Research Journal of Applied Sciences, Engineering and Technology 9(1): 33-39, 2015
DOI:10.19026/rjaset.9.1373
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: June 25, 2014 Accepted: September 20, 2014 Published: January 05, 2015

Research Article

An Effective Method for Protecting Native API Hook Attacks in User-mode

K. Muthumanickam and E. Ilavarasan
Department of Computer Science and Engineering, Pondicherry Engineering College,
Puducherry-605 014, India

Abstract: Today, many modern malware developers is taking the advantage of Application Programming Interface (API) hook technique to take the control of the victim computer which making it difficult to detect their presence. Because of the sophistication of rootkit tools, a remote attacker can use native API to compromise any computer which can later be used for many illegal activities such as sniffing network lines, capturing passwords, sending spam and DDoS attack, etc. Thus to protect end-system by identifying and preventing native API malicious code hooking is a challenging problem to the defenders. Today, many different malware-analysis tools incur specific features against malwares but manual and error-prone. In this study, we proposed a behavior-based monitoring detection system to effectively deal native API hooks in user-mode. Unlike other malware identification techniques, our approach involved dynamically analyzing the behavior of native API call hooking malwares. Comparing our experimental evaluation results with existing tools show better performance with no false positive.

Keywords: API hook, dynamic analysis, malicious code, rootkit, user-mode

INTRODUCTION

Today malicious software code which integrates stealthy rootkit technique has posed a serious challenge to computer security defenders. Hereafter, we use terms malicious code and rootkit, interchangeably. As large number of end users are running Windows operating system on the Internet, malware writers are taking the advantage of developing Windows rootkits recently. According to the history of information produced by Microsoft, 20% of malicious malware were removed from Windows XP operating system are stealthy rootkits (wikipedia). A rootkit is a collection of programs which enable the remote attacker to conceal its presence in the victim system without the user's consent so that they can able to monitor and control the compromised system secretly for an extensive time. In order to achieve their programmed tasks, rootkits try to alter the customary execution flow of the Operating System (OS) that can hide system resources such as processes, threads, files, kernel data structures and also other key information from the end-user. Due to the surreptitious character, rootkits are extremely difficult to detect.

There are two basic classifications of Windows rootkit: the user mode rootkit and the kernel mode rootkit. User-mode rootkits work in Ring 3 mode, which infects the operating system outside the kernel level. They replace drivers, dynamic linked-library files and various processes with their own versions, which don't show the rootkits' presence. They also intercept system calls between the kernel and software programs, making sure the forwarded information doesn't include any evidence of the rootkits. Kernel-level rootkits are based on the OS core, which controls all kinds of communications among hardware, software, processes and itself. They boot with and sometimes even modify the OS kernel. This lets the rootkit work above other system elements, giving the hacker control of the computer. The rootkit can thus present falsify information about its presence.

Before going further, it is necessary to understand how the user application is being executed by Windows operating system. Figure 1 shows the life cycle of execution path of a user-mode application that invokes WriteFile() system service routine which is implemented in kernel space. The following steps explain the same:

- The user application calls the WriteFile() function in user mode.
- The WriteFile() calls ZwWriteFile() native API function which has a stub in ntdll.dll.
- Then, ZwWriteFile() calls KiFastSystemCall function which in turn executes the SYSENTER instruction.
- In response to SYSENTER, the program control is transferred to KiFastCallEntry() which is located in ntoskrnl.exe as executive service.

