# Controlling and Protecting Windows Applications by Analyzing and Manipulating PE File Format

**Rawaa P. Qasha** **Zaid A. Monther**
**rawa_qasha@uomosul.edu.iq**
*College of Computer Sciences and Mathematics*
*University of Mosul*

**Received on: 09/02/2011** **Accepted on: 02/11/2011**

## ABSTRACT

PE (Portable Executable) is the native file format of Windows32. Analyzing and manipulating the PE file gives valuable insights into the structure and work of Windows.

This research includes analysis the components of Windows executable files as a structure and defined values, to provide the capability of protection and controlling Windows programs by applying specified modifications that can be undid on PE specific value to stop the program from being executed by unwanted user. Also it includes analyzing the structure of PE file and comparing a specified part from PE with a same part from common viruses file, this process offers a good way to detect malicious programs and viruses in the computer by saving viruses signatures in a specified file and scanning all PE files. The other part of the research rebuild the Import Address Table of any PE files that may make a call to one of three important and essential registry API functions in order to control the using of these functions in the system using one of the API hooking techniques to control Undesirable programs.

The objective of the research is to control the executable files of the Windows system in order to provide protection for these files and the system as a whole.

Research program was developed using Visual C + + 9.0.

Keywords: Portable Executable file, Windows System , Protection , API Hooking.

## التحكم وحماية تطبيقات ويندوز عن طريق تحليل ومعالجة ملفات PE

**رواء بطرس بولص زيد عبد الإله منذر**
**كلية علوم الحاسوب والرياضيات**
**جامعة الموصل**

**تاريخ الاستلام : 2011/02/9** **تاريخ القبول: 2011/11/2**

## الملخص

إن الـ Portable Execution (المحمولة للتنفيذ) هو تنسيق الملف الأصلي لويندوز 32. إن التحليل والتعامل مع ملف PE يعطي نظرة قيمة في هيكلية وعمل نظام الويندوز.

يتضمن هذا البحث تحليل مكونات الملفات التنفيذية للويندوز كهيكلية وقيم معرفة، لتوفير إمكانية حماية ومراقبة برامج ويندوز عن طريق تطبيق تغييرات محددة بالإمكان التراجع عنها على قيمة محددة من ملف PE لإيقاف تنفيذ البرامج من قبل المستخدم غير المرغوب فيه.

ويشمل أيضا تحليل هيكلية ملف PE ومقارنة جزء محدد من الملف مع نفس الجزء من ملفات الفيروسات الشائعة، هذه العملية توفر وسيلة جيدة للكشف عن البرامج الخبيثة والفيروسات في الحاسبة خزن تواقيع الفيروسات

التي يتم الحصول عليها في ملف خاص ثم إجراء مسح لجميع الملفات التنفيذية في النظام. الجزء الأخر للبحث في إعادة بناء جدول استيراد لأي ملف PE يقوم باستدعاء إحدى دوال الـ API الثلاثة والأساسية في التعامل مع الرجستيري وذلك للسيطرة على استخدام هذه الدوال باستخدام تقنية اصطياد API بهدف التحكم بعمل البرامج والأنظمة الغير مرغوب فيها.

الهدف من البحث هو السيطرة على الملفات القابلة للتنفيذ لنظام ويندوز من أجل توفير الحماية لهذه الملفات والنظام ككل.

تم تطوير برنامج البحث باستخدام فيجوال سي ++ 9.0.

الكلمات المفتاحية :ملفات التنفيذ المحمولة ، نظام ويندوز ، حماية ، التقاط واجهة التطبيقات البرمجية،

## 1. Introduction:

Windows stores its executables in a special format called PE format, PE stands for Portable Executable. It’s the native file format of Win32, even NT's kernel mode driver use PE file format. The format of an operating system's executable file is in many ways a mirror of the operating system built-in assumptions and behaviors. Since, an operating system's executable format and data structures reveal quite a bit about the underlying OS [10].

Microsoft introduces the PE as a part of the original Win32 specifications. However, PE files are derived from the earlier Common Object File Format (COFF) [6].

The term "Portable Executable" was chosen because the intent was to have a common file format for all flavors of Windows, on all supported CPUs.

