International Conference on Computing, Communication System and Informatics Management (ICCCSIM)

Venue : Hotel RAMADA , Bur Dubai, UAE

Date : 29 – 30 July, 2012

Comparative Analysis of Operational Malware Dynamic Link Library (DLL) Injection Live Response vs. Memory Image

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

Master of Information Systems Security Management

Concordia University College of Alberta

Edmonton, Canada

ahmed_alasiri@yahoo.ca, muteb.alzaidi@gmail.com, {pavol.zavarsky, dale.lindskog, ron.ruhl}@concordia.ab.ca, alassmi.shafi@gmail.com,

Abstract :

One advanced tactic used to deliver a malware payload to a target operating system is Dynamic Link Library (DLL) injection, which has the capabilities to bypass many security settings. In cases of compromise involving DLL injection, volatile memory contains critical evidence, as these attacks typically leave no footprint on the hard disk. In this paper, we describe the results of our comparative analysis between a particular live response utility, Redline, and a particular memory image utility, Volatility, in cases where malware is using DLL injection. We show that Redline is significantly limited, by comparison with Volatility, in its ability to collect relevant evidence from memory. Based upon these observations, we draw general conclusions about the advantages of memory image analysis over live response.

Keywords - *DLL; Memory Image; Live Response; DLL Injection; Create Remote Thread*

I. INTRODUCTION

This Dynamic Link Library (DLL) injection is an advanced malware payload delivery technique, used by attackers against a target system, and which has the capability to bypass most security settings. For instance, DLL injection can be employed to exploit a process like Internet Explorer, which can then be used as the process gateway to circumvent a firewall. As a result of DLL injection's security evasion capabilities, many researchers have emphasized the importance of collecting evidence from volatile memory on the victim machine, since there is usually no footprint left on the hard disk after an attack [2]. Volatile memory forensics initiatives have recently gained prominence, as they constitute an effective tool in digital forensics analysis [3][4]. 'Live response' is the term used to refer to the traditional technique for collecting evidence from volatile memory. Utilities used during a live response normally rely on kernel system calls. A system call is a request of a service from an application programming interface (API) to the operating system's kernel However, it is well known that system calls may be intercepted or compromised by malware, which of course will impact the veracity of the data collected by these live response utilities.

Memory image analysis is another technique for collecting evidence from volatile memory. Since live response utilities generally rely on system calls, memory image analysis is perhaps more dependable, because it cannot easily be affected by malware in the kernel. In addition, a memory image may give us more vital evidence, since it directly accesses the memory, rather than relying on the API to the operating system's kernel.

The potential for, and the problems associated with collecting volatile data after a DLL injection attack are explored in this paper, through a comparison of the evidence collecting capabilities of a particular live response utility, Redline, and a particular memory image analysis utility, Volatility, both of which are commonly used in forensic investigations. This paper is organized into six sections. Section II describes how DLLs function. Section III is an overview of DLL injection. Section IV describes our methodology and experimental design, and also information concerning the tools used in the comparative analysis. Section V contains our results and some discussion of them. Section VI is our conclusion and recommendations for future investigations within this area of study.

II. DYNAMIC LINK LIBRARY (DLL) OVERVIEW

A Dynamic Link Library (DLL) is a module that contains functions which can be shared by a number of applications [5]. DLLs are a means “to modularize applications so that their functionality can be updated and reused more easily” [5]. A DLL is loaded once into memory address space, and can be accessed by any running application. When several applications share the same module it reduces memory overhead “because although each application will receive its own copy of the DLL data, the applications share the DLL code”[5]. Kernel32.dll, User32.dll, and GDI32.dll are the important DLLs on the Windows operating system. Kernel32.dll is used to control memory, processes, and threads; User32.dll is used to control the user interface; GDI32.dll is used to draw graphical images and display text [5][7].

The functions that DLLs contain fall into two categories: exported functions and internal functions. Exported functions are functions that can be called by other modules as well as within the DLLs where they are defined; whereas the internal functions are intended to be called only from within the DLLs where they are defined [5].

There are two ways that an executable can dynamically link to a function exported by a DLL:

1) Load-time dynamic linking:

A vital portion of the executable module is the “import section that lists all the DLL module names required by this executable”[7]. Once the DLL and the executable modules are assembled, an application’s operation can commence. Before the executable starts, the operating system loader will perform certain procedures. The loader will develop a virtual address space for the new process and the executable module will be mapped to those new spaces. The executable module’s import section will be parsed by the loader. The loader then pinpoints the DLL module for every DLL name listed in the section and then maps that DLL into the process’ address space.

To simplify this process, the application’s code references the required DLL before it executes. The DLL module can then share the functions and variables from another DLL that helps the executable to be fully initialized on the system. [7][10].

2) Run-time dynamic linking:

At the point that the application is operating, the process may load the necessary DLL explicitly and will then precisely link to the desired exported symbol. In fact, the thread that is contained in the process can decide whether it wants to call a function within a DLL or not. Specifically, the thread can "load the DLL into the process' address space, get the virtual memory address of a function contained within the DLL, and then call the function using this memory address"[7]. This can be accomplished by requesting one of these functions, *LoadLibrary() and LoadLibraryEx():*

```
HMODULE
LoadLibrary(PCTSTR
pszDLLPathName);
HMODULE
LoadLibraryEx(PCTSTR
pszDLLPathName,
HANDLE
hFile,DWORD
dwFlags);
```

The LoadLibrary and LoadLibraryEx functions will assist in locating DLL files on the user's system using a particular search order and map the DLL's file image into the calling process' address space. The virtual memory address where the file image is mapped is identified when the HMODULE value is returned from both functions [7][6][11].

