508.2

Memory Forensics in Incident Response & Threat Hunting

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

FOR508.2 Advanced Digital Forensics and Incident Response

Memory Forensics in Incident Response and Threat Hunting

Author: Rob Lee
rlee@sans.org
http://twitter.com/robtlee
http://twitter.com/sansforensics

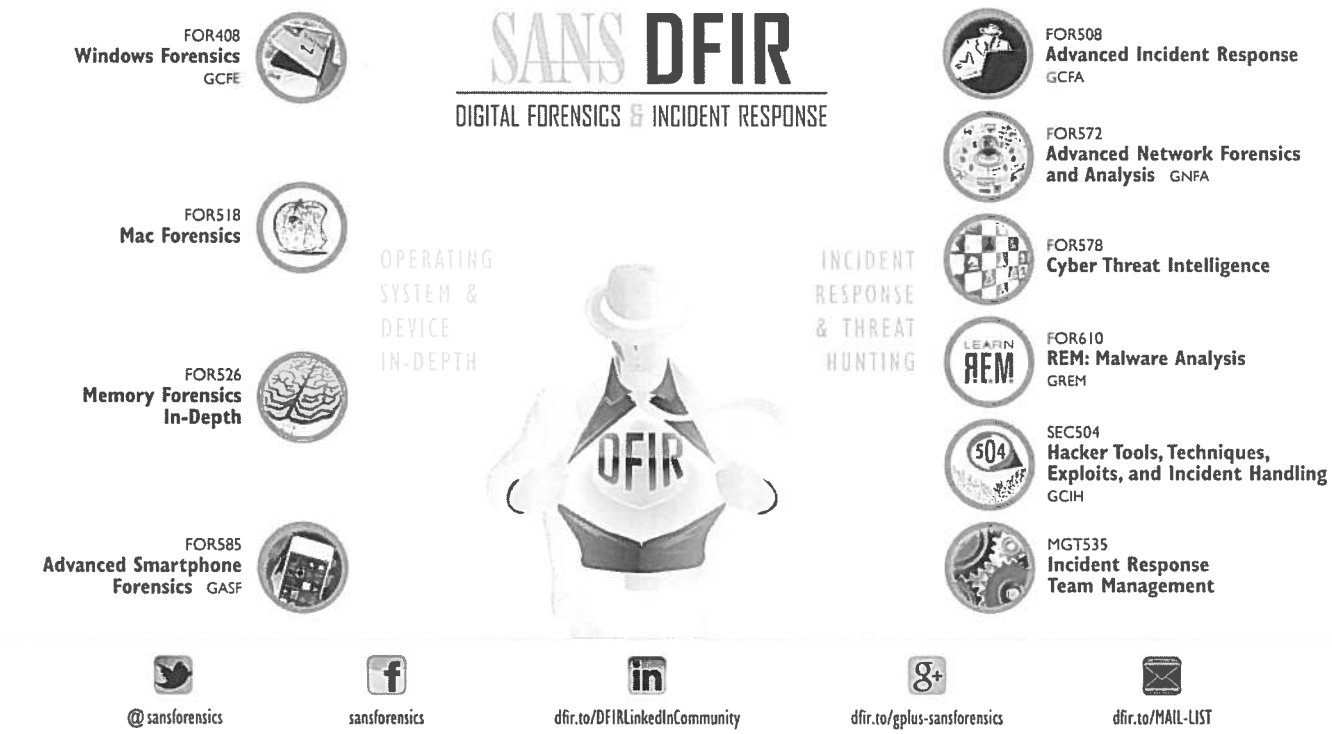
Author: Chad Tilbury
ctilbury@sans.org
http://twitter.com/chadtilbury

http://digital-forensics.sans.org

Optional Exercise 2.1

Optional Redline Pre-Process

Feel free to get started on this exercise while you are waiting for class to start! If you are unable to complete it, do not worry. There will be a pre-cooked version available in the next exercise.



This page intentionally left blank.

SANS DFIR

DIGITAL FORENSICS & INCIDENT RESPONSE

OPERATING SYSTEM & DEVICE IN-DEPTH

FOR408
Windows Forensics

FOR518
Mac Forensics

FOR526
Memory Forensics In-Depth

FOR585
Advanced Smartphone Forensics

INCIDENT RESPONSE & ADVERSARY HUNTING

FOR508
Advanced Incident Response

FOR572
Advanced Network Forensics and Analysis

FOR578
Cyber Threat Intelligence

FOR610
REM: Malware Analysis

SEC504
Hacker Tools, Techniques, Exploits, and Incident Handling

MGT535
Incident Response Team Management

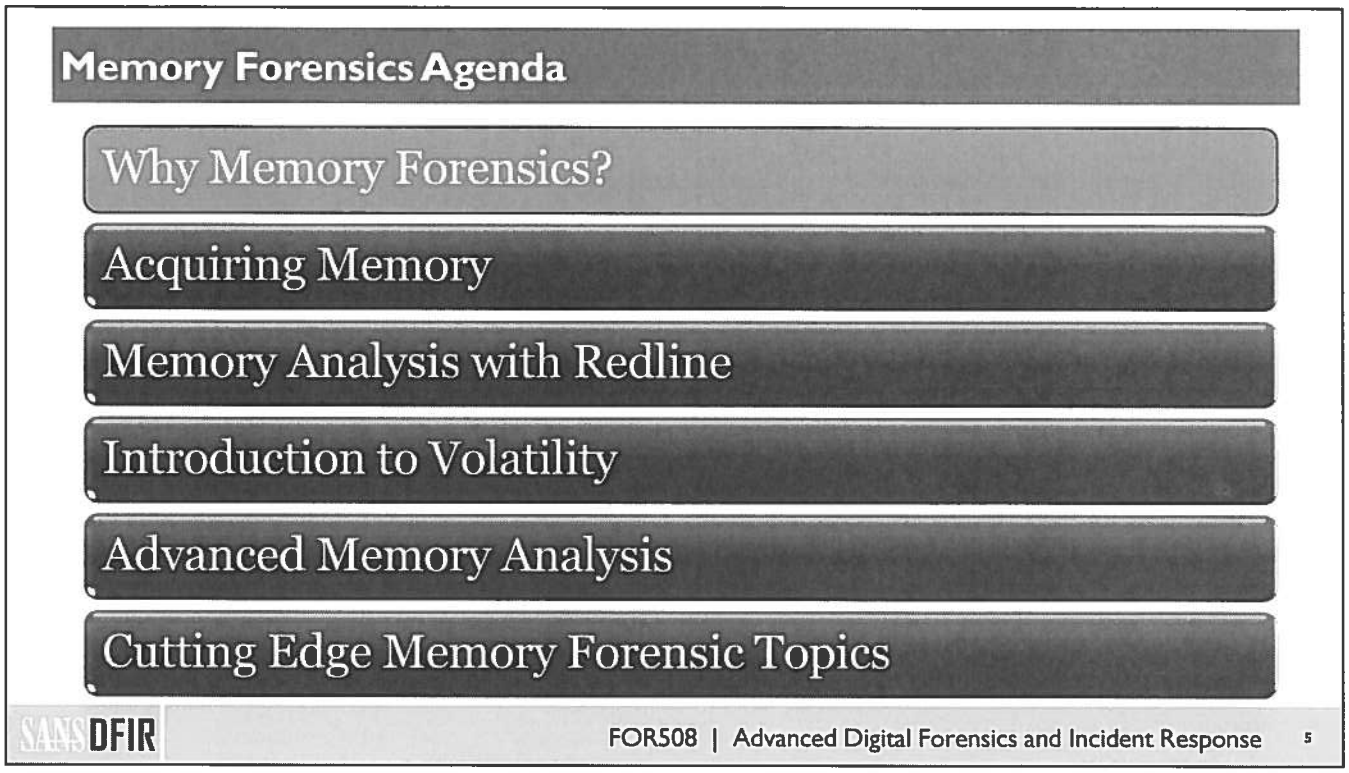
@sansforensics

sansforensics

dfir.to/DFIRLinkedInCommunity

dfir.to/gplus-sansforensics

dfir.to/MAIL-LIST



This page intentionally left blank.

Where We Left Off – Breach Status Update

Incident Response Total Time Elapsed: ~60 Min

Known Hosts Compro...

Name	IP	Function
nromanoff	10.3.58.5	Workstation

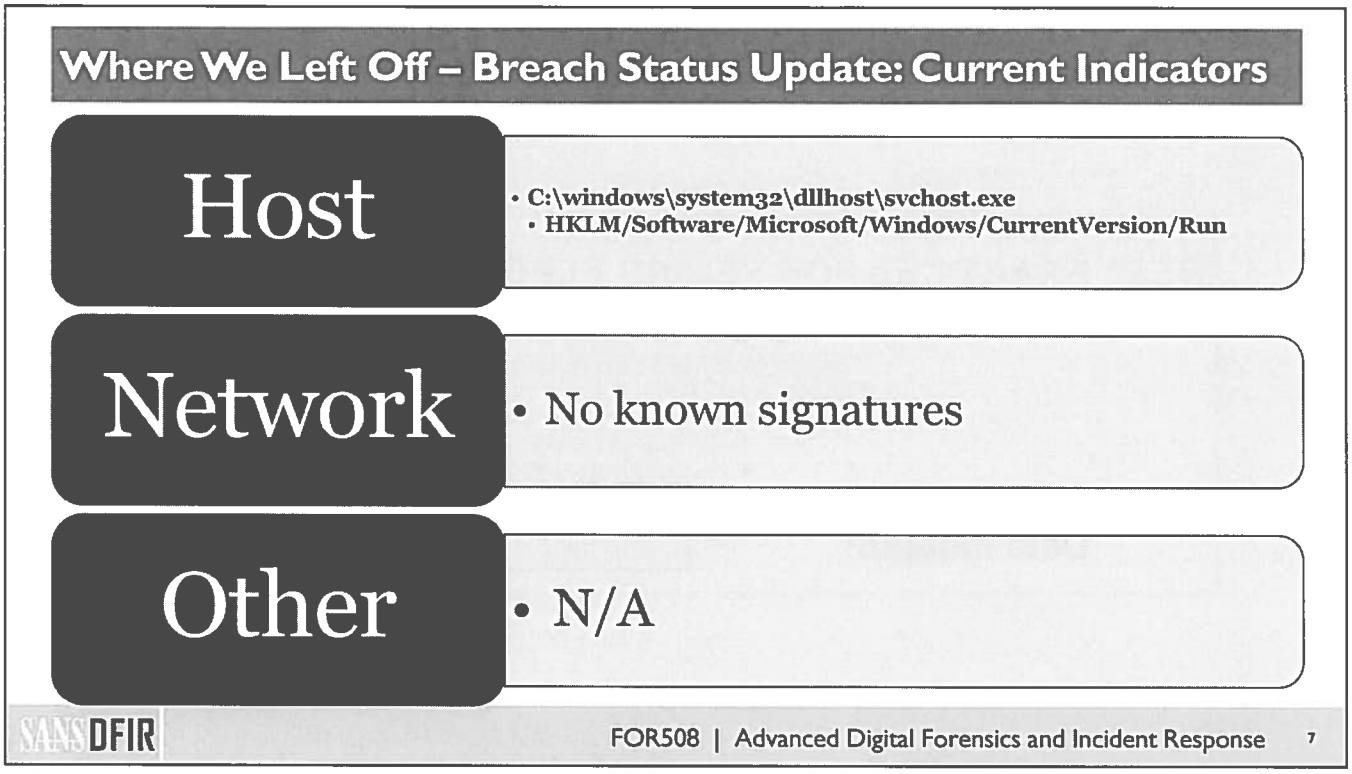
Initial IRT Call & Agent Deployment

- Access Host
  - Memory
  - C-Drive
  - Autostart Locations Examination
- Time: ~60 min

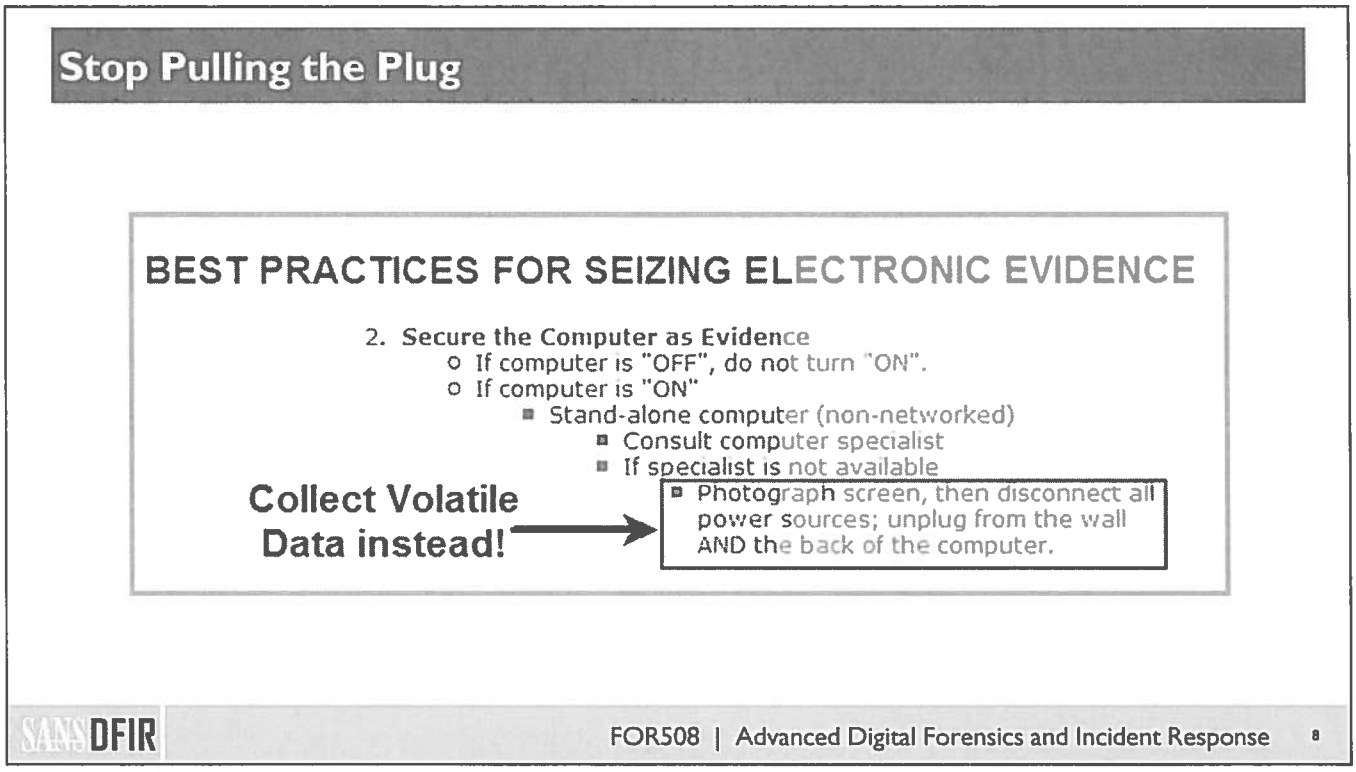
Current Spreadsheet o' Doom

- 10.3.58.5 – nromanoff

This page intentionally left blank.



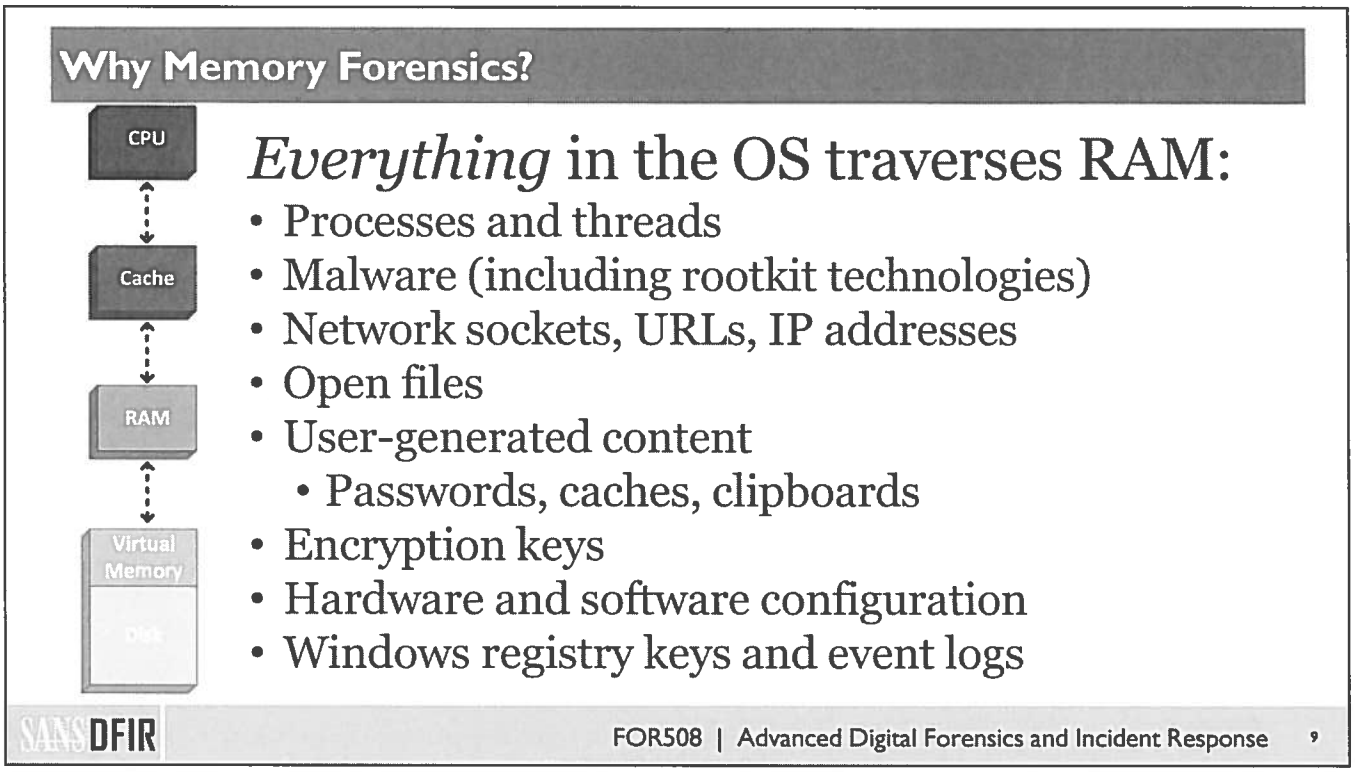
This page intentionally left blank.



Stop Pulling the Plug

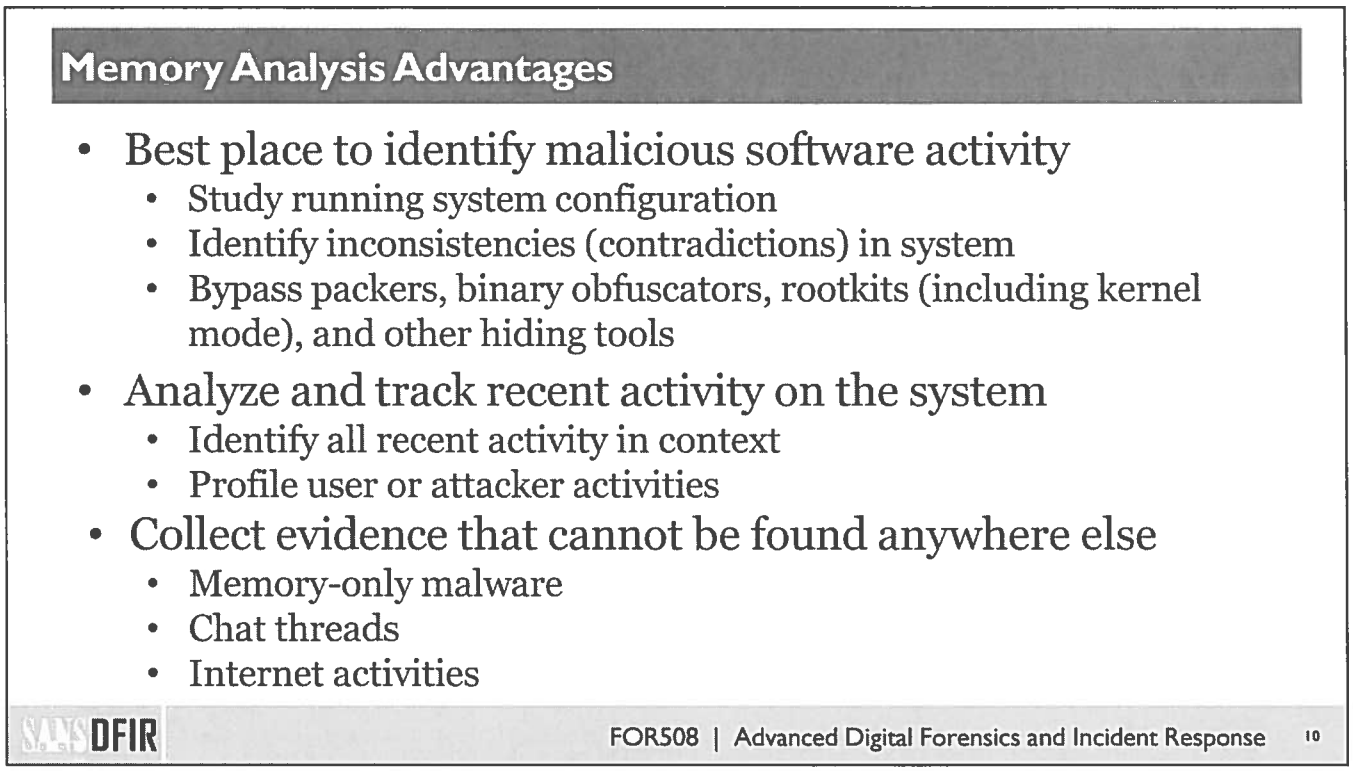
Over the past several years, many tools have been released that have focused on memory acquisition from Windows systems. Acquisition is great, but the real holy grail has always been memory analysis. Starting with the DFRWS 2005 challenge, memory forensics began a life that went beyond a rudimentary string search or data carve. Analysts were finally able to extract process-related data from memory captured from a machine.

In 2008, this culminated with many professionals stating at the SANS Forensic Summit that the day of "pulling the plug" during evidence acquisition is no longer acceptable. In fact, in our discussions with law enforcement, many are already obtaining memory captures during evidence seizure. Still, there are a vast number of organizations not taking advantage of memory forensics. The United States Secret Service Guide for first responders has long held that "pulling the plug" is the best option. To be fair, this is largely because this manual is distributed to a vast number of law enforcement officers—many of whom have absolutely no computer training. Our goals in this section are to open your eyes to how easy memory is to acquire and how valuable memory analysis can be to your organization.



Why Memory Forensics?
RAM is the bridge among the CPU, operating system, and getting things done. Nearly everything of interest that has ever happened on a modern computer has traversed RAM. From files to network connections to registry hives to running malware, a wealth of data is available for analysis.

Until recently, memory analysis was largely limited to performing string and byte searches through seemingly random data. Now, memory structures are better understood and new tools exist that allow for a more granular approach to examining the contents of memory. Just what is sitting in memory? You have all the processes, files, directories, and drivers in addition to a hoard of memory-mapped residue. You can use this information to piece together history and commands that a previous individual might have typed on the system. You might discover old e-mails or the malicious websites that the user surfed to. You might find residue from old, exited applications. And if you are lucky, you might have clear-text passwords still sitting in memory.

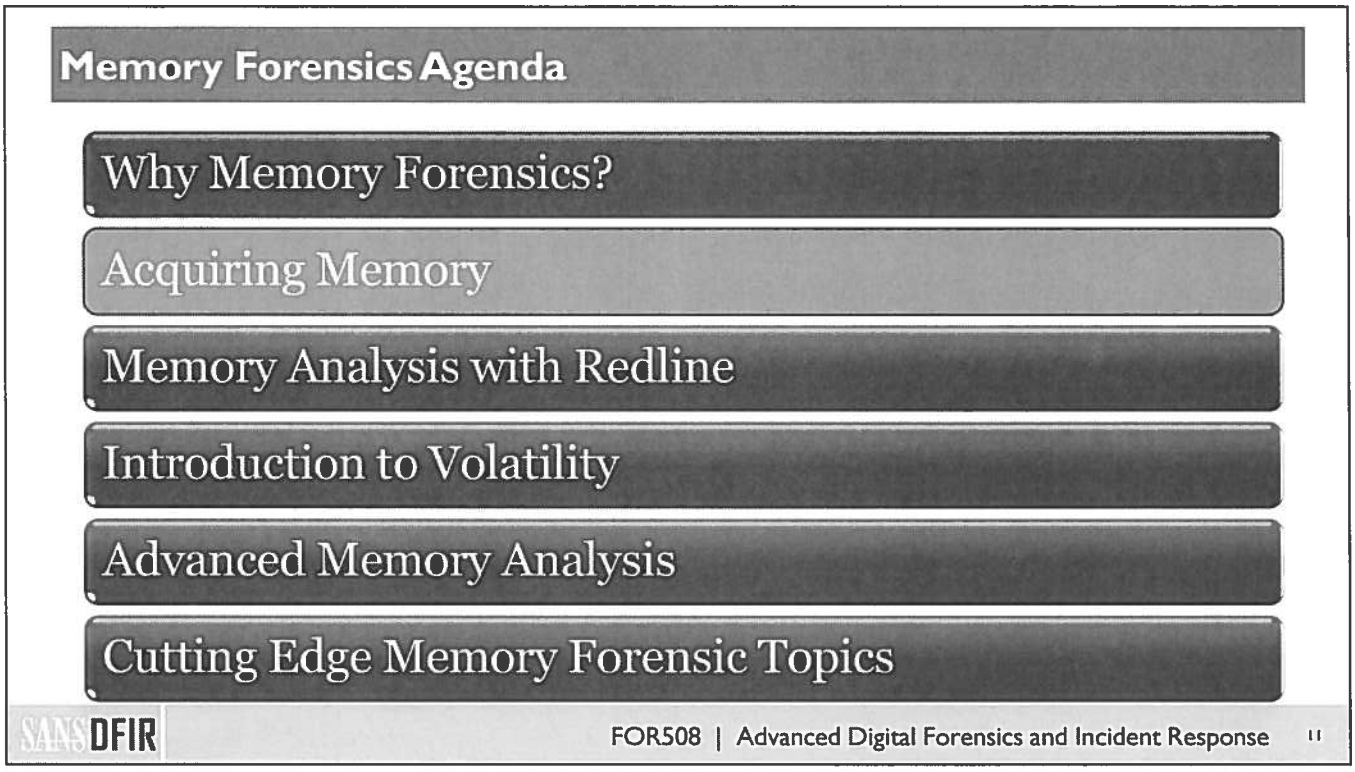


Memory Analysis Advantages

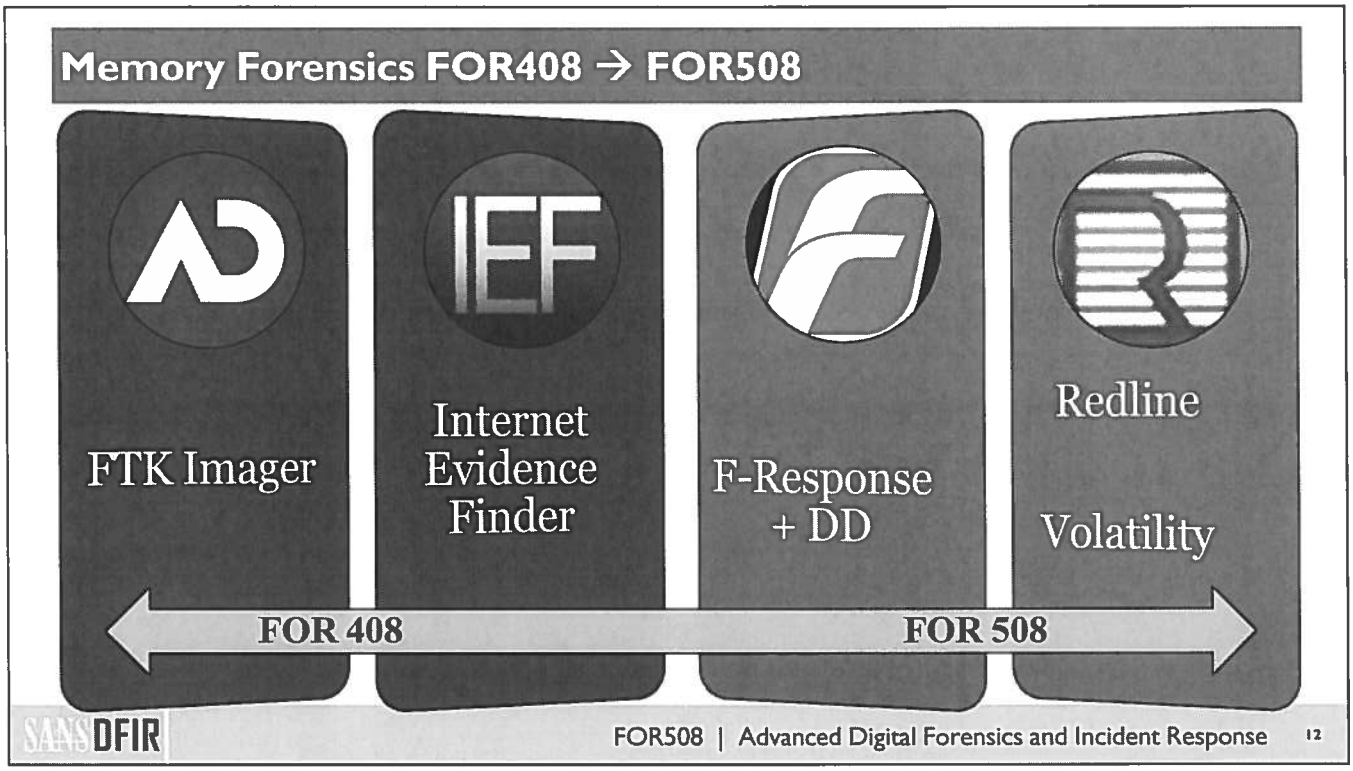
Currently, there is no better place to discover malware than in RAM. There is literally no place left for it to hide. The classic malware dilemma manifests perfectly in memory: Malware wants to hide, but it also has to execute to be effective. Malware might be successful at either hiding or executing, but it is nearly impossible to do both!

With the size of system memory steadily increasing, it is becoming less volatile and more like a secondary file system. We will learn how to identify running processes and threads along with all of their associated DLLs, files, and registry key handles. Seeing how something behaves in memory gives far more information than trying to piece the hundreds of different parts together from disk.

There are a number of key artifacts that might exist only in memory. Advanced malware exists that attempts to never write to disk. The majority of chat applications do not log communications to disk by default. And as users become savvier about privacy, memory might be our best bet for finding artifacts from things like "In-Private" or "Incognito" browsing sessions. Even the Windows registry takes on a life of its own within memory—there are volatile registry keys that can be updated and survive only in memory.



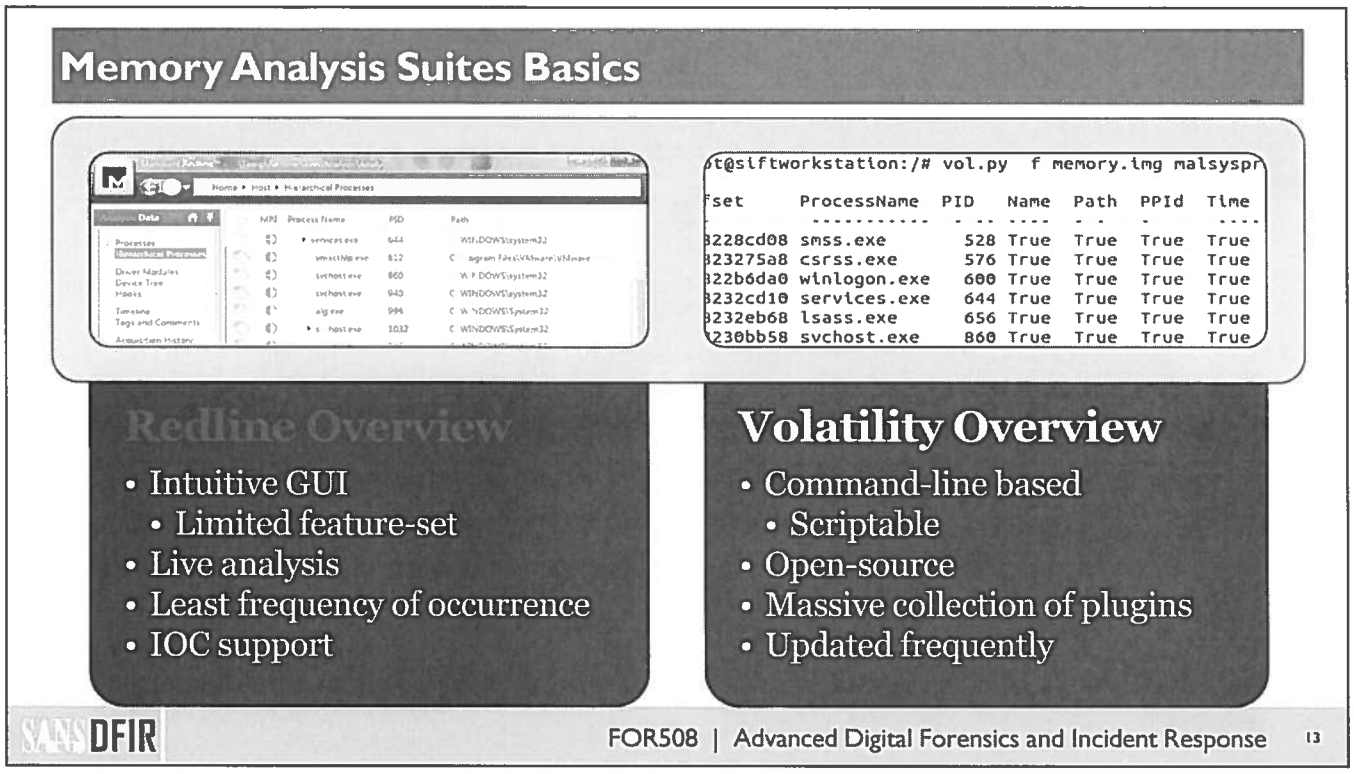
This page intentionally left blank.



