lstrcpyA>] LlstrcpyA
0040930C  . 33C0           XOR EAX,EAX
0040930E  . C9             LEAVE
0040930F  . C3             RETN
00409310  > 837D F8 02     CMP DWORD PTR SS:[EBP-8],2
00409314  . 56             PUSH ESI
00409315  . 8B35 F4404100  MOV ESI,DWORD PTR DS:[<&KERNEL32.lstrcpy kernel32.lstrcpyA
0040931B  . 57             PUSH EDI
0040931C  .v75 41          JNZ SHORT WinPatro.0040935F
0040931E  . 8D85 F4FBFFFF  LEA EAX,DWORD PTR SS:[EBP-40C]
00409324  . 50             PUSH EAX                                 rString2
00409325  . 8D85 F4F7FFFF  LEA EAX,DWORD PTR SS:[EBP-80C]           |
0040932B  . 50             PUSH EAX                                 |String1
0040932C  . FFD6           CALL ESI                                 LlstrcpyA
```

Program entry point Paused

Figure 11. *Structural View of Disassembled View*

The resources are placed in a hierarchy from top to bottom on the left side. The string table node is opened and it is projecting the information regarding strings used in a binary. These strings provide information regarding the association with different type of functions that are used by a binary. Although this resource Handling method is used in cracking certain executables or crack programs, this technique is very flexible and is one of the favorable approaches of reverse engineers.

[4] *Incorporating DLL check Through Import Address Table*: It is also a very good practice of analyzing. It enables a Reverse Engineer to look at the Dynamic Link Libraries loaded during execution of a binary. This process is summarized to check any specific DLL loaded in the memory that affects the working of a binary.

Sometimes a manually designed DLL is coded by the developers to cross check the objects in a binary for certain purposes. Thus, if any added DLL is found it becomes easy to dissect it. First, check the associated remote events. The import DLL of the required software is summarized in Listing 1.

This clearly indicates the import address table of a different module which is loaded during the time of execution. No specific DLL other than the system's DLLs can be seen. This step is crucial to traverse through the DLL table.

[5] *Traversing the Referenced Strings*: This is one of the finest methods to search a specific module in a binary by looking at the strings. This process is termed as *Trapping Strings*. These strings are passed to the core instructions. Then, it comes to an arduous task for the Reverse Engineer – searching through the whole code. This technique comes in handy because a string reference address is provided in a Debugger.

Listing 3. *Disassembled View of Registry Functions*

```
0040929B  /$ 55             PUSH EBP
0040929C  |. 8BEC           MOV EBP,ESP
0040929E  |. 81EC 0C080000  SUB ESP,80C
004092A4  |. 8D45 FC        LEA EAX,DWORD PTR SS:[EBP-4]
004092A7  |. 50             PUSH EAX                                 ; /pHandle
004092A8  |. 68 19000200    PUSH 20019                               ; |Access = KEY_READ
004092AD  |. 6A 00          PUSH 0                                   ; |Reserved = 0
004092AF  |. FF75 0C        PUSH DWORD PTR SS:[EBP+C]                ; |Subkey
004092B2  |. C685 F4FBFFFF >MOV BYTE PTR SS:[EBP-40C],0              ; |
004092B9  |. FF75 08        PUSH DWORD PTR SS:[EBP+8]                ; |hKey
004092BC  |. C685 F4F7FFFF >MOV BYTE PTR SS:[EBP-80C],0              ; |
004092C3  |. FF15 14404100  CALL DWORD PTR DS:[<&ADVAPI32.RegOpenKey>; \RegOpenKeyExA
