Demystifying Windows PE Caveats

Aditya K Sood

Difficulty

The article is comprised of analytical methods that are required to reverse engineer a Windows PE executable . This intrinsic model follows the top to bottom approach. In disseminating a PE Executable, a specific pattern should be followed and applied. This is crucial to trap bugs and determine faults in the application.

The use of reverse engineering is very useful in application testing. The article will take you from peripheral concepts to the core of applied methods to govern the reverse analysis of windows executable. This sets a realm to reverse engineer an application based on the dynamic code execution. Basically, learning to traverse objects in a program is the prime aim of this article.

Anatomy of Debugging

The computer matrix encompasses a number of objects. These objects are comprised of different types of language oriented applications and programs. No doubt it is very easy to code an application with defined stats but the applications require testing to be done to check the strength of code. The testing directly relates to the code traversing from top to bottom. It has been rightly said that *It is very necessary to get to the base to solve the hidden mysteries*. This statement is the foundation of reverse engineering. It is necessary to look at the assembly instructions to find the vulnerable code. These assembly instructions provide an interface between the application and the hardware. Debugging is the art of tracking an application for insecure or weak code. Testing should not be confused with debugging. The testing is a process of executing a program with an intent to find errors in it. Debugging is comprised of two phases. The first phase consists of finding the exact nature of the error and tracking it in the code. The second phase is protection oriented as errors are removed indirectly from the code. The debugging makes the application robust and strength prone with an element of flexibility. A very simple debugging model: see Figure 2.

What you will learn...

- Step by step approach to analyse the Windows PE Executable and methods relative to it,
- The concepts that form the bases of Effective Debugging,
- Formulation of reversing concepts.

What you should know...

- You should be confronted with General Debugging,
- A prior knowledge of PE internals will be effective to understanmd the concepts.

Figure 1. *Logo*

The debugging approach is three step layout. At first it is very crucial to understand the cause of an error. The second step involves the path of error generation and at last is the effect of an error on the execution state of the application. These three elements are critical in defining the debugging approach. Debugging indirectly is comprised of domain knowledge of the application to identify the problems of logic. The debugger is the tool that makes the debugging possible in the context of application. When developing application software, an understanding of the application is as important as the software to be able to diagnose its defects. To know what is unusual, you must know what is common. What is common in an operational piece of software is correct behavior. Correct behavior is defined by an external reference. The debugging holds the internal as well as the external objects. Debugging reproduces a problem in a machine specific language for purpose of testing and securing code. It is an art that makes the application secure. Now I will talk about the techniques used in debugging in detail.

Address Traversing

Address traversing is a technique through which a reverse engineer traces the code with addresses adhering to the machine instructions. Every single instruction carries an address space. This address space grouped together forms a stack. The stacks comprised of the machine oriented layout for the application. The traversing means one should check the cross references in the code based on the address of the instruction. This clears the flow of execution and the procedure of updating and modification of registers. The stack addresses can be referenced directly or indirectly. When you debug an application you must have encountered the strings embedded in the code. These strings leverage a lot of information when the code is heavy. Their dissemination helps us to understand the structure of the code to some extent. Let's look at it with ollydbg. See Figure 3.

This snapshot provides the referenced strings in the kernel32 module. If you look closely, the strings are comprised of instructions and addresses where these are referenced. The instructions tell us the execution flow and the use of these strings. The addresses enable us to jump directly to the part of the code where it is necessary. Let's assume that someone is cracking an application against login parameters. The cracker can easily search the username or password string and jump to that part of code. The rest of the code is useless to him. This makes the debugging process typical. It's a play with strings and addresses to unveil the hidden working functionality of an application. This technique makes the user skillful.

Knocking Import Address Table

This technique plays a crucial role in debugging. The import table holds the information needed to link the API calls. In this I will discuss the role specific objects. Let's understand the Import Address Table [IAT]

When an executable is first loaded, the Windows loader is responsible for reading in the files PE structure and loading the executable image into memory. One of the other steps it takes is to load all of the dlls that the application uses and map them into the process address space. The executable also lists all the functions it requires from each dll. Because the function addresses are not static, a mechanism has to be developed that allows for changing these variables without altering all of the compiled code at runtime. This is accomplished through the use of an import address table. This is a table of function pointer that is filled in by the windows loader as the dlls are loaded.