Memory Forensics: FOR408 → FOR508

We started our discussion of memory analysis in the SANS Forensics 408 course, Windows Forensics In-Depth. You might recall running Access Data's FTK Imager Lite from a USB thumb drive to collect volatile memory and data. After memory acquisition, we performed rudimentary analysis using string searching and data carving. We subsequently used a specialized data carving utility called Internet Evidence Finder to search the recesses of memory for markers indicating chat, webmail, and "In Private" browsing activities.

Although all of these steps are still viable and valuable, the Forensics 508—Advanced Incident Response and Digital Forensics—class aims to take your memory analysis skills to the next level. We will discuss command-line memory acquisition alternatives like Win32/64dd, which in conjunction with F-Response can give us a network-based enterprise memory dumping capability. Then, we will jump into the details of exactly how a memory image is structured, what we can find, and develop a process for performing a detailed analysis. Along the way, we will discuss advanced topics like rootkit analysis, code injection, live memory analysis, and work with some of the Volatility project's newest tools for registry, event log, and timeline analysis.

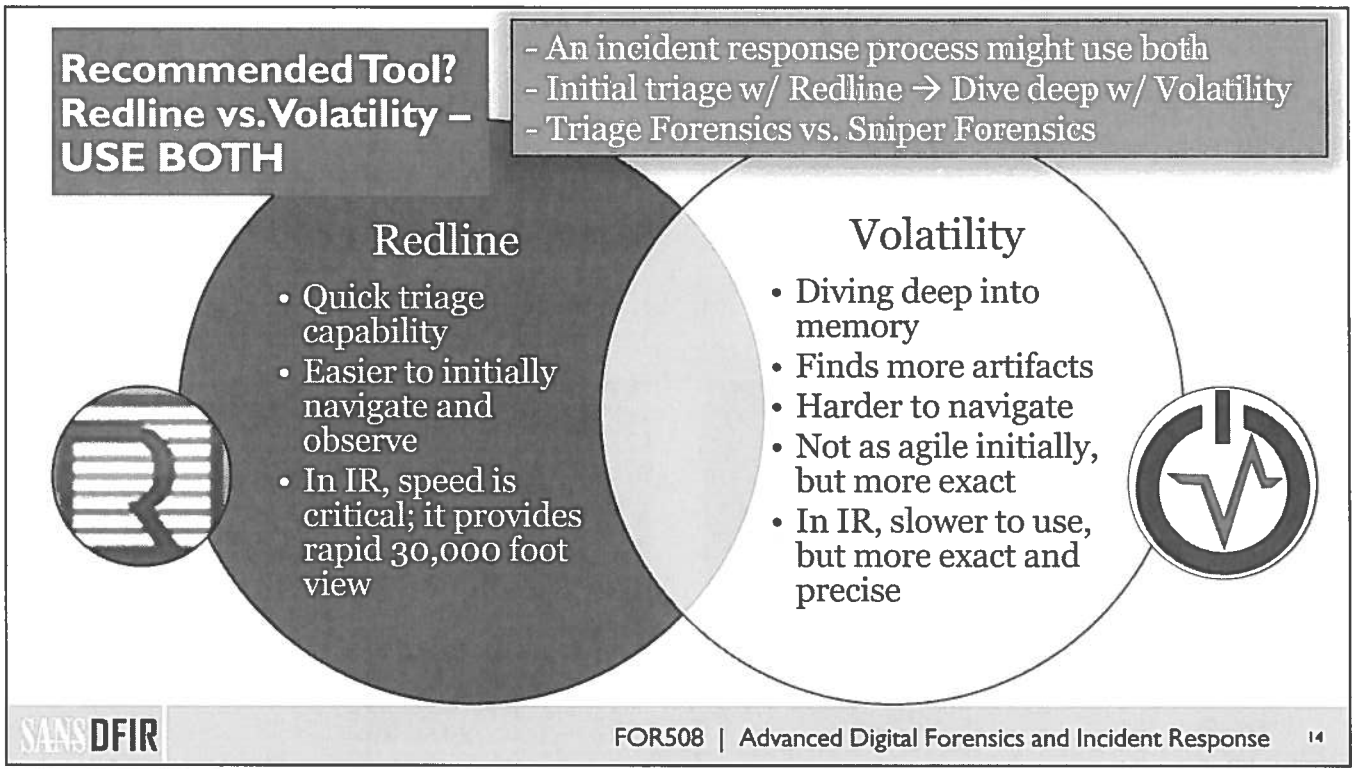


Memory Analysis Suites

As we begin our foray into memory analysis, we introduce two very different memory analysis suites. Both are best-in-class and both have pros and cons associated with them. Redline is an excellent tool for quick triage of a memory image. It front-loads all processing, making it quick to pivot through a variety of artifacts to identify areas of interest. Its GUI interface provides excellent sorting and filtering capabilities, and its malware rating index and support of the OpenIOC indicator format can help automate memory detection. However, with the nice pretty interface comes the downside of the tool not being extensible and thus having only a subset of the features now present in the Volatility framework.

Volatility is by far the most well-supported and powerful memory analysis tool. It is command-line based and hence does have an associated learning curve. Because it is open-source and plugin-based, a cadre of developers from across the globe continue to contribute new features. It also allows a very deep-dive into memory structures, facilitating memory research in addition to memory forensics.

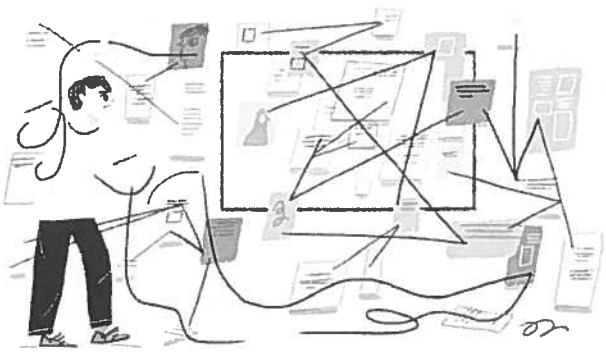
The good news is you do not have to pick one over the other. A suggested workflow might have an analyst first start by reviewing a memory image in Redline, followed by a deeper dive allowed by the Volatility framework. Or perhaps using Redline to perform IOC detection while simultaneously starting to analyze memory with Volatility. After this section, our hope is that you will have the experience and knowledge to create a workflow that works best for you and your team.



Redline Versus Volatility

Depending on the type of response or hunting you are accomplishing, speed of analysis might be a big consideration. Volatility, without a GUI driving it, is slower to analyze data quickly because it relies on the typing skills and command-line kung-fu of the analyst driving the tool. Volatility also needs more discrete tuning and instrumentation to find key data via its robust plugins. Redline is initially easier to use and provides a great way to learn how to answer the important question in memory forensics, "What am I looking for?" Having to battle the keyboard and learn the tool simultaneously sometimes distracts from the main purpose of response.

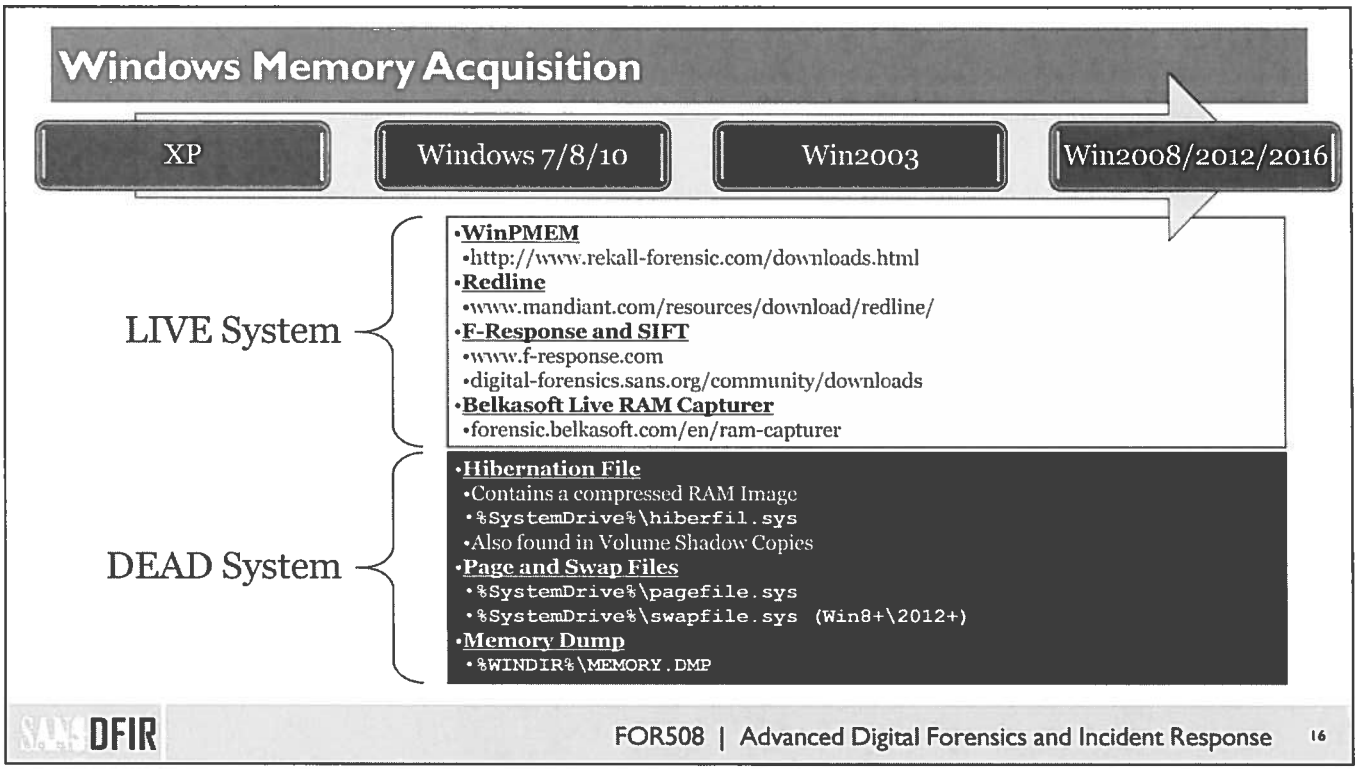
During quick assessment incident response, it helps to have a tool to give you a high-level view of what is going on in memory. An easy tool that accomplishes this is Redline. Redline has many advanced features as well that we will cover, but from a usability standpoint, it clearly enables responders to quickly assess a system rapidly without having to input a series of commands. In comparison, being able to view the process, process ID, parent process name, parent process ID, command-line invocation, process owner, directory/path, and network connections from a single process might take 3–4 Volatility plugins to accomplish, and then you have to tie the outputs all together. Using Volatility sometimes feels like a crime scene "items pinned" with "yarn connecting" that they over dramatize in movies and TV shows.



For hunters and those continually doing threat identification across multiple systems, **Volatility** might be the way to go. Because it is command-line driven, many of the functions and outputs of **Volatility** can be scripted to run across multiple systems. Volatility also has more of a discrete capability to find well-hidden data. This is the type of data that sometimes eludes Redline.

In a nutshell, Redline is great for quick assessments. But **Volatility** allows for finer control and more exact extraction of key data.

When we move from using Redline to **Volatility**, we will sometimes quote *Top Gun* by saying, "He is too close for missiles, switching to guns." Meaning that Redline is going to work here, we think we need to switch to Volatility to pinpoint key data we are after. Both tools are very powerful and in the hands of a trained responder, they will make finding malware and key data easier than ever before.



Windows Memory Acquisition

There are a number of different memory acquisition applications available, and they all operate similarly. Prior to Windows 2003 SP1, a handle named \Device\PhysicalMemory could be used to address and copy memory. Due to the security concerns of allowing access to memory from user-mode, this handle was deprecated and a driver must now be used to access memory through the Windows kernel.[1] The acquisition tool loads a system driver to gain raw access to memory and then dumps the entire contents of memory into a raw file. There are some things to be aware of with this approach. For those of you with malware hunting experience, you might notice that using a loaded driver for accessing raw memory is quite similar to the steps some malware takes. Hence, you might occasionally encounter issues with host protection applications like anti-virus and host intrusion prevention software (HIPS). In 64-bit Windows operating systems, all loaded device drivers must be digitally signed. The tools covered in this class have all taken the steps necessary to get their drivers signed, but note that some older tools might not operate on 64-bit systems.

The most important thing to know about memory acquisition is *regardless of the tool you choose to dump the memory image, any of the major memory analysis tools will be able to analyze it*. We will cover a couple of different acquisition options in this class. WinPMEM is one of the most exciting new memory dumping tools to recently emerge.[2] It supports both 32- and 64-bit systems and also includes an interesting option for "live memory analysis." In live mode, it will load the driver giving access to raw memory and then allow raw access to memory (read-only or even read-write!). Redline is often used as just a memory analysis tool, but we will see shortly that it also has the important ability to conduct something new to this field: live memory analysis. As part of this live analysis, a complete memory image can also be acquired.

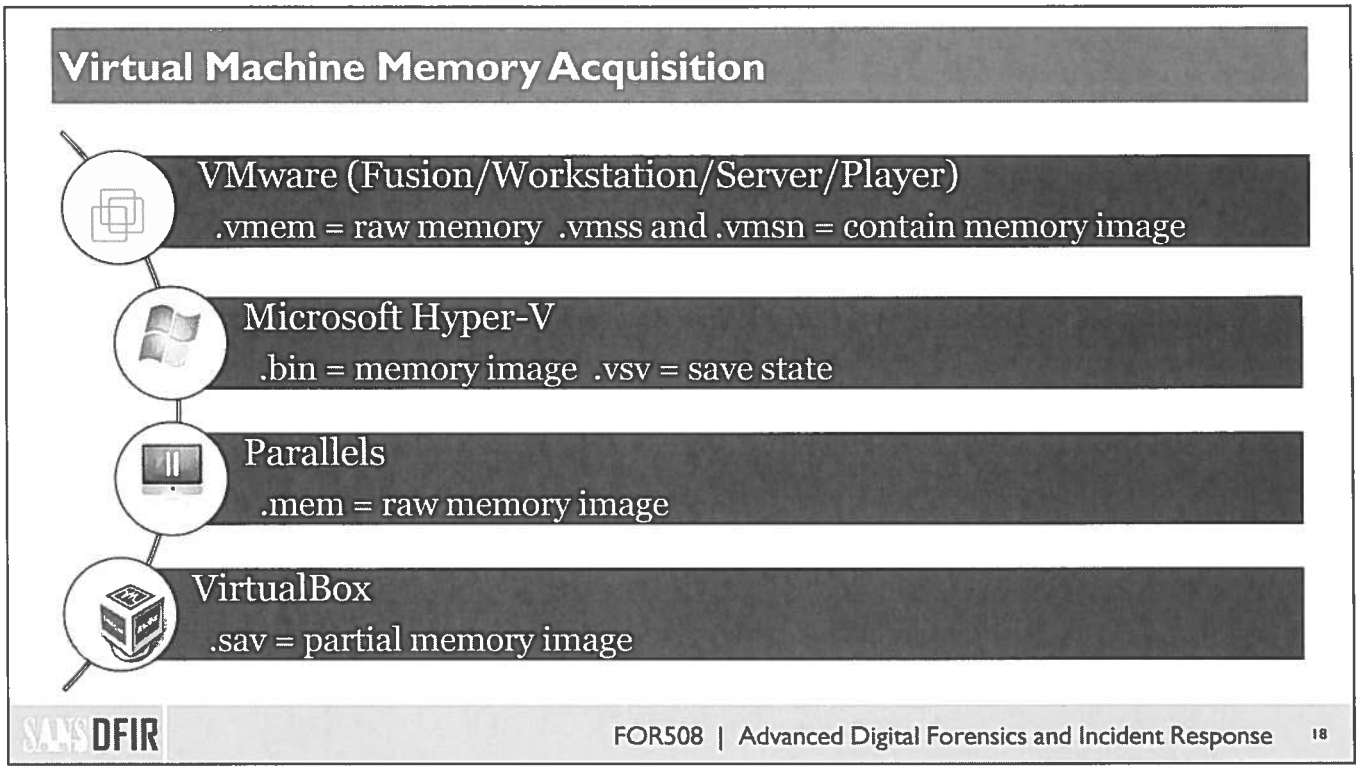
So far, all of the acquisition tools we have talked about require a system to be up and running. This is hardly surprising because RAM is largely considered volatile data that disappears upon system shutdown. Although that is indeed true, don't overlook some of the copies of RAM that are created automatically by various operating systems. As an example, many Windows systems—particularly laptops—maintain a hibernation file named "hiberfil.sys." This file is created when the laptop lid is closed and the system moves into power save, or

hibernation mode. It turns out that "hiberfil.sys" is a complete copy of everything in RAM when that lid was closed! Simply copying this file from the root of the system drive gives us a ready-made memory image ready for analysis. Crash dump files are also great sources for RAM analysis. Look for "memory.dmp" files in the %WINDIR% folder. If a full crash dump was taken, it will be a complete copy of RAM. Finally, the Windows "pagefile.sys" and "swapfile.sys" files are not a complete copy of RAM, but still contain parts of memory that were paged out to disk. The latter, "swapfile.sys," showed up in Windows 8 and Server 2012 and is used to hold the working set of memory for suspended Modern applications that have been swapped to disk.[3]

[1] http://technet.microsoft.com/en-us/library/cc787565.aspx

[2] http://scudette.blogspot.com/2012/11/the-pmem-memory-acquisition-suite.html

[3] http://blogs.technet.com/b/askperf/archive/2012/10/28/windows-8-windows-server-2012-the-new-swap-file.aspx



Virtual Machine Memory Acquisition
When responding to virtual machines, a useful technique is to suspend the virtual machine, forcing a copy of memory to be copied to the host system. In many popular virtualization products, such as VMware, Microsoft Server 2008 Hyper-V, and Parallels, this memory image is a raw copy of memory and can be directly analyzed using our memory analysis tools. However, different configurations or platforms might cause memory to be stored in a more complicated form that must be converted (for example, VMware ESX and Hyper-V).[1]

VirtualBox is one outlier. Its memory file holds only memory actively in use, not the entire amount of memory assigned to the virtual machine. Thus, it will not be recognized by many memory analysis tools (Volatility has an address space that will enable analysis on some images from VirtualBox, depending on the output format). There is also documented research on using the debug features of VirtualBox to force a memory dump.[2,3] The fallback plan for any virtual machine is to run a memory acquisition tool within the virtual guest. A raw memory image can be collected in this manner as long as the acquisition tool is compatible with the running operating system.

Also keep in mind that in some cases, each snapshot stored for a virtual machine will have its own complete copy of memory. VMware products are an excellent example of this with each snapshot having a separate **.vmem** file that can be analyzed. When working in virtualized enterprise environments, you might need to gather memory from VMware ESX servers. ESX servers utilize .vmss (VMware saved state), and .vmsn (VMware snapshot) files to store memory. Although these files are not raw dumps of memory, they do often contain a full memory image. The Volatility project has an address space that supports memory analysis for these files.[4]

Some common places to look for memory files for each product are:

Windows Server 2008 Hyper-V: `<Drive Letter>\XXX\<VM name>\Virtual Machines\GUID\`
VMware Workstation: `<Drive Letter>\XXX\My Virtual Machines\<VM name>\`
VMware Fusion: `/Users/<username>/Documents/VirtualMachines.localized/`
VMware ESX:
`<DatacenterName>\<DatastoreName>\<DirectoryName>\<VirtualMachineName>`
VirtualBox: `.VirtualBox/Machines/<VM name>/Snapshots/`
Parallels: `/Users/<username>/Documents/Parallels/<VM name>/Snapshots/`

Or, alternatively, search for files with these specific file extensions.

[1] http://www.wyattroersma.com/?p=77
[2] http://wiki.yobi.be/wiki/RAM_analysis
[3] http://arvandor2.blogspot.com/2015/02/virtualbox-to-volatility-short-script.html
[4] https://code.google.com/p/volatility/wiki/VMwareSnapshotFile

Case Study: Breaking TrueCrypt (1)

1. Dane County, Wisconsin, police signals jammed over 21 times throughout 2003

2. Rogue signals tracked to home of Rajid Mitra
   - Two previous hacking convictions + HAM operator

3. Mitra sentenced to 7 years in U.S. federal prison
   - TrueCrypt container found on system, but not cracked
   - He REALLY wanted his computer systems returned

Case Study: Breaking TrueCrypt

The Rajid Mitra case is interesting for a couple of reasons. It has the distinction of being the first prosecution of the U.S. Federal Computer Crimes Statute under the post-9/11 Patriot Act due to Mr. Mitra's interference with police and emergency responder communication channels. The Mitra case also provides a very interesting story of breaking into an encrypted file container over six years after it was seized by police. Milwaukee Police Detective Rick McQuowen and Madison Police Detective Cynthia Murphy worked together to defeat the encryption. There are several resources available if you would like to learn more about the case.[1], [2]

[1] http://caselaw.findlaw.com/us-7th-circuit/1031818.html

[2] http://www.theferalscribe.com/dispatches/the-miserable-life-and-sad-death-of-rajib-mitra.html

Case Study: Breaking TrueCrypt (2)

- Mitra sues investigating officer Cindy Murphy for return of his systems and data in 2005
- Murphy attends forensics conference in 2009 and learns about pulling TrueCrypt keys from memory
- Problem: Memory image was not acquired
- Solution: Windows hibernation file exported from image and uncompressed
- TrueCrypt file cracked using key found in hiberfil.sys
  - Contained child pornography
- Mitra sentenced to another 6.5 years in 2011

Case Study: Breaking TrueCrypt (2)

1. Dane County, Wisconsin, police signals jammed over 21 times throughout 2003.
2. Rogue signals tracked to home of Rajid Mitra.
   - Two previous hacking convictions + HAM operator.
3. Mitra sentenced to 7 years in US federal prison.
   - TrueCrypt container found on system but not cracked.
   - He REALLY wanted his computer systems returned.
4. Mitra sues investigating officer Cindy Murphy for return of his systems and data in 2005.
5. Murphy attends forensics conference in 2009 and learns about pulling TrueCrypt keys from memory from Detective Rick McQuowen.
6. Problem: Memory image was not acquired.
7. Solution: Windows hibernation file exported from image and uncompressed.
8. TrueCrypt file cracked using key found in hiberfil.sys.
   - Contained child pornography.
9. Mitra sentenced to another 6.5 years in 2011.

Hibernation File Analysis: `hiberfil.sys`

- Compressed copy of RAM at the time of hibernation
- Some tools can decompress to raw:
  - Volatility **`imagecopy`**
  - Comae **`hibr2bin.exe`**
- Many tools can analyze natively:
  - BulkExtractor
  - Internet Evidence Finder
  - Volatility
  - Belkasoft Evidence Center
  - Passware

Belkasoft
Forensics made easier

Passware

Hibernation File Analysis `hiberfil.sys`

Windows hibernation files are created whenever a system has been placed in hibernation, or "power save" mode. This most commonly occurs in laptop computers when the lid is closed on a running system. However, it can occur on any modern system running Windows and thus should be a routine check performed during an examination. The Windows hibernation file is named "hiberfil.sys" and is located in the root of the system drive (typically "C:\").

Take a step back, and think about the usefulness of a hibernation file. *Even if you respond to a system that has already been shutdown*, you have a chance to still review a memory image of the system. *If the system is up and running, you now might have two different memory images to analyze*: the one you acquire now, and the one derived from the hibernation file saved days, weeks, or even months before.

The Windows tool that enables, disables, and modifies the compression of hibernation files is powercfg.exe:

C:\>powercfg.exe /?

POWERCFG <command line options>
Description:
This command line tool enables users to control the power settings on a system.

Parameter List:
… SNIP …
HIBERNATE, -H
Enables-Disables the hibernate feature. Hibernate timeout is not supported on all systems.

Usage: POWERCFG -H <ON|OFF>

POWERCFG -H -Size <Percent Size>

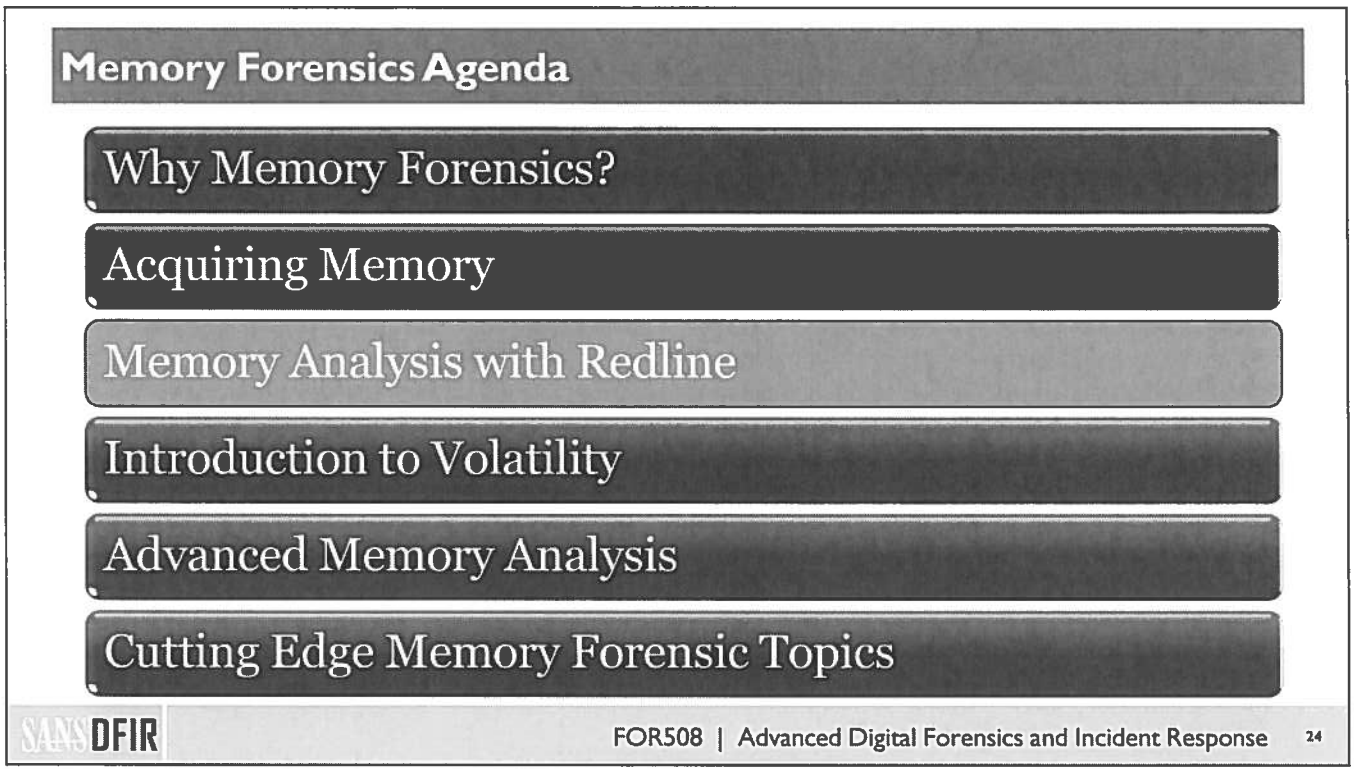
-Size: Specifies the desired hiberfile size in percentage of the total memory. The default size cannot be smaller than 50. This switch will also enable the hiberfile automatically.

As seen in the options for powercfg.exe, hibernation files are commonly compressed. They usually need to be de-compressed before they can be analyzed like a live RAM dump. The Volatility memory analysis framework has a means to read and convert Windows hibernation files. We will cover Volatility and its **`imagecopy`** plugin later in class.[1] Comae has a well-regarded commercial memory analysis suite that contains the **`hiber2bin`** tool to perform conversions.[2]

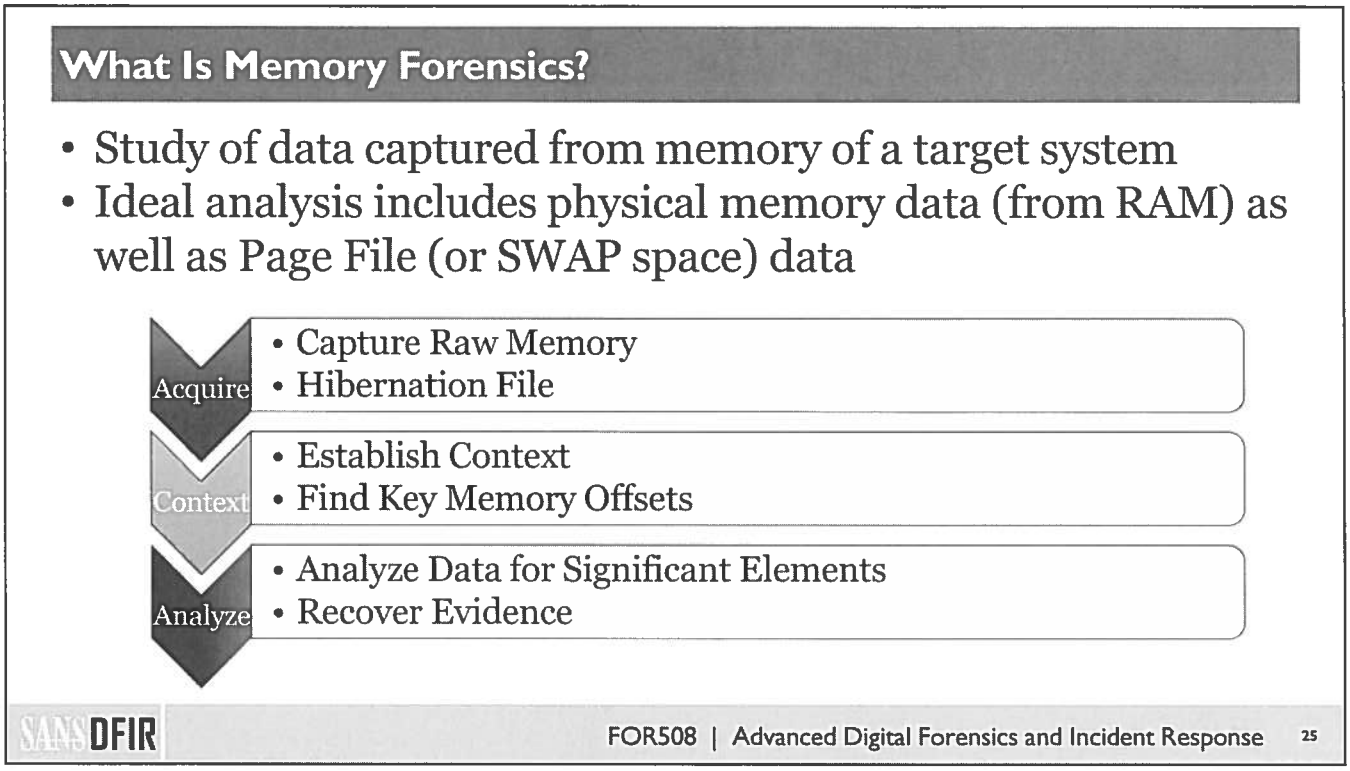
Several other forensic tools have added hibernation file analysis capabilities in the last few years. These tools will typically decompress **`hiberfil.sys`** on the fly and perform string searching and data carving. Examples include BulkExtractor, Magnet Forensics Internet Evidence Finder, Belkasoft Evidence Center, and Passware.

[1] http://code.google.com/p/volatility/wiki/CommandReference#imagecopy

[2] http://www.comae.io/



This page intentionally left blank.



What Is Memory Forensics?

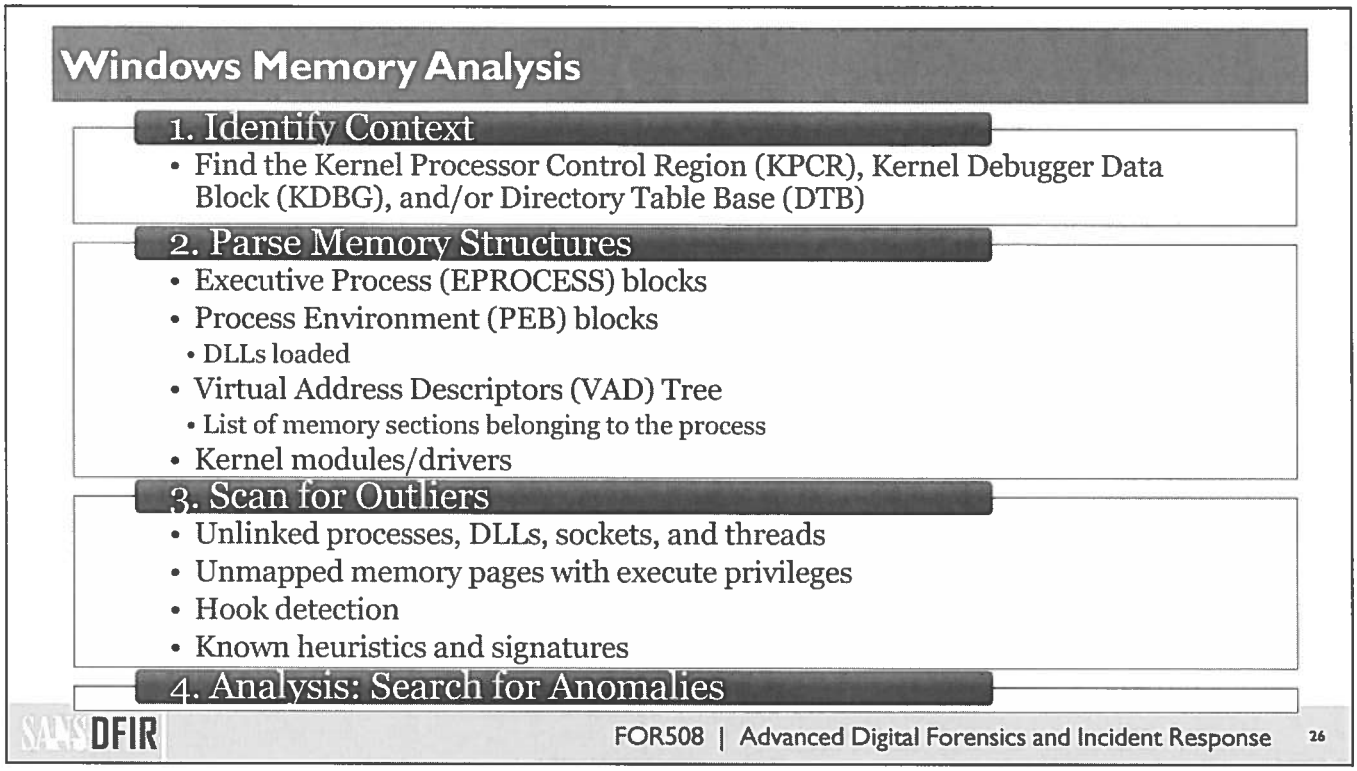
Memory forensics operates on data from the system's memory, both the RAM as well as the virtual memory storage (if available).

