The Study of SSDT Hook through Comparative Analysis between Live Response and Memory Image

Muteb Alzaidi, Ahmed Alasiri, Dale Lindskog, Pavol Zavarsky, Ron Ruhl, Shafi Alassmi

Information Systems Security Management

Concordia University College of Alberta, Edmonton, Canada

{Muteb.Alzaidi, Medo0o.2004, alassmi.shafi}@gmail.com

{dale.lindskog, pavol.zavarsky, ron.ruhl}@concordia.ab.ca

Abstract

The purpose of a kernel rootkit is to prevent detection of a compromised operating system. System Service Dispatch Table (SSDT) hooking has been employed by most Windows kernel rootkits (Kornblum 2006, Rie 2006) as a method of hiding files, processes and registry keys from system and investigative utilities, by determining what functions become the targets within the operating system. This paper describes a comparative analysis between the detection capabilities of a particular live response utility, MANDIANT Redline, and a memory image analysis utility, Volatility, when the SSDT has been hooked by a rootkit. This comparative analysis shows that Redline, when compared with Volatility, is significantly limited in its ability to detect SSDT hooks. We show that the limitations of this live response utility are due to the fact that it relies on system calls for detection of SSDT hooks. We further show that Redline fails to uncover other vital evidence that is both available in the memory image, and helpful to the investigation.

Keywords: SSDT, rootkit, function, hook, system call, live response, memory image

I. INTRODUCTION

Brian D. Carrier recently revealed that one of the more recent and famous worms, Code Red, “resided only in memory and never wrote anything to disk” (Carrier 2003). An example like Code Red further illustrates the well understood importance of collecting and analyzing volatile data. However, examination and collection of volatile data can be done in two quite different ways: through operating system calls, which is the method most live response utilities use, or through the analysis of raw memory, accessed either directly on the compromised system, or indirectly via an image of that system's memory.

In this paper we demonstrate, in practical experiments, and using well-known forensic investigation utilities and bona fide Windows kernel rootkits, that there are significant theoretical and practical limitations to the reliance on operating system calls for the collection and investigation of volatile data related to SSDT hooks, and that these limitations are, in the case of SSDT hooking rootkits, substantially overcome by employing utilities that investigate raw memory. These considerations further demonstrate the pressing, practical importance of continued research into memory forensics, continued development of memory image forensic utilities, and the integration of such theory and practice into real world live response and digital forensic investigations.

This paper is organized in the following manner: Section II presents a full explanation of the system service dispatch table; Section III provides a discussion of system service dispatch table hooking; Section IV presents the details of our experimental design and also provides information on the tools used in the comparative analysis; our results and some discussion of them are presented in Section V; in section VI, our conclusions and recommendations for future work are given.

II. SYSTEM SERVICE DISPATCH TABLE

The System Service Dispatch Table (SSDT) is a dispatch table contained by Windows Operating Systems. Microsoft Windows systems use the SYSENTER instruction to jump from user mode (ntdll.dll) to kernel mode (ntoskrnl.exe). When triggered, the SYSENTER instruction will cause the process to move to kernel mode by activating KiSystemService, the system service dispatcher, which then uses the information delivered from user mode to locate the address of an API routine (Kornblum 2006). The system service number, known as the dispatcher ID, is then used as an index entry into address lookup tables (Hoglund and Butler 2007, Blunden 2009, Light et al 2010), as shown in Figure 1 as step number 1 .

As shown in step number 2 of Figure 1, the system service number is 32 bits in size. Bits 0 through 11 indicate what service will be called, and bits 12 and 13 specify one of four possible service descriptor tables. In a Windows environment, only two tables are in fact used, and those tables are: SSDT (0), which describes the KeServiceDescriptorTab1e, and SSDT (1), which describes the KeServiceDescriptorTab1eShadow. If the value of both bits 12 and 13 are 0, then that indicates that the KeServiceDescriptorTable is being used. Whereas, if bit 12 is 1 and 13 is 0, then that indicates the KeServiceDescriptorTab1eShadow is being used. This is indicated in Figure 1 as step number 3.