III. DLL INJECTION

Injection involves influencing the application's behavior in memory in a way that the user did not anticipate or intend. According to Skape et al, "DLL Injection is the process by which a dynamically linked library is injected, or forcibly loaded, into a process' address space" and it occurs after a program has been executed [8][32].

There are two modes dynamic DLL injection, and each mode is performed by an attacker as a series of steps:

A. Remote Thread Injection

The mechanism of this mode of attack is to load the malicious DLL through the creation of a thread on the target process, which is then used to call LoadLibrary. In this fashion, LoadLibrary is therefore used to load the malicious DLL. As a result, the attacker must fabricate a new thread in the target's process, as one cannot easily control the threads in a process that one did not initially create [7] .In this way, by generating the thread, the attacker gains control over the process. The CreateRemoteThread function on Windows operating systems can be used to achieve this type of injection. Below is the declaration for the CreateRemoteThread function on Windows [7][12].

```
HANDLE

CreateRemoteThread(
HANDLE
hProcess,
PSECURITY_ATTRIBUTES
psa,DWORD
dwStackSize,
PTHREAD_START_ROUTINE
pfnStartAddr,PVOID
pvParam,
DWORD
fdwCreate,PDWORD
pdwThreadId);
```

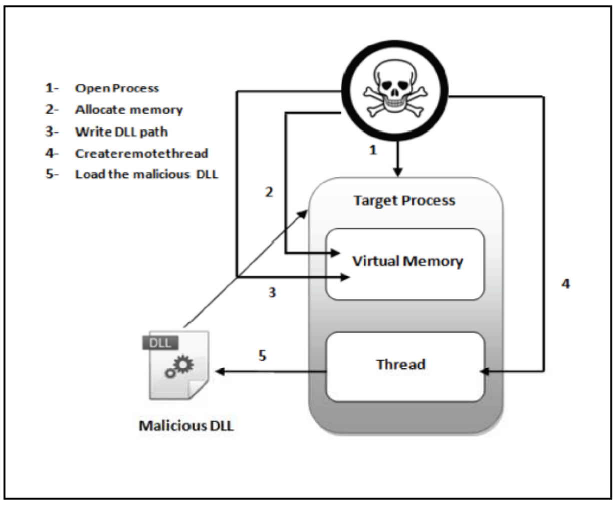


Figure 1. Remote Thread Injection

Fig.1 shows, in simplified form, the method that can be employed by malware to inject the malicious DLL into other processes. First, the malware will open the process using the OpenProcess function, which returns an open handle that is responsible for checking the process privileges; this handle is used to grant the right access to the target process. Secondly, malware will allocate memory using the VirtualAllocEx function in order to specify the correct path for the malicious DLL. Thirdly, it will write the DLL path using the WriteProcessMemory function. Once the path has been created, the malware will initiate the CreateRemoteThread function to create a thread on the target process, instructing the thread to load the malicious DLL remotely. As a result, the malware will have attached the malicious DLLs on the target process, and is able to compromise critical data on the victim's machine. As long as the target process is still running, the attacker will have back-door access to it [9].

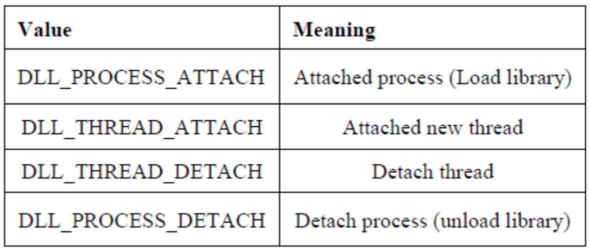
B. Windows Registry DLL Injection:

Most malware in fact uses this method and it can be executed through the registry key AppInit_DLLs . According to Graham et al, “In Windows NT4, 2000, and XP, AppInit_DLLs is a registry key commonly used to inject DLLs into processes”[9]. The AppInit_DLLs key may be given a value corresponding to a single DLL or to a list of DLLs, and it is located in the registry thus [9] :

HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLLs

When a new process has the User32.dll library mapped, it receives a DLL_PROCESS_ATTACH notification [7]. When the notification is processed, the User32.dll will call LoadLibrary for each DLL specified in this key. The entire library is loaded, and the library’s associated DllMain function is called with fdwReason set to DLL_PROCESS_ATTACH to load the library. The fdwReason parameter can be set to one of the values shown in table 1 [7][13].

TABLE I: FDWREASON PARAMETER VALUES



Value	Meaning
DLL_PROCESS_ATTACH	Attached process (Load library)
DLL_THREAD_ATTACH	Attached new thread
DLL_THREAD_DETACH	Detach thread
DLL_PROCESS_DETACH	Detach process (unload library)

To simplify the procedure, malware usually modifies the AppInit_DLLs registry key by injecting the malicious DLLs into its list. user32.dll, which is responsible for the Windows interface, loads DLLs on AppInit_DLLs during DLL_PROCESS_ATTACH. If the injection succeeds, the applications will call user32.dll in order to load the AppInit_DLLs list, which will include the malicious DLLs. However, this type of attack is restricted only to applications that interact with AppInit_DLLs [10][14].

