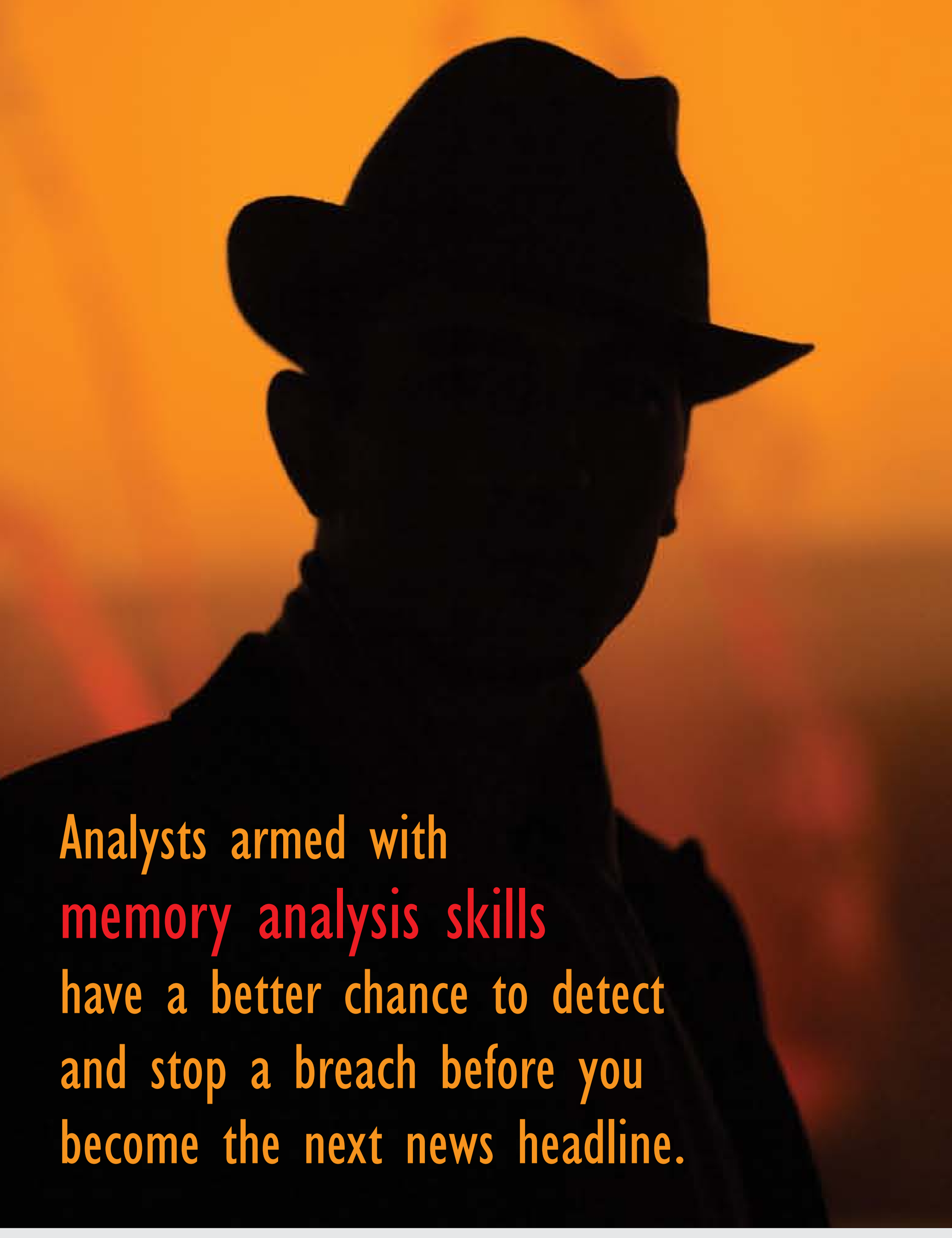


Stop being blind to the adversary – Arm your incident response team today!

Memory Analysis will prepare your team to:

- Discover zero-day malware
- Detect compromises
- Uncover evidence that others miss

Analysts armed with memory analysis skills have a better chance to detect and stop a breach before you become the next news headline.