Memory analysis differs somewhat from traditional media forensics:

- **The data is much more of a snapshot in time:** Things might have changed dramatically from the instant before.
- **Establishing context is more complicated:** We are dealing with much more information than simple files and directories.
- **The data is not in a format designed to be extracted and understood, it is in a format designed to execute:** It requires more analysis and understanding of environment.

The memory analysis process can also be very similar to media forensics:

- **Collect the data for analysis.** This should be done in as forensically clean a fashion as possible. This is far more difficult a problem for memory forensics than for media forensics, because the system must be operating for the physical memory to be read. Unfortunately, the very act of reading the memory usually requires execution of a program, changing the data it is collecting even as it collects it.
- **Put the collected data into context.** For media analysis, this means understanding the disk, the partitions, the file system format, etc. For memory forensics, this is more complex, but just as necessary. You need to understand memory's ever-changing architectures and how (and where) various objects are stored.
- **Analyze your results to understand what the data means and identify important elements.**



Windows Memory Analysis

We have made our best efforts to not turn this into a class on Windows Internals. Although that information is certainly useful, it is covered in other classes (such as SANS FOR526 and FOR610), and there are great reference books on the topic for those interested, including one named oddly enough, *Windows Internals*.[1] That being said, in order to truly understand what is available in memory, the student needs to be familiar with some concepts.

First, how do our tools develop context within a memory image? How do you take a raw dump of memory and make any sense of it? The answer exists in something called the Kernel Debugger Datablock (KDBG). The KDBG is the key to many tools understanding a Windows memory image. It is a data structure whose pointers can be followed to eventually find the process list for the system.[2] The KDBG can be found through two different methods. One is to first find the Kernel Processor Control Region (KPCR), which has a pointer to the KDBG. In some Windows versions (like XP), the KPCR is maintained at a fixed offset within memory. Thus, any tool can point directly at that offset and follow the pointers to the rest of the objects we really care about. The fixed offset for the KPCR changed starting with Vista, but it can still be found with a little more work. Alternatively, the KDBG can be searched for using well-known signatures. You will notice that many memory analysis suites are very specific about what operating systems, architectures, and even service packs that they support. This is because unlike file systems, the memory layout is constantly changing and it is a battle for the tool creators to stay ahead of the curve. Finding the KPCR or KDBG takes time, so be patient while your memory analysis tool is processing! Rekall, a framework we will discuss later in the course, has moved away from KDBG scanning and instead uses very specific profiles to quickly find (not scan for) the PsActiveProcessHead, Directory Table Base (DTB), and other important memory objects.

Once the KDBG (or equivalent) is found, it leads to the EPROCESS, or executive process block list by identifying the PsActiveProcessHead pointer. This is a list of all the currently running processes in memory. Similar to a file in a file system, nearly everything revolves around processes in memory. Each process will have its own Process Environment Block (PEB) that holds a host of data structures that define a process, including the full path of the process executable, the command line that spawned the process, and a linked list of all loaded libraries (DLLs) for the process. Each executive process block points to a Virtual Address Descriptor (VAD) tree that is responsible for

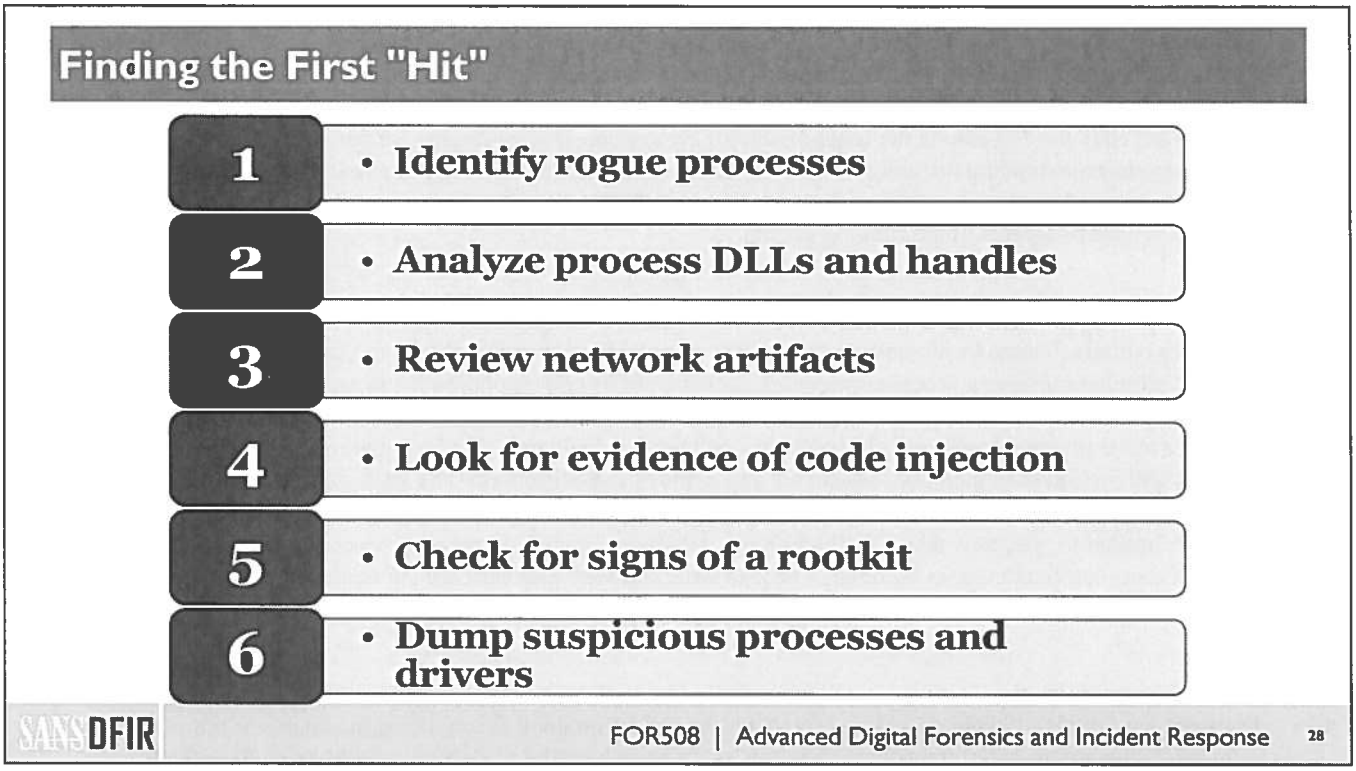
tracking every memory section (also called a memory page) assigned to that process. The VAD tree is of particular importance to our memory analysis tools because it allows them to double check what exists in the various memory sections for a process versus what the various lists say are present. If any discrepancies are found, that can be an important sign that something malicious like code injection has occurred. Kernel modules are code used to extend the functionality of the system. Device drivers are perhaps the most common of these modules, extending the ability of the system to communicate with new hardware. Our memory analysis tools need to identify where in memory these modules exist because they are frequently employed by malware to control aspects of the running system.

Once the memory analysis tool has identified all of the component parts in memory, the next step is to start to search for outliers. Windows memory structures are complex, but surprisingly easy to circumvent. For instance, simply unlinking a process or network socket from its corresponding list in memory will cause it to disappear from our standard incident response tools while it is still running and available behind the scenes. Thus, our more advanced memory analysis suites will not only rely upon the information in the various linked lists, but will instead scan memory looking for any items of a specific type. The same goes for standard kernel activities like hooking. You can think of hooking like a detour sign; instead of going to the memory location you were headed to, you now take a different route. Hooks are standard operating procedure within Windows, but all of those legitimate hooks by devices or anti-virus software may obscure the malicious hooks. Our memory analysis tools will attempt to find these outliers and present them for analysis.

The final stage of memory analysis is where we fit in. Our tools present the data from the various components discussed previously. It is up to us to review that information and look for anything suspicious or out of the ordinary. Although the task might seem daunting, the current batch of tools does a wonderful job of facilitating the analysis process.

[1] http://technet.microsoft.com/en-us/sysinternals/bb963901

[2] http://moyix.blogspot.com/2008/04/finding-kernel-global-variables-in.html



Finding the First "Hit"

Memory analysis takes practice, but it really is no longer a "dark art." Tools like Redline and Volatility make memory analysis much more feasible and less time consuming. However, more important than having tools is following the right process. Many parts of memory forensics lend themselves to identifying malware using the three traditional malware detection methods: signature, contradiction, and heuristic/behavioral. Detection by contradiction is especially fruitful given the variety of analysis methods memory forensics provides to the examiner.

Imagine you are handed a memory image and asked to review it for signs of an intrusion or malware. Where do you start? Your goal should be to find that first "hit"—something suspicious that can be analyzed further and used to help find other suspicious occurrences. Start with a review of processes because they are the most important objects in memory. Continue your analysis by scrutinizing all of the various helper objects assigned to each process. In this second and third phase, you will review items like loaded libraries (DLLs), files, registry keys, mutants, and network sockets. If you do not yet find anything out of the ordinary, consider searching for signs of code injection, looking to see whether malware might have taken over a legitimate process. Also pay attention for signs of rootkit techniques. Rootkits might be difficult to spot on a file system, but are often glaringly obvious when looked at through the lens of memory. Finally, extract any suspicious processes, drivers, and memory pages and continue your analysis outside of the memory image via malware reverse engineering techniques or simple anti-virus scans.

This process takes a layered approach. It is conceivable that you might miss a suspicious process or DLL among the hundreds present in the memory image. But by rigorously working down your checklist, you very well might find a different clue like an injected memory page or hooked driver that will lead you back to that evil process.

Introducing Redline

- GUI tool for "guided" memory analysis:
  - Processes
  - Handles
  - Network Connections
  - Memory sections
  - Hooks and drivers

- x86 and x64 support for:
  - Win2000 | WinXP | Win2003 | Vista | Win2008 | Win7 | Win2008R2 | Win8 | Win8.1 | Win2012
- Built-in heuristics for suspicious processes and code
- Live memory analysis and live response capability
- Indicator of Compromise (IOC) matching
- File whitelisting

Introducing Redline

We are incredibly lucky that others have taken it upon themselves to create user-friendly tools to conduct memory analysis. Memory is much more complicated than a file system. Without these tools, we would still be in the proverbial stone age with respect to memory forensics.

Redline is probably the most user friendly of all memory analysis software. It is a second generation front-end for one of the first free memory analysis tools, Memoryze. It has the best Windows operating system support of any free memory analysis tool, with support for both 32- and 64-bit memory images. It is extremely well suited to assist us through our memory analysis process. In fact, this is how the Mandiant website describes Redline[1]:

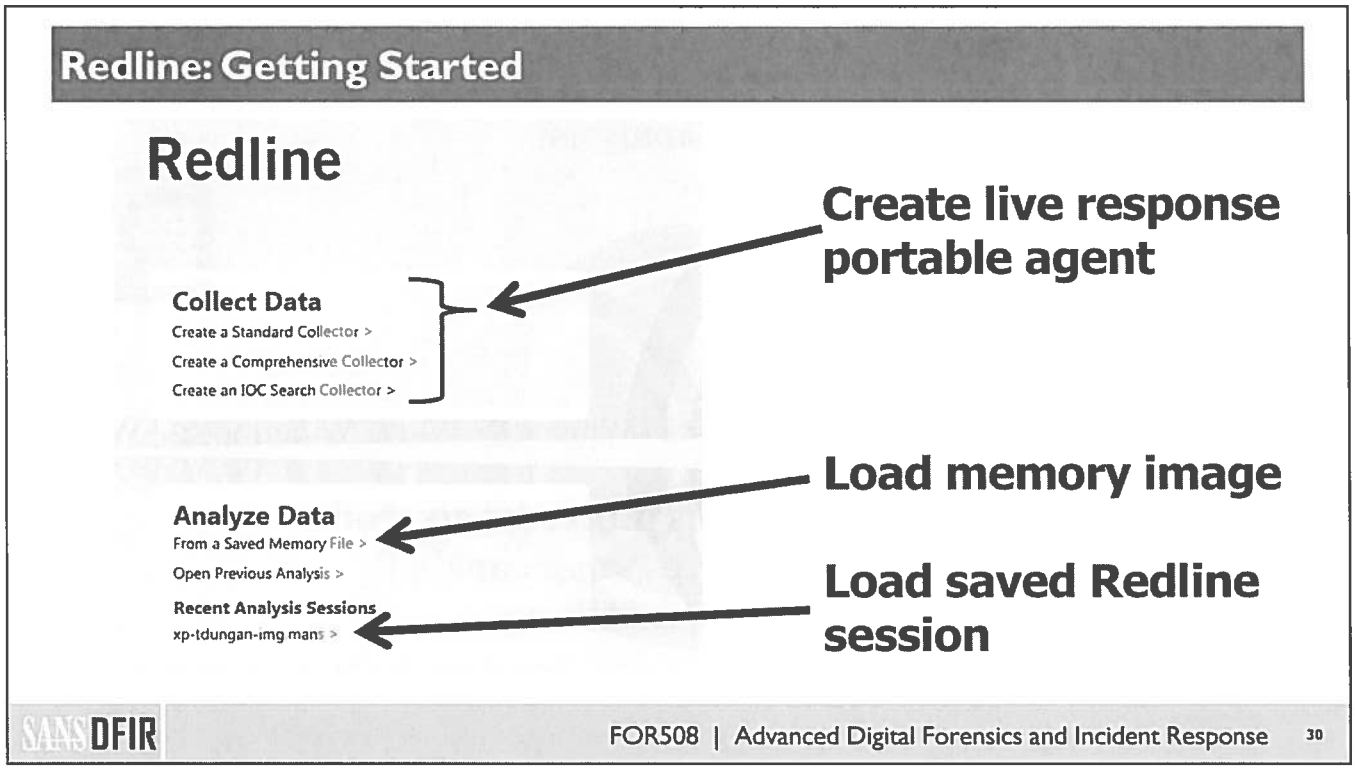
Redline guides investigators through the process of evaluating a system for compromise or infection. Starting from a small (<50MB) memory audit captured locally, Redline provides a series of investigative steps that steer beginning users towards the information most likely to help them find malware.

- Review processes with high Malware Rating Index scores.
- Review network connections.
- Review memory sections.
- Review untrusted handles.
- Review hooks.
- Review drivers.

Redline officially supports images from the following operating systems:

l Windows 8 (32-bit and 64-bit versions) l Windows 7 (32-bit and 64-bit versions) l Microsoft Vista (32-bit version) l Windows XP SP2 (32-bit version) l Windows Server 2008 R2 (64-bit version) l Windows Server 2003 SP2 (32-bit and 64-bit versions) l Windows Server 2000 SP4 (32-bit version)

[1] https://www.mandiant.com/resources/download/redline



Redline: Getting Started

Upon loading Redline, you will be provided with several options.

<u>**Collect Data**</u>

Create a Standard Collector

A "collector" is a portable agent created to collect the data necessary to perform an assessment of a live system. It is implemented as a batch script and designed to be run from removable media, such as a USB thumbdrive. The results are written back to the removable media and can then be imported into Redline via the **From a Collector** option under the **Analyze Data** options. A standard collector collects the data necessary to conduct a full live memory analysis of the target system.

Create a Comprehensive Collector

A comprehensive collector collects data necessary for memory analysis in addition to host-based artifacts such as file metadata, Windows Registry hives, Event Logs, and Prefetch files. This complete data set can be then used for searching using indicator of compromise (IOC) files. Note that this type of collection will take much more time than the other options.

Create an IOC Search Collector

Collect the minimal set of memory and host-based artifacts to perform a scan for the indicators of compromise (IOCs) specified. Full memory analysis cannot be conducted with this minimal set.

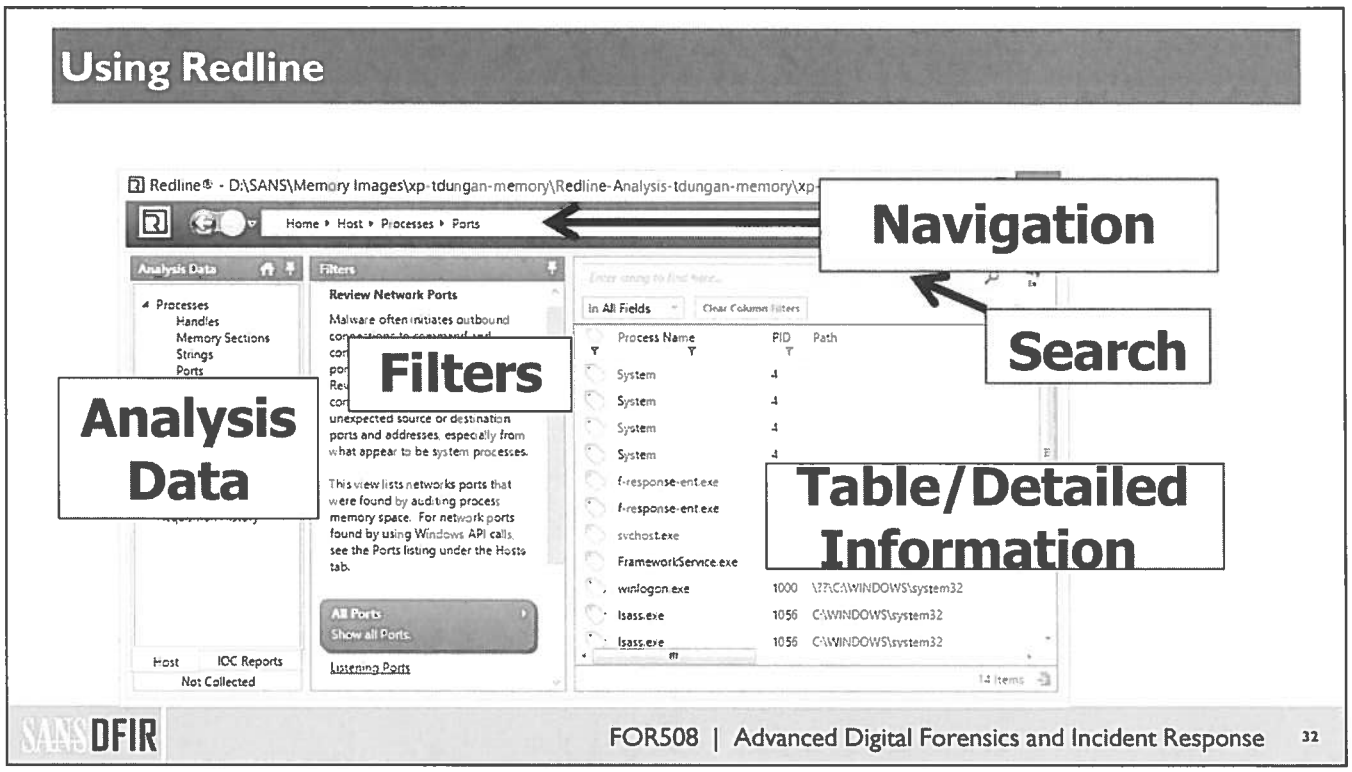
<u>Analyze Data</u>

From a Saved Memory File

This is the option that should have the big blinking lights around it. It allows you to open a raw memory image file for analysis.

Open Previous Analysis

Redline does all of its memory analysis processing upfront. This can take a long time. Thus, instead of having to re-process a memory file every time you want to return to it, you can save the results of your analysis and return back to where you previously left off. You might also see your recent analysis sessions listed in this area. Additionally, the collector output now includes a .mans saved session file, which should be opened using this option.



Redline was built to provide expert-level capabilities to analysts of any ability level. With this in mind, Redline will attempt to guide the analyst to a specific view based on analyst goals. In many cases, clicking through the **I am Reviewing a Full Live Response or Memory Image** is a good place to start. This will drop you into the process overview table, allowing analysis to begin. Alternatively, you can just select **Processes** from the left pane.

In each view, Redline will attempt to quickly point out areas that might be more likely to be malicious. It identifies potentially malicious objects via a combination of behavioral analysis, known bad heuristics, and concepts like "Least Frequency of Occurrence." We will discuss each of these features in depth. For now, it is most important to understand that in many cases only a subset of data that Redline thinks are most suspicious will be initially available. This can be helpful when starting out, or when taking a quick look at a memory image. However, if a tool was perfect and could automate the entire process, what is the analyst for? It turns out that like any tool, Redline sometimes gets it right and sometimes misses completely. Thus, to perform a comprehensive analysis, you are going to need to go much deeper than the standard hits provided by Redline's guided analysis. Hence, pay attention to indicators that describe what subset of data is being displayed. In many cases, you will want to select the **All** category to make sure nothing is missed.

Using Redline

One of Redline's hallmarks is its graphical user interface. Very commonly, we will see a brilliant tool marred by a clunky or unusable interface. In Redline, they do a great job of presenting information in an easy-to-use format. The interface has several panes:

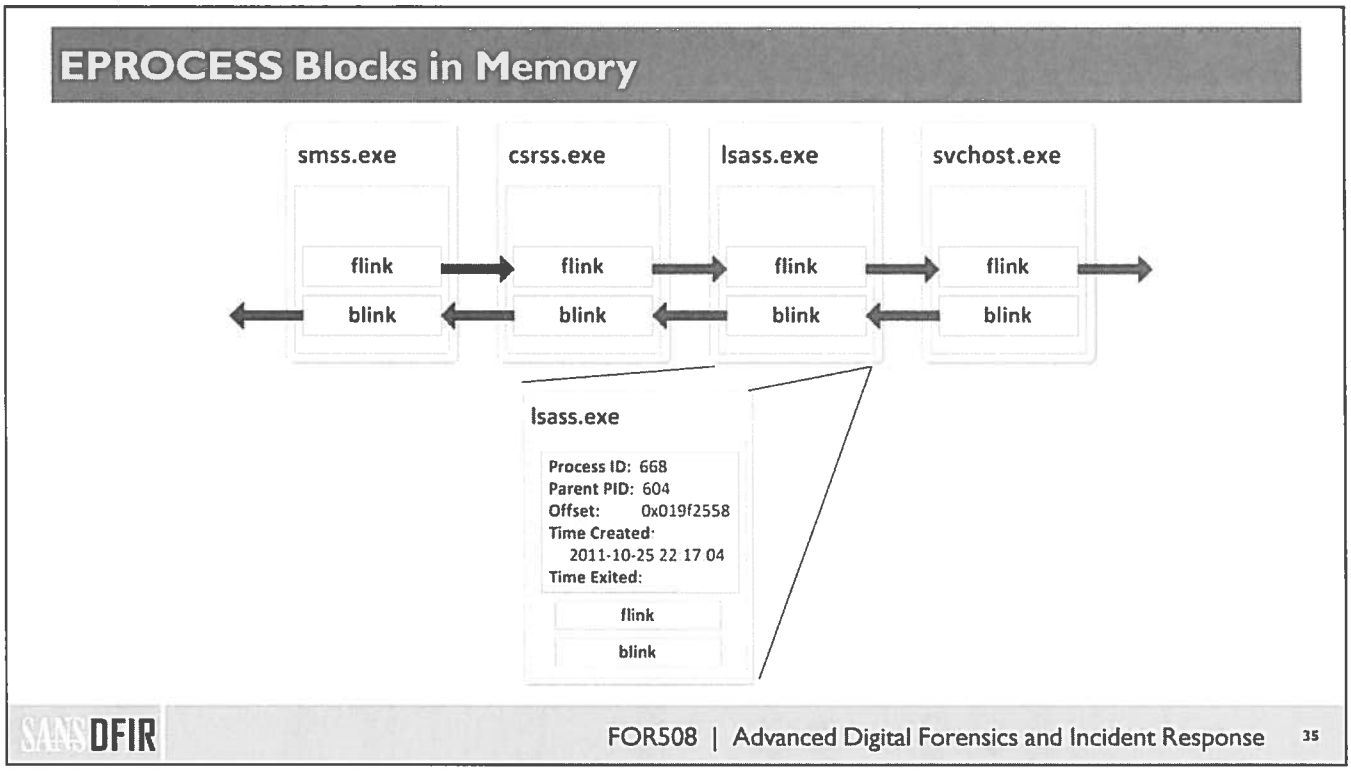
- **Analysis data:** This panel presents a tree structure representing all of the data identified and/or collected by Redline. This is very similar to the "host" view in previous versions of Redline. Anything selected here will display in the Table view pane.

- **Table view pane:** Tables are used extensively in Redline to present large quantities of data. This is the area where you will spend most of your review time. For instance, when **Processes** is selected in the Analysis Data panel, a listing of every process with key information is presented in the table view. If you would like more detailed information on a specific table row, double-click to view details. A right-click on any element provides abilities like copy, tag, and Google lookup.
- **Full detailed information:** When a table row is double-clicked, a new pane replaces the table providing details for that item. As an example, the Full Detailed Information pane for a process has seven tabs: Process, MRI Report, Sections, Handles, Ports, Strings, and Tags and Comments. Everything related to a process can be found here. An alternative to this pane is to use the **Show Details** link at the bottom of the Table View. This alternative is recommended only when you have a lot of screen real estate.
- **Filters:** In each view, Redline will attempt to quickly point out areas that might be more likely to be malicious. It identifies potentially malicious objects via a combination of behavioral analysis, known bad heuristics, and concepts like "Least Frequency of Occurrence." We will discuss each of these features in depth. For now, it is most important to understand that in many cases only a subset of data that Redline thinks are most suspicious will be initially available. This can be helpful when starting out, or when taking a quick look at a memory image. However, if a tool were perfect and could automate the entire process, what is the analyst for? It turns out that like any tool, Redline sometimes gets it right and sometimes misses completely. Thus, to perform a comprehensive analysis, you are going to need to go much deeper than the standard hits provided by Redline's guided analysis. Hence, pay attention to the Filters pane to understand what subset of data is being displayed. In many cases, you will want to select the **All** category to make sure nothing is missed. Both this and the Analysis Data panel can be collapsed to provide more room for the Table View pane.
- **Search:** Nearly every view has the ability to be searched. Hits can be highlighted and navigated using **Next** and **Prev** or applied as a filter, only showing lines with certain characteristics. Simple, regular expressions can also be used for more granular searching.
- **Navigation:** Keep an eye on the top of the tool to keep track of what view you are currently in. Click the bread-crumb trail to go back to a previous view, or use the forward and back arrows.
- **IOC reports:** If a list of Indicators of Compromise (IOC) were included during the initial setup, any hits will be reported using this tab. When using a good set of IOCs, this can be an excellent place to start because much of your analysis work might already be done for you! More on IOCs later in the class...

Step 1:

Identifying Rogue Processes

This page intentionally left blank.

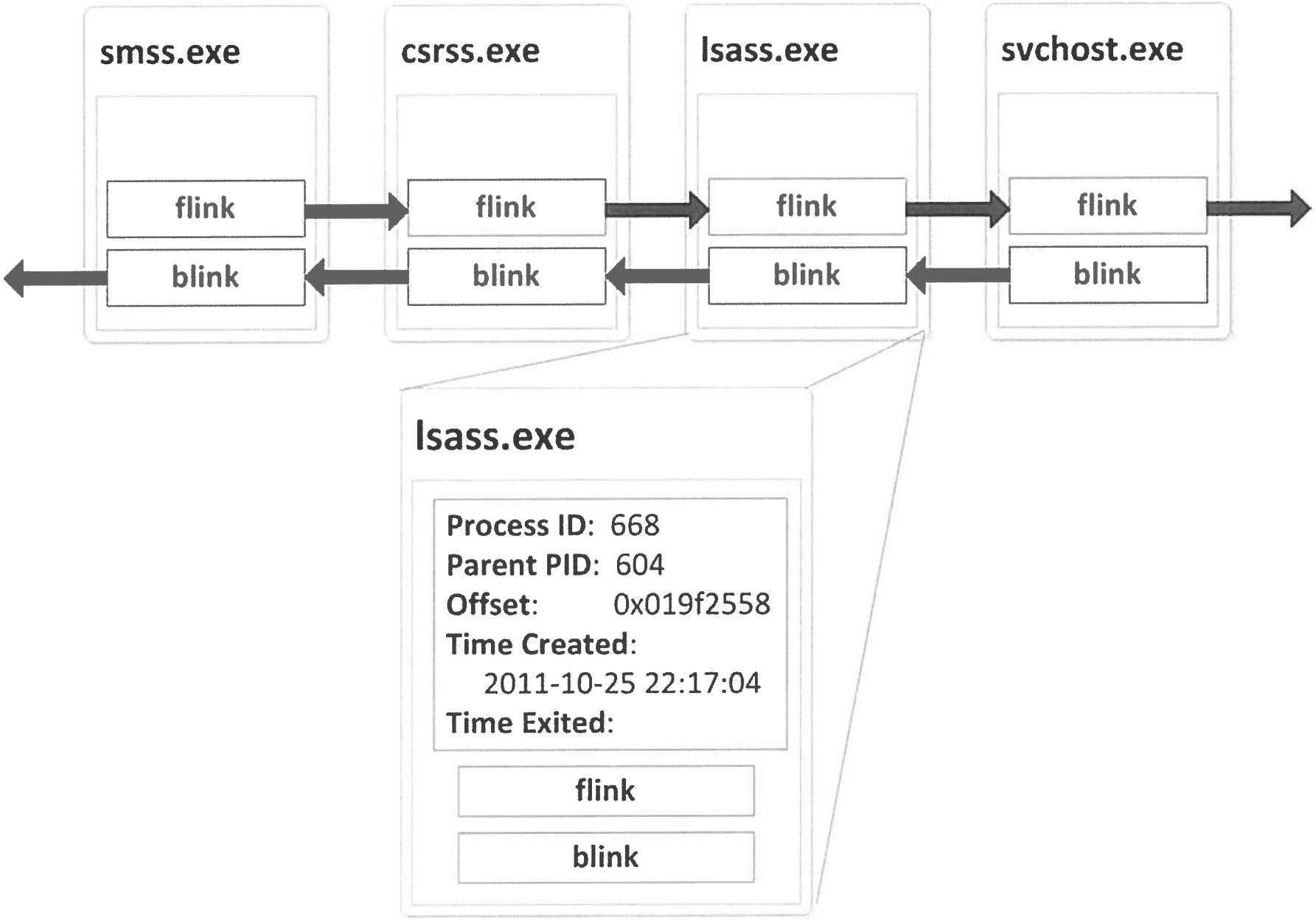


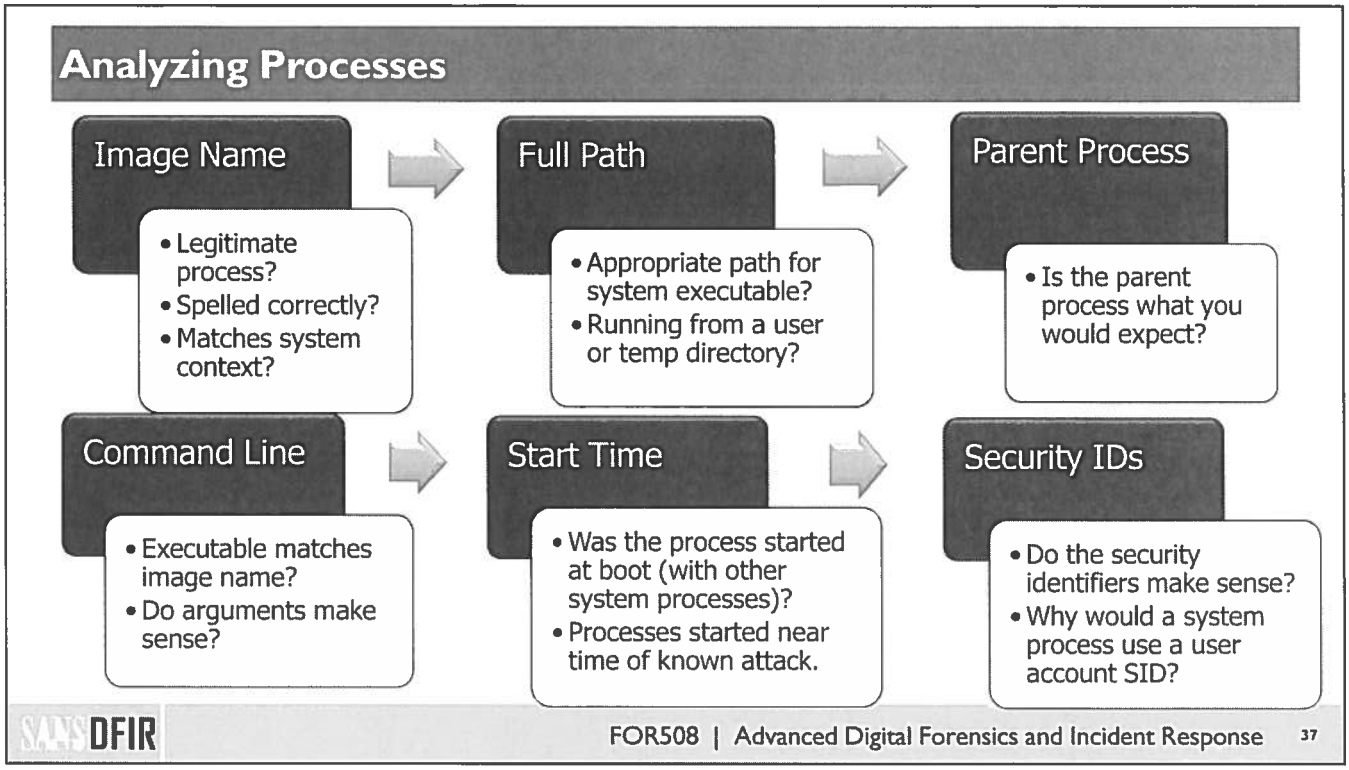
EPROCESS Blocks in Memory

Processes are a logical place to start because they are one of the most important building blocks within memory. Processes tell us what was running on the acquired system. Some of the most critical information we can get in malware investigations is going to be via process blocks. For those familiar with traditional disk-based forensics, think of processes like files in a file system. Some are allocated (currently in use), and some are unallocated and waiting to be overwritten. Process information is tracked by the operating system using an Executive Process Block, or EPROCESS for short. The EPROCESS block holds a majority of the metadata for a process. It gives us:

- Name of process executable (image name)
- Process Identifier (PID)
- Parent PID
- Location in memory (offset)
- Creation time
- Termination (exit) time
- Threads assigned to the process
- Handles to other operating system artifacts
- Link to the Virtual Address Descriptor tree
- Link to the Process Environment Block

With multiple processes running simultaneously, the kernel uses a doubly linked list, like the one seen in the slide, to track processes. Notice the flink (forward link) and blink (back link) fields in each EPROCESS block. This doubly linked list is very important to understanding processes. Only "allocated," or currently running processes, will be found in the list; when a process exits, it is unlinked. Similarly, malware with access to the Windows kernel can also unlink itself using a rootkit technique called Direct Kernel Object Manipulation, which we will see in a later slide. The takeaway is that our memory analysis tools can't just read this doubly linked EPROCESS list from the kernel and be done. To do a thorough job of identifying every process, the tools will need to scan the entire memory image for any orphan process blocks. This is done automatically in Redline, but we will see later that Volatility will have a special plugin that must be used to scan for processes.


smss.exe
csrss.exe
lsass.exe
svchost.exe
flink
flink
flink
flink
blink
blink
blink
blink
lsass.exe
Process ID: 668
Parent PID: 604
Offset: 0x019f2558
Time Created:
2011-10-25 22:17:04
Time Exited:
flink
blink



Analyzing Processes

Memory analysis tools like Redline will easily get us a list of all processes and their associated metadata. The harder part is knowing how to analyze the information and identify anomalies. When reviewing processes in a memory image, concentrate on these six items:

- **Image name:** The name of the process executable itself might give us useful information. Seasoned incident responders can often spot process names that don't belong on a standard Windows system. Attackers can name their processes to blend in, or hide in plain sight. This can be a successful technique owing to how complicated Windows systems and how many processes might be running. Look for subtle misspellings or legitimate process names that shouldn't necessarily be running on the system you are analyzing. For instance, Microsoft Security Essentials is named "msseces.exe." Seeing that process on a workstation might not cause much interest, but on a server that process might be much more rare.
- **Full path:** Malware will sometimes camouflage itself with legitimate image names that are run from different locations. Explorer.exe should always run from the Windows directory and iexplore.exe from the Program Files directory. If you see a process with these names running from the \Windows\System32 directory, you should take notice. Malware will sometimes be spawned from a Temp directory or even an odd directory like the Recycle Bin. Because very few legitimate processes run from these locations, this should be an obvious clue to look further into that process.
- **Parent process:** Knowing what process spawned another can help us in our search. Many user applications have a parent process of Explorer.exe, which is the user shell. If you see a process named like a system process, say "svchost.exe," running with Explorer.exe as its parent, that is suspicious. Similarly, knowing the parent process tells you where in the system boot hierarchy something was loaded. A process parented by Explorer.exe was likely spawned after logon, whereas one parented by Services.exe could have been spawned at boot and hence have a persistence mechanism.

