



Volatility and RegRipper User Manual

Mark Morgan: Mark.Morgan@iarc.nv.gov

702-942-2556

Table of Contents

Introduction	5
Locating the Registry Hives	6
Cell Indexes	6
Management Mechanisms	7
Implementation	8
Guidelines for Investigators	8
Detecting the Cached Data Attack.....	9
Volatility and RegRipper Together at Last	9
Plugins.....	10
IDENT	10
HIVESCAN.....	11
HIVELIST	12
PRINTKEY	13
REGJOBKEYS	14
HASHDUMP.....	14
LSADUMP & HIVEDUMP	15
RegRipper.....	17
Advanced Plugins.....	19
PSLIST.....	19
PSSCAN2.....	21
CONNECTIONS	22
SOCKETS.....	23
FILES.....	24
GETSIDS.....	25
DLLLIST.....	26
PROC_DUMP.....	27
MEM_DUMP.....	28

<i>MODULES.....</i>	<i>28</i>
<i>MODDUMP</i>	<i>28</i>
<i>MODSCAN.....</i>	<i>29</i>
<i>THRDSCAN2</i>	<i>30</i>
<i>VADINFO</i>	<i>31</i>
<i>DRIVERSCAN</i>	<i>32</i>
<i>DRIVERIRP</i>	<i>33</i>
<i>FILEOBJSCAN.....</i>	<i>34</i>
<i>IDT.....</i>	<i>34</i>
<i>KERNEL HOOKS</i>	<i>35</i>
<i>KEYBOARD BUFFER</i>	<i>36</i>
<i>LDR MODULES.....</i>	<i>37</i>
<i>MALFIND2.....</i>	<i>39</i>
<i>MUTANTSCAN.....</i>	<i>40</i>
<i>OBJTYPESCAN</i>	<i>42</i>
<i>ORPHAN THREADS.....</i>	<i>44</i>
<i>SSDT</i>	<i>44</i>
<i>SYMLINKOBJSCAN.....</i>	<i>47</i>
<i>THREAD QUEUES.....</i>	<i>51</i>
<i>API HOOKS</i>	<i>53</i>
<i>CRYPTOSCAN.....</i>	<i>54</i>
<i>VOLATILITY OUTPUT RENDERING FUNCTIONS</i>	<i>54</i>
<i>Plugin Structure</i>	<i>55</i>
<i>Schema.....</i>	<i>56</i>
<i>Installation</i>	<i>57</i>
<i>VOL2HTML PERL SCRIPT</i>	<i>58</i>
<i>VOLATILITY BATCH FILE MAKER.....</i>	<i>59</i>

<i>PRETTY PROCESS MAPPING</i>	61
<i>REFERENCES</i>	64

DRAFT

The Windows registry is a hierarchical database used in the Windows family of operating systems to store information that is necessary to configure the system (Microsoft Corporation, 2008). It is used by Windows applications and the OS itself to store all sorts of information, from simple configuration data to sensitive data such as account passwords and encryption keys. Researchers have found that the registry can also be an important source of forensic evidence when examining Windows systems. Another important yet non-traditional source of forensic data is the contents of volatile memory. By examining the contents of RAM, an investigator can determine a great deal about the state of the machine when the image was collected. Although techniques for analyzing and extracting meaningful information from the raw data found in memory are still relatively new, guidance on the collection of physical memory is now a common part of many forensic best practice documents, such as the NIST Special Publication “*Guide to Integrating Forensic Techniques into Incident Response*” (Kent et al., 2006). This work seeks to bring these two areas of research together by allowing investigators to apply registry analysis techniques to physical memory dumps. It will begin by explaining the structure of the Windows registry as it is represented in memory, and describe techniques for accessing the registry data stored in memory. A prototype implementation of an in-memory registry parser will then be presented, along with some experimental results from several memory images. It will also discuss particular considerations investigators should be aware of when looking at the registry in memory. Finally, it will show that although under normal conditions the stable keys (see Section 3.3 for details on the distinction between stable and volatile keys) recovered from the in-memory copy of the registry are essentially a subset of those found in the on-disk copy, an attacker with access to kernel memory can alter the cached keys and leave those on disk unchanged. The operating system will then make use of the cached data from the registry, and a forensic examination of the disk will not detect the changes. This guide will show how analyzing the registry in memory can detect this attack.

Although the Windows registry appears as a single hierarchy in tools such as “*RegEdit*”, it is actually made up of a number of different binary files called hives on disk. These files and their relationship to the hierarchy normally seen are described in KB256986 (Microsoft Corporation, 2008). The hive files themselves are broken into fixed sized bins of 0 _ 1000 bytes, and each bin contains variable-length cells, which hold the actual registry data. References in hive files are made by cell index, which is essentially a value that can be used to derive the location of the cell containing the referenced data. As for the structure of the registry data itself, it is generally composed of two distinct data types: key nodes and value data. The structure can be thought of as analogous to a file system, where the key nodes play the role of directories and the values act as files. One key difference, however, is that data in the Registry always has an explicit associated type, whereas data on a file system is generally only weakly typed (for example, through a convention such as file extension). To work with registry data in memory, it is necessary to find out where in memory the hives have been loaded and know how to translate cell indexes to memory addresses. It will also be helpful to understand how the Windows Configuration Manager works with the registry internally, and how we can make use of its data structures to tell us what the operating system itself maintains about the state of the registry.

The Configuration Manager in Windows XP references each hive loaded in memory using the `_CMHIVE` data structure. The `_CMHIVE` contains several pieces of metadata about the hive, such as its full path, the number of handles to it that are open, and pointers to the other loaded hives on the system (using the standard `_LIST_ENTRY` data structure used in many Windows kernel structures to form linked lists). It also has another important structure embedded within it, the `_HHIVE`, which contains the mapping table used to translate cell indexes.

This approach to finding hives in memory has two stages. First, scan physical memory to find a single hive; this is easily accomplished, as each `_HHIVE` begins with a constant signature `0_bee0bee0` (a little-endian integer). Furthermore, the structure is allocated from the kernel's paged pool, and has the pool tag `CM10`; these two indicators are sufficient to find valid `_HHIVE`s in all Windows XP images we have examined. Once a single instance has been found, the *HiveList* member is used to locate the others in memory. The pointers to the previous and next hives in the list are virtual addresses in kernel memory space, and must be translated to physical addresses using the page directory of some process. In typical Windows XP SP2 memory images, 13 hives were found: the *NTUSER* and *UsrClass* hives for the currently logged on user, the *LocalService* user, and the *NetworkService* user (total of six hives); the template user hive ("*default*"); the Security Accounts Manager hive ("*SAM*"); the system hive; the *SECURITY* hive; the *SOFTWARE* hive; and, finally, two volatile hives that have no on-disk representation. The two volatile hives deserve some special mention: one, the *HARDWARE* hive, is generated at boot and provides information on the hardware detected in the system. The other, the *REGISTRY* hive, contains only two keys, *MACHINE* and *USER*, which are used to provide a unified namespace in which to attach all other hives.

Cell Indexes

Unlike their layout on disk, hive files in memory need not be contiguous. Moreover, keys and values may be added while the operating system is running, and while it would be inefficient to have to search for free space in the registry file, simply appending a new bin to the end of the hive would quickly cause the hive to grow to an unmanageable size. To solve this, the Configuration Manager creates a mapping between cell indexes and addresses in virtual memory, in much the same way that a process gets a map between its virtual addresses and the physical address space of main memory (*Russinovich and Solomon, 2004*). The map for a given hive is stored in the `_HHIVE` structure's *Storage* member, and once located; it can be used to give us full access to the registry data stored in memory. To perform the translation, the cell index is broken into four parts: a one bit selector saying whether the cell index's main storage is stable (on-disk) or volatile (only in memory), 10 bits giving the index into the list of hive address tables, 9 bits that select a single entry from that table, and finally a 12-bit offset into the block given by that table entry. Now, given the address of the table directory and a cell index, we can translate that cell index into a virtual address in kernel memory (*Dolan-Gavitt, 2008a*). Once this

translation can be performed, reading the registry is relatively straightforward, as the data structures are identical to those used for the on disk. Several open source programs exist that can read binary registry data, such as Samba's regfio ([Samba](#)) or the Perl module Parse::Win32Registry ([Macfarlane](#)), and the data structures and algorithms from these tools can be applied directly to the task of reading registry data in memory with only small modification to use the in-memory translation method for cell indexes. To walk the entire hive, start at the root key (always at cell index 0 _ 20), and then walk the sub keys just as we would an on-disk hive. One crucial difference, however, is the existence of volatile keys and values in in-memory hives. The `_CM_KEY_NODE` structure has two members, **SubKeyCounts** and **SubKey-Lists** that give the number of sub keys and a pointer to the sub key list, respectively. Each member, however, is actually an array of length two: the first entry in the array refers to the stable keys, while the second refers to the volatile keys. Most existing implementations of Windows registry parsers, such as Samba's regfio library ([Samba](#)) and the Perl Parse::Win32Registry module ([Macfarlane](#)), do not handle volatile sub key lists, and describe those portions of the key structure as "unknown." The registry implementation in [ReactOS](#) handles volatile keys correctly, however. These volatile keys are never stored on disk, and are automatically generated by the operating system when the machine is booted. Examples of information stored in these keys include an enumeration of all hardware detected on the system, the volatile portions of a user's environment, mounted volumes, and the current machine name. Although it is possible to access this information while the system is booted using live response techniques, it cannot be recovered using the on-disk hives from an image of the system. Using these techniques, however, an investigator will be able to access the volatile keys and values in a stable, repeatable way by examining a dump of physical memory.

Management Mechanisms

In addition to simply extracting as many keys and values from the cached copy of the registry in memory as possible, an investigator might wish to gain an understanding of what data the Configuration Manager was working with. For example, the examiner might wish to know what keys were open on the system, and how many things were referencing them. To answer these questions, one can make use of several data structures used by the Configuration Manager to provide fast access to currently open keys. When an application attempts to open a key, the Configuration Manager must be able to quickly determine if the key is already open, and if so, return a handle to the same object, to ensure that all applications are always referencing the same data. To accomplish this, each open key, as well as all of its ancestors, has associated with it a Key Control Block (data structure `_CM_KEY_CONTROL_BLOCK`), or KCB, that keeps track of its reference count, last write time, and cell index. The different handles that each process gets will then all point to the same KCB. To satisfy the requirement that finding a KCB for a given key be fast, the Configuration Manager uses a hash table to keep track of all the KCBs. The address of this table is given by the kernel global variable `CmpCacheTable` and its size by `CmpHashTableSize`; the address of these variables can be found either by using the debug symbols for the kernel that was loaded in the memory image, or by searching through the mapped kernel image and using some heuristic to validate whether a given address is a pointer to the hash table. Each entry is a pointer to the `KeyHash` member of a Key Control Block, which is of type `_CM_KEY_HASH`. Entries in the hash table can point

to more than one Key Control Block, and the full list for a given table entry can be found by repeatedly following the NextHash pointer in the key hash structure ([Dolan-Gavitt, 2008c](#)).

Implementation

The techniques described above are implemented as a plugin for Volatility ([Walters, 2007](#)). This allows for the use of the pre-existing libraries to do virtual address translation, process listing, and so on, as well as easily define the data structures needed to parse registry data (through the data model exposed in `vtypes.py`). Each hive is represented as its own virtual address space; the cell indexes described earlier are treated as memory addresses within the space of the hive. Thus when you want to find the root of a hive, it suffices to get the `_CM_KEY_NODE` structure at hive address 0 _ 20; the hive address space object then handles translating that cell index into a virtual address, and the virtual address space object in turn translates the virtual address into a physical offset in the memory dump. A small library of functions has been developed that handles the most common tasks associated with reading information from registry hives: reading keys and values, opening a key given its full path, getting the root of the hive, and so on. Using these functions, other researchers should be able to easily create their own plugins to extract and Interpret portions of the registry that have forensic relevance, such as the *UserAssist* keys ([Stevens, 2006](#)). Plugins have also been developed to automate the process of finding hives and examining the Key Control Blocks used by the Configuration Manager.

Guidelines for Investigators

In general, any standard forensic methods for examining registry data can be applied to the registry data in memory. However, there are certain caveats that apply uniquely to the examination of the data cached in RAM. Most importantly, it cannot be assumed that the data found in memory is complete. In addition to the usual consideration that the contents of main memory may be swapped out to the page file, it is also possible that parts of the registry may have never been brought into memory in the first place. In this case, its entry in the cell map table will be set to zero, and the data is unlikely to be found in memory. The most immediate consequence of this is that tools that deal with the registry from memory must be robust and able to handle missing data without crashing. Initial attempts to work with registry data stored in memory involved extracting the hive from memory and saving it to disk, writing NULLs when data from memory was missing. However, existing tools were unable to deal with the missing sections, and would crash or display incomplete output, and a new parser was written that was able to detect invalid structures and ignore them. Because not all of the registry will necessarily be in memory, the data recovered from RAM should be examined alongside the on-disk hives. This will allow the investigator to get a complete picture of the registry: every key and value will be available on disk, and data that has not yet been written to disk can also be recovered from memory. In

addition, it is possible for an attacker to alter the data stored in the hive in memory without modifying the copy on disk; however, comparing the two views will reveal this activity.

Detecting the Cached Data Attack

The so-called “stable” registry data in memory is generally a subset of what can be found on disk, due to the fact that any modified keys or values are written out to disk every five seconds by default (*Russinovich and Solomon, 2004*). However, we have found that it is possible for an attacker with the ability to modify kernel memory to alter the cached registry data in memory, and thus alter the behavior of the operating system, without the changes being visible in the on-disk storage. For example, an attacker could find the key in memory that holds the password hashes for the Administrator user, and replace them with pre-computed hashes for a known password. The attacker would then be able to log in as Administrator using the password of his choice. This attack has been tested using *WinDbg* on a VMware virtual machine (though it could be accomplished fairly easily by any piece of code with access to kernel memory). First, the virtual memory address of the key ***SAM\Domains\Account\Users\000001f0*** was located, and determined the address of its “V” value (this is the data value that contains the password hashes for the Administrator account). Then, the ***eb*** command was used to overwrite the hashes with pre-computed LanMan and NT hashes for the password “***foobar***”. After logging out of the account to allow the new value to be read by the logon process, it was possible to log in with the password “***foobar***” through both the standard Windows login screen and Remote Desktop. The system was then used normally for the next 15 minutes (ample time for the Configuration Manager’s hive flush mechanism to take effect), created a memory dump by pausing the VM and copying its vmem file, and rebooted the virtual machine to verify that the new hashes had not been saved. As expected, the hashes had not been written out to the disk hive, and the original password was once again in effect. The changes were not flushed to the hive on disk because the modification was made without using the Configuration Manager’s normal mechanisms, which update a list of “dirty” bins when registry data is written and schedule a hive flush (*Russinovich and Solomon, 2004*). To verify that it was possible to detect this modification, we then extracted the cached value of the “V” key from the saved memory dump, and compared it with the version on disk, and thus detected the inconsistency. An examiner looking only at the on-disk hive would have found nothing amiss in this situation.

Volatility and RegRipper Together at Last

This document is the 3rd part of installing and using RegRipper and Volatility together to parse through memory image created during an intrusion investigation. All this information was ripped from papers written by smarter people than me on how to use these tools. This document is a version of how to use these tools and in what order based on many hours of reading through books and white papers on the subject. This paper will be describing the syntax using a Linux box so if you are using Windows then replace the image location with the

appropriate syntax for Windows. (ie., **c:\image\test.001**) Also if for any reason your version of volatility does not work then enter the following into a terminal console:

svn checkout <http://volatility.googlecode.com/svn/volatility>

Make sure you are in your home directory because this will download the latest version of volatility which is Volatility-1.3.2. (This is the version that was used for this document) This version is a complete re-write of the previous version and works a little different so if you do use this one make sure you read the help files with each command. (i.e., **python volatility ident -h**) This syntax will bring up the help for each plugin.

Plugins

The first steps in processing any crime scene is taking photographs of the scene to document what is where. So the first step in processing memory is creating an image of the memory and there are multitudes of software that can do this from **Encase**, **FTK** and other miscellaneous tools that create the image in raw format which is the required format for these tools. It is suggested is to use **win32dd**, **FTK**, **Memoryze**, or **dd**. Once you have the image you will need to identify the image by using the volatility plugin "**ident**." The syntax is as follows:

./volatility ident -f /media/usb/test.001

IDENT

This will give you a similar result as below:

morgan@morgan-laptop:/digitalforensics/Volatility-1.3_Beta\$./volatility ident -f /home/morgan/Memory\ Images/PhysicalMemory.bin	
Image Name:	/home/morgan/Memory Images/PhysicalMemory.bin
Image Type:	Service Pack 3
VM Type:	pae
DTB:	0x33e000
Datetime:	Tue Aug 04 11:02:35 2009

As you can see you have the image name, service pack, type of image, data/time image was created and the hex offset (physical address) of the directory table base.