SANS DFIR
DIGITAL FORENSICS & INCIDENT RESPONSE

Memory Forensics POSTER

Malware Can Hide, But It Must Run

digital-forensics.sans.org

SPRING 2015 – 36TH EDITION – $25.00

FIND EVIL WHERE IT LIES

Memory Forensics

Essential in Effective Incident Response Today

sans.org/FOR526

Live Memory Analysis with Rekall

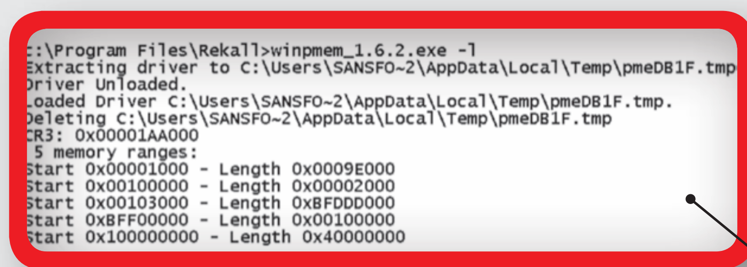
Rekall

rekall-forensic.com

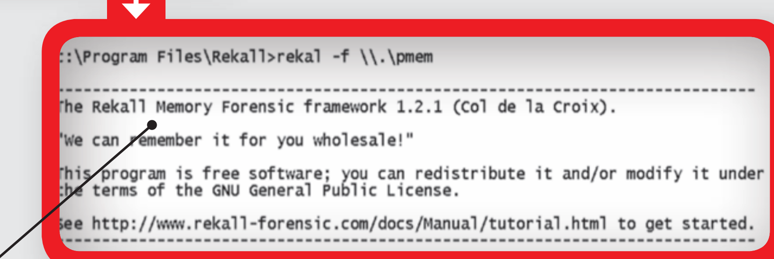
The Rekall Memory Forensic Framework is a collection of memory acquisition and analysis tools implemented in Python under the GNU General Public License. It originated in 2011 as the "Technology Preview" branch of the Volatility® Framework, with goals of streamlining code and improving efficiency, performance and usability. Code differences over years of development made it difficult to remerge the Volatility Framework with this rapidly developing branch, so the developers deemed it necessary to fork the project in Dec 2013 The Rekall Framework has been included in the development of Google Rapid Response, a live enterprise IR/forensics triage tool.

Some of the key differences that most analysts notice with Rekall is its ease of use, as it does not require the specification of a Windows target system profile when invoking a plugin. Rekall uses an alternative means of deciphering the profile of the Windows system other than reading the KDBG (Kernel Debugging Data Block). Rekall also uses interactive analysis sessions that cache information in memory, allowing data to remain available for increased speed in subsequent module analysis. Rekall's most exciting feature is its ability to work with winpmem for LIVE system memory analysis - further reducing the time responders must take in triaging a possibly compromised system.

