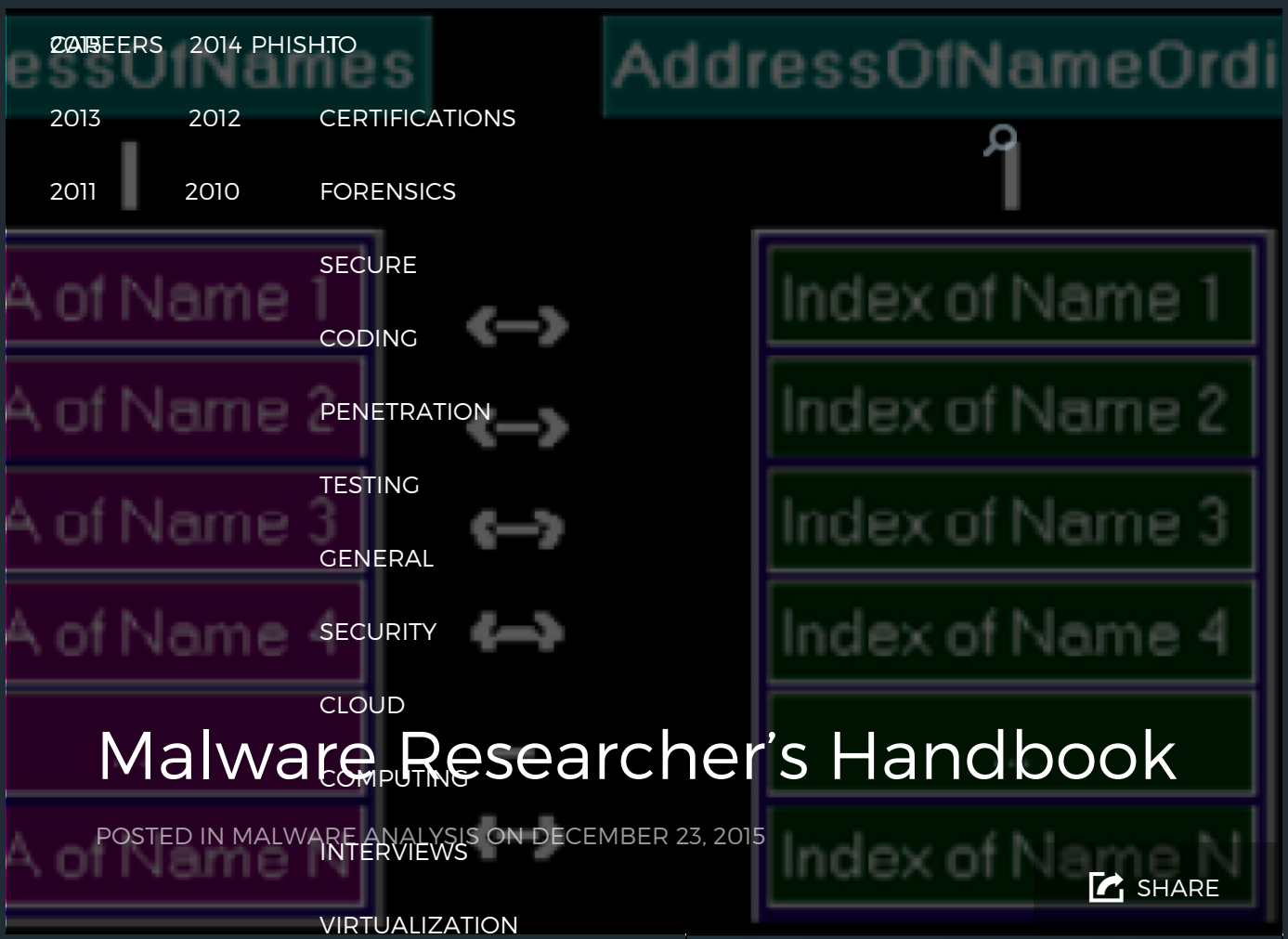
CONTENT ARCHIVES
2015
2013
2011

HACKING CONTENT ARCHIVES
CAREERS 2014 PHISHING
2012
2010

JOB BOARD

CERTIFICATIONS
FORENSICS
SECURE CODING
PENETRATION TESTING
GENERAL SECURITY
CLOUD COMPUTING
INTERVIEWS
VIRTUALIZATION
SECURITY WIRELESS SECURITY
SCADA
REVERSE ENGINEERING
DATA RECOVERY
EXPLOIT DEVELOPMENT



Malware Researcher's Handbook

POSTED IN MALWARE ANALYSIS ON DECEMBER 23, 2015

SHARE

Demystifying PE File (Part 2: Continue)

As per our previous article we will continue on this article here by the rest of section of PE file. Those who don't know the previous section please have a look on here(http://resources.infosecinstitute.com/2-malware-researchers-handbook-demystifying-pe-file/).

Because this will be more advance section as well some form of automation stuff, where I will implement windows API as well as Python programming.

