Global Information Assurance Certification Paper

Using Windows Crash Dumps for Remote Incident Identification

GIAC (GCIH) Gold Certification

Author: Zong Fu Chua, zfchua@gmail.com
Advisor: Rob VandenBrink
Accepted: 05 June 2015

Abstract

With the proliferation of defense mechanisms built into Windows Operating System,, such as ASLR, DEP, and SEHOP, it is getting more difficult for malware to successfully exploit it. The Microsoft Enhanced Mitigation Exploitation Toolkit further increases the difficulty. A common symptom of a failed exploit is a crash. A crash dump is generated whenever this happensa crash happens. The Windows operating system has a built--in error reporting mechanism to troubleshoot such instabilities. It is possible for an enterprise running Windows--based servers to leverage on this mechanism to collect the volatile memory of client machines for offsite investigation. This would allow system administrators to remotely determine if the crash is due to a badly programmed application (event), or a real malware exploitation attempt (incident). This is advantageous to an enterpriseenterprise, as an incident handling team need not be dispatched on-site to perform incident identification. This paper will provide detailed steps on how to configure the enterprise network to facilitate such an analysis. In addition, a python memory analysis script is included,, which when run against the collected memory, would indicate the percentage probability that a machine is infected with malware.

1. Introduction

An efficient incident response procedure is important for enterprises as it makes the difference between catastrophic losses of data versus a mild disruption of business. This is evident in the numerous prolific and impactful cyber-attacks on big enterprises such as eBay, Target and Sony in 2014.

An important lesson learnt from such attacks is the need for defense in depth. With adequate system hardening and monitoring, it is possible to detect and stop cyber-attacks before real damages are done. With reference to the "cyber kill chain" (refer to Figure 1) coined by Lockheed Martin, cyber defenders should aim to stop an attack in its early stages so as to minimize loss.

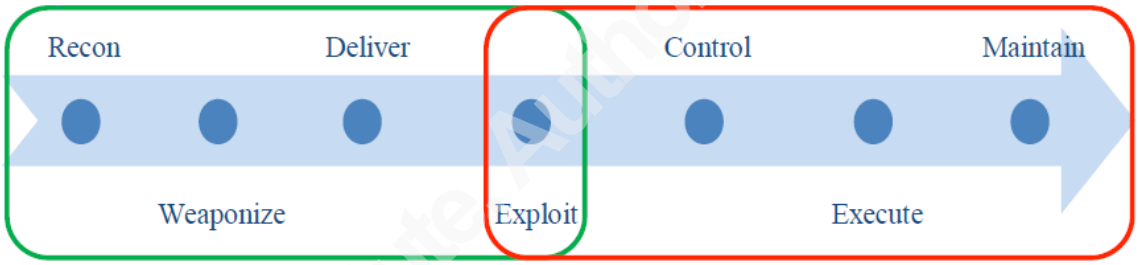


Figure 1 Lockheed Martin Cyber Kill Chain (Eric Hutchins 2011)

For a cyber-attack to succeed, adequate preparation needs to be done. It should be noted that the cyber kill chain is an iterative process whereby the attacker would go back to the drawing board with every failed exploit. This is evident in Stuxnet, whereby the first identified version – Stuxnet 0.5 was released in November 2007 while the real attack took place in 2010. (McDonald, Murchu et al. 2013)

Even though Stuxnet did not cause any crashes, it would have been discovered if adequate monitoring mechanisms were put in place. For example, the additional spawning of the "lsass.exe[1]" process would have been a red flag, since a legitimate Windows should have only one such process. It is proposed that system administrators retrieve the memory of systems exhibiting any such anomalies for further investigation.

[1] lsass.exe is a Windows security process which manages and enforces the security policy.

Traditionally, this is prohibitive, as memory collection would require manpower to be dispatched on site to collect the memory. Moreover, traditional memory collection using software tools,, such as FTK Imager, Mandiant Memoryze or F-Response, on a live system runs the risk of "smearing",, since the system is still running and changing its memory content. One known way of preventing memory collection smear is to extract the virtual memory file (vmem or vmsn) from the snapshot of a virtualized client. However, this requires an enterprise to be running on virtualized infrastructure, which may not be possible due to cost or operational issues. An alternative method would be to use the native Windows's Error Reporting mechanism as a reliable remote memory collection mechanism, which would require minimal changes to the enterprise architecture. However, this only works for Windows based infrastructure. Onsite collection may still be required for other network using Operating Systems (OS) such as Unix/Linux.

Apart from an administratively triggered crash to collect memory, OS crashes are often an indication of underlying problems. Common causes are impending hardware failure, driver incompatibilities, programming error, or malware infection. With the increasing defense mechanisms built into the Windows operating system, it is very likely for exploits to fail and crash the system instead. Such crashes offer valuable insights into attacks and should be tapped as an integrated part of a defense in depth framework.

2. Windows Operating System Defenses

As shown in Figure 2, Microsoft Windows Operating System continues to retain its leadership at 90% of worldwide desktop use (NetMarketShare 2014).

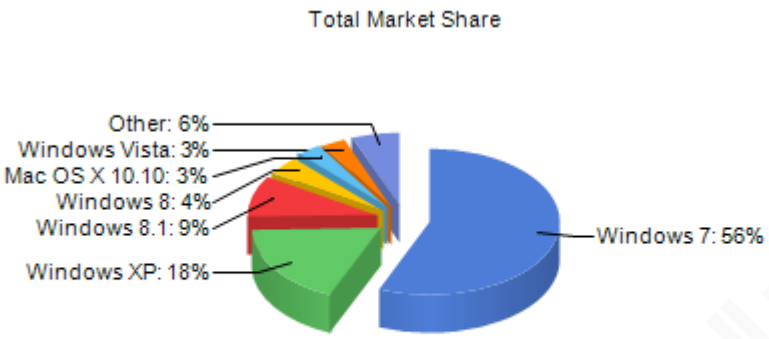


Figure 2 Desktop Operating System Market Share

Over the years, from Windows XP to Windows 8.1, Microsoft has continuously improved its flagship operating system by adopting kernel level malware defenses to better protect itself. These techniques include Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) and Structured Exception Handling Overwrite Protection (SEHOP), which both separately and in combination significantly reduces the chance of successful exploitations.

DEP is a defense mechanism that has been implemented across many platforms. The aim is to prevent an attacker from exploiting vulnerabilities in programs to execute arbitrary code. It is also known as W^X, indicating that a software memory region is either writable or executable but not both. However, though DEP increases the difficulty of a successful exploit, it can be bypassed using a technique known as return-oriented programming (ROP). ROP exploits uses libraries already present in the OS to build malicious instruction sets through shrewd manipulation of its offset. These libraries are often static or have predictable instruction addresses. As such, a countermeasure to thwart ROP attacks is to randomize libraries. This is also known as Address Space Layout Randomization (ASLR),, which is often used in tandem with DEP.

As mentioned earlier, ASLR is a technique to randomize program and libraries addresses at startup. This introduces a certain degree of uncertainty, which significantly decreases the chance of a successful exploit when used with DEP. When an exploit attempt fails, there is a high likelihood that it will create instability in the system, resulting in an application or kernel crash.

Author Name, email@address

Even with DEP and ASLR, malware writers have found ways to succeed. In earlier implementation of ASLR, such as in Windows Vista, there is limited entropy in the randomized address (Whitehouse 2007). Using this increased predictability, malware are able to use brute-force techniques to a valid vulnerable address space. Other advanced exploit techniques includes exploiting information leakage vulnerabilities to disclose critical memory content, revealing vulnerable libraries and addresses. Examples of such bypasses are documented in CVE-2013-1690, CVE-2013-0640 (Chen 2013).

SEHOP is introduced in Windows Server 2008 and Windows Vista SP1 (Microsoft Secure Windows Initiative Team 2009). It aims to mitigate stack overflows attacks where the goal is to redirect the exception--handling pointer to execute arbitrary code. When an exception is triggered, this code is likely to execute with administrative privileges. SEHOP mitigates such exploits by inserting a special marker at the end of the exception handling link list. Before any exception routine is executed, the integrity of the link list is verified by ensuring that a the inserted special marker has not been overwritten. This is similar to the implementation of canary, which is used to mitigate buffer overflow attacks. Once a corruption is detected, the affected process is terminated and a crash dump is generated.

The above kernel defense mechanisms are implemented by default in Windows 8. As a retrospective means to cover the older versions of Windows such as Windows 7, Vista and XP[2], Microsoft launched the Enhanced Mitigation Experience Toolkit (EMET). EMET is a free utility that helps to enforce DEP, SEHOP, ASLR and other newer Windows OS defense mechanismss on 3rd party applications. With the latest EMET 5.2 release, additional defense mechanismss, such as Control Flow Guard, Attack Surface Reduction, Export Address Table Filtering and Stack Pivot Checks are added (Microsoft 2015).

Although these mitigation technologies do not guarantee that vulnerabilities cannot be exploited, it makes it significantly harder for them to succeed. The intent of this paper is to leverage the analysis of crash dumps generated in the event of a failed exploitation attempt as a means of detecting them. In addition, mechanisms are put in

[2] Windows XP has reached End-Of-Life on 8 April 2014 and will no longer be patched or updated by Microsoft Corporation.

place to allow manual triggering of a a crash dump as a means to collect memory from suspected victim machines for further analysis.

2.1. Windows Crashes

Windows Error Reporting (WER) is a crash information submission tool to send crash information back to Microsoft for analysis (Russinovich, Solomon et al. 2012). WER is based on Dr. Watson, a windows built-in application debugger, which has been removed since Windows Vista and Server 2008. The key difference between Dr. Watson and WER is that Dr. Watson does not have the capability to send a memory dump over the network.

There are numerous eventss thatthat could cause Windows or a Windows application to crash. These events range from application fault to kernel fault such as page fault, memory management issues or access violations. As shown in figure 3, the categories of crashes for Windows 7 in May 2012 are as shown (Russinovich, Solomon et al. 2012).

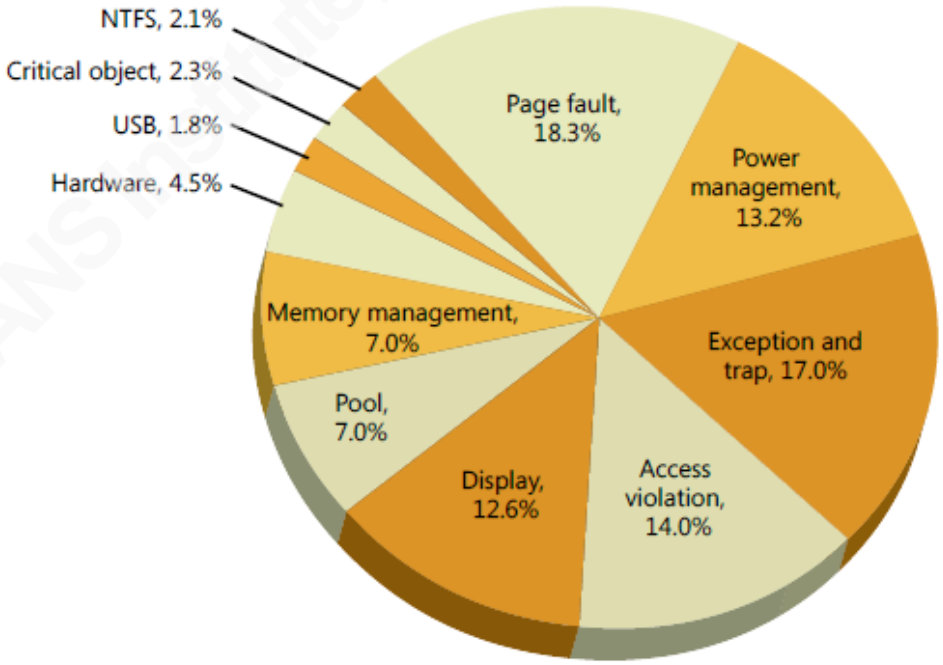


Figure 3 Distribution of Error Categories for Windows 7 and Windows 7 SP1 in May 2012

3. Proposed Solution

3.1. Overview

It is proposed that changes be made to a Windows domain controlled enterprise network via the Group Policy. By editing the Group Policy Object to modify a client's registry settings, full memory captures could be retrieved from clients during a crash. Ad Hoc crashes can then be triggered via the SysInternals - NotMyFault utility. During the crash, the memory dump would be saved locally. After the crash reboot, the crash dump would be moved to a centralized server for archival and further analysis.