004092C9  |. 85C0           TEST EAX,EAX
004092CB  |. 75 31          JNZ SHORT Afind.004092FE
004092CD  |. 8D45 F4        LEA EAX,DWORD PTR SS:[EBP-C]
004092D0  |. 50             PUSH EAX                                 ; /pBufSize
004092D1  |. 8D85 F4FBFFFF  LEA EAX,DWORD PTR SS:[EBP-40C]           ; |
004092D7  |. 50             PUSH EAX                                 ; |Buffer
004092D8  |. 8D45 F8        LEA EAX,DWORD PTR SS:[EBP-8]             ; |
004092DB  |. 50             PUSH EAX                                 ; |pValueType
004092DC  |. 6A 00          PUSH 0                                   ; |Reserved = NULL
004092DE  |. FF75 10        PUSH DWORD PTR SS:[EBP+10]               ; |ValueName
004092E1  |. C745 F4 000400>MOV DWORD PTR SS:[EBP-C],400             ; |
004092E8  |. FF75 FC        PUSH DWORD PTR SS:[EBP-4]                ; |hKey
004092EB  |. FF15 2C404100  CALL DWORD PTR DS:[<&ADVAPI32.RegQueryVa>; \RegQueryValueExA
004092F1  |. 85C0           TEST EAX,EAX
004092F3  |. 74 1B          JE SHORT Afind.00409310
004092F5  |. FF75 FC        PUSH DWORD PTR SS:[EBP-4]                ; /hKey
004092F8  |. FF15 00404100  CALL DWORD PTR DS:[<&ADVAPI32.RegCloseKe>; \RegCloseKey
004092FE  |> 68 36434100    PUSH Afind.00414336                  ; /String2 = ""
00409303  |. FF75 14        PUSH DWORD PTR SS:[EBP+14]               ; |String1
00409306  |. FF15 F4404100  CALL DWORD PTR DS:[<&KERNEL32.lstrcpyA>] ; \lstrcpyA
0040930C  |. 33C0           XOR EAX,EAX
0040930E  |. C9             LEAVE
0040930F  |. C3             RETN
00409310  |> 837D F8 02     CMP DWORD PTR SS:[EBP-8],2
00409314  |. 56             PUSH ESI
00409315  |. 8B35 F4404100  MOV ESI,DWORD PTR DS:[<&KERNEL32.lstrcpy>;  KERNEL32.lstrcpyA
0040931B  |. 57             PUSH EDI
0040931C  |. 75 41          JNZ SHORT Afind.0040935F
0040931E  |. 8D85 F4FBFFFF  LEA EAX,DWORD PTR SS:[EBP-40C]
00409324  |. 50             PUSH EAX                                 ; /String2
00409325  |. 8D85 F4F7FFFF  LEA EAX,DWORD PTR SS:[EBP-80C]           ; |
0040932B  |. 50             PUSH EAX                                 ; |String1
0040932C  |. FFD6           CALL ESI                                 ; \lstrcpyA
0040932E  |. BF FF030000    MOV EDI,3FF
00409333  |. 57             PUSH EDI                                 ; /DestSizeMax => 3FF (1023.)
00409334  |. 8D85 F4FBFFFF  LEA EAX,DWORD PTR SS:[EBP-40C]           ; |
0040933A  |. 50             PUSH EAX                                 ; |DestString
0040933B  |. 8D85 F4F7FFFF  LEA EAX,DWORD PTR SS:[EBP-80C]           ; |
00409341  |. 50             PUSH EAX                                 ; |SrcString
00409342  |. FF15 F0404100  CALL DWORD PTR DS:[<&KERNEL32.ExpandEnvi>; \ExpandEnvironmentStringsA
00409348  |. 3BC7           CMP EAX,EDI
0040934A  |. 76 13          JBE SHORT Afind.0040935F
0040934C  |. 8D85 F4F7FFFF  LEA EAX,DWORD PTR SS:[EBP-80C]
00409352  |. 68 105C4100    PUSH Afind.00415C10                  ;  ASCII
   "Registry Error #1023: String can not be expanded"
00409357  |. 50             PUSH EAX
00409358  |. E8 4FB3FFFF    CALL Afind.004046AC
0040935D  |. 59             POP ECX
0040935E  |. 59             POP ECX
0040935F  |> FF75 FC        PUSH DWORD PTR SS:[EBP-4]                ; /hKey
00409362  |. FF15 00404100  CALL DWORD PTR DS:[<&ADVAPI32.RegCloseKe>; \RegCloseKey
```

Thus, you can find the string related to any operation and it is redirected to the required code for further analysis (see Figure 9).