In KeServiceDescriptorTab1e, the system service number ranges from 0x0000 to 0x0FFF, while the KeServiceDescriptorTableShadow will have system service numbers ranging from 0xl000 to 0xlFFF (MOYIX 2008). Moreover, the two service descriptor tables contain two substructures, generally referred to as System Service Tables (SSTs), and specifically as KiServiceTable and W32pServiceTable. An SST is an address lookup table that can be defined in terms of the following C structure (Zhang and Bi 2011, Blunden 2009, Zhang et al 2009) which we depict in step number 4 of Figure 1. The C structure is declared like this:

```
typedef struct _SYSTEM_SERVICE_TABLE
{
PDWORD      serviceTable;   //array of function pointers
PDWORD      filed2;     //not used in Windows free build
DWORD    nEnters; //number of function pointers in SSDT
PBYTE          argumentTable;       //array of byte counts
} SYSTEM_SERVICE_TABLE;
```

This C code shows that the first attribute, serviceTable, “is a pointer to the first function of an array in virtual addresses” (Blunden 2009), where each of those addresses is an entry point to a routine in kernel space. This array is known as the System Service Dispatch Table (SSDT) which contains KiServiceTable and W32pServiceTable. The second attribute in the C structure, filed2, is not used in the Windows free build. The third attribute, nEnters, specifies the number of functions in the SSDT array. The last attribute,

argumentTable, is a pointer to the first component of an array; this array is referred to as a System Service Parameter Table of bytes. Each byte in the System Service Parameter Table shows the amount of space allocated for function arguments when the corresponding SSDT routine is called (Wu et al 2010, Blunden 2009).

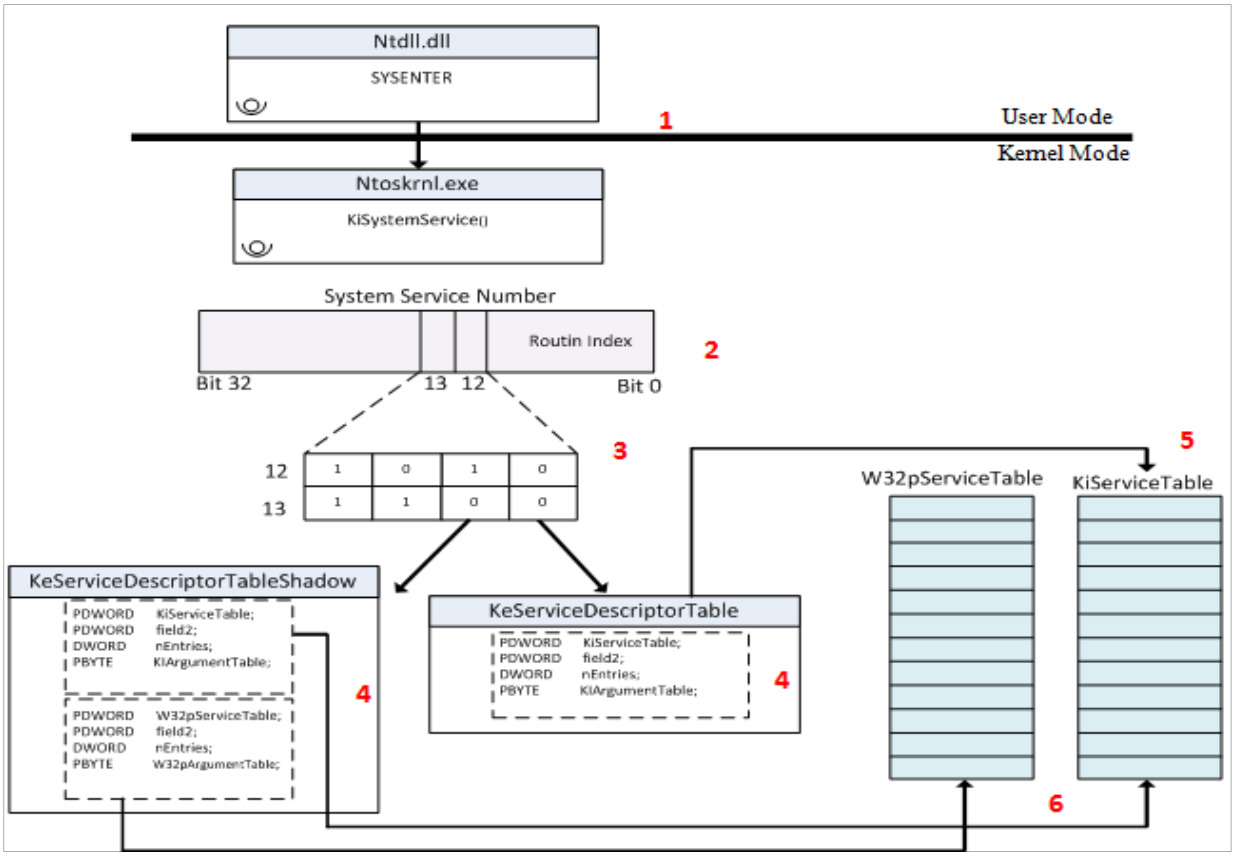

Fig. 1 System Service Dispatch Table Architecture

Additionally, the first 16 bytes of the KeServiceDescriptorTab1e structure in the SST (KiServiceTable) describe the SSDT for Windows API routines, as indicated in step number 5 of Figure 1. In addition, the first 32 bytes of the KeServiceDescriptorTab1eShadow structure indicate two SSTs which describe two SSDTs. The first 16 bytes is just a replication of SST in the KeServiceDescriptorTab1e, and the second 16 bytes indicate a SSDT (W32pServiceTable) for both the user and graphical user interface, and which "is implemented by the win32k.sys kernel-mode drive"(Blunden 2009), as shown in Figure 1 as step number 6.