IV. REVIEW OF THE LITERATURE

Skape , J Turkulainen described the DLL injection method on two different operating systems, namely Linux and Windows, including such details as On-Disk Library Injection and In-Memory Library Injection, which are two ways of injecting the library remotely. On-Disk Library Injection, as could be inferred from its title, indicates “the library is written to disk and then loaded into the processes address space” [8]. In contrast, In Memory Library Injection loads the library into a running process in memory without writing it to the disk. Skape et al concluded that DLL injection makes it possible for malware developers to write extremely advanced worms and viruses that are capable of executing their payload under the radar of present day virus scanners [8].

S. Daly discussed the inability of current countermeasures to detect or prevent DLL injection. He demonstrated a method by which attackers can create malware which is difficult to detect, even by the latest antivirus products, thereby allowing data to be leaked while bypassing firewalls. Daly also examined the effectiveness of modern anti-virus products such as Viper, Comodo and Kaspersky to detect DLL injection techniques. The findings of this noteworthy research can be used by anti-virus developers in order to enhance their applications' ability to deal with DLL injection [1].

C. Waits et. al., in a paper entitled "Computer Forensics: Results of Live response Inquiry vs. Memory Image Analysis" compared these two forensics techniques by comparing the evidence collected with various 'live response' tools, such as pslist, ListDLLs, FPort, PTFinder, with the evidence collected using the memory image analysis tool, Volatility. The paper illustrates the benefits and drawbacks of both techniques, but concludes that memory image analysis is generally speaking more useful [3].

Work carried out by A. Aljaedi et al. shows that one of the drawbacks of live response is the overwriting of critical evidence. Additionally, this research shows that memory image analysis can be leveraged as an alternative in mitigating the risk of losing volatile evidence such as terminated and cashed processes, which are generally missed during the live response. He conducted several experiments to emphasize the importance of using a memory image instead of the actual RAM. This research has also demonstrated that dumping the memory image using advanced tools can extract critical data such as passwords and credit card details even though they are encrypted on disk [4].

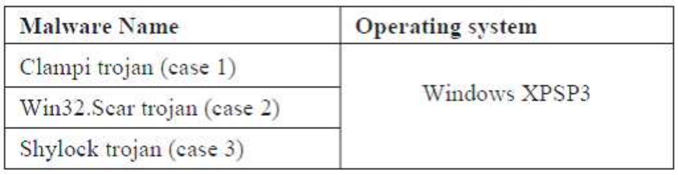
V. METHODOLOGY

Our experiments relied upon five machines; the host machine was running on a Windows 7 Professional platform with 4 GB RAM, equipped with Intel (R) Core (TM) 2 CPU T7250 2.00GHz and hosting four other virtual machines; three virtual machines were running Windows XPSP3 targeted by DLL injections, while the fourth machine, running Windows XP SP3, was the machine on which we investigated the memory image.

Our experimentation involved three cases, corresponding to the three randomly chosen DLL injection exploits. In each case, we launched one of the malware against the victim machines running Windows XPSP3. At this point, the memory image was taken from the victim machine via a virtual machine snapshot. The memory image was investigated using Volatility, including the use of several plug-ins helpful when investigating DLL injection. At the same time, the live response investigation was conducted on the victim machines using Redline to observe malicious DLLs and score the "riskiness of DLLs based on how many process load them"[20]. The results of these two investigations (live response using Redline and memory image analysis using Volatility) were then compared in terms of the evidence gained from both techniques. The investigation process for both techniques was repeated three times in order to observe any dynamic change that would occur, as well as increase the reliability of the analysis. Table 2 below illustrates that our research was conducted as three separate cases.

As noted, the experimental methods were identical in each case, with the exception of the specific malware samples.

TABLE 2: THREE CASES IN THE RESEARCH EXPERIMENT



Malware Name	Operating system
Clampi trojan (case 1)	Windows XPSP3
Win32.Scar trojan (case 2)	
Shylock trojan (case 3)	

A. Memory Image Analysis

Until very recently, forensic investigation of raw memory consisted of little more than string searches on a memory dump. Investigators now have much more powerful tools and methods for the investigation of memory, including tools and methods for malware reverse-engineering tasks and malware detection. In our experiments we relied on Volatility Framework and Interactive Disassembler Professional [9] for these purposes. Volatility Framework is an open collection of tools, and supports in-depth investigation of DLL injection using various plug-ins, such as imageinfo, malfind, psscan, dlllist, procmemdump, ldrmodules and Vadinfo. There are many alternate plug-ins that can further assist investigation and analysis [16]. Interactive Disassembler Professional (IDA PRO) is a dissembler and debugger used to analyze malware code [17]. It is the most commonly used software to disassemble binary code, in order to extract assembly instructions from machine level language [19].