As we said in previous article we will discuss how TLS callback function is very helpful for attacker and also we will show demonstration by an application. But before that we should finish the rest of part of PE section

The Export Section:

Exports are functions and values in one module that have been declared to be shared with other modules. This is done through the use of the "Export Directory", which is used to translate between the name of an export (or "Ordinal", will discuss more in later point), and a location in memory where the code or data can be found. The start of the export directory is identified by the IMAGE_DIRECTORY_ENTRY_EXPORT entry of the resource directory like below:

```
struct IMAGE_EXPORT_DIRECTORY {

    long Characteristics;

    long TimeDateStamp;

    short MajorVersion;

    short MinorVersion;

    long Name;
```

```
    long Base;

    long NumberOfFunctions;

    long NumberOfNames;

    long *AddressOfFunctions;

    long *AddressOfNames;

    long *AddressOfNameOrdinals;

}
```

This section is particularly reference to DLL file and its structure.

In Microsoft Windows A DLL are the modules that contains functions and data.A DLL is loaded at runtime by its calling module that may be exe or a DLL.When a DLL is loaded,it is mapped into address peocess of calling function.

A DLL can have two sections**: Exported and Internal.**

The Exported functions can be called by other modules. Internal functions can be called within the module/DLL where they defined.

The actual exports themselves are described through AddressOfFunctions, which is an RVA to an array of RVA's, each pointing to a different function or value to be exported. The size of this array is in the value NumberOfFunctions. Each of these functions has an ordinal. The "Base" value is used as the ordinal of the first export, and the next RVA in the array is Base+1, and so forth.

Each entry in the AddressOfFunctions array is identified by a name, found through the RVA AddressOfNames. The data where AddressOfNames points to is an array of RVA's, of the size NumberOfNames. Each RVA points to a zero terminated ASCII string, each being the name of an export. There is also a second array, pointed to by the RVA in AddressOfNameOrdinals. This is also of size NumberOfNames, but each value is a 16 bit word, each value being an ordinal. These two arrays are parallel and are used to get an export value from AddressOfFunctions. To find an export by name, search the AddressOfNames array for the correct string and then take the corresponding ordinal from the AddressOfNameOrdinals array. This ordinal is then used to get an index to a value in AddressOfFunctions.

If we analyze the members of 11 sections of Image_Import_Directory,we will only

discuss the important sections as below:

nName – The internal name of the module. This field is necessary because the name of the file can be changed by the user. If that happens, the PE loader will use this internal name.

nBase – Starting ordinal number (needed to get the indexes into the address-of-function array – see below).

NumberOfFunctions – Total number of functions (also referred to as symbols) that are exported by this module.

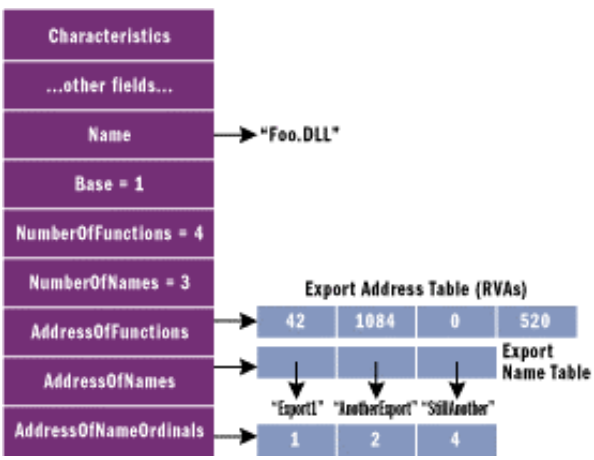
NumberOfNames – Number of symbols that are exported by name. This value is **not** the number of **all** functions/symbols in the module. For that number, you need to check NumberOfFunctions. It can be 0. In that case, the module may export by ordinal only. If there is no function/symbol to be exported in the first case, the RVA of the export table in the data directory will be 0.

AddressOfFunctions – An RVA that points to an array of pointers to (RVAs of) the functions in the module – the Export Address Table (EAT). To put it another way, the RVAs to all functions in the module are kept in an array and this field points to the head of that array.

AddressOfNames – An RVA that points to an array of RVAs of the names of functions in the module – the Export Name Table (ENT).