The objects that are used inherit a definite system structure. The import table works on an understated structure.

Looking at the structure of the Import descriptor we deduce the three critical elements.

- 01] Original First Thunk.
- 02] Name.
- 03] First thunk.

Original First Thunk

Before we get little deeper, let's understand the meaning of THUNK. Thunks are the definitive pieces of codes in an operating system that handle the transitions between 16 and 32-bit code. Thus they ensure backward compatibility between the calls made by the application.

```
   16 Bit ----------------> call Occurs
--------------------> 32 bit
[Operation Done]
16 Bit <--------------- call back <----
-----------------32 bit
```

The OS is subjected with this facility for backward compatibility, so that the 16 bit applications can run smoothly in 32 bit environment.

Original First Thunk: It is the dword value. It gives the information to loader about the call to API from which the starting address is to be taken. It is mainly defined as:

```
RVA --------> Relative Virtual
                    Address.
RVA [ XXXXXXXX ] -----------> First
                    API Call
XXXXXXXX ---> It is called as First
                    RVA. Example:-
                       42004568 ----
                    --> ShellExecute
```

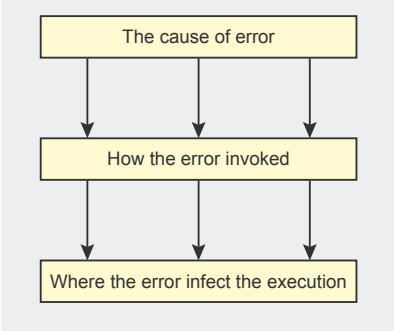

Figure 2. *Debugging model*

So at first RVA an API call is made to ShellExecute function. That's how the information is extracted.

Name

It holds the RVA of the dll to be loaded into memory. It is a dword value. It is comprised of :

```
RVA = Image Base + Endian Order
```

Note: As we know OllyDbg gives the result in the Big Endian order where as the x86 architecture holds the address in the Little Endian order, so the address gets exchanged reversibly in the di tuples.

First Thunk

Once the API is linked and gets loaded into memory the first thunk which is a RVA points to the IAT table.

```
First Thunk [RVA] ------------------
                  ----> IAT table
                     [MMMMMMMM] -
                  ----------> Call
                  First
```

It actually comprises of the IAT entry addresses which further relates to the IAT table for call generation. So

```
OriginalThunk +Name+ FirstThunk ----
                  ---> IAT TABLE.
```

Up to this we have understood the IAT.

Call and Jump Caveat

These are two assembly instructions controlling the execution flow of code and are used instantaneously. This comes in handy when checking a condition. If the returned value meets the requirement then normal execution occurs otherwise the vector is shifted to another part of the code through JMP instruction. The call instruction is used to add some module during the execution state. Let's see: Figure 4

You can easily notice the call and jmp instruction in the snapshot above. The instruction addresses are undertaken from the IAT table. This table maps the addresses to the various API calls. This is necessary to link the application. The windows provide the pre-requisites to link the API calls to the calls that are made in the execution of the program. Windows is always ready to disclose the information to the linker so that linking is executed without any complexity. This ensures the presence of the module into the memory space whenever the linking is done.

We know that once the linking is done, the API calls are ready to get loaded in the memory for the PE file to execute properly. Then the memory is mapped for these calls i.e. the specific modules needed for the execution gets loaded into the address space of that PE file. If the size of the modules crosses a defined limit then virtual space is used to execute the calls properly. The Loader then fills the IAT table i.e. the import table consisting of the addresses, mapped to the specific system dynamic link libraries. The crash happens when the IAT table is not generated properly and the API's are not efficiently loaded into the context of the memory space of that application. The result can be seen in the memory dump files.

Listing 1. *IMAGE_IMPORT_DESCRIPTOR*

```
IMAGE_IMPORT_DESCRIPTOR struct
                        OriginalFirstThunk    dd 0 ; RVA to original unbound
                        IAT (table of names)
                        TimeDateStamp     dd 0 ; not used here
                        ForwarderChain     dd 0 ; not used here
                        Name        dd 0 ; RVA to DLL name string
                        FirstThunk        dd 0 ; RVA to IAT array (table of
                        doors)
IMAGE_IMPORT_DESCRIPTOR ends
```

The Intermodular calls

These are the standard calls used in the inclusion of number of system modules in the memory. This technique is very generic because a single application requires multiple modules to be executed. It is very critical because it checks the cross reference calls made to the system libraries or manually defined modules. See Figure 5.