It is noted that Microsoft has a paid application called the Microsoft Operations Manager 2007,, which allows the setup of a local enterprise WER Server. This would allow crashes to be automatically forwarded via the network to this local server instead of to Microsoft. However, as this utility is unavailable for this Proof of Concept (POC), WER is disabled.

Conventionally, it is recommended that Windows memory dumps are analyzed using the Windows debugger, as shown in Figure 4 (Russinovich, Solomon et al. 2012). The advantage of using Windows debugger is the versatility to analyze all dump formats – from minidumps to full memory dump. However, dump analysis is often manual and requires a high level of technical competency and knowledge of Windows internal data structures to extract useful information from the crash dumps. In order to facilitate a faster analysis requiring less technical resources, the use of an open-source memory analysis project - Volatility is proposed. This solution utilizes a memory analysis script, which would run a specific sequence of Volatility plugins, process the result and present the outcome as a percentage probability of malware infection. Depending on the size of the memory, this can be completed within 5 – 30 minutes.

```
Use !analyze -v to get detailed debugging information.

BugCheck D1, {a3005008, 2, 0, 949a05ab}

*** ERROR: Module load completed but symbols could not be loaded for myfault.sys
*** ERROR: Module load completed but symbols could not be loaded for NotMyfault.exe
Probably caused by : myfault.sys ( myfault+5ab )

Followup: MachineOwner
---------

kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: a3005008, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: 949a05ab, address which referenced memory

Debugging Details:
------------------


READ_ADDRESS:  a3005008 Paged pool

CURRENT_IRQL:  2

FAULTING_IP:
myfault+5ab
949a05ab 8b08            mov     ecx,dword ptr [eax]

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT

BUGCHECK_STR:  0xD1

PROCESS_NAME:  NotMyfault.exe

TRAP_FRAME:  a2a53a3c -- (.trap 0xffffffffa2a53a3c)
ErrCode = 00000000
eax=a3005008 ebx=861b8930 ecx=00000000 edx=8513d240 esi=a3001008 edi=00000004
eip=949a05ab esp=a2a53ab0 ebp=a2a53ab8 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010282
myfault+0x5ab:
949a05ab 8b08            mov     ecx,dword ptr [eax]  ds:0023:a3005008=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 949a05ab to 82882cdb

STACK_TEXT:
a2a53a3c 949a05ab badb0d00 8513d240 a3001008 nt!KiTrap0E+0x2cf
WARNING: Stack unwind information not available. Following frames may be wrong.
a2a53ab8 949a09db 861b8918 a2a53afc 949a0b26 myfault+0x5ab
a2a53ac4 949a0b26 86860dc0 00000001 8684e840 myfault+0x9db
a2a53afc 82878c29 86177238 861b8918 861b8918 myfault+0xb26
a2a53b14 82a6db29 86860dc0 861b8918 861b8988 nt!IofCallDriver+0x63
a2a53b34 82a70cfb 86177238 86860dc0 00000000 nt!IopSynchronousServiceTail+0x1f8
a2a53bd0 82ab763b 86177238 861b8918 00000000 nt!IopXxxControlFile+0x6aa
a2a53c04 8287f8fa 00000094 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
a2a53c04 77937094 00000094 00000000 00000000 nt!KiFastCallEntry+0x12a
000bf760 77935864 75ca9815 00000094 00000000 ntdll!KiFastSystemCallRet
000bf764 75ca9815 00000094 00000000 00000000 ntdll!NtDeviceIoControlFile+0xc
000bf7c4 77a8b9b5 00000094 83360018 000bf900 KERNELBASE!DeviceIoControl+0xf6
000bf7f0 004825a0 00000094 83360018 000bf900 kernel32!DeviceIoControlImplementation+0x80
000bf908 004827dc 00480000 00000000 00281cf7 NotMyfault+0x25a0
000bf998 77a8ed6c 7ffda000 000bf9e4 7795377b NotMyfault+0x27dc
000bf9a4 7795377b 7ffda000 7781d540 00000000 kernel32!BaseThreadInitThunk+0xe
000bf9e4 7795374e 00482945 7ffda000 00000000 ntdll!__RtlUserThreadStart+0x70
000bf9fc 00000000 00482945 7ffda000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Figure 4 Debugging a Crash Dump using Windbg

As the analysis is conducted at the Security Operations Center (SOC), where all the enterprise's sensors are monitored centrally, the memory analysis outcome could immediately be correlated with other sensor inputs to ascertain the impact of the incident. This would allow management to make better decisions, given the more accurate and timely information provided. The drawback of remote memory collection is the amount of sensitive data, such as passwords, that could potentially be collected via the memory dump (Patrick Jungles 2012). It is recommended that SOC operators and forensic

analysts who are given access to such memory dumps be required to undergo more stringent security background checks to reduce the possibility of abuse.

The proposed solution uses native Windows features, such as the Group Policy Object, for deployment, collection and analysis. This requires minimal changes to existing enterprise network infrastructure, which translates to lower cost of implementation.

When Windows crashes, the Windows Error Reporting mechanism can be used to create a Cabinet file containing information about the crash. Even though the cabinet file contains other files that could also be analyzed, this paper is only focused on the memory dump generated. Using Volatility, we will be able to discern if the client machine has been infected with malware.

3.2. Assumptions

Conditions exist whereby no crash dumps are generated even though a system crash has occurred. Examples of such cases could be due to corrupted kernel procedures, which handle crash dumps, or hard disk failure whereby the crash dump is unable to be written to the hard disk (Russinovich, Solomon et al. 2012). The frequency of such occurrence differs, depending on an enterprise's hardware and software configuration. One method of troubleshooting such crashes would be to boot the affected system in debugging mode. Any further kernel crashes would then trigger the Windows debugger for further analysis. However, this method requires the specific crash symptom to be reproducible which is not always possible. Due to the complexity and rarity of such scenarios, the solution proposed in this paper is based on the assumption that all anomalies and coding errors will crash should they trigger any of the Windows defense mechanism.

3.3. Proof of Concept

A virtualized test environment was setup with a Windows 2008R2 server, a Windows 7 client and a Windows XP client. The specifications of the virtual machines are as follow:

Role	Domain Controller and Crash Collection Server

Operating System	Windows Server 2008R2
Processor	Intel i7-4900MQ CPU @ 2.8Ghz
Memory (RAM)	1GB
System Type	32 Bit

Role	Client
Operating System	Windows 7 Ultimate SP1
Processor	Intel i7-4790 CPU @ 3.6Ghz
Memory (RAM)	1GB
System Type	32 Bit

Role	Client
Operating System	Windows XPSP2
Processor	Intel i7-4790 CPU @ 3.6Ghz
Memory (RAM)	1GB
System Type	32 Bit

The Windows 2008R2 server is configured as the domain controller (DC) of the domain "corp.zf.org". This server is also the crash dump storage server. 2 clients are connected to this DC, one installed with Windows XPSP2 and the other Windows 7. For ease of demonstration, the analysis engine (Volatility) is installed on the Windows 2008R2 Server. However, for live deployment in an enterprise network, it is recommended that the DC, crash dump storage server and analysis engine be installed on separate OS for security and performance consideration. Key components in this POC are as follow:

(1) Memory Dump Collection Shared Drive. A share is created on the server to host the target host list, the crash triggering mechanism – SysInternals - NotMyFault binary and the memory dump collection folder. The target host list is named "collectionlist.txt." and is a text file that lists the clients from which the SOC would like to request a memory dump from. The computers can be identified by their IP address, Username or Computer name. For this POC, the Computer name is used as the unique identifier.

(2) Registry Changes. Client side registry changes are required so that a full memory dump is generated for all crashes. In addition, the Windows Error Reporting mechanism is disabled to prevent any crashes from being sent to Microsoft. These changes will be deployed via group policy using the Computer Configuration startup script (available in **Appendix**) as shown in Figure 5.

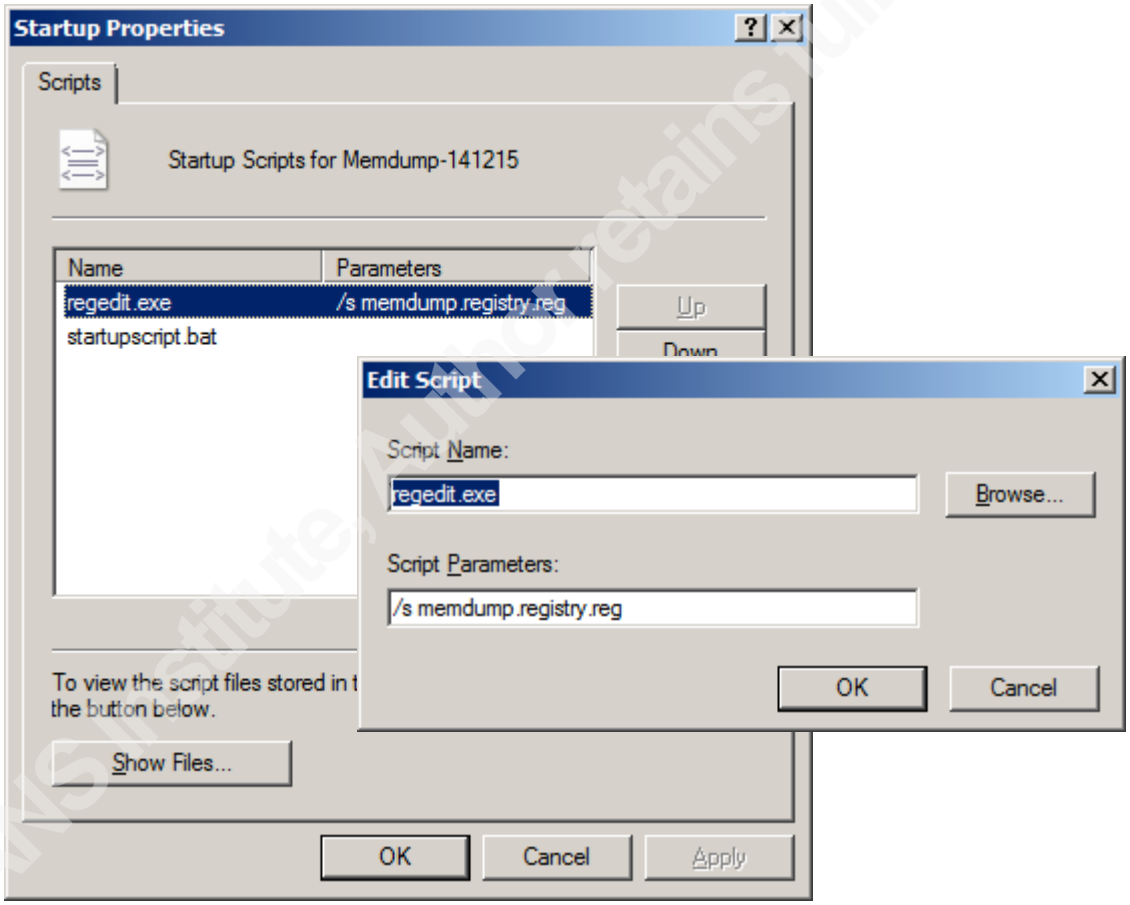


Figure 5 : Startup Scripts

(3) Polling Startup Script. A polling function is created to perform two tasks: (1) Move any crash dumps stored on client machines to the shared drive, (2) Check for a hostname match in the collection list and trigger a crash if it matches. This script (available in the **Appendix**) is deployed to all clients via the startup script as "startupscript.bat" as shown in Figure 5 above.