Now from here there are a lot of directions you could probably go in but based on some of my reading most of the experts in Forensics say you would want to look at the registry entries to determine what is opened and so forth. But a lot of this can also depend upon what you know thus far into the investigation. This guide was written as if you are blind going into the examination. It is suggested that you first run the **Hivescan** which finds hive offsets in memory images. This is not very useful by itself but the information is needed to be used with other plugins. So let see what you get by running the following:

```
./volatility hivescan -f /media/usb/test.001
```

HIVESCAN

This will give you a similar result as below:

```
morgan@morgan-laptop:/digitalforensics/Volatility-1.3_Beta$ ./volatility hivescan -f  
/home/morgan/Memory\ Images/PhysicalMemory.bin
```

Offset	(hex)
181006344	0xac9f008
181033824	0xaca5b60
189972488	0xb52c008
202671368	0xc148508
544586592	0x2075bb60
642878304	0x26518b60
643895304	0x26611008
678736920	0x2874b418
740933640	0x2c29c008
742706016	0x2c44cb60
789179232	0x2f09eb60
798029088	0x2f90f520
1107776776	0x42075508
1874516240	0x6fbad910

Once you get one hive offset then it will give you all other hive offsets in memory. Make sure you keep this list around as you will need it to run other plugins. You can re-run the above syntax to include an export by using the **> /home/morgan/results/result.txt**.

Next use the highlighted hex value on the 1st offset as follows:

`./volatility hivelist -f /media/usb/test.001 -o 0xac9f008`

HIVELIST

This will produce the following:

```
morgan@morgan-laptop:/digitalforensics/Volatility-1.3_Beta$ ./volatility hivelist -f
/home/morgan/Memory\ Images/PhysicalMemory.bin -o 0xac9f008
```

Address	Name
0xe6348910	\Documents and Settings\144553\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
0xebe6e508	\Documents and Settings\144553\NTUSER.DAT
0xe8287508	\WINDOWS\system32\config\systemprofile\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
0xe1895520	\Documents and Settings\LocalService\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
0xe1882b60	\Documents and Settings\LocalService\NTUSER.DAT
0xe1396008	\Documents and Settings\NetworkService\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
0xe139ab60	\Documents and Settings\NetworkService\NTUSER.DAT
0xe4f8eb60	\WINDOWS\system32\config\SAM
0xe77b9b60	\WINDOWS\system32\config\SECURITY
0xe77cd008	\WINDOWS\system32\config\SOFTWARE
0xe77ca418	\WINDOWS\system32\config\DEFAULT
0xe18b6008	[no name]
0xe1035b60	\WINDOWS\system32\config\SYSTEM
0xe102e008	[no name]

This result gives you the hive list along with the physical offset of each. With this information you can now use **printkey** to view the keys, subkeys and values associated with each key. It will also show the last modified time, formats and values according to type and also includes any **volatile** keys and values.

It is also good for just looking around. Now let's see what you get when you run **printkey** against the **SAM** and **SECURITY** hives.

PRINTKEY

```
root@morgan-laptop:/digitalforensics/Volatility-1.3_Beta$ ./volatility printkey -f  
/home/morgan/Memory\ Images/PhysicalMemory.bin -o 0xe1035b60
```

Key name: \$\$\$PROTO.HIV (Stable)

Last updated: Mon Jul 4 18:16:59 2005

Subkeys:

ControlSet001 (Stable)

ControlSet002 (Stable)

LastKnownGoodRecovery (Stable)

MountedDevices (Stable)

Select (Stable)

Setup (Stable)

WPA (Stable)

CurrentControlSet (Volatile)

Values:

As you can see there is some **volatile** data under **CurrentControlSet** so now you want to take a look at that by entering the following into the console terminal:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility.py printkey -f /home/morgan/Memory\  
Images/PhysicalMemory.bin -o 0xe1035b60 -k CurrentControlSet
```

Volatile Systems Volatility Framework 1.3.2

Key name: CurrentControlSet

(Volatile)

Last updated: 2009-07-29 09:08:26

Subkeys:

Values:

REG_LINK SymbolicLinkValue : \Registry\Machine\System\ControlSet001 (Volatile)

The above output identifies the link to the volatile data and shows the last time it was updated. In this specific case, the last update above matched the date the intrusion occurred. This would be a good place to start looking for malware.

REGJOBKEYS

At this point you would probably want to run the “*regobjkeys*” plugin to determine what registry hives are currently opened in memory. This will give you the registry hives by PID number and once you identify a hive of interest then you can run the “*files*” plugin and insert the appropriate PID number for that hive. The syntax would be as follows for each:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility.py regobjkeys -f
/home/morgan/Memory\ Images/PhysicalMemory.bin
```

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility.py files -f /home/morgan/Memory\
Images/PhysicalMemory.bin -p 1564
```

This information is too large for insertion into this document so you will have to play with this plugin to get a good feel for it.

HASHDUMP

The next thing you should do is dump all the hash information out of memory so you can see what accounts were sitting in memory. Local account password hashes are stored in the registry (encrypted) and hashdump module decrypts and prints these hashes. If *LanMan* is being used then the rainbow tables can recover passwords in minutes. So to do this you will need the offset in HEX for the *SYSTEM* and *SAM* files so enter the following syntax:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility.py hashdump -f
/home/morgan/Memory\ Images/PhysicalMemory.bin -y 0xe1035b60 -s 0xe4f8eb60
```

```
Volatile Systems Volatility Framework 1.3.2
```

```
Renamed_admin:500:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
```

```
Renamed_guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
```

SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:f7bbc74ec99bf11963c4d5ffbfd9d373:::
HelpAssistant:1004:542dc2c46a6d9a31d14b566985585172:4b5e9c918ffe8031dadd9bc2c9ebab5f:::
XXXXXX:1005:aad3b435b51404eeaad3b435b51404ee:ee2d2ac787e57f27ee34a1b03d7bd678:::

The 1st offset is the **SYSTEM** hive and the 2nd one is for the **SAM** hive. You can see there are a few account names that look a little suspicious so you would need to get with the system administrator to determine the naming convention used to determine if the above accounts were real or were possibly created by an intruder. (XXXX represents a user name that was sanitized for obvious reasons)

From here you can use **Rainbow Tables** or **John the Ripper** to crack the passwords and proceed from there.

LSADUMP & HIVEDUMP

There are two other dump plugins available for use and they are **lsadump** and **hivedump**. The plugin **lsadump** dumps LSA protected storage found in the registry. These areas can contain passwords and it is always a good idea to take a quick look. For this plugin you will need the HEX offset for the **SYSTEM** and **SECURITY** hives which you should have saved from the [hivelist](#) you did earlier.

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility.py lsadump -f /home/morgan/Memory\
Images/PhysicalMemory.bin -y 0xe1035b60 -s 0xe77b9b60
```

Volatile Systems Volatility Framework 1.3.2

L\$HYDRAENCKEY_28ada6da-d622-11d1-9cb9-00c04fb16e75

0000 52 53 41 32 48 00 00 00 00 02 00 00 3F 00 00 00 RSA2H.....?...

0010 01 00 01 00 69 B4 94 B5 75 C8 E4 56 D2 22 42 B0i...u..V."B.

0020 BD AA 3C 25 1B 25 2D 5B 98 80 A8 E9 1D FE 15 07 ..<%.%-[.....

0030 62 0E F6 32 AD 5D 68 CD 4C 9F 43 D7 27 98 1D 7E b..2.]h.L.C.'..~

0040 05 4B E0 C0 64 EB 38 7C 20 12 D8 D5 D8 C8 34 B6 .K..d.8|4.

0050 07 5B 64 C8 00 00 00 00 00 00 00 00 B1 EB FD EF .[d.....

0060 F0 5D C6 9E 5E 3C 80 71 4C 85 2B F8 9F 31 95 6F .]..^<.qL.+..1.o

0070 94 C8 66 E9 24 22 7F FA 1A 65 AD E9 00 00 00 00 ..f.\$"...e.....

0080 39 5A DF 52 62 4C A6 A8 9E 0F CF F9 7F F7 FE 46 9Z.RbL.....F

0090 27 0B 09 86 28 76 03 C8 C0 71 37 A1 F4 F6 88 DB '...(v...q7.....

00A0 00 00 00 00 D1 B0 0A 50 0E 95 7E 74 25 61 96 01P..~t%a..

00B0	C3 3F 32 BC 12 6F 68 E8 E4 F9 B4 58 4A 22 8D 03	.?2..oh....XJ"..
00C0	A5 10 D6 3E 00 00 00 00 A9 56 A1 50 2C CB FD DC	...>.....V.P,...
00D0	27 C7 F2 B1 51 B0 DB 6F 22 8B 98 35 81 E0 4E D5	'...Q..o"..5..N.
00E0	AD 48 BB 81 57 9D 14 96 00 00 00 00 68 52 C6 12	.H..W.....hR..
00F0	72 E4 6D B1 98 70 A3 DA D9 DC EA 77 E7 46 E5 89	r.m..p.....w.F..
0100	15 98 10 9C 02 D6 08 83 2D D0 57 E6 00 00 00 00-.W.....
0110	81 7F 8D 97 4E A1 09 19 DF 45 6B DE 52 26 08 16N....Ek.R&..
0120	17 D1 FA D2 01 35 4B 61 DC E9 AE C6 44 35 90 825Ka....D5..
0130	AC 52 AE 32 AA 32 25 E4 18 C1 30 A1 88 50 6F A7	.R.2.2%...0..Po.
0140	DF B1 BD A7 E0 41 BC C9 09 82 6E A9 10 DE 0C 51A.....n....Q
0150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

This information is pretty raw so you will just have to look and see if you found anything. In this case we did not. The last dump plugin is for dumping the entire registry to a **.csv** file but since this could take a while we will only be dumping the **SAM** file.

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility.py hivedump -f
/home/morgan/Memory/Images/PhysicalMemory.bin -o 0xe4f8eb60 --output-file=sam.csv
```

Remember the **csv** file will be placed in the root of your volatility folder unless you re-direct it to another location.

From here it is suggested that you begin reviewing what registry hives are sitting in memory. Remember there are some registry hives that are in memory that will never be written back to the **SYSTEM** hives even on a re-boot or shutdown. This is why memory can become very important. This is why it is suggested that you use **RegRipper** with **Volatility** which when combined gives you the ability to point Perl scripts against the memory image and extract the hives you are looking for. In order to do this you need to have the plugins "**Volrip**" and "**Volreg**" installed and this procedure only works on Linux box so throw your Windows machine out the window for now. You will also need to install the python module "**Inline: Python**" in order for this to work properly. There are other documents already in existence that walks you through the installation of **Volatility** and **RegRipper** on a Linux platform.

RegRipper

Now at this point it is assumed that everything is installed properly and you are ready to proceed. The first thing you might want to do is to take a quick look at the **system** hives with the defaulted plugins that come with **RegRipper**. You do this by entering the following syntax in the console terminal:

```
perl rip.pl -r /home/morgan/Memory Images/PhysicalImage.bin -o 0xe1035b60 -f system
```

Remember to look at your results you got from [HiveList](#) in the above previous steps to get the Offset needed for the **SYSTEM** hive. The result will be similar to the below listed items but this is not inclusive because of the size. You will most likely want to direct this out to a text file or some other format of your choice to review. To export the results to a text file use the following syntax:

```
perl rip.pl -r /home/morgan/Memory Images/PhysicalImage.bin -o 0xe1035b60 -f system > /home/morgan/results/system.txt
```

\\Volume{413e42c1-dbd4-11da-a8f2-806d6172696f}
Drive Signature = 16 23 ab 41
\\DosDevices\\C:
Drive Signature = 16 23 ab 41
\\Volume{93a205b2-225c-11db-a90b-00166f67ad84}
Drive Signature = 4e 33 46 5a
\\DosDevices\\E:
Drive Signature = 73 3e a8 c0
\\DosDevices\\F:
Drive Signature = 5b 6c 39 26
\\Volume{c6fc7d14-924e-11dc-ac71-001839973fb4}
Drive Signature = 5b 6c 39 26
Device: \\Volume{STORAGE#RemovableMedia#7&5d9da52&0&RM#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}}
\\Volume{cde9fc6a-149b-11de-ad12-00059a3c7800}
Device: \\Volume{USBSTOR#CdRom&Ven_HL-DT-ST&Prod_CDRW#DVD_GCC4244&Rev_B101#7&2dd9639d&0#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}}
\\Volume{8054d6c1-4724-11db-ac28-00059a3c7800}
Device: \\Volume{weo9rvojtQYd^UE\$)6Ac{c/wbY9''2R#^]Ñ#Æxú#oF#[v
ù#ez#Kv`2ÿ~¾ZD

ù#œUI8Dx [Z]M kp2ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#ŸWB/ @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez#OpfŸ~¼Zt:È: [Z]C!O##4Ÿ~¼Zu3Ë,ŸSB8Mn`Ÿ~¼Z'0¥uÇ
#y
Ê?)9Ë^#dR\$=IŸ~¼ZD 3
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼ZD
ù#®ez# @QŸ~¼Z
\??\Volume{622b0e69-8136-22de-09aa-114066a1010a}

The above is only of the **Mounted Devices** and is intended to show you what type of information you need to look for. If you look at the last entry on the Volume highlighted in yellow, you will note that the binary information associated with this Volume has been obfuscated. This is actually the source code (instructions) for a piece of malware that was put on the system. When decrypted it will tell you what the malware is named, its location, and the IP address the malware will communicate with. Obviously most information will not be this easy to identify but this gives you an idea that you should be looking for information and to identify information that looks unusual or doesn't seem to fit.

Once you have done this for the **SYSTEM** hive then you can run it against all the hives but you will need to get the correct offset from the [hivelist](#) scan. You can also request individual keys from the hives by telling **RegRipper** which plugin you want to use. Checkout the subdirectory “**rrplugins**” as this is where all the plugins are located. For instance if you want to see just the **MountedDevices** then your syntax would look like this:

```
perl rip.pl -r /home/morgan/Memory Images/PhysicalImage.bin -o 0xe1035b60 -f mountdev
```

If you look at the files **SYSTEM**, **SOFTWARE**, **SECURITY** and **SAM** in the plugins folder (these are the ones without extensions) they will tell you which plugins goes to which hive.

Now that you are done with this part, it is time to put the remaining plugins in some type of logical order instead of just arbitrarily picking the plugins without knowing what you are going to get.

