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,
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.

Figure 4. Executable DachlChk is Identified with PEID

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...
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 jump 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

Figure 9. Traversing Referenced String
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 Header is divided into three objects which are .text, .data 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 section object and the information it presents when the executable is edited.

**Listing 1. Import DLL Summary**

<table>
<thead>
<tr>
<th>Executable modules</th>
<th>Base</th>
<th>Size</th>
<th>Entry</th>
<th>Name</th>
<th>File version</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400000</td>
<td>0003C000</td>
<td>0040B753</td>
<td>WinPatro</td>
<td>9, 8, 1, 0</td>
<td>C:\Program Files\BillP Studios\Afind\Afind1.exe</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>00000000</td>
<td>100012BE</td>
<td>PatrolPro</td>
<td>1.2.0.8</td>
<td>C:\Program Files\BillP Studios\Afind\PATROLPRO.DLL</td>
<td></td>
</tr>
<tr>
<td>68000000</td>
<td>000001A0</td>
<td>SYNCOR11</td>
<td>1.2.3</td>
<td></td>
<td>C:\WINNT\system32\SYNCOR11.DLL</td>
<td></td>
</tr>
<tr>
<td>759B0000</td>
<td>00006000</td>
<td>759B1A6A</td>
<td>LE32</td>
<td>5.0.2195.6611</td>
<td>C:\WINNT\system32\LE32.DLL</td>
<td></td>
</tr>
<tr>
<td>77570000</td>
<td>00003000</td>
<td>77574164</td>
<td>WINMM</td>
<td>5.0.2161.1</td>
<td>C:\WINNT\system32\WINMM.dll</td>
<td></td>
</tr>
<tr>
<td>77820000</td>
<td>00007000</td>
<td>77821334</td>
<td>VERSION</td>
<td>5.0.2195.6623</td>
<td>C:\WINNT\system32\VERSION.DLL</td>
<td></td>
</tr>
<tr>
<td>77850000</td>
<td>00007000</td>
<td>77852CE2</td>
<td>ole32</td>
<td>5.0.2195.6622</td>
<td>C:\WINNT\system32\ole32.dll</td>
<td></td>
</tr>
<tr>
<td>77850000</td>
<td>00089000</td>
<td>77856484</td>
<td>COMCTL32</td>
<td>5.81</td>
<td>C:\WINNT\system32\COMCTL32.dll</td>
<td></td>
</tr>
<tr>
<td>777C0000</td>
<td>0004A000</td>
<td>777C99A5</td>
<td>SHLWAPI</td>
<td>5.0.3502.6601</td>
<td>C:\WINNT\system32\SHLWAPI.DLL</td>
<td></td>
</tr>
<tr>
<td>77D30000</td>
<td>00071000</td>
<td>77D34884</td>
<td>RPCRT4</td>
<td>5.0.2195.6701</td>
<td>C:\WINNT\system32\RPCRT4.DLL</td>
<td></td>
</tr>
<tr>
<td>77E11000</td>
<td>00065000</td>
<td>77E111C5</td>
<td>USER32</td>
<td>5.0.2195.6688</td>
<td>C:\WINNT\system32\USER32.DLL</td>
<td></td>
</tr>
<tr>
<td>77F40000</td>
<td>0003C000</td>
<td>77F43500</td>
<td>GD132</td>
<td>5.0.2195.6660</td>
<td>C:\WINNT\system32\GD132.DLL</td>
<td></td>
</tr>
<tr>
<td>77F90000</td>
<td>0007B000</td>
<td>ndt1</td>
<td>5.0.2195.6685</td>
<td>C:\WINNT\system32\ntdll.dll</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78220000</td>
<td>00248000</td>
<td>7822FFE9</td>
<td>SHELL32</td>
<td>5.0.3700.6705</td>
<td>C:\WINNT\system32\SHELL32.dll</td>
<td></td>
</tr>
<tr>
<td>7C200000</td>
<td>00042000</td>
<td>7C201FE4</td>
<td>ADVAPI132</td>
<td>5.0.2195.6710</td>
<td>C:\WINNT\system32\ADVAPI132.DLL</td>
<td></td>
</tr>
<tr>
<td>7C4E0000</td>
<td>00089000</td>
<td>7C4EC5E1</td>
<td>KERNEL32</td>
<td>5.0.2195.6688</td>
<td>C:\WINNT\system32\KERNEL32.DLL</td>
<td></td>
</tr>
</tbody>
</table>