(4) Memory Malware Detection Script. A user-triggered script named "malwarescoretest.py" (provided in the **Appendix**) can be executed as shown in Figure 6. The output is a percentage figure on the likelihood of malware infection displayed on screen. 100% is a confirmed infected client while 0% means that the client is not likely to have been infected with malware.

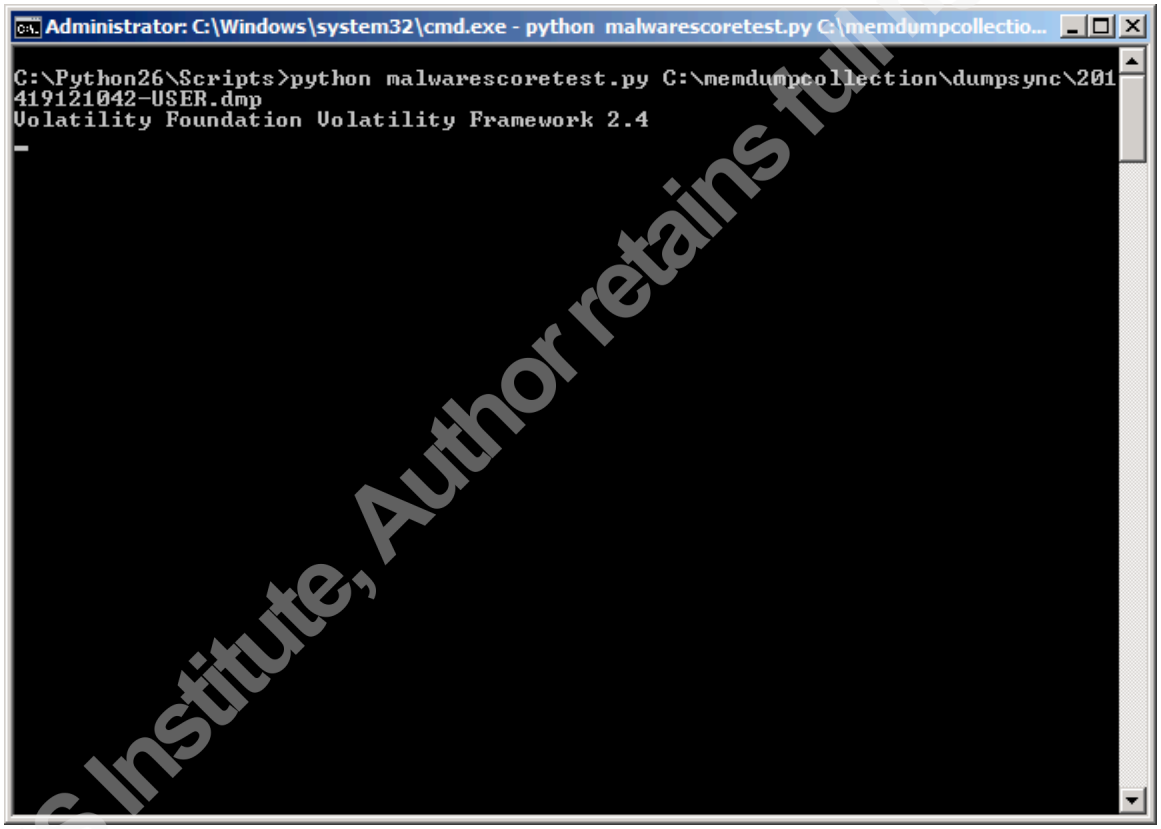


Figure 6 : Running malwarescoretest.py

As a POC, vulnerable applications were installed on the client's machines. The application chosen are popular document viewing and processing suites – Microsoft Office 2007 and Adobe Reader 9.0. The malware were used to infect the client machines and the memory of the machines was collected immediately after the malware was executed. In addition, the script was also tested against zeus.vmem, provided in Malware Analyst's Cookbook and DVD (Michael Ligh 2010).

VULNERABILITY	DESCRIPTION	DOCUMENT TYPE
CVE-2011-0609	Vulnerability in Adobe Flash Player that would allow remote attackers to execute arbitrary code via embedded	xls

Author Name, email@address

	swf content in Excel Spreadsheet.	
CVE-2011-0611	Vulnerability in AuthPlayLib.dll in Adobe Reader before 9.4.4 that would allow remote attackers to execute arbitrary code via crafted Flash Content.	pdf
CVE-2012-1535	Vulnerability in Adobe Flash Player that would allow remote attackers to execute arbitrary code via embedded swf content in a word document.	doc

3.4. Memory Analysis

Essentially, any application that needs to run in an operating system needs to be loaded into memory in order to execute. Through the memory management module of various operating systems, data remnants are often present after execution completes. This is an excellent source of information that is easily accessible to forensic analysts. This is a valuable source of forensic evidence for criminal investigators, as there is often limited amount of time available for the prosecution to prove their case. Critical information, such as encryption keys, network transaction information, and evidence of injected code, hidden processes and files, can often be found in memory (Michael Hale Ligh 2014).

With the proliferation of software packers, such as Armadillo, Themida and PELock, which are often used to prevent software reverse engineering attempts, memory analysis is sometimes the only option for analysts to analyze the executable in the shortest time possible.

In addition, advanced malware, such as Lurk Trojan (LoneStar 2012), only exist in memory and does not reside on the hard disk. This would be hard to detect without conducting memory forensic analysis.

3.4.1. Framework for Memory Analysis

Volatility is often the first tool that comes to mind when it comes to memory analysis. It is open Source and freely redistributable. Volatility is written in Python, which does not have library dependencies or require prior compilation to run. The output of Volatility is conveniently presented in text, which allows analysts to parse it easily using any text processing tools.

As part of this paper, a python script is written to call multiple Volatility plugins and process its output. A percentage metric indicating the probability of malware infection is presented at the end. The aim of this script is not to create a comprehensive scanning engine but rather a triage tool for quick assessment. The script is written to follow the 6-phase memory analysis methodology covered in depth in SANS forensic course 526. The 6 phases are (1) Rogue Processes (2) DLLs and Handles (3) Network Artifacts (4) Code Injection (5) Rootkits (6) Drivers. A detaileded explanation of the script is explained in the following sections. In addition, for forensic soundness and accountability, all commands executed by the script are logged with their date and time executed saved in as "log.txt". Apart from running the scripts, phases of investigation that cannot be scripted and require manual investigation are highlighted as well.

3.4.2. Preprocessing

Though Volatility is able to process Windows memory dumps without any conversion, it is recommended that the memory dump be converted to a raw memory image so that other memory forensic tools can process it. This would facilitate cross checking of results using different forensic tools, which is generally considered a best practice (Casey 2011). This conversion process can be initiated using the Volatility plugin "imagecopy". However, running "imagecopy" would require a few minutes of processing time, depending on the size of the client's memory. In view of the need for speedy diagnosis, Volatility is run directly on the Windows memory dump in this POC.

In addition, the plugin "crashinfo" can also be used to obtain more useful information about the crash, as shown in **Figure 7**. Useful information of note includes the "System Time" for timeline analysis and the "SystemUpTime" to know how long the client has been powered on and active.

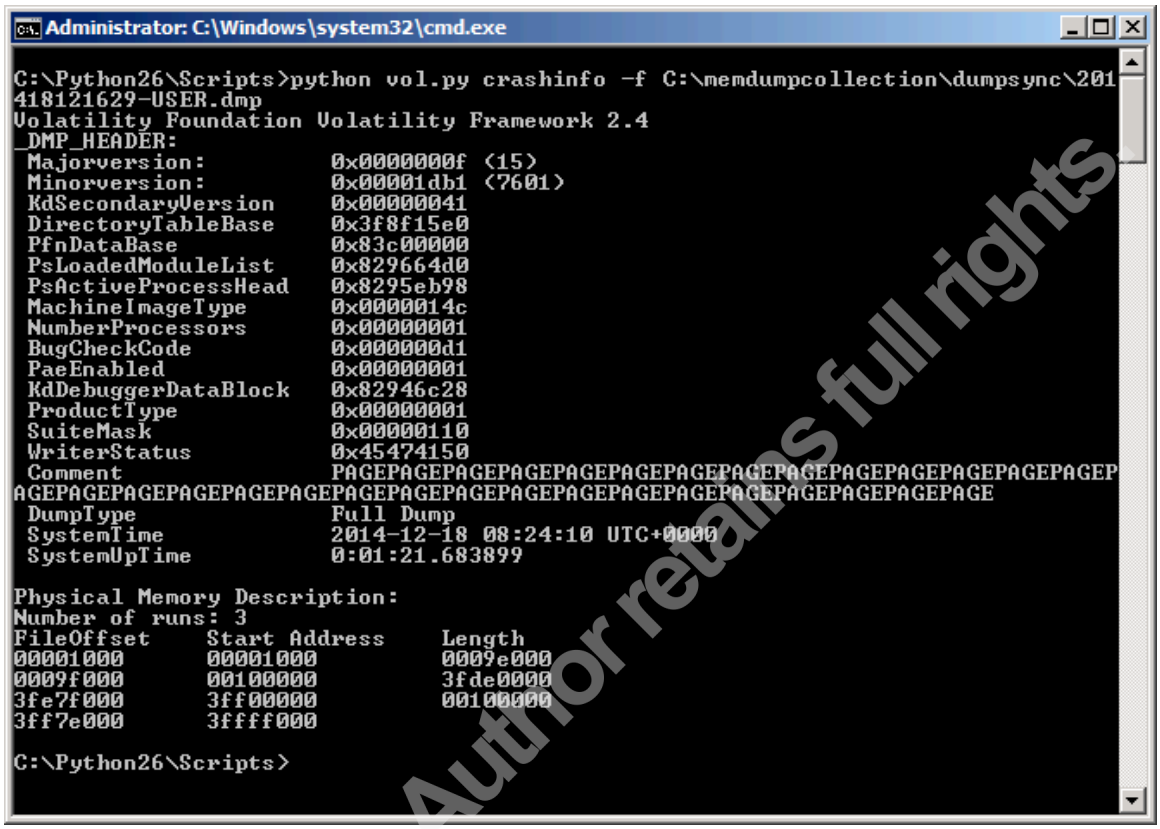


Figure 7: Output from running Crashinfo plugin

3.4.3. Phase 1 - Rogue Processes

3.4.3.1. Direct Kernel Object Manipulation

The Windows Operating System is largely written in C, C++ and C# (Waite 2009). These languages are objected oriented and modular in nature. Windows often uses linked lists to track its processes and tasks (e.g Network connections, Processes, Systems service table, Interrupt description Table, etc). Malware often exploits such linked lists to hide or inject themselves into the list to exploit the system. This is also known as Direct Kernel Object Manipulation (DKOM). Some of Volatility plugins target DKOM and scan the memory for any such discrepancies in the link lists and objects found. These objects within the memory pool can be identified by their pool tags or unique headers. The method of "bruteforce" searching of objects via its pool tag is known as pool tag scanning and can be used as an effective way to detect malware or other system anomalies (Michael Hale Ligh 2014).

Volatility plugins "pslist" and "psscan" can be used to detect DKOM of the EPROCESS structure. When malware uses this method of hiding itself, it will not appear when a user or process uses the task manager to list the active processes. Common malware that uses this method includes the ADORE rootkit (Ryan Riley 2009). Instead of manually comparing the output of "pslist" and "psscan", another Volatitility plugin called "psxview" can be used to verify the EPROCESS link list. The process list is checked against other lists such as the pspcid, csrss, session, deskthrd and thrdproc.

As the chances of getting false positive through pool tag scanning is high, one way of filtering out such cases would be to use the object's timestamp field. For example, an object with the pool tag (EPROCESS) has two timestamps – "CreateTime" and "ExitTime". The timestamp for "CreateTime" should not be zero or any arbitrarily huge or small UTC value. These information are summarized in the table below:

Name	Process
Pool Tag	EPROCESS
False Positive Identifier	EPROCESS.CreateTime EPROCESS.ExitTime
Object Scanner Plugin	Pslist
List Scanner	Psscan

It is noted that "psxview" is likely to reveal processes from previous boot up sessions, resulting in false positives. One means of false positive reduction is to compare the creation time of such processes against the key startup process "smss.exe". "smss.exe" is the Windows Session Manager Subsystem and is always executed during the boot up process of Windows. The creation time of "smss.exe" can be obtained from "pslist". If the suspicious process has a creation time earlier than smss.exe, it is likely to be a false positive. This process of verification is automated in the script. In addition, several other services are whitelisted as false positives. Examples of this include services.exe, lsass.exe, lsm.exe and svchost.exe. In addition, antivirus processes should also be whitelisted. For the remaining processes, any "false" entries will generate a malware score of 8/10.

Author Name, email@address

3.4.3.2. Legitimate Parent-Child Relationship

The Volatility plugin "pstree" can be used to display the hierarchical structure of processes. In general, the following rules apply and any deviation should be investigated:

1) "services.exe" should be the parent of all "svchost.exe".

2) "smss.exe" should be the parent of "winlogon.exe".

The above rules are checked by the script,, and any discrepancy will generate a malware score of 10/10.

3.4.3.3. Pass-The-Hash Attacks