By incorporating this technique large code analysis becomes easier. In Figure 9 you can see that GETREGNUMBER string is passed. A reference address is provided with respect to that. This address provides some information on the use of this function in the defined code of software. In this process specific information is collected, as you can see below:

```
Text strings referenced in Afind:
   .text, item 641 Address=0040D6CF
   Disassembly=PUSH afind.004164EC Text
    string=ASCII "GETREGNUMBER"
Text strings referenced in Afind:
   .text, item 642 Address=0040D6D4
    Disassembly=PUSH afind.004166C0 Text
    string=ASCII "Get Initial Values"
Text strings referenced in Afind:
   .text, item 643 Address=0040D6E5 Dis
   assembly=PUSHafind.00414B30 Text
    string=ASCII "RegNumber"
```

The above mentioned strings are used for code analysis related to specific process only. Reviewing whole code line by line is of no use to a Reverse Engineer.

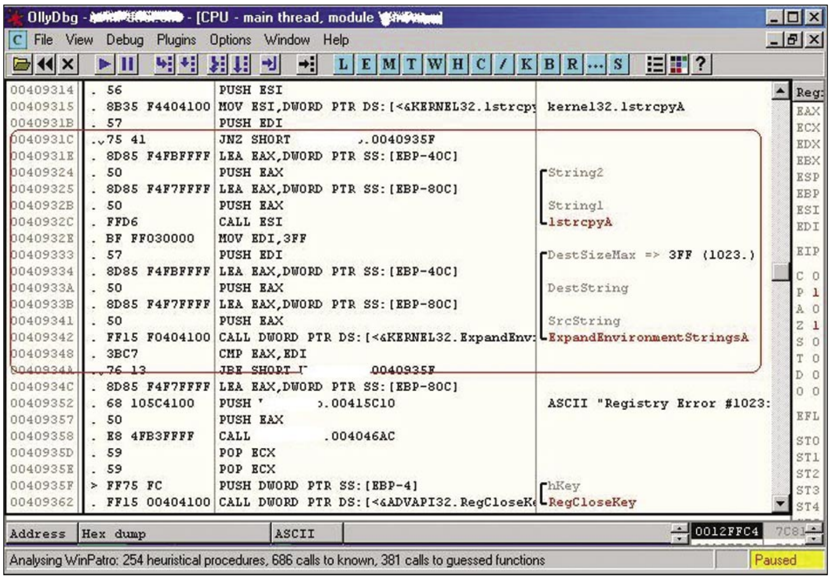
[6] *Analyzing Code Flow in Binaries*: At this point, we have got the structural design of the binary that is a must-know about parameters. For better understanding of the code simulation, it is important to determine the code flow of a binary. In order to execute required functios we need to execute the instructions collected together. The process of code flow analysis is critical from an analytical point of view. The cross referenced functions are analyzed. The CALL instruction, after the passing of strings, is used to call the remote functions. This process is shown in Figure 10.