**Listing 2. Disassembled View**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Instruction</th>
<th>Segment</th>
<th>Type</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400000</td>
<td>20</td>
<td>PUSH Afind.0041468C</td>
<td>Afind</td>
<td>CALL</td>
<td>0041468C</td>
<td></td>
</tr>
<tr>
<td>00400020</td>
<td>20</td>
<td>PUSH Afind.004146C0</td>
<td>Afind</td>
<td>CALL</td>
<td>004146C0</td>
<td></td>
</tr>
<tr>
<td>004000A0</td>
<td>20</td>
<td>CALL Afind.00446AC</td>
<td>Afind</td>
<td>CALL</td>
<td>00446AC</td>
<td></td>
</tr>
<tr>
<td>00400100</td>
<td>20</td>
<td>CALL Afind.0041A780</td>
<td>Afind</td>
<td>CALL</td>
<td>0041A780</td>
<td></td>
</tr>
<tr>
<td>0040010E</td>
<td>20</td>
<td>PUSH Afind.0040B753</td>
<td>Afind</td>
<td>CALL</td>
<td>0040B753</td>
<td></td>
</tr>
<tr>
<td>00400200</td>
<td>20</td>
<td>PUSH Afind.00441B30</td>
<td>Afind</td>
<td>CALL</td>
<td>00441B30</td>
<td></td>
</tr>
<tr>
<td>00400260</td>
<td>20</td>
<td>PUSH EDI</td>
<td>Afind</td>
<td>CALL</td>
<td>00440000</td>
<td></td>
</tr>
<tr>
<td>00400280</td>
<td>20</td>
<td>PUSH EDI</td>
<td>Afind</td>
<td>CALL</td>
<td>00440000</td>
<td></td>
</tr>
</tbody>
</table>
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.