All legitimate Windows processes should be launched with a valid SID. The Volatility plugin "getsids" will display all processes and the process owner identified via its SID. If a process does not have an ASCII account name (displayed in brackets) as shown in Figure 8, it is likely to be running with a domain account.

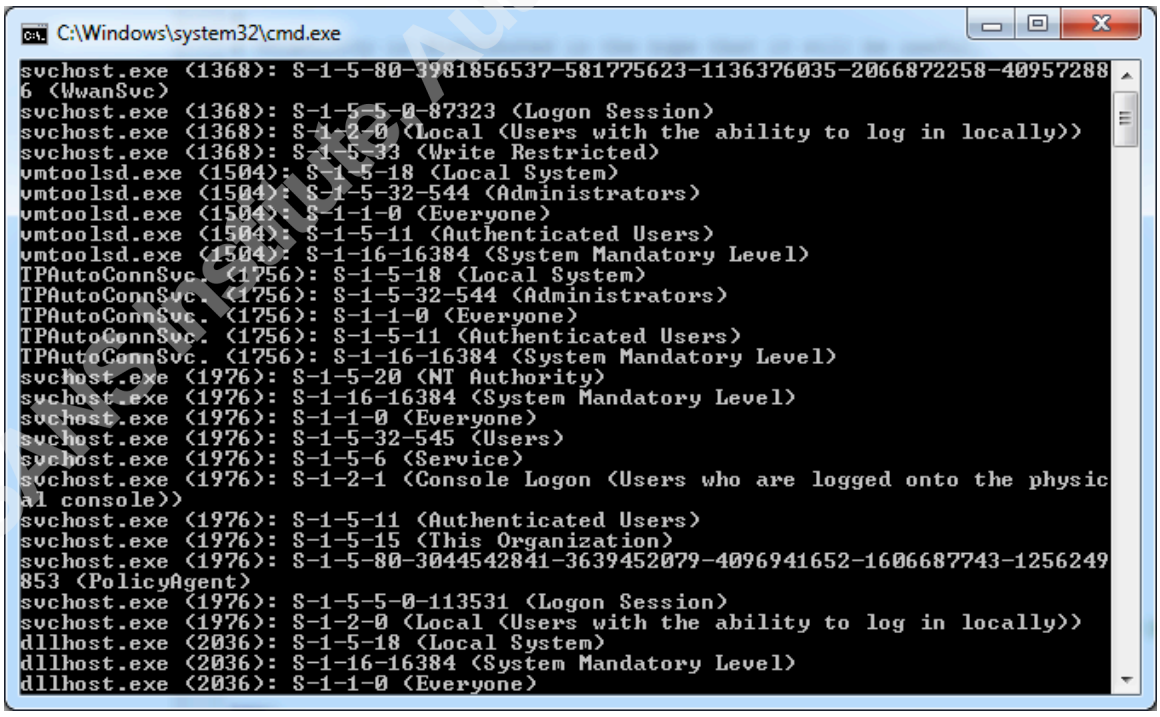


Figure 8 : Result of "getsids"

This should be verified against the domain controller using the sysinternals tool, "psgetsid". Due to the need for communication with the domain controller, which is sensitive for operations, this is not included in the script.

Author Name, email@address

3.4.4. Phase 2 - Dynamic Link Library and Handles

3.4.4.1. Suspicious DLL

The Volatility plugin, "verinfo", can be used to display information embedded in portable executable (PE) files and Dynamic Link Librarys (DLL). Unknown DLLs should be further analyzed using the plugin, "enumfunc". This would enumerate all functions used by the DLL. Legitimate exported functions should follow a standard naming convention, as shown in Figure 9.

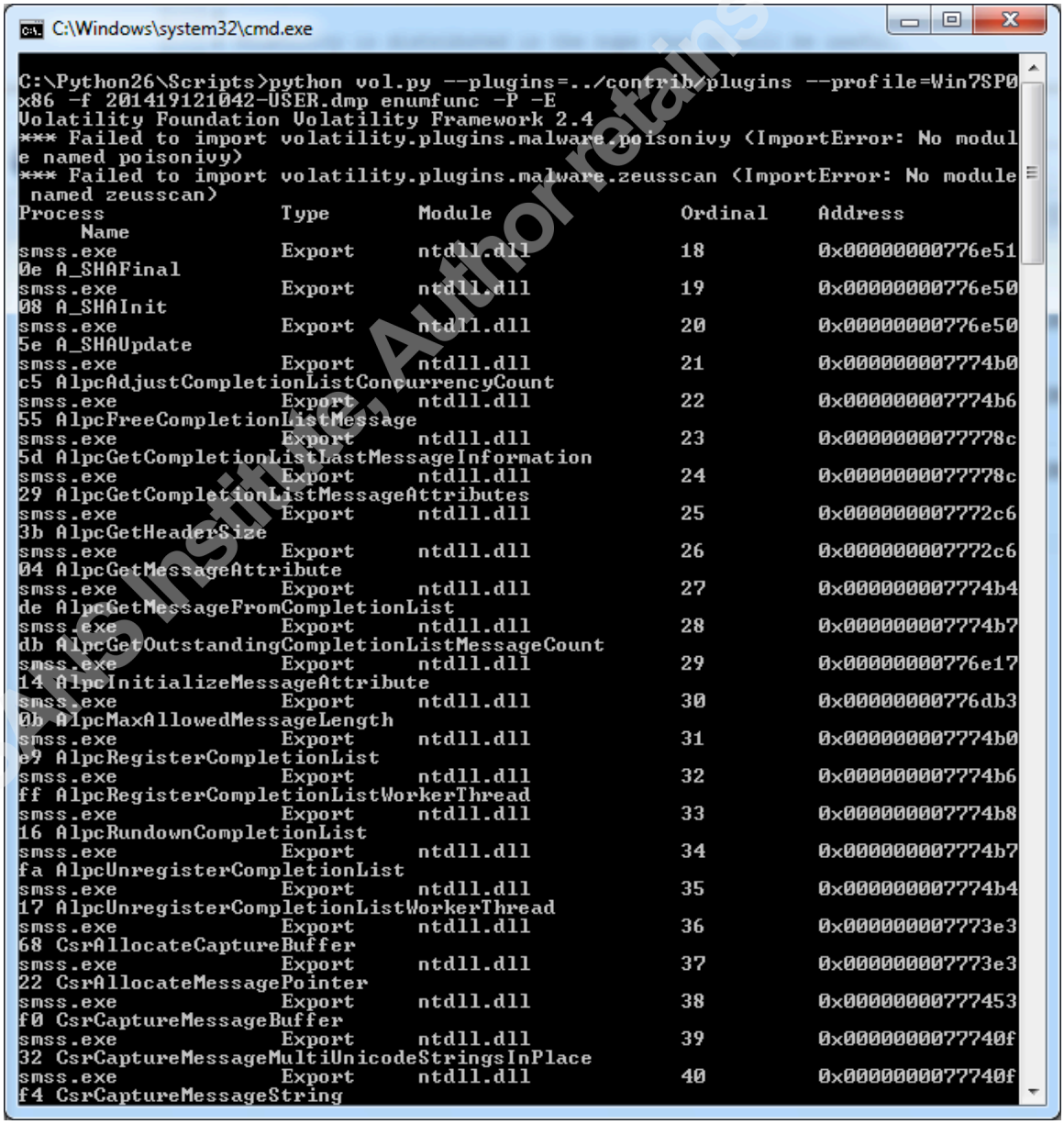


Figure 9 Running "enumfunc"

Similarly, imported functions are likely to be using familiar Windows libraries such as ntoskrnl.exe or ntdll.dll. Any anomalies could be investigated further using the plugin, "dlllist". "dlllist" would provide the base address to extract the dll for use with "dlldump". The extracted library can then be hashed and verified through Virustotal or analyzed statically. As this is a manual process, this is not included in the script.

3.4.4.1. Search Order Hijacking

DLL Search Order Hijacking is a way for malware to establish persistence on a victim machine by saving itself in strategic locations on a file system. This is different from other persistence mechanisms, as it does not require modification to the registry.

The vulnerability relies on the way Windows searches for a program's library to load. The first location it looks for is the folder where the program's executable is. The exception to this is when the DLL is a "known DLL" in the registry list "HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs". It then searches in the System directory, Windows directory, current directory and finally the PATH in the environment variable, in the mentioned order (Microsoft 2013). As such, to hijack and load a malicious DLL, the following detective measures are undertaken:

(1) Path Modification. The Volatility plugin, "envars", can be used to extract all exported environment variables. While going through the result, it is essential to note that repeated entries are likely as child processes will inherit the parent's environment variables. The script would match the PATH against the default or whitelisted value of "C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\\v1.0\\". Any deviation would get a malware score of 10/10. If the enterprise has a standardized OS image with a fixed path for all clients, the malware score is increased to 30.

(2) DLL Loaded Path. The plugin, "dlllist", would list all the imported functions in DLLs that are loaded. All imported functions not running from "C:\Windows\system32" would get a malware score of 6. In addition, the analyst should investigate all the suspicious dll using "dlldump" for a more conclusive confirmation.

Author Name, email@address

(3) Registry Key Verification. The plugin "printkey" can be used to dump the value of "HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs". The enterprise should maintain a list of known DLLs and compare this against the benchmarked white list.

3.4.4.2. Remotely Mapped Drives

Attackers often map network drive to move laterally or exfiltrate data. If SMB (net use) is used to map such as a network drive, it can be easily detected through the enumeration of all file object handles. File handles that are mapped using SMB have the prefix of "\Device\Mup". This is done automatically by the "handles" plugin. In addition, the plugin "symlinkscan" could be used to obtain additional information such as the time the shared drive is mounted. The script would search for the above SMB strings and any non-whitelisted shared drive detected would be assigned the malware score of 7/10.

3.4.5. Phase 3 - Network Artifacts

3.4.5.1. TCP Connections

Hidden TCP connections can be detected using the "connscan" plugin. This plugin searches for the TCPT_OBJECT pool tag within the memory and displays its remote connection address and the responsible PID. For Windows Vista, 2008 and 7, the equivalent plugin is "netscan". "netscan" searches for different pooltags of "TcpE, TcpL and UdpA" instead. As anomaly detection, the plugin "connections" is used to walk through the tcp link list pointed to by tcpip.sys. Any discrepancies in results between "connscan" and "connections" are assigned the malware score of 9. However, the "connections" plugin only works for Windows XP and 2003 memory dumps. As such, this scan is not performed for Windows 7 and above.

Name	Network Connections
Pool Tag	TCPT_OBJECT | TcpE, TcpL, UdpA
False Positive Identifier	Pid
Object Scanner Plugin	connscan (XP, 2003) netscan (Vista, 7, 2008)
List Scanner	Connections

3.4.5.1. Sockets

The plugin "sockets" could be used to detect socket structures created for network communication. This encompasses all communication protocols, including both TCP and UDP. Similarly, the plugin "sockscan" can be used to traverse the socket link list pointed to by the Windows tcpip.sys module. Similarly, the main limitation of the "socket" plugin is that it only works for Windows XP and 2003 memory dumps. Any discrepancies in sockets are highly suspicious and are assigned a multiplied malware score of 30.

Name	Network Sockets
Pool Tag	ADDRESS_OBJECT
False Positive Identifier	PID, Protocol, Creation Time
Object Scanner Plugin	sockets
List Scanner	sockscan

3.4.5.2. Domain and URL Scanning

Apart from pool tag scanning, Volatility has a plugin called "yarascan" which allows analyst to scan for specific patterns within the memory. The plugin is able to search the memory dump for patterns such as URL, IP addresses or domain name. This is a viable way to detect malware command and control channels. The script uses the yara rule "*/(www|net|com|org)/"* and any detection can be compared against known malicious domain list provided at http://www.malwaredomainlist.com/. Due to the need for manual verification and comparison, this is not scripted.

3.4.6. Phase 4 - Code Injection

Code injection can take place in multiple forms. The three main methods are (1) Remote Library Injection whereby the entire library is loaded from the disk onto another process to run, (2) Remote Shellcode Injection whereby shellcode instead of libraries, are injected or (3) Reflective DLL Loading whereby the library is injected from memory instead of being loaded from the hard disk (Michael Hale Ligh 2014).

Dynamic Link Libraries (DLL) are compiled binaries which can be loaded by processes to run generic functions. Malware often tries to inject malicious libraries into running processes (instead of spawning new suspicious processes) to reduce the chance of detection. This is also known as DLL injection. This malicious injection move can be further hidden by manipulating 3 linked lists: (1) LoadOrderList, (2) MemoryOrderList, (3) InitOrderList. These lists are maintained within the structure _LDR_DATA_TABLE_ENTRY. The Volatility plugin "ldrmodules" can be used to detect this by comparing the 3 lists against the process's memory allocation list – Virtual Address Descriptor tree. The script would assign a malware score of 30 to the client if the 3 lists show true,, but there is no mapped path. Analysts should investigate this further by extracting the suspected dll using "vaddump". Further alarm bells should be raised if the extracted dll has a PE32 – MZ header, indicating that it is an executable binary.

Instead of "ldmodules", another plugin called "malfind" can be used to find hidden injected code. "Malfind" will scan and flag out suspicious code in memory that is both executable and writable. These code can then be saved into a directory for further analysis using "dlldump" or "vaddump". Similar to malfind, the plugin "apihooks" can be used to detect malicious activities such as process hollowing.

3.4.7. Phase 5 - Rootkits

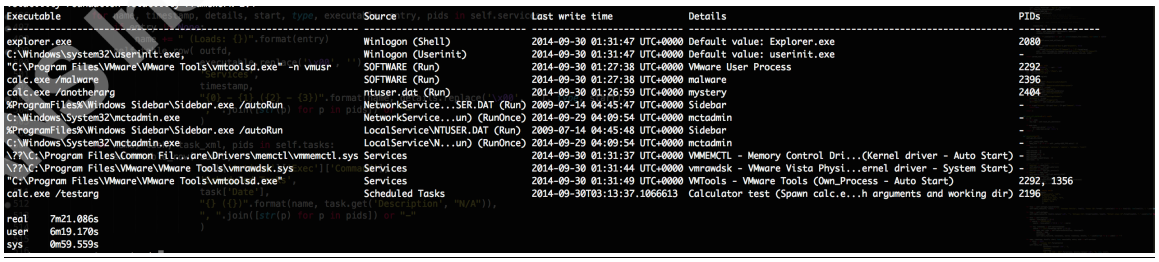
This phase aims to detect the persistence nature of rootkits. Key areas to look for are : (1) Autorun keys, (2) Services, (3) Scheduled Tasks. (4) Master Boot Record. Due to the need for semantic interpretation, the most of the tasks in this section, unless mentioned, is not included in the script.