Each PE uses number of API functions reside in one or more of the DLL files to performs its task and, only the information about the functions is kept in a specific part of PE. That information includes the function names and the names of the DLLs in which they reside [6].

The registry is a simple, hierarchical database of information that Windows operating systems and applications use to define the configuration of the system. Without the registry, Windows would be nothing more than a collection of programs, unable to perform even the basic tasks that we expect from an operating system. Registry API functions are used to access and make modifications to registry components [1].

## 2. Related Work:

Many previous works presented and implemented analyzing PE format since is the standard executed file for Windows system. Shengying [15] surveys binary code analysis from the most fundamental perspective views: the binary code formats, several of the most basic analysis tools.

Shengying [15] and Galen [4] describe various ways of function interception and present a generic method to achieve this task without relying on commercial packages; the ways depend on modifying the intercepted function. In [5] Greg also presents different methods for API hooking methods, one of them is used and implemented in this work which depends on the format of PE file.

## 3. The Structure of PE File Format

The meaning of "portable executable" is that the file format is universal across win32 platform: the PE loader of every win32 platform recognizes and uses this file format [6].

The general layout of a PE file format is shown in the following figure [7]:

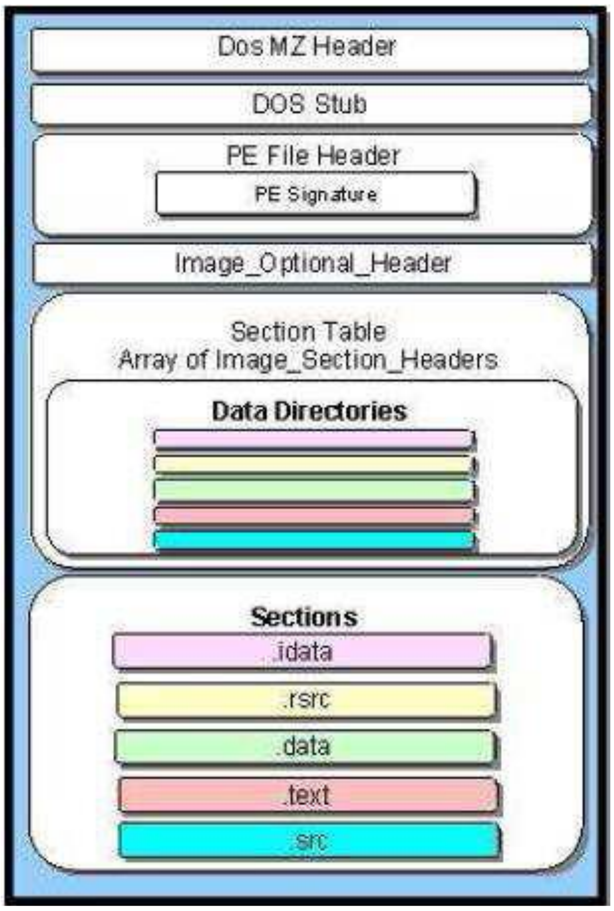

**Figure (1).** PE General Layout

All PE files (even 32-bit DLLs) must start with a simple DOS MZ header. It's provided to be used when the program is run from DOS, so DOS can recognize it as a valid executable and can thus run the DOS stub which is stored next to the MZ header. The DOS stub is actually a valid EXE that is executed in case the operating system doesn't know about PE file format. It can simply display a string like "This program requires Windows" or it can be a full-blown DOS program depending on the intent of the programmer [6][9].

The following few sections demonstrate some of PE parts that are used to accomplish the research work.

### 3.1 PE Header

The PE header is a general term for the PE-related structure named IMAGE_NT_HEADERS. This structure contains many essential fields that are used by the PE loader. In case the program is executed in the operating system that knows about PE file format, the PE loader can find the starting offset of the PE header from the DOS MZ header, since PE loader can recognize and use the file format in all win32 platforms depending on . It is the real file header [4][12].

The PE header is defined and has fields as follows:

Signature is the PE signature, "PE" followed by two zeroes.

FileHeader is a structure that contains the information about the physical layout/properies of the PE file in general.