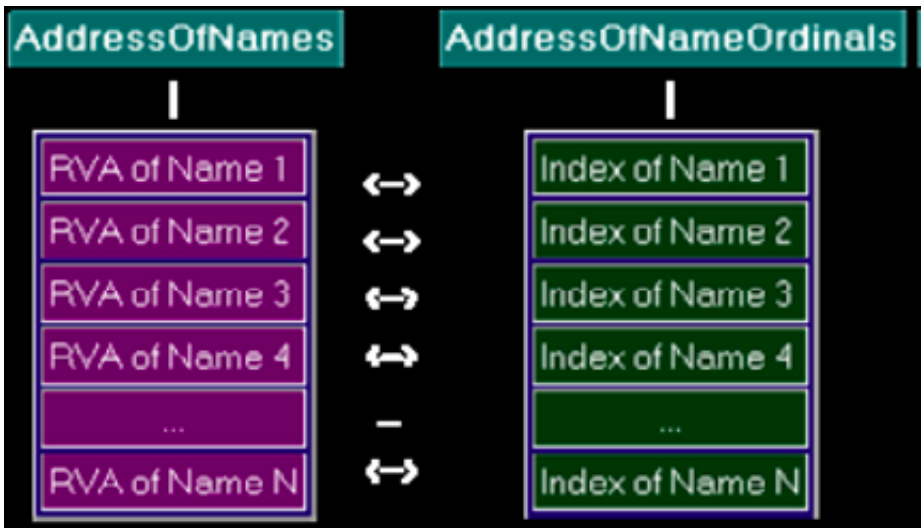
AddressOfNameOrdinals – An RVA that points to a 16-bit array that contains the ordinals of the named functions – the Export Ordinal Table (EOT).

We will discuss more in graphical way below:



So **Image_Export_Directory** points to three arrays and a table of ASCII strings.The important is Export Address Table,which is an array of function

pointer,that contains the address of exported function. The other 2 arrays (EAT & EOT) run parallel in ascending order based on the name of the function so that a binary search for a function's name can be performed and will result in its ordinal being found in the other array. The ordinal is simply an index into the EAT for that function.



So we can say if function is exported by name we need to walk both AddressOfNames and AddressOfNameOrdinals arrays to obtain the index into the AddressOfFunctions array.

If we already have the ordinal of a function, we can find its address by going directly to the EAT. Although obtaining the address of a function from an ordinal is much easier and faster than using the name of the function, the disadvantage is the difficulty in the maintaining the module. We can see that when we're using the ordinals, obtaining the address of the function is much faster because we only have to calculate one subtract operation

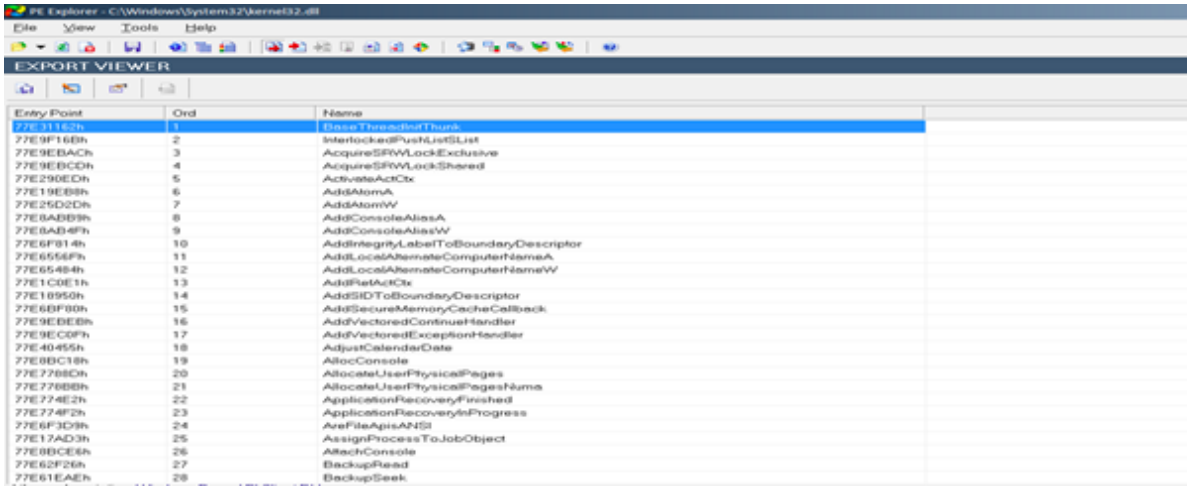
Export by Ordinal Only: Number of functions must be equal to the number of names. Sometimes number of name is less than number of functions. So the function that don't have a name are exported by ordinal only.