(4) Autorun. The plugin "printkey" is used to extract the value of the registry key "Microsoft\Windows\CurrentVersion\Run" and

"\Microsoft\Windows\CurrentVersion\Runonce" This is often the registry keys malware modifies to establish persistency. The default value for these registry keys should be whitelisted and any discrepancy would be assigned a malware score of 10. The anomalous binary should also be extracted physically and examined further by tracing the run path on the client machine.

(5) Services. A malware could establish persistence on a system by running or masquerading as a service. This can be detected via the plugin "svcscan". In the script, any services that are in the "SERVICE_RUNNING" state and does not have the path "C:\WINDOWS\System32" is assigned a malware score of 7/10. To maximize the effectiveness for this plugin, the enterprise should benchmark legitimate running services on a machine and main this as a whitelist.

(6) Scheduled Tasks. A separate Volatility plugin called "autoruns" is created by author Thomas Chopitea from Paris. It will extract information from registry hives for malware persistence information as shown in **Figure 10**. It will also extract scheduled tasks for OS Windows 7 and above. Due to the richness of information provided, it is recommended that the output be analyzed manually.



Executable	Source	Last write time	Details	PIDs
explorer.exe	Winlogon (Shell)	2014-09-30 01:31:47 UTC+0000	Default value: Explorer.exe	2080
C:\Windows\system32\userinit.exe,	Winlogon (Userinit)	2014-09-30 01:31:47 UTC+0000	Default value: userinit.exe	-
"C:\Program Files\VMware\VMware Tools\vmtoolsd.exe" -n vmusr	SOFTWARE (Run)	2014-09-30 01:27:38 UTC+0000	VMware User Process	2292
calc.exe /malware	SOFTWARE (Run)	2014-09-30 01:27:38 UTC+0000	malware	2396
calc.exe /anotherarg	ntuser.dat (Run)	2014-09-30 01:26:59 UTC+0000	mystery	2404
%ProgramFiles%\Windows Sidebar\Sidebar.exe /autoRun	NetworkService...SER.DAT (Run)	2009-07-14 04:45:47 UTC+0000	Sidebar	-
C:\Windows\System32\mctadmin.exe	NetworkService...un) (RunOnce)	2014-09-29 04:09:54 UTC+0000	mctadmin	-
%ProgramFiles%\Windows Sidebar\Sidebar.exe /autoRun	LocalService\NTUSER.DAT (Run)	2009-07-14 04:45:48 UTC+0000	Sidebar	-
C:\Windows\System32\mctadmin.exe	LocalService\N...un) (RunOnce)	2014-09-29 04:09:54 UTC+0000	mctadmin	-
\??\C:\Program Files\Common Fil...are\Drivers\memctl\vmmemctl.sys	Services	2014-09-30 01:31:37 UTC+0000	VMMEMCTL - Memory Control Dri...(Kernel driver - Auto Start)	-
\??\C:\Program Files\VMware\VMware Tools\vmrawdsk.sys	Services	2014-09-30 01:31:44 UTC+0000	vmrawdsk - VMware Vista Physi...ernel driver - System Start)	-
"C:\Program Files\VMware\VMware Tools\vmtoolsd.exe"	Services	2014-09-30 01:31:49 UTC+0000	VMTools - VMware Tools (Own_Process - Auto Start)	2292, 1356
calc.exe /testarg	Scheduled Tasks	2014-09-30T03:13:37.1066613	Calculator test (Spawn calc.e...h arguments and working dir)	2196

real 7m21.086s
user 6m19.170s
sys 0m59.559s

Figure 10 Autoruns Output

(7) Master Boot Record. The plugin "mbrparser" can be used to scan for and extract the Master Boot Record that is loaded onto memory at boot time. The analyst can look for signs of exploitation in the assembly code (Levy 2012). To facilitate higher accuracy and faster analysis, it is recommended that a baseline of the "clean" MBR is kept by an enterprise in raw and MD5 format.

Memory Extraction

Volatility provides numerous ways for an analyst to extract suspicious files and objects from memory. Below is a brief description of each method. Although it is possible to run a script to indiscriminately carve all recognizable data types from the memory image, this process is time exhaustive. It is recommended that analysts use the above mentioned methods to identify suspicious objects and then proceed to carve specific objects instead. The carved object can then be hashed and uploaded to Virustotal for a quick analysis or it can be disassembled and analyzed manually using tools such as IDA Pro.

Extracting Process Memory. The plugin "memdump" can be used to extract all objects in a process memory. When "memdump" creates the dumpfile, all objects are saved in contiguous blocks. In order to separate each of them, the plugin "memmap" can be used.

Extracting Executables. If the analyst is only interested in PE objects, the plugin "procexedump" can be used. However, the extracted executable is unlikely to be able to run due to incorrect Import Address Table entries. However, this executable sample can still be disassembled and statically analyzed. In Volatility 2.4, procexedump has been subsumed into procdump and is the default behavior.

Extracting Packed Executables. A packed program would often exhibit the following signature: (1) Sections of zero bytes on disk but non-zero in memory (2) Sections with Unique Header Signatures, sometimes with ASCII string e.g UPX (3) Sections with high entropy value, typically above 7.5-9 (Robert Lyda 2009). Packed programs often allocate more "spaces" within their code to allow for extraction of packed code. As such, "procexedump" should **NOT** be used as it would read the PE header and re-align the executable, removing all the pre-allocated "spaces". These spaces may contain valuable unpacked code. When there is suspicion of a packed program, the plugin "procmemdump" should be used instead. "Procmemdump" will correctly extract the executable with the slack space intact. This would allow the identification of the packer header and the unpacked content. In Volatility 2.4, procmemdump is subsumed into procdump and can be called with "procdump -- memory"

Author Name, email@address

Rebuilding Executable Images with Imported Function Names. All the memory extraction methods mentioned thus far does not rebuild the Import Address Table. If the analyst were to load the extracted executable into disassembly programs such as IDA pro, all external function calls would be "gibberish", instead of the actual function name. These function names are typically extracted from the Import Address Table (IAT). However, malware is often known to modify the PE header or relocate the IAT after successful execution to make reverse engineering difficult. Thankfully, it is possible to rebuild the IAT using Volatility. The plugin "impscan" can be used to scan a process to extract the **Export** Address Table of all DLLs in the process (Morgan 2011). The output is an IDC file that can be loaded into IDA pro to give context to all imported function calls.

3.4.8. Phase 6 – Drivers

3.4.8.1. Interrupt Descriptor Table Hooking

Malware is known to hook the Interrupt Descriptor table to hijack the instruction pointer (IPtr) when an interrupt or exception happens. The plugin "idt" can be used to scan for any such hooking. Any IDT entries not pointing to ntoskrnl.exe is assigned a malware score of 30/10.

3.4.8.2. Inline Interrupt Descriptor Table Hooking

Hooking the IDT to external libraries can be easily detected. Instead of redirecting the execution to an external library, a much less obtrusive method would be to redirect the IPtr to a jump instruction within ntoskrl.exe which then jumps to the malicious code. This can be achieved by either modifying ntoskrnl.exe directly, or through ROP gadgets. Inline IDT hooking can be detected by using the plugins "apihooks". "Apihooks" would list all hooked addresses which can then be used as the base address in "dlldump" to extract the suspected DLL. The described process is not included in the script.

3.4.8.3. System Service Descriptor Table Hooking

The SSDT holds the pointer to kernel functions which can be called by legitimate applications. Windows has 4 SSDTs by default but normally (Bunden 2013), only 2 are used: ntoskrnl.exe and Win32k.sys. By exploiting the SSDT, malware would be able to

get administrator privilege. The ntoskrnl.exe is responsible for process related tasks such as process creation or destruction while Win32k.sys handles graphic rendering. Win32k.sys is a popular privilege escalation avenue as it handles font rendering which are often user inputs. Any successful exploitation would result in a root level compromise. To detect any such malicious activity, the plugin "ssdt" can be used. Any entries not pointing to ntoskrnl.exe and win32k.sys are given the malware score of 30.

Further analysis on the suspicious modules can be performed by using the plugin "modules". This would list down the Base address of the library/module which can then be used to extracted using "moddump" for further analysis.

3.4.8.4. Driver Hooking