- **Command line:** The Process Environment Block records the full command line used to start a process. This allows us to check that the actual executable matches its image name and identify strange arguments that might have been used.
- **Start time:** The start time is underutilized, but can be a great source of information. If you have evidence of an attack occurring near a certain time, look for processes started after that time. Do you see six svchost.exe processes, and all but one started within seconds of one another? It might be worth looking at the outlier. Once you identify a suspicious process, you can look more closely at other processes started around that time. Finally, you can usually determine whether a process was started at boot time or later by comparing the timestamp with the timestamps of known system processes like smss.exe or winlogon.exe.
- **Security identifiers:** The security identifier can tell us what level of account spawned a process. Seeing user SIDs are most obvious (they are much longer than standard system account SIDs), but you can also use SIDs to determine exactly what account was used to start a process. Was the process started by System when it should have been started by LocalService?

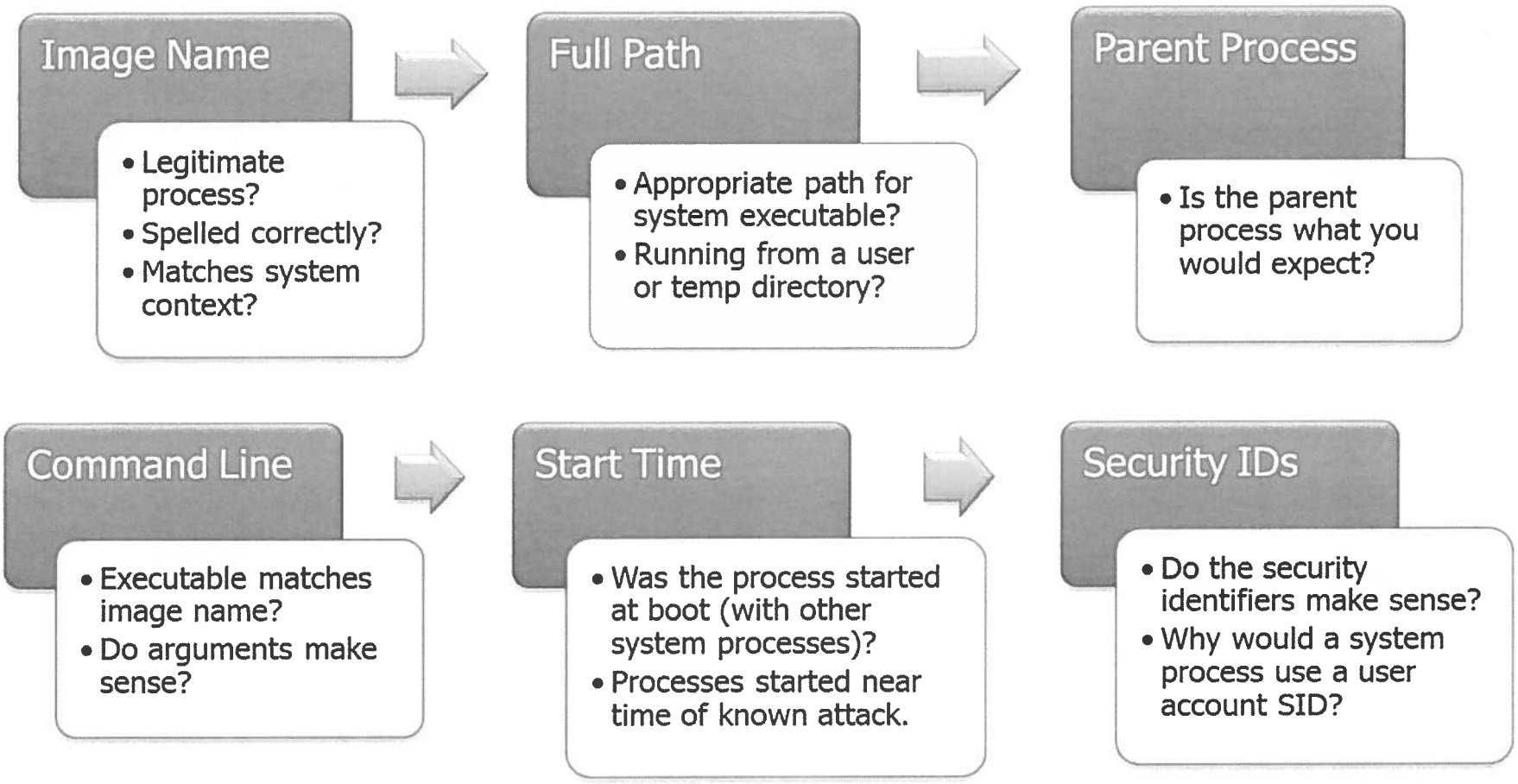
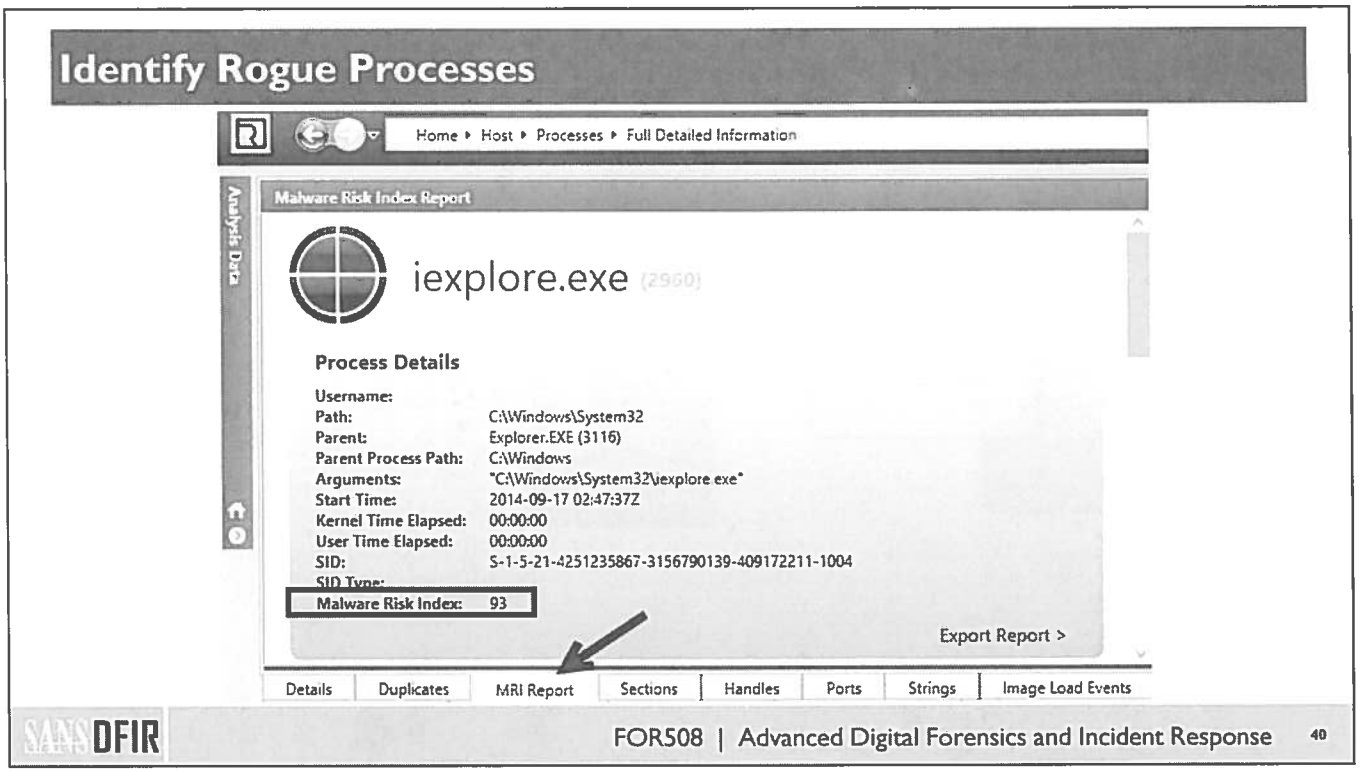


Image Name
• Legitimate process?
• Spelled correctly?
• Matches system context?
Full Path
• Appropriate path for system executable?
• Running from a user or temp directory?
Parent Process
• Is the parent process what you would expect?
Command Line
• Executable matches image name?
• Do arguments make sense?
Start Time
• Was the process started at boot (with other system processes)?
• Processes started near time of known attack.
Security IDs
• Do the security identifiers make sense?
• Why would a system process use a user account SID?

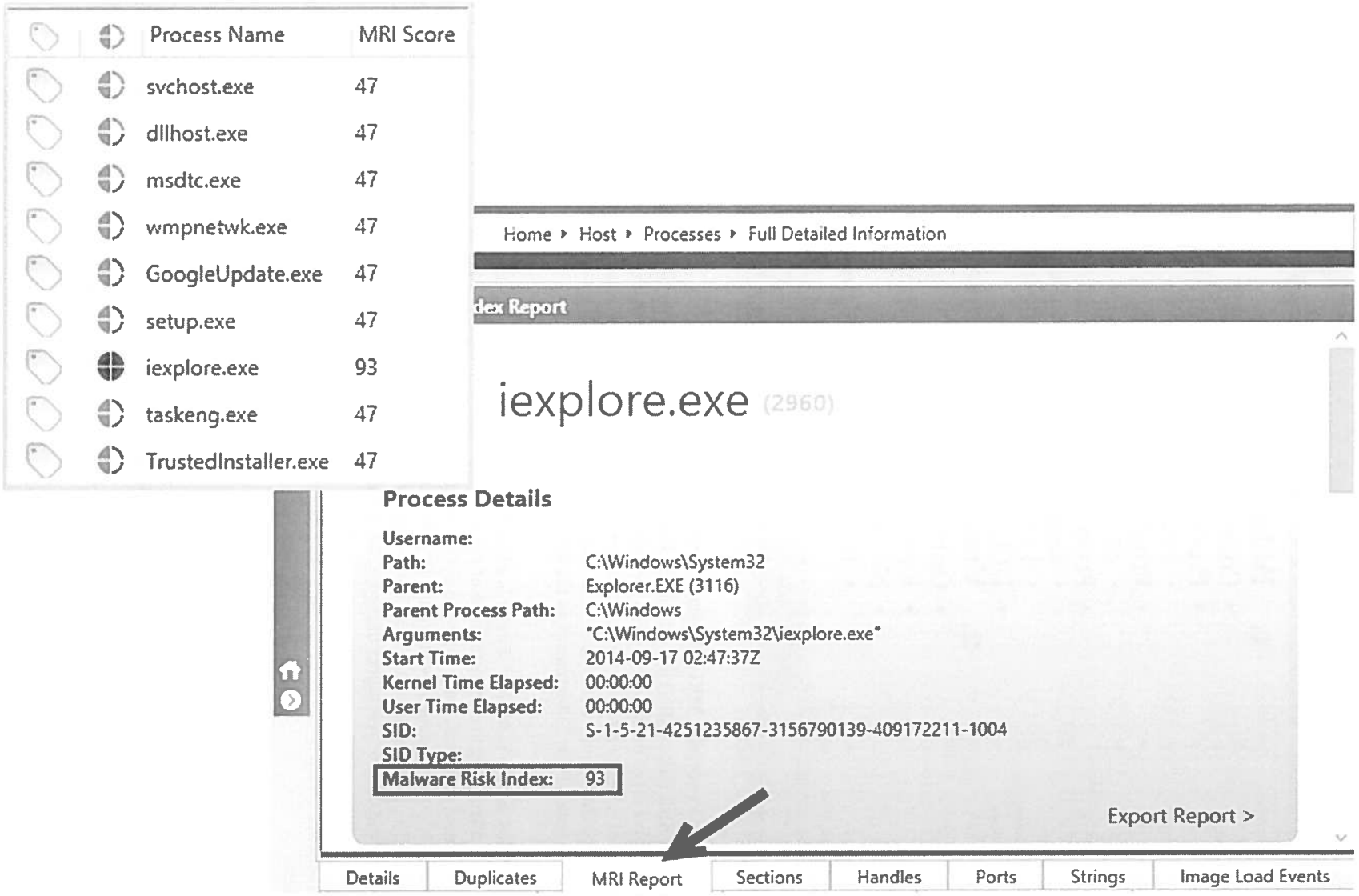


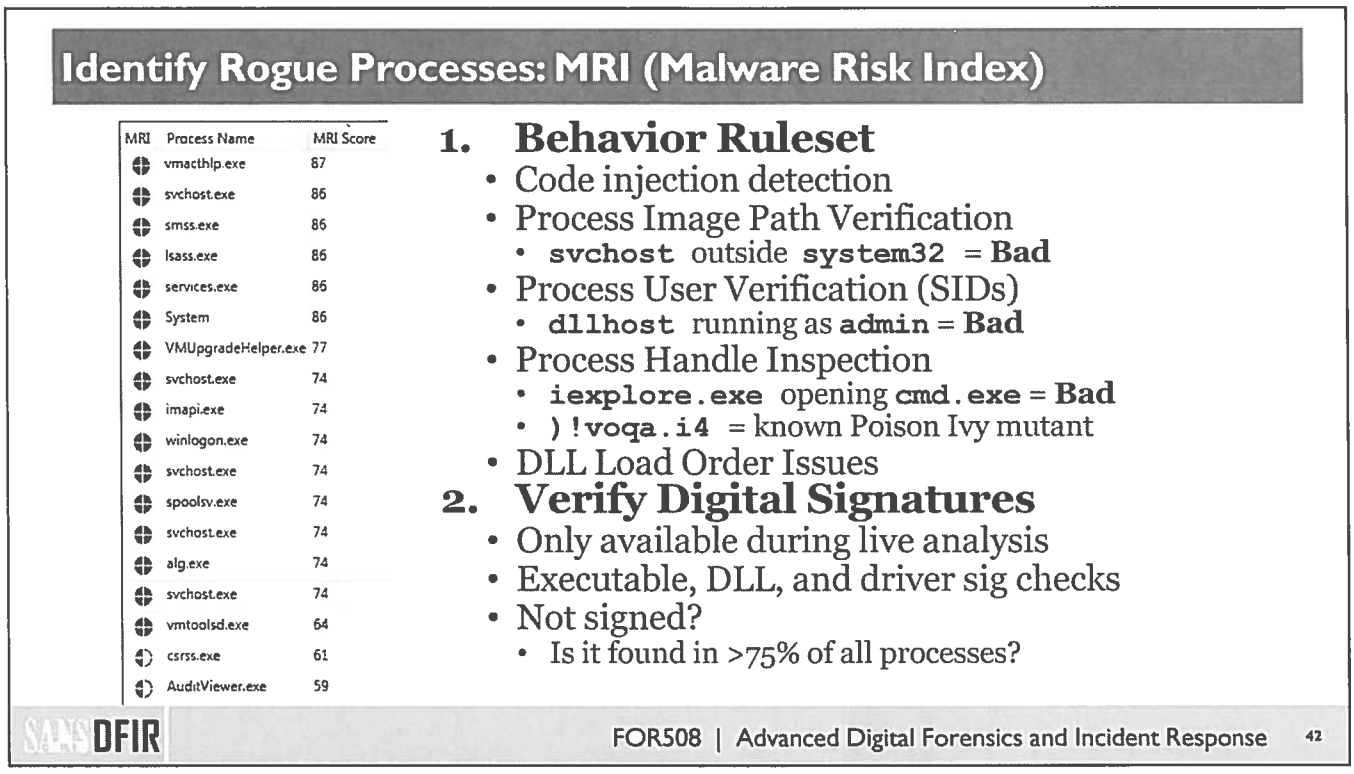
Identify Rogue Processes

Our first example in identifying rogue processes is a process named "iexplore.exe." You might recognize the executable name as being related to Internet Explorer. Because Internet Explorer is often running in Windows, it might be easy to overlook this process. However, in this case, Redline makes it very clear that it believes iexplore.exe to be malicious. It puts it at the top of the list and Redlines it, denoted by the red marker to the left of the process. Most other memory analysis tools would not have made it so obvious. Going through our process, how could we have determined that this was a bad process?

- The parent process is explorer.exe. In this case, that would be a legitimate parent for Internet Explorer.
- The process was running from "C:\Windows\System32." This could be legitimate because most system binaries run from this folder. However, this turns out to be a bit of camouflage as IE (and many other applications that run on top of Windows) always runs from the C:\Program Files folder.
- The Security Identifier (SID) for this process appears to be a user SID. In this case, it makes sense for a user account to execute Internet Explorer.
- The Redline Malware Risk Index is 93/100 (higher values indicate more suspicion). We will see why Redline believes the process to be bad in a moment.

As an aside, you might often notice a Redlined process in a memory image that pertains to a memory dumping tool (winpmem, Memoryze, etc.). Memory acquisition tools often look evil because they load drivers, access raw memory, and are often spawned from command prompts. Why doesn't Redline just whitelist these processes? Because a clever attacker could rename their malware to something with the same name in order to trick the tool (or the analyst).


Process Name
MRI Score
svchost.exe 47
dllhost.exe 47
msdtc.exe 47
wmpnetwk.exe 47
GoogleUpdate.exe 47
setup.exe 47
iexplore.exe 93
taskeng.exe 47
TrustedInstaller.exe 47
Home ▸ Host ▸ Processes ▸ Full Detailed Information
dex Report
iexplore.exe (2960)
Process Details
Username:
Path: C:\Windows\System32
Parent: Explorer.EXE (3116)
Parent Process Path: C:\Windows
Arguments: "C:\Windows\System32\iexplore.exe"
Start Time: 2014-09-17 02:47:37Z
Kernel Time Elapsed: 00:00:00
User Time Elapsed: 00:00:00
SID: S-1-5-21-4251235867-3156790139-409172211-1004
SID Type:
Malware Risk Index: 93
Export Report >
Details
Duplicates
MRI Report
Sections
Handles
Ports
Strings
Image Load Events



Identify Rogue Processes: MRI (Malware Risk Index)

The Malware Risk Index (MRI) is a prominent feature in Redline, and a fantastic innovation. The idea behind MRI is that if we have all of these heuristics to identify a bad process, why not create a means to automatically check them instead of relying upon the analyst to remember them all? You can think of this feature as a first pass on the memory image from a junior analyst. It won't catch everything, but it can catch the most obvious anomalies. In the words of co-creator Peter Silberman from the Mandiant blog[1]:

The goal of this feature is twofold. First it is going to help pinpoint specific processes that should be investigated further while attempting to eliminate some of the non-suspicious processes and get them out of the analyst's way. It's also designed to try and make malware detection easier. A lot of work went into looking at samples and how they behave etc., and coming up with definable behaviors that trap those little creatures. MRI is made up of two components. The first component is a definable behavior rule set that is completely customizable. Each process is given a score out of 100. The higher the score, the more likely that it is evil.

How is the score created? The score is created by two components. 1. Behavior rule set. 2. Verification of digital signatures.

The behavior ruleset is made up of **four** different types of rules:

1. **Process Path Verification**: Allows users to define what processes should be launched from what directories. This triggers on malware that copies and names itself after svchost or other system processes to subdirectories within system folders. For example, a default rule is that svchost can only be executed from \windows\system32. Any time we see it running from somewhere else, we flag the process.

2. **Process User Verification**: Allows users to define what processes should be running under what users. This triggers on malware spawning svchost for purposes of unmapping image bases or hiding

DLLs within spawned svchost. So, for example, if malware copies itself to system32\dllcache and then names itself svchost.exe, you can define a rule saying svchost.exe should be running as local service, network service, or system. When Audit Viewer sees svchost running as administrator, it gets flagged.

3. **Process Handle Inspection**: This allows you to define specific rules pertaining to malware or generic behavior. For example, a default rule is to flag svchost or iexplore anytime it has a process handle to cmd.exe. There is just no good reason for this to *ever* happen. You can also define rules based on specific malware, for example if [the] "a3c" mutant is present, then flag the process as being infected with [the virus] "sality."
4. **DLL Load Order Issues**: When enabled, evidence of DLL hijacking causes an MRI hit.

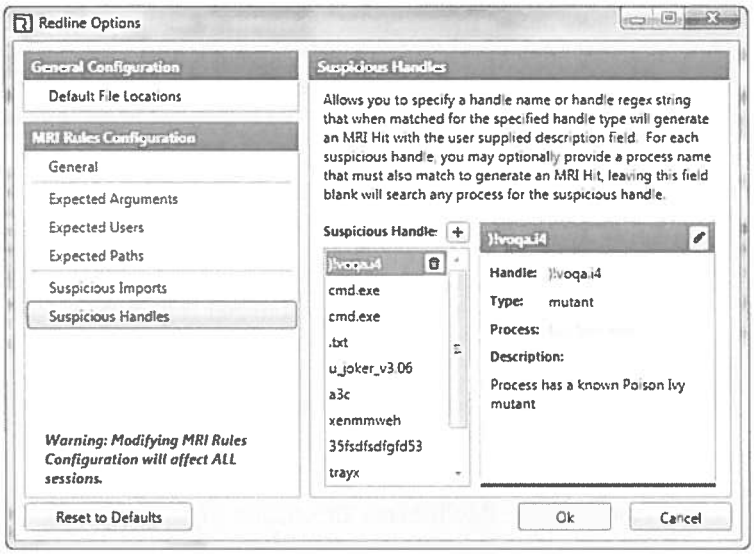
The second component of MRI is a process address space scoring mechanism. The new release will contain bug fixes as well as a new feature called "Verify Digital Signatures." When this parameter is turned on, [Redline] will perform a digital signature check on all loaded modules. This can only be enabled on live memory analysis. The digital signature check verifies the module on disk is digitally signed. We do a bunch of math and use our Least Frequency of Occurrence to trust modules that aren't signed but occur in more than X% of processes.

In addition to all of the checks discussed in the Mandiant blog, Redline now also checks for the following:

- Unmapped processes (evidence of code injection)
- Processes started by command shell
- DLL load order/hijacking detection
- Expected command-line arguments

In addition to all of the behavioral checks, the MRI will also take into account built-in signatures. However, Redline ships with a very small signature base. Unlike an anti-virus product with hundreds of thousands of signatures, Redline has less than fifty. This might change in the future, but more importantly, Redline gives you the capability to create your own alerting heuristics. So every time you find malware in your environment, you can add a signature to make it easier for you or someone else in your organization to find the same malware during the next examination. This allows you to customize the tool to be more effective in your day-to-day work. The signature creation tool is GUI based and very simple to operate. All that being said, we suggest using the OpenIOC format instead (covered later in this book) because it provides more flexibility and sharing among analysts.

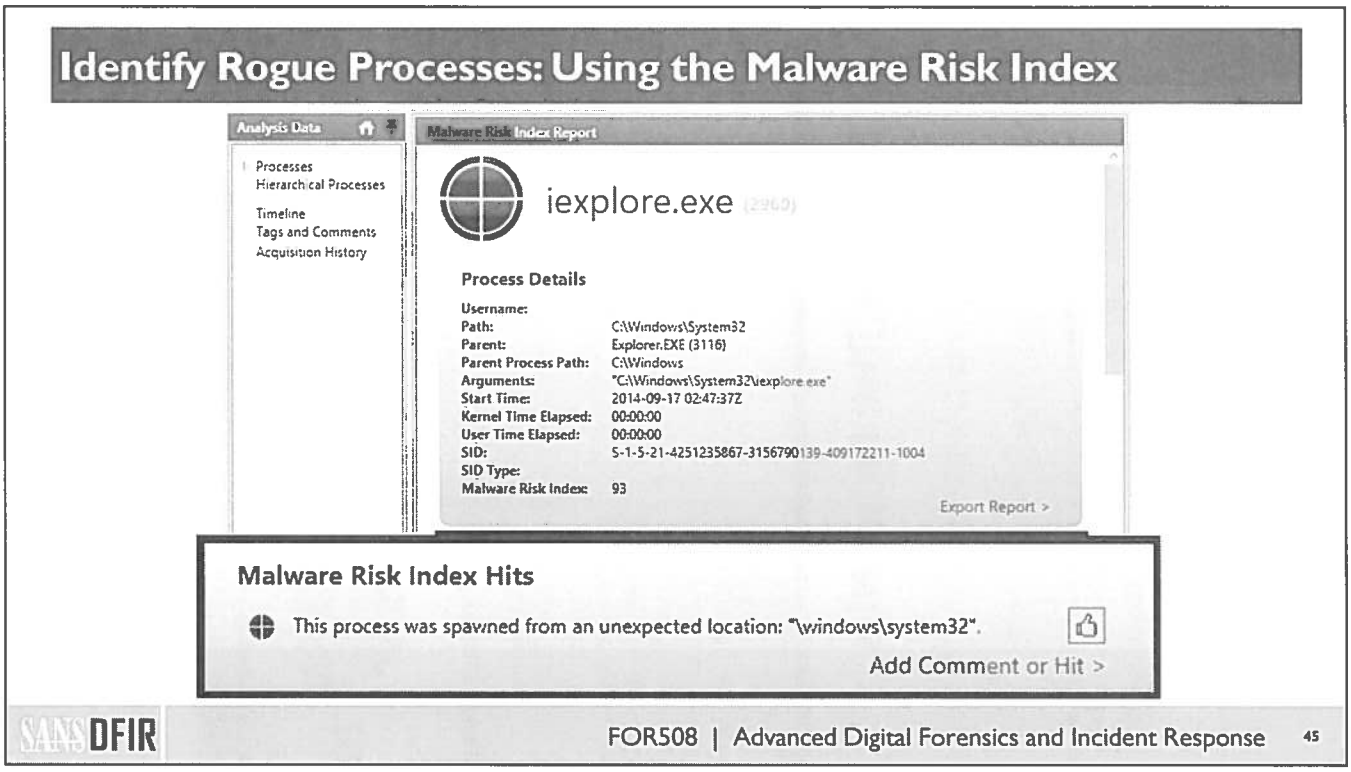
→ Options → MRI Rules Configuration



- ✓ Trust threshold
- ✓ Trusted Signers
- ✓ Exec Arguments
- ✓ Process owner
- ✓ Binary path
- ✓ Function Imports
- ✓ Handle Object

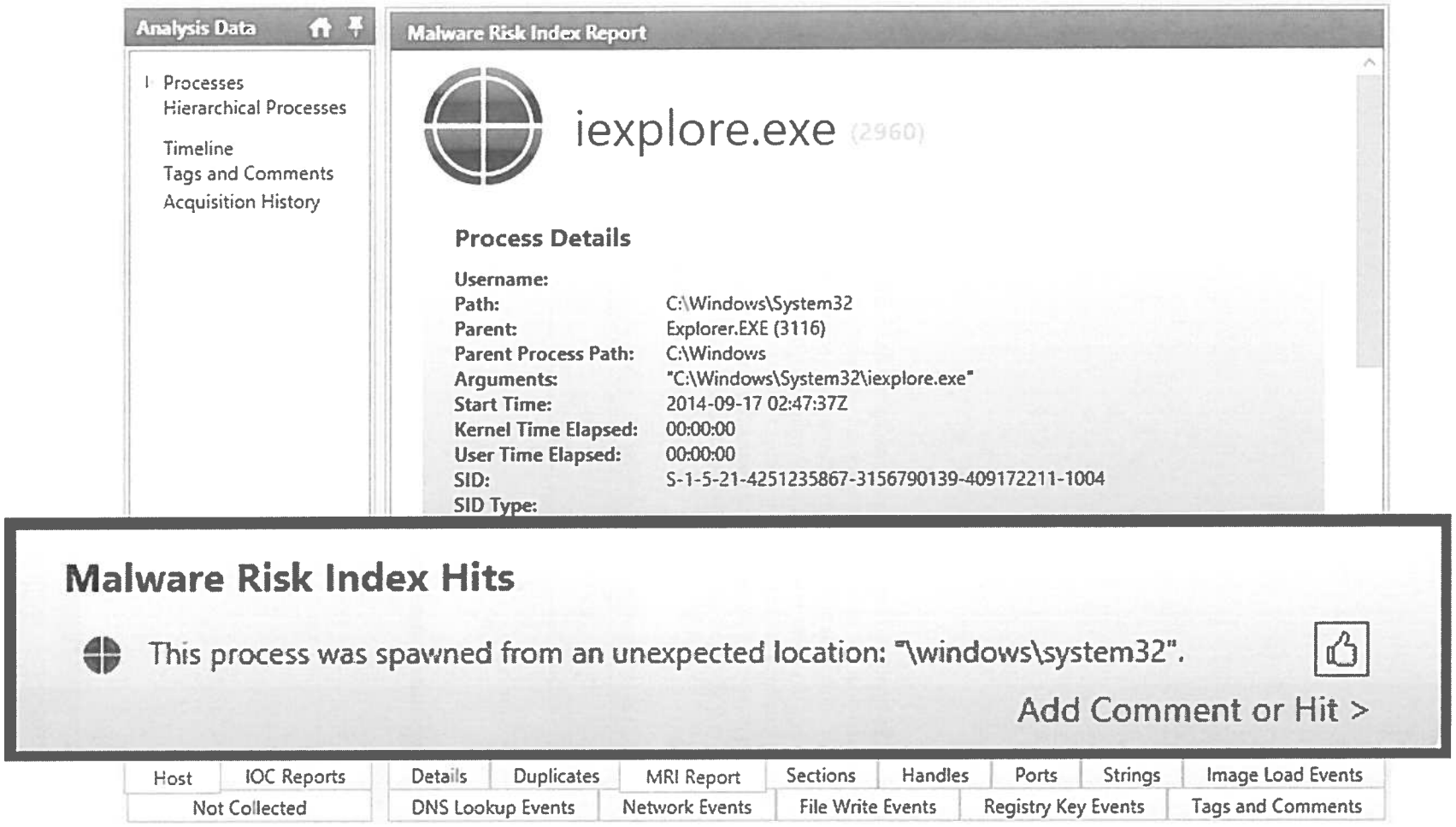
If you would like to see the current set of signatures, click the Redline logo, and then in the bottom, right-hand corner, you will see the Options button. You should see the MRI Rules Configuration dialog shown on this slide. There are very useful example signatures in each category to help you learn how to define a signature. In the image here, we see the Suspicious Handles option has been selected and a mutant is displayed. This is a very famous mutant in Advanced Persistent Threat (APT) circles because it points to the Poison Ivy Remote Access tool that has been very commonly used by that category of attackers. A great best practice would be to take the report from your malware reverse engineering team after each incident and build IOCs or Redline signatures for any new malware.