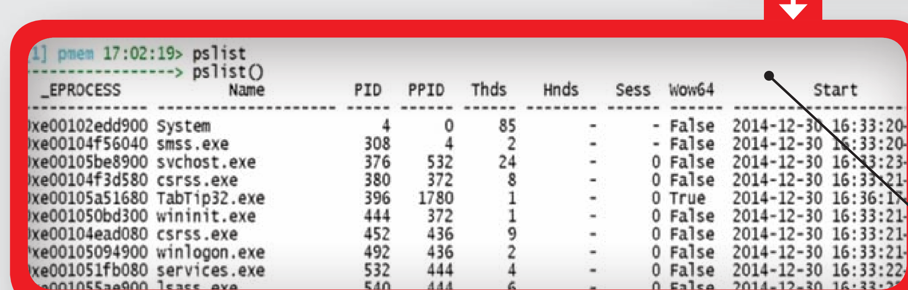
rekall-forensic.com



1 Load the winpmem driver from an administrative cmdshell



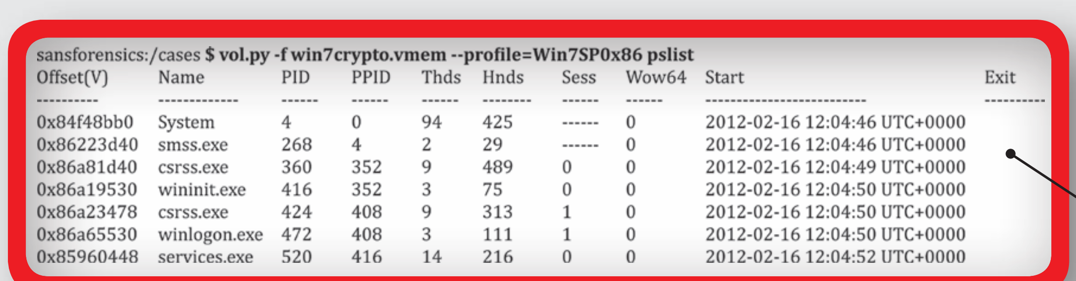
2 Launch Rekall with access to \\.\pmem



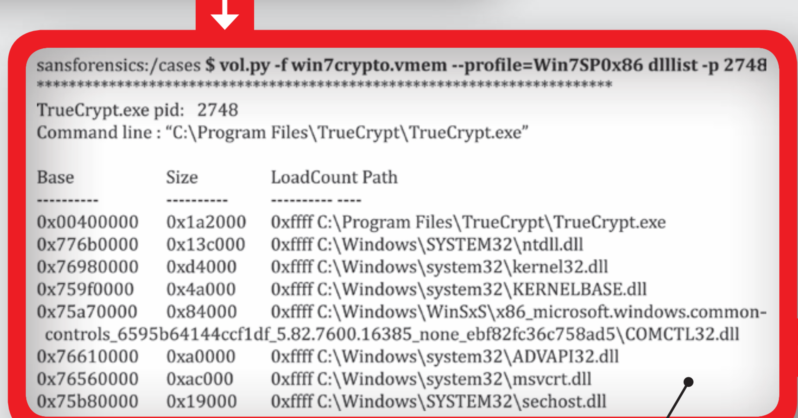
3 Conduct Analysis with Rekall Plugins

How to Parse a Memory Image with the Volatility® Framework

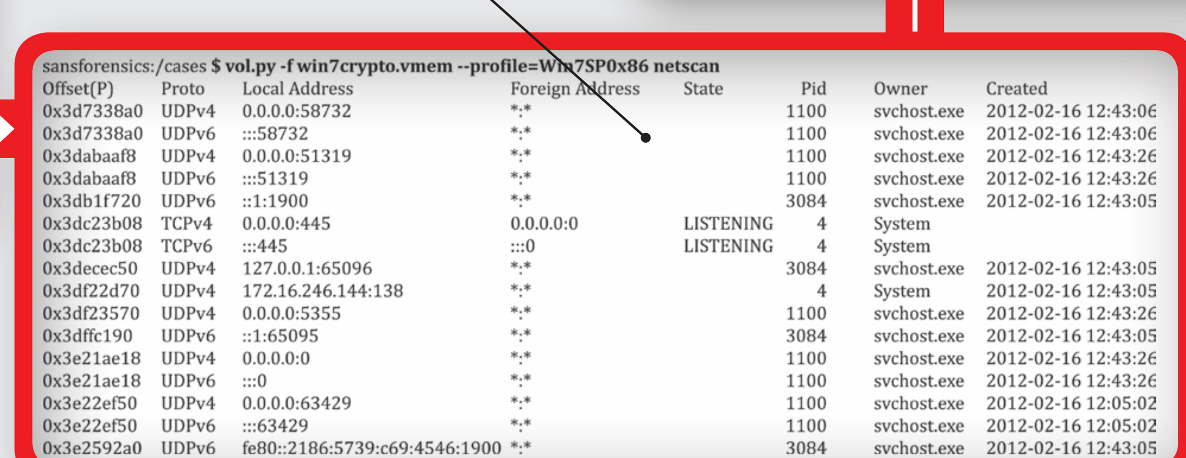
Six-Step Investigative Methodology Walkthrough