The IO Manager handles all Inputs and Outputs between the OS and drivers. The Driver dispatch table hold the API which the driver uses to communicate with the IO manager. The plugin "modules" can be used to list all drivers and kernel modules loaded in memory. These drivers are typically located within system32 and any anomaly are assigned a malware score of 10. Instead of "modules", the plugin "modscan" can also be used. Any discrepancy in result from "modules" is indicative of root kit behavior and would be assigned a multiplied malware score of 30.

Name	Kernel Modules
Pool Tag	LDR_DATA_TABLE_ENTRY pointed to by PSLoadedModuleList
False Positive Identifier	N/A
Object Scanner Plugin	modscan
List Scanner	modules

Since all device drivers are definitely kernel modules, the plugins "modscan" and "modules" can also be used to list all device drivers. If the avenue of exploitation is suspected to come from driver modules, the plugin "driverscan" can be used to list down only device drivers instead. This is an effective way of data reduction.

Lastly, the plugin "unloadedmodules" can be used to identify suspicious modules that have been used and subsequently unloaded. This would be compared against the

output of "modscan" and "modules" and if they are missing, it would be assigned a malware score of 20 as kernel modules once loaded, are rarely unloaded.

3.4.8.5. I/O Request Packet Hooking

Similar to IDT, it is possible to hook a device driver's interrupt table to hijack a program's execution. The plugin "driverirp" can be used to detect any such malicious hooking. As such, any hooks not pointing back to itself or to ntoskrnl.exe are assigned a malware score of 30.

3.4.9. Other Forensic Evidence

For investigation and attribution purpose, it is often necessary to identify the account which was logged in during the occurrence of the incident. This can be done by using the plugin "mutantscan" and searching for the string "Documents and Settings". Similarly, the plugins "hivelist" and "hivescan" can be used.

The "prinkey" plugin can also be used to extract the value of "ControlSet001\Enum\USBSTOR" to get a history of all connected USB devices to the machine.

In the event that a suspicious USB device has been connected to the machine, it is possible to identify any binaries that were executed from it using the plugin "filescan". This plugin searches for FILE_OBJECT and displays the path of execution. Mount point and suspicious filenames can then be extracted for further investigation.

The plugin "Shimcache" can be used to check if any malicious programs were executed. This can be correlated against the user that was logged in. Ironically, this was not the purpose Microsoft created the registry for. ShimCache or AppCompatCache was originally created to track all applications that ran before the system was shutdown (Newton 2011). This was to be used for application compatibility debugging. As such, this registry is only updated during system shutdown. This registry value can be used to identify malicious user activities such as installing a non-legitimate program, running it and then uninstalling it.

3.5. Results Interpretation

The scanning script uses several whitelists that should be maintained by the system administrator in order to maximize the benefit of the script. With well-maintained whitelists, the scan should show a 0% malware score as seen in **Figure 11** for a clean client.

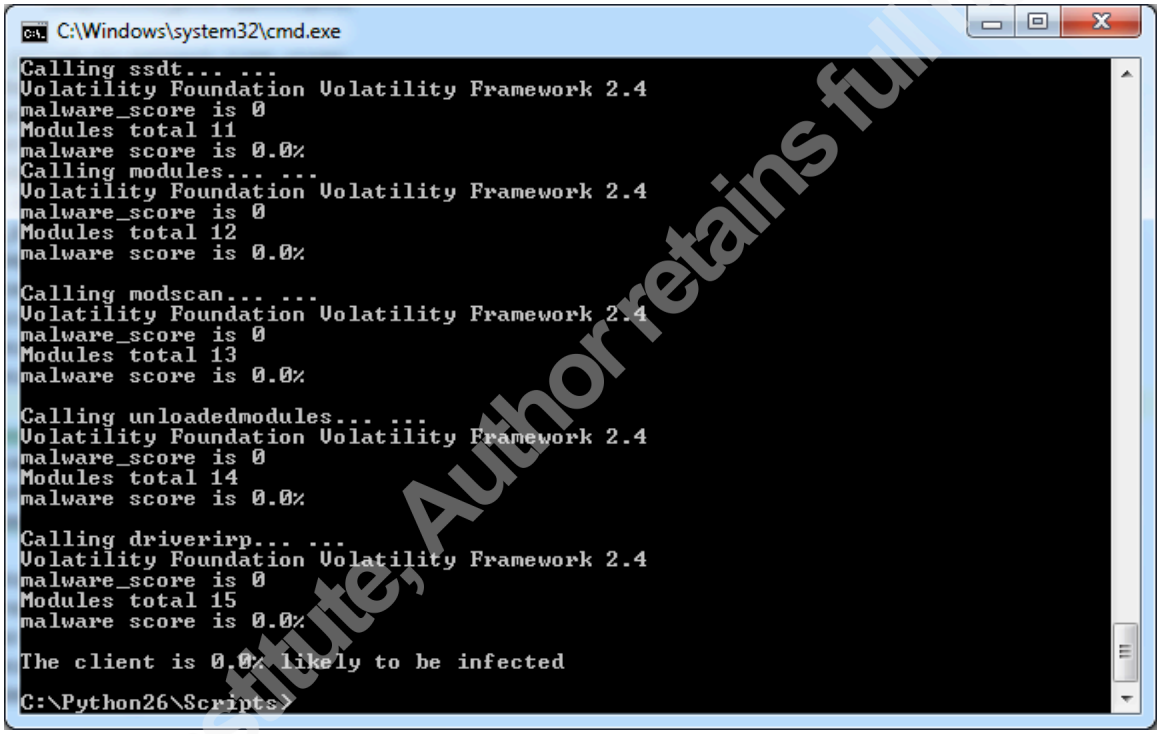


Figure 11: Result from an uninfected client

The script aims to provide a fast way for SOC operators to quickly diagnose a malware infection and its means. Based on the script, kernel level exploitations are scored higher to emphasize the urgency to follow up. Similarly, socket anomalies are also scored higher to because malware that establishes an outbound network connection hightens the risk of data loss and reputation damage that an enterprise might incur. **Figure 12** shows the result of a zeus infected client.

Author Name, email@address

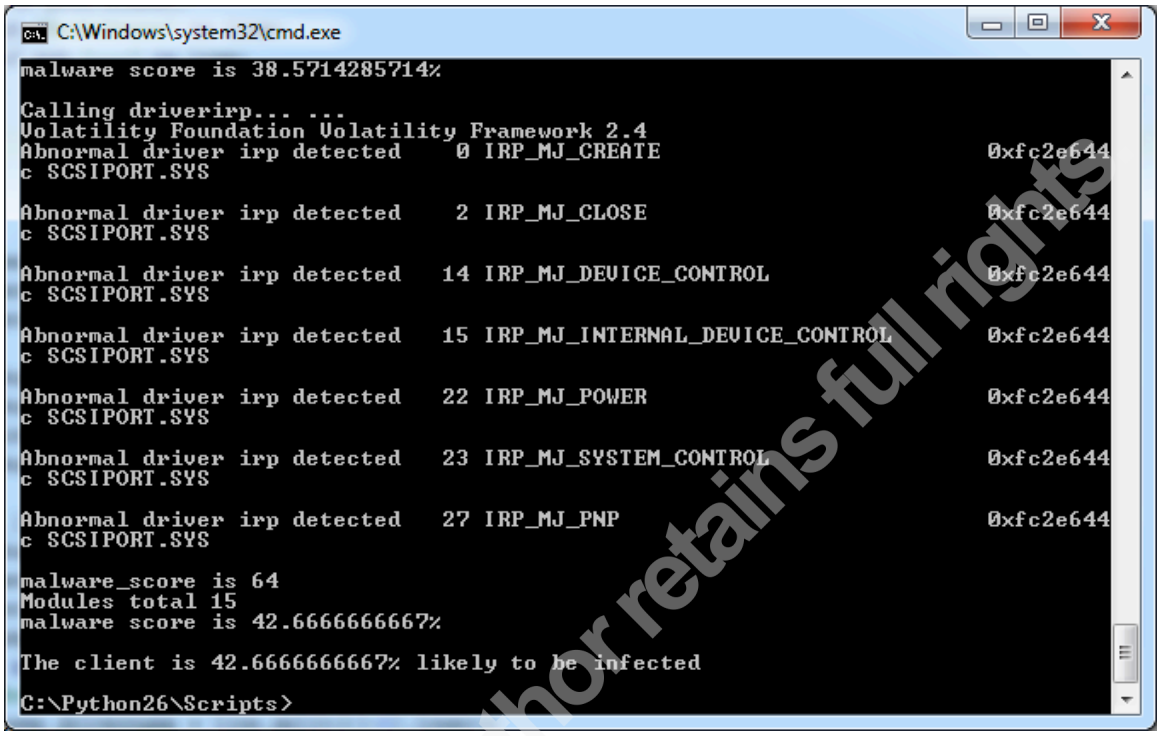


Figure 12: Result from scanning zeus.vmem

3.6. Challenges

With the proliferation of 64 bit architectures and the lowering cost of memory, it is now more likely that client machines will be deployed with more than 4GB of ram. This is likely to cause a strain on an enterprise's bandwidth, should memory collection be done on a large scale. However, this can be resolved with proper memory collection management. The collection latency of client machines should be prioritized based on threat and risk assessment. For example, the memory from high value targets such as C-level personnel in an enterprise should be given the highest priority for memory transfer. Remaining users' memory can be collected during off-office hour where there is less bandwidth usage.

As most Volatility plugins are written for Windows based memory analysis, an enterprise with a heterogeneous deployment may not be able to fully utilize this solution. However, it is possible to facilitate memory collection off *nix machine with customized kernel plugin such as Linux Memory Extractor (LiME) which is a loadable kernel module. The extracted memory can then be analyzed manually. File system agnostic tools

such as Bulk Extractor can be used to extract information such as suspicious outgoing network connections.

4. Further Work

Additional work can be done to analyze the benefit of using Windows debugger to analyze crash dumps. Tools to merge paged out memory content would also be highly beneficial for a comprehensive forensic analysis.

5. Conclusion

It is highly recommended that organizations leverage the Windows crash dump collection mechanism to provide them with a quick triage solution for speedy incident response.

Author Name, email@address

References

Bunden, R. B. (2013). The Rootkit Arsenal.

Casey, E. (2011). Digital Evidence and Computer Crime: Forensic Science, Computers and the Internet.

Chen, X. (2013). "ASLR Bypass Apocalypse in Recent Zero-Day Exploits." Retrieved 30 December, 2014, from https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html.

Eric Hutchins, M. C., Dr. Rohan Amin (2011). Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains.

Levy, J. (2012). "MoVP 4.3 Recovering Master Boot Records (MBRs) from Memory." from http://Volatility-labs.blogspot.sg/2012/10/movp-43-recovering-master-boot-records.html.

LoneStar (2012). "Trojan-Spy.Win32.Lurk." from http://www.enigmasoftware.com/trojanspywin32lurk-removal/.

McDonald, G., et al. (2013). Stuxnet 0.5: The Missing Link**:** 18.

Michael Hale Ligh, A. C., Jamie Levy, Aaron Walters (2014). The Art of Memory Forensics.

Michael Ligh, S. A., Blake Hartstein, Matthew Richard (2010). Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code.

Microsoft (2013). "Search Path Used by Windows to Locate a DLL." from https://msdn.microsoft.com/en-us/library/7d83bc18.aspx.

Microsoft (2015). "Enhanced Mitigation Experience Toolkit."

Microsoft Secure Windows Initiative Team (2009). "Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP." Retrieved 30 December, 2014, from http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx.

Morgan, M. (2011). "Rebuilding Executable Images From Memory." from https://marksforensicblog.wordpress.com/2011/11/29/rebuilding-executable-images-from-memory/.

Author Name, email@address

NetMarketShare (2014). "Desktop Operating System Market Share." Retrieved 30 December, 2014, from http://www.netmarketshare.com/operating-system-market-share.aspx.

Newton, T. (2011). "Demystifying Shims - or - Using the App Compat Toolkit to make your old stuff work with your new stuff." from http://blogs.technet.com/b/askperf/archive/2011/06/17/demystifying-shims-or-using-the-app-compat-toolkit-to-make-your-old-stuff-work-with-your-new-stuff.aspx.

Patrick Jungles, M. S., Roger Grimes (2012). "Mitigating Pass-the-Hash (PtH) Attacks and Other Credential Theft Techniques." from http://download.microsoft.com/download/7/7/A/77ABC5BD-8320-41AF-863C-6ECFB10CB4B9/Mitigating%20Pass-the-Hash%20(PtH)%20Attacks%20and%20Other%20Credential%20Theft%20Techniques_English.pdf.

Robert Lyda, J. H. (2009). "Using Entropy Analysis to Find Encrypted and Packed Malware."

Russinovich, M., et al. (2012). Windows Internals. United States of America.

Ryan Riley, X. J., Dongyan Xu (2009). Multi-Aspect Profiling of Kernel Rootkit Behavior.

Waite, R. (2009). "What Programming Language is Windows written in?". from https://social.microsoft.com/Forums/en-US/65a1fe05-9c1d-48bf-bd40-148e6b3da9f1/what-programming-language-is-windows-written-in.

Whitehouse, O. (2007). An Analysis of Address Space Layout Randomization on Windows Vista**:** 20.

Author Name, email@address

Appendix

Domain Controller Installation

Startup Script