This layout is composed of intermodular calls made in the kernel32.dll. A number of system calls have been made to the API's defined in the system. A particular address is placed in front of a call and is used directly in the application. The call to system module is made through these addresses.

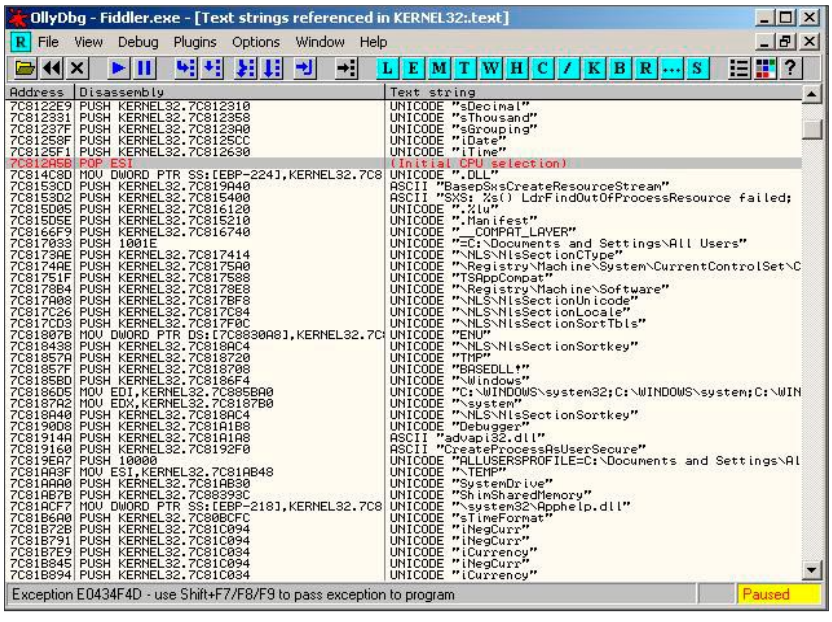

Figure 3. *Address Traversing*

Headers Specificity

Header objects are the informational elements. They provide a lot of information needed to understand the flow of execution. It also throws light on the raw data used in the application. The header possesses information about the import address table and virtual space. It is comprised of four basic information headers:

- 01 .text.
- 02 .rdata.
- 03 .data.
- 04 .rsrc.

In accordance with the law:

- .text and .rdata sections are pointed by the headers and cannot be deleted.
- .data and .rsrc sections are the directories which can be deleted or ignored.

The .text and .rdata and all other sections hold:

- Name – The section name from the header.
- Virtual Size – Virtual size of the section.
- Virtual Address – Virtual load address.
- Size of Raw Data – Physical size of the section.
- Pointer To Raw Data – The section's offset in the file.
- Characteristics – The flags describe the type of section and how to treat section's memory.
- Pointing Directories – Data Directories may point to a section.

The Virtual Address in .rdata section is the base address for the import address table. It acts as a base address upon which further addressing is set. The pointing directory in the .rdata section is always the IAT table of the application you are going to disassemble.

Dependency Walking

This relates to the dependency status of various APIs that are linked together. These modules get loaded in the memory space of the executing PE file. It is necessary to understand the dependency status.

It gives us information about the calls usage in the code and helps us in tracking changes. So it's always preferable to scan the dependency.

Here I am providing you with some dependency status. This gives information regarding the modules required for executing application. Let's look at it Figure 6.

This snapshot shows that a number of dynamic link libraries are required to execute an application. The dependency walking enhances the debugging process to track changes and find flaws in the code. Remember cross reference analysis is crucial in the debugging process for executing application.