Some fields in FileHeader are also important such as the OptionalHeader field, which is also a structure called IMAGE_OPTIONAL_HEADER32. It contains information about the logical layout in the PE file, such as:

DataDirectory structure that includes the **VirtualAddress** of the **IMAGE_IMPORT_DESCRIPTOR** which represent the address of the IAT (Import Address Table) [5][6].

## 3.2 Import Address Table (IAT):

The import address table is actually an array of **IMAGE_IMPORT_DESCRIPTOR** structures. Each structure contains information about a DLL the PE file imports symbols from, such as API functions.

This work uses the following fields:

**Name** field contains the RVA (Relative Virtual Address) to the name of the DLL, in short, the pointer to the name of the DLL.

**FirstThunk** contains an RVA of an array of **IMAGE_THUNK_DATA** structures which have the field **Function** that contains the address of the imported API function [5][13].

## 4. API Functions and DLL:

API stands for Application Programming Interface. An API is a set of calls, functions and routines that can be used by any application to access the operating system's functionality. The use of a common API allows applications to be developed once and deployed on multiple operating systems [13].

These functions are contained in Dynamic Link Libraries (DLL), which each program has access it when it’s executed. These functions are added only when the application is loaded into memory for execution.

A DLL is a file on disk (usually with a DLL extension) consisting of global data, compiled functions, and resources, that becomes part of a process. It is compiled to load at a preferred base address, and if there's no conflict with other DLLs, the file gets mapped to the same virtual address of a process. The DLL has various exported functions, and the client program (the program that loaded the DLL in the first place) imports those functions.

Windows matches up the imports and exports when it loads the DLL [2]. In Win32, each process gets its own copy of the DLL's read/write global variables. So, for sharing memory among processes, a shared data section must be declared in the DLL [8].

## 5. Windows Registry:

The registry plays a key role in the configuration and control of Windows systems. It is the repository for both system-wide and per-user settings. It is also a window into various in-memory structures maintained by the Windows executive and kernel. Application and system settings stored in the registry.

The registry is a database whose structure is similar to that of a logical disk drive. The registry contains keys, which are similar to a disk's directories, and values, which are comparable to files on a disk. A key is a container that can consist of other keys (subkeys) or values. Values, on the other hand, store data. Top-level keys are root keys[8].

The registry is accessible to the Win32 programmer via API functions. There are functions to create and delete keys, look up values within keys, and more [1][3].

Monitoring the registry API functions gives the ability to monitor all programs that attempt to change values in the registry, especially viruses and malicious programs, and control accessing to registry keys and values by hooking these functions and gives permission for continuing calling these functions or denied the operation.

## 6. Software Implementation:

The software contains three parts each of which deal with certain part of PE files to offer protection capabilities for the files from unauthorized user and detect viruses which are PE-like files and protect registry system from any ineligible using. The following sections demonstrate software parts:

### 6.1 Changing in PE Values:

This part performs modification on specified PE field to prevent unauthorized users from using the modified PE file and executing it. Each PE contains "**Machine**" field that is used to indicate the permission to execute the file or prevent it. "**Machine**" is the first field from the **IMAGE_FILE_HEADER STRUCT** that is part from the PE header structure **IMAGE_NT_HEADERS STRUCT.**

Machine field contains the CPU platform of the file is intended for. For Intel platform, the value is IMAGE_FILE_MACHINE_I386 (14Ch). So, changing the value of this field can be used as a quick way to prevent a program to be executed.

Changing Machine field is accomplished as shown in the following code:

```
HANDLE hFileMapping; LPVOID lpFileBase;
PIMAGE_DOS_HEADER dosHeader;
PIMAGE_NT_HEADERS pINTH;
hfile=CreateFile (name,GENERIC_READ|GENERIC_WRITE,0,
(LPSECURITY_ATTRIBUTES)NULL, OPEN_EXISTING, 0, NULL);
hFileMapping = CreateFileMapping(hfile, NULL, PAGE_READWRITE, 0,
0, NULL);
lpFileBase = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS,
0, 0, 0);
pINTH = (PIMAGE_NT_HEADERS)
pINTH->FileHeader.Machine=0x0000;
```

### 6.2 Detecting PE-Like Viruses:

The second part of the work presents a method to detect any virus and malicious program that has PE-like format.