```
REM Variables
SET TFOLDER=\\DC\memdumpcollection

echo %DATE% %TIME% Script Begin >>%TFOLDER%\%COMPUTERNAME%.txt

:CHECKER

FOR /f "tokens=2-4 delims=/ " %%a IN ('date /t') DO (set mydate=%%c%%b%%a)
FOR /f "tokens=1-2 delims=/:" %%a IN ("%TIME%") DO (set mytime=%%a%%b)
FOR /f "usebackq" %%i in (`hostname`) do (set hostnamestring=%%i)
set memdumpstring=%mydate%%mytime%-%hostnamestring%.dmp


echo %DATE% %TIME% Renaming in progress>>%TFOLDER%\%COMPUTERNAME%.txt
rename C:\Windows\Minidump\memdump.dmp "%memdumpstring%"

echo %DATE% %TIME% Moving file to dumpsync >>%TFOLDER%\%COMPUTERNAME%.txt
move C:\Windows\Minidump\* %TFOLDER%\dumpsync

echo %DATE% %TIME% Checking collectionlist for match for %COMPUTERNAME%
>>%TFOLDER%\%COMPUTERNAME%.txt
SET CRASHCONDITION="NO"

FOR /F "tokens=*" %%a in (%TFOLDER%\collectionlist.txt) DO (

        echo %DATE% %TIME% Comparing %COMPUTERNAME% vs %%a
>>%TFOLDER%\%COMPUTERNAME%.txt

        IF %%a==%COMPUTERNAME% (
                set CRASHCONDITION="YES"
                echo %DATE% %TIME% Match found for %%a >>%TFOLDER%\%COMPUTERNAME%.txt
                @echo CRASHCONDITION SET TO YES) ELSE (

                echo %DATE% %TIME% Match NOT FOUND for %%a >>%TFOLDER%\%COMPUTERNAME%.txt

                echo %%a>> %TFOLDER%\newcollectionlist.txt)
)

echo %DATE% %TIME% Comparison Complete, replacing collectionlist >>%TFOLDER%\%COMPUTERNAME%.txt

move /Y %TFOLDER%\newcollectionlist.txt %TFOLDER%\collectionlist.txt

echo %DATE% %TIME% Check Condition for crash, Condition = %CRASHCONDITION%
>>%TFOLDER%\%COMPUTERNAME%.txt

IF %CRASHCONDITION%=="YES" (
        echo %DATE% %TIME% Crashing... ... >>%TFOLDER%\%COMPUTERNAME%.txt
        %TFOLDER%\NotMyFault\x86\NotMyfault.exe /crash)

echo %DATE% %TIME% Delaying >>%TFOLDER%\%COMPUTERNAME%.txt
ping 1.1.1.1 -n 1 -w 10000>nul
echo %DATE% %TIME% Delay for 10sec Completed, Looping... ... >>%TFOLDER%\%COMPUTERNAME%.txt

goto CHECKER

echo %DATE% %TIME% CompletedDump >>%LOGFOLDER%\%mydate%%mytime%-%hostnamestring%.txt
```

Author Name, email@address

Registry settings (memdump.registry.reg)

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CrashControl]
"AutoReboot"=dword:00000001
"LogEvent"=dword:00000001
"MinidumpsCount"=dword:00000032
"DumpFilters"=hex(7):64,00,75,00,6d,00,70,00,66,00,76,00,65,00,2e,00,73,00,79,\
  00,73,00,00,00,00,00
"CrashDumpEnabled"=dword:00000001
"Overwrite"=dword:00000000
"MinidumpDir"=hex(2):43,00,3a,00,5c,00,57,00,69,00,6e,00,64,00,6f,00,77,00,73,\
  00,5c,00,4d,00,69,00,6e,00,69,00,64,00,75,00,6d,00,70,00,00,00
"DumpFile"=hex(2):43,00,3a,00,5c,00,57,00,69,00,6e,00,64,00,6f,00,77,00,73,00,\
  5c,00,4d,00,69,00,6e,00,69,00,64,00,75,00,6d,00,70,00,5c,00,6d,00,65,00,6d,\
  00,64,00,75,00,6d,00,70,00,2e,00,64,00,6d,00,70,00,00,00
@=dword:00000000
```

Malware Scoring Python Script

```
import subprocess
import sys
import datetime
import time
import getopt

# def main(argv):
#     inputfile = ''
#     outputfile = ''
#     try:
#         opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
#     except getopt.GetoptError:
#         print 'malwarescoretest.py -i <inputfile> -o <outputfile>'
#     sys.exit(2)
#
#     for opt, arg in opts:
#         if opt == '-h':
#             print 'test.py -i <inputfile> -o <outputfile>'
#             sys.exit()
#         elif opt in ("-i", "--ifile"):
#             inputfile = arg
#         elif opt in ("-o", "--ofile"):
#             outputfile = arg
#     print 'Input file is "', inputfile
#     print 'Output file is "', outputfile
#
# if __name__ == "__main__":
#    main(sys.argv[1:])
#
def debug(debug_flag, output):
    if debug_flag == 1:
        print output

def utc_to_epoch(timestamp):
    pattern = '%Y-%m-%d %H:%M:%S'
    epoch = int(time.mktime(time.strptime(timestamp,pattern)))
    return epoch

def malwarescoretest(opt):

    if opt == '-h':
        print 'malwarescoretest.py <memorydumpfile> --profile=<Windows Version>'
        sys.exit()

    #debugging flag, 1 is on, 0 is off
```

Author Name, email@address

```
    debug_flag = 1

    #Logging Preparation
    log_file = datetime.datetime.now().strftime('%Y%m%d_%H%M%S-log.txt')
    logging = open(log_file, 'w')

    #Base variables to keep track of scores
    malware_score = 0
    malware_score_modules = 0

    #Variables for 3.3.3.1
    suspicious_pid = []
    suspicious_pid_dictionary = {}
    suspicious_pid_count = 0
    whitelist_psxview = ['lsass.exe', 'services.exe', 'lsm.exe', 'svchost.exe', 'System', 'csrss.exe', 'cmd.exe',
'csrss.exe', 'smss.exe', 'HOSTNAME.EXE']

    #Variables for 3.3.3.2
    svchost_parent = []
    abnormal_parent_count = 0

    #Variables for 3.3.4.1(1)
    matched_path_count = 0
    whitelist_path =
["C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\\v
1.0\\",
                "C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem"]

    #Variables for 3.3.4.1(2)
    false_positive_dll_path = 0
    legit_dll_path = "C:\Windows\System32"
    whitelist_dll_path = ["C:\Windows\WinSxS",
                    "C:\Program Files\VMware\VMware Tools",
                    "C:\Windows\servicing",
                    "C:\ProgramData\Microsoft\Windows Defender",
                    "C:\Program Files\Common Files\VMware\Drivers",
                    "C:\Windows\Explorer.EXE",
                    "C:\Program Files\Common Files\microsoft shared",
                    "C:\Program Files\Internet Explorer\ieproxy.dll",
                    "C:\Windows\ehome",
                    "C:\Windows\Microsoft.NET\Framework",
                    "C:\Users\user\Documents\NotMyFault\\x86\NotMyfault.exe"]
    abnormal_dll_path_count = 0

    #Variables for 3.3.4.2
    false_positive_smb_status = 0
    abnormal_smb_mount = 0
    whitelist_smb_share = ["DC\memdumpcollection"]

    #Variables for 3.3.5.1
    sockscan_result = {}
    pid_match = 0
    mismatch_socket_sockscan = 0

    #Variables for 3.3.6
    abnormal_dll_with_no_path = 0

    #Variables for 3.3.7 (1)
    false_positive_runkey = 0
    abnormal_run = 0
    whitelist_runkey_values = ["C:\Program Files\VMware\VMware Tools",
                    "%SYSTEMROOT%\SYSTEM32\WerFault.exe"]

    #Variables for 3.3.7 (2)
    false_positive_service = 0
    abnormal_service = 0
    whitelist_service_path = ["C:\Windows\system32",
                    "\driver\\",
                    "\\filesystem\\",
                    "c:\windows\servicing\\trustedinstaller.exe",
```

```
                    "c:\program files\\vmware\\vmware tools"]


#Variables for 3.3.8.1
abnormal_idt_entry = 0
false_positive_idt = 0
whitelist_idt_entries = ["hal.dll"]

#Variables for 3.3.8.3
abnormal_ssdt_entry = 0
false_positive_ssdt = 0
whitelist_ssdt_entries = [""]

#Variables for 3.3.8.4 (1)
abnormal_driver_path = 0
false_positive_driver_path = 0
whitelist_driver_entries = [ "C:\Program Files\VMware\VMware Tools",
                             "C:\Program Files\Common Files\VMware\Drivers",
                             "C:\Windows\system32\drivers\myfault.sys"]

#Variables for 3.3.8.4 (2)
modules_list = []
abnormal_module = 0

#Variables for 3.3.8.4 (3)
abnormal_unloaded_module = 0
whitelist_unloaded_module = ["agp440.sys"]

#Variables for 3.3.8.5
abnormal_driver_irp = 0
false_positive_driverirp = 0
whitelist_driverirp = ["HIDCLASS.SYS",
                       "Unknown",
                       "Wdf01000.sys",
                       "HdAudio.sys",
                       "ks.sys",
                       "portcls.sys",
                       "VIDEOPRT.SYS",
                       "ndis.sys",
                       "wanarp.sys",
                       "dxgkrnl.sys",
                       "USBPORT.SYS",
                       "PCIIDEX.SYS",
                       "CLASSPNP.SYS",
                       "ataport.SYS",
                       "storport.sys",
                       "hal.dll"]

#3.3.2 Preprocessing
#Running imageinfo to get Windows profile automatically. Assuming last profile is the correct one.
debug(debug_flag, "\nChecking profile... ...")
log_command = 'python vol.py imageinfo -f ' + opt
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if "Suggested Profile" in line:
        profile = line.split()[3]
        profile = profile[:-1]
        debug(debug_flag, "The profile is " + profile)

##3.3.3.1 Direct Kernel Object Manipulation
debug(debug_flag, "\nCalling psxview... ...")
log_command = 'python vol.py -f ' + opt + ' psxview --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if "False" in line:
        process_name = line.split()[1]
        if not process_name in whitelist_psxview:
```

```
            pid = line.split()[2]
            suspicious_pid.append(pid)

    #Running pslist to extract time stamp
    debug(debug_flag, "Calling pslist... ...")
    log_command = 'python vol.py -f ' + opt + ' pslist --profile=' + profile
    logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
    proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
    for line in proc.stdout:
        temp_pid = line.split()[2]
        if temp_pid in suspicious_pid:
            temp_array = line.split()
            utc  = temp_array[-3] + ' ' + temp_array[-2]
            suspicious_time = utc_to_epoch(utc)
            suspicious_pid_dictionary[str(temp_pid)] = suspicious_time

        if "smss.exe" in line:
            temp_array = line.split()
            utc  = temp_array[-3] + ' ' + temp_array[-2]
            smss_time = utc_to_epoch(utc)

    debug(debug_flag, "Comparing pid with smss timing... ...")
    for key,var in suspicious_pid_dictionary.items():
        if var > smss_time:
            suspicious_pid_count += 1
            debug(debug_flag, "Suspicious pid found! " + key)

    if suspicious_pid_count > 0:
        malware_score += 8
    malware_score_modules += 1

    debug(debug_flag, "malware_score is " + str(malware_score))
    debug(debug_flag, "Modules total " + str(malware_score_modules))
    malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
    debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

    #3.3.3.2 Legitimate Parent-Child Relationship
    debug(debug_flag, "\nCalling pstree... ...")
    log_command = 'python vol.py -f ' + opt + ' pstree --profile=' + profile
    logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
    proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
    for line in proc.stdout:
        temp_process_name = line.split()[0]
        temp_pid = line.split()[1]
        temp_ppid = line.split()[2]
        if "." == temp_process_name or ".." in temp_process_name:
            temp_process_name = line.split()[1]
            temp_pid = line.split()[2]
            temp_ppid = line.split()[3]
        if "services.exe" in temp_process_name:
            service_pid = temp_pid
        if "smss.exe" in temp_process_name:
            smss_pid = temp_pid
        if "csrss.exe" in temp_process_name:
            csrss_parent = temp_ppid
        if "winlogon.exe" in temp_process_name:
            winlogon_parent = temp_ppid
        if "svchost.exe" in temp_process_name:
            svchost_parent.append(temp_ppid)

    debug(debug_flag, "Checking winlogon and csrss parent... ...")
    if not winlogon_parent == smss_pid and not csrss_parent == winlogon_parent:
            abnormal_parent_count += 1
            debug(debug_flag, "Abnormal winlogon or csrss parent found")

    debug(debug_flag, "Checking svchost parent... ...")
    for i in svchost_parent:
        if not i == service_pid:
            abnormal_parent_count += 1
            debug(debug_flag, "Abnormal svchost parent found for pid " + i)
```

```
if abnormal_parent_count > 0:
    malware_score += 10
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")


#3.3.4.1 Search Order Hijacking - Path Modification
debug(debug_flag, "\nCalling envars... ...")
log_command = 'python vol.py -f ' + opt + ' envars --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    temp_process_name = line.split()[1]
    temp_variable=line.split()[3]
    temp_path = line.split()[4]
    if temp_process_name == "csrss.exe" and "Path" == temp_variable:
        for i in whitelist_path:
            if temp_path == i:
                matched_path_count += 1

if matched_path_count == 0:
    malware_score += 30
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.4.1 Search Order Hijacking - DLL Loaded path
debug(debug_flag, "\nCalling dlllist... ...")
log_command = 'python vol.py -f ' + opt + ' dlllist --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if "C:\\" in line:
        if not legit_dll_path.lower() in line.lower():
            for i in whitelist_dll_path:
                if i.lower() in line.lower():
                    false_positive_dll_path += 1
            if false_positive_dll_path > 0:
                false_positive_dll_path = 0
            else:
                abnormal_dll_path_count += 1
                debug(debug_flag, "Abnormal DLL path found! " + line)

if abnormal_dll_path_count > 0:
    malware_score += 6
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.4.2 Remotely Mapped Drives
debug(debug_flag, "\nCalling handles... ...")
proc = subprocess.Popen('python vol.py -f ' + opt + ' handles -t File --profile=' + profile, stdout=subprocess.PIPE)
for line in proc.stdout:
    if "\Device\Mup\;" in line:
        for i in whitelist_smb_share:
            if i.lower() in line.lower():
                false_positive_smb_status += 1
        if false_positive_smb_status > 0:
            false_positive_smb_status = 0
```

```
        else:
            abnormal_smb_mount += 1
            debug(debug_flag, "Abnormal smb_mount detected " + line)

if abnormal_smb_mount > 0:
    malware_score += 7
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.5.1 Sockets
if "WinXP" in profile:
    debug(debug_flag, "\nCalling sockets... ...")
    log_command = 'python vol.py -f ' + opt + ' sockets --profile=' + profile
    logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
    proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
    for line in proc.stdout:
        if not "PID" in line:
            if not "---" in line:
                temp_pid = line.split()[1]
                temp_port = line.split()[2]
                sockscan_result[temp_pid] = temp_port

    proc = subprocess.Popen('python vol.py -f ' + opt + ' sockscan --profile=' + profile, stdout=subprocess.PIPE)
    for line in proc.stdout:
        if not "PID" in line:
            if not "---" in line:
                temp_pid = line.split()[1]
                temp_port = line.split()[2]
                for key,var in sockscan_result.items():
                    if temp_pid == key and temp_port == var:
                        pid_match += 1
                if pid_match > 0:
                    pid_match = 0
                else:
                    mismatch_socket_sockscan += 1
                    debug(debug_flag, "Abnormal socket detected " + line)

if mismatch_socket_sockscan > 0:
    malware_score += 30
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.6 Code Injection
debug(debug_flag, "\nCalling ldrmodules... ...")
log_command = 'python vol.py -f ' + opt + ' ldrmodules --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    inload_status = line.split()[3]
    ininit_status = line.split()[4]
    inmem_status = line.split()[5]
    mapped_path = line.split()[-1]
    if inload_status == "True" and ininit_status == "True" and inmem_status == "True":
        if mapped_path == "False":
            abnormal_dll_with_no_path += 1
            debug(debug_flag,"Abnormal dll with no path detected " + line)

if abnormal_dll_with_no_path > 0:
    malware_score += 30
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
```