We can see two call procedures that are undertaken in Figure 10. The first one is at address `0040929B` and second call procedure is at `0040CAF3`. These are the calling addresses where the remote function is defined. The inclusion of these functions is directly referenced by calling CALL procedure. To dig deeper, a Reverse Engineer has to traverse through these remote modules in order to analyze other codes. It makes it easier to understand the code flow and lets us look for other

Listing 4. *Instructions to be manipulated*

```
0040D71E   |. 83C4 28        ADD ESP,28
0040D721   |. 68 584B4100    PUSH Afind.00414B58                    ; /String2
                   = "de"
0040D726   |. 8D45 E8        LEA EAX,DWORD PTR SS:[EBP-18]             ; |
0040D729   |. 50             PUSH EAX                                  ;
                   |String1
0040D72A   |. FFD6           CALL ESI                                  ;
                   \lstrcmpiA

0040D72C      85C0           TEST EAX,EAX
0040D72E   |. 75 09          JNZ SHORT Afind.0040D739
```

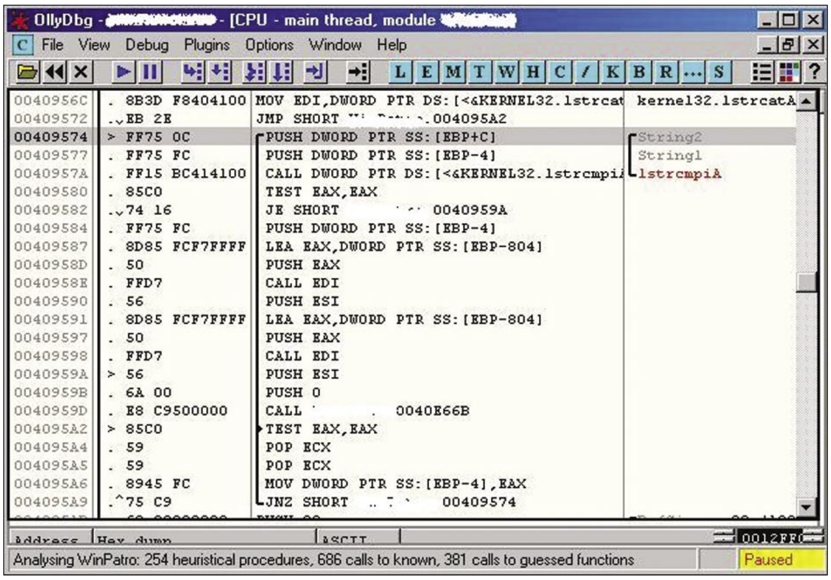


OllyDbg - [redacted] - [CPU - main thread, module [redacted]

```
00409314  . 56             PUSH ESI
00409315  . 8B35 F4404100  MOV ESI,DWORD PTR DS:[<&KERNEL32.lstrcp  kernel32.lstrcpyA
0040931B  . 57             PUSH EDI
0040931C  .v75 41          JNZ SHORT          .0040935F
0040931E  . 8D85 F4FBFFFF  LEA EAX,DWORD PTR SS:[EBP-40C]
00409324  . 50             PUSH EAX                               ┌String2
00409325  . 8D85 F4F7FFFF  LEA EAX,DWORD PTR SS:[EBP-80C]         │
0040932B  . 50             PUSH EAX                               │String1
0040932C  . FFD6           CALL ESI                               └lstrcpyA
0040932E  . BF FF030000    MOV EDI,3FF
00409333  . 57             PUSH EDI                               ┌DestSizeMax => 3FF (1023.)
00409334  . 8D85 F4FBFFFF  LEA EAX,DWORD PTR SS:[EBP-40C]         │
0040933A  . 50             PUSH EAX                               │DestString
0040933B  . 8D85 F4F7FFFF  LEA EAX,DWORD PTR SS:[EBP-80C]         │
00409341  . 50             PUSH EAX                               │SrcString
00409342  . FF15 F0404100  CALL DWORD PTR DS:[<&KERNEL32.ExpandEnv└ExpandEnvironmentStringsA
00409348  . 3BC7           CMP EAX,EDI
0040934A  .v76 13          JBE SHORT I            0040935F
0040934C  . 8D85 F4F7FFFF  LEA EAX,DWORD PTR SS:[EBP-80C]
00409352  . 68 105C4100    PUSH '          .00415C10              ASCII "Registry Error #1023:
00409357  . 50             PUSH EAX
00409358  . E8 4FB3FFFF    CALL            .004046AC
0040935D  . 59             POP ECX
0040935E  . 59             POP ECX
0040935F  > FF75 FC        PUSH DWORD PTR SS:[EBP-4]              ┌hKey
00409362  . FF15 00404100  CALL DWORD PTR DS:[<&ADVAPI32.RegCloseK└RegCloseKey
```

Analysing WinPatro: 254 heuristical procedures, 686 calls to known, 381 calls to guessed functions

Figure 12. *Detail Lookup of Instructions*



OllyDbg - [redacted] - [CPU - main thread, module [redacted]

```
0040956C  . 8B3D F8404100  MOV EDI,DWORD PTR DS:[<&KERNEL32.lstrcat  kernel32.lstrcatA
00409572  .vEB 2E          JMP SHORT          .004095A2
00409574  > FF75 0C        ┌PUSH DWORD PTR SS:[EBP+C]               ┌String2
00409577  . FF75 FC        │PUSH DWORD PTR SS:[EBP-4]               │String1
0040957A  . FF15 BC414100  │CALL DWORD PTR DS:[<&KERNEL32.lstrcmpiA└lstrcmpiA
00409580  . 85C0           │TEST EAX,EAX
00409582  .v74 16          │JE SHORT            0040959A
00409584  . FF75 FC        │PUSH DWORD PTR SS:[EBP-4]
00409587  . 8D85 FCF7FFFF  │LEA EAX,DWORD PTR SS:[EBP-804]
0040958D  . 50             │PUSH EAX
0040958E  . FFD7           │CALL EDI