Moreover, when a process from ntdll.dll calls the system service dispatch, KiSystemService, through kernel space, it loads the EAX registry with the system service identifier number (i.e., an index into the system function requested). The process then loads the EDX registry which contains the address of the function parameter in user mode. At this stage, the system service dispatcher confirms the number of parameters and copies them from the user stack onto the kernel stack. It then calls the function stored at the address indexed in the SSDT by the service identifier number in EAX (Blunden 2009, Hoglund and Butler 2007).

III. SYSTEM SERVICE DISPATCH SERVICE HOOK

Successfully hooking a function in the SSDT requires an ordered series of steps. Firstly, the rootkit needs to disable memory access protection in kernel mode by modifying control register CR0 (Blunden 2009, Hoglund and Butler 2007). After disabling that protection, information leading to the target function needs to be acquired. In particular, the base address of the function table and the index (EAX registry) of the target function need to be known. Rootkits usually call a function called MmGetSystemRoutineAddress (Microsoft 2010) which will locate the base address of KeServiceDescriptorTab1e and KeServiceDescriptorTab1eShadow. Once the base address of the target table is obtained, then the attacker can get the function needed from that table and the rootkit can hook a target function (Ligh et al 2011). The rootkit employs the SSDT hook by replacing the address of the system routine with the address of the target function. When the system calls the function that has been hooked in SSDT, it will be pointed to the rootkit code. The SSDT will search for the appropriate system call address of the function requested through user mode and return it; the returned value will point to the rootkit code. The rootkit at this stage has the ability to act as a man-in-the middle, deciding what information and programs the user does and does not see (Hoglund and Butler 2007, Light et al 2010, NIST 800-61, Wu et al 2010).

IV. COMPARATIVE ANALYSIS BETWEEN LIVE MEMORY AND MEMORY IMAGE

An important consideration in digital forensics is a tool's capability to detect rootkits, and to analyze their impact on the system. Our experimentation with Redline and Volatility evaluates these forensic utilities' ability to detect and analyze SSDT hooking rootkits, and in this section, we describe these two utilities and our experimental design and methodology.

A. **Research Methodology**

We used Redline and Volatility 2.0 to investigate systems in three separate cases where the system was infected with a SSDT hooking rootkit. The only difference between these three cases was the specific rootkit under investigation (see Table 1). All cases had the same experimental environment, and the rootkits were chosen randomly.

TABLE I

THE MODE OF THE EXPERIMENTAL RESEARCH

Rootkit's name	**Target System**
Runtime2 (Case 1)	Windows XPSP3
Wincom32 (Case 2)	
Blackenergy2 (Case 3)	

Memory image analysis was performed on a VMware workstation 8.0 guest machine running Windows XP SP3, with the host machine running Ubuntu 10.4 Linux, equipped with an Intel (R) Core (TM) 2 CPU T7250 2.00GHz.

In all cases, the rootkit sample was launched, and then a memory image was taken by suspending the system using the suspend feature provided by VMware. This memory image was analysed using Volatility 2.0. Starting with pslist, a Volatility plugin used to observe the processes that were running on the system when the memory image was taken; the rootkit's process termination was captured. Next, the SSDT plugin was used to observe which functions had been hooked by the sample. That, along with

further analysis using other plugins such as SSDT_EX, psxview, impscan and others, enabled us to investigate memory image.

Memory acquisition and analysis was conducted three times, and between each iteration, a five minute time interval was allowed to elapse in order to observe any dynamic change that would occur, as well as to help ensure reliable comparison.

Each time we launched the rootkit sample and took the memory image, live analysis using Redline was performed. Redline has the capability to detect SSDT hooks applied by a rootkit. Redline also flags processes as suspicious, and then investigates the potentially infected processes. Volatile data is constantly changing, and we monitored for such change by analysing the live system with Redline over an interval of time after rootkit infection, and compared these results with the static memory image, and with subsequent memory images obtained at these intervals.

B. **Memory image analysis**

We captured a full memory image in all three cases, and employed Volatility framework and the Interactive Disassembler Professional (IDA PRO) to investigate the images. The Volatility Framework is an open collection of tools which can be used for extracting digital artifacts from volatile memory (RAM). These techniques for digital artifact extraction are conducted entirely independently from the system being examined, yet they are able to provide insight into the runtime condition of the system (The Volatility Framework, IDA Pro). The utility has the ability to detect SSDT hooks via the SSDT plugin, and further investigate rootkit hooks using other plugins, such as ssdt_ex, pxsview, callbacks, threads, procmemdump, modules, filescan, hivelist, printkey and impscan. IDA PRO is a dissembler and a debugger used to analyse malware code. It is a dependency for some of the Volatility plugins that we used.