Corresponding Author: K. Muthumanickam, Department of Computer Science and Engineering, Pondicherry Engineering College, Puducherry-605 014, India

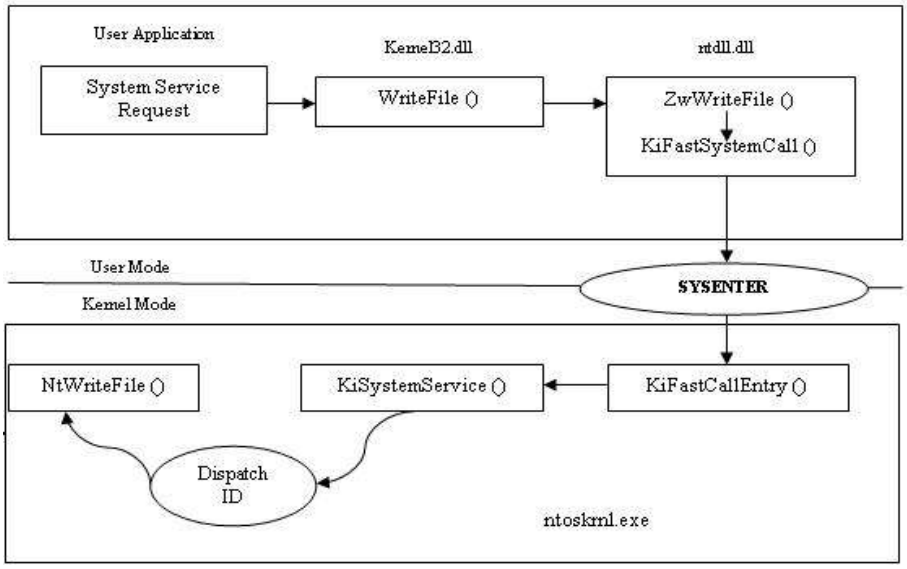

Fig. 1: Life cycle of a system call

- This will cause the KiSystemService dispatcher to call NtWriteFile() function using the dispatch ID.

Rootkits use several variations of hooking techniques during its lifetime. There have been many rootkit detection tools available. Each time such a tool is run, a log file is generated to keep a list of detected hooks. The amount of data in these log files is overwhelming as they hold information about each and every hook that had been detected on the system. On an average, each of these log files contains several hundred lines of data. The main contribution of this study is that we have devised a new procedure that can be used to make sense of the vast amount of information in these log files. In this study, we propose a user-mode native API hook detection system to protect system resources from injecting malicious code that uses Import Address Table (IAT) hook and inline hook. The objective of the proposed work is API calls to get assistance from Windows OS. If we can detect any unauthorized use of these API calls, the malicious code can be detected and stopped before misusing kernel level system services.

Hooking various API functions into the victim computer is an important attacking technique employed by sophisticated malware. To defeat current hook detectors, modern malware writers maintain discovering new hooking mechanisms. However, the existing malware analysis technique is typically manual or error-prone. This study proposed a behavior based monitoring mechanism that does not require prior information about hooking method to defeat user-mode hook attacks.

Background information: This section outlines the importance of Application Programming Interface in Windows OS. To launch malicious instructions, malicious codes need to interact with the OS through Windows subsystem API libraries. The actual implementation of the native API functions reside in ntoskrnl.exe which is located in the kernel. Each native API has a reference inside ntdll.dll which is isolated in the user mode. After the malicious instructions are deposited in a victim computer, code-injection attacks must use native API calls to do further damage.

Hooking is a set of code which alters the normal behavior of the operating system by intercepting the system API functions or information exchange passed between different system resources. Hooking can be used for either legal purpose, such as debugging and extending functionality or to host many illegal activities with the use of rootkit technique. Hooking can be used by malicious code such as rootkits, which try to hide themselves. As mentioned earlier, rootkits use different types of hooking techniques in order to remain hidden. In this study, we focus two user-mode API hooking techniques: IAT hooking and Inline function hooking.

Import address table hooking: The IAT is the most important call table of the user space modules. The IAT keeps the references of all routines exported by a particular Dynamic Link Library (DLL). And each DLL that an application is linked with, particularly at load time, will have its own IAT. Many executable files have embedded one or more IATs in their structure that are used to store the addresses of existing libraries that they import from DLLs. Most of the user land rootkits use the IAT hooking technique to intercept the API function calls. IAT entries are filled by the Windows loader at boot time. Thus, to maneuver an IAT, we have to access the address space of the request. One way to achieve this is by using DLL injection technique. Normally rootkits use DLL injection techniques to modify the address of the