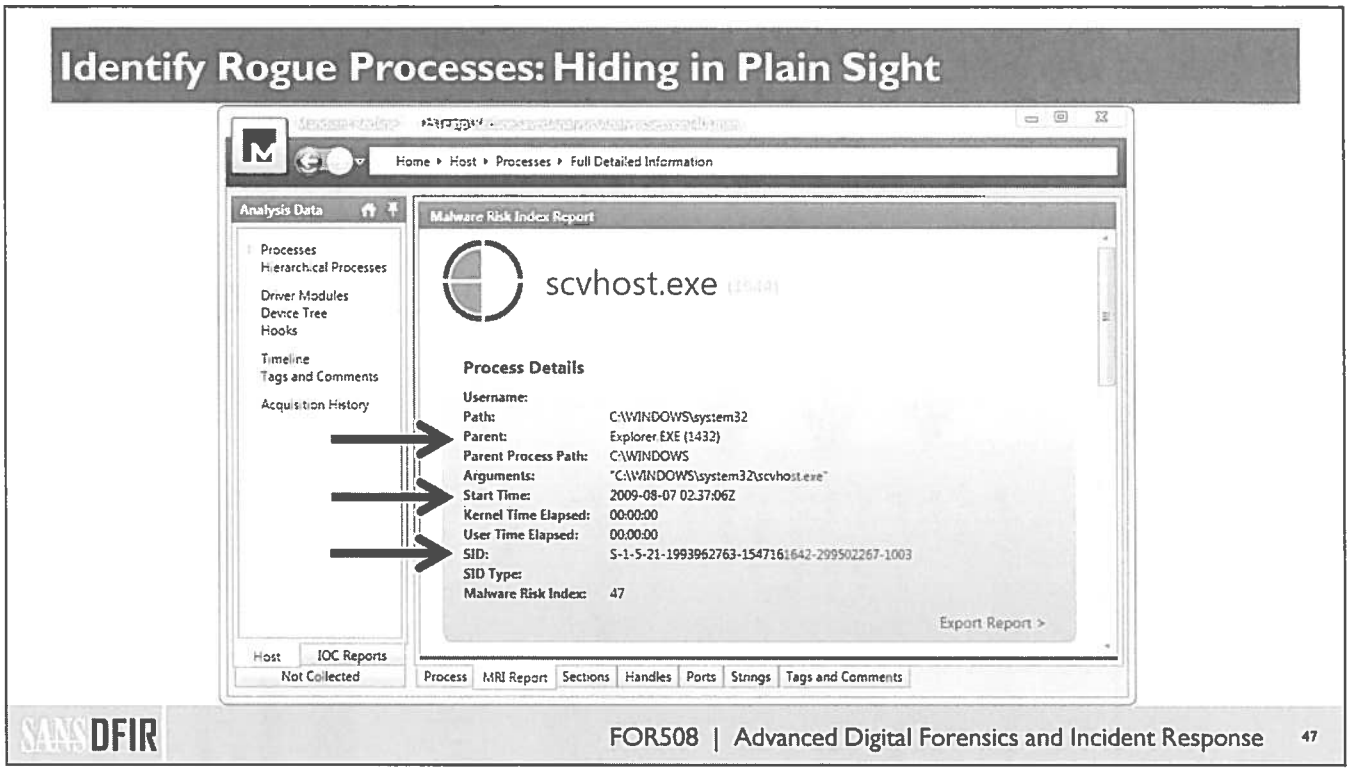
[1] https://blog.mandiant.com/archives/741



Identify Rogue Processes: Using the Malware Risk Index

Even if we didn't catch the path anomaly with the iexplore.exe process, Redline brings it to our attention. The reason Redline gave the iexplore.exe process a Malware Rating Index Score of 93 is listed here. It turns out that Redline's MRI was looking for iexplore.exe running from a folder other than Program Files or Program Files (x86). Although you might never be fooled by a rogue iexplore.exe process in the future, you would be surprised at how many seasoned Windows administrators might miss this. The real magic is that you didn't have to recognize it! Redline has a built-in heuristic, so we don't have to waste our time remembering things like this. Of course, those heuristics only get us so far, so we need to learn the multitude of the other ways we can identify evil processes. Read on!


Analysis Data
Processes
Hierarchical Processes
Timeline
Tags and Comments
Acquisition History
Malware Risk Index Report
iexplore.exe (2960)
Process Details
Username:
Path: C:\Windows\System32
Parent: Explorer.EXE (3116)
Parent Process Path: C:\Windows
Arguments: "C:\Windows\System32\iexplore.exe"
Start Time: 2014-09-17 02:47:37Z
Kernel Time Elapsed: 00:00:00
User Time Elapsed: 00:00:00
SID: S-1-5-21-4251235867-3156790139-409172211-1004
SID Type:
Malware Risk Index Hits
This process was spawned from an unexpected location: "\windows\system32".
Add Comment or Hit >
Host
IOC Reports
Details
Duplicates
MRI Report
Sections
Handles
Ports
Strings
Image Load Events
Not Collected
DNS Lookup Events
Network Events
File Write Events
Registry Key Events
Tags and Comments

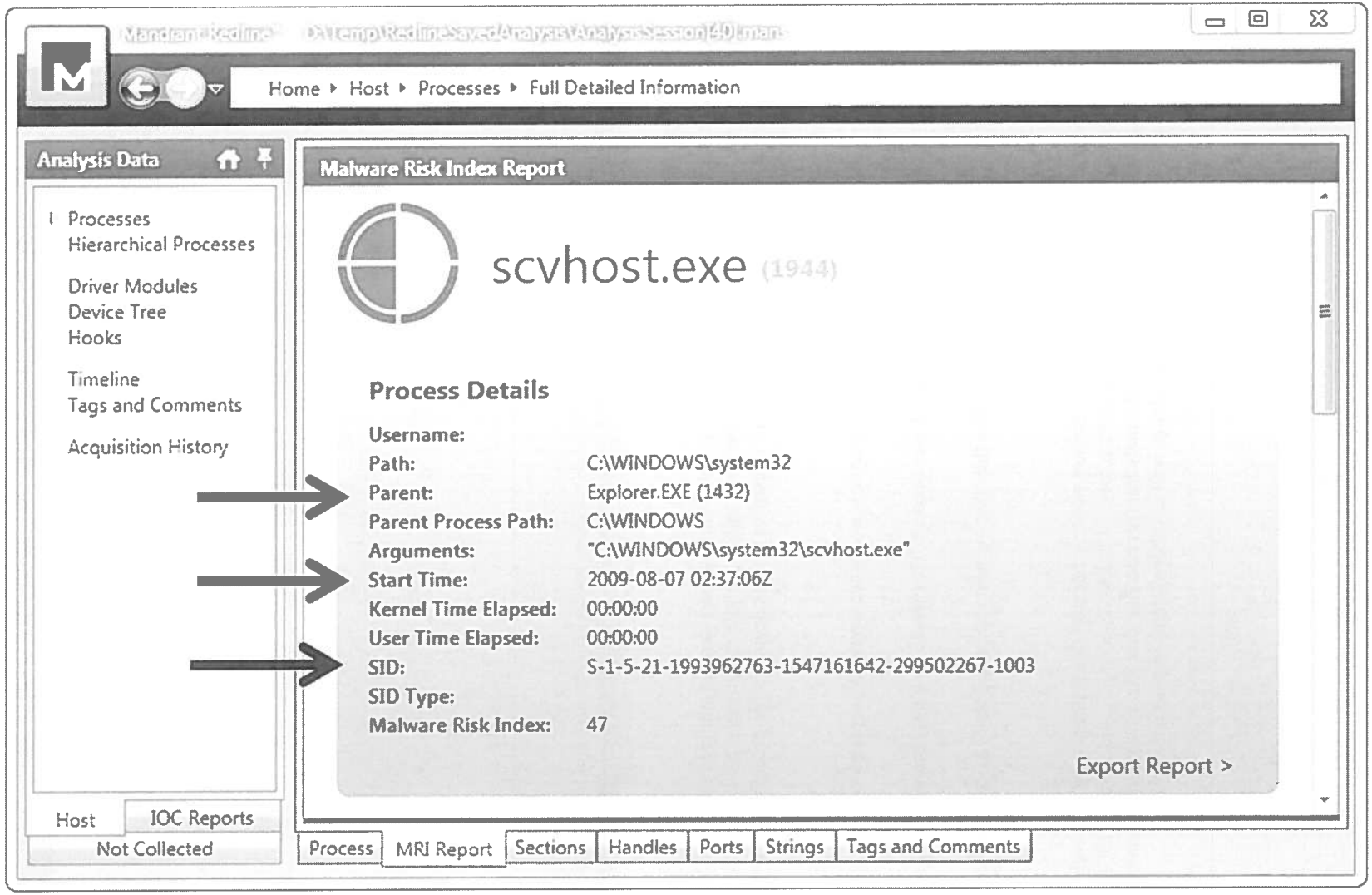


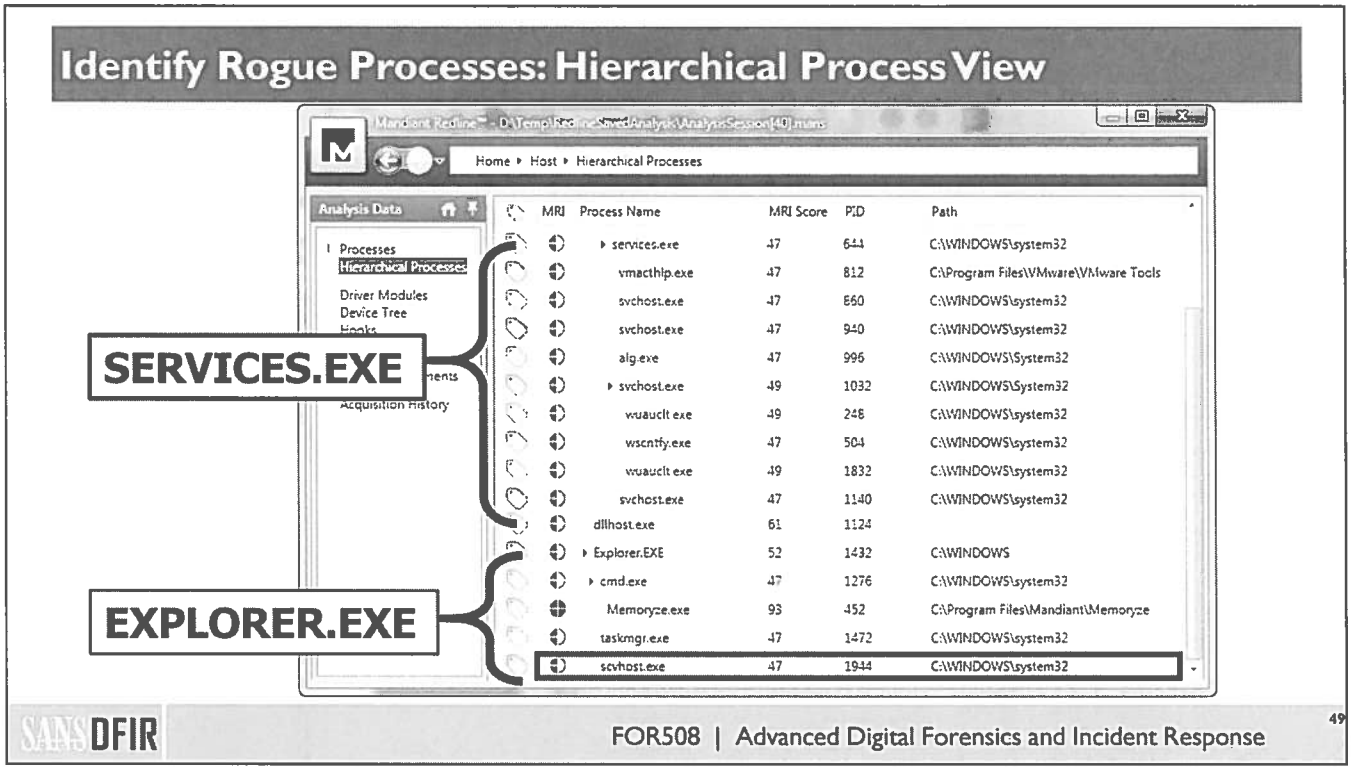
Identify Rogue Processes: Hiding in Plain Sight

In this example, we have an example of hiding in plain sight. This memory sample stumped us at first. We completely missed the misspelling and believed it to be just another svchost.exe process. Notice that Redline doesn't give it a particularly high MRI value. But what caused us to look deeper in the process details? Is there anything out of the ordinary in the process list?

- The parent of the process is Explorer.exe, not "Services.exe as we would expect for a svchost.exe process.
- The Security Identifier (SID) is for a standard user account. This would be highly unusual for a real svchost.exe process.
- This process was not started at the same time as the other svchost.exe processes (you can't tell that from this view).

Of course, it turned out that all of the previous information was true because this process is NOT a svchost.exe process. It just happens to be running from the same directory, \Windows\System32\, and have a VERY similar name. The good news is that once you find a malicious process like this, that misspelled name can be a very useful string to search for on the current system as well as other systems in your enterprise.

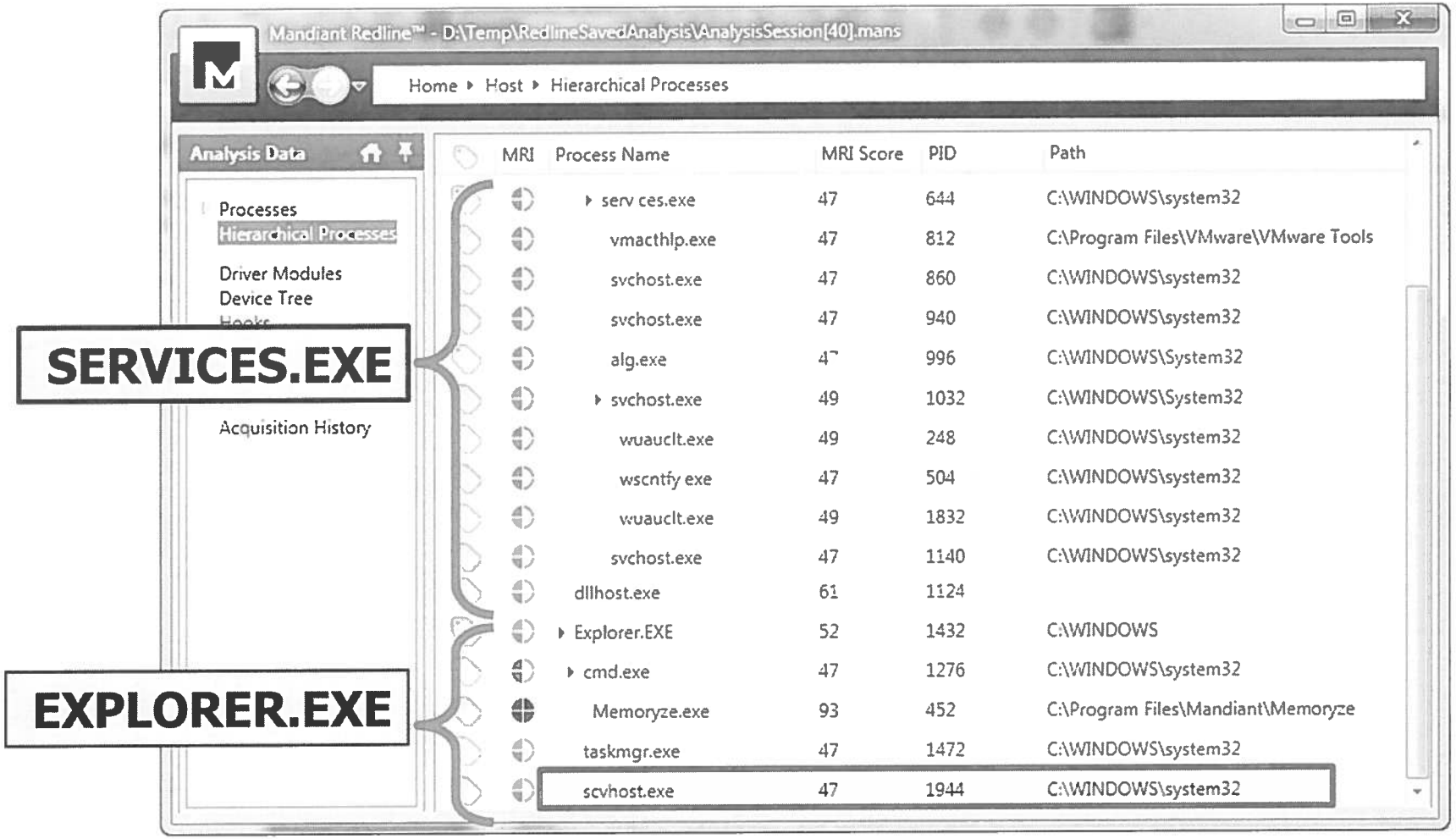

Home ▸ Host ▸ Processes ▸ Full Detailed Information
Analysis Data
Processes
Hierarchical Processes
Driver Modules
Device Tree
Hooks
Timeline
Tags and Comments
Acquisition History
Malware Risk Index Report
scvhost.exe (1944)
Process Details
Username:
Path: C:\WINDOWS\system32
Parent: Explorer.EXE (1432)
Parent Process Path: C:\WINDOWS
Arguments: "C:\WINDOWS\system32\scvhost.exe"
Start Time: 2009-08-07 02:37:06Z
Kernel Time Elapsed: 00:00:00
User Time Elapsed: 00:00:00
SID: S-1-5-21-1993962763-1547161642-299502267-1003
SID Type:
Malware Risk Index: 47
Export Report >
Host
IOC Reports
Not Collected
Process
MRI Report
Sections
Handles
Ports
Strings
Tags and Comments

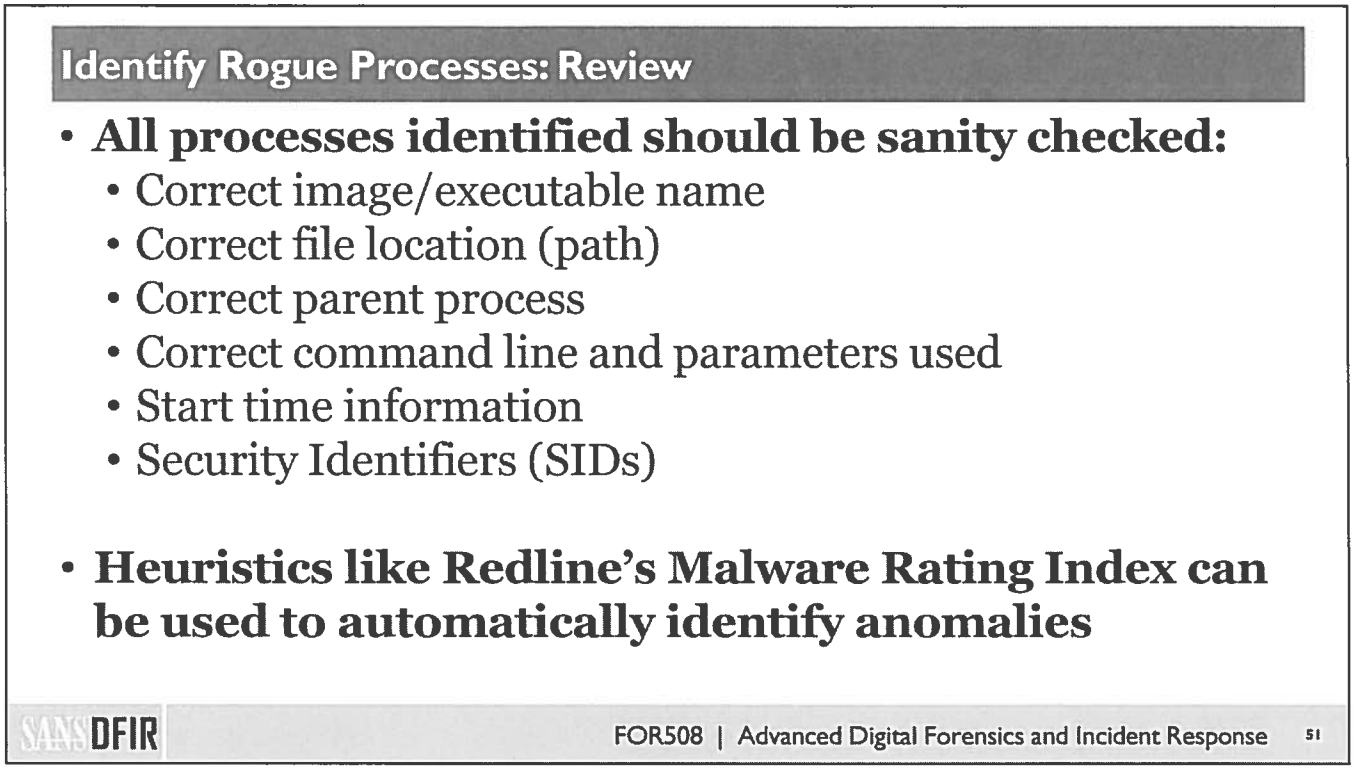


Identify Rogue Processes: Hierarchical Process View

Imagine that we didn't notice the process irregularities in scvhost.exe. How else might we have noticed that something was awry? If you go to the **Host View** tab in Redline, you can view the **Hierarchical Process** list. This view allows you to visually see what processes are parents to other processes. Looking under services.exe, we see all of the svchost.exe processes. Except under Explorer.exe, we also see one! Though a closer look shows it is named "scvhost.exe" and not the real svchost.exe. We often are looking at a lot of data and having the ability to visually recognize anomalies can sometimes be the difference between winning and losing.

The Volatility framework has this feature and HBGary added it in Nov 2011 to its Responder product.


Mandiant Redline™ - D:\Temp\RedlineSavedAnalysis\AnalysisSession[40].mans
Home ▸ Host ▸ Hierarchical Processes
Analysis Data
Processes
Hierarchical Processes
Driver Modules
Device Tree
Hooks
Acquisition History
SERVICES.EXE
EXPLORER.EXE
MRI
Process Name
MRI Score
PID
Path
serv ces.exe 47 644 C:\WINDOWS\system32
vmacthlp.exe 47 812 C:\Program Files\VMware\VMware Tools
svchost.exe 47 860 C:\WINDOWS\system32
svchost.exe 47 940 C:\WINDOWS\system32
alg.exe 47 996 C:\WINDOWS\System32
svchost.exe 49 1032 C:\WINDOWS\System32
wuauclt.exe 49 248 C:\WINDOWS\system32
wscntfy exe 47 504 C:\WINDOWS\system32
wuauclt.exe 49 1832 C:\WINDOWS\system32
svchost.exe 47 1140 C:\WINDOWS\system32
dllhost.exe 61 1124
Explorer.EXE 52 1432 C:\WINDOWS
cmd.exe 47 1276 C:\WINDOWS\system32
Memoryze.exe 93 452 C:\Program Files\Mandiant\Memoryze
taskmgr.exe 47 1472 C:\WINDOWS\system32
scvhost.exe 47 1944 C:\WINDOWS\system32



Identify Rogue Processes: Review

No one becomes an expert in memory forensics on his or her first day. Through experience, you will learn to quickly identify anomalies in processes related to:

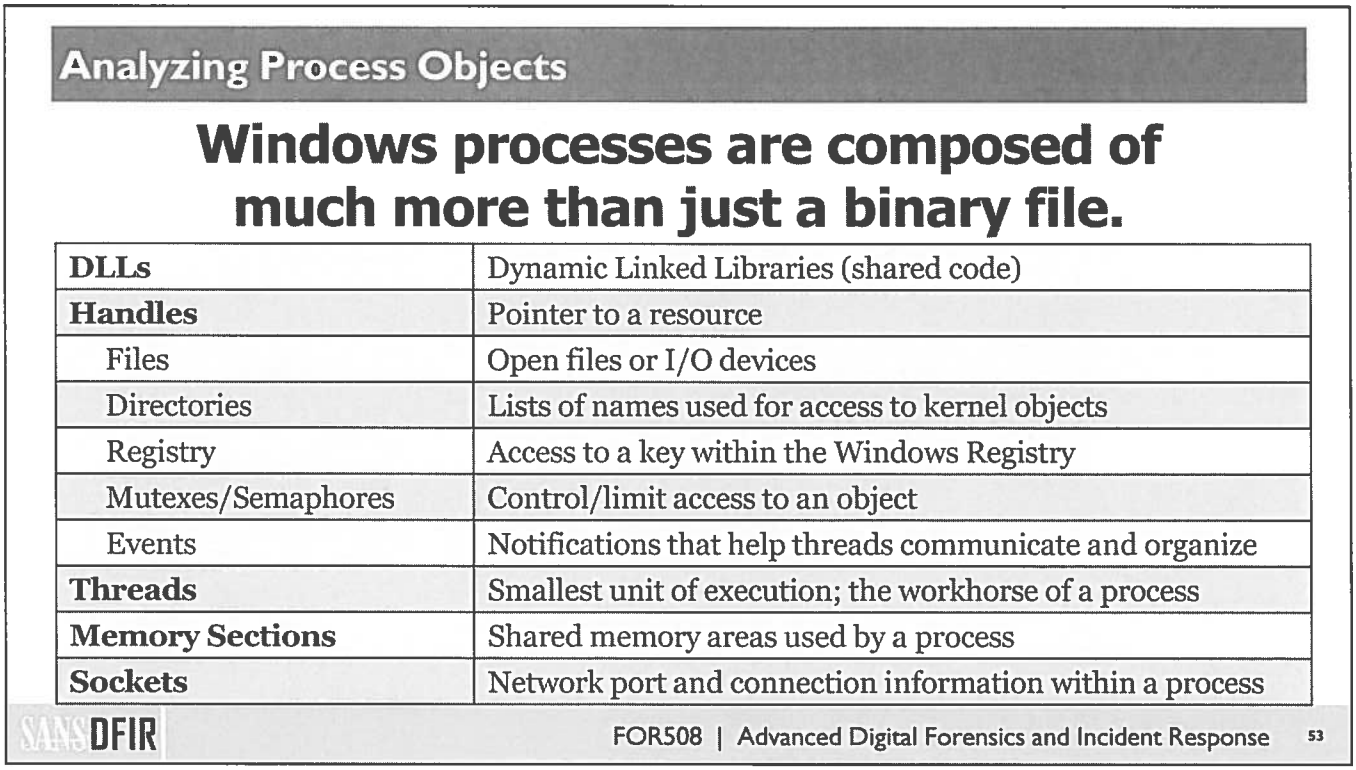
- Correct image/executable name
- Correct file location (path)
- Correct parent process
- Correct command line and parameters used
- Start time information
- Security Identifiers (SIDs)

The good news is that by starting out with tools like Redline, you have some training wheels that will help point you towards things you might otherwise have overlooked. Technologies like the Malware Rating Index aren't perfect, but they can be very helpful when learning, or when you need to search for something quickly.

Step 2:

Analyzing Process Objects

This page intentionally left blank.



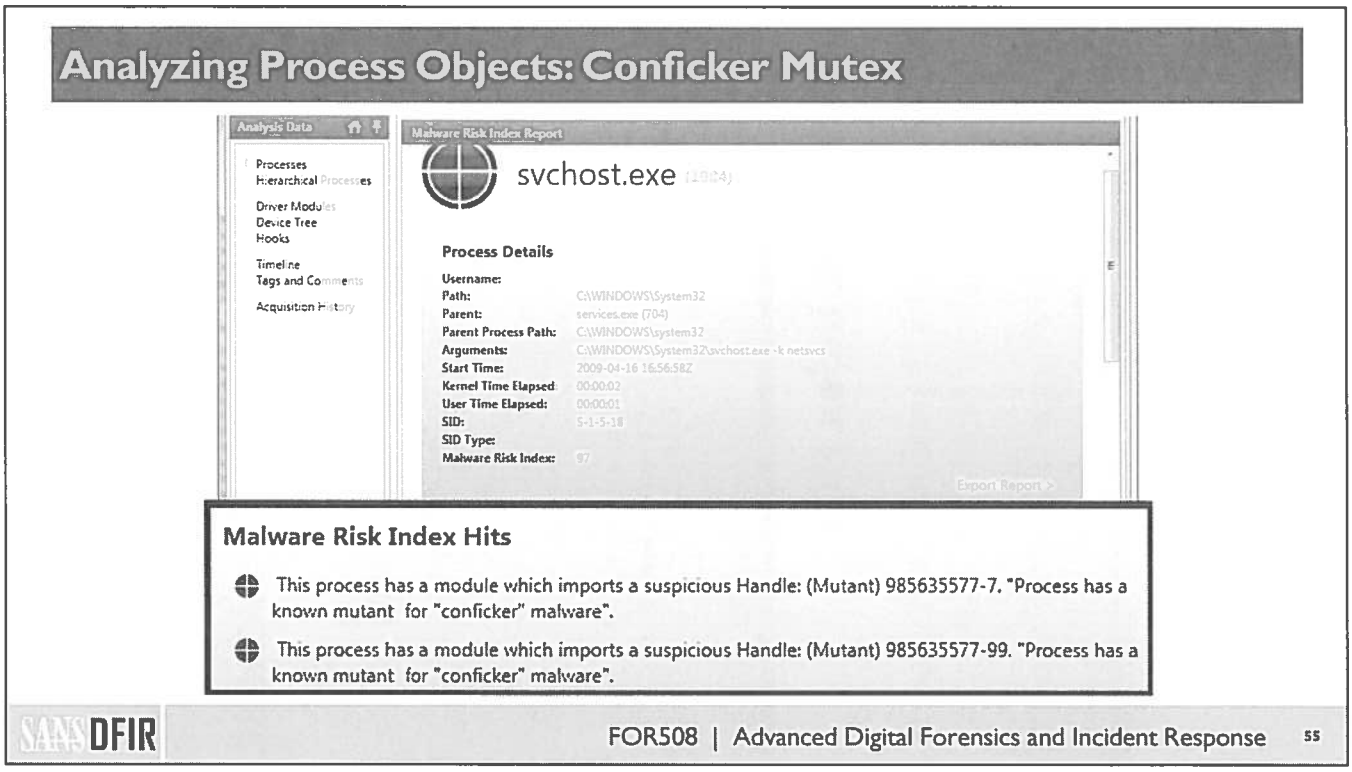
Analyzing Process Objects

There is a whole lot more to processes than just image names and parent processes. To identify some harder-to-find malware, we are going to have to dig deeper. Processes can have hundreds of associated objects.[1] In fact, the per-process limit on kernel handles is 2^24, so there can be a vast number of objects to review! Luckily, most processes don't get near the upper limit. Identifying a suspicious process object can help inform how much we trust a given process. The following objects can be reviewed:

- **DLLs**: Dynamically Linked Libraries define the capabilities of a process. For instance, if a process needs to communicate via HTTP, it will load the WININET.dll file. In some cases, malware will load its own malicious DLLs to take control of a process.
- **Handles**: A pointer to a resource, handles exist in many different forms. Some of the most important to memory analysis are:
    - **File handles:** Identify which items in the file system or which I/O devices are being accessed by the process.
    - **Directory handles:** This is not your standard file system directory. Instead, directory handles are known lists within the kernel that allow the process to find kernel objects. Common examples are KnownDlls, BaseNamedObjects, Callbacks, Device, and Drivers.
    - **Registry handles:** These are the registry keys the process is reading or writing to.
    - **Mutex or semaphore handles:** Also called "mutants," these objects control or limit access to a resource. For instance, a mutex might be used by an object to enforce that only one process at a time can access it. Worms commonly set mutexes as a way of "marking" a compromised system so that it does not get re-infected.
    - **Event handles:** Events are a way for process threads to communicate. Malware will occasionally use unique event handles
- **Threads**: A process is just a container for all of the items that do the real work. Multiple threads run within every process interacting with various system objects.

- **Memory sections:** Every process has a collection of virtual memory pages where DLLs and files are loaded and code and data are stored. The Virtual Address Descriptor tree (VAD) maintains a list of these assigned memory sections.
- **Sockets:** These are network connection endpoints. Every network socket is assigned to a specific process, allowing us to trace back suspicious network activities.