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 ;%5 55  PUSH EBX
0040929C ;%8EC  MOV EBX,ESP
0040929E ;%8EC 0C080000  SUB ESP,80C
004092A4 ;%8D4F  FC  LEA EAX,DWORD PTR SS:[EBP-4]
004092A7 ;%8A  50  PUSH EAX  ;/pHandle
004092A9 ;%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 F4BFF000  MOVX BYTE PTR SS:[EBP-40C],0  ;
004092B9 ;%FF75  08  PUSH DWORD PTR SS:[EBP+8]  ;/hKey
004092BC ;%C685 F47FF000  MOVX BYTE PTR SS:[EBP-80C],0  ;
004092C3 ;%FF15 14404100  CALL DWORD PTR DS:[<ADVAPI32.RegOpenKey>:\RegOpenKeyExA
004092C9 ;%85C0  TEST EAX,EAX
004092CB ;%75 31  JNE SHORT AFind._004092FE
004092CD ;%8D4F  F4  LEA EAX,DWORD PTR SS:[EBP-C]
004092C0 ;%50  50  PUSH EAX  ;/pBufSize
004092C1 ;%8D85 F4BFF000  LEA EAX,DWORD PTR SS:[EBP-40C]  ;
004092C7 ;%50  50  PUSH EAX  ;/Buffer
004092C8 ;%8D45  F8  LEA EAX,DWORD PTR SS:[EBP-8]  ;
004092CB ;%50  50  PUSH EAX  ;/pValueType
004092CD ;%6A  00  PUSH 0  ;/Reserved = NULL
004092CE ;%FF75  10  PUSH DWORD PTR SS:[EBP+10]  ;/ValueName
004092E1 ;%C745 F4 00404000  MOV DWORD PTR SS:[EBP-C],408  ;
004092E8 ;%FF75  FC  PUSH DWORD PTR SS:[EBP-4]  ;/hKey
004092EB ;%FF15 2C04100  CALL DWORD PTR DS:[<ADVAPI32.RegQueryValue>:\RegQueryValueExA
004092F1 ;%85C0  TEST EAX,EAX
004092F3 ;%54 1B  JE SHORT AFind._00409310
004092F5 ;%FF75  FC  PUSH DWORD PTR SS:[EBP-4]  ;/hKey
004092FA ;%FF15 00404100  CALL DWORD PTR DS:[<ADVAPI32.RegCloseKey>:\RegCloseKey
004092FE ;%68 36434100  PUSH AFind._00414336  ;/String2 = **
00409303 ;%FF75  14  PUSH DWORD PTR SS:[EBP+14]  ;/String1
00409306 ;%FF15 4F04100  CALL DWORD PTR DS:[<KERNEL32.1strcpyA>:\lstrcpyA
0040930C ;%33C0  XOR EAX,EAX
0040930E ;%C9  LEAVE
0040930F ;%C3  RETN
00409310 ;%0837D  P8  02  CMP DWORD PTR SS:[EBP-8],2
00409314 ;%56  PUSH ESI
00409315 ;%8935 F404100  MOV ESI,DWORD PTR DS:[<KERNEL32.1strcpyA>:KERNEL32.1strcpyA
0040931B ;%57  PUSH EDI
0040931C ;%7E 75 41  JNZ SHORT AFind._0040935F
0040931E ;%8D85 F4BFF000  LEA EAX,DWORD PTR SS:[EBP-40C]  ;
00409324 ;%50  50  PUSH EAX  ;/String2
00409325 ;%8D85 F47FF000  LEA EAX,DWORD PTR SS:[EBP-80C]  ;
0040932B ;%50  50  PUSH EAX  ;/String1
0040932C ;%FFD6  CALL ESI  ;\lstrcpyA
0040932E ;%BF FP030000  MOV EDI,3FF
00409333 ;%57  PUSH EDI  ;/DestSizeMax => 3FF (1023.)
00409334 ;%8D85 F4BFF000  LEA EAX,DWORD PTR SS:[EBP-40C]  ;
0040933A ;%50  50  PUSH EAX  ;/DestString
0040933B ;%8D85 F47FF000  LEA EAX,DWORD PTR SS:[EBP-80C]  ;
00409341 ;%50  50  PUSH EAX  ;/SrcString
00409342 ;%FF15 0404100  CALL DWORD PTR DS:[<KERNEL32.ExpandEnv>:\ExpandEnvironmentStringsA
00409348 ;%3BC7  CMP EAX,EDI
0040934A ;%76 13  JBE SHORT AFind._0040935F
0040934C ;%8D85 F47FF000  LEA EAX,DWORD PTR SS:[EBP-80C]  ;
00409352 ;%68 105C4100  PUSH AFind._00415C10  ;/ASCII
00409354 ;%00  "Registry Error $1023: String can not be expanded"
00409357 ;%50  50  PUSH EAX
00409358 ;%EB 4F03FF00  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.RegCloseKey>:\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=PUSH afind.00416B50 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 functions 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 0040092B 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
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 ExpandEnvironment-Strings 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 GETREG-NUMBER string. During this analysis the required code is presented (see Listing 3).

This code shows the use of registry functions for querying some parts 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 ExpandEnvironment-Strings 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.

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 recompling 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
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
```

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 interfacing 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.

---

**On the 'Net**
- [http://www.openrce.org](http://www.openrce.org)
- [http://www.openrce.org/blog/browse/aditya_ks](http://www.openrce.org/blog/browse/aditya_ks)
- [http://www.rynaev.net/](http://www.rynaev.net/)
- [http://home.arcor.de/idapalace/](http://home.arcor.de/idapalace/) – Index of IDAPalace
- [http://www.exetools.com](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](http://www.secniche.org)