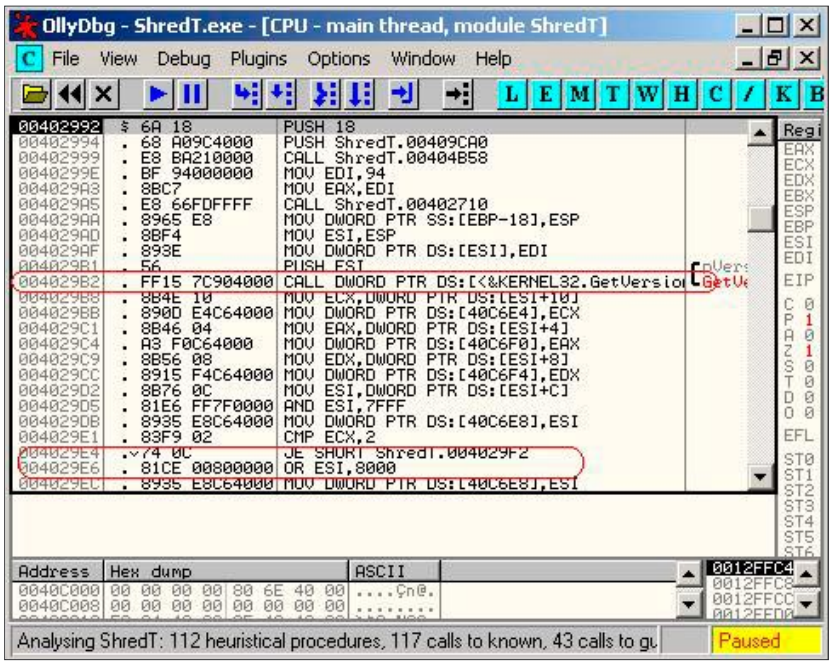

Figure 4. *Call Instruction Layout*

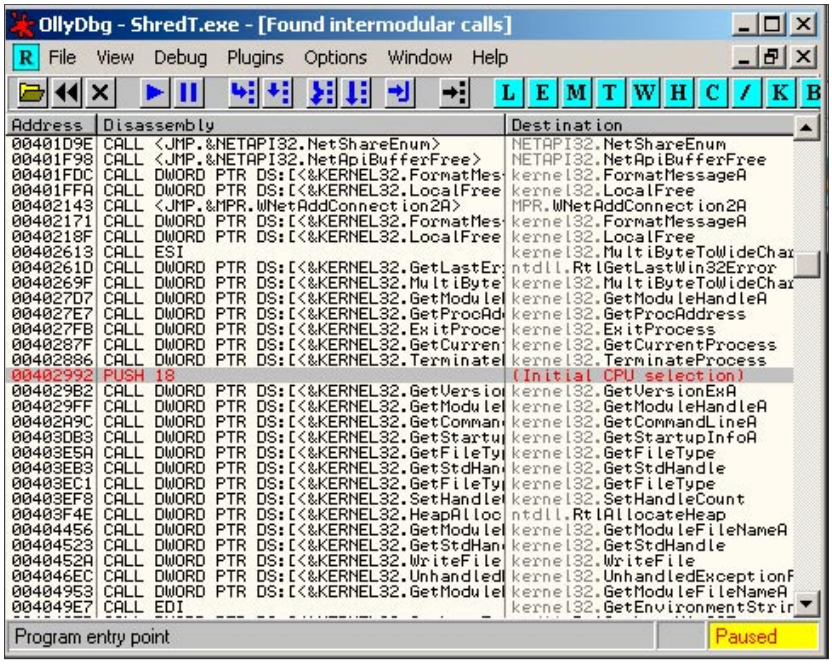

Figure 5. *Intermodular Calls*

Playing with the Entry Points

The entry point is the definite point from where the starting address of application is encountered. It means that entry point is itself an address.

```
Entry Point ---------> DDDDDDDD [
                       Address ]
Let's say the range is from DDDDDDDD
                       - FFFFFFFF
```

As you can see if the address is found between the desired range the PE tool entry point can be easily altered. The application crashes very badly when the range limit is crossed. This happens because the whole base address gets altered and the IAT tables are crossed with other addresses. This too is stored in the header information. The question is why the entry point is altered. This is done for testing purposes and cross table analysis. See Figure 7.

One can see the entry points defined with base addresses.