Security products such as virus scanners look for characteristics byte sequence (signature) to identify malicious code. The quality of the detector is determined by the techniques employed for detection. A good malware detection technique must be able to identify malicious code that is hidden or embedded in the original program and should have some capability for detection of yet unknown malware [16].

In this work, detection operation is implemented by analyzing the virus file and gives it a signature by reading first 128 bytes (it is a unique value for each PE), storing the signature with virus name in a file and then use this signatures to detect the occurrence of this kind of viruses in the system.

This part consists of several steps including:

- Obtaining a signature for virus file using special purpose programs, such as OllyDbg. The first 128 byte is selected as virus signature. Each new signature is added to a file contains all virus signatures.

- Parsing the PE file and compare between the values obtained from a specific part from the PE file, that correspond to the same part of virus file, with the signature of virus file as follows:

```
hFile=CreateFile(virus_file_name, GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_READONLY, NULL);
ReadFile(hFile, pMem, sizeof(IMAGE_DOS_HEADER),
&ReadBytes, NULL);
DosHeader = (PIMAGE_DOS_HEADER) pMem;
if(memcmp(DosHeader,DosHdrSignature,128)==0)
MessageBox (NULL, "Safe","Scanner",MB_OK);
else
MessageBox (NULL,"infected", "Scanner",MB_OK);
```

This method had been applied on Windows systems XP and 7 and it differs from the traditional antivirus programs that it checks only exe files, so it consume less time and CPU usage.

### 6.3 Import Table Intercepting:

Third part of this work includes controlling three registry API functions [14]:

-RegCreateKeyEx(): used to create new keys.
-RegDeleteKey(): used to delete a key.
-RegOpenKeyEx(): used to open a subkey.

which are the most important ones. Controlling process is performed by hooking these functions through intercepting IAT of any PE attempt to call them as demonstrate in the following sections.

When an application uses a function in another DLL or PE file, the application must import the address of the function. Most applications that use the Win32 API do so through an IAT. Each DLL the application uses is contained in the application's image in the file system in a structure called the IMAGE_IMPORT_DESCRIPTOR. This structure contains the name of the DLL whose functions are imported by the application, and two pointers to two arrays of IMAGE_IMPORT_BY_NAME structures. The IMAGE_IMPORT_BY_NAME structure contains the names of the imported functions used by the application.

When the operating system loads the application in memory, it parses these IMAGE_IMPORT_DESCRIPTOR structures and loads each required DLL into the application's memory. Once the DLL is mapped, the operating system then locates each imported function in memory and overwrites one of the IMAGE_IMPORT_BY_NAME arrays with the actual address of the function[11][13].

So, to control the API used by an application, the address of the target function in the IAT can must be replaced by the address of the new own function.

The steps required to perform the replacement operation are:

- Parsing the PE file of the target application in memory to obtain the address of the IAT in the PE file.

- Replacing the IAT entry which represent the address of the original imported API function in DLL with the address of the own new function.