B. Live response Analysis

Live incident response entails gathering forensic evidence from a machine while it is still operating. Traditionally, this is the first (and sometimes the lone) step in a forensic investigation. Though it is capable of returning vital data, live response is imperfect, since the forensic investigator must rely on the execution environment of the system being investigated. In a typical live incident response, the investigator will introduce into the infected computer a trusted set of volatile data collection utilities, and will direct the output from their execution to an external USB or network drive, or live-stream the data over an encrypted network channel. No matter how the live incident response is carried out, the responder necessarily relies on the suspect environment. Redline is an example of a tool that can be employed in live response. It is designed to detect malware generally, and is capable of investigating DLL injection specifically. Redline also rates every running process on a system according to its perceived level of risk [15][20].

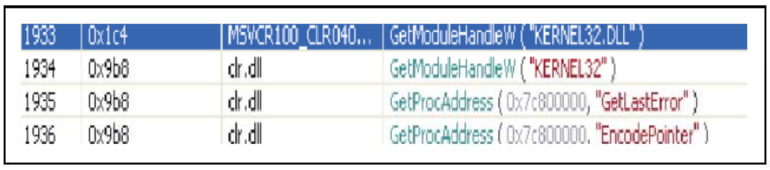
VI. DISCUSSION AND RESULTS

This section discusses the results of our experiments; the following observations are noteworthy. First, in all three cases DLL injection was detected by both Redline and Volatility. However, there were number of injection processes involved in the cases, but unrelated to the malware, due to the fact that injection techniques can be used by any application. Some applications make legitimate use of DLL injection; for example, VMware Workstation uses this technique to allow copy and paste features between the host and guest machines. Second, three cases (clampi, Win32.Scar, shylook) injected the malicious DLLs into particular processes such as Internet

Explorer and Explorer.exe during run time dynamic linking. Additionally, these three trojans waited until the injection process succeeded, and then terminated themselves in order to hide their activities. Redline was not able to detect terminated processes, whereas this information was available by memory image analysis using Volatility. This difference is explained by the fact that information about terminated processes is not mapped to the kernel mode, but rather, is found in the unallocated data in memory (RAM). Since Redline relies on system calls to interact with the kernel as a means to gain access to memory, it has no method by which to access this data, whereas a memory image tool like Volatility can bypass this and directly access this unallocated data.

Detecting terminated processes can simplify the investigation by providing valuable information such as the target process, name of the malware on the victim machine, the registry key used to maintain itself, and the point of origin on the system. It must be acknowledged, however, that in our experiments memory image analysis was unable to provide information about these terminated process in the second and third images, acquired further subsequent to exploitation, and this underscores the fact that a live response can overwrite critical data, as demonstrated e.g. by A. Aljaedi et al.

Third, the list of loaded DLLs for each process was obtained during both live response and memory image analysis; however, the hidden/unlinked DLLs were not obtainable during the live response. Fig. 2 shows the functions which were requested by Redline, in order to show the listed DLLs. Figure 2.



```
1933  0x1c4  MSVCR100_CLR040...  GetModuleHandleW ( "KERNEL32.DLL" )
1934  0x9b8  clr.dll             GetModuleHandleW ( "KERNEL32" )
1935  0x9b8  clr.dll             GetProcAddress ( 0x7c800000, "GetLastError" )
1936  0x9b8  clr.dll             GetProcAddress ( 0x7c800000. "EncodePointer" )
```

Figure 2. functions requested by the live response utility

Two of the trojans (Clampi, Win32.Scar) used in our experiment hid their malicious DLLs by remotely loading them, without calling the LoadLibrary or LoadLibraryEx functions on the host system. This is known as reflective injection which means “the reflective loader does not register the loaded DLL within the process list of loaded modules”[9]. Consequently, no entry was created in the Process Environment Block (PEB) that used by API calls to retrieve this information of the target process. Since no entry was created in PEB, Redline was unable to detect the hidden /unlinked DLLs.

The results were quite different when using Volatility’s virtual address descriptor (vadinfo) plug-in, which successfully tracked these DLLs. A Virtual Address Descriptor (VAD) shows the start and end address for each process, along with the corresponding DLL file. The VAD is “used by the Windows memory manager to describe memory ranges used by a process as they are allocated” [23]. When the process allocates virtual memory space using the VirutalAlloc function, the VAD creates entry points for each DLL loaded by the process, as illustrated below in Fig. 3 [23].

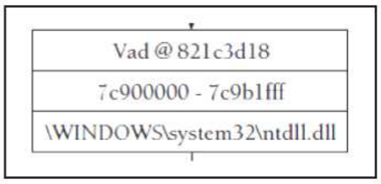
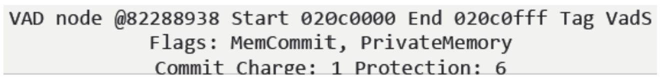


Figure. 3 Example of VAD entry with corresponding file [14]

The first row represents the address of the VAD entry in kernel memory, while the second row is arepresentation of the virtual addresses in the process's memory space [14]. Finally, the third row represents the name of a memory-mapped file (ntdll.dll). This information is only available if the tag is type "Vad" or "Vadl" [14]. The DLLs can be found in the virtual memory of the host process even though they do not exist in the PEB.