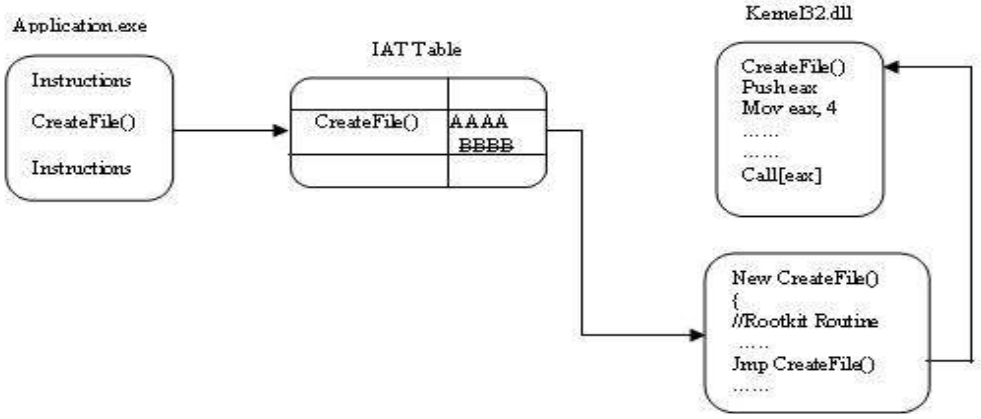

Fig. 2: IAT hook by malicious rootkit

specific function in the IAT to point to the address of the rootkit function where it is presented. So when the application calls a specific function, the rootkit function is called instead.

Figure 2 shows IAT hook through CreateFile API function. The rootkits had managed to create a hook by overwriting the address of the CreateFile function in the IAT of the user application. If we successfully modify the entry point of CreateFile in the IAT with the address of rootkit routine, all native API calls in the target process are rerouted to rootkit routine. Hooking a module's IATs using DLL injection can be accomplished by calling HookAPI() function as shown in Fig. 3. It pulls out the module's starting address and then parses the memory image to identify necessary IATs.

The walkImportLists() function checks the module's PE signature by adding a Relative Virtual Address (RVA) to the base address. Then checking each import descriptor will list all routines that are imported from the corresponding DLL. If Import Lookup Table (ILT) and IAT contain entries, then the names in the descriptor's IAT are compared against the name of the function we want to restore. If there is a match, substitute the address of the hooked function.

Detour hooking: Detour patching is another technique to divert the predefined execution path to malicious code without altering IAT call table entries. This technique is implemented by inserting a JUMP statement into the target routine to divert the execution path. So, whenever the currently executing thread executes this jump instruction, the control is transferred to a detour routine. The original portion of the code from the target function which we want to redeposit, in coincidence with the jump instruction returns back to the target code, is known as 'trampoline'. So the initial jump in the trampoline replaces a certain code when it is inserted and at the end, we execute the necessary instructions which was replaced and then bounce back to the target code. Figure 4 depicts this technique. Using this technique we can arbitrarily intercept the flow of execution.

```
HookAPI (File *fptr, char* apiName)
{
DWORD bAddress;
bAddress = (DWORD) GetModuleHandle (NULL)
return (walkImportLists (fptr, bAddress, apiName))
}
```

Fig. 3: Hook API function

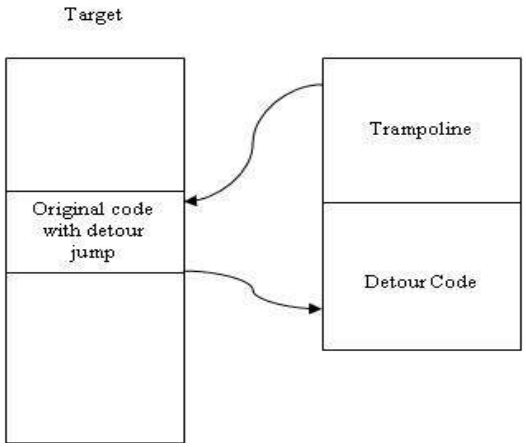

Fig. 4: Inline function hook by detour code

LITERATURE REVIEW

There have been few ideas proposed to detect native API hooks in Windows operating system. Ye *et al*. (2007) described an association mining based technique to analyze API execution flow. By associating API sequence using Portable Executable (PE) parser, they construct association rules and finally the malicious malware is identified. But this approach did not focus on various characteristic of a stealthy rootkit. As most computer malware has developed with the intent of infecting a Windows operating system, Kumar (2010) proposed a cross-view comparison based method to