C. **Live response analysis**

In a live, real-time state, a system under investigation stores crucial, transitory data that can expose to the examiner the precise state of the system. Redline can be used as a live response utility to investigate a system infected with a SSDT hook. Redline, a free utility from MANDIANT, hastens the procedure of treating hosts suspected of being infected or otherwise compromised, while simultaneously providing support for in-depth analysis of live memory. It was designed to assist in uncovering even well-hidden malware, and has the ability to detect and examine SSDT hooks in some detail; Redline aids in the investigation of processes, hooks, drivers and devices, DLL injection, network ports and connections, and untrusted handlers (Redline, Malin 2008).

V**. DISCUSSION AND RESULTS**

Table 2 presents the data observed and investigated using both forensic utilities: Redine in a live response context, and Volatility for the investigation of memory images. A check mark (√) in Table 2 indicates evidence successfully collected; an x mark (×) in Table 2 indicates where data was not obtained from the utility.

TABLE 2 COMPARISONS BETWEEN REDLINE AND VOLATILITY

	Case 1 (runtime2)		Case 2 (wincom32)		Case 3 (blackenergy2)	
	Redline	Volatility	Redline	Volatility	Redline	volatility
Processes list	√	√	√	√	√	√
Process termination	x	√	x	√	x	√
SSDT Hooks	√	√	√	√	x	√
Imported functions	√	√	√	√	x	√
String section	√	√	√	√	x	√
Suspicious processes	√	√	√	√	√	√
Threads analysis	x	√	x	√	x	√
Device tree	√	√	√	√	x	√
Files scan	√	√	√	√	√	√
Call-backs routines	x	√	x	√	x	√

In all three cases we found that Redline was unable to detect terminated processes, infected threads, call-back routines and, in one case (case 3), SSDT hooks. The inability to detect blackenergy2 in case 3 indicates that Redline's success in detecting the SSDT hook depends upon the of type of function that has been hooked. We discuss this in more detail below.

Redline failed to obtain certain types of evidence about the rootkits, and there are many reasons for this. Information about terminated processes is unavailable to Redline, since this information resides in unallocated space in RAM. Information can be difficult for a live response utility like Redline to uncover when it is located in unallocated RAM, since unallocated RAM is not mapped to kernel mode (Aljaedi, et al 2011). Another limitation is that a live response utility like Redline tends to operate in kernel mode.

Volatility directly accesses memory (via a memory image), and therefore can retrieve unallocated data; it was in unallocated memory that we were able to find evidence of the rootkit's installation on the system, including terminated process that indicated the rootkit had infected the system. We must bear in mind; however, that memory image analysis is only able to obtain this kind of data if it has not been overwritten. This fact was verified in our experiments, where after a sufficient interval of time, Volatility was no longer able to detect the terminated process corresponding to the execution of the rootkit installer.

Furthermore, kernel space needs to access kernel thread block (KTHREAD) to perform thread preparation and synchronization for running threads. Some vital key fields of the KTHREAD structure are described briefly in Table 3 (Russinovich and Solomon 2009).

TABLE 3 KTHREAD STRUCTURE ELEMENTS

Element	Description
Dispatcher header	KTHREAS start with it because thread is object and can be waited to execute
Execution time	Total CPU time
Cycle time	CPU cycle time
Pointer to kernel stack information	It point to base and upper address of the kernel stack
Pointer to system service table	Any thread starts by point to the main system service table (KeServiceDescriptorTable).
Mutant list	List all mutant thread have
Scheduling information	Thread scheduling
Pointer to TEB	Thread ID, TLS (Transport Layer Security) and, PEB pointer, and other user-mode information

The KTHREAD structure includes an entry which points to the SDT that is used by the thread, as Table 3 shows (honeynet 2008). This pointer in KTHREAD points to KeServiceDescriptorTable. This fact is used by Volatility's threads plugin to identify threads infected by the rootkit. In our experiments, the Volatility threads plugin was able to scan threads associated with each process to identify orphans, i.e. threads with a start address that does not map back to the PsLoadedModuleList in kernel space, threads attached to other process's address space, and threads infected by SSDT hooks (Light 2011). Since thread initiation is a necessary step in process creation, infected processes were uncovered in our experiments. Figure 2 is one sample of Volatility's threads analysis, and shows the services.exe process (736) and its own thread (1276) is infected with rootkit hooks; in fact the rootkit targeted this function for DLL injection. In addition, when infected threads were analysed using the memory image, processes infected with the rootkit hooked functions were detected. On the other hand, the live response utility, Redline, does not provide threads analysis.