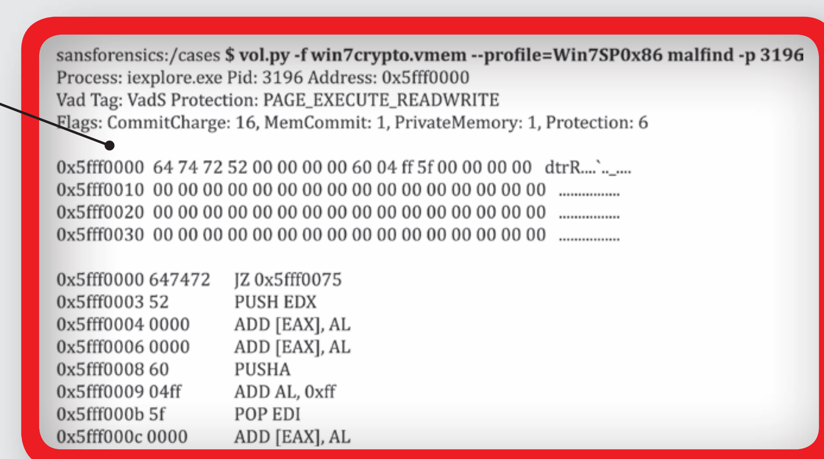
1 Identify rogue processes



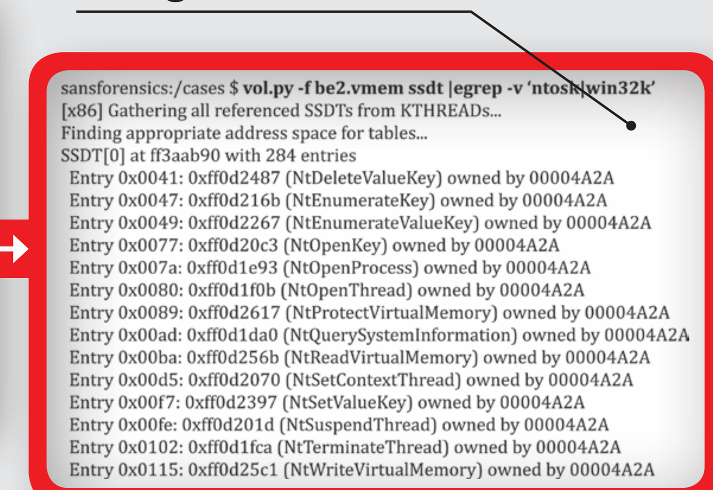
2 Analyze process DLLs and handles



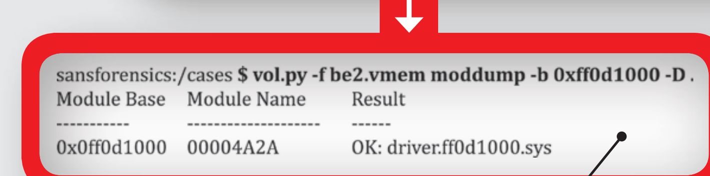
3 Review network artifacts



4 Look for evidence of code injection

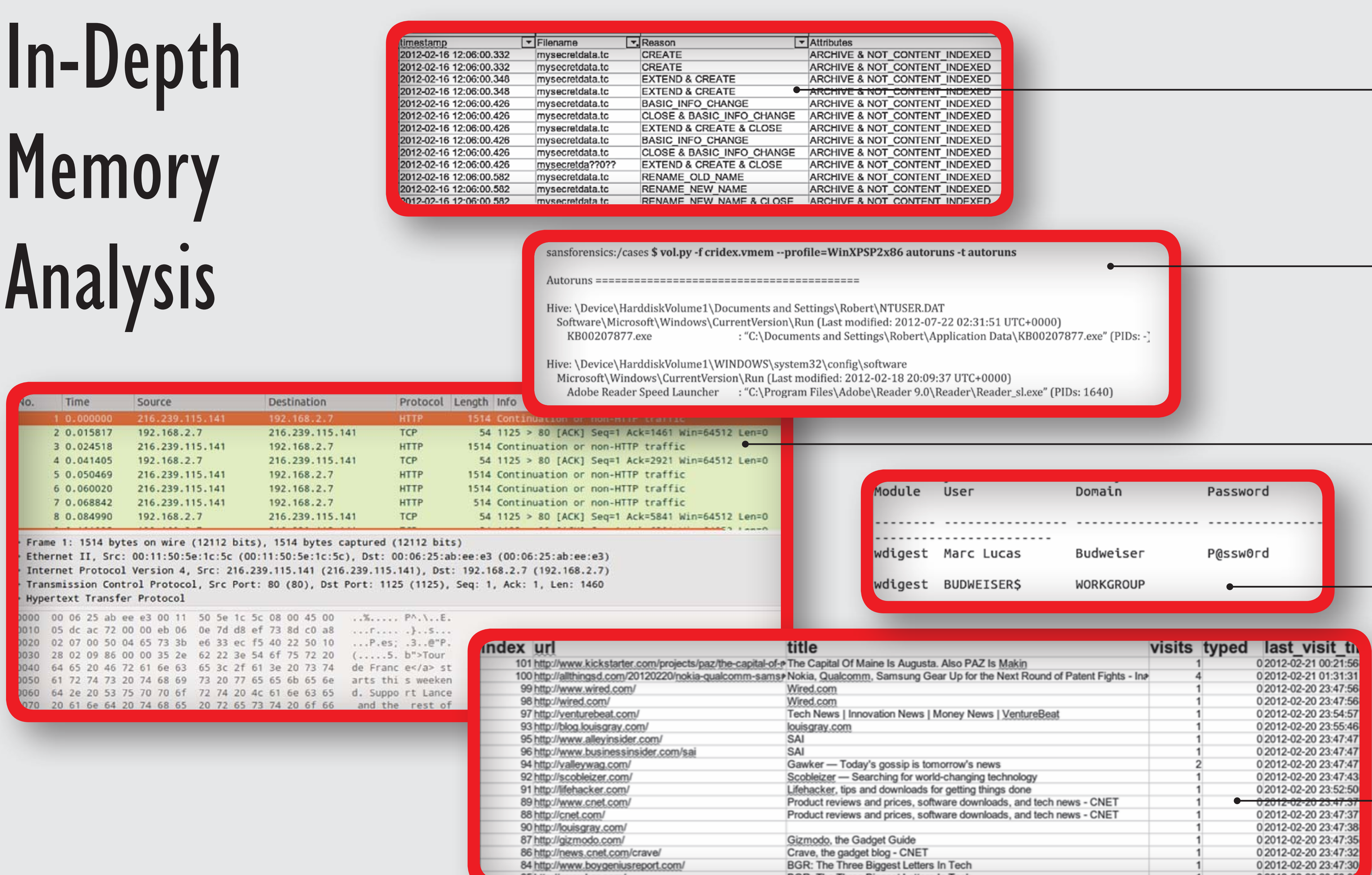


5 Check for signs of a rootkit



6 Dump suspicious processes and drivers

In-Depth Memory Analysis



Retrieve USN Journal Entries USNParser *by Tom Spencer*

The $USNJournal can hold trace artifacts for files/directories that USED to be present on a volume. Spencer's plugin carves entries from a memory image – awesome!

```
$ vol.py -f system_image.vmem --profile=Win7SP1x64 usnparser --output=csv --output-file=usn.csv -CS
```

Identify Persistence Mechanisms AutoRuns *by Thomas Chopitea*

Identifying persistence mechanisms implemented on a compromised system to reinstantiate a malicious process after a reboot or process termination can be performed in a variety of ways. Using only a memory image of that system, we can extract some of the most common persistence mechanisms used by malware today by invoking Chopitea's autoruns plugin.

```
$ vol.py -f system_image.vmem --profile=Win7SP1x64 autoruns -t autoruns
```

Extract Network Packets ethscan *by Jamaal Speights*

In some investigations, the sole source of network traffic must be carved out of the system memory image. Using Speights' plugin, we are able to extract network packets from memory, with an output option ("-C") of creating a pcap file.

```
$ vol.py -f system_image.vmem --profile=Win7SP1x64 ethscan --save-pcap=out.pcap
```

Extract Plaintext Passwords mimikatz *by Francesco Picasso*

A huge thanks to Benjamin Delpy for creating the mimikatz plaintext credential harvester that acts on the lsass process to extract usernames and passwords from interactive sessions of the target system. Picasso has given us easy access to this functionality on memory images through his Volatility plugin.

```
$ vol.py -f system_image.vmem --profile=Win7SP1x64 mimikatz
```

Reconstruct Browser History Chrome/Mozilla *by John Lassalle*

Browser forensics may reveal suspicious web activity, and there are multiple tools that examiners can use to reconstruct browser artifacts from disk. What is the gain in reconstructing browser history from memory? One case is when the browser history is trapped in a hibernation file but has since been deleted by the user (this may even indicate intent). Lassalle's Chrome and Mozilla plugins grant easy access to these artifacts.

```
$ vol.py -f system_image.vmem --profile=Win7SP1x64 firefoxhistory --output=csv --output-file=firefox.csv
```