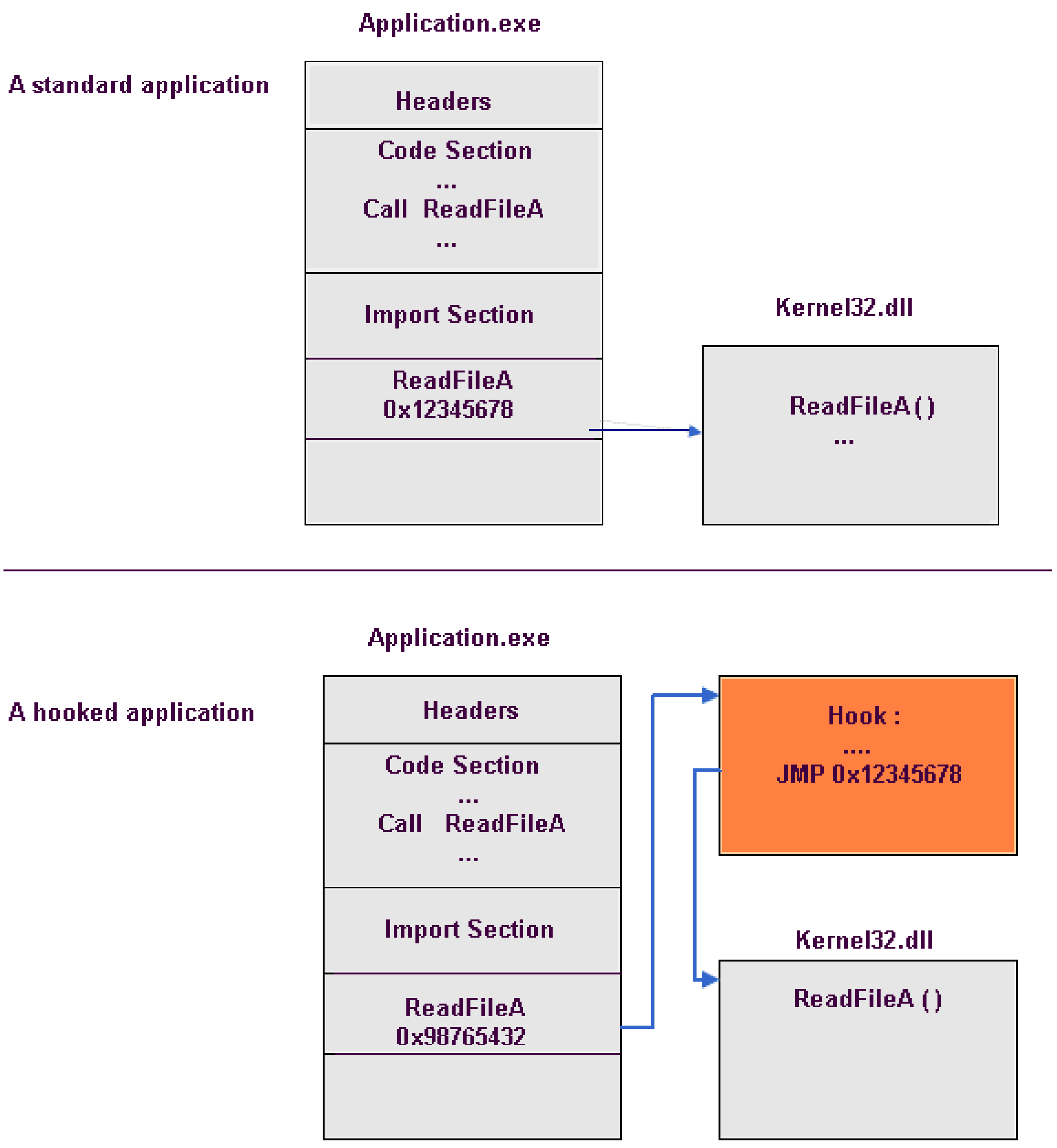

**Figure (2).** IAT Intercepting

This will cause all calls made by applications to the original function to be routed to the new function. The new one then can decide to pass control to original function or take some other actions [5].

Also this method required that the new function must be in the application's address space, when it be there, the hooking program can pares the PE of the target application in memory and replace the target function's address in the IAT with the address of the special function. Then, when the function is called, new function will be executed instead of the original function. Figure (2) depict the intercepting operation.

### 6.3.1 Executing code inside another process

The replacement process and the new function in the intercepting operation must be resided in the target applications (processes) address space for them to work. To achieve this, they must be a part of a DLL, and then the DLL must inject in the target process address space. Once the DLL is loaded into the target process, it can alter the execution path of commonly used APIs [5].

The code below implements the replacement process in this work using RegOpenKeyEx API function .

```
typedef LONG (CALLBACK *MY_RegOpenKey) (HKEY ,
LPCTSTR , DWORD , REGSAM , PHKEY );
MY_RegOpenKey Orig_RegOpenKeyEx =
(MY_RegOpenKey)GetProcAddress(hmod,"RegOpenKeyExW");
LONG CALLBACK Hook_RegOpenKeyEx(HKEY ,
LPCTSTR , DWORD , REGSAM , PHKEY);
hDllInst = hInst;
ReplaceIATEntryInAllMod("kernel32.dll",(PROC)Orig_RegOpenKeyEx,
(PROC)Hook_RegOpenKeyEx);
```

Injecting a DLL into the address space of an external process is a key element of a hooking system. It provides an excellent opportunity to have a control over process's thread activities.