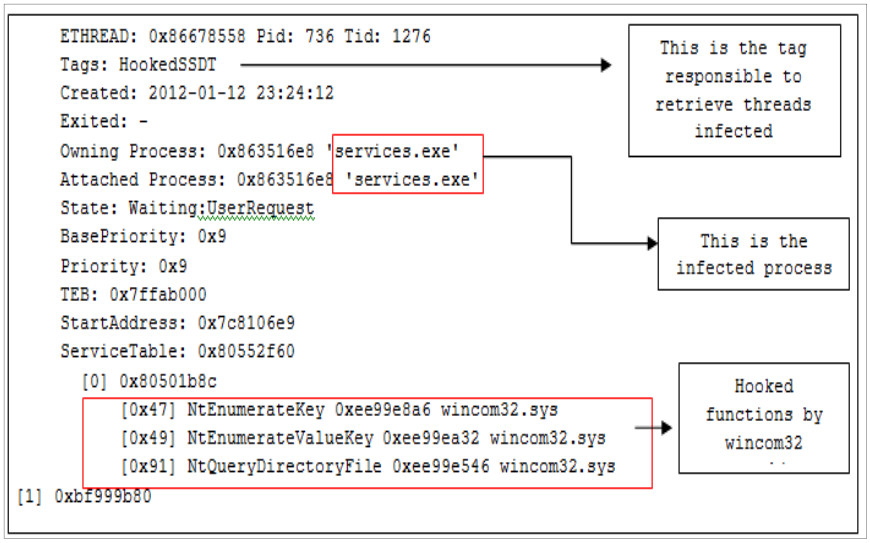

Fig. 2 Thread analysis sample in memory image

Volatility includes a call-back routines plugin that enables us to identify routines hooked by the rootkit. When the runtime2.sys device driver is loaded into the kernel, it is able to covertly gain control of the system. In fact, in all three cases the rootkit at this stage changed the SSDT to point to a function it provided, instead of win32k.sys, in order to create keys in the registry. In case 1, the runtime2 hook PsSetCreateProcessNotifyRoutine created a process in the system and an IoRegisterShutdownNotification, which is a shutdown callback routine (see figures 5 and 6 in our appendix). The Redline utility is unable to produce this evidence.

In all three cases the rootkits hook the functions in the SSDT and don't release their hook. Furthermore, there are common functions among the sample rootkits that target the functions NtEnumerateKey, NtEnumerateValueKey NtSetValueKey. The functions are employed to set and return key or subkey values in the registry. When any calls are placed to these registry functions, the rootkits' hooked function returns modified results in order to hide their presence on the target system.

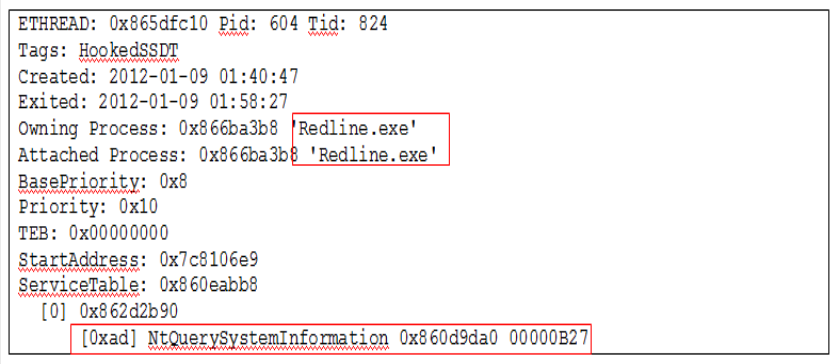

Fig. 3 Volatility detect live response utility infection by rootkit hooks

For the remainder of this section, we will elaborate on our most remarkable experimental finding. In case three (blackenergy2), Redline was unable to detect the hooks by this rootkit. As we shall see, this failure is explained by the fact that Redline, like most live response utilities, relies on the compromised system. Most rootkit developers are aware of the detection methods used by live response utilities, and as is shown in Figure 3, Redline itself was infected with blackenergy2's hooks.

Specifically, the rootkit hooks the NtQuerySystemInformation function, either to conceal itself or to deceive a detection utility, by returning false information from the system call made by the utility. Detection via NtQuerySystemInformation operates by determining the address range of SSDT in kernel space. The first step is to call the NtQuerySystemInformation function, and a failure to retrieve the necessary information from this function will result in the failure of this method. In fact, the NtQuerySystemInformation function is responsible for retrieving valuable information that most live response utilities rely on (see Table 4 in appendix). Figure 4 shows the trace of system calls made by Redline to acquire data, including calls to the NtQuerySystemInformation function.