Advanced Plugins

PSLIST

If you want to see what was running on the system you will want to take a look at the processes. The syntax is as follows:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility pslist -f /home/morgan/Memory\
Images/PhysicalMemory.bin
```

Name	Pid	PPid	Thds	Hnds	Time
System	4	0	70	798	Thu Jan 01 00:00:00 1970
smss.exe	180	4	3	19	Wed Jul 29 09:08:56 2009
csrss.exe	524	180	13	837	Wed Jul 29 09:09:00 2009
winlogon.exe	548	180	22	640	Wed Jul 29 09:09:02 2009
services.exe	592	548	15	364	Wed Jul 29 09:09:04 2009
lsass.exe	604	548	24	512	Wed Jul 29 09:09:04 2009
svchost.exe	840	592	17	224	Wed Jul 29 09:09:08 2009
svchost.exe	988	592	10	518	Wed Jul 29 09:09:09 2009
svchost.exe	1352	592	77	2685	Wed Jul 29 09:09:09 2009
EvtEng.exe	1448	592	8	126	Wed Jul 29 09:09:10 2009
S24EvMon.exe	1500	592	6	128	Wed Jul 29 09:09:11 2009
WLKEEPER.exe	1564	592	5	114	Wed Jul 29 09:09:11 2009
Smc.exe	1864	592	37	643	Wed Jul 29 09:09:11 2009

svchost.exe	484	592	6	85	Wed Jul 29 09:09:15 2009
svchost.exe	1044	592	7	137	Wed Jul 29 09:09:15 2009
SNAC.EXE	1388	592	13	360	Wed Jul 29 09:09:16 2009
ccSvcHst.exe	272	592	33	506	Wed Jul 29 09:09:17 2009
spoolsv.exe	1584	592	15	181	Wed Jul 29 09:09:20 2009
scardsvr.exe	1900	592	9	95	Wed Jul 29 09:09:20 2009
svchost.exe	332	592	4	102	Wed Jul 29 09:09:26 2009
BAsfIpM.exe	880	592	8	109	Wed Jul 29 09:09:26 2009
cvpnd.exe	1460	592	6	200	Wed Jul 29 09:09:27 2009
etlsvr.exe	400	592	8	112	Wed Jul 29 09:09:28 2009
lap.exe	896	592	6	48744	Wed Jul 29 09:09:28 2009
jqs.exe	1228	592	5	129	Wed Jul 29 09:09:28 2009
LxrSII1s.exe	1312	592	2	24	Wed Jul 29 09:09:29 2009
MDM.EXE	1424	592	4	86	Wed Jul 29 09:09:29 2009
NicConfigSvc.ex	2624	592	4	110	Wed Jul 29 09:09:32 2009
RegSvc.exe	2700	592	3	77	Wed Jul 29 09:09:32 2009
svchost.exe	2792	592	5	117	Wed Jul 29 09:09:32 2009
Rtvsan.exe	3088	592	37	620	Wed Jul 29 09:09:33 2009
Wuser32.exe	3276	592	10	113	Wed Jul 29 09:09:34 2009
CcmExec.exe	3436	592	13	754	Wed Jul 29 09:09:35 2009
wmiprvse.exe	2348	840	17	317	Wed Jul 29 09:09:39 2009
alg.exe	3044	592	5	104	Wed Jul 29 09:09:40 2009
wmiprvse.exe	2188	840	6	145	Wed Jul 29 09:09:42 2009
wmiprvse.exe	2172	840	6	153	Wed Jul 29 09:09:56 2009
COH32.exe	416	3088	0	-1	Wed Jul 29 10:10:57 2009
gsalrt.exe	1960	592	6	150	Mon Aug 03 14:25:52 2009
ssonsvr.exe	4052	3064	1	28	Tue Aug 04 12:33:59 2009
ZCfgSvc.exe	2912	548	5	154	Tue Aug 04 12:34:04 2009
explorer.exe	3472	2824	14	766	Tue Aug 04 12:34:06 2009
1XConfig.exe	3924	840	9	5644	Tue Aug 04 12:34:07 2009
SmcGui.exe	2508	1864	18	357	Tue Aug 04 12:34:11 2009
Apoint.exe	2448	3472	2	137	Tue Aug 04 12:34:26 2009
hkcmd.exe	376	3472	2	86	Tue Aug 04 12:34:26 2009
igfxpers.exe	2128	3472	3	102	Tue Aug 04 12:34:26 2009
iFrmewrk.exe	1892	3472	4	148	Tue Aug 04 12:34:27 2009

igfxsrvc.exe	1468	840	3	113	Tue Aug 04 12:34:27 2009
quickset.exe	3232	3472	5	225	Tue Aug 04 12:34:28 2009
DVDLauncher.exe	2868	3472	2	46	Tue Aug 04 12:34:28 2009
GoogleDesktop.e	2776	3472	3	83	Tue Aug 04 12:34:30 2009
tfswctrl.exe	748	3472	3	98	Tue Aug 04 12:34:30 2009
WPC54GX4.exe	3868	3472	8	703	Tue Aug 04 12:34:32 2009
jusched.exe	1992	3472	1	125	Tue Aug 04 12:34:34 2009
hpwuSchd2.exe	448	3472	1	28	Tue Aug 04 12:34:35 2009
acrotray.exe	3796	3472	1	36	Tue Aug 04 12:34:36 2009
ccApp.exe	2824	3472	14	293	Tue Aug 04 12:34:37 2009
ctfmon.exe	2900	3472	1	98	Tue Aug 04 12:34:38 2009

This output also gives you the PID for each process. That way if you see a process that you are not familiar with you can check it out. For instance, if you want to see what “svchost” is running then you would input that PID number with another plugin in the list that will allow you to look at each PID individually. We will touch more on this process later on.

PSSCAN2

Before moving on let me mention another plugin called “psscanner2” that will give you the same results but with additional information. You should run both plugins to ensure you get all the information you need. Below is a table that shows you what the psscanner2 output looks like:

PID	PPID	Time created	Time exited	Offset	PDB	Remarks
2900	3472	Tue Aug 04 12:34:38 2009		0x06bd6918	0x24d40820	ctfmon.exe
480	3472	Tue Aug 04 12:34:40 2009		0x06c2fda0	0x24d40840	LxrAutorun.exe
2508	1864	Tue Aug 04 12:34:11 2009		0x06d0aae8	0x24d40640	SmcGui.exe
3796	3472	Tue Aug 04 12:34:36 2009		0x06dca9e0	0x24d407c0	acrotray.exe
3152	2776	Tue Aug 04 12:34:53 2009		0x06f81af8	0x24d40900	GoogleDesktopDi
448	3472	Tue Aug 04 12:34:35 2009		0x07088678	0x24d407a0	hpwuSchd2.exe
2776	3472	Tue Aug 04 12:34:30 2009		0x07090790	0x24d40720	GoogleDesktop.e
2156	3472	Tue Aug 04 12:34:49 2009		0x071097c0	0x24d408e0	etlitr50.exe
2056	2448	Tue Aug 04 12:34:58 2009		0x07164da0	0x24d409a0	hidfind.exe
1696	1992	Tue Aug 04 12:39:35 2009		0x07191c98	0x24d40560	jucheck.exe

2736	2776	Tue Aug 04 12:34:48 2009	0x071988c0 0x24d408c0 GoogleDesktopIn
2872	3472	Tue Aug 04 12:34:55 2009	0x071f6da0 0x24d40940 WZQKPICK.EXE
1892	3472	Tue Aug 04 12:34:27 2009	0x072e8a10 0x24d40660 iFrmewrk.exe
3868	3472	Tue Aug 04 12:34:32 2009	0x0735d168 0x24d40540 WPC54GX4.exe
2008	2960	Tue Aug 04 12:34:58 2009	0x07379880 0x24d409c0 ApntEx.exe
2912	548	Tue Aug 04 12:34:04 2009	0x074205a8 0x24d40580 ZCfgSvc.exe

CONNECTIONS

Now that you have the above information and you find something of interest, you might want to see if any connections are being made between the infected system and a remote location. This is done by typing the following syntax:

`./volatility connections -f /home/morgan/Raw\ Memory/PhysicalMemory.bin`

Local Address	Remote Address	Pid
xxx.xxx.xxx.88:4412	xxx.xxx.xxx.2:8080	1696
xxx.xxx.xxx.88:947	xxx.xxx.xxx.7:4105	1960
xxx.xxx.xxx.88:4411	xxx.xxx.xxx.217:80	1696
xxx.xxx.xxx.88:947	xxx.xxx.xxx.88:3462	1960
xxx.xxx.xxx.88:947	xxx.xxx.xxx.7:4103	1960
xxx.xxx.xxx.88:947	xxx.xxx.xxx.88:3447	1960

In this case there are no unknown transfers being done as all the above connections are authorized by the administrator as remote connections [**validated by the system administrator**]. Before moving on just be aware of another plugin called “**connscan2**.” This plugin searches for **POOL_HEADER** and is much faster. The information is the same as the “**connections**” plugin.

Let's determine if anything is listening that we are not aware of. You can do this by looking for any created sockets.

morgan-laptop:/digitalforensics/Volatility-1.3.2\$./volatility sockscan2 -f /home/morgan/Raw\ Memory/PhysicalMemory.bin

PID	Port	Proto	Create Time	Offset
2736	4664	6	Tue Aug 04 12:34:52 2009	0x07489a88
1960	947	6	Mon Aug 03 14:25:53 2009	0x0756aca0
1696	4412	6	Tue Aug 04 12:39:35 2009	0x07ce77c8
1352	68	17	Tue Aug 04 17:02:38 2009	0x07d55c90
1460	62514	6	Wed Jul 29 09:10:06 2009	0x07fba008
1584	1037	17	Wed Jul 29 09:09:36 2009	0x07fc0a70
3276	2702	6	Wed Jul 29 09:09:35 2009	0x07fe5790
604	4500	17	Wed Jul 29 09:09:32 2009	0x07ff0e98
4	1038	6	Wed Jul 29 09:09:37 2009	0x07ffa008
1864	1036	17	Wed Jul 29 09:09:29 2009	0x08000370
1228	5152	6	Wed Jul 29 09:09:29 2009	0x08074630
1352	123	17	Wed Jul 29 09:10:20 2009	0x08abea98
4	137	17	Wed Jul 29 09:10:20 2009	0x08bfb520
3044	1051	6	Wed Jul 29 09:09:40 2009	0x08c03488
3436	1058	17	Wed Jul 29 09:10:07 2009	0x08c06328
460	62514	17	Wed Jul 29 09:10:06 2009	0x08c2f008
4	0	47	Wed Jul 29 09:09:39 2009	0x08c3e008
548	1042	17	Wed Jul 29 09:09:39 2009	0x08c44860
3276	2701	6	Wed Jul 29 09:09:34 2009	0x08c6e650
1352	123	17	Wed Jul 29 09:10:20 2009	0x08c74008
4	139	6	Wed Jul 29 09:10:20 2009	0x08c90e98
1352	123	17	Wed Jul 29 09:10:20 2009	0x08cb7bf8
604	500	17	Wed Jul 29 09:09:32 2009	0x08cbdcc0
604	0	255	Wed Jul 29 09:09:32 2009	0x08cdce98
604	1025	17	Wed Jul 29 09:09:27 2009	0x08ce48f8

4	138	17	Wed Jul 29 09:10:20 2009	0x08eb48e8
1388	39999	17	Wed Jul 29 09:09:17 2009	0x0a134a90
4	138	17	Wed Jul 29 09:08:56 2009	0x0a322ac8
4	445	17	Wed Jul 29 09:08:56 2009	0x0a367ac8
4	445	6	Wed Jul 29 09:08:56 2009	0x0a492728
988	135	6	Wed Jul 29 09:09:09 2009	0x0a531bd8
1696	4411	6	Tue Aug 04 12:39:35 2009	0x0a5c3198

You should also note that you will find two plugins that will give you the above information: “**sockscan**” and “**sockscan2**”, the later being the new and improved version. From here you can look further into the PID to determine where the process is writing. For instance, let’s take a closer look at **PID 1696, Port 4411, and Protocol 6** (for more information on protocols reference)

<http://www.networksorcery.com/enp/protocol/ip.htm#Protocol>).

FILES

This plugin will list open files for each process. Once you identify a PID that you want to check out, you can see what files are opened. Run the following command:

```
morgan-laptop:/digitalforensics/Volatility-1.3.2$ ./volatility files -f /home/morgan/Raw\
Memory/PhysicalMemory.bin -p 1696
```

Pid: 1696

File \Documents and Settings\144553

File \WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83

File \WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83

File \WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83

File \Documents and Settings\144553\Local Settings\Temporary Internet Files\Content.IE5\index.dat

File \Documents and Settings\144553\Cookies\index.dat

File \Documents and Settings\144553\Local Settings\History\History.IE5\index.dat

File \WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83

File \ROUTER

File \ROUTER

File \Endpoint

File \AsyncConnectHlp
File \Endpoint

GETSIDS

During the course of an examination, it may be critical to be able to link up a process that's running to a particular user account. Particularly in a multi-user environment such as Windows Terminal Server, this isn't always as easy as checking who was logged in at the time.

Luckily, each process in Windows has an associated token, a chunk of metadata that describes what Security Identifier (SID) owns the process and what privileges have been granted to it. A SID is essentially a unique ID that is assigned to a user or group, and is broken into several parts: the revision (currently always set to 1), the identifier authority (describing what authority created the SID, and hence how to interpret the sub-authorities), and finally a list of sub-authorities.

In general, when users see SIDs (which they rarely do), they are in what's called the Security Descriptor Definition Language (SDDL) form. This is a SID string looks like:

S-1-5-21-1957994488-484763869-854245398-513

Here, "1" is the revision, "5" is the identifier authority, and the remaining portions are the sub-authorities.

The SID_IDENTIFIER_AUTHORITY here is actually an array of 6 characters. However, at the moment only the final character is used. For our purposes we will be focusing on the NT authority, which is {0,0,0,0,0,5}. This is the authority which describes accounts managed by the NT security subsystem. Each token contains a list of user and group SIDs. The relevant members of the _TOKEN structure (for our immediate purpose) are UserAndGroupCount (unsigned long) and UserAndGroups (pointer to array of _SID_AND_ATTRIBUTES), at offsets 0x4c and 0x68, respectively. _SID_AND_ATTRIBUTES, in turn, contains a pointer to the SID itself and a DWORD of flags giving the SID's attributes (the meaning of which are dependent on the type of SID; for group SIDs, the flags can be found in winnt.h). Unfortunately, just having the SIDs by themselves may not be so meaningful to you. Actual account names would be better; luckily, these can be found by looking in the registry. The key **HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\ProfileList** contains all the local user account SIDs on the machine, along with the location of their profile on disk. The username can usually be inferred from this; e.g. a profile directory of **SystemDrive%\Documents and Settings\Bob** would imply that the username is Bob. Aside from the individual account SIDs, there are also a number of well-known groups SIDs. These are SIDs that Microsoft has set aside for specific purposes, and will be the same on any Windows machine. A full list is available in KB243330, "Well-known security identifiers in Windows operating systems". Below you will find the syntax and results of the plugin "**GetSids.**"

```
morgan-laptop:/digitalforensics/Volatility-1.3.2$ ./volatility getsids -f /home/morgan/Raw\
Memory/PhysicalMemory.bin > getsids.txt
```

Wuser32.exe (3276): S-1-5-18 (Local System)

Wuser32.exe (3276): S-1-5-32-544 (Administrators)

Wuser32.exe (3276): S-1-1-0 (Everyone)

Wuser32.exe (3276): S-1-5-11 (Authenticated Users)

CcmExec.exe (3436): S-1-5-18 (Local System)

CcmExec.exe (3436): S-1-5-32-544 (Administrators)

CcmExec.exe (3436): S-1-1-0 (Everyone)

CcmExec.exe (3436): S-1-5-11 (Authenticated Users)

wmiprvse.exe (2348): S-1-5-18 (Local System)

wmiprvse.exe (2348): S-1-5-32-544 (Administrators)

wmiprvse.exe (2348): S-1-1-0 (Everyone)

wmiprvse.exe (2348): S-1-5-11 (Authenticated Users)

alg.exe (3044): S-1-5-19 (NT Authority)

As you can see above the results were piped out to a text file for easier viewing. This can become quite large so you will want to pipe it out to some format to review it later. With this information you can determine what process was started by which **USER** and go from there.

DLLLIST

To determine what dll's are associated with what process, run the following command:

```
./volatility dlllist -f /home/morgan/Raw\ Memory/PhysicalMemory.bin
```

logon.scr pid: 3204

Command line : C:\WINDOWS\system32\logon.scr /s

Service Pack 3

Base	Size	Path
------	------	------

0x1000000	0x39000	C:\WINDOWS\system32\logon.scr
-----------	---------	-------------------------------

0x7c900000	0xb2000	C:\WINDOWS\system32\ntdll.dll
0x7c800000	0xf6000	C:\WINDOWS\system32\kernel32.dll
0x7e410000	0x91000	C:\WINDOWS\system32\USER32.dll
0x77f10000	0x49000	C:\WINDOWS\system32\GDI32.dll
0x77f60000	0x76000	C:\WINDOWS\system32\SHLWAPI.dll
0x77dd0000	0x9b000	C:\WINDOWS\system32\ADVAPI32.dll
0x77e70000	0x92000	C:\WINDOWS\system32\RPCRT4.dll
0x77fe0000	0x11000	C:\WINDOWS\system32\Secur32.dll
0x77c10000	0x58000	C:\WINDOWS\system32\msvcrt.dll
0x5d090000	0x9a000	C:\WINDOWS\system32\COMCTL32.dll
0x76390000	0x1d000	C:\WINDOWS\system32\IMM32.DLL
0x5ad70000	0x38000	C:\WINDOWS\system32\uxtheme.dll

Note: If you only want to see a specific PID then you would insert **-p (pid #)** at the end of the syntax.

PROCDUMP

Earlier you conducted a **psscan** on the image to determine the processes currently running on the system. Now you know the processes and the dlls' associated with each process, lets see how you would dump the process if you found a file you wanted to take a closer look at. The following syntax allows you to dump a specific process to an executable with the given offset and PID. You would get the offset and PID from the [psscan](#) that was conducted earlier in this document.

```
morgan-laptop:/digitalforensics/Volatility-1.3.2$ ./volatility procdump -f /home/morgan/Raw\
Memory/PhysicalMemory.bin -o 0x06bd6918 -p 2508
```

```
Dumping ctfdmon.exe, pid: 2900 output: executable.2900.exe
```

Once you have that file you can take a closer look with some Reverse Engineering **[RE]**tools or editors of your choice.

MEMDUMP

Next you can dump the addressable memory for a process using the following syntax:

```
./volatility memdump -f /home/morgan/Raw\ Memory/PhysicalMemory.bin -o 0x06bd6918 -p 2508
```

This will give a file with the extension .dmp (dump file) and once you have this you can use other **RE** tools to conduct deeper analysis. You can also use the plugin “*memmap*” to print the memory map but it is uncertain what additional use this may provide.

MODULES

The next step is to look at the modules loaded in memory which will give you the name of the module, the offset and PID location. The syntax is as follows:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility modules -f /home/morgan/Raw\ Memory/PhysicalMemory.bin
```

File	Base	Size	Name
\WINDOWS\system32\ntkrnlpa.exe	0x00804d7000	0x1f8680	ntoskrnl.exe
\WINDOWS\system32\hal.dll	0x00806d0000	0x020300	hal.dll
\WINDOWS\system32\KDCOM.DLL	0x00ba5a8000	0x002000	kdc.com.dll
\WINDOWS\system32\BOOTVID.dll	0x00ba4b8000	0x003000	BOOTVID.dll
ACPI.sys	0x00b9f79000	0x02e000	ACPI.sys