SANS DFIR
DIGITAL FORENSICS & INCIDENT RESPONSE

Memory Forensics

MALWARE CAN HIDE, BUT IT MUST RUN

This poster shows some of the structures analyzed during memory forensic investigations. Just as those practicing disk forensics benefit from an understanding of filesystems, memory forensic practitioners also benefit from an understanding of OS internal structures. The internal structures detailed in the poster are the most important in most investigations, but by no means are they complete. Similarly, each structure has far more members than are shown on the poster. Some structures have hundreds of members. We have again chosen to show those that are most useful to our investigations.

FOR526
Memory Forensics
In-Depth
AUTHORS:

Alissa Torres *@sibertor*
Jake Williams *@malwarejake*

Memory analysis is now a crucial skill for any incident responder who is analyzing intrusions. The malware paradox is key to understanding that while intruders are becoming more advanced with anti-forensic tactics and techniques, it is impossible to hide their footprints completely from a skilled incident responder performing memory analysis.

Learn more about **FOR526: Memory Forensics In-Depth** at **sans.org/FOR526**

_EPROCESS

The _EPROCESS is perhaps the most important structure in memory forensics. As opposed to the KDBG (used only by Volatility), it is also used by Rekall. The _EPROCESS structure has more than 100 members, many of them pointers to other structures.

The _EPROCESS gives us the PID and parent PID of a given process. Analyzing PID relationships between processes can reveal malware. For more information, see the SANS DFIR poster "Know Normal, Find Evil."

The _EPROCESS block also contains the creation and exit time of a process. Why would the OS keep track of exited processes? The answer is that when a process exits, it may have open handles which must be closed by the OS. The OS also needs time to gracefully deallocate other structures used by the process. The ExitTime field allows us to see that a process has exited but has not yet been completely removed by the OS. Note that the task manager and other live response tools will not show exited processes at all, but they are easy to see with the use of memory forensics!

VAD

VADs (Virtual Address Descriptors) are used by the memory manager to track ALL memory allocated on the system. Malware and rootkits can hide from a lot of different OS components, but hiding from the memory manager is unwise. If it can't see your memory, it will give it away!

Process Environment Block

The PEB contains pointers to the _PEB_LDR_DATA structure (discussed below). It also contains a flag that tells whether a debugger is attached to a process. Some malware will debug a child process as an anti-reversing measure. Finally, the PEB also contains a pointer to the command line arguments that were supplied to the process on creation.

PLUGINS: `modules`, `ldrmodules`, `dlllist`, `pstree -v`

Unloaded Modules

The Windows OS keeps track of recently unloaded kernel modules (device drivers). This is useful for finding rootkits (and misbehaving legitimate device drivers).

_MMVAD

- **LeftChild** – Pointer to the left VAD child
- **RightChild** – Pointer to the right VAD child
- **StartingVpn** – Starting address described by VAD
- **EndingVpn** – Ending address described by VAD
- **VadsProcess** – Pointer to the _EPROCESS block that owns this VAD

System Process DTB (directory table base)

The directory table base of a process points to the base of the page directory table (sometimes called the page directory base, or PDB). The CR3 register points to this location, which is unique per process. From the DTB, the complete list of the processes' page tables can be discovered. Rekall locates the DTB for the Idle process (the Idle process is really just an accounting structure) and then uses this to find the image base of the kernel. Then, the KDBG (if needed at all) can be found deterministically, rather than using the scanning approach to find the KDBG used by Volatility. From the Idle process DTB, all other required structure offsets can be determined.