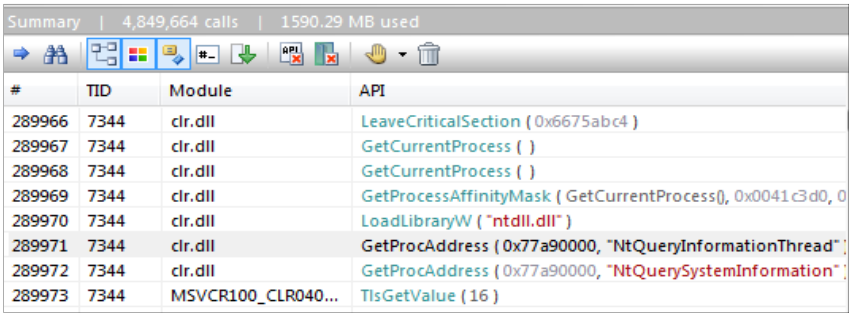

Fig. 4 Redline utility usage of system calls

Redline utility usage of system calls indicates the main reason rootkits hook the NtQuerySystemInformation function, namely, to prevent that information from being retrieved by a rootkit detection utility. This technique is used by many rootkits to thwart detection, and thus ensure a longer life on the target system. Since memory image analysis does not rely on system calls, detection of the hooks made by the rootkit cannot be evaded in this manner and in our experiments were detected using Volatility.

Volatility's method of SSDT hook detection is to scan the Executive Thread Block objects (ETHREAD), which have structures that contain the kernel thread (KTHREAD) block, the thread identification

information, the process identification information, security information (in the form of a pointer to the access token), and impersonation information (Russinovich and Solomon 2009, The Volatility Framework, Light 2011). Accessing ETHREADS through Volatility's threads plugin captured the hook's functions, including the hooked NtQuerySystemInformation function.

Our experiments demonstrate, in practical cases, that system calls cannot be trusted in compromised systems, because these calls may be intercepted by rootkits, and therefore return incomplete, untrustworthy, and even positively misleading results. Our experiments also demonstrate how memory image utilities are inherently more reliable, because they cannot be tricked by kernel rootkits into providing incomplete or false information, generated from an infected system call, and presented to an investigator.

VI. REVIEW OF RELATED RESEARCH

The system service dispatch table hook is used by many security applications, such as Diamond CS Process Guard v2, Kerio Personal Firewall4, and Sebek v2, to apply their security features. Chew discovered this in his research of the hooking techniques applied in those applications (Keong, 2004). Pan Shen introduced the Windows Kernel Hook (WKH) technique to control behaviour of the replication and store, to prevent contents from being re-stored and replicated, by using ZwWriteFile, a function that is responsible for writing tasks. The WKH technique hooks and modifies the corresponding functions to target operations which will lead to a failure in accomplishing the operation. The author used an SSDT hook to demonstrate his technique by hooking the ZwWriteFile function in the SSDT. The WKH was tested on Adobe Reader 9.0 where the "Save" function did not actually save any modifications to the file even though a message was returned indicating that the save was successful. This demonstrated that a hook was in place which prevented the "Save" function from occurring (Wang et al 2009).

Moreover, Zhang and Bi discuss a method of integrity detection and restoration based on the kernel file; they provide a method to detect infection and restore the system after a SSDT hook has been employed. The author's compare the number of the function's addresses in the KiserviceTable's to the number in the KeserviceDescriptorTable's (Yangquan Zhang and Hai Bi 2011). Mahpatre and Selvakumar developed a technique named Kernel Rootkit Trojan Detector (KeRTD) to detect and scan against rootkits that operate in kernel mode-ring 0. It also blocks a rootkit from spreading infection of the target system. KeRTD was set into the kernel mode to "monitor all the processes and drivers being loaded into the system at every moment of their creation and deletion." The Task Manager then returns a list of all processes that are running and drivers that are loaded, which are then examined by KeRTD to detect any processes hidden by rootkits. The technique was able to unhook SSDT functions that were hooked by rooktits (Mahpatre and Selvakumar, 2011).

VII. CONCLUSION AND FUTURE WORK

Rootkits employ increasingly sophisticated methods to avoid traditional detection, due to developers' awareness of the ways in which memory structures can be used to conceal their rootkit's existence within the system. This paper shows the limitations of a particular live response utility, Redline, for detecting SSDT hooks, and explains how rootkits can, and in our experiments did evade live response detection. In light of this, we suggest that live response procedures include memory acquisition. The most valuable method for collecting evidence is a hybrid approach, with current technologies, employing situational recognition and triage techniques. With this approach, live response can function as triage, gathering

enough data to determine the next logical action, rather than being used to amass comprehensive, unmanageable, and unreliable bodies of data. Under many conditions, it is crucial to resolving a case that the examiner gains a more complete awareness of the running state of the machine. Using full memory analysis and the requisite memory acquisitions to support and complement traditional digital forensic investigations is appropriate in such cases.