identify hidden processes in user-mode. Liu *et al*. (2012) presented a review of rootkit detection techniques. Also, the authors developed X-Anti, a multi-way based detection method to detect different rootkits. In order to maintain their system, each node's information needs to be updated frequently and timely. Yi *et al*. (2010) presented a review to analyze Windows rootkits and various stealth techniques to attack the windows system. They also discussed various detection techniques that have been used by the detection tools today. Unfortunately, these techniques also bring new challenges to the detection and defense against rootkits.

White *et al*. (2012) developed a plug-in to effectively identify the contents of all user allocations. But it will not describe every possible allocation. Additionally paging issues, data structure invariant and some undocumented APIs in Windows environment were not discussed. Hejazi *et al*. (2009) reviewed API calls on the stack to locate some data structure, especially those which handles encryption. Their approach works without knowing the structure of data which was in user space. This limits their ability to retrieve user data. Deng *et al*. (2012) developed IntroLib, a tool to reveal user-level library call and behaviors which are generated by a malware based on hardware virtualization. In order to intercept library calls made by malware, IntroLib used page-table mechanism at the hypervisor level. This, however fails to detect malware that could obfuscate its memory structure and library calls directly invoked by malware.

Researchers have proposed a few solutions to protect Windows APIs. Wang *et al*. (2006) presented a static analysis method to detect malicious programs. Their method collects the calling sequences of native APIs from legitimate programs and sets up a data model using Support Vector Machine (SVM). Then, it detects malicious code by analyzing its calling sequences. Unfortunately, this method is unable to stop malicious code in real time and malicious code an easily mimic a legitimate calling sequence.

A kernel-mode protection system, named WHIPS introduced in Windows environment (Battistoni *et al*., 2004). WHIPS inspect every system request in the kernel. It validates the caller's the service it requests, process name and the parameters of the requests using a predefined access control database. It blocks requests that are invalid. The major challenge in using WHIPS is to define the access control database appropriately, in particular to decide the parameters safeness.

Malware is the commonly used weapon by attackers to compromise the victim machine in a network. As signature-based detection is not good against unknown malware, Ma *et al*. (2012) developed compiler level prototype tool (Auto Shadow) to generate the shadow process of malware related to original malware. This can be accomplished by generating a behavior-based sequence graph of the system call. Their solution detects malicious behavior by matching the system call with existing malicious behavior. This technique is more robust and more evade detecting known attacks.

Host level intrusion detection mechanism is used for detecting the general classes of malicious code. Malicious code can interact with Windows OS through Windows API. Rabek *et al*. (2003) presented a static analysis approach to monitor system calls at run time and to identify software executables. This approach is simple, practical and effective for user land malware detection. But it fails to perform, if malicious code directly invokes the kernel level service request. Wagner and Soto (2002) proposed a method to handle mimicry attacks in Linux environment. Their method records addresses of system call services into Interrupt Address Table (IAT). Whenever a process is waiting to get system service it was intercepted by their framework and checks whether the caller address is in the IAT. This method was not tested over Windows systems. Method such as Mansoori *et al*. (2012) can also be used to understand the activities of a malware that comes from un-trusted outside network using Honeypot.

PROPOSED METHODOLOGY

To monitor and detect native API hooking in the user space, we intercept native API calls in user mode and looking the traces of IAT entry modification and inline hooking. This section describes the principle and architecture of the proposed method. The proposed system is supposed to be installed in a clean system. The proposed system seizes native API system calls in user mode before they get service from the kernel. Figure 5 shows the proposed architecture of our system which consists of three important modules: DLL classification, IAT hook detector and inline hook detector.