We used Volatility's VAD plug-in to access the EPROCESS structure that contains kernel mode information for each running process. Memory Manager Virtual Address Descriptors (MMVAD) are a significant part of the EPROCESS and hold information about the virtual start and end address and mapped DLL. [31]

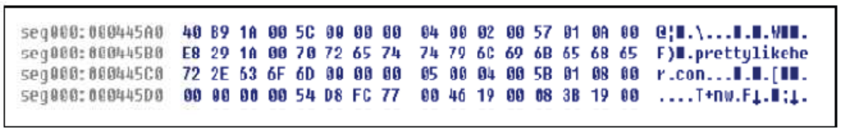


```
VAD node @82288938 Start 020c0000 End 020c0fff Tag VadS
          Flags: MemCommit, PrivateMemory
          Commit Charge: 1 Protection: 6
```

Figure 4. Suspicious VAD entry

Fig. 4. is an example of a suspicious VAD entry that we found in our investigation and, although it does not have a corresponding file mapped to it, the protection nonetheless indicates that there was execution (MM_EXECUTE_READWRITE) on the target process. The vaddump command is able to reconstruct the VAD entry and dump it to disk for analysis[16].

Our successful identification of hidden / unlinked DLLs that were used to inject processes further guided our investigation of the memory image. First, we were able to extract significant information about the malware and what actions were performed on the victim machine. Second, we were able to discover methods used by the malware to evade firewalls. For example, the Calmpi trojan used Internet Explorer, and the Shylock trojan used Explore.exe. Finally, we discovered the Registry key value that was created by the all malware in order to position itself on a system and ensure its execution. We also noticed that two trojans (Clampi, Win32.Scar) communicate with their own malicious server. For example, the Win32.Scar trojan established a session with a server named prettylikeher.com, in order to upload information from the victim host, as shown in Fig 5.



```
seg000:000445A0  40 B9 1A 00 5C 00 00 00  04 00 02 00 57 01 0A 00  @¦■.\...■.■.W■■.
seg000:000445B0  E8 29 1A 00 70 72 65 74  74 79 6C 69 6B 65 68 65  F)■.prettylikehe
seg000:000445C0  72 2E 63 6F 6D 00 00 00  05 00 04 00 5B 01 08 00  r.con...■.■.[■■.
seg000:000445D0  00 00 00 00 54 D8 FC 77  00 46 19 00 08 3B 19 00  ....T+nw.F↓.■;↓.
```

Figure 5. The Win32.Scar trojan communicates with prettylikeher.com server

TABLE 3:THE FINAL RESULT OF COMPARING THE LIVE RESPONSE UTILITIES WITH MEMORY IMAGE ANALYSIS

	Case 1		Case 2		Case 3	
	R	V	R	V	R	V
Processes List	✓	✓	✓	✓	✓	✓
Terminated Processes	x	✓	x	✓	x	✓
DLL List	✓	✓	✓	✓	✓	✓
DLL Injection	✓	✓	✓	✓	✓	✓
Hide / Unlinked DLL	x	✓	x	✓	x	✓
Registry Key	x	✓	x	✓	x	✓
Server Communication	x	✓	x	✓	x	✓
Processes Dump	x	✓	x	✓	x	✓
DLL Dump	x	✓	x	✓	x	✓

R= Redline
V= Volatility

Table 3 depicts in brief the data we observed and investigated, and whether that data was discovered during our live response using Redline, or during our memory image analysis using Volatility, or both. A check symbol P□' on the table represents successful detection, whereas an 'x' symbol represents the failure. The graph also therefore depicts concisely the final results of our comparisons between Redline and Volatility. It is clear that the live response utility, Redline, is less effective than the memory image analysis utility, Volatility. As noted above, these results are substantially explained by the fact that malware can be, and in our cases often were designed cleverly enough to not load the DLL via the LoadLibrary or LoadLibraryEx function, and hence hide the malicious DLL from detection methods that rely on system calls.

VII. CONCLUSION

It is a continuous struggle to protect our systems and networks from malware, and researchers must persist in uncovering new and enhancing existing methods of gathering evidence. An important component of this is the examination of volatile memory.

A partial answer to the problems faced by traditional live response forensics, when presented with DLL injection attacks but also generally, is to ensure that procedures are in place for the timely and forensically sound acquisition of memory images of victim hosts. In considering both the role that DLL injection plays in deceiving the system operator, as well as the method that DLL injection uses to carry out its attack, it is necessary to consider combined approaches to properly analyze the nature and scope of the attack. In this way, live response can be considered an initial step towards diagnosing the range of the attack and to help investigators to determine further courses of action. In situations where live response is unable to resolve the conflict, a more complete analysis of the machine's operating state must be taken. Therefore, memory image analysis has to be performed as well. By performing these two approaches in concert, the digital forensic examination will be more reliable.

ACKNOWLEDGMENT

I am heartily thankful to my Country Saudi Arabia for offering my this opportunity, and my supervisor, Pavol Zavarsky, Dale Lindskog, Ron Ruhl, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject. I also wanted to thank my family who inspired, encouraged and fully supported me for every trial that comes in my way, and In giving me not just financial, but morally and spiritually. Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project, Sami Alshaheri, Steve McGowan, Eyad Bogari, Scott Mcintyre, and Des Fernando.