There is much need for further investigation into the capabilities and limitations of live response and memory image analysis. To note two specific examples, we hope for (1) further inquiry into such capabilities when rootkits target the control registry (CR3, CR2) in the processor. When rootkits target those control registries they gain control over the processor's performance. Another promising area for further research is the analysis of function hooking by much antivirus software, in cases where rootkits are also in place. Finally, we hope for further development of live response utilities that do not rely on system calls.

ACKNOWLEDGMENT

As first author, I am deeply indebted to the Ministry of High Education in Saudi Arabia represented in Saudi Culture bureau in Canada for giving me a scholarship. Special regards and gratitude go to my brother Fahad Abdullah Alzaidi for his advices and discussions. I give special thanks to my family for their support all the time. Also, Thanks go to my advisors Dale Lindskog Pavol Zavarsky, and Ron Ruhl for their continuous support. Also, I would like to acknowledge Michael Hale Ligh for his advices and discussions. I am also indebted to my best friends for their support, Sami Alshaheri, Steven L. McGowan, and Amanda Solyom and others.

REFERENCES

Yangquan Zhang and Hai Bi (2011, July). Anti-rootkit Techonolgy of Kernel Integrity Detection and Restoration [Online]. Available: http://ieeexplore.ieee.org

Yulin Wang, Yang Shen and Jian Pan. (2009, December). Usage Control Based on Windows Kernel Hook [Online]. Available http://ieeexplore.ieee.org

Yunlong Wu, Dong Cui and Qiang Zhang. (2010, July). A Malicious Software Evaluation System Based on Behavior Association [Online]. Available http://ieeexplore.ieee.org

Chew Keong. "Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration", SIG, 2004

D. Kornblum, "Exploiting the Rootkit Paradox with Windows Memory Analysis" (2006).

Bill Blunden, "Windows System Architecture, Hooking Call Tables, and Defeating Live Response" in the rootkit arsenal, Texas, Wordware, 2009, ch 3, 5, 9. pp. 79-499.

GreG Hoglund and James Butler, "the age old art of hooking, subverting the kernel and Rootkit detection" in Rootkits: Subverting the windows kernel. boston ,Personal Education, 2006, ch, 2,4,10. pp, 21-47,71-112,295-31.

Mark E.Russinovich and David Solomon, in Windows Internals, Microsoft Press 2009,.

Jiayuan Zhang, Shufen Liu, Jun Peng, and Aijie Guan, (2009, May). Techniques of user mode detection System Service Description Table [Online]. Available at http://ieeexplore.ieee.org

B. D. Carrier. (2003, December). Hardware – Based Memory Acquisition Procedure for Digital Investigations. [Online]. Available at http://www.digital-evidence.org/papers/tribble-preprint.pdf

Aljaedi, Amer, Lindskog, Dale, Zavarsky, Pavol, Ruhl, Ron, and Almari, Fares (2011, October) Comparitve anaylsis of Volatile Memory forensics: Live respond vs. Memory image. [Online]. Available at http://ieeexplore.ieee.org

Guide to Malware Incident Prevention and Handling NIST 800-83, 2005 MmGetSystemRoutineAddress routine, Internet http://msdn.microsoft.com/en-us/library/windows/hardware/ff554563 (v=vs.85).aspx 2010 [September, 13 2011]

Michael Hale Ligh, Steven Adair Blake Hartstein, and Matthew Richard "Memory Forensics: Rootkits"in Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code, New York Wiley, 2011 ch 17, pp, 636-772 Chris Rie, "Inside Windows Rootkit", VigiantMinds, Pittsburgh, 2006.

The Volatility Framework website (2011) [Online]. Available at http://code.google.com/p/volatility/

Redline website [Online]. Available http://www.mandiant.com/products/free_software/redline/

Memoryze website [Online]. Available at http://www.mandiant.com/products/free_software/memoryze/

Mahpatre and Selvakumar, (2011, July), an online view difference and behaviour based kernel rootkit detection [Online]. Available: ACM digital library.

IDA Pro, [Online] Available http://www.hex-rays.com/products/ida/index.shtml

Runtime2.sys rootkit , [Online] Available http://www.kernelmode.info, [Oct 10, 2011]

Blackenergy2 rootkit , [Online] Available http://www.offensivecomputing.net/ [Oct 10, 2011]

Wincom32 rootkit [Online] Available http://www.kernelmode.info [Oct 10, 2011]

Michael "Light Investigating Windows Threads with Volatility", [Online] Available http://mnin.blogspot.com/2011/04/investigating- windows- threads-with.html APRIL 18, 2011 [Oct 3, 2011]

MOYIX, Auditing the System Call Table Internet: http://moyix.blogspot.com/2008/08/auditing-system-call-table.html AUGUST 20, 2008 [Oct 3, 2011]

Cal Waits, Joseph Ayo Akinyele, Richard Nolan, and Larry Rogers "Computer Forensics: Results of Live Response Inquiry vs. Memory Image Analysis" software engineering intuition, Omaha, 2008.