[1] http://en.wikipedia.org/wiki/Object Manager (Windows)

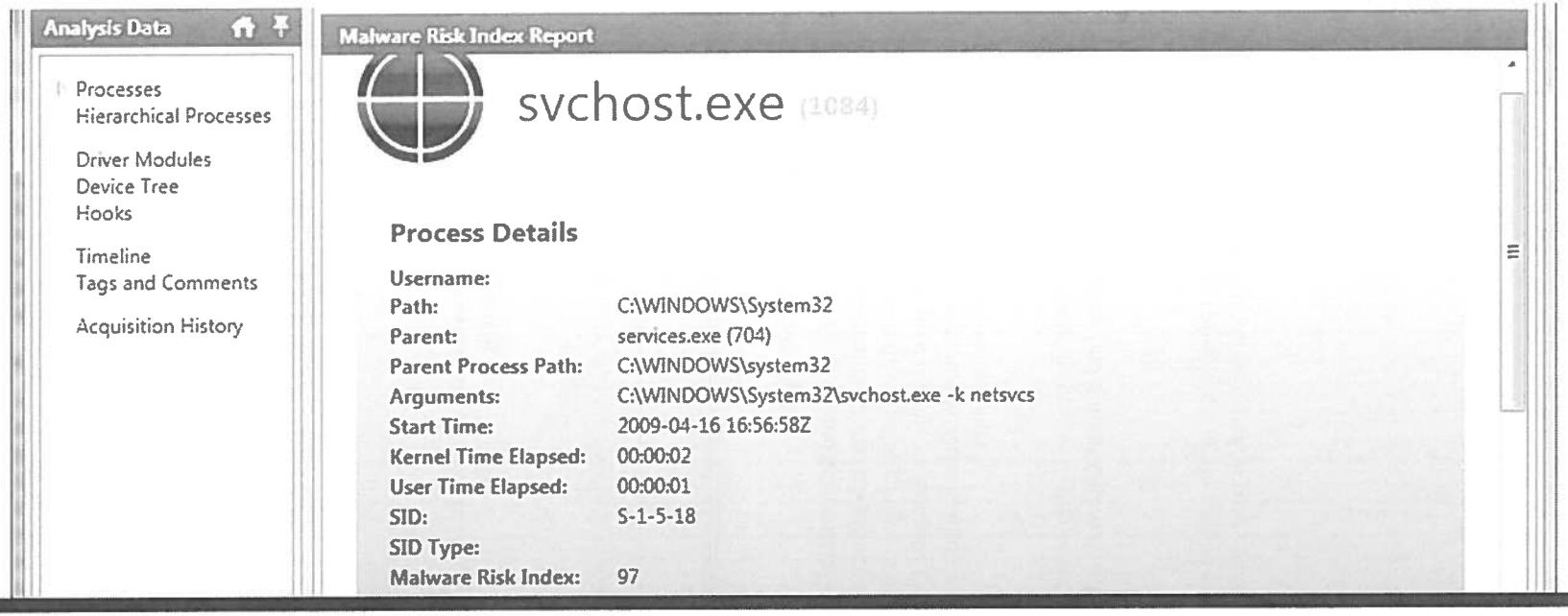


Analyzing Process Objects: Conficker Mutex

The Conficker, or Kido, worm was one of the most successful worms in history and is still active in the wild today. This variant of Conficker injected itself into a legitimate svchost.exe process. Thus, if you were to look closely at the process details, you would see nothing anomalous. The SID, launch time, path, and parent process are all as expected. It is the process objects that give Conficker away. In this case, Redline picks up on two known mutants for Conficker, seen in this slide. Redline assigns an MRI value of 97 to this svchost.exe process due to the existence of these two mutants.

Imagine now that Redline did not have the proper signature because this was a new variant of Conficker or maybe a 0-day threat. How would we find it then? One way is by leveraging Least Frequency of Occurrence information.

Special thanks to Mandiant for providing some of the sample memory images seen in these examples. Most of the example memory images shown can be found on your course USB drive.



Malware Risk Index Hits

- This process has a module which imports a suspicious Handle: (Mutant) 985635577-7. "Process has a known mutant for "conficker" malware".
- This process has a module which imports a suspicious Handle: (Mutant) 985635577-99. "Process has a known mutant for "conficker" malware".

Analyzing Process Objects: Least Frequency of Occurrence

- Malware and its associated artifacts should be among the rarest objects in a memory image
- Redline keeps a count of each time a process object is referenced
- Sort by the Occurrences column to identify outliers

Name	Occurrence	Trust Status
\Device\HarddiskVolume1\Documents and Settings\Owner\Local Settings\History...	2	
\Device\HarddiskVolume1\WINDOWS\system32	19	
\Device\HarddiskVolume1\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-...	55	

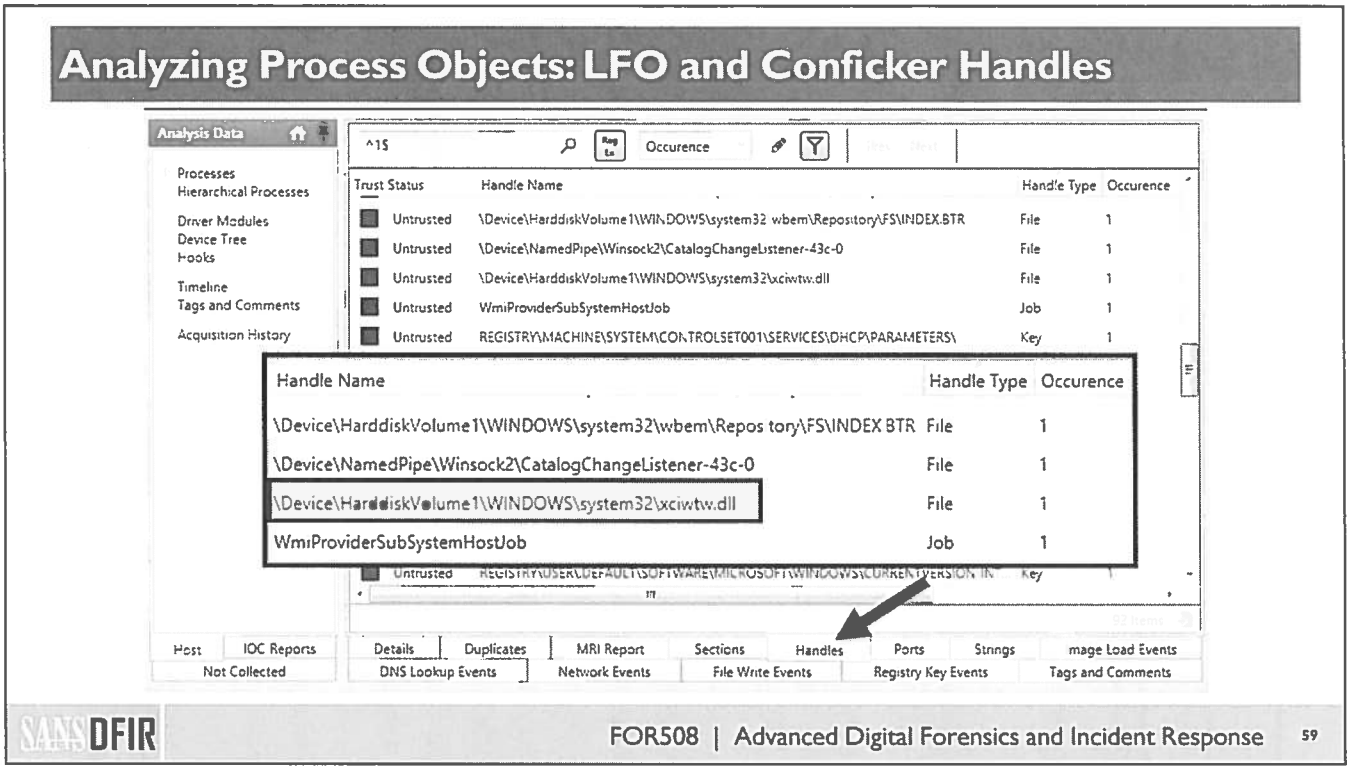
**** A process object occurring only once is not de facto malicious, but should be trusted less than one that appears in 23 instances**

Analyzing Process Objects: Least Frequency of Occurrence

The problem with analyzing process objects is that they can be so numerous. How do you identify the one malicious process mutex out of hundreds? Redline (and Memoryze before it) is pioneering the use of Least Frequency of Occurrence (LFO) to help with this problem. The LFO concept is easy: Anything related to malware should be among the least frequently occurring objects on a system (or in an enterprise for that matter). If we can keep track of how many processes reference a DLL, executable, mutant, or registry key, then we can sort our lists of the objects by count and focus on the outliers. In Redline, this column is named "Occurrences." Every handle type listed in Redline has this column available.

It is important to recognize that this is but one tool and isn't a silver bullet for every situation. There are plenty of objects that occur only in one process. Having a low occurrence count does not prima facie indicate something is malicious. But it has a higher chance of being malicious than another similar object found in 75% of the processes. Thus, we can use LFO to help us eliminate large numbers of objects so we can focus our efforts on a smaller set.

Name	Occurrence	Trust Status
\Device\HarddiskVolume1\Documents and Settings\Owner\Local Settings\History...	2	
\Device\HarddiskVolume1\WINDOWS\system32	19	
\Device\HarddiskVolume1\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-...	55	

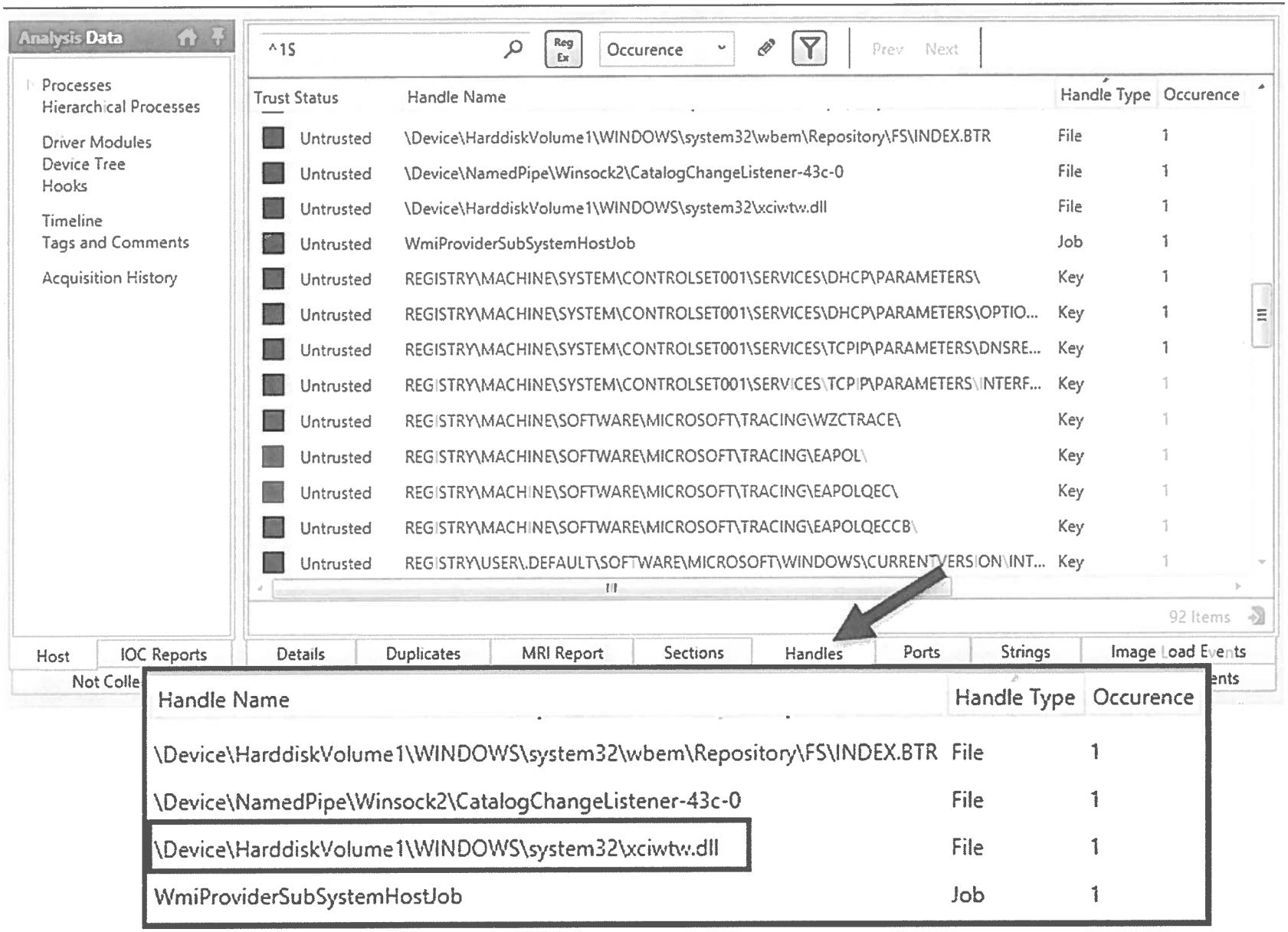


Analyzing Process Objects: LFO and Conficker Handles

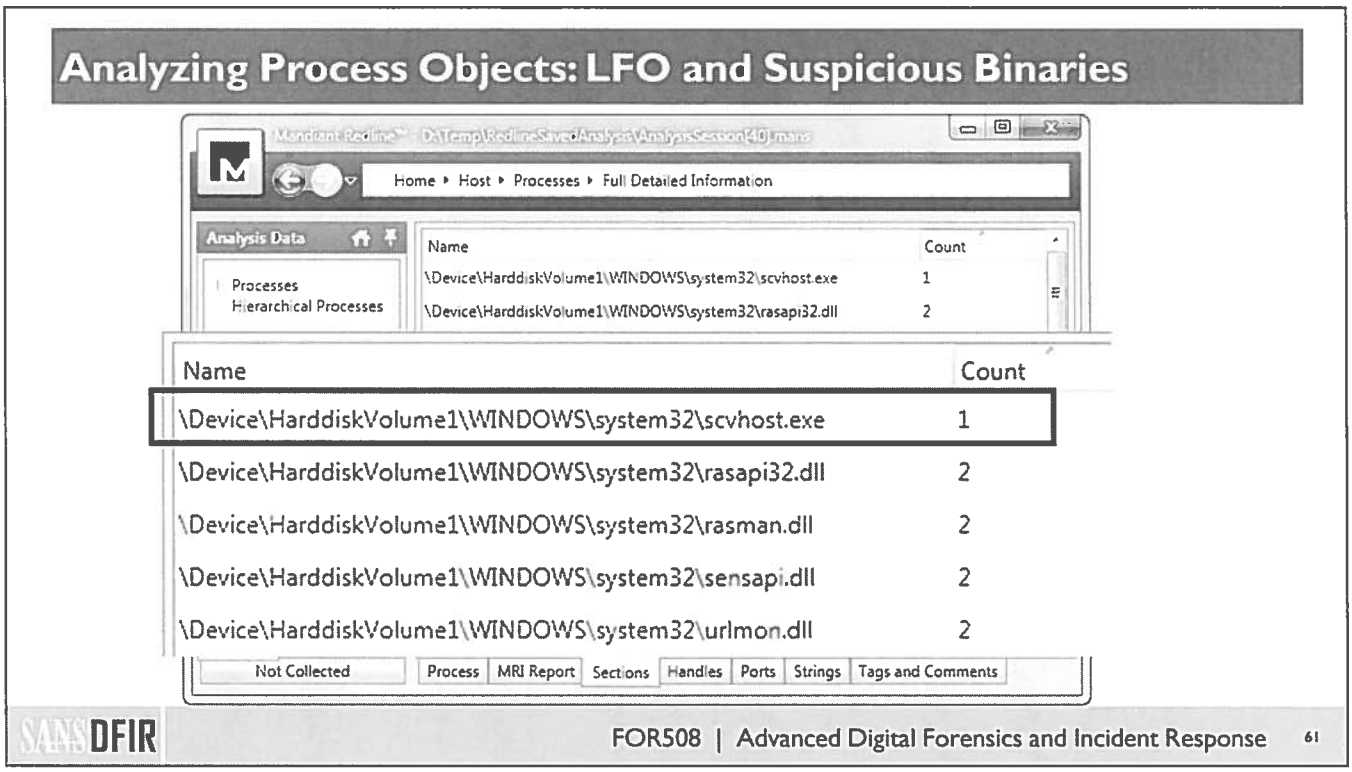
Returning to our Conficker example, let's pretend that the Conficker mutant signature wasn't built into Redline. Maybe this is a new variant or some completely new 0-day piece of malware that has no available signatures. Let's see how Least Frequency of Occurrence (LFO) can help us find those same artifacts *without* pre-built signatures.

Without any pre-existing indicators, a reasonable place to start might be by analyzing all of the svchost.exe processes (we are playing the odds here because a high percentage of malware can be tied to this process name). Here, we selected the svchost.exe process and opened its file handle list. We then filtered for any handles with an occurrence count of one (this could also have been done by sorting the column). In this real-world case, there were 1017 handles in this process, but only 92 that were referenced only by this specific process (a count of one means this is the only existence of that handle in the entire memory image). By performing a quick review of all of the LFO handles, an interestingly named DLL was identified. This turned out to be a malicious DLL on disk that led us to additional artifacts used by this particular variant of Conficker.

Redline is currently the only memory analysis tools with LFO capabilities.



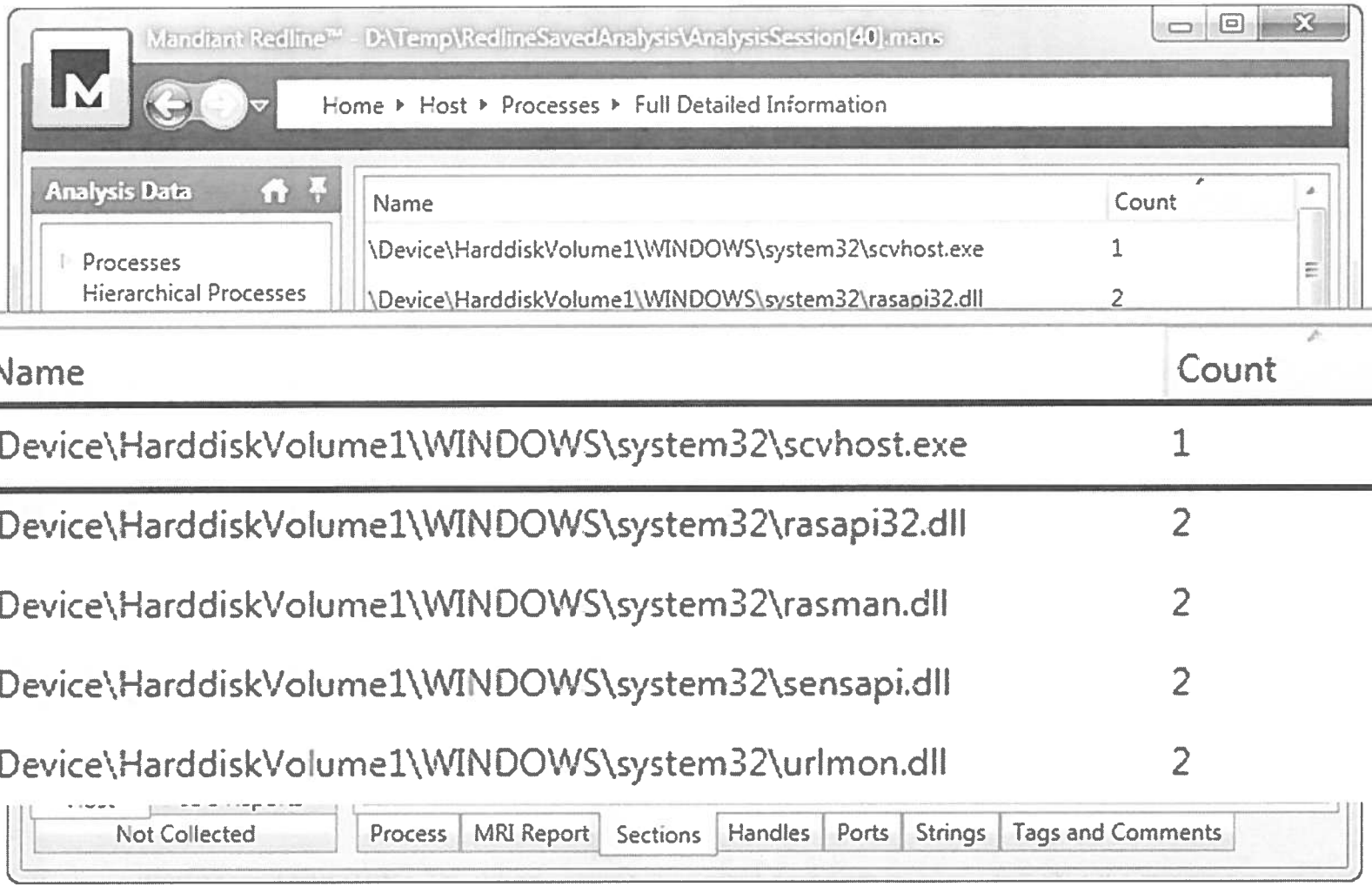
Handle Name	Handle Type	Occurence
\Device\HarddiskVolume1\WINDOWS\system32\wbem\Repository\FS\INDEX.BTR	File	1
\Device\NamedPipe\Winsock2\CatalogChangeListener-43c-0	File	1
\Device\HarddiskVolume1\WINDOWS\system32\xciwtw.dll	File	1
WmiProviderSubSystemHostJob	Job	1



Analyzing Process Objects: LFO and Suspicious Binaries

Here is one more example using Least Frequency of Occurrence (LFO). This time, we will look back at that process hiding in plain sight named "scvhost.exe." In this case, we sorted the processes' 38 memory sections in order of occurrence. There was one memory mapped file with only one occurrence throughout memory, and it happened to be the process image executable itself. Imagine that we hadn't yet identified this process as suspicious through other means. If this were a true svchost.exe process, we would see a very high number of occurrences because svchost.exe is used throughout the operating system. The fact that we see only one occurrence should make us look a little harder, finally noticing that the process name is actually scvhost.exe and providing very strong evidence that this process should be examined more closely.

The goal in any forensic analysis is to not have to be perfect. we want our tools and processes to be layered so that if we happen to miss something on the first iteration, we have a safety net that will guide us to the answer in a following iteration. It is really hard to be perfect and tools like LFO help point us to things we might have ordinarily missed.



Name	Count
\Device\HarddiskVolume1\WINDOWS\system32\scvhost.exe	1
\Device\HarddiskVolume1\WINDOWS\system32\rasapi32.dll	2
\Device\HarddiskVolume1\WINDOWS\system32\rasman.dll	2
\Device\HarddiskVolume1\WINDOWS\system32\sensapi.dll	2
\Device\HarddiskVolume1\WINDOWS\system32\urlmon.dll	2

Analyzing Process Objects: Review

- **In many cases, the objects that make up a process will provide a clue that something is amiss:**
  - DLLs
  - Handles
  - Threads
  - Memory sections
  - Sockets
- **There can be thousands of objects associated with a given process**
  - Least Frequency of Occurrence can help identify outliers

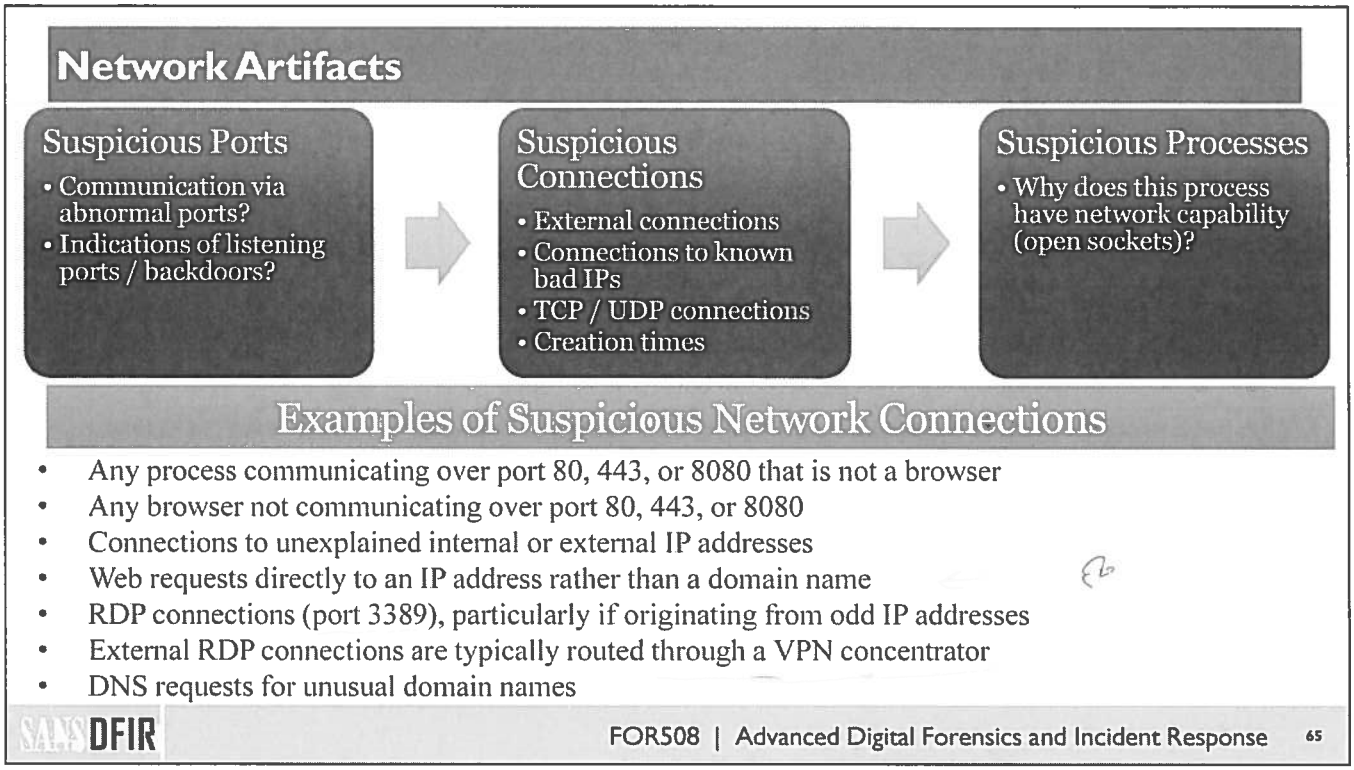
Analyzing Process Objects: Review

We started our analysis checklist with reviewing process attributes because they are less numerous and anomalies are often easy to spot. When we move down to process objects, our job becomes more difficult. Now instead of maybe one hundred or less processes to review, we might have thousands of process objects. We can limit them by focusing on objects for processes we believed to be suspicious in the previous step, or by using the heuristics and signature-based checks built into the Redline tool. Perhaps the most effective tool for making process object analysis feasible is Least Frequency of Occurrence, allowing us to sort objects by how often they are referenced and focusing our limited resources on just the outliers.

Step 3:

Network Artifacts

This page intentionally left blank.



Network Artifacts

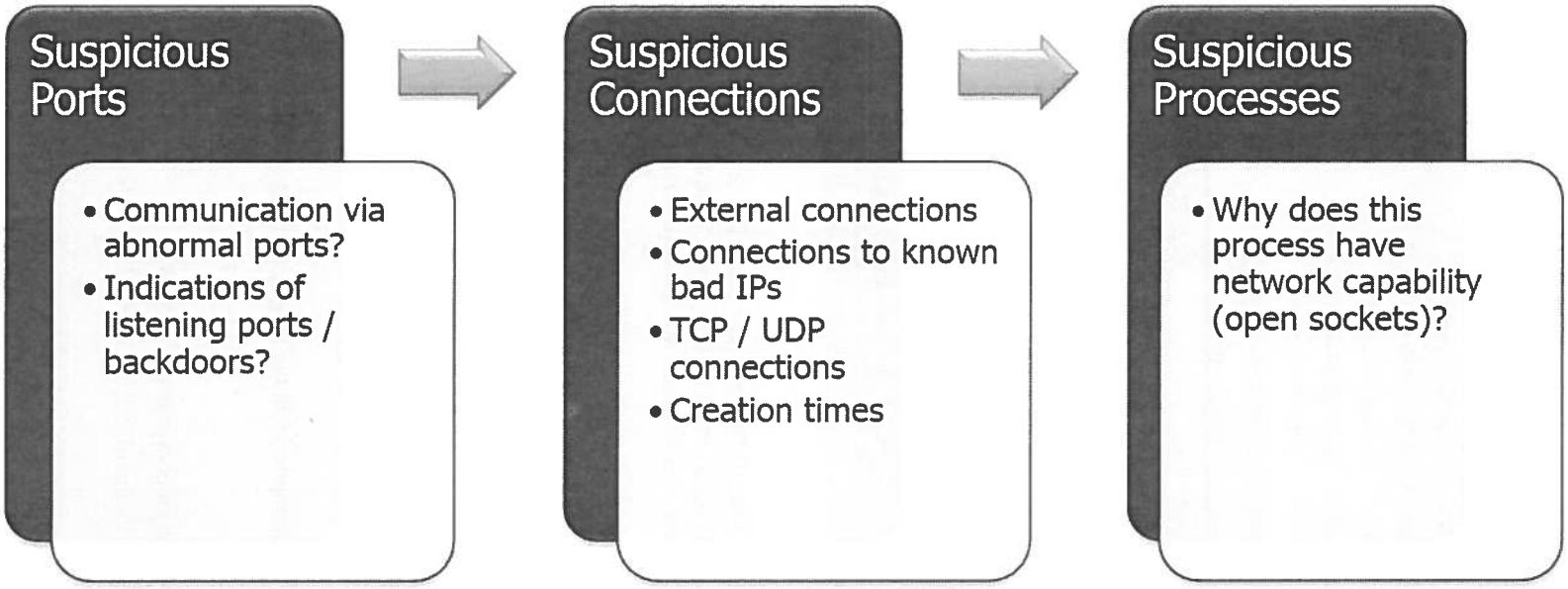
Network connections are a tried-and-true method for finding malware. Remember, those old listening ports for Back Orifice? Although it isn't quite that easy these days with outbound beaconing at infrequent intervals being the norm, it can still often lead us in the right direction. There are several things we look for when analyzing network connections for suspicious activity:

- **Suspicious ports:** Although we don't see listening ports serving as backdoors nearly as frequently, they are still out there. Pay attention to what ports are active on a typical system in the environment and look for those that don't fit. What is that process communicating via port 6667? Odds are that it is Internet Relay Chat and probably isn't considered normal.
- **Suspicious connections:** Modern malware has largely moved to outbound connectivity (for example, beaconing) over highly utilized ports, such as 80 and 443. Although this traffic blends in well, it is still possible to detect, especially during memory analysis. In fact, this can sometimes be advantageous. Instead of seeing a connection from a svchost.exe process to a random high port, if you see that same connection communicating to port 80 of an external machine, you can be much more assured that it is a malicious connection. Pay attention to the ports used and what that tells you about directionality. A reserved or well-known port often indicates an inbound connection. Whether the suspicious connection was activated inbound or outbound might give you more information about the malware and your own network defenses. Don't focus only on TCP connections. Some malware purposely uses UDP. For instance, several worms use UDP for peer-to-peer communication with their command and control servers. Finally, don't forget that network sockets have a creation time. This can be very useful for finding other artifacts like related processes and threads initiated near the same time.
- **Suspicious processes:** Should the process communicate via the network at all? One of the amazing abilities we have with memory analysis is that old, terminated network sockets might still be recovered out of memory. Thus even if the evil process wasn't communicating during the short time when memory was acquired, we have a chance to identify it with previously opened sockets.

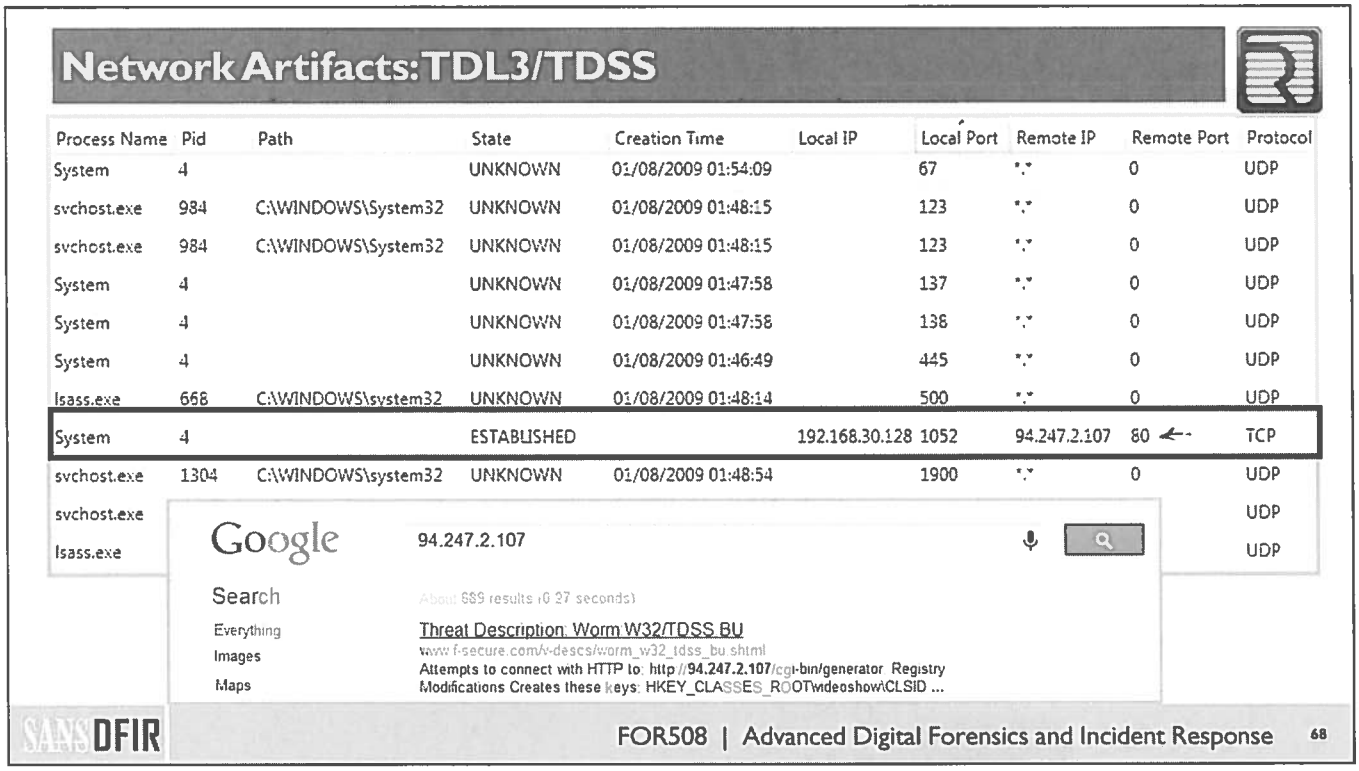
The example in this slide comes from Redline. We see an established network connection owned by a svchost.exe process. That alone is not suspicious, but the communication ports are. This turned out to be Metasploit communicating outbound to the attacking system. The tip off was that it would be highly unusual to see a svchost process communicating outbound to an external server.