DLL classification: Intercepting every system service calls that use native API is a tedious and time

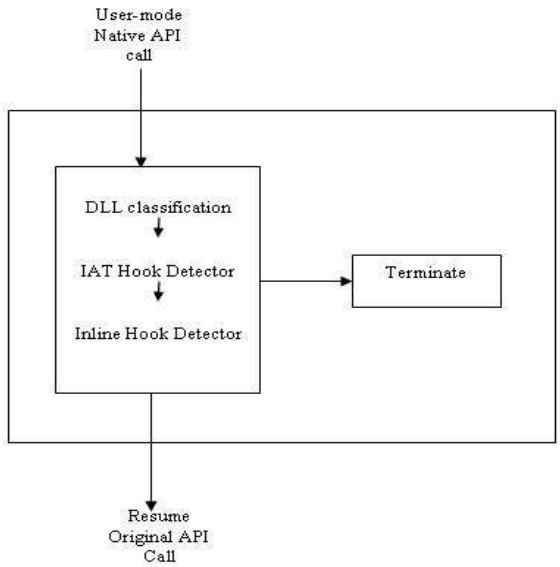

Fig. 5: Bare bones of the architecture

consuming process. So in order to allow legitimate system service calls to be serviced as normal, we have developed a new algorithm named it as "DLL Classification Algorithm":

```
/*DLL Classification Algorithm*/
Input(s): Imported DllName
Output: Classification of DLL's either as legitimate or malicious returntype FunctionName (DLL)
{
    //declarations
    Get dllName and handle
      using GetFileVersionInfo() check whether ((szDllName, dwHandle, dwCount, pBuffer) ! = 0)
      {
If not, use VerQueryValue() extract VarFileInfo and ValueLen;
      if (bVer && dwValueLen ! = 0)
      {
print dll file informations
if (extracred dll informations are valid)
{
legitimate dll
}
      }
     }
malicious dll
}
```

Since most of the malicious code cannot include properties such as vendor name, description and version details, the DLL/process classification algorithm verifies their properties to check whether the given process is malicious or not. To get the vendor name of the process, the GetFileVersionInfo() function is called to get the file version information buffer which contains all the property values of a dll. To get the specified property value of a dll, the VerQueryValue() function is invoked. Finally VerQueryValue function tells whether the dll is either legitimate or malicious.

IAT hook detector: The IAT Hook Detector (IHD) is the first level of defense module against native API hook in the user-mode. All processes which are classified as suspicious are given as input to this module. To detect native API hooks in IAT, the IHD performs the following steps:

- The IHD obtains a list of currently running processes by calling the EnumProcesses function.
- For each process, the PrintProcessNameAndID function is called by passing it to the process identifiers which in turn call functions OpenProcess to get the process handle, EnumProcessModules to extract the module handles and GetModuleBaseName to find the name of the executable file along with process id.
- Then IHD compares each process with unknown processes. If legitimate, the LoadLibrary function is invoked to load the process into memory. After reading the MS-DOS header (MZ), PE, PE extended and section header from the executable, IHD determines DLL of an application which has been loaded and also the address range of each DLL in memory. Then IHD examines the IAT of the executable to examine the entries in each IAT.
- Finally, if any entry drops outside of the module's address collection, the IHD stop executing the DLL; otherwise it will be serviced as a legitimate system service call.

Inline hook detector: As an alternative approach to IAT hooking, many malware writers keep call table entries within the requested range and instead modify the code that it points to. Inline Hook detector (LHC) is another level of defense to strengthen our system. First, LHC reads the executable file to reach the Export Address Table (EAT). So it calls the ExportAddrTable function to get addresses of each function in EAT which are invoking native APIs. To detect the detour patches in every DLL's function, LHC uses the CheckForOutside function to trap 0xE9 or 0xEA which will be opcode for the unconditional near and far jump respectively, in the first five bytes of the DLL's API function. Finally LHC will get the address where the CPU will jump to the function and then checks the CPU jump address with the acceptable range to determine whether it lies outside the address range of the DLL. If any function which are not in the predetermined address range is considered to be malicious.