Checksum Realm

The file checksum is computed at opening. It is used in Windows NT for validation at load time. All drivers, any DLL loaded at boot time, and any DLL that ends up in the server are checked. The checksum is supposed to prevent loading of damaged binaries that would crash anyway (a crashing driver would result in a BSOD, so it is better not to load it at all). That is, a checksum is intended to detect simple memory failures leading to corruption (whether or not a block of memory on disk has gone bad and the values stored there have become corrupted). Some Microsoft System DLLs also use the linker checksum to count how many instances of a particular file are loaded. When the limit is reached, Under any circumstances, Windows NT will not load such marked files regardless of their admin status etc. Usually no error is reported if nothing happens after execution of a program dependent on one of these libraries. Example is common control library with a limit of 32 instances.

Here you can compare the real checksum to the value reported by the header. If necessary, it is possible to update the value of the checksum in the header. Usually compilers do not fill this field, with the exception of NT-drivers.

Art of Exporting/ Importing API's

In this section I am going to discuss about:

Exporting APIs

As the name suggests this means the calling of modules being used by the other applications. The actual name of a procedure *as it is to be called* is found in the export table of any executable, EXE or DLL. The name is present in the program's import table. When the loader runs a program, it loads the associated DLLs into the process address space. It then extracts information about the import functions from the main program. It uses the information to search the DLLs for the addresses of the functions to be patched into the main program. The place in the DLLs where the PE loader looks for the addresses of the functions is the export table.

Let's have a look at the Export Table Arguments. It consists of:

- Entry Point – Entry point of the exported function.

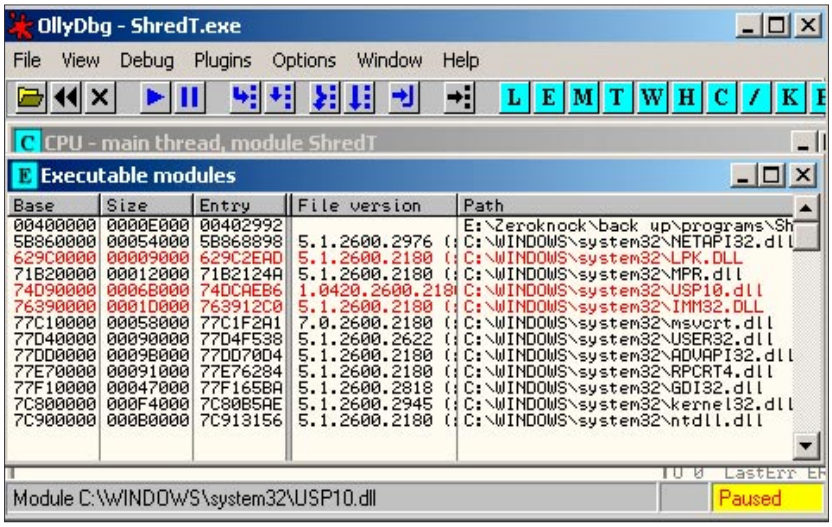

Figure 6. *DLL Mapping To System Calls*

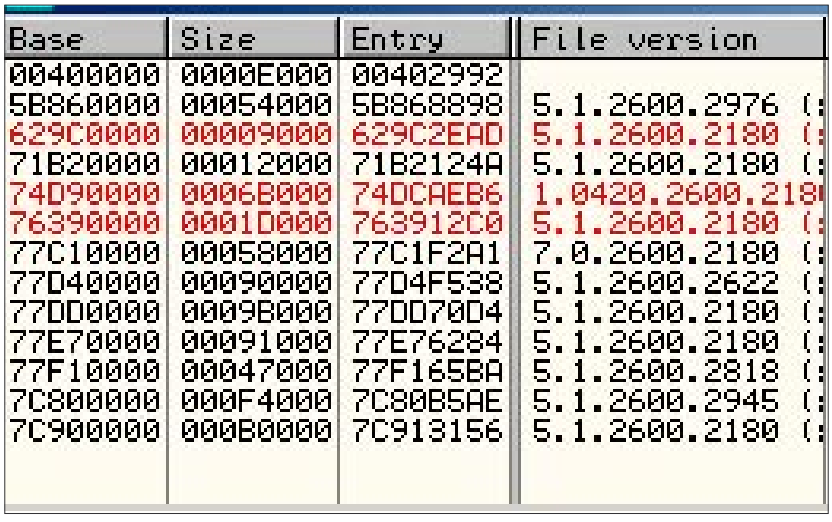

Figure 7. *Entry Checks*

- Ord – An ordinal, a number that uniquely identifies a function in a particular DLL.
- Name – Function name.