This research uses Remote Threads method to inject DLL to another process as described below.

### 6.3.2 Injecting a DLL using Remote Threads

The common way to load the DLL into the target process is by creating what is called a remote thread in that process, the main function used for this purpose is CreateRemoteThread function, which has the following prototype[12][15]:

```
HANDLE CreateRemoteThread(
HANDLE hProcess, LPSECURITY_ATTRIBUTES lpThreadAttributes,
SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter, DWORD dwCreationFlags,
LPDWORD lpThreadId );
```

CreateRemoteThread as the name suggest is used to create a thread in another process. In this method, it can be exploit from the prototype matching between thread function and LoadLibrary() API function, as shown below:

```
DWORD WINAPI ThreadProc (LPVOID lpParameter);
HMODULE WINAPI LoadLibrary (LPCTSTR lpFileName);
```

This match gives a hint that the LoadLibrary can be used as a thread function, which will be executed after the remote thread has been created. Therefore, the target process will be forced to call LoadLibrary.

In other word, CreateRemoteThread() can be called with LoadLibrar() address and it will cause LoadLibrary call to be executed in target process's context. This way, LoadLibrary can load the DLL containing the hooking processes in the target process [5].

## 6. Software Overview:

Software interface window of this research consists of three parts, as depicts in figure (3):

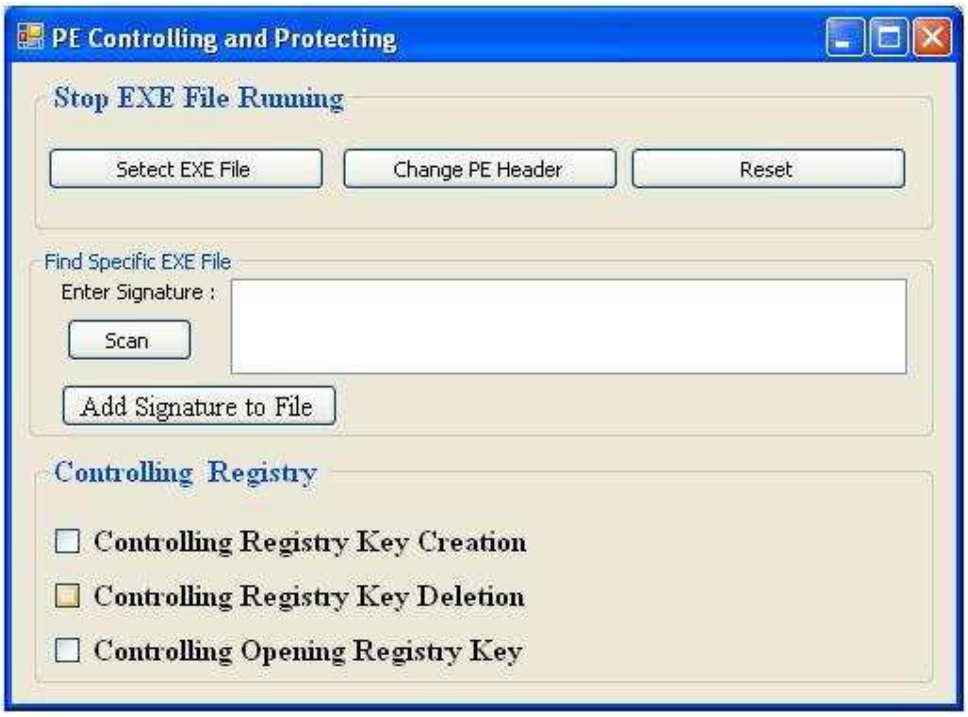

**Figure (3).** Software Interface Window

The first part can be used to prevent a selected PE from being executed using the following buttons:

- Select EXE File: is used to select a PE file from the system.
- Change PE Header: is used to modify Machine field in the selected file.
- Reset: returns Machine field to its original value.