MODDUMP

Now what do you do once you identify a module that you want to dump to a file so you can take a closer look with some **RE** tools or editors. As it turns out, dumping malicious kernel modules is quite straightforward, and in fact, it's downright trivial -- kernel modules are just [PE files](#) mapped into kernel memory (in exactly the same way as normal programs are PE files mapped into user memory). So to dump a particular kernel module, we can use Volatility's built-in PE dumper (the source is in */forensics/win32/executable.py*, and point it at the memory address of a kernel module. The syntax would be as follows:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility moddump -f /home/morgan/Raw\ Memory/PhysicalMemory.bin
```

Remember all the files will be dumped using the text format by default unless otherwise noted (See help for additional options) and they will have an extension of “**sys.**” The dumped files will be dumped in the root of the volatility folder so you will need to move them to a temp folder somewhere for review. Created a “**tmp**” directory in your home folder and below is what the dumped files will look like.

<i>total 25M</i>					
-rw-r--r--	1	morgan morgan	2.0M	2010-02-03 12:44	driver.804d7000.sys
-rw-r--r--	1	morgan morgan	129K	2010-02-03 12:44	driver.806d0000.sys
-rw-r--r--	1	morgan morgan	82K	2010-02-03 12:44	driver.a7590000.sys
-rw-r--r--	1	morgan morgan	141K	2010-02-03 12:44	driver.a75a5000.sys
-rw-r--r--	1	morgan morgan	81K	2010-02-03 12:44	driver.a75c9000.sys
-rw-r--r--	1	morgan morgan	850K	2010-02-03 12:44	driver.a75de000.sys
-rw-r--r--	1	morgan morgan	259K	2010-02-03 12:44	driver.a7753000.sys
-rw-r--r--	1	morgan morgan	56K	2010-02-03 12:44	driver.a791e000.sys
-rw-r--r--	1	morgan morgan	60K	2010-02-03 12:44	driver.a7b66000.sys
-rw-r--r--	1	morgan morgan	14K	2010-02-03 12:44	driver.a7bd6000.sys
-rw-r--r--	1	morgan morgan	327K	2010-02-03 12:44	driver.a82ce000.sys
-rw-r--r--	1	morgan morgan	71K	2010-02-03 12:44	driver.a8320000.sys
-rw-r--r--	1	morgan morgan	576K	2010-02-03 12:44	driver.a83fa000.sys
-rw-r--r--	1	morgan morgan	47K	2010-02-03 12:44	driver.a851a000.sys **

****The numbering scheme used is the offset number.**

The above is a shortened list of what actually was dumped so if you do not specify the offset or a regular expression then what you receive will be quite large. If you know a name of a piece of malware being used then use the (-p) option to specify the regular expression.

MODSCAN

Of course, if a driver is stealthy, it could unlink itself from the kernel's list of loaded modules (just as processes can be hidden by removing from the system wide process list). Often, these hidden drivers can be found by scanning memory for the data structure that represents a kernel module; Volatility's **modscan** and **modscan2** are good for this. Once you've found the hidden module, you can pass its base address to **moddump** using the -o option. Regardless of how you choose the modules to dump, they will be saved in a file called **driver.BASE_ADDRESS.sys**, where **BASE_ADDRESS** is the module's address in memory. Below is a sample:

./volatility modscan -f /home/morgan/Raw\ Memory/PhysicalMemory.bin > modscan.txt

File	Base	Size	Name
\\??C:\WINDOWS\system32\gsalrt_.sys	0x00a791e000	0x00e000	gsalrt_.sys
\\u1000\u0101\u0500##\ua002\u0201\u0500 \u0220	0x00a6fc7000	0x02b000	kmixer.sys
\\SystemRoot\system32\drivers\wdmaud.sys	0x00a7590000	0x015000	wdmaud.sys
\\SystemRoot\System32\Drivers\SYMREDRV.SYS	0x00ba388000	0x006000	SYMREDRV.SYS
File	Base	Size	Name
\\SystemRoot\System32\Drivers\HTTP.sys	0x00a7753000	0x041000	HTTP.sys
\\SystemRoot\system32\DRIVERS\srvc.sys	0x00a82ce000	0x052000	srvc.sys
\\SystemRoot\system32\DRIVERS\mdmxsdk.sys	0x00a869f000	0x003000	mdmxsdk.sys
\\SystemRoot\system32\drivers\sysaudio.sys	0x00a7b66000	0x00f000	sysaudio.sys
\\??C:\WINDOWS\system32\Drivers\BASFND.sys	0x00ba610000	0x002000	BASFND.sys
\\??C:\WINDOWS\system32\Drivers\CVPNDRVA.sys	0x00a83fa000	0x090000	CVPNDRVA.sys
\\SystemRoot\system32\DRIVERS\mrxdav.sys	0x00a859e000	0x02d000	mrxdav.sys
\\??C:\WINDOWS\system32\drivers\WpsHelper.sys	0x00a75a5000	0x024000	WpsHelper.sys
\\SystemRoot\System32\ialmdd5.DLL	0x00bf077000	0x0e2000	ialmdd5.DLL
\\SystemRoot\System32\drivers\dxgthk.sys	0x00ba732000	0x001000	dxgthk.sys
\\SystemRoot\system32\drivers\drvnddm.sys	0x00ba308000	0x00a000	drvnddm.sys
\\??C:\WINDOWS\system32\WNIPROT5.SYS	0x00ba3c8000	0x005000	WNIPROT5.SYS
\\SystemRoot\System32\Drivers\WGX.SYS	0x00ba428000	0x008000	WGX.SYS
\\SystemRoot\system32\DRIVERS\s24trans.sys	0x00a8cdb000	0x003000	s24trans.sys
\\SystemRoot\System32\DRIVERS\ws2auth.sys	0x00ba188000	0x00a000	ws2auth.sys

Once you have this you can pass the base offset to **moddump** which would look like this:

root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility moddump -f /home/morgan/Raw\ Memory/PhysicalMemory.bin -o 0x00ba5a0000
Dumping (unknown module name) @ba5a0000
Memory Not Accessible: Virtual Address: 0xba5a3000 File Offset: 0x3000 Size: 0xc80

The next internal plugin is the **thrdscan** which scans the memory for ETHTREAD objects. Once you have this information then you can dump any files of interests to dmp files as explained above.

./volatility thrdscan2 -f /home/morgan/Raw\ Memory/PhysicalMemory.bin

No. PID	TID	Offset
4	2636	0x06965548
4	3176	0x06965da8
3868	3156	0x06b31020
1352	2284	0x06b493a0
548	3528	0x06b58b30
4	3800	0x06b59bc0
4	568	0x06b8dda8
2052	712	0x06b93020
376	3552	0x06b93318
4	1016	0x06ba1ba8
3472	2096	0x06ba3608
4	2712	0x06ba5410
3472	3588	0x06bce6c0
896	2952	0x06bd1020
3232	3844	0x06bf3b30
748	1820	0x06bf3da8
1892	1096	0x06bf7020
1696	3792	0x06c023c8
656	1580	0x06c14020
2508	2616	0x06c15020

VADINFO

Virtual Address Descriptors (VAD) record the usage of virtual addresses by a process. VAD are kept in a balanced tree whereas a member of the `_EPROCESS` structure points to the root node. Rebuilding the VAD tree allows to reconstruct a process' memory space along with all the files mapped into it.

To retrieve this information you have 3 plugins at your disposal "**vadinfo**", "**vadwalk**" and "**vaddump**." The 1st plugin **vadinfo** allows you to dump the information from memory onto the screen or in a file. The syntax is as follows:

./volatility vadinfo -f /home/morgan/Raw\ Memory/PhysicalMemory.bin

VAD node @8709fb10 Start 7ffde000 End 7ffdefff Tag Vadl
Flags: MemCommit, PrivateMemory, NoChange
Commit Charge: 1 Protection: 4
First prototype PTE: 00000000 Last contiguous PTE: 00000000
Flags2: LongVad, OneSecured
File offset: 00000000
Secured: 7ffde000 - 7ffdefff
Pointer to _MMEXTEND_INFO (or _MMBANKED_SECTION ?): 00000000

This will show you the information needed to dump the information to a file if needed. The syntax to dump the file is as follows:

The next plugin is “vadwalk” and it allows you to walk the vad tree with different outputs. You can either have the output in tree format, dot format or table format. I did not put an output in the document as this is self-explanatory.

`./volatility vadwalk -f /home/morgan/Raw\ Memory/PhysicalMemory.bin -o 0x06bd6918 -p 2508`

There will be no output on the screen it will just dump the information into a file for further processing. The file will be dumped into the volatility folder.

`./volatility vaddump -f /home/morgan/Raw\ Memory/PhysicalMemory.bin -o 0x06bd6918 -p 2508`

DRIVERSCAN

The next plugin is a scan for all drivers in memory. This plugin will provide an output showing the physical address, object type, start location, service key and the name of the driver. See output below:

`root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility driverscan -f /home/morgan/Raw\ Memory/PhysicalMemory.bin`

Phys.Addr.	Obj Type	#Ptr	nd	Start	Size	Service key	Name
0x06f1af10	0x8a8a15b8	3	0	0xa75c9000	82400	NAVENG	\Driver\NAVENG
0x070a3870	0x8a8a15b8	3	0	0xa7b66000	60800	sysaudio	\Driver\sysaudio
0x071033c8	0x8a8a15b8	7	0	0xa7590000	83072	wdmaud	\Driver\wdmaud
0x07293c38	0x8a8a15b8	3	0	0xa791e000	56704	gsalrt_	\Driver\gsalrt_

With this information you can dump the driver file as described earlier in this document for further examination.

DRIVERIRP

This plugin also requires [Andreas Schuster's driverscan.py plug-in](#). It works by over-riding the object_action method of the PoolScanDriver class. This plugin will print the driver IRP function addresses as shown below:

. /volatility driverirp -f /home/morgan/Raw\ Memory/PhysicalMemory.bin

[0] IRP_MJ_CREATE	0x804f355a	ntoskrnl.exe!0x804f355a
[1] IRP_MJ_CREATE_NAMED_PIPE	0x804f355a	ntoskrnl.exe!0x804f355a
[2] IRP_MJ_CLOSE	0x804f355a	ntoskrnl.exe!0x804f355a
[3] IRP_MJ_READ	0x804f355a	ntoskrnl.exe!0x804f355a
[4] IRP_MJ_WRITE	0x804f355a	ntoskrnl.exe!0x804f355a
[5] IRP_MJ_QUERY_INFORMATION	0x804f355a	ntoskrnl.exe!0x804f355a
[6] IRP_MJ_SET_INFORMATION	0x804f355a	ntoskrnl.exe!0x804f355a
[7] IRP_MJ_QUERY_EA	0x804f355a	ntoskrnl.exe!0x804f355a
[8] IRP_MJ_SET_EA	0x804f355a	ntoskrnl.exe!0x804f355a
[9] IRP_MJ_FLUSH_BUFFERS	0x804f355a	ntoskrnl.exe!0x804f355a
[10] IRP_MJ_QUERY_VOLUME_INFORMATION	0x804f355a	ntoskrnl.exe!0x804f355a
[11] IRP_MJ_SET_VOLUME_INFORMATION	0x804f355a	ntoskrnl.exe!0x804f355a
[12] IRP_MJ_DIRECTORY_CONTROL	0x804f355a	ntoskrnl.exe!0x804f355a
[13] IRP_MJ_FILE_SYSTEM_CONTROL	0x804f355a	ntoskrnl.exe!0x804f355a
[14] IRP_MJ_DEVICE_CONTROL	0x804f355a	ntoskrnl.exe!0x804f355a
[15] IRP_MJ_INTERNAL_DEVICE_CONTROL	0x804f355a	ntoskrnl.exe!0x804f355a
[16] IRP_MJ_SHUTDOWN	0x804f355a	ntoskrnl.exe!0x804f355a
[17] IRP_MJ_LOCK_CONTROL	0x804f355a	ntoskrnl.exe!0x804f355a
[18] IRP_MJ_CLEANUP	0x804f355a	ntoskrnl.exe!0x804f355a
[19] IRP_MJ_CREATE_MAILSLOT	0x804f355a	ntoskrnl.exe!0x804f355a
[20] IRP_MJ_QUERY_SECURITY	0x804f355a	ntoskrnl.exe!0x804f355a
[21] IRP_MJ_SET_SECURITY	0x804f355a	ntoskrnl.exe!0x804f355a
[22] IRP_MJ_POWER	0x804f63ac	ntoskrnl.exe!0x804f63ac

[23] IRP_MJ_SYSTEM_CONTROL	0x80591c96	ntoskrnl.exe!0x804f355a
[24] IRP_MJ_DEVICE_CHANGE	0x804f355a	ntoskrnl.exe!0x804f355a
[25] IRP_MJ_QUERY_QUOTA	0x804f355a	ntoskrnl.exe!0x804f355a
[26] IRP_MJ_SET_QUOTA	0x804f355a	ntoskrnl.exe!0x804f355a
[27] IRP_MJ_PNP	0x80592510	ntoskrnl.exe!0x80592510

FILEOBJSCAN

This plugin gives you the ability to link a file to a hidden or terminated process. The technical details of this process can be reviewed on:

http://computer.forensikblog.de/en/2009/04/linking_file_objects_to_processes.html/.

./volatility fileobjscan -f /home/morgan/Raw Memory/PhysicalMemory.bin

Phys.Addr.	Obj Type	#Ptr	#Hnd	Access	EProcess	PID	Name
0x06afb40	0x8a8d8040	1	1	R--rw-	0x8793a838	3472	\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83
0x06b25f70	0x8a8d8040	1	1	RW-rw-	0x87391c98	1696	\Documents and Settings\144553\Local Settings\Temporary Internet Files\Content.IE5\index.dat
0x06b2aa50	0x8a8d8040	1	1	-----	0x87c73638	656	\ROUTER
0x06b34f70	0x8a8d8040	1	0	R--r-d	0x00000000	0	\Program Files\Common Files\Symantec Shared\ccL60U8.dll
0x06b492e8	0x8a8d8040	1	0	R--rwd	0x00000000	0	\Program Files\Citrix\ICA Client\pnipcn.dll
0x06b50008	0x8a8d8040	1	0	R--r--	0x00000000	0	\WINDOWS\Fonts\CALIBRI.TTF
0x06b805c0	0x8a8d8040	1	0	R--r-d	0x00000000	0	\WINDOWS\system32\etmimres.dll
0x06b83008	0x8a8d8040	1	0	R--rw-	0x00000000	0	\WINDOWS\system32\urlmon.dll

With this information you can begin to look at the processes and dlls associated with any file that looks suspicious.

IDT

This plugin prints the VAP Interrupt Descriptor Table entries. The following information was taken from <http://mnin.blogspot.com/2009/07/new-and-updated-volatility-plugin-ins.html>.

It only prints the IDT for one processor, so if anyone wants to update it to handle multiple processors, then feel free. Also if the system's KPCR is located at an address other than 0xFFDF000 (see Brendan Dolan Gavitt's blog or Edgar Barbosa's paper), then you'll need to add the correct address to the script.

In order to test the script, use the **Greg Hoglund's basic_interrupt.zip from rootkit.com**. Below you can see that interrupt #46 (0x2E) is pointing inside the basic_int.sys driver.

```
# python volatility idt -f ../hooked_int.bin
IDT# ISR unused_lo segment_type system_segment_flag DPL P
0 0008:804df350 0 0e 0 0 1
1 0008:804df4cb 0 0e 0 0 1
[...]
46 0008:f8bcd550 0 0e 0 3 1
```

KERNEL HOOKS

This plugin locates IAT/EAT-in-line API hooks in kernel space and dumps the information into sys files to a directory of your choice. See below syntax:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility kernel_hooks -f /home/morgan/Raw\
Memory/PhysicalMemory.bin -d /home/morgan/tmp/
```

Dump the files into a temp directory in your home directory where you can investigate each file further with other RE Tools or editors to determine if any malware exists. Below is an example of contents of a temp directory:

-rw-r--r--	1	morgan	morgan	2.0M	2010-02-04 08:56	driver.804d7000.sys
-rw-r--r--	1	morgan	morgan	129K	2010-02-04 08:56	driver.806d0000.sys
-rw-r--r--	1	morgan	morgan	82K	2010-02-04 08:56	driver.a7590000.sys
-rw-r--r--	1	morgan	morgan	141K	2010-02-04 08:56	driver.a75a5000.sys
-rw-r--r--	1	morgan	morgan	81K	2010-02-04 08:56	driver.a75c9000.sys
-rw-r--r--	1	morgan	morgan	850K	2010-02-04 08:56	driver.a75de000.sys
-rw-r--r--	1	morgan	morgan	259K	2010-02-04 08:56	driver.a7753000.sys
-rw-r--r--	1	morgan	morgan	56K	2010-02-04 08:56	driver.a791e000.sys
-rw-r--r--	1	morgan	morgan	60K	2010-02-04 08:56	driver.a7b66000.sys
-rw-r--r--	1	morgan	morgan	14K	2010-02-04 08:56	driver.a7bd6000.sys

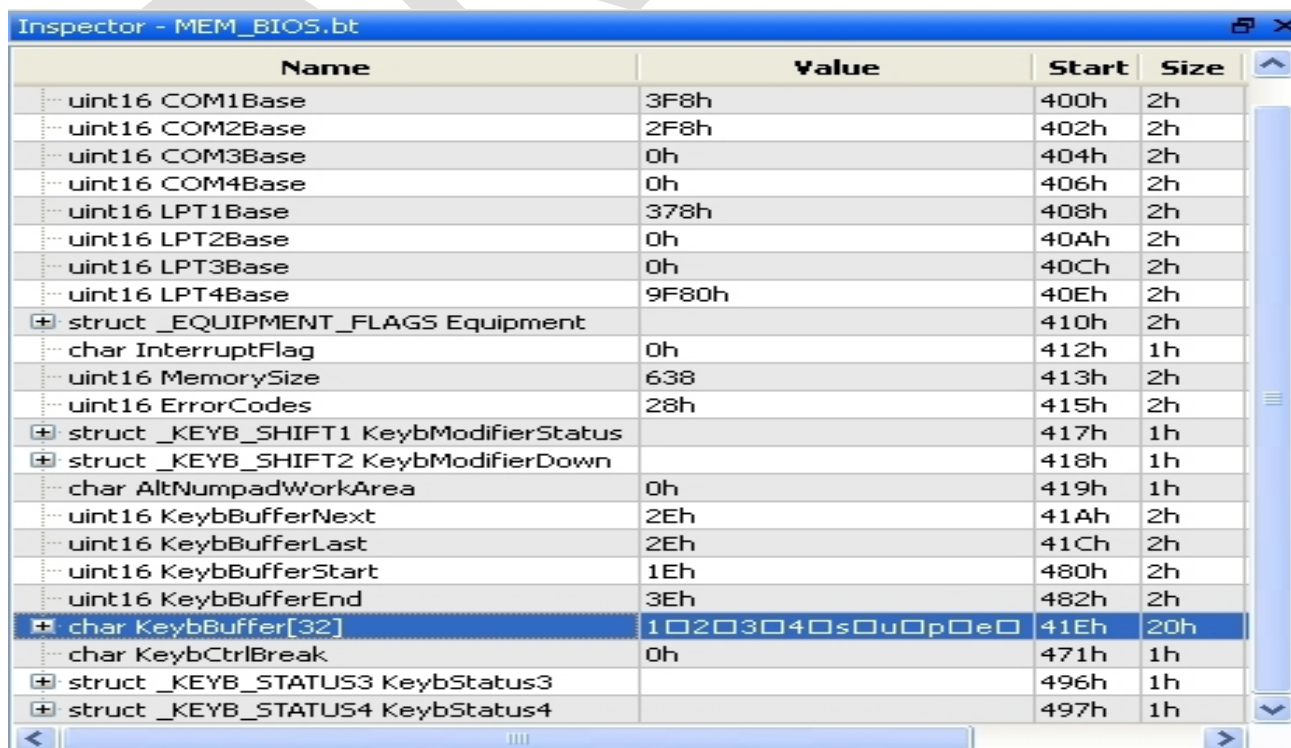
```
-rw-r--r-- 1 morgan morgan 327K 2010-02-04 08:56 driver.a82ce000.sys
-rw-r--r-- 1 morgan morgan 71K 2010-02-04 08:56 driver.a8320000.sys
-rw-r--r-- 1 morgan morgan 576K 2010-02-04 08:56 driver.a83fa000.sys
```

KEYBOARD BUFFER

This plugin extracts keyboard buffer used by the BIOS, which may contain BIOS or disk encryption passwords. The PC's BIOS among many other functions also provides a simple routine to read data in from the keyboard. Information about the keys pressed is stored in a ring buffer that provides space for about 16 characters. As Jonathan Brossard has shown in a [paper](#) and [presented](#) at DEFCON 16, the buffer's contents may be available for a while after it has been read by the BIOS. Chances are that passwords of the BIOS or disk encryption software can be recovered.