REFERENCES

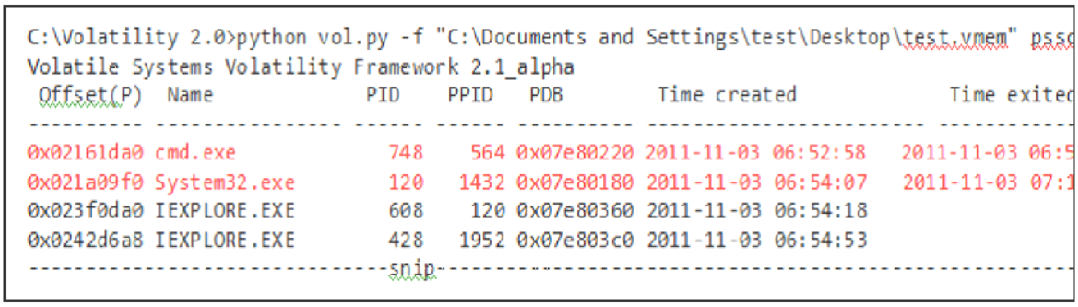
1. Scott Daly, "Preventing Malicious Dll Library Injection," M.S. thesis, Dept. Comput and Eng Systems., Abertay Univ., Dundee, UK, 2011.
2. Brian D. Carrier, Joe Grand (2004, March). Hardware – Based Memory Acquisition Procedure for Digital Investigations. [Online]. Available:http://www.digital-evidence.org/papers/tribble-preprint.pdf
3. Cal Waits, Joseph Ayo Akinyele , Richard Nolan, Larry Rogers (2008): [Online]: ftp://ftp.sei.cmu.edu/pub/documents/08.reports/08tn017.pdf
4. Amer Aljaedi , Dale Lindskog, Pavol Zavarsky, Ron Ruhl, Fares Almari ,"Comparative Analysis of Volatile Memory Forensics" IEEE International Conference on Privacy, Security, Risk and Trust and IEEE International Conference on Social Computing, Boston, USA , pp 1253-1258 ,Oct. 2011.
5. (2011) Windows Dynamic-Link Libraries [Online]: http://msdn.microsoft.com/en us/library/windows/desktop/ms682589(v=vs.85).aspx
6. (2011) The Dynamic-Link Library Search Order [Online]: http://msdn.microsoft.com/enus/ library/windows/desktop/ms682586(v=vs.85).aspx
7. Jeffrey Richter, Christophe Nasarre "DLL Advanced Techniques" , "Windows via C/C++ (softcover)", Fifth Edition, Microsoft Press,2011, ch 20 , pp 553-595.
8. Skape, Jarkko Turkulainen (2004) Remote Library Injection [Online]. Available: http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf
9. James Graham , Richard Howard, Ryan Olson (2011) "DLL Injection", "Cyber Security Essentials", CRC Press, 2011 , ch 4, pp 253- 259.
10. (2011) Using Load-Time Dynamic Linking (2011), [Online].: http://msdn.microsoft.com/enus/library/ms684184(v=VS.85).aspx
11. (2011) Using Run-Time Dynamic Linking , [Online]. http://msdn.microsoft.com/enus/

library/windows/desktop/ms686944(v=vs.85).aspx

12. (2011) CreateRemoteThread function, [Online]: http://msdn.microsoft.com/enus/library/windows/desktop/ms682437(v=vs.85).aspx
13. 2011) DllMain entry point [Online]. http://msdn.microsoft.com/enus/library/windows/desktop/ms682583(v=vs.85).aspx Hale Ligh, Adair,
14. Michael Hale Ligh, Steven Adair, Blake Hartstein , Matthew Richard “Working with DLL” “Malware Analyst’s Cookbook and DVD: Tools and Techniques for Fighting Malicious Code”, Wiley Publishing, Inc 2011,ch 13, pp 487- 510.
15. Bill Blunden ”Hooking Call Table”, “The Rootkit Arsenal”, Wordware Publishing, Inc, 2009, ch 5, pp 246 . 255.
16. (2006) Volatility [Online]: https://www.volatilesystems.com/default/volatility#overview
17. (2011) IDA Pro, [Online] http://www.hex-rays.com/products/ida/index.shtml
18. Ulrich Bayer, Andreas Moser, Christopher Kruegel , Engin Kirda(2006) [Online]. Available: Journal in Computer Virology
19. Abhishek Singh, Baibhav Singh “Assembly Language” ,”Identifying Malicious Reverse Engineering Code”, (2009), Springer,2009, ch 1 , pp 1-28.
20. Redline Mandiant [Online]: http://www.mandiant.com/products/free_software/redline/
21. Nicolaou George, (2009) Win Vista DLL Injection (32bit) ,[Online]. Available: http://www.insecure.in/papers/vista_dll_injection.pdf
22. Mark E Russinovich, David A. Solomon, Alex Ionescu “Processes, Threads, and Jobs” ,”Windows Internals”, 5th Edition Microsoft Press, 2009, ch 5, pp 320- 419.
23. Brendan Dolan Gavitt , “The VAD tree: A process-eye view of physical memory”, DFRWS, US , pp s62- s64, 2007.
24. VirScan.org [Online]: http://r.virscan.org/bb9f65800c81c2c3c832ace29a966715
25. Clampi trojan [Online]. http://www.kernelmode.info
26. Win32.Scars trojan [Online]: http://contagiodump.blogspot.com
27. Shylock trojan [Online]: http://contagiodump.blogspot.com
28. StraceNT - A System Call Tracer for Windows [Online]. at http://www.intellectualheaven.com/default.asp?BH=projects&H=strace.htm
29. (2008) The WIN32 Memory Model, [Online]: http://grayscaleresearch.org/new/pdfs/The%20WIN32%20Memory%20Model.pdf
30. (2008) Reconstructing the Scene of the Crime, [Online]: http://www.blackhat.com/presentations/bh-usa-09/SILBERMAN/BHUSA09-Silberman-MetasploitAutopsy-PAPER.pdf
31. Alex Ionescu ,“Processes, Threads, Fibers and Jobs” (2004), [Online]: http://www.alexionescu.com/part1.pdf
32. James Shewmaker , “Analyzing DLL Injection” (2006), [Online]:

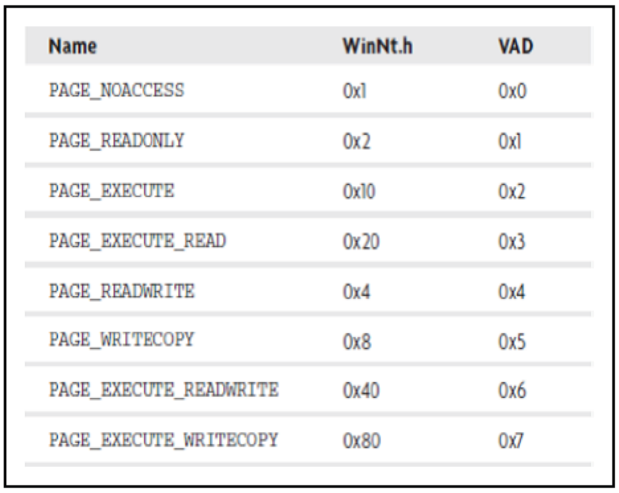
http://www.scribd.com/rahul_agarwal_42/d/75989904-Analyzing-DLL-Injection-by-James-Shewmaker- 2006

APPENDIX



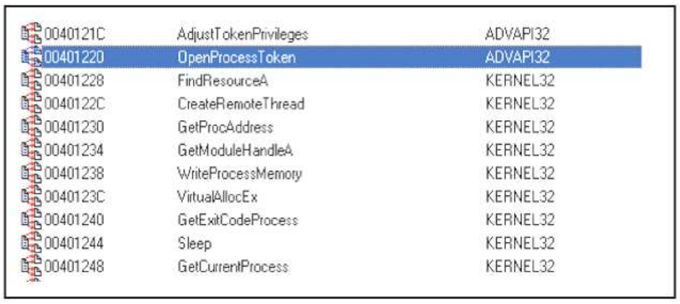
```
C:\Volatility 2.0>python vol.py -f "C:\Documents and Settings\test\Desktop\test.vmem" pssc
Volatile Systems Volatility Framework 2.1_alpha
 Offset(P)  Name                PID    PPID   PDB        Time created             Time exited
---------- ---------------- ------ ------ ---------- ------------------------ -----------
0x02161da0 cmd.exe             748    564 0x07e80220 2011-11-03 06:52:58      2011-11-03 06:5
0x021a09f0 System32.exe        120   1432 0x07e80180 2011-11-03 06:54:07      2011-11-03 07:1
0x023f0da0 IEXPLORE.EXE        608    120 0x07e80360 2011-11-03 06:54:18
0x0242d6a8 IEXPLORE.EXE        428   1952 0x07e803c0 2011-11-03 06:54:53
----------------------------snip--------------------------------------------------------------
```

Figure 6: terminated Process on the memory image



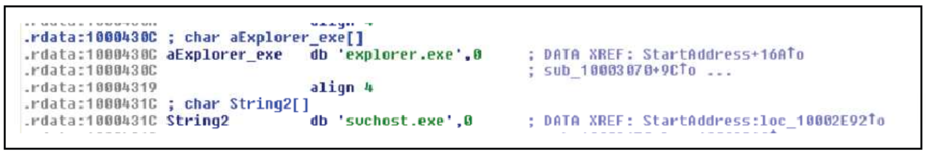
Name	WinNt.h	VAD
PAGE_NOACCESS	0x1	0x0
PAGE_READONLY	0x2	0x1
PAGE_EXECUTE	0x10	0x2
PAGE_EXECUTE_READ	0x20	0x3
PAGE_READWRITE	0x4	0x4
PAGE_WRITECOPY	0x8	0x5
PAGE_EXECUTE_READWRITE	0x40	0x6
PAGE_EXECUTE_WRITECOPY	0x80	0x7

Figure 7: Sample of Page Protection Translations [14]