Export Properties

Time Date Stamp : gives the time the table was created (some linkers set it to 0).

Ver: Version info ('Major Version' and 'Minor Version'), and these, too, are often enough set to 0.

DLL Name: The internal DLL name. The name is necessary in case the DLL file is renamed.

Exported Functions: The total number of exported functions.

Exported Names: The number of functions that are exported by name. This value can be 0. In that case, the module may export by ordinal only.

Pointers to Entry Point: points to the head of the array of entry points Address Of Functions (given as 32-bit-RVAs).

Pointers to Name: An RVA that points to an array of RVAs of the names of functions in the module.

Pointers to Ordinal: An RVA that points to a 16-bit array that contains the ordinals associated with the function names in the 'AddressOfNames' array.

Importing APIs

Importing as we know is a technique to import all the functions or modules that are needed to be debugged by the application. This is done by the loader as discussed earlier.

About the author

Aditya K Sood, Founder of the Metaeye Security Group [MSG], an independent security research arena. He holds a MS In Cyber Law And Information Security from IIIT-A. He is known in the security world with handle of Zeroknock.By profession the author is a penetration tester and specialized in web exploitation and protocol analysis. The author is working in the security field since last six years.
Handle: Zeroknock
http://zeroknock.metaeye.org/mlabs

The basic specifications are

RVA – RVA of an array of 32-bit numbers for PE32, 64-bit for PE32+. The collection of these entries describe all imports from the image to a given DLL.

Hint – Index into the Export Table of the DLL the function resides in.

Name – Function names.

Import Properties

Name Table: RVA of the string that must be matched to the public name in the DLL.

Time Date Stamp: Set to zero until bound; then this field is set to the *TimeDateStamp* of the exporting DLL's '*FileHeader*'.

Forwarder Chain: The 32-bit index of the first forwarder in the list of imported functions.

RVA: Address of ASCII string containing the DLL name. This address is relative to the Image Base. *Address Table: Relative virtual address of the Import Address Table.*

Exporting/Importing By Name

This is a technique in which importing or exporting of a specific function is called by name. As I specified above, the calling that is made in `kernel32` is done by name. So the import is done by NAME. It means that the function must be specified in the Import Table.

Exporting/Importing By Ordinal

Before getting into this technique we must understand what an Ordinal is. An ordinal is a 16-bit number that uniquely identifies a function in a particular DLL. This number is unique only within the DLL it refers to. So when we extract this ordinal value we pass it to the GetProcAddress function and we get the address of that function.

Delay Importing

The Delay Import tables are added to the image in order to support a uniform mechanism for applications. This is done to delay the loading of a DLL until the first call is undertaken into that DLL.

The table specifications are as follows

RVA – RVA of an array of 32-bit numbers for PE32, 64-bit for PE32+. The collection of these entries describes all delay imports from the image to a given DLL.

Hint – Index into the Export Table of the DLL the function resides in.

Name – Function names.

Bound – Relative virtual address of the Bound Delay-Load Address Table, if it exists.

Unload – Relative virtual address of the unload delay-load address table, if it exists.

Delay Import Properties

Characteristics RVA of the delay-load name table, which contains the names of the imports that may need to be loaded.

Module handle: RVA of the module handle (in the data section of the image) of the DLL to be delay-loaded. Used for storage by the routine supplied to manage delay loading.

Time Date Stamp Time stamp of DLL to which this image has been bound.

Bound Address Pointer: The 32-bit index of the first forwarder in the list of imported functions

Thats all about the export/import functionality.

Conclusion

The hierarchy plays a crucial role in debugging because the step by step analysis proved beneficial in undertaking the functionality of modules. If you're going to get serious about improving your debugging productivity, you need to keep records.

I have already described several ways to design skillful debugging. If you want to measure whether your productivity is improving, than you must follow a defined pattern of debugging.

It is very crucial to debug an application with defined benchmarks. This not only enhances the productivity but rather increase the level of understanding. ●