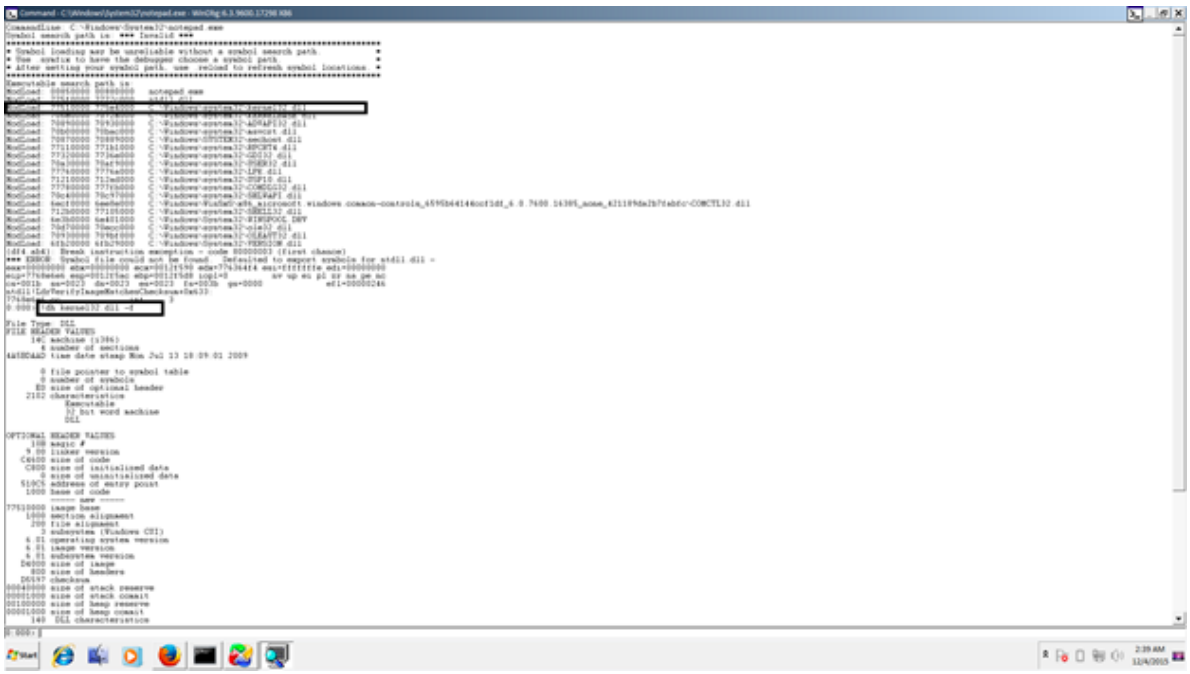
Export Forwarding: Sometimes functions that appear to be exported by a particular DLL that actually reside in a completely different DLL. This is so called Export Forwarding.

For example, in WinNT, Win2k and XP, the kernel32.dll function HeapAlloc is forwarded to the RtlAllocHeap function exported by ntdll.dll. NTDLL.DLL also contains the native API set which is the direct interface with the windows kernel.

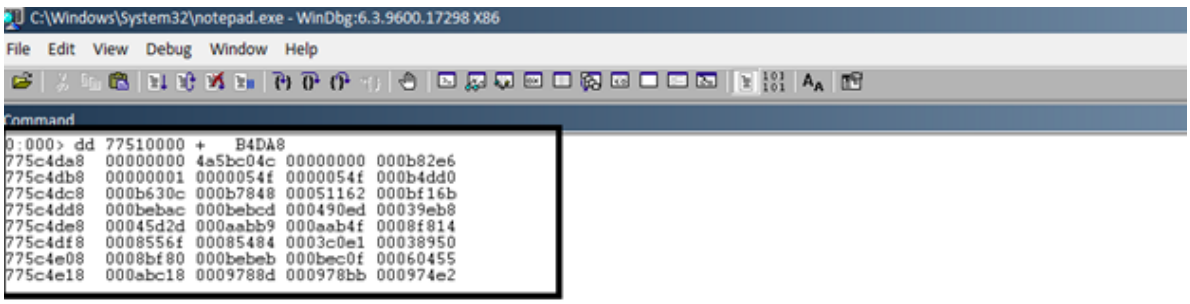
Example/Demonstration:

Let's print the export header for Kernel32.dll and its subsection. First I will use PE Explorer and after that I will dump all EAT and IAT from notepad.exe by Windbg. After that I will hook on kernel32.dll





So the offset (A915) is Export directory of imagebase(77510000).so the actual address is 77510000+A915.we will see this address by **dd 77510000+B4DA8**



Focus on the first 3 rows. The first column has the memory address. We need to focus on the values in the next 4 columns for the first 3 rows. Parsing these values and comparing them with the IMAGE_EXPORT_DIRECTORY structure

definition we get:

```
Characteristics = 00000000

TimeDateStamp = 4a5bc04c

MajorVersion = 0000

MinorVersion = 000b82e6

lpName = 00000001

Base = 0000054f

NumberOfFunctions = 0000054f

NumberOfNames = 000b4dd0

lpAddressOfFunctions = 000b630c

lpAddressOfNames = 000b7848

lpAddressOfNameOrdinals = 00051162
```

Getting RVA: Now we will get to know the RVA of a different function address.

The RVA of the pointer to AddressOfNames array is: 000b7848

to dump the contents of this array, let's add it to the base address and display:

dd imagebase +AddressOfName

```
0:000> dd 76810000 + 000b7848
768c7848  00030002 00050004 00070006 00090008
768c7858  000b000a 000d000c 000f000e 00110010
768c7868  00130012 00150014 00170016 00190018
768c7878  001b001a 001d001c 001f001e 00210020
768c7888  00230022 00250024 00270026 00290028
768c7898  002b002a 002c0000 002e002d 0030002f
768c78a8  00320031 00340033 00360035 00380037
768c78b8  003a0039 003c003b 003e003d 0040003f
```