Limitation: Even though our system successfully defeat native API hooks in user-mode, it still suffers from the following limitations. First, the proposed system is assumed to be installed in a clean system. Any hook which attempts to alter some bytes randomly instead of the first five bytes of a DLL function could not be detected as malicious. Additionally, the proposed method cannot detect malicious code attacks that directly target kernel-mode data structures, specifically System Service Dispatch Table (SSDT). Hence, to provide a tight protection, it needs to work with anti kernel hooking mechanism.

RESULTS AND DISCUSSION

We have setup an assorted experimental environment in a computer with Microsoft Windows 7

Table 1: Malware family with hook type

Malware family name	Type of hook
Papras	IAT
Bacalid/DetNat	IAT
Haxdoor-B	INLINE
Agent	IAT
Feebs-A	INLINE
Qukart	INLINE
Virut	INLINE
Alman-B	IAT
ProAgent	INLINE
Alman-A	IAT

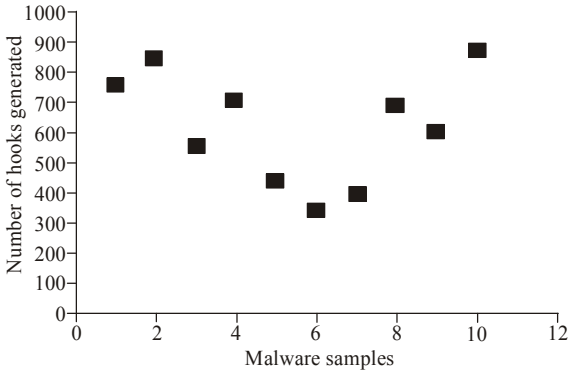

Fig. 6: Total number of hooks generated by different malwares

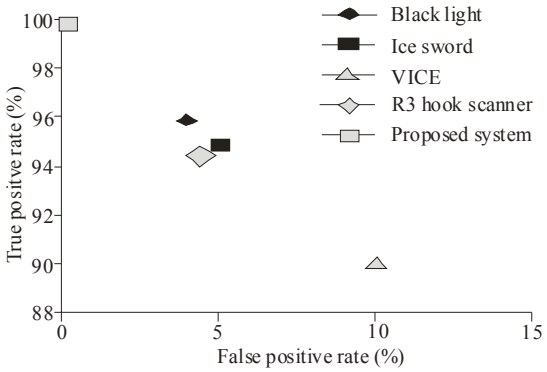

Fig. 7: ROC performance analysis using ROC

Table 2: Malware detection analysis

	Tool				
Malware	BlackLight	IcsSword	VICE	R3 hook scanner	Proposed system
Papras	Yes	Yes	Yes	Yes	Yes
Bacalid/DetNat	Yes	No	Yes	No	Yes
Haxdoor-B	No	No	No	No	No
Agent	Yes	Yes	Yes	No	Yes
Feebs-A	Yes	Yes	No	Yes	Yes
Qulart	No	No	Yes	No	Yes
Virut	Yes	No	No	No	No
Alman-B	No	Yes	Yes	Yes	Yes
ProAgent	No	No	No	No	Yes
Alman-A	No	No	Yes	No	Yes

SP3 with ACER core 2 Duo 2.93 GHz CPU and 2 GB RAM. The virtual machine runs windows XP. In order to evaluate the effectiveness and performance of the proposed mechanism, we have obtained 20 user-mode malware samples (two samples from each family) as shown in Table 1 from http://www.offensivecomputing.net.

We have run each malware sample in a controlled environment to detect the hooks generated by them in the victim machine. Few malware executable does not hook library functions when it starts running. But, as time increases, the number of hooks also increased. Figure 6 shows the result of the total number of hooks generated by each malware sample.

To test the precision rate of the presented approach, we have run the malware samples against existing tools BlackLight, IceSword, VICE tool, R3 Hook scanner and also against the proposed mechanism. Table 2 shows the resulting data.