Author Name, email@address

```
    debug(debug_flag, "Modules total " + str(malware_score_modules))
    malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
    debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")


    #3.3.7 Rootkits (1) Autoruns
    debug(debug_flag, "\nCalling printkey for Run... ...")
    log_command = 'python vol.py -f ' + opt + ' printkey -K "Microsoft\Windows\CurrentVersion\Run" --profile=' +
profile
    logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
    proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
    for line in proc.stdout:
        if "REG_" in line:
            for i in whitelist_runkey_values:
                if i.lower() in line.lower():
                    false_positive_runkey += 1
            if false_positive_runkey > 0:
                false_positive_runkey = 0
            else:
                abnormal_run += 1
                debug(debug_flag, "Abnormal runkey detected " + line)

    debug(debug_flag, "Calling printkey for Runonce... ...")
    log_command = 'python vol.py -f ' + opt + ' printkey -K "Microsoft\Windows\CurrentVersion\Runonce" --profile='
+ profile
    logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
    proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
    for line in proc.stdout:
        if "REG_" in line:
            for i in whitelist_runkey_values:
                if i.lower() in line.lower():
                    false_positive_runkey += 1
            if false_positive_runkey > 0:
                false_positive_runkey = 0
            else:
                abnormal_run += 1
                debug(debug_flag, "Abnormal runkey detected " + line)


    if abnormal_run> 0:
        malware_score += 10
    malware_score_modules += 1

    debug(debug_flag, "malware_score is " + str(malware_score))
    debug(debug_flag, "Modules total " + str(malware_score_modules))
    malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
    debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

    #3.3.7 Rootkits (2) svcscan
    debug(debug_flag, "\nCalling svcscan... ...")
    log_command = 'python vol.py -f ' + opt + ' svcscan --profile=' + profile
    logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
    proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
    for line in proc.stdout:
        if "Process ID:" in line:
            temp_pid = line.split()[-1]
        if "Service State:" in line:
            temp_service_state = line.split()[-1]
        if "Binary Path" in line:
            temp_binary_path = line
            if temp_service_state == "SERVICE_RUNNING":
                for i in whitelist_service_path:
                    if i.lower() in temp_binary_path.lower():
                        false_positive_service += 1
                if false_positive_service > 0:
                    false_positive_service = 0
                else:
                    abnormal_service += 1
                    debug(debug_flag, "Abnormal service found in " + temp_binary_path.lower())
```

```
if abnormal_service > 0:
    malware_score += 7
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.8.1 Interrupt Descriptor Table Hooking
debug(debug_flag, "\nCalling idt... ...")
log_command = 'python vol.py -f ' + opt + ' idt --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if not "Module" in line:
        if not "---" in line:
            if not "ntoskrnl.exe" in line:
                if not line.split()[-1] == "UNKNOWN":
                    for i in whitelist_idt_entries:
                        if i.lower() in line.lower():
                            false_positive_idt += 1
                    if false_positive_idt > 0:
                        false_positive_idt = 0
                    else:
                        abnormal_idt_entry += 1
                        debug(debug_flag,"Abnormal idt detected in " + line)

if abnormal_idt_entry > 0:
    malware_score += 10
malware_score_modules += 1

debug(debug_flag, "idt malware_score is " + str(malware_score))
debug(debug_flag, "malware_score_module is " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score) / (malware_score_modules *10)) * 100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.8.3 ssdt
debug(debug_flag, "\nCalling ssdt... ...")
log_command = 'python vol.py -f ' + opt + ' ssdt --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if "owned by" in line:
        temp_module = line.split()[-1]
        if not temp_module == "ntoskrnl.exe":
            if not temp_module == "win32k.sys":
                for i in whitelist_ssdt_entries:
                    if i.lower() in temp_module.lower():
                        false_positive_ssdt += 1
                if false_positive_ssdt > 0:
                    false_positive_ssdt = 0
                else:
                    abnormal_ssdt_entry += 1
                    debug(debug_flag, "Abnormal ssdt entries detected in " + line)


if abnormal_ssdt_entry > 0:
    malware_score += 10
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.8.4 Driver Hooking (1)log_command =
debug(debug_flag, "Calling modules... ...")
log_command = 'python vol.py -f ' + opt + ' modules --profile=' + profile
```

```
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if not "Offset" in line:
        if not "---" in line:
            temp_module = line.split()[1]
            modules_list.append(temp_module)
            if not "SystemRoot\system32".lower() in line.lower():
                for i in whitelist_driver_entries:
                    if i.lower() in line.lower():
                        false_positive_driver_path += 1
                if false_positive_driver_path > 0:
                    false_positive_driver_path = 0
                else:
                    abnormal_driver_path += 1
                    debug(debug_flag, "Abnormal driver path found " + line)

if abnormal_driver_path > 0:
    malware_score += 10
malware_score_modules += 1


debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.8.4 Driver Hooking (2)
debug(debug_flag, "\nCalling modscan... ...")
log_command = 'python vol.py -f ' + opt + ' modscan --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if not "Offset" in line:
        if not "---" in line:
            temp_module = line.split()[1]
            if not temp_module in modules_list:
                abnormal_module += 1


if abnormal_module > 0:
    malware_score += 10
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

#3.3.8.4 Driver Hooking (3)
debug(debug_flag, "\nCalling unloadedmodules... ...")
log_command = 'python vol.py -f ' + opt + ' unloadedmodules --profile=' + profile
logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
for line in proc.stdout:
    if not "EndAddress" in line:
        if not "---" in line:
            temp_module = line.split()[0]
            if not temp_module in modules_list:
                if not temp_module in whitelist_unloaded_module:
                    abnormal_unloaded_module += 1
                    debug(debug_flag, "Abnormal unloaded module detected " + line)

if abnormal_unloaded_module > 0:
    malware_score += 20
malware_score_modules += 1

debug(debug_flag, "malware_score is " + str(malware_score))
debug(debug_flag, "Modules total " + str(malware_score_modules))
malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
```

```
    debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")

    #3.3.8.5 I/O Request packet Hooking
    debug(debug_flag, "\nCalling driverirp... ...")
    log_command = 'python vol.py -f ' + opt + ' driverirp --profile=' + profile
    logging.write( (str(datetime.datetime.now()) + " : " + log_command + '\n') )
    proc = subprocess.Popen(log_command, stdout=subprocess.PIPE)
    for line in proc.stdout:
        if "DriverName" in line:
            temp_drivername = line.split()[-1].lower()
        if "IRP_MJ" in line:
            temp_device_driver = line.split()[-1].lower()
            if not temp_drivername.lower() in temp_device_driver.lower():
                if not "ntoskrnl" in temp_device_driver:
                    for i in whitelist_driverirp:
                        if i.lower() in temp_device_driver.lower():
                            false_positive_driverirp += 1
                    if false_positive_driverirp > 0:
                        false_positive_driverirp = 0
                    else:
                        abnormal_driver_irp += 1
                        debug(debug_flag, "Abnormal driver irp detected " + line)

    if abnormal_driver_irp > 0:
        malware_score += 10
    malware_score_modules += 1

    debug(debug_flag, "malware_score is " + str(malware_score))
    debug(debug_flag, "Modules total " + str(malware_score_modules))
    malware_infection_percentage = (float(malware_score)/(malware_score_modules *10))*100
    debug(debug_flag, "malware score is " + str(malware_infection_percentage) + "%")


    malware_infection_percentage = (float(malware_score) / (malware_score_modules *10)) * 100
    print "\nThe client is " + str(malware_infection_percentage) + "%" + " likely to be infected"

    logging.close()

if __name__ == "__main__":
    malwarescoretest(sys.argv[1])
```

Author Name, email@address