So, we got the list of RVAs now. Each of these RVAs when added to the base address of gdi32.dll will point to the Function Name string. Let's check by taking the first RVA from this list: 00030002

```
0:000> dd kernel32 + 00030002
76840002  74636572 4579726f 90004178 48539090
76840012  61657243 65446574 6c756166 74784574
76840022  74636172 6e6f6349 48539000 61657243
76840032  65446574 6c756166 6e6f4374 74786574
76840042  756e654d 48539000 61657243 61446574
76840052  624f6174 7463656a 48539000 6e616843
76840062  6f4e6567 79666974 70737553 52646e65
76840072  6d757365 48530065 6e616843 6f4e6567
0:000> dd kernel32.dll + 00030002
76840002  74636572 4579726f 90004178 48539090
76840012  61657243 65446574 6c756166 74784574
76840022  74636172 6e6f6349 48539000 61657243
76840032  65446574 6c756166 6e6f4374 74786574
76840042  756e654d 48539000 61657243 61446574
76840052  624f6174 7463656a 48539000 6e616843
76840062  6f4e6567 79666974 70737553 52646e65
76840072  6d757365 48530065 6e616843 6f4e6567
```

The IMPORT Section

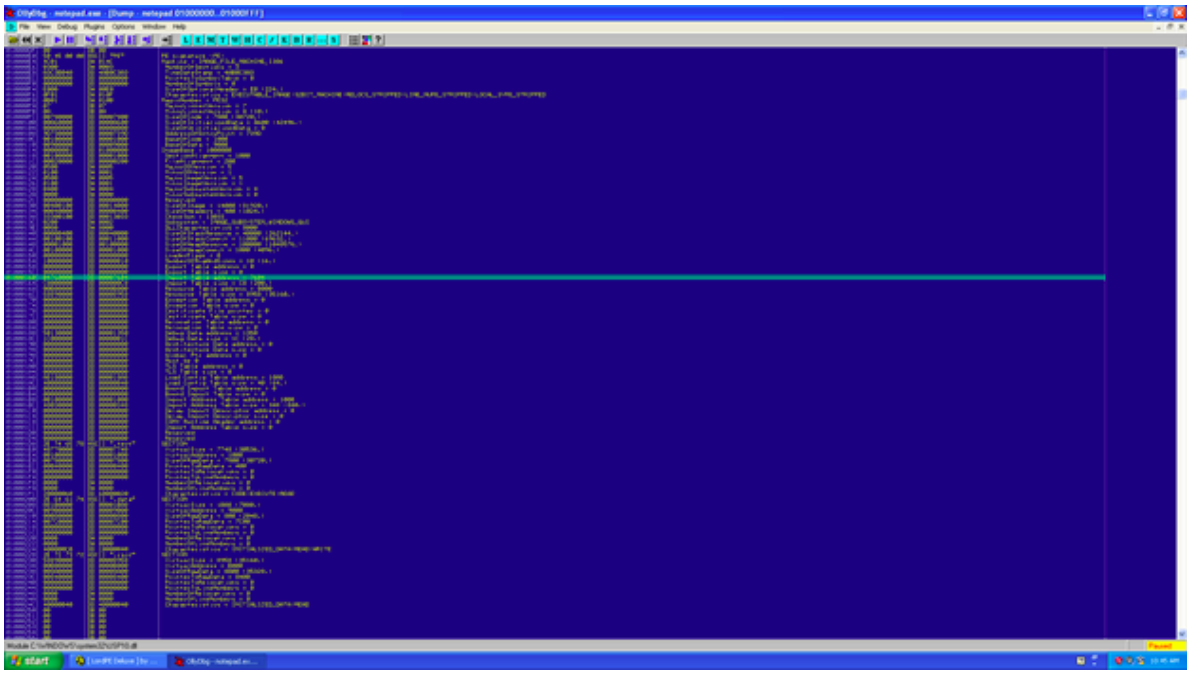
The Import section contains information about all the functions imported by executable from DLLs. This information is stored in several data structures. The most important data structure is import directory and Import Address Table. In some executable, there are also bound_import and delay_Import directories, in which the important one is bound_import.

The Windows loader is responsible for loading all of the DLLs that application uses and mapping them into process address space.it has to find all imported functions in the various DLL and make them available for the executable being loaded.

IMPORT DIRECTORY:

The IMPORT Directory structure is 80th offset of PE header.check the below ollydbg screen.