Jonathan Brossard paper provides a detailed description and several examples. Therefore this is restricted to the recovery of the BIOS startup password. In order to create the following scenario, the BIOS protection enabled and set a password of "**1234supe**". The machine was rebooted, the password was entered to unlock it and a memory dump created. Information about the position and status of the keyboard buffer is readily available from the BIOS data area. You can also spot the password.

At a closer look most characters are represented by two bytes: the character's ASCII code and a scan code that identifies the pressed key. The character "1", for instance, is represented by an ASCII code of 0x31 and its scan code 0x02. Of course there are a few exceptions. There's only the scan code but no ASCII value for the return key, for instance. Also, the BIOS does not translate keys to national character sets, but leaves this task to the operating system.



Name	Value	Start	Size
uint16 COM1Base	3F8h	400h	2h
uint16 COM2Base	2F8h	402h	2h
uint16 COM3Base	0h	404h	2h
uint16 COM4Base	0h	406h	2h
uint16 LPT1Base	378h	408h	2h
uint16 LPT2Base	0h	40Ah	2h
uint16 LPT3Base	0h	40Ch	2h
uint16 LPT4Base	9F80h	40Eh	2h
struct _EQUIPMENT_FLAGS Equipment		410h	2h
char InterruptFlag	0h	412h	1h
uint16 MemorySize	638	413h	2h
uint16 ErrorCodes	28h	415h	2h
struct _KEYB_SHIFT1 KeybModifierStatus		417h	1h
struct _KEYB_SHIFT2 KeybModifierDown		418h	1h
char AltNumpadWorkArea	0h	419h	1h
uint16 KeybBufferNext	2Eh	41Ah	2h
uint16 KeybBufferLast	2Eh	41Ch	2h
uint16 KeybBufferStart	1Eh	480h	2h
uint16 KeybBufferEnd	3Eh	482h	2h
char KeybBuffer[32]	1 02 03 04 05 0u 0p 0e	41Eh	20h
char KeybCtrlBreak	0h	471h	1h
struct _KEYB_STATUS3 KeybStatus3		496h	1h
struct _KEYB_STATUS4 KeybStatus4		497h	1h

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
03E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
03F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0400h:	F8	03	F8	02	00	00	00	00	78	03	00	00	00	00	80	9F	ø.ø.....x.....eÿ
0410h:	27	44	00	7E	02	28	00	00	00	00	2E	00	2E	00	31	02	'D.~.(.....1.
0420h:	32	03	33	04	34	05	73	1F	75	16	70	19	65	12	00	00	2.3.4.s.u.p.e...
0430h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	80e
0440h:	00	00	03	42	FF	FF	00	E0	EF	12	50	00	00	A0	00	00	...Byy.àI.P.. ..
0450h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0460h:	00	00	00	D4	03	29	30	03	00	00	C8	00	AB	1B	0C	00	...Ô.)O...È.«...

Unfortunately the BIOS data area is not a part of every memory dump. Crash dumps and dumps obtained through *LiveKd* do not contain page zero. *Win32dd* by *Matthieu Suiche* needs a special option "-t 1" to include page zero with the BIOS data area.

The plugin is rather primitive. It just dumps the keyboard buffer, but does not decode it.

Below is what you will get if it does not contain a buffer area:

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2# ./volatility keyboardbuffer -f /home/morgan/Raw\
Memory/PhysicalMemory.bin
```

Memory image does not provide buffer address, assuming defaults.

Buffer start: 0x1e

Buffer end: 0x3e

Memory image does not contain a BIOS data area.

LDR MODULES

This is a new plugin and it can detect unlinked LDR_MODULE entries. This is a technique implemented by [CloakDII](#), [NtIllusion](#), and it's also discussed with source code examples on an [OpenRCE post](#). In short, the PEB for a process contains 3 doubly-linked lists containing the loaded modules in different orders (load order, memory order, and initialization order). If you unlink a module from all 3 lists, it will essentially be hidden from tools like *listdlls.exe*, *Process Explorer*, *Process Hacker*, etc.

When DLLs are loaded with LoadLibrary, they are essentially mapped into memory (see CreateFileMapping/MapViewOfFile) and this leaves a noteworthy artifact on the system, besides the 3 lists in the PEB. The VAD entry for the memory range where the DLL is loaded contains a CONTROL_AREA structure, and this contains a FILE_OBJECT structure that identifies the full path of the mapped file. The new ldr_modules.py plugin enumerates the base addresses for memory mapped executable files using the VAD API from Volatility and makes sure that there is a corresponding entry in the 3 lists of the PEB. If any are missing, it has probably been unlinked.

There are two main arguments about using this method for detection. First, a rootkit can use DKOM and directly overwrite the FILE_OBJECT structure after unlinking from the PEB. Then it will appear as if there is no memory mapped file. However, this would require a kernel rootkit rather than one that works completely in user mode, requiring more work on the attacker's part to produce reliable and portable code. Second, it's possible to load DLLs into a process without using LoadLibrary (see [Reflective DLL Injection](#)) which doesn't create a mapped file in the VAD or any entries in the PEB. However, as Aaron Walters [hinted a while back](#), it leaves various other artifacts that are easily detectable using malfind2.

To test the plugin, a program called unlinker.exe was compiled. It unlinks the 3 PEB entries for kernel32.dll. Although kernel32.dll isn't malicious, it's fine for the demo. You can view the basic module counts for processes like this:

```
$ python volatility ldr_modules -f unlinkerldr.bin
```

```
Pid Name PEB nLoad nMem nInit nMapped
248 smss.exe 0x7ffde000 2 2 1 2
294 csrss.exe 0x7ffdf000 14 14 13 18
2ac winlogon.exe 0x7ffdd000 78 78 77 0
2e0 services.exe 0x7ffdf000 26 26 25 26
2ec lsass.exe 0x7ffde000 58 58 57 0
388 svchost.exe 0x7ffdb000 47 47 46 47
3d8 svchost.exe 0x7ffdd000 38 38 37 38
[...]
ac4 unlinker.exe 0x7ffd4000 2 2 1 3
```

Here you can see that the LoadOrder list and MemoryOrder list both contain 2 entries. The InitOrder list contains 1 entry - but it's normal for the InitOrder list to contain 1 less entry than the LoadOrder and MemoryOrder lists (the InitOrder does not contain the process EXE whereas the other two do). Then you see that there are 3 memory mapped executable files in unlinker.exe. What is the full path to the extra mapped executable file that isn't accounted for in the LoadOrder or MemoryOrder lists?

```
$ python volatility ldr_modules -f unlinkerldr.bin -v
```

```
ac4 unlinker.exe 0x7ffd4000 2 2 1 3
```

InLoadOrderModuleList

No. Map? Offset Base Size Path

[1] [x] 0x251ec0 0x400000 0x11000 C:\unlinker.exe

[2] [x] 0x251f18 0x7c900000 0xb2000 C:\WINDOWS\system32\ntdll.dll

InMemoryOrderModuleList

No. Map? Offset Base Size Path

[1] [x] 0x251ec8 0x400000 0x11000 C:\unlinker.exe

[2] [x] 0x251f20 0x7c900000 0xb2000 C:\WINDOWS\system32\ntdll.dll

InInitializationOrderModuleList

No. Map? Offset Base Size Path

[1] [x] 0x251f28 0x7c900000 0xb2000 C:\WINDOWS\system32\ntdll.dll

Mapped Files

No. Load? Mem? Init? 0xBase Name

[1] [x] [x] [] 0x400000 \unlinker.exe

[2] [x] [x] [x] 0x7c900000 \WINDOWS\system32\ntdll.dll

[3] [] [] [] 0x7c800000 \WINDOWS\system32\kernel32.dll

With the verbose output (-v flag), you can list the individual modules along with their base addresses and sizes. Each of the 3 lists contain a checkbox in the Map column which indicates if the DLL is a memory mapped file or not (they all should be). The most interesting output is in the Mapped Files section where it shows which mapped files exist in the 3 PEB lists. For the EXE itself (\unlinker.exe), it's missing an 'x' for the InitOrder list, but as we said before - this is normal. For kernel32.dll, it's missing an 'x' in all 3 lists, and that's how we know it's been unlinked.

MALFIND2

This is a new version of malfind2 which uses [YARA](#) to scan all process memory, regardless of whether it's classified as hidden/injected in the usual manner (i.e. VAD tags/permissions). Now you can find bad stuff in mapped files, loaded DLLs, or memory segments that the process allocated itself (i.e. not the result of another process calling VirtualAllocEx). All you need to do is create some YARA signatures and pass the path to your rules file on command line, like this:

```
./volatility malfind2 -d out_dir -f zeus.vmem -y rules.yar
```

#

alg.exe (Pid: 1320)

#

[!] Range: 0x006b0000 - 0x006d7fff (Tag: VadS, Protection: 0x18)

Dumping to out_dir/malfind.1320.6b0000-6d7fff.dmp

PE sections: [.odkx, .itiz, .ryd, .rsrc,]

YARA rule: passwords

Hit: IE Cookies:

0x006b1081 49 45 20 43 6f 6f 6b 69 65 73 3a 0a 00 00 00 4d IE Cookies:....M

YARA rule: loginstrings

Hit: &email=

0x006c7fa0 26 65 6d 61 69 6c 3d 00 62 74 6e 3d 00 00 00 00 &email=.btn=....

YARA rule: zbot

Description: This is just an example

Hit: ---PaNdA

0x006b1d20 3d 2d 3d 2d 50 61 4e 64 41 21 24 32 2b 29 28 2a ---PaNdA!\$2+)(*

You can see in the last YARA hit that a description is available. This is making use of the new meta tags in YARA 1.3. Although malfind2 works great with YARA, pydasm, and pefile, it no longer requires anything besides the core Volatility files. This was done because some people have trouble installing Python modules on Windows for various reasons. If you're one of those people and you've even read all the [new Volatility documentation](#), now you can still detect, dump, and see a hexdump preview of hidden/injected code segments.

MUTANTSCAN

A mutex helps to serialize access to a resource. Some applications employ a mutex to ensure that only a single instance is running. And that way, a mutex may lead us directly into the dark realms of some malware. Scary, isn't it?

A mutex is a concept that is widely used in concurrent programming. Assume that there is a global counter variable that is set to 0 and two threads. The threads will increment the variable by reading its value, incrementing and finally storing the new value. If both threads process the variable in sequence, it will end with a value of 2. Now assume that thread 1 gets interrupted after it has read the counter (still at 0). While thread 1 is suspended, thread 2 processes the variable and the counter now is 1. Finally the scheduler resumes thread 1. It also increments the previously stored value and writes it back into the global variable. And the counter now reads... 1! Crash!

In order to ensure mutual-exclusive access to the counter both threads should mark their three steps as a "critical section" or acquire a mutex on the global variable, compute and store the new value and then end the critical section or release the mutex. The second thread could then wait for the mutex to be released and then process the counter in a safe way.

There are two kinds of mutexes in the Windows kernel. Fast mutexes, also known as executive mutexes, are frequently used all over the kernel. As their name implies, they provide a mutual exclusion mechanism at a low overhead (guard mutexes of Windows Server 2003 and later are even faster). However, for the rest of this article we will be interested in mutex (or mutant) objects only.

Applications may create a global mutant with their name or some magic string. If a second instance of that application is launched, the creation of the mutant will fail and the program terminates. The mutant provides an easy means to ensure that only a single instance of that application is running. Malicious software uses this technique to avoid multiple infections of the same system.

Other software, both malicious and non-malicious, may create named mutants of other strains of malware in order to "vaccinate" a system. So, the existence of a known-bad mutant is not proof of an infection, but it justifies a closer examination.

The definition of the `_KMUTANT` structure is available from the kernel's debug symbol file:

```
kd> dt _KMUTANT
nt!_KMUTANT
+0x000 Header          : _DISPATCHER_HEADER
+0x010 MutantListEntry : _LIST_ENTRY
+0x018 OwnerThread     : Ptr32 _KTHREAD
+0x01c Abandoned       : UChar
+0x01d ApcDisable      : UChar
```

From the proper `_OBJECT_TYPE` structure we learn about the pool tag ("Muta") and that mutant objects reside in the non-paged pool. The Microsoft debugger tells us about the typical allocation sizes:

```
kd> !poolfind Muta
```

Searching NonPaged pool (810c2000 : 812c2000) for Tag: Muta

```
810dd5e0 size: 40 previous size: 18 (Allocated) Muta (Protected)
810dd650 size: 40 previous size: 30 (Allocated) Muta (Protected)
810e15f0 size: 50 previous size: 30 (Allocated) Muta (Protected)
810e1a48 size: 40 previous size: 30 (Allocated) Muta (Protected)
810e3d50 size: 50 previous size: 30 (Allocated) Muta (Protected)
810e3e20 size: 50 previous size: 30 (Allocated) Muta (Protected)
810e5dd0 size: 40 previous size: 40 (Allocated) Muta (Protected)
```

A closer examination shows that `_KMUTANT` always is of the same size. Again it is the `_OBJECT_HEADER` that varies in size; some headers also contain an object name, which is exactly what we're interested in.

Let's try the plugin on [exemplar7](#) of [hogfly's](#) fine collection of malware memory dumps. There are two suspicious mutants:

`./volatility mutantscan -s -f exemplar7.vmem`

Phys.Addr.	Obj Type	#Ptr	#Hnd	Signal	Thread	CID	Name
0x01825480	0x817c9858	2	1	0	0x813dada8	312:532	9g234sdfdfgj2304
0x018e2278	0x817c9858	2	1	0	0x8145a020	1092:1440	DLL32M

The [full list](#) contains a lot of nameless mutants. Therefore I recommend applying the `-s` or `--silent` switch, which cuts the list down to [named mutants](#).

The above output tells us whether the mutant is in its signaled ("released") state and which thread acquired the mutant. From the associated thread object we then learn about the client ID (CID), which is shown in the format of process: thread. Both identifiers are printed in decimal.

Now we query the list of running processes (truncated) to find the processes by their PID:

Name	Pid	PPid	Thds	Hnds	Time
pp04.exe	312	264	1	33	Thu Jan 08 01:51:57 2009
rundll32.exe	1092	1948	2	46	Thu Jan 08 01:52:03 2009

Searching sites like [ThreatExpert](#) for the suspicious mutant names [9g234sdfdfgj2304](#) and [DLL32M](#) then may provide us with further leads for our investigation.

OBJTYPESCAN

The Microsoft Windows kernel represents opened files by an `_FILE_OBJECT` structure. With some help from the Microsoft Debugger, the object type information about files and the Volatility memory analysis framework it is an easy task to craft a file object scanner. This scanner may reveal files even if they are hidden by a rootkit.

Fortunately, `_FILE_OBJECT` is a common structure. So we don't have reverse parts of the kernel or have to rely on rare debug symbols with full type information in order to learn about the details. Here is what the debugger tells about the file object on Windows XP with Service Pack 2:

```
kd> dt _FILE_OBJECT
ntdll!_FILE_OBJECT
struct _FILE_OBJECT, 27 elements, 0x70 bytes
+0x000 Type           : Int2B
+0x002 Size           : Int2B
+0x004 DeviceObject   : Ptr32 _DEVICE_OBJECT
+0x008 Vpb            : Ptr32 _VPB
+0x00c FsContext       : Ptr32 Void
+0x010 FsContext2      : Ptr32 Void
+0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
+0x018 PrivateCacheMap : Ptr32 Void
```

```

+0x01c FinalStatus      : Int4B
+0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
+0x024 LockOperation    : UChar
+0x025 DeletePending    : UChar
+0x026 ReadAccess       : UChar
+0x027 WriteAccess      : UChar
+0x028 DeleteAccess     : UChar
+0x029 SharedRead       : UChar
+0x02a SharedWrite      : UChar
+0x02b SharedDelete     : UChar
+0x02c Flags            : Uint4B
+0x030 FileName         : _UNICODE_STRING
+0x038 CurrentByteOffset : _LARGE_INTEGER
+0x040 Waiters          : Uint4B
+0x044 Busy             : Uint4B
+0x048 LastLock         : Ptr32 Void
+0x04c Lock             : _KEVENT
+0x05c Event            : _KEVENT
+0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEX

```

From the [list of kernel object types](#) we learn that file objects reside in the paged pool. Allocations are tagged with "File". Running! poolfind File in the debuggers retrieves a list of matching pool allocations. As you will find out, the minimum block size is 0x98. Knowing about the data structure, pool type, pool tag and allocation size, it is easy to craft a plugin for the Volatility memory analysis framework.

Below is an excerpt of the results from the scanning [hogfly's exemplar18.vmem](#) for file objects.

Phys.Addr.	Obj Type	#Ptr	#Hnd	Access	Name
0x016cea70	0x817b8730	1	0	R--r-d	\WINDOWS\Temp\tempo-447187.tmp
0x016c76f0	0x817b8730	1	0	R--r-d	\WINDOWS\Temp\gaopdx447031.tmp
0x016dcc08	0x817b8730	1	0	R--r-d	\WINDOWS\system32\drivers\gaopdxserv.sys
0x018b3ae0	0x817b8730	1	0	-W-r--	\WINDOWS\system32\gaopdxcounter
0x019e3398	0x817b8730	1	0	R--r-d	\WINDOWS\system32\gaopdxtnsnsftaavppfgmkbshkvxtlvnrjypjq.dll

These are typical file names for W32/TDSS or W32/Tidserv.

The plugin displays the following values:

- **Phys.Addr** - the physical address of the file object in the memory image
- **Obj Type** - object type pointer of the "File" class. The value may change between systems or even reboots, but while Windows is running it should be the same among all objects of the same class.
- **#Ptr** - the number of pointers to the File object
- **#Hnd** - the number of handles to the File object
- **Access** - R/W/D indicates Read, Write and Delete access, r/w/d indicates shared read, shared write and shared delete access
- **Name** - name associated with the File object

ORPHAN THREADS

This plugin finds kernel modules that don't map back to loaded modules. No detailed information was found about this plugin.