00409590  . 56             │PUSH ESI
00409591  . 8D85 FCF7FFFF  │LEA EAX,DWORD PTR SS:[EBP-804]
00409597  . 50             │PUSH EAX
00409598  . FFD7           │CALL EDI
0040959A  > 56             │PUSH ESI
0040959B  . 6A 00          │PUSH 0
0040959D  . E8 C9500000    │CALL            0040E66B
004095A2  > 85C0           ►TEST EAX,EAX
004095A4  . 59             │POP ECX
004095A5  . 59             │POP ECX
004095A6  . 8945 FC        │MOV DWORD PTR SS:[EBP-4],EAX
004095A9  .^75 C9          └JNZ SHORT           00409574
```

Analysing WinPatro: 254 heuristical procedures, 686 calls to known, 381 calls to guessed functions

Figure 13. *Strings View*

differential code structures. Without wasting any time, the Reverse Engineer can jump to the required address to see what is being called. In Figure 11 the call at `0040929B` is made.

The module points to routine presented in Figure 11. One can look clearly at registry functions that play a crucial part. The required code in this executable is used for some kind of registration process by the executable. The registration process comprises of passing user and registration code. As soon as the strings are passed to the registration argument, a procedure is defined and strings are queried with the registry settings. The system's APIs like RegOpenKey, RegQueryValue and RegCloseKey are used. Once the string is passed through a specified procedure, the strings are compared through `strcmp` function. This is done to check whether strings are processed in the correct manner or not. Our analysis is defined on the basis that are practically feasible.

It is time to look up the output in detail as shown in Figure 12.

This layout is of some concern because direct string compare function is being used. Once the strings are matched and there is success the ExpandEnvironmentStrings module is called and executed. It provides the information on the environmental objects after the string matching operation.

This code is one of the prime points to test registration processes. It is one of the main code section of a dissected binary. Other remote functions will be related to it. The Reverse Engineer further traverses code and finds out what is presented in Figure 13.

The code specified above holds a routine after another string comparison. If strings are compared in a well defined manner then JUMP is allowed to make at the address `0040959A`. The code flow analysis is very helpful in determining the working state of a binary.

[7] *Byte Patching*: It is a technique of changing the flow of decisive instructions. In this, the required byte is patched with manipulated arguments to completely reverse the direction of execution. It means when a single instruction is used to check the condition of authenticity of program, the action can be reversed by tempering the contents of registers. This plays a crucial role in breaking the registration code of software. This process is entirely applicable in CALL/JMP instruction duo.

As we know, these specified instructions are used to control the flow of execution. A vernacular change in instruction alters the state of execution. This is considered to be Flow Tempering and the last step in reversing an application prior to patching in full. The underlined three factors have to be noticed first:

- Checking the protection on installer
- Traversing the Registration check
- Analyzing the algorithm specifically and the context in which it is applied

These factors are crucial for reversing an application.

Let us put it into practice as shown in Listing 2.

This is the code used to dissect the functional calling of *GETREGNUMBER* string. During this analysis the required code is presented (see Listing 3).