010000E0+80 = 01000160



The Import Directory is actually an array of **IMAGE_IMPORT_DESCRIPTOR** structures.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {

union {

DWORD Characteristics; // 0 for terminating null import
descriptor

DWORD OriginalFirstThunk; // RVA to original unbound IAT
(PIMAGE_THUNK_DATA)

} DUMMYUNIONNAME;

DWORD TimeDateStamp; // 0 if not bound,

// -1 if bound, and real date\time stamp

// in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)

// O.W. date/time stamp of DLL bound to (Old BIND)

DWORD ForwarderChain; // -1 if no forwarders

DWORD Name;

DWORD FirstThunk; // RVA to IAT (if bound this IAT has actual
addresses)

}
```

Each structure is 20 bytes and contains information about a DLL which our PE file imports functions from. Let's fire up windbg and extract the detailed symbol

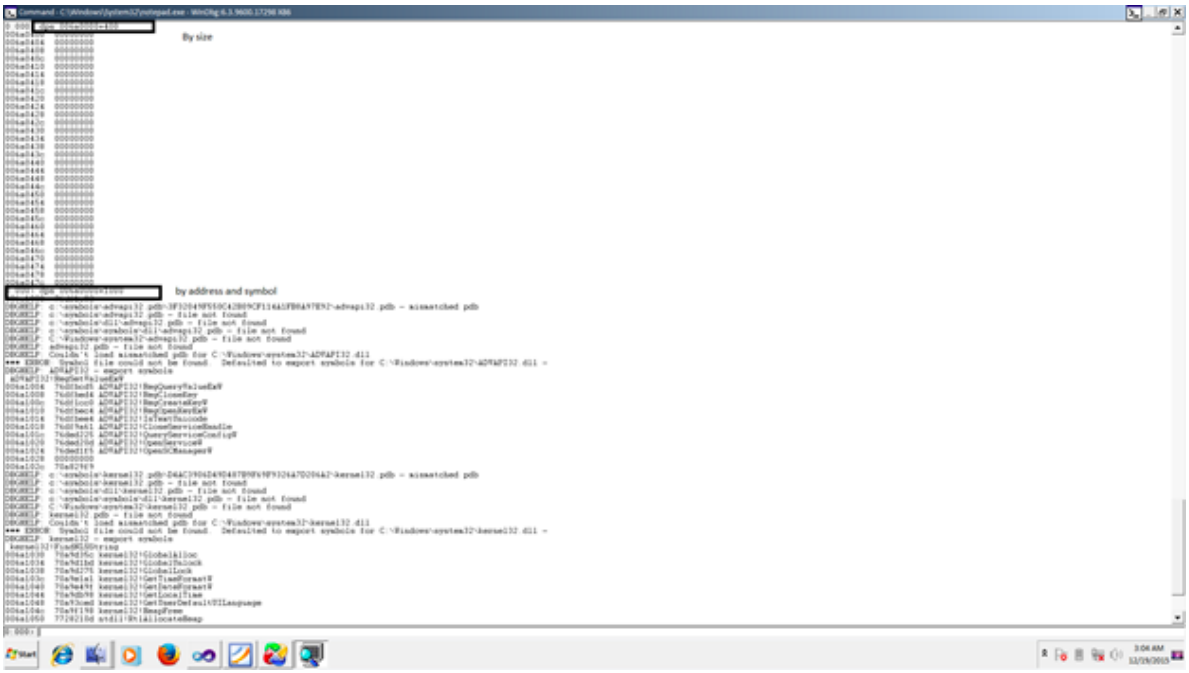
```
:000> !dh 006a0000 -f

ile Type: EXECUTABLE IMAGE
ILE HEADER VALUES
    14C machine (i386)
      4 number of sections
A5BC60F time date stamp Mon Jul 13 16:41:03 2009

      0 file pointer to symbol table
      0 number of symbols
     E0 size of optional header
    102 characteristics
          Executable
          32 bit word machine

PTIONAL HEADER VALUES
    10B magic #
   9.00 linker version
   A800 size of code
  22400 size of initialized data
      0 size of uninitialized data
   3689 address of entry point
   1000 base of code
        ------ new -------
06a0000 image base
   1000 section alignment
    200 file alignment
      2 subsystem (Windows GUI)
   6.01 operating system version
   6.01 image version
   6.01 subsystem version
  30000 size of image
    400 size of headers
  39741 checksum
00040000 size of stack reserve
00011000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
   8140  DLL characteristics
          Dynamic base
          NX compatible
          Terminal server aware
      0 [       0] address [size] of Export Directory
   A048 [      12C] address [size] of Import Directory
   F000 [    1F160] address [size] of Resource Directory
      0 [       0] address [size] of Exception Directory
      0 [       0] address [size] of Security Directory
  2F000 [     E34] address [size] of Base Relocation Directory
   B62C [      38] address [size] of Debug Directory
      0 [       0] address [size] of Description Directory
      0 [       0] address [size] of Special Directory
      0 [       0] address [size] of Thread Storage Directory
   6D58 [      40] address [size] of Load Configuration Directory
    278 [     120] address [size] of Bound Import Directory
   1000 [     400] address [size] of Import Address Table Directory
      0 [       0] address [size] of Delay Import Directory
      0 [       0] address [size] of COR20 Header Directory
      0 [       0] address [size] of Reserved Directory
```

Here 1000 is the RVA to the Image base(06a0000).so Image Base Address +RVA will point to Import Address Table. We will Use the "dps" command to dump the address at that offset and try to resolve them to symbols. You will likely need to run dps a number of times to cycle through the entire import table.we already knew that size is 400,so we will force windbg using this address to show all relevant addresses. We can also see the address.



It is good to know how to fetch an IAT of a PE image since we can use this output to detect any sort of IAT Hooks. We will look into IAT hooks later which is a technique used by rootkits to take control of the functions in a DLL by overwriting the function pointers in the IAT.

Now let's discuss different structure of IAT. The first member **OriginalFirstThunk**, which is a DWORD union, may at one time have been a set of flags. However, Microsoft changed its meaning and never bothered to update WINNT.H. This field really contains the RVA of an array of **IMAGE_THUNK_DATA** structures.

The **TimeDateStamp** member is set to zero unless the executable is bound when it contains -1 (see below). The **ForwarderChain** member was used for old-style binding and will not be considered here. The last member **FirstThunk**, also contains the RVA of an array of DWORD-sized IMAGE_THUNK_DATA structures – **a duplicate of the first array.**