```
root@morgan-laptop:/digitalforensics/Volatility-1.3.2#. /volatility orphan_threads -f /home/morgan/Raw\
Memory/PhysicalMemory.bin -p 1696
```

PID	TID	Offset	Start Address
1696	3792	0x6c023c8	0x7c8106f9
1696	3540	0x72c67e0	0x7c8106f9
1696	3500	0x7991020	0x7c810705
1696	2380	0x90db020	0x7c8106f9

With this information you will need to go back and take a look at these files using the hex offset files.

SSDT

When malicious, kernel-level code is installed on the system, one action it may take is to hook various system services. What this means is that it takes some standard piece of operating system functionality and replaces it with its own code, allowing it to alter the way all other programs use the OS. For example, it may hook functions involved in opening registry keys, and modify their output so as to hide registry keys the rootkit uses. As system calls are the primary interface between user and kernel mode, the system call table is a popular place to do such hooking.

It's worth noting that many security products also make heavy use of hooking. One common example is antivirus software; among the many functions it hooks is `NtCreateProcess` (used, as the name suggests, to start a new process) so that it can do its on-demand scanning of any newly launched programs. For this reason, it's not safe to assume that any hooking of system calls is malicious; in fact, some of the most suspicious-looking things initially often turn out to be security software.

Still, it may be quite useful to be able to examine the system call table of a memory image during an investigation, in order to detect any hooks that shouldn't be there. To do this, we'll first look at how system calls work in Windows and lay out the data structures that are involved. I'll then describe a Volatility plugin that examines each entry in the system call table, gives its symbolic name, and then tells what kernel module owns

the function it points to.

If you look at any of the native API functions, like `ZwCreateFile`, you'll notice that they all look almost identical:

```
Lkd> u nt!ZwCreateFile
nt!ZwCreateFile:
804fd724 b825000000    mov     eax,25h
804fd729 8d542404    lea     edx,[esp+4]
804fd72d 9c         pushfd
804fd72e 6a08       push    8
804fd730 e83cf10300 call    nt!KiSystemService (8053c871)
804fd735 c22c00     ret     2Ch
```

We see that the function just places the value 0x25 into `eax`, points `edx` at the stack, and calls `nt!KiSystemService`. It turns out that this value, 0x25, is the system call number that corresponds to the `CreateFile` function. Without going into too much detail about how `KiSystemService` works, the function essentially takes the value in the `eax` register, and then looks up that entry in a global system call table. The table contains function pointers to the actual kernel-land functions that implement that system call.

But, of course, the situation isn't quite as simple as that. In fact, Windows is designed to allow third party developers to add their own system calls. To support this, each `_KTHREAD` contains a member named `ServiceTable` which is a pointer to a data structure that looks like this:

```
typedef struct _SERVICE_DESCRIPTOR_TABLE {
    SERVICE_DESCRIPTOR_ENTRY Descriptors[4];
} SERVICE_DESCRIPTOR_TABLE;
typedef struct _SERVICE_DESCRIPTOR_ENTRY {
    PVOID KiServiceTable;
    PULONG CounterBaseTable;
    LONG ServiceLimit;
    PCHAR ArgumentTable;
} SERVICE_DESCRIPTOR_ENTRY;
```

As you can see, we can actually have up to four separate system service tables per thread! In practice, however, we only see the first two entries in this array filled in: the first one points **to `nt!KiServiceTable`**, which contains the functions that deal with standard OS functionality, and the second points to `win32k!W32pServiceTable`, which contains the functions for the GDI subsystem (managing windows, basic graphics functions, and so on). For system call numbers up to 0x1000, the first table is used, while for the range 0x1000-0x2000 the second table is consulted (this may generalize for 0x2000-0x3000 and 0x3000-0x4000, but I haven't tested it).

To take a look at the contents of these two tables, we can use the `dps` command in **WinDbg**, which takes a memory address and then attempts to look up the symbolic name of each `DWORD` starting at that address. To examine the full table, you should pass `dps` the number of `DWORDS` you want to examine -- the exact number will be the value found in the `ServiceLimit` member for the table you're interested in. For example:

```

lkd> dps nt!KiServiceTable L11c
805011fc 80598746 nt!NtAcceptConnectPort
80501200 805e5914 nt!NtAccessCheck
80501204 805e915a nt!NtAccessCheckAndAuditAlarm
80501208 805e5946 nt!NtAccessCheckByType
[...]
8050128c 8060be48 nt!NtCreateEventPair
80501290 8056d3ca nt!NtCreateFile
80501294 8056bc5c nt!NtCreateIoCompletion
[...]

```

Note that NtCreateFile is the 0x25th entry in the table, as we expected. On a system with no hooks installed, all functions in nt! KiServiceTable will point into the kernel (ntoskrnl.exe), and all functions in win32k! W32pServiceTable will be inside win32k.sys. If they don't, it means the function has been hooked.

The plugin works as follows. First, we go over each thread in each process, and gather up all distinct pointers to service tables. We examine all of them in case one thread has had its ServiceTable changed while the others remain untouched. Then we display each entry in each (unique) table, along with the name it usually has (in an unhooked installation), and what driver the function belongs to. Here's some sample output:

. /volatility ssdt -f xp-laptop-2005-07-04-1430.img

```

Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 804e26a8 with 284 entries
Entry 0x0000: 0x805862de (NtAcceptConnectPort) owned by ntoskrnl.exe
Entry 0x0001: 0x8056fded (NtAccessCheck) owned by ntoskrnl.exe
Entry 0x0002: 0x8058945b (NtAccessCheckAndAuditAlarm) owned by ntoskrnl.exe
[...]
Entry 0x0035: 0xf87436f0 (NtCreateThread) owned by wpsdrvnt.sys
[...]
SSDT[1] at bf997780 with 667 entries
Entry 0x1000: 0xbf93517d (NtGdiAbortDoc) owned by win32k.sys
Entry 0x1001: 0xbf946c1f (NtGdiAbortPath) owned by win32k.sys
[...]

```

Here we can see that the NtCreateThread function has been hooked by wpsdrvnt.sys. A little Googling shows that this driver is a part of Sygate Personal Firewall -- as mentioned before, security products are the most common non-malicious software that hooks kernel functions.

In closing, one caveat should be noted when using this tool: at the moment, the names of the system calls are hardcoded with the values derived from **WinDbg** on Windows XP SP2. As demonstrated by the [Metasploit System Call Table](#) page, the order and number of entries in the system call table change between different versions of Windows, so make sure that you only analyze **SP2** images with this plugin! As always, patches are welcome if you want to adapt this to deal with other versions of Windows.

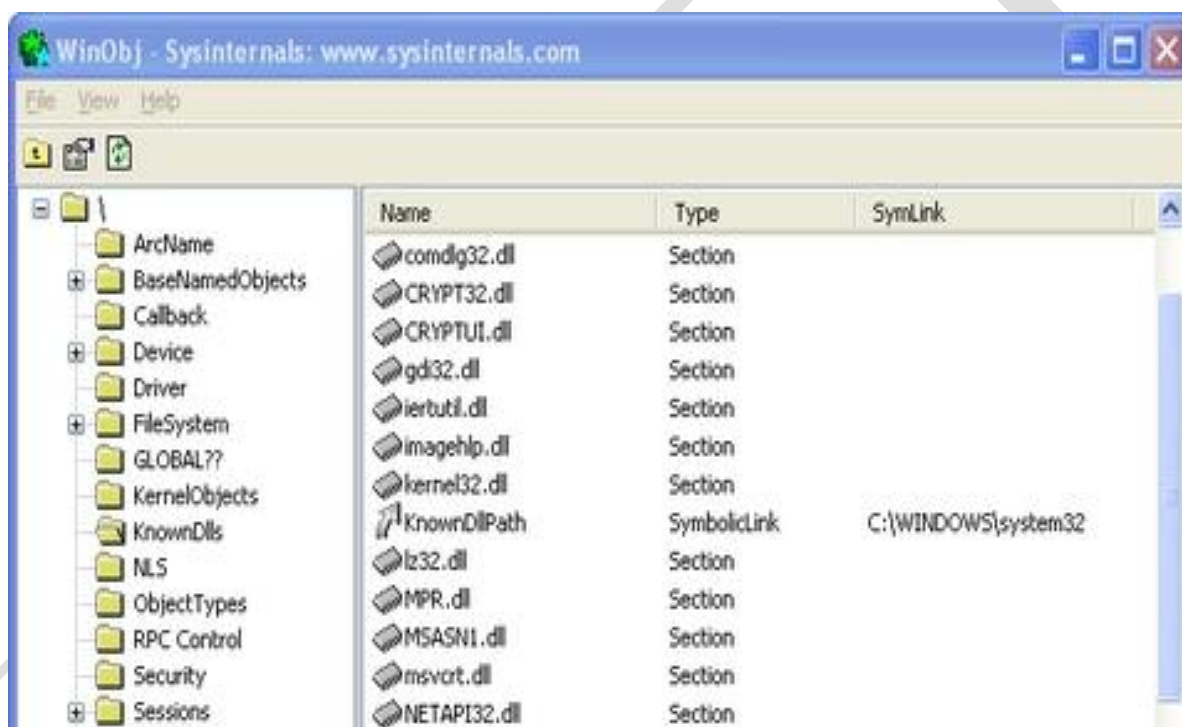
You can use this syntax to filter out *ntoskrnl* and *win32k.sys*.

```
./volatility ssdt -f /samples/exemplar15.vmem | grep -v ntoskrnl.exe | grep -v win32k.sys
```

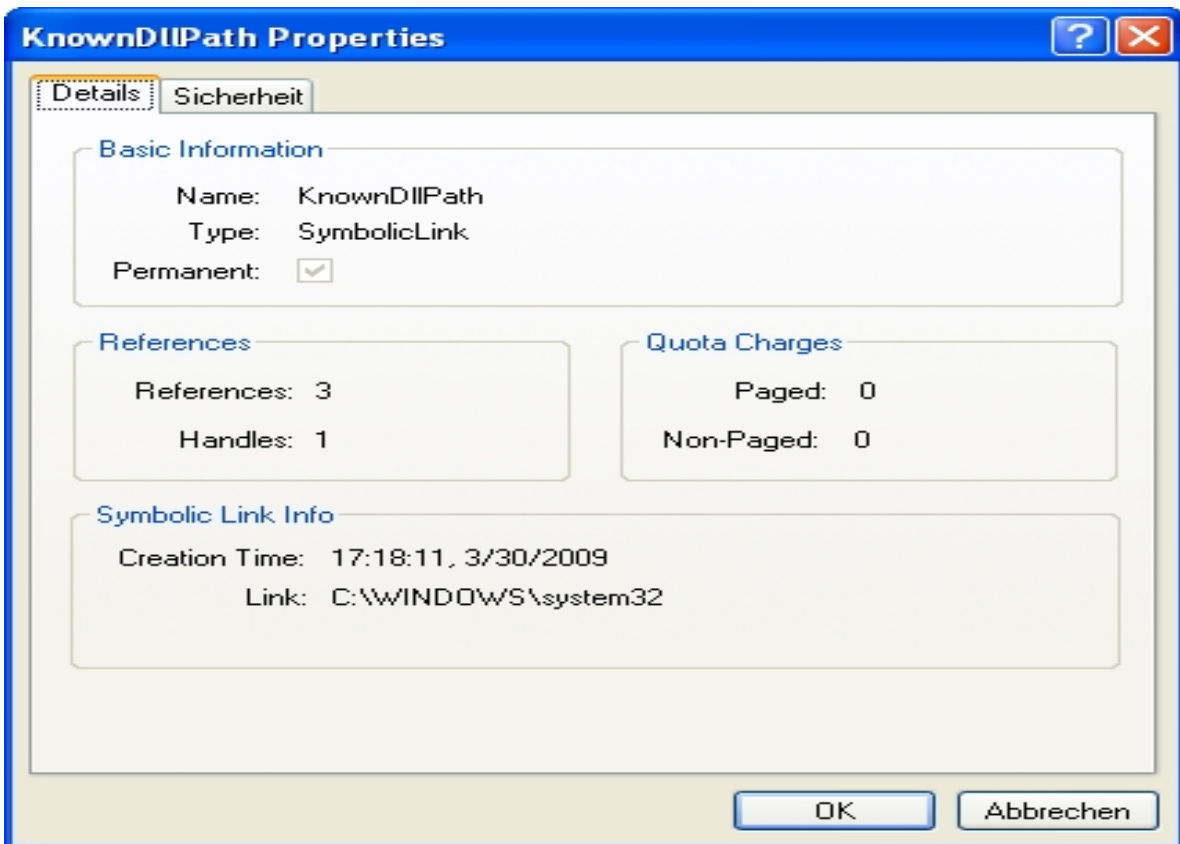
SYMLINKOBJSCAN

The concept of symbolic links is widely implemented in file systems. But there is also a symbolic link object for kernel objects. Generally, a symbolic link will make an object accessible under a different and probably much shorter name. But symbolic link objects also provide some forensic value.

Again, [WinObj](#) by Mark Russinovich might be a good start to explore symbolic link objects. The following objects link the name "KnownDllPath" to "C:\WINDOWS\system32".



But there's more in the properties of a symbolic link object. As you can see from the following screen shot, Windows stores an object creation time stamp, for whatever reason.



From the `_OBJECT_TYPE` structure we learn about the pool tag "Symb" and that its memory gets allocated from the paged pool. Matching allocations are always 0x50 bytes in size (on Windows XP with ServicePack 2). Here is an example of how it looks like in a debugger or hex editor:

```
kd> db e1347358
e1347358  05 02 0a 0c 53 79 6d e2-88 47 00 e1 86 00 86 00  ....Symb..G.....
e1347368  d8 84 39 e1 01 00 00 00-01 00 00 00 00 00 00 00  ..9.....
e1347378  a8 e0 2b 81 10 00 00 32-01 00 00 00 d7 73 00 e1  ..+....2.....s..
e1347388  22 84 97 cc 98 98 c7 01-20 00 22 00 00 d8 34 e1  "....."....4.
e1347398  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  .....
e13473a8  0a 02 01 00 4b 65 79 a0-01 02 02 0c 4f 62 44 69  ....Key.....ObDi
```

In the first line you will spot the "Symb" pool tag. Note that the most significant bit is set to indicate a system object. In the last line there's already the next allocation (with a "Key" pool tag). The symbolic link object starts at 0xe1347388:

```
kd> !object e1347388
Object: e1347388  Type: (812be0a8) SymbolicLink
  ObjectHeader: e1347370 (old version)
  HandleCount: 0  PointerCount: 1
  Directory Object: e1004788  Name: ACPI\FixedButton#2&daba3ff&0#{4afa3d53-74a7-11d0-
be5e-00a0c9062857}
  Target String is '\Device\00000035'
```

The object's payload starts with the creation time stamp. You'll have to copy the bytes by hand because the function doesn't take a pointer to the time stamp. The time stamp as usual is in 64bit Windows filetime format.