When you are just starting to try to identify unusual network behavior, keep an eye out for the following:

- Any process communicating over port 80, 443, or 8080 that is not a browser
- Any browser not communicating over port 80, 443, or 8080
- Connections to unexplained internal or external IP addresses. For example, why did a process have a TCP connection to a system in Moldova?
- Web requests directly to an IP address rather than a domain name
- RDP connections (port 3389), particularly if originating from odd IP addresses. External RDP connections are typically routed through a VPN concentrator.
- DNS requests for unusual domain names



Process Name	PID	Path	State	Creation Time	Local IP	Local Port	Remote IP	Remote Port	Protocol
svchost.exe	1012	C:\WINDOWS\System32	UNKNOWN	2008-11-25 02:25:41Z		123	*.*	0	UDP
svchost.exe	1012	C:\WINDOWS\System32	UNKNOWN	2008-11-25 02:25:41Z		123	*.*	0	UDP
svchost.exe	1012	C:\WINDOWS\System32	ESTABLISHED		192.168.1.10	40810	184.84.0.60	4444	TCP
svchost.exe	1108	C:\WINDOWS\system32	UNKNOWN	2008-11-25 02:25:42Z		1900	*.*	0	UDP
svchost.exe	1108	C:\WINDOWS\system32	UNKNOWN	2008-11-25 02:25:42Z		1900	*.*	0	UDP



Network Artifacts: TDL3/TDSS

The example in this slide comes from the TDL3/TDSS bootkit, which is one of the more advanced pieces of malware in wild.[1] It is a highly advanced rootkit, even taking steps to hide its files outside of the defined partition. It provides a great example of how even the most advanced malware can be readily identified through memory forensics.

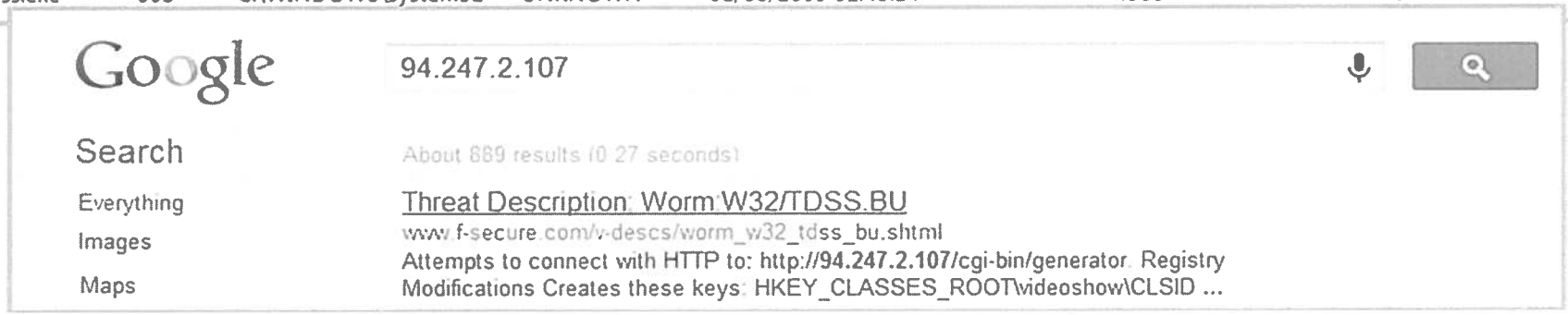
In this case, we see an established connection from the System process outbound (using a non-reserved port) to an external IP address on port 80. This is a classic outbound beaconing connection to a command and control server. What makes it most obvious is that it is coming from the System process. What if the malware instead created a new firefox.exe process and used it for the outbound beaconing? It is possible that we would not find much suspicious activity in network artifacts and would instead need to find clues in the process details or process objects. However, if we did a search on the external IP address, we would quickly find evidence indicating that IP address is a well-known command and control server for TDSS.

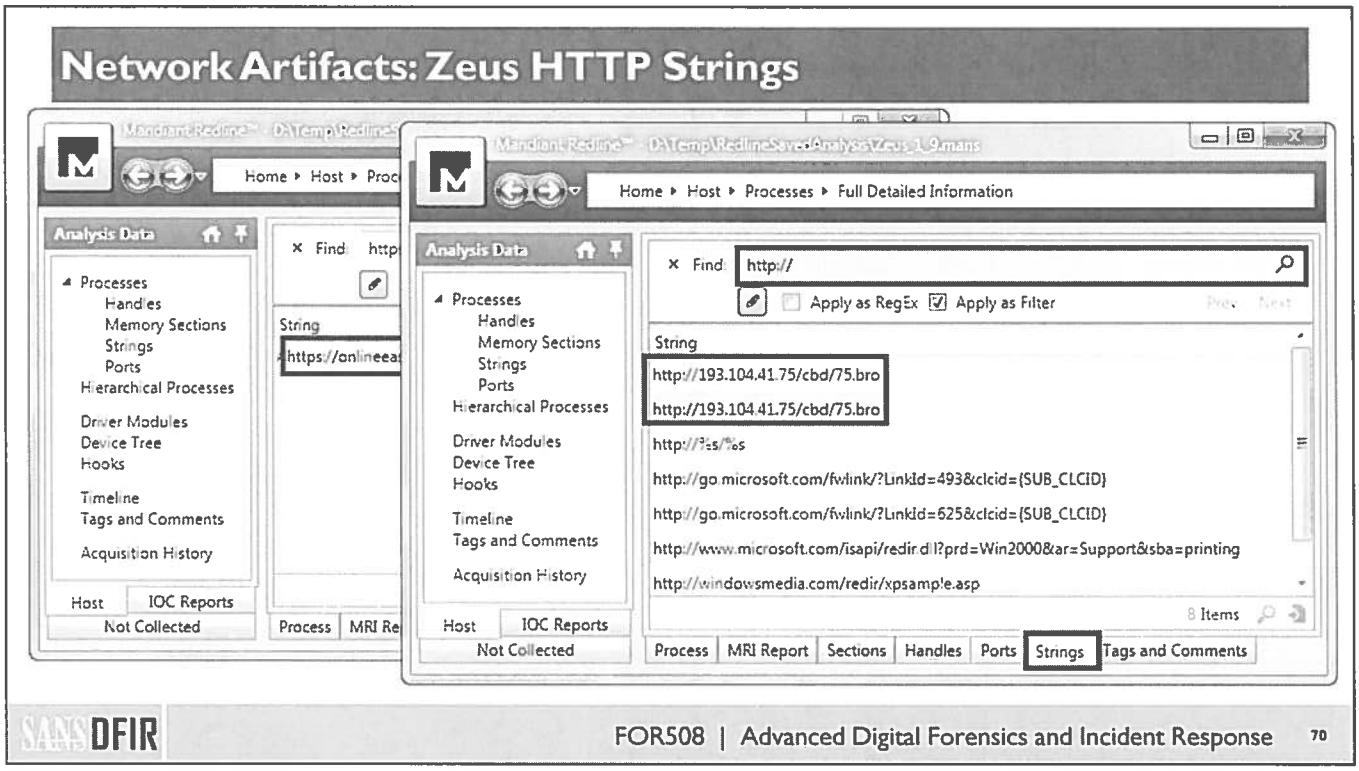
The TDSS memory sample used here is Exemplar 18 in the Hogfly memory sample collection.[2] It is provided on your course USB drive.

[1] http://www.liutilities.com/malware/computer-worm/w32-tidserv/

[2] https://skydrive.live.com/?cid=5694a755c9c6a175&id=5694A755C9C6A175!106

Process Name	Pid	Path	State	Creation Time	Local IP	Local Port	Remote IP	Remote Port	Protocol
System	4		UNKNOWN	01/08/2009 01:54:09		67	*.*	0	UDP
svchost.exe	984	C:\WINDOWS\System32	UNKNOWN	01/08/2009 01:48:15		123	*.*	0	UDP
svchost.exe	984	C:\WINDOWS\System32	UNKNOWN	01/08/2009 01:48:15		123	*.*	0	UDP
System	4		UNKNOWN	01/08/2009 01:47:58		137	*.*	0	UDP
System	4		UNKNOWN	01/08/2009 01:47:58		138	*.*	0	UDP
System	4		UNKNOWN	01/08/2009 01:46:49		445	*.*	0	UDP
lsass.exe	668	C:\WINDOWS\system32	UNKNOWN	01/08/2009 01:48:14		500	*.*	0	UDP
System	4		ESTABLISHED		192.168.30.128	1052	94.247.2.107	80	TCP
svchost.exe	1304	C:\WINDOWS\system32	UNKNOWN	01/08/2009 01:48:54		1900	*.*	0	UDP
svchost.exe	1304	C:\WINDOWS\system32	UNKNOWN	01/08/2009 01:48:54		1900	*.*	0	UDP
lsass.exe	668	C:\WINDOWS\system32	UNKNOWN	01/08/2009 01:48:14		4500	*.*	0	UDP



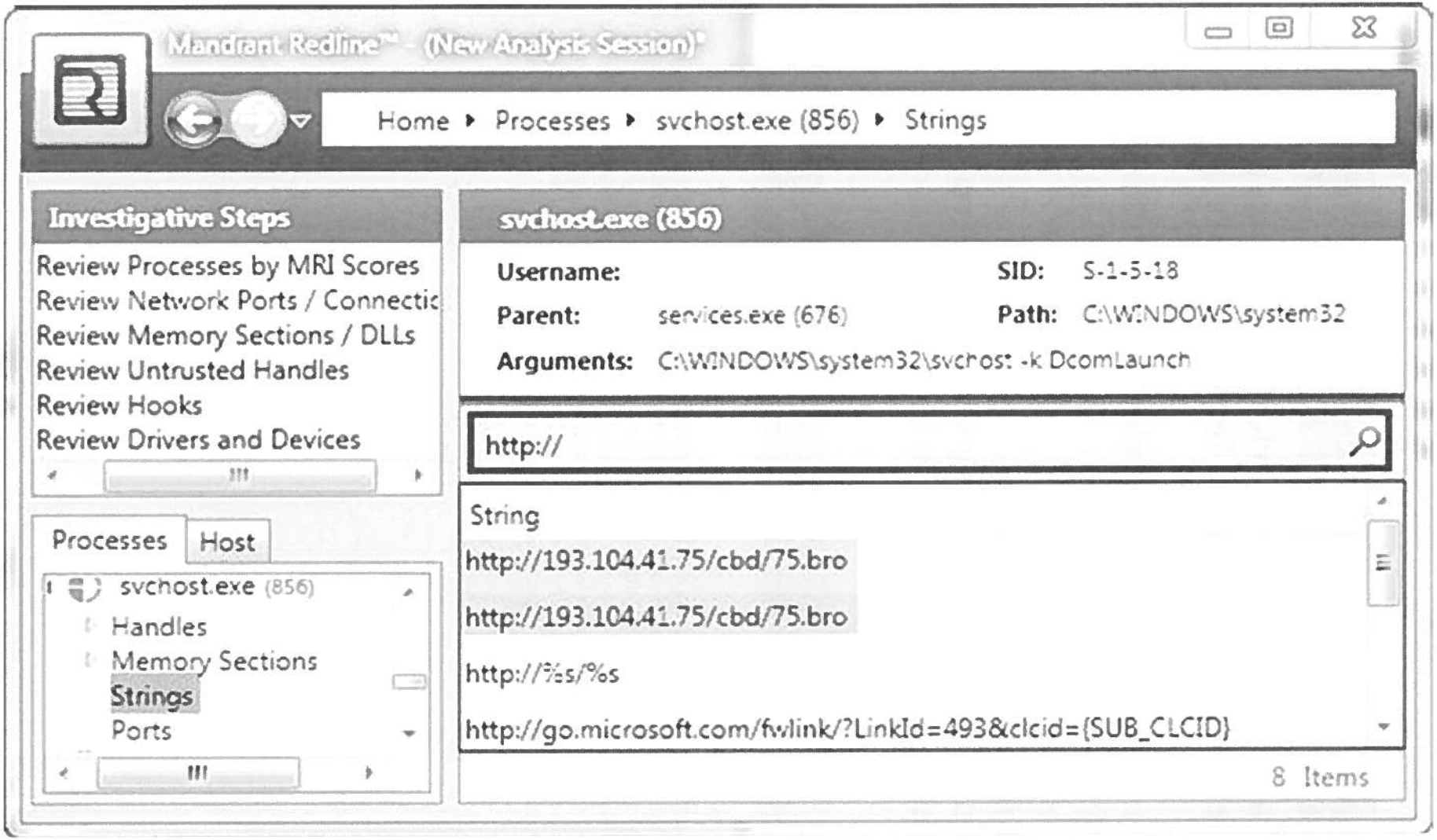


Network Artifacts: Zeus HTTP Strings

Here is where old school memory analysis meets the new school. String searching is still a very viable means to find artifacts of interest in memory. With a memory analysis tool, we can now add some additional intelligence into those searches. Redline has an option to extract the strings from all processes and drivers. The reason why it is an option is that it greatly increases the time necessary to perform the initial pre-processing of the memory image. It is worth it.

If strings have been extracted, you will see them under the Strings label in processes or under the Strings tab within the Device Tree (under the Host View tab). You can manually review all strings, but that can be a very long process. A more expedient means is to use the built-in search dialog, as we see in this slide. Searching for known bad IP addresses, domains or known bad file names are all good ideas. Looking for "http://," "https://," and "ftp://" might help you find output beacon requests, as shown here. In this example, the system was infected with the Zeus bot, which is well known for its outbound connections via port 80. Although no active port 80 connections were found in the network sockets view, the process still had the HTTP requests resident. These in particular stick out because of the request for a resource named "75.bro." A search of the IP addresses shows them as blacklisted for Zeus activity, quickly identifying our malware for us.

The Zeus image used for this example comes from a public sample hosted at https://code.google.com/p/volatility/wiki/SampleMemoryImages. It is provided on your course USB drive.

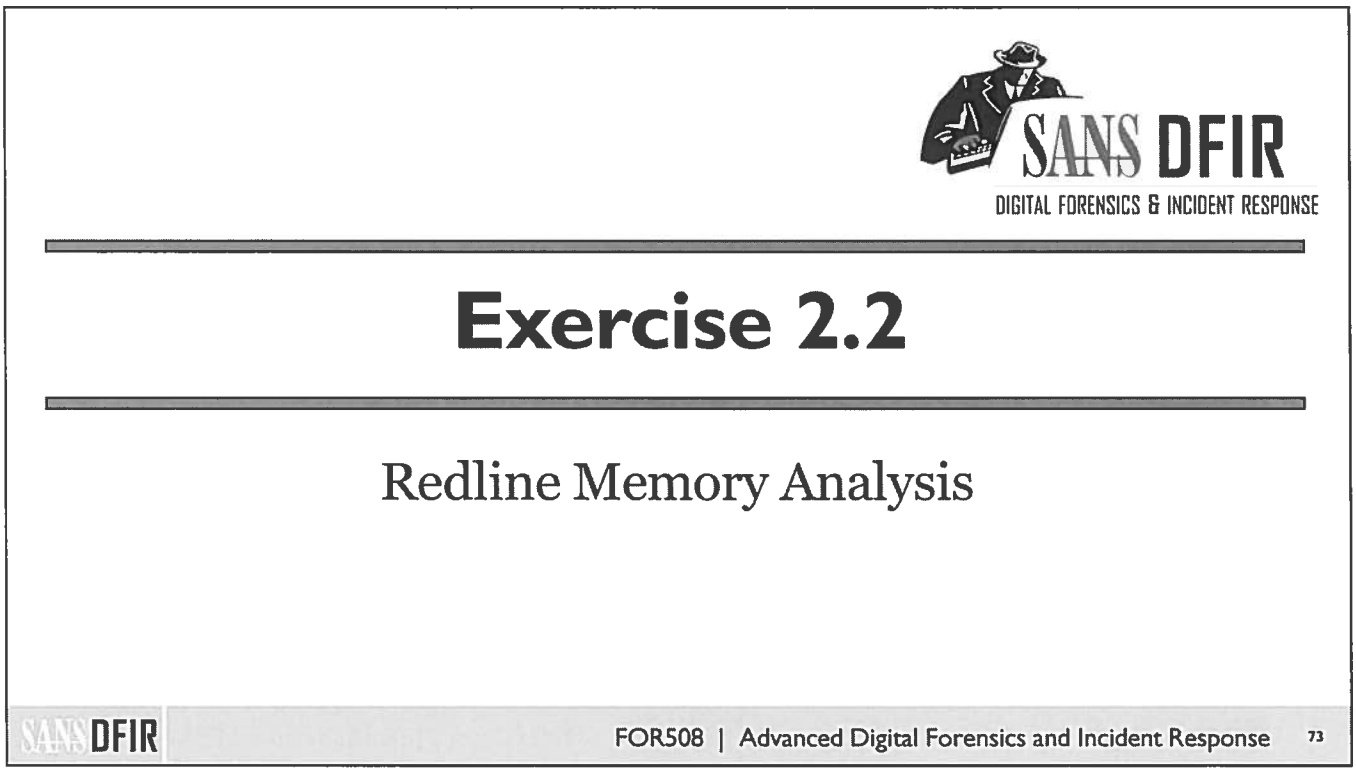

Mandiant Redline™ - (New Analysis Session)*
Home ▸ Processes ▸ svchost.exe (856) ▸ Strings
Investigative Steps
Review Processes by MRI Scores
Review Network Ports / Connectic
Review Memory Sections / DLLs
Review Untrusted Handles
Review Hooks
Review Drivers and Devices
Processes
Host
svchost.exe (856)
Handles
Memory Sections
Strings
Ports
svchost.exe (856)
Username:
SID: S-1-5-18
Parent: services.exe (676)
Path: C:\WINDOWS\system32
Arguments: C:\WINDOWS\system32\svchost -k DcomLaunch
http://
String
http://193.104.41.75/cbd/75.bro
http://193.104.41.75/cbd/75.bro
http://%s/%s
http://go.microsoft.com/fwlink/?LinkId=493&clcid={SUB_CLCID}
8 Items

Network Artifacts: Review

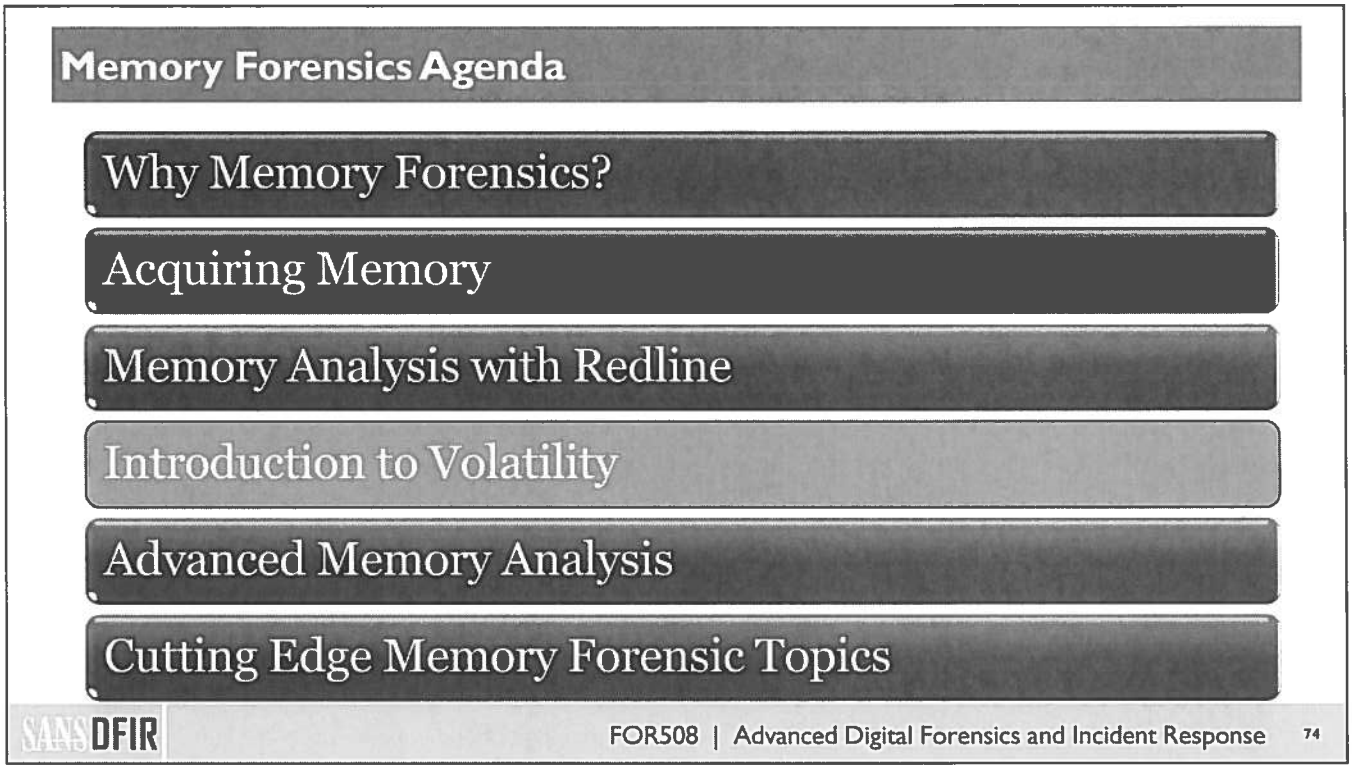
- **When available, network artifacts can provide interesting clues to the legitimacy of a process**
- **When reviewing network data, you should focus on:**
  - Suspicious ports
  - Suspicious connections
  - Known bad IP addresses
  - Suspicious network behavior from processes
  - Interesting creation times of network sockets

Network Artifacts: Review

Checking network connections has long been a staple of incident response, and its usefulness does not fade when we move towards a memory-centric approach. Similar to the malware paradox of wanting to hide, but needing to run, we can add to that the need to communicate. What use is a compromised system if you can't communicate with it? Searching for connections and open sockets can help us identify those malicious connections, even if they occurred long before we acquired memory. If we do find a suspicious connection, we can immediately tie that to a specific process because every socket is owned by a process. Further, we will have a creation time of the socket, allowing us to start performing timeline analysis using other artifacts.



This page intentionally left blank.



This page intentionally left blank.

Introducing Volatility

- Volatility is a framework for performing digital investigations on Windows, Linux, and Mac memory images
- **Volatility supports:**
  - Windows XP Service Pack 1-3 (x86 and x64)
  - Windows 2003 Service Pack 0-2 (x86 and x64)
  - Windows Vista Service Pack 0-2 (x86 and x64)
  - Windows 2008 Service Pack 1-2 (x86 and x64)
  - Windows 2008R2 Service Pack 0-1 (x64)
  - Windows 7 Service Pack 0-1 (x86 and x64)
  - Windows 8 Service Pack 01- (x86 and x64)
  - Windows 2012 (x64)
  - Windows 2012R2 (x64)
  - Windows 10 (x86 and x64)

Introducing Volatility

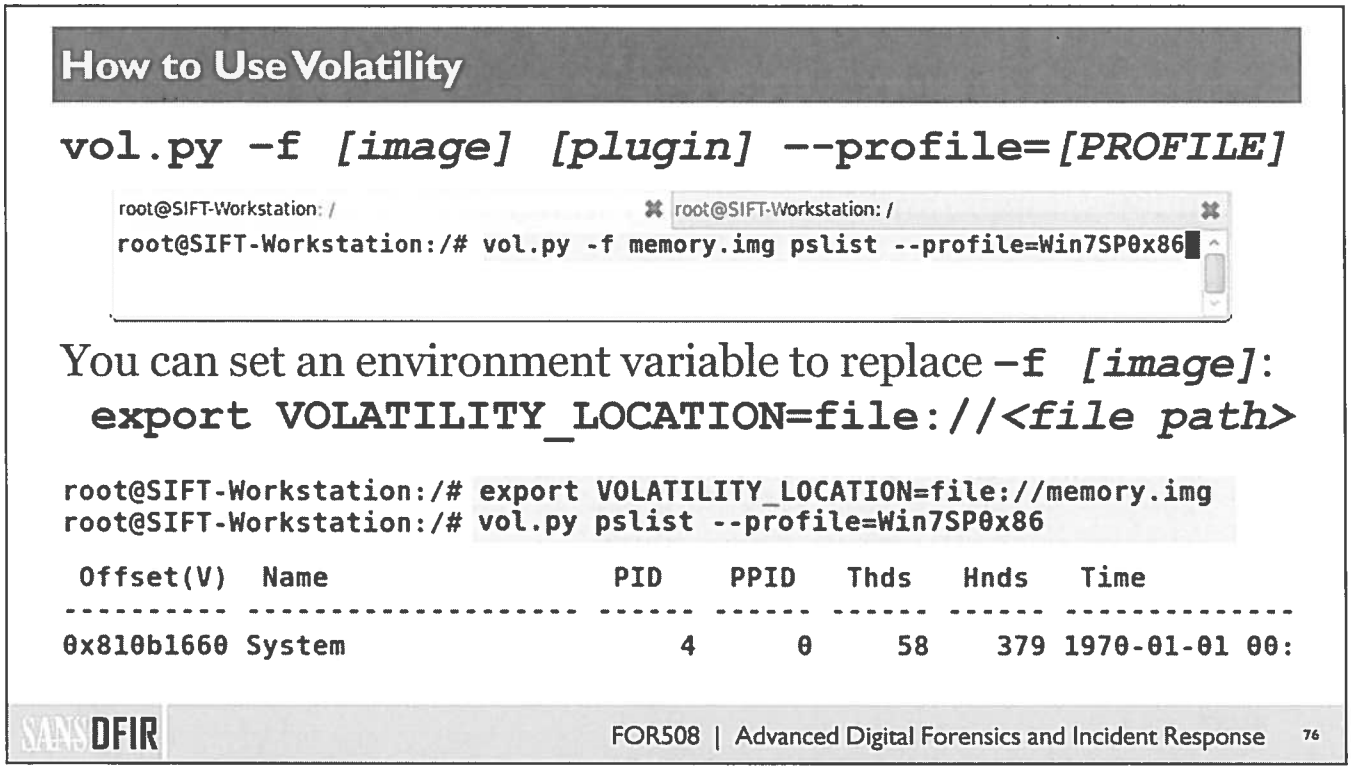
Volatility is the best known memory analysis tool in existence. It has been available for several years and recently has evolved dramatically. Although Volatility has shown great promise over its history, there was one serious thing holding it back. Up until version 2 (originally slated to be version 1.4), Volatility supported analysis only on Windows XP x86, and service packs 2 and 3. Although this was useful as a proof of concept and for memory research, it was severely limiting because a vast number of systems you are likely to encounter don't fit into that category. The release of Volatility version 2 changed that with support for Windows XP, 2003, Vista, 2008, and Windows 7 (32 bit only). Version 2.1 added support for 64-bit systems.[1] The power of Volatility is that it is a memory analysis framework, allowing and encouraging development from multiple entities. It is not nearly as user friendly as Redline, but it is more powerful and has capabilities that Redline does not. And this trend is likely to continue as memory analysis research and plugin writing is crowd-sourced. Although a little challenging to get used to, once you get an understanding of Volatility, you will see how powerful it is as a memory analysis tool.

Volatility is open source and can be retrieved from `https://code.google.com/p/volatility/`

The Volatility wiki keeps a command reference for each Volatility release. As an example, the following URL is valid for Volatility 2.3: `https://code.google.com/p/volatility/wiki/CommandReference23`

Contributors to Volatility include Mike Auty, Andrew Case, Michael Cohen, Brendan Dolan-Gavitt, Jamie Levy, Michael Ligh, AAron Walters, and many others.

** Volatility is a trademark of Verizon. The SANS Institute is not sponsored or approved by, or affiliated with Verizon. **



How to Use Volatility

Volatility is written completely in Python, hence the ".py" extension. It is command-line based and has already been placed in the user path on the SIFT workstation, so you can simply use the **vol.py** command to invoke it.

There are a few options you will use with almost every plugin. First, the "-f" option is required and should be followed with the name and path of the memory image you want to analyze. The [plugin] option tells Volatility what plugin to run and hence what you want it to do. Finally, for any operating system other than Windows XP Service Pack 2, you must specify the operating system version using the "—profile=" parameter. We will talk about how to determine this information on an unknown image shortly.

If you are doing a lot of analysis on the same memory image, you can save a lot of typing by pre-setting your memory image information using an environment variable. The command for doing this is:

```
export VOLATILITY_LOCATION=file://<file path>

(note if you are giving it a full directory path, add the "root"
directory slash -- the command would be:

export VOLATILITY_LOCATION=file:///cases/memory/memory.img )
```

When you are finished with your analysis, you can remove the environment variable by using:

```
unset VOLATILITY_LOCATION
```

When the VOLATILITY_LOCATION variable is set, you can leave off the "-f [image]" parameter on the command line.

If you would like to look at the available plugins, or add more, the plugins for Volatility v2 on the SIFT workstation are located in:

```
/usr/local/src/Volatility/volatility/plugins/
```

Volatility Profiles

Volatility requires the system type for a memory image be specified using the `--profile=[PROFILE]` parameter:

WinXPSP2x86	WinXPSP3x86	WinXPSP1x64	WinXPSP2x64
Win2003SP0x86	Win2003SP1x86	Win2003SP2x86	Win2003SP1x64
Win2003SP2x64	VistaSP0x86	VistaSP1x86	VistaSP2x86
VistaSP0x64	VistaSP1x64	VistaSP2x64	Win2008SP1x86
Win2008SP2x86	Win2008SP1x64	Win2008SP2x64	Win7SP0x86
Win7SP1x86	Win7SP0x64	Win7SP1x64	Win2008R2SP0x64
Win2008R2SP1x64	Win8SP0x86	Win8SP1x86	Win8SP0x64
Win8SP1x64	Win2012x64	Win2012R2x64	Win10x32
Win10x64			

Alternatively, you can set an environment variable and omit `--profile`

```
export VOLATILITY_PROFILE=Win7SP1x64
```


Volatility Profiles

The memory image profile must be correctly specified for Volatility to run correctly. Like many tools, the error messaging in Volatility needs work, and often you might not realize what you are doing wrong until you remember to include the "--profile" parameter. Listed on this slide are the currently supported profiles as of Volatility v2.14.

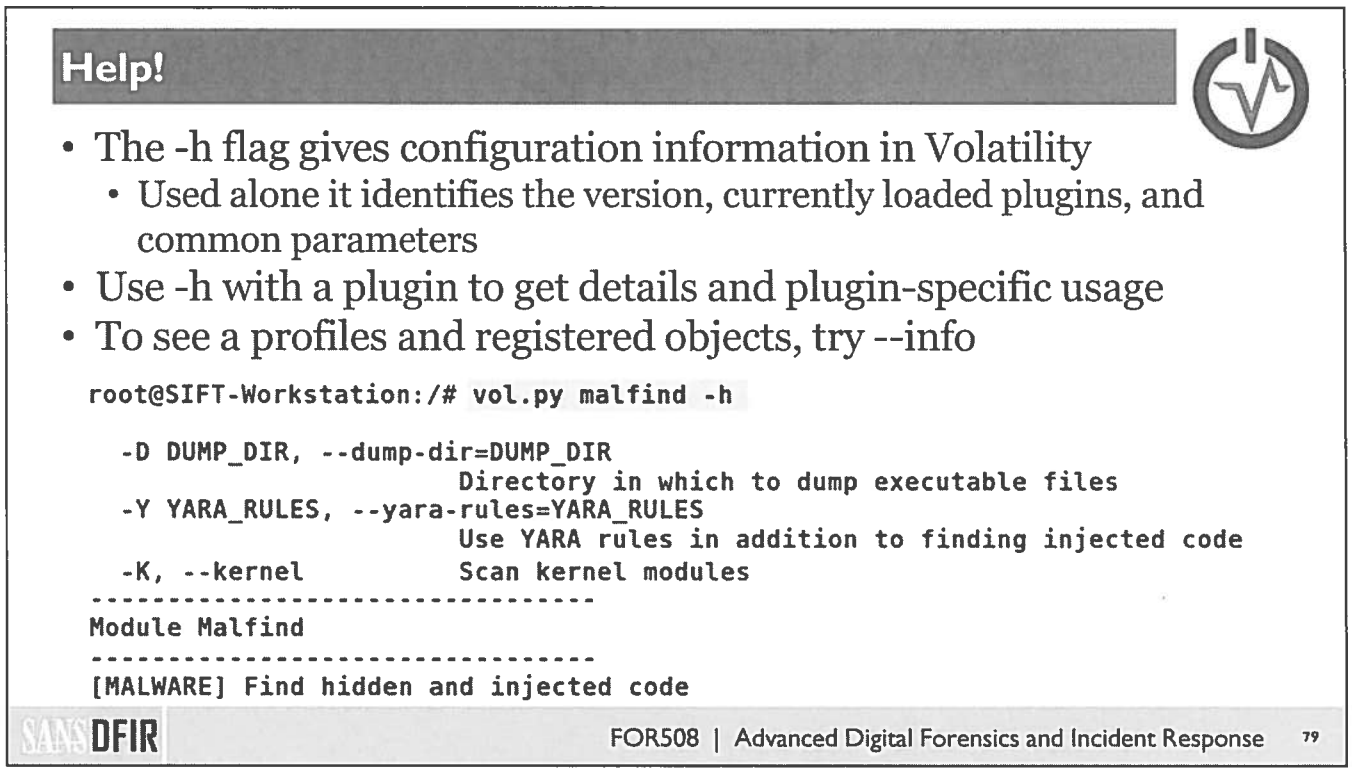
Similar to how we were able to create an environment variable to hold the memory image filename, we can do the same for the profile of the image we are working on. This syntax is:

```
export VOLATILITY_PROFILE=Win7SP0x86
```

and you can remove this variable with:

```
unset VOLATILITY_PROFILE
```

If you do not know what kind of operating system the memory image came from, there is a plugin named **`imageinfo`** that will provide this information.