The second part is used to perform scanning operation for comparing between virus signature and system PE files, then display the path of matched ones. After Virus signature is entered in the textbox, “Add Signature to File” button must be pressed to store virus name and its signature to a specified file then Scan button is used to start scanning operation between virus signatures from the specified signatures file and PE files in the system.

The third part contains three checked box:

- Controlling Registry Key Creation: is checked to control the operation of creating new registry key.
- Controlling Registry Key Deletion: is checked to control delete registry key operation.
- Controlling Opening Registry Key: is checked to control open registry key operation.

When any of the above checked boxes is unchecked, the corresponding API function will be unhooked and is called directly by any application.

## 7. Testing and Results:

Testing process was applied on data sets contains 50 viruses obtained from infected computers and from the website VX Heavens (http://vx.netlux.org) and 200 PE executable files produced by different runtime environments in the system. Also numbers of viruses’ signatures are stored in the file’s signature to be used for scanning stage such as: Fun.exe, DC.exe, Other.exe, SVIQ.exe, win.exe, WinSit.exe and killerjeff.exe The above data sets are used with research software and three antivirus programs. The results of detection rate are depicted in the following table:

**Table (1).** The Comparison Results

| **Detection program** | Research software | Kaspersky Antivirus 12.0.0.20 | Avira Antivirus 10.2.0.700 | ESET NOD32 Antivirus 5.0.93.0 |
|---|---|---|---|---|
| **Detection Rate** | 98% | 88% | 92% | 94% |

## 8. Conclusion

PE management process is a critical and important operation, since the sophisticated structure and organization of PE format, in addition to the fact that it’s an essential part of Windows operating system, so any error made through PE manipulation will cause serious problem in the system.

The Import Address Table is an essential part of the PE, and the knowledge of the import table performance helps to realize how an API is requested during running time, and how can be controlled. Also this research deals with another important part of Windows, the registry, so monitoring its main functions gives a powerful tools to protect the system from many malicious program and viruses.

## ***REFERENCES***

[1] Andrew S. Tanenbaum, 2009, "Modren Operating Systems", Pearson Prentice Hal, 3rd Edition.

[2] Charles Petzold, 1998, "Programming Windows", Microsoft Press, 5th Edition.

[3] David A. Solomon and Mark E. Russinovich, 2009, "Windows Internals, Including Windows Server 2008 and Windows Vista", 5th Edition.

[4] Galen Hunt and Doug Brubacher, 2000, " Detours: Binary Interception of Win32 Functions", Microsoft Research http://research.microsoft.com/sn/detours.

[5] Greg Hoglund, James Butler, 2005, " Rootkits: Subverting the Windows Kernel", Addison Wesley Professional.

[6] Iczelion, “Iczelion’s Tutorials for Win32ASM, Tutorial 24: Windows Hooks”, 2002, http://win32assembly.online.fr/tut24.html.

[7] “ Bypassing Anti-Virus Scanners”, 2004, InterNot Security Team.

[8] Johnson M. Hart, 2004, "Windows System Programming", Addison Wesley Professional, 3rd Edition.

[9] M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format", Microsoft Systems Journal, March 1994.

[10] Matt Pietrek, 2002, “An In-Depth Look into the Win32 Portable Executable File", MSDN Magazine, February 2002.

[11] Mathias Rauen, 2006, “API Hooking Methods”, http://www.madshi.net/apihooking.htm.

[12] Reverend Bill Blunden, 2009, “The Rootkit Arsenal - Escape and Evasion in the Dark Corners of the System", Wordware Publishing, Inc.

[13] Richter Jeffrey, 1999, "Programming Application for Microsoft Windows", 4th Edition, Microsoft Press.

[14] Richard Simon, 2000, "Microsoft Windows 2000 API SuperBible", Sams Publishing, 5th Edition.

[15] Seung-Woo Kim, 2005, "Intercepting System API Call", Intel Corporation.

[16] Vinod P. & V.Laxmi,M.S.Gaur, “Survey on Malware Detection Methods”, Department of Computer Engineering, Malaviya National Institute of Technology, Jaipur, Rajasthan.