Michael A. Davis, Sean Bodmer, and Aaron LeMasters "Kernel-Mode Rootkits", in Hacking Exposed Malware & Rootkits, McCraw Hill 2010 Computer Security Incident Handling Guide (800-61), NIST

Cameron H. Malin, Eoghan Casey, and James M. Aquilina "Malware Incident Response: Volatile Data Collection and Examination on a

Live Windows System" in Malware Forensics: Investigating and Analyzing Malicious Code, Syngress 2008 ch 1.

Get system call address from SSDT, (2008) Available at http://www.honeynet.org/node/438.

Microsoft, "NtQuerySystemInformation", [Online] Available http://msdn.microsoft.com/ [December 20, 2011]

APPENDIX

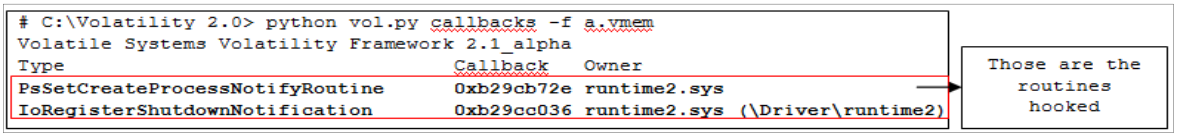

Fig. 5 Callback routines hooked by runtime2.sys

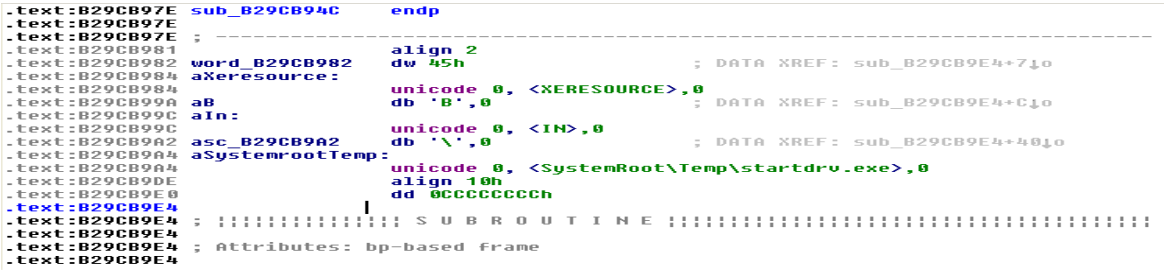

Fig. 6 The process created by runtime2.sy

TABLE 4

THE INFORMATION RETRIEVED THROUGH NTQUERYSYSTEMINFORMATION FUNCTION

Information	Description
SystemBasicInformation	Provide the number of processors running in the system in a SYSTEM_BASIC_INFORMATION structure.
SystemExceptionInformation	Provide an opaque SYSTEM_EXCEPTION_INFORMATION structure that can “be used to generate an unpredictable seed for a random number generator”.
SystemInterruptInformation	Provide an obscure SYSTEM_INTERRUPT_INFORMATION structure that can “be used to generate an unpredictable seed for a random number generator”.
SystemLookasideInformation	Provide an obscure SYSTEM_LOOKASIDE_INFORMATION structure that can “be used to generate an unpredictable seed for a random number generator”.
SystemPerformanceInformation	Provide an opaque SYSTEM_PERFORMANCE_INFORMATION structure which is used to “generate an unpredictable seed for a random number generator”.
SystemProcessInformation	Provide an array of SYSTEM_PROCESS_INFORMATION structures, “one for each process running in the system. These structures contain information about the resource usage of each process, including the number of handles used by the process, the peak page-file usage, and the number of memory pages that the process has allocated”.
SystemProcessorPerformanceInformation	Provide an array of SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION structures, “one for each processor installed in the system”.
SystemQueryPerformanceCounterInformation	“Returns a SYSTEM_QUERY_PERFORMANCE_COUNTER_INFORMATION structure that can be used to determine whether the system requires a kernel transition to retrieve the high-resolution performance counter information through aQueryPerformanceCounter function call”.
SystemRegistryQuotaInformation	“Returns a SYSTEM_REGISTRY_QUOTA_INFORMATION structure”.
SystemTimeOfDayInformation	“Returns an opaque SYSTEM_TIMEOFDAY_INFORMATION structure that can be used to generate an unpredictable seed for a random number generator. Use the CryptGenRandom function instead”

** The content of this table is obtained from Microsoft