Process Struct (_EPROCESS)

- **Pcb** – Process control block
- **CreateTime** – Time when the process was started.
- **ExitTime** – Exit time of the process – process is still stored in the process list for some time after it exits. It allows for graceful deallocation of other process structures.
- **UniqueProcessId** – PID of the process
- **ActiveProcessLinks** – Doubly linked list to other process' EPROCESS structures (process list)
- **ObjectTable** – Pointer to the process' handle table
- **Peb** – Pointer to the process environment block
- **InheritedFromUniqueProcessId** – The parent PID
- **ThreadListHead** – List of active threads (_ETHREAD)
- **VadRoot** – Pointer to the root of the VAD tree

Unloaded Drivers

- **Name** – Driver name
- **StartAddress** – Start address where driver was loaded
- **EndAddress** – End address where driver was loaded
- **CurrentTime** – Time when driver was unloaded

Process Environment Block (_PEB)

- **BeingDebugged** – Is a debugger attached to the process
- **ImageBaseAddress** – Virtual address where the executable is loaded
- **Ldr** – Pointer to _PEB_LDR_DATA structure
- **ProcessParameters** – Full path name and command-line arguments

Kernel Debugger Data Block (_KDDEBUGGER_DATA64)

- **PsLoadedModuleList** – Pointer to the list of loaded kernel modules
- **PsActiveProcessHead** – Pointer to the list head of active processes
- **PspCidTable** – Table of processes used by the scheduler
- **MmUnloadedDrivers** – List of recently unloaded drivers

PEB Loader Data (_PEB_LDR_DATA)

- **InLoadOrderModuleList** – List of loaded DLLs
- **InMemoryOrderModuleList** – List of loaded DLLs
- **InInitializationOrderModuleList** – List of loaded DLLs

_LDR_DATA_TABLE_ENTRY

- **DllBase** – The base address of the DLL
- **EntryPoint** – Entry point of the DLL
- **SizeOfImage** – Size of the DLL in memory
- **FullDllName** – Full path name of the DLL
- **TimeDateStamp** – The compile time stamp for the DLL

PsLoadedModuleList

The PsLoadedModuleList structure of the KDBG points to the list of loaded kernel modules (device drivers) in memory. Many malware variants use kernel modules because they require low level access to the system. Rootkits, packet sniffers, and many keyloggers use may be found in the loaded modules list. The members of the list are _LDR_DATA_TABLE_ENTRY structures. Stuxnet, Duqu, Regin, R2D2, Flame, etc. have all used some kernel mode module component – so this is a great place to look for advanced (supposed) nation-state malware. However, note that some malware has the ability to unlink itself from this list, so scanning for structures may also be necessary.

ThreadListHead

Where are the thread list structures on the poster? Sorry, we just don't have room to do them justice. But most investigations don't require us to dive into thread structures directly. Threads are still important though. In Windows, a process is best thought of as an accounting structure. The Windows scheduler never deals with processes directly, rather it schedules individual threads (inside a process) for execution. Still, you'll find yourself using process structures more in your investigations.

PEB Loader Data

This structure contains pointers to three linked lists of loaded modules in a given process. Each is ordered differently (order of loading, order of initialization, and order of memory addresses). Sometimes malware will inject a DLL into a legitimate Windows service and then try to hide. But they'd better hide from all three lists or you'll detect it with no trouble.

PLUGINS: `ldrmodules`

ObjectTable

For a process in Windows to use any resource (registry key, file, directory, process, etc.) it must have a handle to that object. We can tell a lot about a process just by looking at its open handles. For instance, you could potentially infer the log file a keylogger is using or persistence keys used by the malware, all by examining handles.

_LDR_DATA_TABLE_ENTRY

This structure is used to describe a loaded module. Loaded modules come in two forms. The first is the kernel module (aka device driver). The second type of loaded module are dynamic link libraries (DLLs), which are loaded into user mode processes.

PLUGINS: `modules`, `ldrmodules`, `dlllist`

DFIR

DFIR Resources	**DFIR Blog**	**DFIR Twitter**	**DFIR Facebook**	**DFIR Google+**	**DFIR Mailing List**
digital-forensics.sans.org	digital-forensics.sans.org/blog	@sansforensics	sansforensics	gplus.to/sansforensics	dfir.to/MAIL-LIST

Note that many internal OS structures are doubly linked lists. The pointers in the lists actually point to the pointer in the next structure. However, for clarity of illustration, we have chosen to show the type of structure they point to. Also, note that the PsActiveProcessHead member of the KDBG structure points to ActiveProcessLinks member of the _EPROCESS structure. However, for clarity we depict the pointer pointing to the base of the _EPROCESS structure. We feel that this depiction more clearly illustrates the relationship between the various structures.