```
typedef struct _IMAGE_THUNK_DATA32 {

union {

DWORD ForwarderString; // PBYTE

DWORD Function; // PDWORD

DWORD Ordinal;

DWORD AddressOfData; // PIMAGE_IMPORT_BY_NAME

} u1;
```

Each **IMAGE_THUNK_DATA** is a DWORD union that effectively only has one of 2 values. In the file on disk it either contains the ordinal of the imported function or an RVA to an **IMAGE_IMPORT_BY_NAME** structure. Once loaded the ones pointed at by FirstThunk are overwritten with the addresses of imported functions – this becomes the Import Address Table.

Each IMAGE_IMPORT_BY_NAME structure is defined as follows:

```
typedef struct _IMAGE_IMPORT_BY_NAME {

WORD Hint;

BYTE Name[1];

}
```

Hint — contains the index into the Export Address Table of the DLL the function resides in. This field is for use by the PE loader so it can look up the function in the DLL's Export Address Table quickly.

Name1 — contains the name of the imported function. The name is a null-terminated ASCII string.

Functions Exported by Ordinal Only

As we discussed in the export section, some functions are exported by ordinal only. In this case, there will be no IMAGE_IMPORT_BY_NAME structure for that function in the caller's module. Instead, the IMAGE_THUNK_DATA for that function contains the ordinal of the function.

Bound Imports

When the loader loads a PE file into memory, it examines the import table and loads the required DLLs into the process address space. Then it walks the array pointed at by FirstThunk and replaces the IMAGE_THUNK_DATAs with the real addresses of the import functions. This step takes time. If somehow the programmer can predict the addresses of the functions correctly, the PE loader doesn't have to fix the IMAGE_THUNK_DATAs each time the PE file is run as the correct address is already there. Binding is the product of that idea.

The Bound_Import Directory

The information the loader uses to determine if bound addresses are valid is kept in a IMAGE_BOUND_IMPORT_DESCRIPTOR structure.

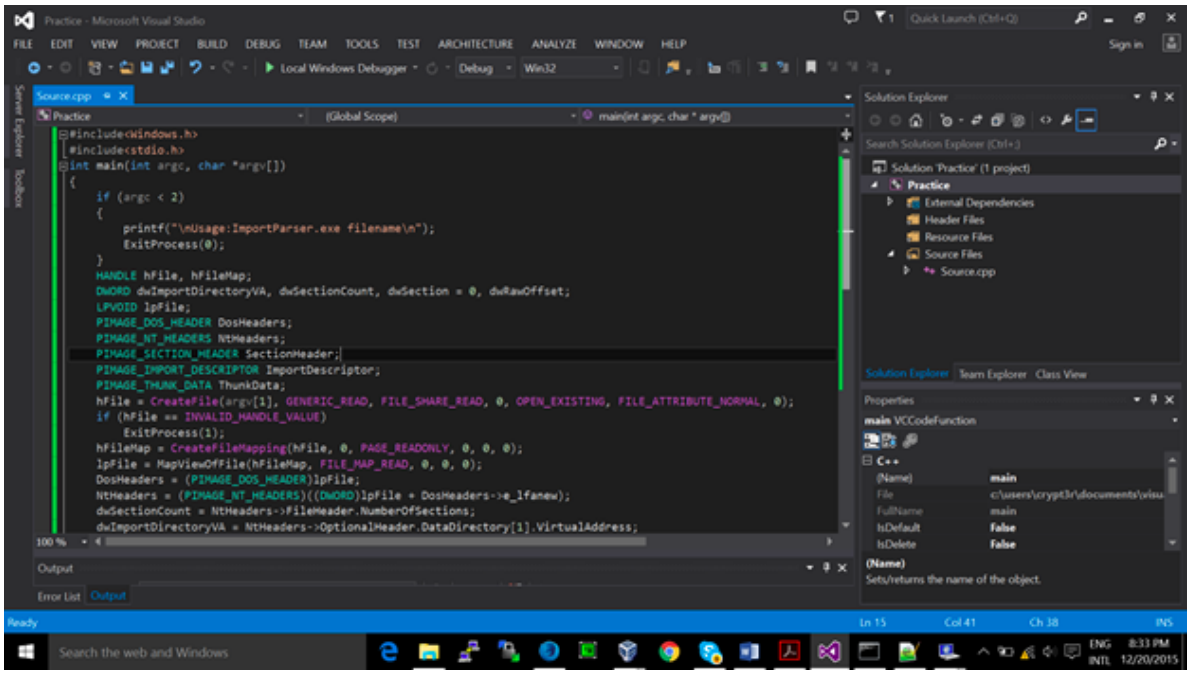
The Loader

When an executable is run, the windows loader first create vir and tual address space for the process and maps the executable from disk into process's address space.it tries to load the image at the preferred base address but relocates it if that address is already occupied.The loader goes through section table and maps each section at the address calculated by adding RVA of the section to its base address.

The Import table is then checked and any other required DLL are mapped into process address Space.After all the dll are located and mapped.it will check each DLL's Export section and IAT is fixed to point to the actual imported function address. Once all loaded modules are loaded, execution passed to the app's entry point

Automation

First we will develop a IAT parser using WINDOWS API via c/c++.we will extract relevant info about a file