```
kd> !filetime 01c79898cc978422
5/17/2007 16:34:03.828 (local time)
```

Yes, this is from an old memory image, the time stamp looks plausible. Next is an Unicode string:

```
kd> !ustr e1347390
String(32,34) at e1347390: \Device\00000035
```

We now know how to decode the symbolic link object's creation time stamp and target string. For the curious: the name string is stored in an "ObNm" (Object Name) allocation, while the target string is stored in an "Symt" (Symbol Target) allocation. Both are in the paged pool.

Now let's have a look at the `_OBJECT_HEADER`:

```
kd> dt _OBJECT_HEADER e1347370
nt!_OBJECT_HEADER
+0x000 PointerCount      : 1
+0x004 HandleCount      : 0
+0x004 NextToFree       : (null)
+0x008 Type              : 0x812be0a8 _OBJECT_TYPE
+0x00c NameInfoOffset    : 0x10
+0x00d HandleInfoOffset : 0 ''
```

```

+0x00e QuotaInfoOffset : 0 ''
+0x00f Flags           : 0x32
+0x010 ObjectCreateInfo : 0x00000001 _OBJECT_CREATE_INFORMATION
+0x010 QuotaBlockCharged : 0x00000001
+0x014 SecurityDescriptor : 0xe10073d7
+0x018 Body            : _QUAD

```

The NameInfoOffset member tells us that there is information about the object's name available 0x10 bytes in front of the common object header.

```

kd> db e1347370-10 L 10
e1347360 88 47 00 e1 86 00 86 00-d8 84 39 e1 01 00 00 00 .G.....9.....

```

Now what does that tell us? The first four bytes (0xe1004788) are a pointer back to the object directory that contains this object. The next eight bytes again are a Unicode string:

```

kd> !ustr e1347360+4
String(134,134) at e1347364: ACPI\FixedButton#2&daba3ff&0#{4afa3d53-74a7-11d0-be5e-00a0c9062857}

```

So, this is the last piece of information that was missing, the object's name. With the help of the [Volatility memory analysis framework](#) it is not too hard to turn that knowledge into a scanner for symbolic link objects.

Now issue **`./volatility symlinkobjscan -f filename`** to scan your memory image for symbolic link kernel objects. Here are excerpts from two sample images to demonstrate the symbolic link scanner.

The following symbolic links show the mapping between drive letters and devices. The user mapped the "T:" drive to a VMware shared folder a couple of minutes after the system was booted.

```

Time created      Name -> Target
Fri Apr 10 09:55:17 2009 C: -> \Device\HarddiskVolume1
Fri Apr 10 09:55:35 2009 A: -> \Device\Floppy0
Fri Apr 10 09:55:35 2009 D: -> \Device\CdRom0
Fri Apr 10 08:58:51 2009 T: -> \Device\HGFS

```

The following example shows the well-known Hacker Defender rootkit. Again the creation time clearly sticks out, but that would not be the case after a reboot.

```

Time created      Name -> Target
Fri Apr 10 09:55:35 2009 NdisWan -> \Device\NdisWan
Fri Apr 10 08:58:37 2009 HxDefDriver -> \Device\HxDefDriver
Fri Apr 10 09:55:34 2009 RdpDrDvMgr -> \Device\RdpDrDvMgr

```

In Windows, the GUI system is event-driven—actions occur in response to various events generated by the system. When you press a key on the keyboard, Windows generates a window message (specifically, [WM_KEYDOWN](#)) and sends it to the thread that owns the window that's in focus. That thread then calls the window's event handling procedure (the so-called [WindowProc](#)) to process the message. There are [many such messages](#), covering input events such as [keyboard](#) and [mouse](#) actions, system-level events such as notification of a [time change](#) or a [change in the system's power state](#), and events related to the windowing system, such as [resizing](#) or [moving](#) a window.

How can these be forensically relevant? Well, as it turns out, some threads in buggy applications aren't always good at processing their messages, and messages they receive pile up in the queue. This means that by looking at the message queues on a system, we can get some information about its past state. To make this more concrete, let's look at the message queue for a thread belonging to a certain full-disk encryption vendor on one of the images in my collection:

```
0:03:42.812,WM_WTSSESSION_CHANGE,WTS_CONSOLE_CONNECT
0:03:42,WM_WTSSESSION_CHANGE,WTS_SESSION_LOGON
1:45:05,WM_WTSSESSION_CHANGE,WTS_CONSOLE_DISCONNECT
1:45:06,WM_WTSSESSION_CHANGE,WTS_REMOTE_CONNECT
1:45:34,WM_WTSSESSION_CHANGE,WTS_REMOTE_DISCONNECT
1:46:19,WM_WTSSESSION_CHANGE,WTS_CONSOLE_CONNECT
2:28:50,WM_WTSSESSION_CHANGE,WTS_SESSION_LOCK
18:18:07,WM_WTSSESSION_CHANGE,WTS_SESSION_UNLOCK
22:05:51,WM_WTSSESSION_CHANGE,WTS_SESSION_LOCK
22:23:47,WM_WTSSESSION_CHANGE,WTS_SESSION_UNLOCK
23:32:34,WM_WTSSESSION_CHANGE,WTS_SESSION_LOCK
23:34:22,WM_WTSSESSION_CHANGE,WTS_SESSION_UNLOCK
1 day, 0:07:54.468,WM_WTSSESSION_CHANGE,WTS_SESSION_LOCK
[...]
8 days, 0:18:13.046,WM_WTSSESSION_CHANGE,WTS_SESSION_UNLOCK
8 days, 0:32:02.640,WM_WTSSESSION_CHANGE,WTS_SESSION_LOCK
8 days, 0:36:01.500,WM_WTSSESSION_CHANGE,WTS_SESSION_UNLOCK
```

(I've omitted many of the details here to save space. Window messages also include things like the handle of the window the message is for and cursor position at the time the message was sent.)

If we look up [WM_WTSSESSION_CHANGE on MSDN](#), we find that it's a message related to fast user switching; one of these is sent whenever someone logs in, locks the screen, or connects remotely (i.e. via remote desktop). The message is sent to all applications that have called [WTSRegisterSessionNotification](#). However, in this case, despite the fact that the application asked to be notified of such changes, it didn't bother to process the messages it received! This means that we can now look through its queue and find out when the user logged in and out, when the screen was locked, and so on. (The times given are relative to the time the system

was booted, and are only good up to 49.7 days—this is because the timestamp comes from the [GetTickCount](#) function). It should be clear why such information might be useful in a forensic investigation.

I want to stress that we simply got lucky in this case by finding such a wealth of information in the message queue. It would be unwise to rely on every system having a misbehaving application like the one in this example; on the NIST images, for example, and on my own (clean) test VMs, there were no messages found on the system at all—the applications involved had processed them, and so they were no longer queued. Still, on real-world systems, buggy applications may be more common, so this trick could come in handy.

This plugin enumerates all threads on the system and lists any messages it finds. It has an internal table mapping numeric IDs to names for a large number of standard window messages; however, it is very likely that the list is not complete. In addition, applications can define their own message types; in these cases, interpreting the message is impossible without analyzing the program.

One quick note of warning, though: when you're exploring these structures, you may initially think that all of the information is paged out, because none of the memory addresses seem valid. As it turns out, although Win32Thread and its friends all live in kernel space, this portion of kernel space is not visible from all threads. This flies in the face of a very common assumption in Windows memory analysis—that the kernel portion of memory looks the same to every process. In fact, the portions related to the GUI subsystem are only visible from processes that have at least one GUI thread! So, in particular, the System process, which is what's most commonly used in Volatility to access kernel memory, can't see any of the structures we're interested in. In my plugin, I make sure to use the address space of the process that owns the threads we're examining, so that the GUI structures will be accessible if the thread is a GUI thread.

. /volatility thread_queues -f /home/morgan/Raw Memory/Physical Memory.bin

Thread 850:ef8 (igfxpers.exe) -- 0 messages		
Thread 850:160 (igfxpers.exe) -- 0 messages		
Thread 850:b6c (igfxpers.exe) -- 1 messages		
hWnd: [0x000102f0]	Message: [0x00000400]	cursor: (757,186)
wParam: [0x0000babe]	lParam: [0x00168114]	
dwUnk: [0x00000000]	dwExtra: [0x00000000]	
time: 6 days, 3:31:00.359000		

I am showing the above because it is unusual that if you google “Message: [0x000000400]” then you will find the following:

0X00000400 error in the Windows operating system is associated with a certain application. Errors like 0X00000400 could occur because of incorrect records in Windows Registry which is a crucial part of your Windows operating system. If it is not functioning correctly certain file extensions won't be recognized and Windows will report an error.

This file “**igfxpers.exe**” is a piece of malware that appears to have changed something in the registry. From this point you would want to extract that file from memory and have it analyzed.

The older `usermode_hooks.py` and `kernel_hooks.py` plugins have been combined into a single plugin named **`apihooks.py`**. The usage is still basically the same:

`./volatility apihooks -d out_dir -f coreflood.vmem [-p PID]`

Type Process PID Hooked Module Hooked Function From => To/Instruction Hooking Module
IAT IEXPLORE.EXE 248 USERENV.dll ADVAPI32.dll!RegSetValueExW [0x769c11f8] => 0x7ff82080
IAT IEXPLORE.EXE 248 USERENV.dll KERNEL32.dll!LoadLibraryW [0x769c12ac] => 0x7ff82ac0
IAT IEXPLORE.EXE 248 USERENV.dll KERNEL32.dll!CreateFileW [0x769c12b8] => 0x7ff82240
IAT IEXPLORE.EXE 248 USERENV.dll KERNEL32.dll!GetProcAddress [0x769c1380] => 0x7ff82360
IAT IEXPLORE.EXE 248 USERENV.dll KERNEL32.dll!LoadLibraryA [0x769c138c] => 0x7ff82a50

[...]

Total IAT hooks in user space: 287

Total EAT hooks in user space: 0

Total INLINE hooks in user space: 0

The output is much easier to read when its not pasted into a blog post. Basically it is showing that there are multiple IAT hooks inside the IEXPLORE.EXE process with Pid 248. In particular, the IAT entries in USERENV.dll which should be pointing at functions in KERNEL32.dll and ADVAPI32.dll are actually pointing at some memory in the 0x7ff82xxx range.

To scan kernel modules, just pass the `-k` flag, like this:

`./volatility apihooks -d out_dir -f skynet.vmem -k`

Type Hooked Module Hooked Function From => To/Instruction
INLINE ntoskrnl.exe IoCallDriver 0x804e37c5 => jmp 0x8217f20b
INLINE ntoskrnl.exe IoCompleteRequest 0x804e3bf6 => jmp 0x820babc3

Total IAT hooks in kernel space: 0

Total EAT hooks in kernel space: 0

Total INLINE hooks in kernel space: 2

Scans a memory image (or really anything) for True Crypt passphrases using the method described in Brian Kaplan's thesis, [RAM is Key, Extracting Disk Encryption Keys From Volatile Memory](#), pages 22-23. According to that paper, passphrases are stored in a structure containing a passphrase length and then 64 bytes of passphrase data. The data must contain exactly length ASCII characters and all remaining bytes must be zeros.

`./volatility cryptoscan -f /home/morgan/Raw Memory/Physical Memory.bin`

The last few plugins were never used or tested because there was no hiberfil.sys file or a crash dump file available. The plugins are:

dmp2raw	Dmpchk	Hibinfo	Strings
---------	--------	---------	---------

VOLATILITY OUTPUT RENDERING FUNCTIONS

This was just written by one of the coders for volatility and thought it should be included in case someone wanted all the information in a database. ***(This information was taken directly from a blog post)***

Lately the coder had been playing around writing plugins for [Volatility](#). During the writing of some of the more complicated plugins, it was decided that there was a need to have some temporary storage while doing complex processing.

[SQLite](#) is good for this. There's an option to use an [in-memory database](#) (":memory:") that will remain in memory until the process dies.

Luckily Volatility has an option for plugins to have more than one output option. If you look at the code in ***forensics/commands.py*** you'll see the following (line numbers included):

```

82  function_name = "render_%s" % self.opts.output
83  if not self.opts.out_file:
84      outfd = sys.stdout
85  else:
86      outfd = open(self.opts.out_file, 'w')
```

This allows plugins to have more than one output function. For example a plugin might have a `render_text` function that would print to stdout as usual, a `render_html` function that prints out in html style, a `render_sql` function that does some SQL actions etc etc. The framework allows the user to pick which output option s/he wants and the output file on the command line as defined in `utils.py`:

```

45 def get_standard_parser(cmdname):
59 op.add_option('-H', '--output', default = 'text',
60 help='(optional, default="text") Output format (xml, html, sql)')
61 op.add_option('-O', '--out_file', default=None,
62 help='(output filename to write results onto - default stdout)')

```

Therefore, if there is a plugin that has an xml output option, it can be invoked from the command line like so:

```
./volatility <plugin> -H html -O <out_file> -f mem.dd
```

Plugin Structure

If you are interested in writing plugins for Volatility, you really should read [Andreas Schuster's slides](#). They go into nice detail on how to write plugins for the framework. Here I will simply give you the gist :-)

The "skeleton" for the plugins is defined in `forensics/commands.py`. Items of interest include the `help()` function which is the plugin description you see when you run Volatility with the help option:

```
./volatility -h
```

Also of interest is the `parser()` function, which allows the plugin to modify its command line options. There is also the `calculate()` function, which is where the real work is done. The last item of our interest is the `execute()` function which allows us to calculate and collect the desired data from the memory image and then output it using the plugin's chosen `render_*` function.

The plugins I'm releasing now consist of core commands (defined in `vmodules.py`) that have been converted to this code structure so I could have more than one type of output for each of these commands. The plugins in this package are:

<code>memory_plugins/connections_2.py</code>	<code>memory_plugins/modules_2.py</code>
<code>memory_plugins/dlllist_2.py</code>	<code>memory_plugins/pslist_2.py</code>
<code>memory_plugins/files_2.py</code>	<code>memory_plugins/sockets_2.py</code>

The schema for these plugins is quite simple and not much different than the original output for these core commands. There is an extra field for the name of the memory image that was analyzed in case someone would like to place information for more than one memory image into a SQLite database. This may change at some point and of course you are free to change it as you like. It's enough for what I needed, however.

connections_2.py	
<i>Table Name: connections</i>	
pid	<i>Process ID</i>
local	<i>Local connection information</i>
remote	<i>Remote connection information</i>
memimage	<i>Memory image information was extracted from</i>
memory_plugins/dlllist_2.py	
<i>Table Name: dlls</i>	
image_file_name	<i>Process name</i>
pid	<i>Process ID</i>
cmdline	<i>Command Line text</i>
base	<i>Base Address</i>
size	<i>Size</i>
path	<i>Path of DLL</i>
memimage	<i>Memory image information was extracted from</i>
memory_plugins/files_2.py	
<i>Table Name: files</i>	
pid	<i>Process ID</i>
file	<i>Open file</i>
num	<i>Number of times file is open by pid</i>
memimage	<i>Memory image information was extracted from</i>
memory_plugins/modules_2.py	
<i>Table Name: modules</i>	
file	<i>Module Path</i>
base	<i>Base Address</i>
size	<i>Size</i>
name	<i>Module Name</i>
memimage	<i>Memory image information was extracted from</i>

memory_plugins/pslist_2.py	
Table Name: process	
pname	Process Name
pid	Process ID
ppid	Parent Process ID
thrds	Threads
hndl	Handle Count
ctime	Creation Time
memimage	Memory image information was extracted from
memory_plugins/sockets_2.py	
Table Name: sockets	
pid	Process ID
port	Port
proto	Protocol
ctime	Creation Time
memimage	Memory image information was extracted from

Installation

First, make sure you have [SQLite3 installed](#) along with support for Python. Now download the [Volatility code from the SVN](#). Download the plugins [from here](#). A listing of the plugins are as follows:

\$ tar -tzf vol_sql.tgz	vutils.py
forensics/commands.py	memory_plugins/connections_2.py
memory_plugins/dlllist_2.py	memory_plugins/files_2.py
memory_plugins/modules_2.py	memory_plugins/pslist_2.py
memory_plugins/sockets_2.py	

Make a backup of your **vutils.py** and **forensics/commands.py** files if you like. Small modifications had to be made to both of these files to get the plugins working properly. Then place the tar file into your Volatility directory and type:

```
tar -xvzf vol_sql.tgz
```

Each of the redefined core commands end with "_2" so pslist becomes pslist_2 and connections becomes connections_2 and so on. So if you wanted to dump the output of the connections_2 plugin to a SQLite file type the following:

```
./volatility connections_2 -H sql -O test.db -f mem.dd
```

After running all of the new commands to the same SQLite3 file, look at what is stored:

```
$ sqlite3 test.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .table
connections dlls files modules process sockets
sqlite> .schema
CREATE TABLE connections (pid integer, local text, remote text, memimage text);
CREATE TABLE dlls (image_file_name text, pid integer, cmdline text, base text, size text, path text, memimage text);
CREATE TABLE files (pid, file, num, memimage);
CREATE TABLE modules (file text, base text, size text, name text, memimage text);
CREATE TABLE process (pname text, pid integer, ppid integer, thrds text, hndl text, ctime text, memimage text);
CREATE TABLE sockets (pid integer, port integer, proto text, ctime text, memimage text);
sqlite> select * from files where pid = 4 and file like '%SEC%';
4|\WINDOWS\system32\config\SECURITY|1|/home/levy/forensic/evidence/10.vmem
4|\WINDOWS\system32\config\SECURITY.LOG|1|/home/levy/forensic/evidence/10.vmem
sqlite> .quit
```

You can make as many complex queries as you like now :-)

VOL2HTML PERL SCRIPT

[vol2html.pl](#). You can see an example report [here](#). The output files for this report and the Perl script are bundled together [here](#). There is minimal error checking.

To use, first redirect the output of Volatility for pslist, dlllist and files to text files:

```
./volatility pslist -f mem.dd > pslist.txt
```

```
./volatility files -f mem.dd >files.txt
```

```
./volatility dlllist -f mem.dd >dlllist.txt
```

Then feed the perl script these files:

```
./vol2html.pl -pslist pslist.txt -files files.txt -dlllist dlllist.txt -D [Output DIR]
```

VOLATILITY BATCH FILE MAKER

The Tool: Volatility-batch File Maker [Download](#)



This enables you to take the text output of the various tools (**Ptfinder**, **PtFinderFE** and **Volatility >PsScan2**) which identifies all the offsets for (running) processes and input that offset data into several [Volatility](#) tools (**ProcDump**, **MemDmp** and **VadDump**). This program creates three batch files. After running the batch files you can quickly leverage additional investigation techniques at the output.

1. Run **Ptfinder**, **PtFinderFE** or **Volatility >PsScan2** to create a text file that contains process offsets.

2. Run **Volatility Batch File Maker**.

- a. Select your memory image with "Browse for Memory Capture".
- b. Select the Offset Text File for your memory image with "Browse for Offset Text File".
- c. Create Batch (Which is hidden until the previous fields are populated)

Three Batch files are created, upon execution of the "Create Batch", in the root folder where the memory capture resides. The three batch programs created are **procdump.bat**, **memdmp.bat** and **vaddump.bat**. When you run the batch files each one will create a folder in the residing directory and populate that directory with the selected Volatility Output. Each batch file also creates an additional text output showing any errors (**procdumpinfo.txt**, **memdmpinfo.txt** and **vaddumpinfo.txt**).

The following is an example of the **procdump** batch file with the following two inputs:

Location of Volatility = "C:\volatility"

Location of Memory Dump = "E:\exemplar\6\exemplar6.vmem"

mkdir procdump

cd procdump

```
python "c:\volatility\volatility" procdump -f "E:\exemplar\6\exemplar6.vmem" -u -o 0x00551b80>>procdumpinfo.txt
```

```
python "c:\volatility\volatility" procdump -f "E:\exemplar\6\exemplar6.vmem" -u -o 0x0166f7b0>>procdumpinfo.txt
```

```
python "c:\volatility\volatility" procdump -f "E:\exemplar\6\exemplar6.vmem" -u -o 0x01690920>>procdumpinfo.txt
```

```
python "c:\volatility\volatility" procdump -f "E:\exemplar\6\exemplar6.vmem" -u -o 0x016aa3c0>>procdumpinfo.txt
```

```
python "c:\volatility\volatili*****Truncated - You get the idea*****
```

Uses for the Output of the Batch Files:

After re-creating all the "executables" from the running processes you can run a virus scanner at the **procdump** folder. This can be another tool in the arsenal of defeating the **Trojan Horse Defense**.

How about one for the incident response guys...

You could run this protocol at a full **memory.dmp** or on a converted **hiberfil.sys** (Converted with Suiche's [Hibershell](#)) created on a machine prior to your actual response and collection. How many times has someone "helped" you out by deleting the malware from the target machine just before you walked into the door? If the **memory.dmp** or **hiberfil.sys** is recent you might be able to "**recreate**" the malware executable. You could also show a machine has been compromised (or not) when the **memory.dmp** or **hiberfil.sys** was captured

The **VadDump** folder now contains the output which can easily be used in a program like EnCase to give you context to your memory image.

You could also show a machine has been compromised (or not) when the **memory.dmp** or **hiberfil.sys** was captured

Using the Volatility –d option for Pretty Process Mapping

(This process will work on Linux and Windows both.)

The aim of this document is to illustrate to the end user how to use and view the dot image format that can be generated using the PSLIST, PSSCAN2 Volatility commands.

If you've been playing around with Volatility you have more than likely run either **PSLIST** or **PSSCAN2** command to generate a list of processes found in a memory image:

PID	PPID	Time created	Time exited	Offset	PDB	Remarks
2900	3472	Tue Aug 04 12:34:38 2009		0x06bd6918	0x24d40820	ctfmon.exe
480	3472	Tue Aug 04 12:34:40 2009		0x06c2fda0	0x24d40840	LxrAutorun.exe
2508	1864	Tue Aug 04 12:34:11 2009		0x06d0aae8	0x24d40640	SmcGui.exe
3796	3472	Tue Aug 04 12:34:36 2009		0x06dca9e0	0x24d407c0	acrotray.exe
3152	2776	Tue Aug 04 12:34:53 2009		0x06f81af8	0x24d40900	GoogleDesktopDi
448	3472	Tue Aug 04 12:34:35 2009		0x07088678	0x24d407a0	hpwuSchd2.exe
2776	3472	Tue Aug 04 12:34:30 2009		0x07090790	0x24d40720	GoogleDesktop.e
2156	3472	Tue Aug 04 12:34:49 2009		0x071097c0	0x24d408e0	etlitr50.exe
2056	2448	Tue Aug 04 12:34:58 2009		0x07164da0	0x24d409a0	hidfind.exe
1696	1992	Tue Aug 04 12:39:35 2009		0x07191c98	0x24d40560	jucheck.exe
2736	2776	Tue Aug 04 12:34:48 2009		0x071988c0	0x24d408c0	GoogleDesktopIn
2872	3472	Tue Aug 04 12:34:55 2009		0x071f6da0	0x24d40940	WZQKPICK.EXE
1892	3472	Tue Aug 04 12:34:27 2009		0x072e8a10	0x24d40660	iFrmewrk.exe
3868	3472	Tue Aug 04 12:34:32 2009		0x0735d168	0x24d40540	WPC54GX4.exe
2008	2960	Tue Aug 04 12:34:58 2009		0x07379880	0x24d409c0	ApntEx.exe
2912	548	Tue Aug 04 12:34:04 2009		0x074205a8	0x24d40580	ZCfgSvc.exe
2052	3472	Tue Aug 04 12:35:35 2009		0x0743d378	0x24d408a0	OUTLOOK.EXE
2868	3472	Tue Aug 04 12:34:28 2009		0x07457828	0x24d406e0	DVDLauncher.exe
3924	840	Tue Aug 04 12:34:07 2009		0x075382d0	0x24d401c0	1XConfig.exe
2128	3472	Tue Aug 04 12:34:26 2009		0x0753f8f8	0x24d405a0	igfxpers.exe
1468	840	Tue Aug 04 12:34:27 2009		0x07579608	0x24d406a0	igfxsrvc.exe
360	3472	Tue Aug 04 12:34:41 2009		0x075a0020	0x24d40880	DLG.exe
748	3472	Tue Aug 04 12:34:30 2009		0x075bc020	0x24d40740	tfsctrl.exe
1960	592	Mon Aug 03 14:25:52 2009		0x0764a498	0x24d402e0	gsaLrt.exe
3472	2824	Tue Aug 04 12:34:06 2009		0x0773a838	0x24d40500	explorer.exe
2448	3472	Tue Aug 04 12:34:26 2009		0x07841da0	0x24d40600	Apoint.exe
376	3472	Tue Aug 04 12:34:26 2009		0x078598a8	0x24d405c0	hkcmd.exe

2156	548 Tue Aug 04 06:23:48 2009 Tue Aug 04 12:33:37 2009 0x079d58d8 0x24d405e0 Logon.scr
3232	3472 Tue Aug 04 12:34:28 2009 0x07a0b518 0x24d406c0 quickset.exe
656	3472 Tue Aug 04 12:34:56 2009 0x07a73638 0x24d40960 infoclient.exe
1992	3472 Tue Aug 04 12:34:34 2009 0x07d75b88 0x24d407e0 jusched.exe
2824	3472 Tue Aug 04 12:34:37 2009 0x07e0f7c8 0x24d40780 ccApp.exe
4052	3064 Tue Aug 04 12:33:59 2009 0x07f19508 0x24d40440 ssonsvr.exe
416	3088 Wed Jul 29 10:10:57 2009 Wed Jul 29 10:11:12 2009 0x07f5b288 0x24d40620 COH32.exe
3436	592 Wed Jul 29 09:09:35 2009 0x07fd85f0 0x24d40460 CcmExec.exe

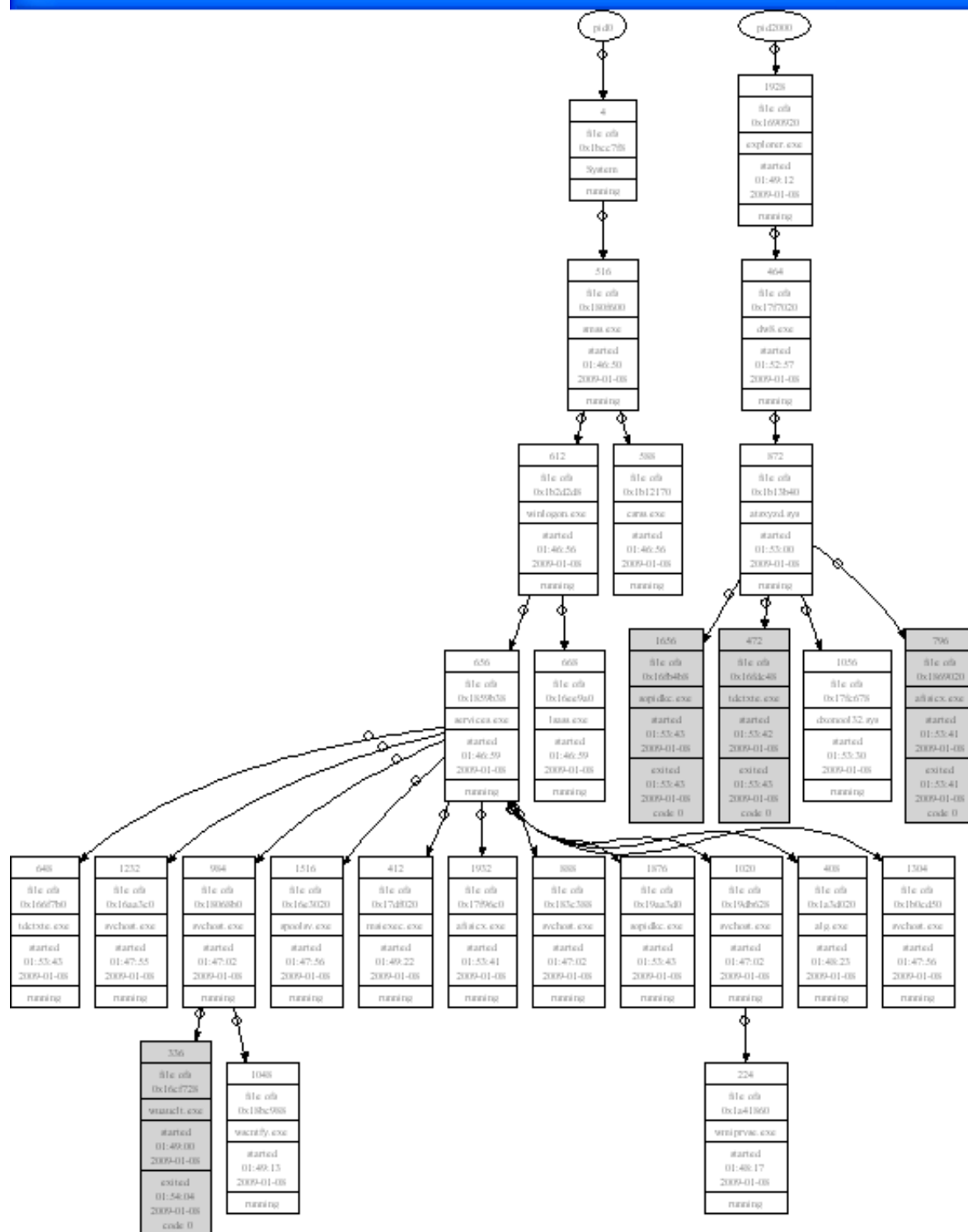
Notice that this list presents both the PID process id and the PPID (Parent PID). You can manually map PPID to PID to determine what process spawned another process. Now you can either mentally map this process tree or draw it out physically if that helps you, but there IS another way to visualize this process mapping.....

Have you taken a look at the options for the Volatility PSSCAN2 command? If you type “**python Volatility psscan2 -h**”, you’ll see a list of supported options for that Volatility command (and any other Volatility command). **Notice the -d option (“Print processes in dot format”)?**

Lets run **PSSCAN2** with the -d option and output the results to a file. By itself this output file is not going to be much good to you. In fact, if you are running Office, the output file might even be identified as a WORD DOT file type....it is NOT. It is a DOT IMAGE FORMATED file. In order to view the output of the dot format file, on Windows you will need download and install GraphViz-2.24 (stable) <http://www.graphviz.org/Download.php>. Select the appropriate source for you OS (Windows/Linux).

On Linux make sure you install all the dependencies files that are listed and then at the command terminal type “dotty psscan.dot” and the file will open. In Windows right click your psscan2.dot output file and choose to **OPEN WITH**. Now browse to the **/GraphViz/bin** folder and select “**dotty**.” Dotty will now display your DOT image **PSSCAN** output file.

By right clicking within the Dotty program, you can see different option such as Zoom in, Zoom Out, Print, etc. If you zoom in you can much more clearly see the relationships between PPID and PIDs.



REFERENCES

ReactOS, <<http://www.reactos.org/en/index.html>>.

Anand G. Internal structures of the Windows registry.

<<http://blogs.technet.com/ganand/archive/2008/01/05/internalstructures-of-the-windows-registry.aspx>>, 2008.

Carvey H. The Windows registry as a forensic resource. *Digital Investigation* 2005a; 2(3):201–5.

Carvey H. Registry mining. <<http://windowsir.blogspot.com/2005/01/registry-mining.html>>, 2005b.

Carvey H. Windows's forensic analysis. Norwell, MA, US: Syngress, ISBN 159749156X; 2007.

DFRWS. The DFRWS 2005 forensic challenge. <<http://www.dfrws.org/2005/challenge/index.html>>, 2005.

Dolan-Gavitt B. The VAD tree: a process-eye view of physical memory. *Digital Investigation*, <http://dfrws.org/2007/proceedings/p62-dolan-gavitt.pdf>, September 2007; 4:62–4.

Dolan-Gavitt B. Cell index translation. <<http://moyix.blogspot.com/2008/02/cell-index-translation.html>>, 2008a.

Dolan-Gavitt B. Enumerating registry hives. <<http://moyix.blogspot.com/2008/02/enumerating-registry-hives.html>>, 2008b.

Dolan-Gavitt B. Reading open keys. <<http://moyix.blogspot.com/2008/02/reading-open-keys.html>>, 2008c.

Dolan-Gavitt B. SysKey and the SAM. <<http://moyix.blogspot.com/2008/02/syskey-and-sam.html>>, 2008d.

F-Secure. F-Secure virus descriptions: slammer. <<http://www.f-secure.com/v-descs/mssqlm.shtml>>, 2003.

Farmer DJ. A forensic analysis of the Windows registry. <<http://eptuners.com/forensics/Index.htm>>, 2007.

Kent K, Chevalier S, Grance T, Dang H. NIST special publication 800-86: guide to integrating forensic techniques into incident response. 2006.

Macfarlane J. Parse:Win32Registry. <<http://search.cpan.org/jmacfarla/Parse-Win32Registry-0.30/>>.

Metasploit. Metasploit framework user guide. <http://www.metasploit.com/documents/users_guide.pdf>, 2008.

Microsoft Corporation. Windows registry information for advanced users.

<<http://support.microsoft.com/kb/256986>>, 2008.

National Institute of Standards and Technology (NIST). The CFReDS project. <<http://www.cfreds.nist.gov/>>.

Petroni Jr NL, Fraser T, Walters A, Arbaugh WA. Architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: USENIXSS' 06: Proceedings of the 15th Conference on USENIX Security Symposium. Berkeley, CA, USA: USENIX Association; 2006. p. 20.

Russinovich ME, Solomon DA. Microsoft Windows internals, Fourth edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (pro-developer). Redmond, WA, USA:

Microsoft Press, ISBN 0735619174; 2004. Samba. Regio library. <http://viewcvs.samba.org/cgi-bin/viewcvs.cgi/branches/SAMBA_4_0/source/lib/registry/>.

Schuster A. Searching for processes and threads in Microsoft Windows memory dumps. In: Proceedings of the sixth Annual Digital Forensic Research Workshop (DFRWS 2006). <<http://www.dfrws.org/2006/proceedings/2-Schuster.pdf>>, 2006.

Schuster A. PoolTools version 1.3.0. <http://computer.forensikblog.de/en/2007/11/pooltools_1_3_0.html>, 2007.

Stevens D. UserAssist. <<http://blog.didierstevens.com/programs/userassist/>>, 2006.

Walters A. FATKit: detecting malicious library injection and upping the “anti”, Technical report. 4TFResearch Laboratories; July 2006.

Walters A. The Volatility framework: volatile memory artifact extraction utility framework. <<https://www.volatilesystems.com/default/volatility>>, 2007.

Walters A, Petroni NL, Jr., Volatools: integrating volatile memory forensics into the digital investigation process. In: Black Hat DC; 2007.

Websites:

<http://computer.forensikblog.de/>

<http://mnin.blogspot.com/2009/12/new-and-updated-volatility-plugin-ins.html>

http://computer.forensikblog.de/en/2009/04/searching_for_mutants.html#more

<http://gleeda.blogspot.com/>

<http://moyix.blogspot.com/>

<http://scudette.blogspot.com/2008/10/pstree-volatility-plugin.html>

<http://forensiczone.blogspot.com/2009/10/walk-through-volatility-batch-file.html>