This code shows the use of registry functions for querying some

Tools

OllyDbg

Olly Debugger is a user mode debugger. The beauty of Olly is that it appears to have been designed from the ground up as a reversing tool, and as such it has a very powerful built-in disassembler. OllyDbg's greatest strength is in its disassembler, which provides powerful code-analysis features. OllyDbg's code analyzer can identify loops, switch blocks, and other key code structures. One of the most reliable tools preference of any reverse engineer.

Fetch: *http://www.ollydbg.de/*

Resource Hacker

It is Resource hacking tool and it works on the concept of object hooking of *.Res files*. It hooks all the objects present in the binary with properties. It enable the reverse engineer to tamper the characteristics of an object. The another preferential part is the recompiling function of this tool.

Fetch: *http://angusj.com/resourcehacker/*

PEID

PEID is a portable executable identifier tool. This tool provides the information regarding the present structure of a binary.

Fetch: *http://www.peid.info/*

WISE

It support advanced installation authoring in either Windows* Installer (.MSI) or WiseScript formats. With exclusive features for development teams of any size, Wise Installation Studio helps you create high-quality installations for complex environments. It is also used as a reverse engineering tool for analyzing the Binary Installer.

Fetch: *http://www.altiris.com/Products/WiseInstallStudio.aspx*

EXESCOPE

eXeScope can analyze, display various information, and rewrite resources of executable files, that is, EXE, DLL, OCX, etc. without source files.

Fetch: *http://hp.vector.co.jp/authors/VA003525/emysoft.htm#6*

Other tools you can find at *http://exetools.com*

On the 'Net

- *http://www.openrce.org*
- *http://www.openrce.org/blog/browse/aditya_ks*
- *http://www.nynaeve.net/*
- *http://home.arcor.de/idapalace/* – Index of IDAPalace
- *http://www.exetools.com*

About the Author

Aditya K Sood aka 0kn0ck is an independent security researcher and founder of SecNiche Security, a security research arena. He is a regular speaker at conferences like XCON, OWASP, CERT-IN etc. Other projects include Mlabs, CERA, TrioSec etc.
Website: *http://www.secniche.org*

value. The register specific view will let us understand the arguments passed to various functions. The prime aspect is to look after `strcmp` functions and the return values. This shows the flow control because the return value is controlled with JMP/CALL instruction to near and far pointers that then points to certain addresses (see Listing 4).

The the code in Listing 4 is extracted from the reversed view of the software. The Reverse Engineer can analyze the flow. TEST operation is used followed by `strcmp` instruction.

Remember, one can encounter a number of instructions like this in a code. The testing can be performed one by one to check the program flow. This is called *Debugging Iteration*. The reverser manipulates the code as:

```
0040D72A  |. FFD6           CALL ESI
                              ; \
lstrcmpiA
0040D72C     85C0           XOR
  EAX,EAX
0040D72E  |. 75 09          JNZ SHORT
 Afind.0040D739
```

or:

```
0040D72A  |. FFD6           CALL ESI
                                  ;
 \lstrcmpiA
0040D72C     85C0           TEST
 EAX,EAX
0040D72E  |. 75 09          JZ SHORT
                      Afind.0040D739
```

In the first layout the instruction is changed with XOR operation and the rest of code is to remain the same. In the second part a reverser does not temper the TEST instruction but changes the JNZ to JZ. Both the conditions totally change the status of an application. When these bytes are patched with certain other modifications, the executable is considered to be as patched.

Above presented techniques are helpful in examining a binary from scratch.

Conclusion

It has been rightly stated *To have control of the system, you have to capture the source*. This adage holds the reverse engineering nature. Reverse engineering is all about understanding the source of an object and analyzing the working behavior. The real taste of knowledge about internals of any binary executable lies in reverse engineering. This process not only helps in knowing the hidden instances of code but also the inter facial effect with system. The motto is to learn new techniques and the art of reverse engineering. The techniques are useful when a time constraint is subjected during analysis. To complete targets in a required period of time, a good layout of reverse engineering procedure should be implemented. ●