```
ImportParser.exe abc.exe

DLL Name : KERNEL32.DLL

        Function : LoadLibraryA

        Function : GetProcAddress

        Function : VirtualProtect

        Function : VirtualAlloc

        Function : VirtualFree

        Function : ExitProcess

DLL Name : GDI32.dll

        Function : BitBlt
```

Please find the code here.

Remember: If you are compiling the code in 32 bit,then you have to run in 32 bit system only with DEP,ASLR disabled

Also we can print same info by using

python(http://stackoverflow.com/questions/19325402/getting-iat-and-eat-from-pe)

So this is our end of this part. We will discuss more interesting topics like the .tls virus, PE packer development in the next series.....

AUTHOR

Revers3r

Revers3r is a Information Security Researcher with considerable experience in Web Application Security, Vulnerability Assessment, Penetration Testing. He is also well-versed in Reverse Engineering, Malware Analysis. He's been a contributor to international magazines like Hakin9, Pentest, and E-Forensics. In his free time, he's contributed to the Response Disclosure Program. website: www.vulnerableghost.com

EDITORS CHOICE

- Malware Researcher's Handbook
- Deep Packet Inspection in Cloud Containers
- Exploiting CVE-2015-8562 (A New Joomla! RCE)

RELATED BOOT CAMPS

- Information Security
- Security Awareness
- CCNA

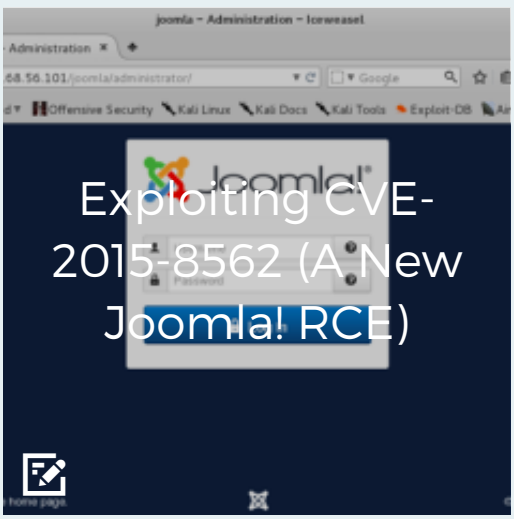
- PMP
- Microsoft
- Incident Response
- Information Assurance
- 8570

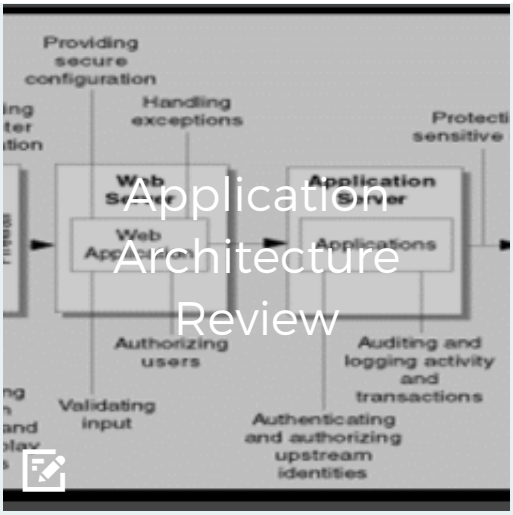
MORE POSTS BY AUTHOR

Malware Researcher's Handbook (Demystifying PE File)

Malware Researcher's Handbook: Introduction

Fingerprinting: Identifying Applications





About InfoSec

InfoSec Institute is the best source for high quality information security training. We have been training Information Security and IT Professionals since 1998 with a diverse lineup of relevant training courses. In the past 16 years, over 50,000 individuals have trusted InfoSec Institute for their

Connect with us

Stay up to date with InfoSec Institute and Intense School - at info@infosecinstitute.com

Like 567

Follow @infosecedu

Join our newsletter

Get the latest news, updates & offers straight to your inbox.

ENTER Y SUBSCRIBE

professional development needs!