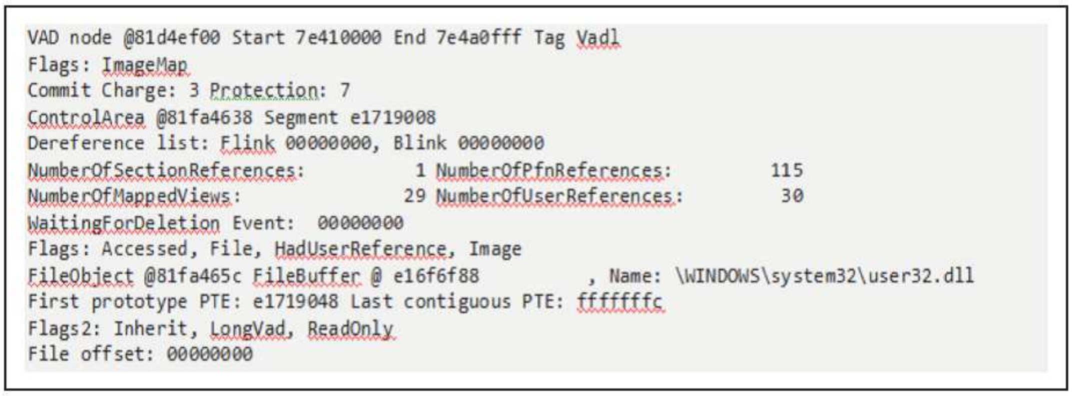
```
0040121C  AdjustTokenPrivileges  ADVAPI32
00401220  OpenProcessToken       ADVAPI32
00401228  FindResourceA          KERNEL32
0040122C  CreateRemoteThread     KERNEL32
00401230  GetProcAddress         KERNEL32
00401234  GetModuleHandleA       KERNEL32
00401238  WriteProcessMemory     KERNEL32
0040123C  VirtualAllocEx         KERNEL32
00401240  GetExitCodeProcess     KERNEL32
00401244  Sleep                  KERNEL32
00401248  GetCurrentProcess      KERNEL32
```

Figure 8: Dumping Suspicious DLL is used to inject the Internet Explorer which used the create remote thread injection method



```
.rdata:1000430C ; char aExplorer_exe[]
.rdata:1000430C aExplorer_exe   db 'explorer.exe',0      ; DATA XREF: StartAddress+16A↑o
.rdata:1000430C                                          ; sub_10003070+9C↑o ...
.rdata:10004319                 align 4
.rdata:1000431C ; char String2[]
.rdata:1000431C String2         db 'svchost.exe',0       ; DATA XREF: StartAddress:loc_10002E92↑o
```

Figure 9: Two processes is implied by (Shylock trojan) to ensure the existence and remain hidden



```
VAD node @81d4ef00 Start 7e410000 End 7e4a0fff Tag Vadl
Flags: ImageMap
Commit Charge: 3 Protection: 7
ControlArea @81fa4638 Segment e1719008
Dereference list: Flink 00000000, Blink 00000000
NumberOfSectionReferences:          1 NumberOfPfnReferences:         115
NumberOfMappedViews:               29 NumberOfUserReferences:         30
WaitingForDeletion Event:  00000000
Flags: Accessed, File, HadUserReference, Image
FileObject @81fa465c FileBuffer @ e16f6f88          , Name: \WINDOWS\system32\user32.dll
First prototype PTE: e1719048 Last contiguous PTE: fffffffc
Flags2: Inherit, LongVad, ReadOnly
File offset: 00000000
```

Figure 10: Trusted VAD entry

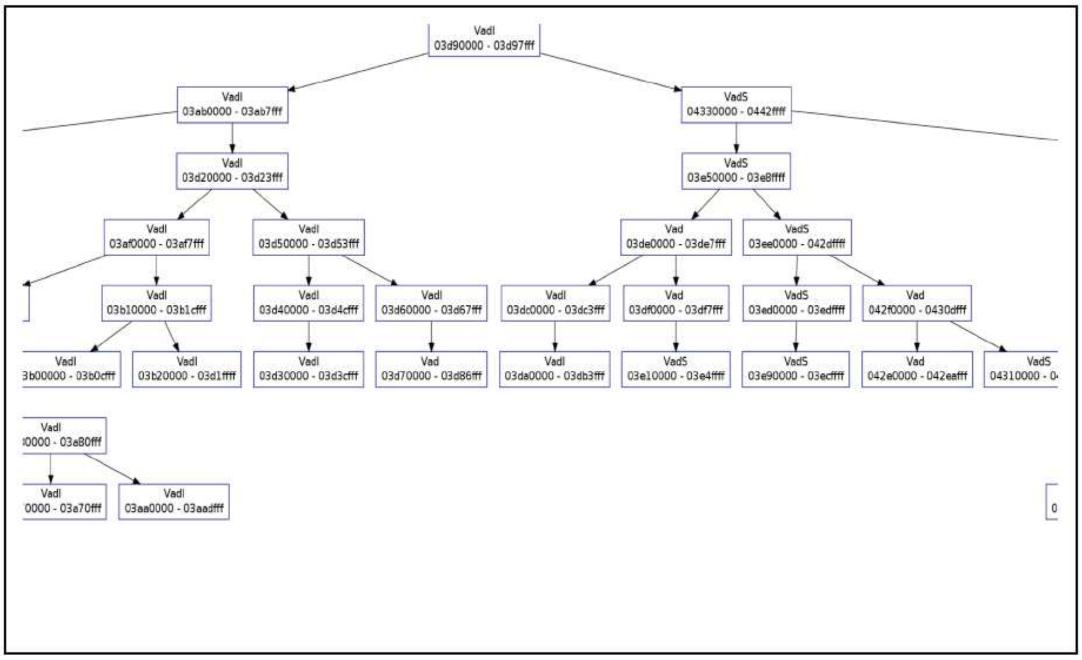


Figure11: Partial structure of VAD tree for the Internet Explorer which is available on memory image utility