We also tried to assess the number of False Positive (FP) that the proposed prototype would fabricate. In order to determine FP, we have taken 10 legitimate API hooks which implemented as a C++ file in Microsoft Visual studio. Every time we executed a C++ file, our proposed method identifies all with no false positives. We have produced the standard ROC curve analysis to estimate the accurateness of our approach. Generating a ROC curve is a useful model to estimate the transaction between the false positive rate and the true positive rate. We have shown ROC curve in Fig. 7 for different existing methods, including our system.

The ROC curve confirms that the proposed system achieves 100% accuracy rate with no false positive while detecting native API hooks in user-mode, especially in Windows operating system.

CONCLUSION

With an increasing amount of malware adopting rootkit techniques to evade antivirus software, further research into defenses against rootkit attacks is absolutely essential. In this study, we have devised a method to trace and prevent user-mode malware with native API hook functionality in windows operating system. There are several tools available which can be used to detect these types of hooks, but these tools cannot focus rootkits with native API hooks. We have evaluated our work using 10 real-world windows user-mode malware rootkit samples. The performance shows that the proposed approach incurs 20% better performance. With an increasing amount of malware adopting rootkit technologies to evade antivirus software, further research into defenses against rootkit attacks is absolutely essential. In the future, we plan to design a kernel level Process Authentication Mechanism (PAM) to prevent malicious code attacks. To optimize our PAM, we plan to devise a novel System Call Classification algorithm that will verify only suspected processes.

REFERENCES

Battistoni, R., E. Gabrielli and L.V. Mancini, 2004. A host intrusion prevention system for windows operating systems. Proceeding of 9th European Symposium on Research in Computer Security (ESORICS '04), pp: 352-368.

Deng, Z., D. Xu, X. Zhang and X. Jiang, 2012. IntroLib: Efficient and transparent library calls introspection for malware forensics. Digit. Invest., 9: S13-S23.

Hejazi, S.M., C. Talhi and M. Debbai, 2009. Extraction of forensically sensitive information from windows physical memory. Digit. Invest., 6: S121-S131.

Kumar, E.U., 2010. User-mode Memory Scanning on 32-bit & 64-bit windows. J. Comput. Virol., 6(2): 123-141.

Liu, L., Z. Yin, S. Yuli, H. Lin and H. Wang, 2012. Research and design of rootkit detection method. Phys. Proc., 33: 852-857.

Ma, W., P. Duan, S. Liu, G. Gu and J.C. Liu, 2012. Automatically evading system-call-behavior based Malware detection. J. Comput. Virol., 8: 1-13.

Mansoori, M., O. Zakaria and A. Gani, 2012. Improving exposure of intrusion deception system through implementation of hybrid honeypot. Int. Arab J. Inf. Techn., 9(5).

Rabek, J.C., R.I. Khazan, S.M. Lewandowski and R.K. Cunningham, 2003. Detection of Injected, dynamically generated, and obfuscated malicious code. Proceeding of the 2003 ACM Workshop on Rapid Malcode, pp: 76-82.

Wagner, D. and P. Soto, 2002. Mimicry attacks on host-based intrusion detection systems. Proceeding of 9th ACM Conference on Computer and Communications Security, pp: 255-264.

Wang, M., C. Zhang and J. Yu, 2006. Native API based windows anomaly intrusion detection method using SVM. Proceeding of IEEE International Conference on Sensor Networks, Ubiquitous and Trustworthy Computing (SUTC'06), 1: 514-519.

White, A., B. Schatz and E. Foo, 2012. Surveying the user space through user allocations. Digit. Invest., 9: S3-S12.

Ye, Y., D. Wang, T. Li and D. Ye, 2007. IMDS: Intelligent malware detection system. Proceeding of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp: 1043-1047.

Yi, X., H. Da-Rong and S. Jun, 2010. Analysis of windows rootkits stealth and detection technologies. Proceeding of the 2nd International Conference on Applied Robotics for Power Industry, 2010.