Help!

It is easy to get lost when you are first using Volatility, but luckily the amount of documentation on the project has increased greatly. If you are stuck, try the "-h" option. This can be used without any plugins specified to get general information about Volatility and the currently loaded plugins.

root@SIFT-Workstation:/# **vol.py -h**

Usage: Volatility - A memory forensics analysis platform.

Options:

-h, --help	List all available options and their default values Default values might be set in the configuration file (/etc/volatilityrc)
--conf-file=/root/.volatilityrc	User-based configuration file
-d, --debug	Debug volatility
--plugins=PLUGINS	Additional plugin directories to use (colon separated)
--info	Print information about all registered objects
--cache-directory=/root/.cache/volatility	Directory where cache files are stored
--cache	Use caching
--tz=TZ	Sets the time zone for displaying timestamps
-W, --warnings	Disable warning messages

-f FILENAME, --filename=FILENAME
Filename to use when opening an image
--profile=WinXPSP2x86
Name of the profile to load
-l LOCATION, --location=LOCATION
A URN location from which to load an address space
-w, --write Enable write support
--dtb=DTB DTB Address
--output=text
Output in this format (format support is module specific)
--output-file=OUTPUT_FILE
write output in this file
-v, --verbose Verbose information
--shift=SHIFT Mac KASLR shift address
-g KDBG, --kdbg=KDBG
Specify a specific KDBG virtual address
-k KPCR, --kpcr=KPCR
Specify a specific KPCR address

Supported Plugin Commands:

apihooks	Detects API hooks in process and kernel memory
atoms	Prints session and window station atom tables
atomscan	Pools scanner for _RTL_ATOM_TABLE
bioskbd	Reads the keyboard buffer from Real Mode memory
callbacks	Prints system-wide notification routines
clipboard	Extracts the contents of the windows clipboard
cmdscan	Extracts command history by scanning for _COMMAND_HISTORY
connections	Prints list of open connections [Windows XP and 2003 Only]
connscan	Scans Physical memory for _TCPT_OBJECT objects (tcp connections)
consoles	Extracts command history by scanning for _CONSOLE_INFORMATION
crashinfo	Dumps crash-dump information
datetime	A simple example plugin that gets the date/time information from a Windows image
deskscan	Poolscanner for tagDESKTOP (desktops)
devicetree	Shows device tree
dlldump	Dumps DLLs from a process address space
dlllist	Prints list of loaded dlls for each process
driverirp	Driver IRP hook detection
driverscan	Scans for driver objects _DRIVER_OBJECT
dumpcerts	Dumps RSA private and public SSL keys
dumpfiles	Extracts memory mapped and cached files

enumfunc	Enumerates imported/exported functions
envars	Displays process environment variables
eventhooks	Prints details on windows event hooks
evtlogs	Extracts Windows Event Logs (XP/2003 only)
filescan	Scans Physical memory for _FILE_OBJECT pool allocations
gahti	Dumps the USER handle type information
gditimers	Prints installed GDI timers and callbacks
gdt	Displays Global Descriptor Table
getservicesids	Gets the names of services in the Registry and return Calculated SID
getsids	Prints the SIDs owning each process
handles	Prints list of open handles for each process
hashdump	Dumps password hashes (LM/NTLM) from memory
hibinfo	Dumps hibernation file information
hivedump	Prints out a hive
hivelist	Prints list of registry hives.
hivescan	Scans Physical memory for _CMHIVE objects (registry hives)
hpakextract	Extracts physical memory from an HPAK file
hpakinfo	Info on an HPAK file
idt	Displays Interrupt Descriptor Table
iehistory	Reconstructs Internet Explorer cache/history
imagecopy	Copies a physical address space out as a raw DD image
imageinfo	Identifies information for the image
impscan	Scans for calls to imported functions
kdbgscan	Searches for and dumps potential KDBG values
kpcrscan	Searches for and dumps potential KPCR values
ldrmodules	Detects unlinked DLLs
lsadump	Dumps (decrypted) LSA secrets from the registry
machoinfo	Dumps Mach-O file format information
malfind	Finds hidden and injected code
malsysproc	Finds malware hiding in plain sight as system processes
mbrparser	Scans for and parses potential Master Boot Records (MBRs)
memdump	Dumps the addressable memory for a process
memmap	Prints the memory map
messagehooks	Lists desktop and threads window message hooks
mftparser	Scans for and parses potential MFT entries
mimikatz	mimikatz offline
moddump	Dumps a kernel driver to an executable file sample
modscan	Scans Physical memory for _LDR_DATA_TABLE_ENTRY objects
modules	Prints list of loaded modules

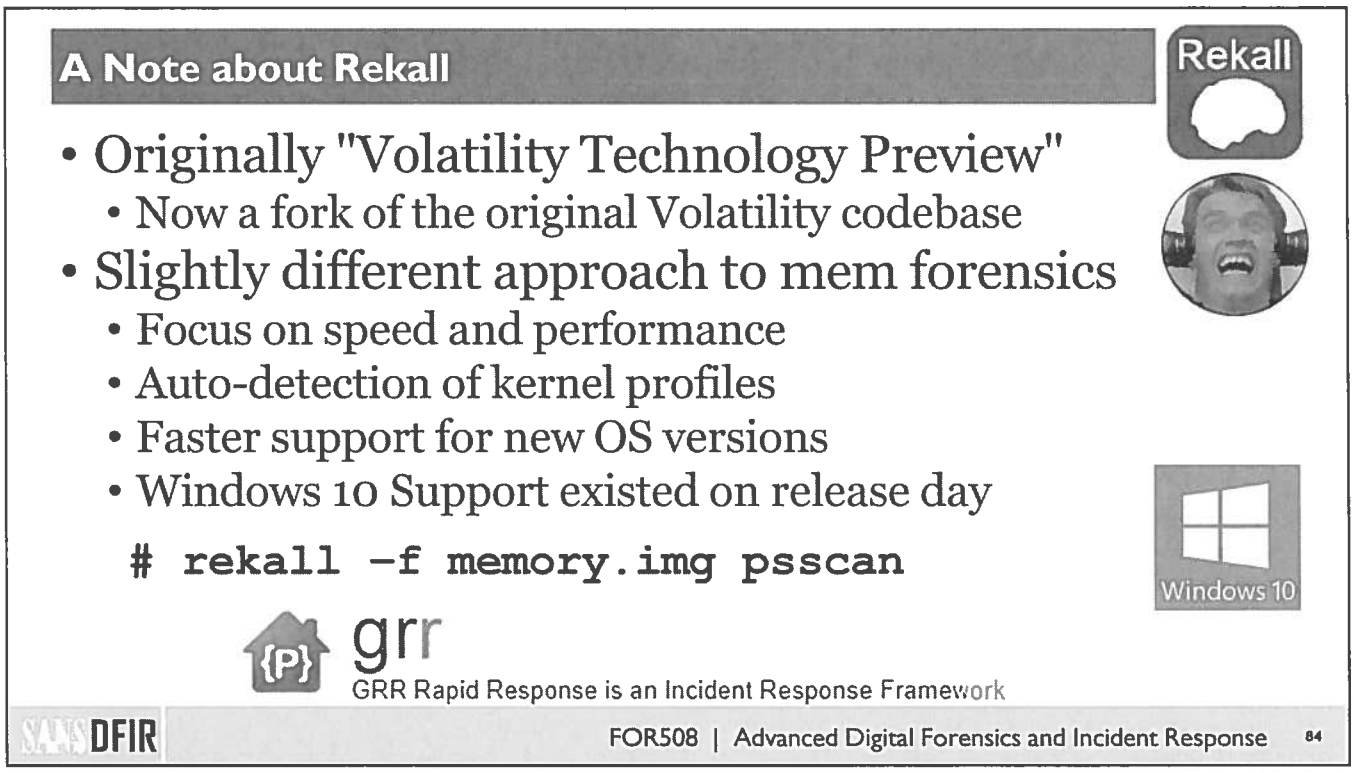
mutantscan	Scans for mutant objects _KMUTANT
pagecheck	Reads the available pages and reports if any are inaccessible
patcher	Patches memory based on page scans
prefetchparser	Scans for and parses potential Prefetch files
printkey	Prints a registry key, and its subkeys and values
privs	Displays process privileges
procexedump	Dumps a process to an executable file sample
procmemdump	Dumps a process to an executable memory sample
psdispscan	Scans Physical memory for _EPROCESS objects based on their Dispatch Headers
pslist	Prints all running processes by following the EPROCESS lists
psscan	Scans Physical memory for _EPROCESS pool allocations
pstotal representation	Combination of pslist, psscan, and pstree --output=dot gives graphical
pstree	Prints process list as a tree
psxview	Finds hidden processes with various process listings
raw2dmp	Converts a physical memory sample to a windbg crash dump
screenshot	Saves a pseudo-screenshot based on GDI windows
sessions	Lists details on _MM_SESSION_SPACE (user logon sessions)
shellbags	Prints ShellBags info
shimcache	Parses the Application Compatibility Shim Cache registry key
sockets	Prints list of open sockets
sockscan	Scans Physical memory for _ADDRESS_OBJECT objects (tcp sockets)
ssdt	Displays SSDT entries
strings verbose)	Matches physical offsets to virtual addresses (might take awhile, VERY
svcscan	Scans for Windows services
symlinkscan	Scans for symbolic link objects
thrdscan	Scans physical memory for _ETHREAD objects
threads	Investigates _ETHREAD and _KTHREADs
timeliner	Creates a timeline from various artifacts in memory
timers	Prints kernel timers and associated module DPCs
uninstallinfo	Extracts installed software info from Uninstall registry key
unloadedmodules	Prints list of unloaded modules
userassist	Prints userassist registry keys and information
userhandles	Dumps the USER handle tables
usnparser	Scans for and parses USN journal records
vaddump	Dumps out the vad sections to a file
vadinfo	Dumps the VAD info
vadtree	Walks the VAD tree and displays in tree format
vadwalk	Walks the VAD tree
vboxinfo	Dumps virtualbox information

verinfo	Prints out the version information from PE images
vmwareinfo	Dumps VMware VMSS/VMSN information
volshell	Shells in the memory image
windows	Prints Desktop Windows (verbose details)
wintree	Prints Z-Order Desktop Windows Tree
wndscan	Pools scanner for tagWINDOWSTATION (window stations)
yarascan	Scans process or kernel memory with Yara signatures

When the "-h" option is used in conjunction with a plugin, it will give you a brief description of the plugin and the identify available options (as seen on the slide). Keep in mind that the options can be misleading. Volatility is an open framework and some of the available plugins inherit options that don't actually work. If you are frustrated with an option, do some additional research, because the help screen could be misleading. Throughout this training, we have done our best to provide the most up-to-date list of useful options for each plugin.

A good first step when learning Volatility is the project wiki.[1] It is a one-stop shop for all things Volatility, containing a command reference with examples, documentation links, sample images to work with, and even an archive of the current bugs and issues being worked on.

[1] http://code.google.com/p/volatility/wiki/BasicUsage



A Note about Rekall

Rekall is led by Michael Cohen of Volatility Project.[2] Mr. Cohen, or "scudette," was a primary contributor to Volatility and started a Technology Preview branch to research and develop some of his interests. When it was clear that this branch would not be included in the main Volatility trunk, Cohen forked the project and named it "Rekall." Rekall still relies on a large portion of the Volatility codebase and has been integrated into the open-source GRR (Google Rapid Response) project allowing memory forensics via agent-based deployments.[2] To better support GRR, the developers wanted to make some fundamental changes to the code, including making it easier to use as a library, speeding up initial analysis by stopping KDBG scans, and better kernel-specific profiles that can be readily shared. The latter innovation is one of the most readily apparent to previous users of Volatility. The Rekall team removed OS profiles from the codebase and replaced them with a lighter weight and more kernel specific data file.[3] These files are served up from an online repository and can represent the 200+ different kernel versions in modern Windows versions. This new profile management allows easier auto-detection of profiles (no need for the --profile= argument!) and faster support for new OS releases. As a proof of concept, Rekall supported Windows 8 memory analysis over six months prior to the Volatility release.

Rekall also was the first to support Windows 10 and had initial capability existing on the first day of Win10 release.[4]

The Rekall project supports most modern Windows, Linux, and OS X kernels. It also encompasses the winpmem and osxpem acquisition and live analysis tools.

The Rekall project is an exciting new chapter in the history of memory forensics, and it will be interesting to see where it ends up. A recent version is present on your SIFT workstation. If you would like to start playing with the tool, try **`rekall -h`** or **`rekall -f memory.img`** <volatility plugin>.

[1] https://code.google.com/p/rekall/

[2] https://code.google.com/p/grr/

[3] http://www.rekall-forensic.com/blog.html

[4] http://rekall-forensic.blogspot.com/2015/06/adding-rekalls-windows-10-support.html

`

Memory Image Tools

This page intentionally left blank.

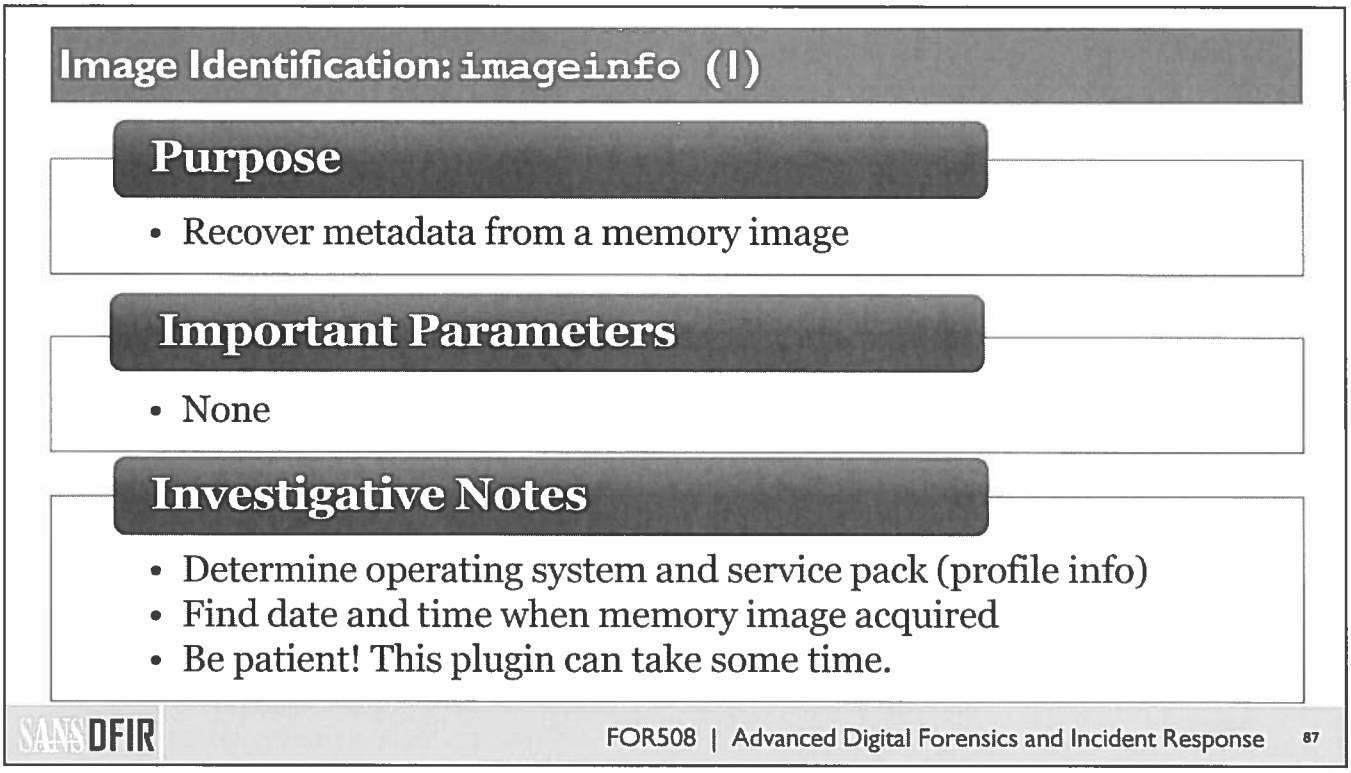


Image Identification: `imageinfo` (1)

The **`imageinfo`** plugin is the first plugin you should run when starting any examination using Volatility. It interrogates the memory image and returns information such as the system date and time when the memory image was acquired, the Kernel Processor Control Region (KPCR), or starting point for identification of structures in memory, system information like number of processors, and most importantly, the operating system time and service pack information. This information is so important because this is the "--profile=" value that you will include when using the rest of the Volatility plugins for your analysis.

Remember to be patient with this plugin! Most Volatility plugins give immediate results, so sometimes it is hard to wait for those that do not. In some cases, the plugin must do a lot of searching to find items like the KDBG and KPCR structures to identify the proper operating system type. This plugin has no important parameters.

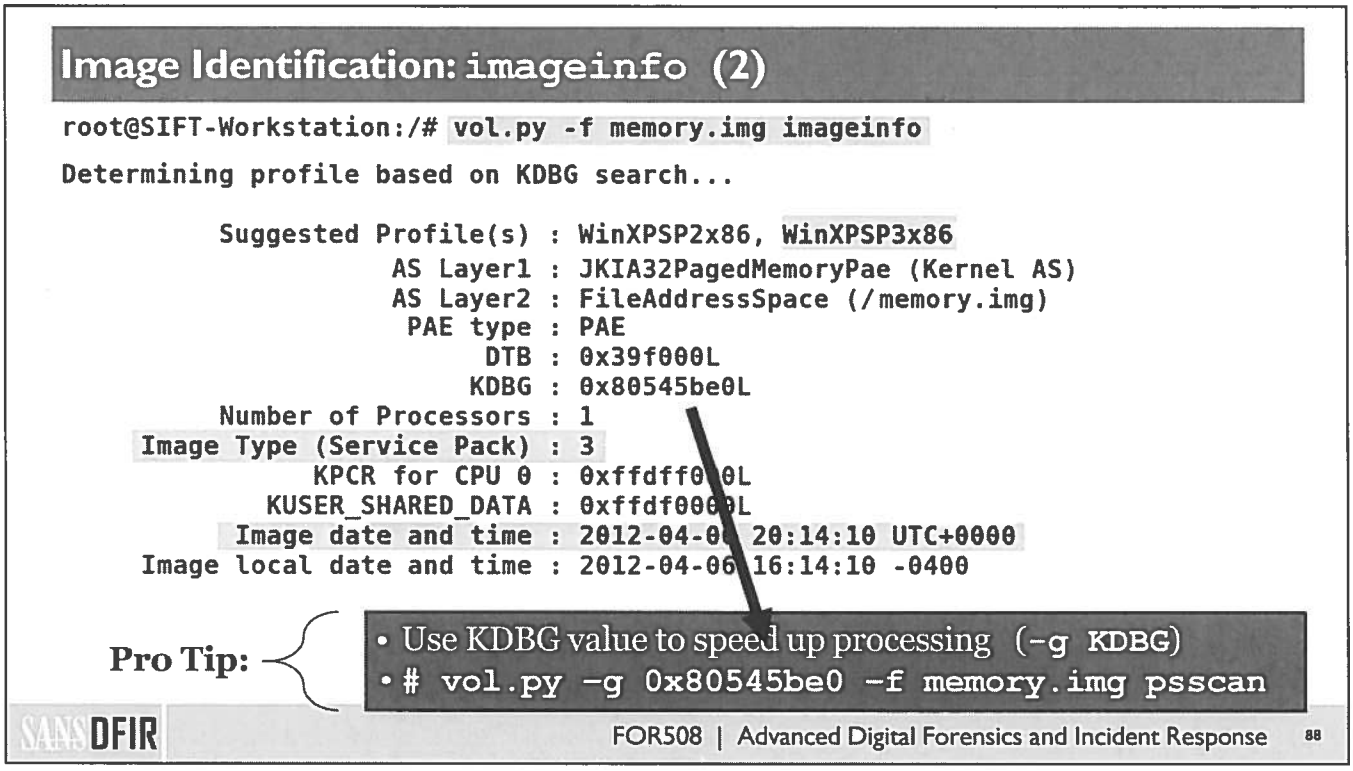


Image Identification: `imageinfo` (2)

Imageinfo might not be too exciting, but it does provide some information that can be very useful. Knowing the system time upon acquisition can help you put other artifact times in perspective. The suggested profile(s) is also of critical importance. This is the value you will provide via the "profile=" option to each subsequent plugin, either explicitly through the command line or by setting the **VOLATILITY_PROFILE** environment variable.

Example command line:

root@SIFT-Workstation:/# **vol.py -f memory.img imageinfo**

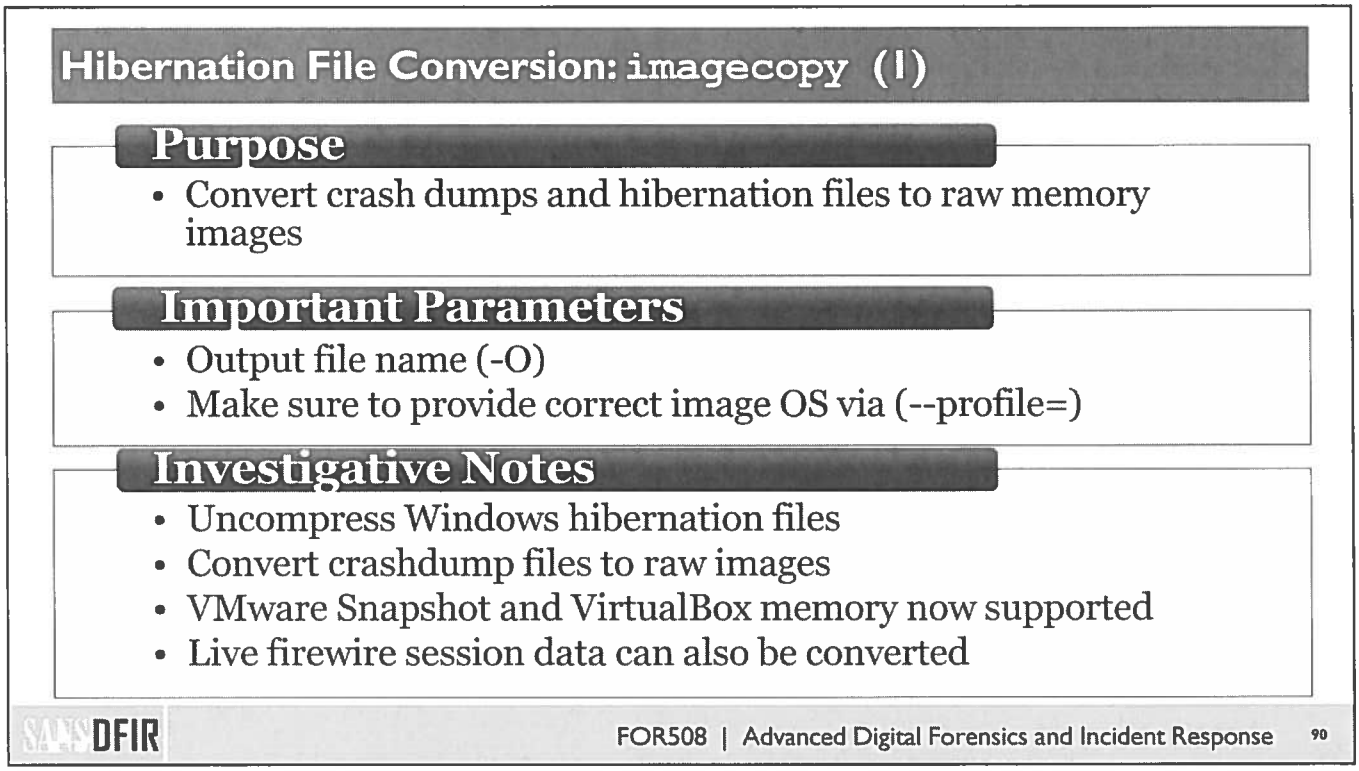
Determining profile based on KDBG search...

```
        Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86
                    AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                    AS Layer2 : FileAddressSpace (/memory.img)
                     PAE type : PAE
                          DTB : 0x39f000L
                         KDBG : 0x80545be0L
       Number of Processors : 1
  Image Type (Service Pack) : 3
             KPCR for CPU 0 : 0xffdff000L
          KUSER_SHARED_DATA : 0xffdf0000L
        Image date and time : 2012-04-06 20:14:10 UTC+0000
  Image local date and time : 2012-04-06 16:14:10 -0400
```

Pro Tip: Use the KDBG value to speed up processing speed (**`-g KDBG`**). When used, Volatility will not need to search for that location in many of the plugins that rely upon it. In Windows 8 and 2012, the value should be the virtual address of the KdCopyDataBlock (found via the **`kdbgscan`** plugin):

```
# vol.py -g  0x80545be0 -f memory.img psscan
```

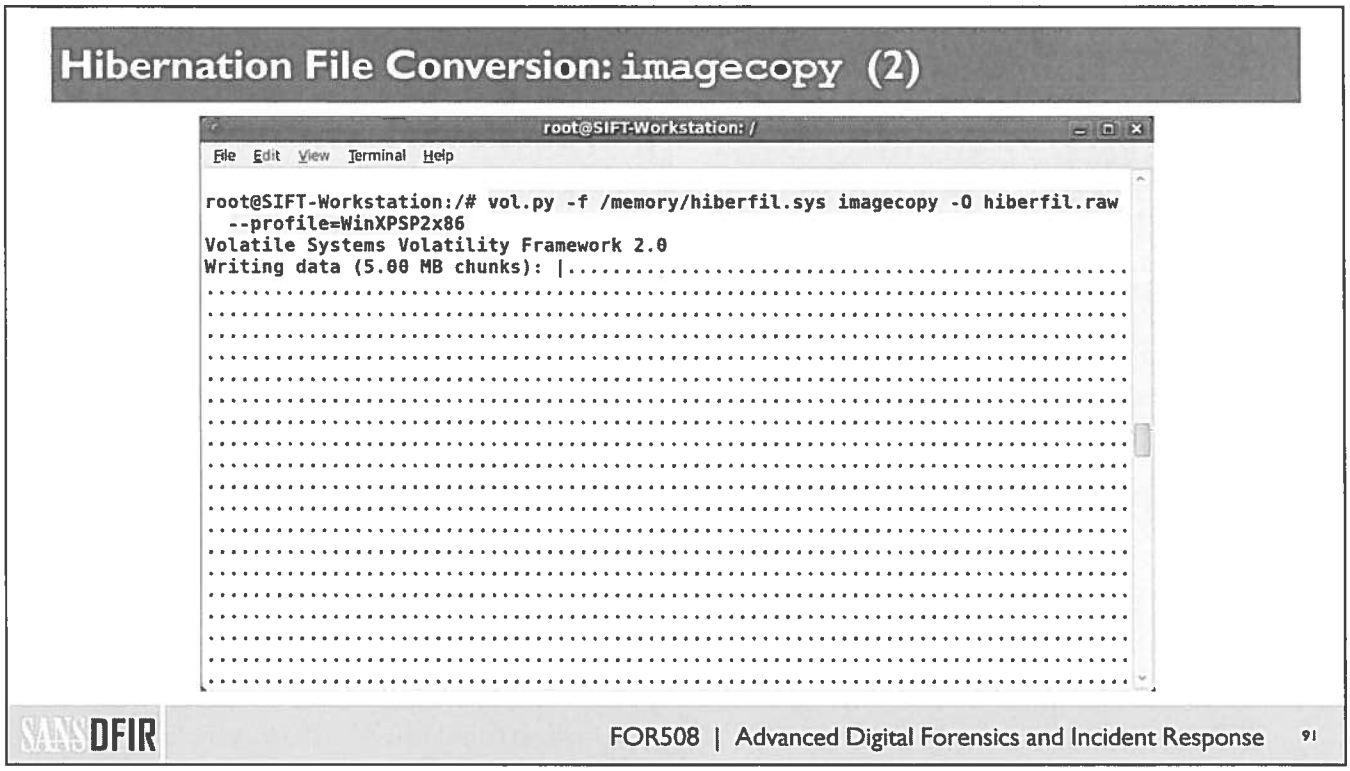
Another good tip comes in handy if you have a memory image that does not seem to work with plugins, such as **`pslist`**. If you can get **`imageinfo`** data, try to pass the Directory Table Base (DTB) value as an optional parameter. In the example on this slide, you would add **`--dtb 0x39f000`** to any Volatility command line.



Hibernation File Conversion: `imagecopy` (1)

Our first Volatility plugin is used to prepare non-standard memory images for analysis. Although we have largely talked about raw memory images acquired using tools like win32/64dd, there are other formats we might encounter. Volatility will attempt to auto-detect these other formats and read them on the fly, but in some situations Volatility will fail or perhaps you want a hibernation file converted permanently to a raw image for use in other applications like Redline or using string based searching (recall that Windows hibernation files are often compressed). If that is your aim, **`imagecopy`** is the plugin you want to use. It has only one option "-O," providing the output file name.

Although this is a wonderful capability, note that it is not perfect. We have encountered multiple hibernation images that were not able to be successfully converted (or recognized) by Volatility. Remember the previous commercial tool covered, hibr2bin.exe, as it can be used as a potential backup.



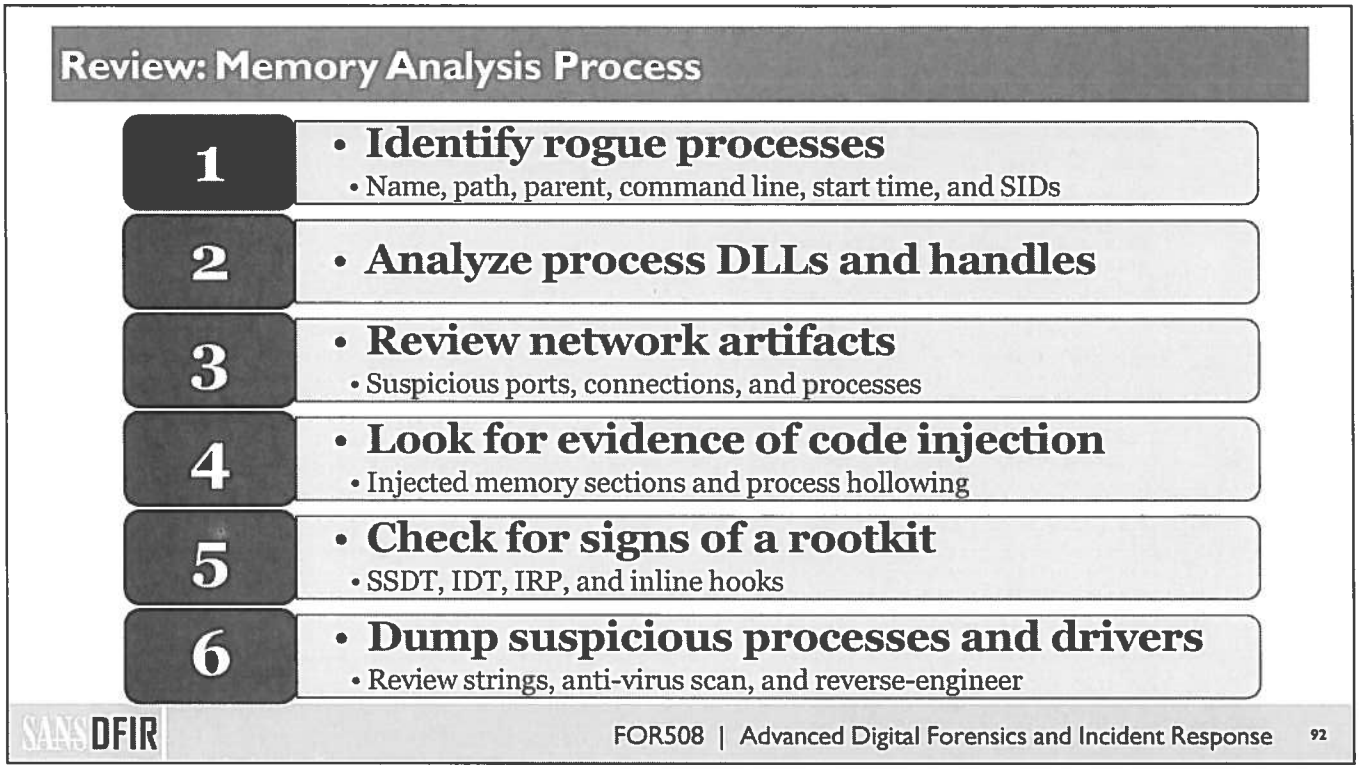
Hibernation File Conversion: `imagecopy` (2)

The imagecopy plugin is one of the simpler to use within the Volatility framework. You simply specify the plugin name, file to convert using the "-f" option, and the output file name using "-O," and wait for completion.

Example:

root@SIFT-Workstation:/# **vol.py -f /memory/hiberfil.sys imagecopy -O hiberfil.raw**
--profile=WinXPSP2x86

Writing data (5.00 MB chunks): |.....



Review: Memory Analysis Process

By this point in the class, we have already covered each of these six steps illustrating with Redline. When using Volatility, our process will be the same, but we will use a completely different toolkit. You will notice that some items are easier to review in Redline (like process objects), whereas others are more information rich in Volatility (like code injection). Both tools have their benefits and we recommend using both to check one another and provide the best of both worlds when doing your memory analysis.

In the next sections, we will follow these six steps illustrating the Volatility framework.

Step 1:

Identify Rogue Processes

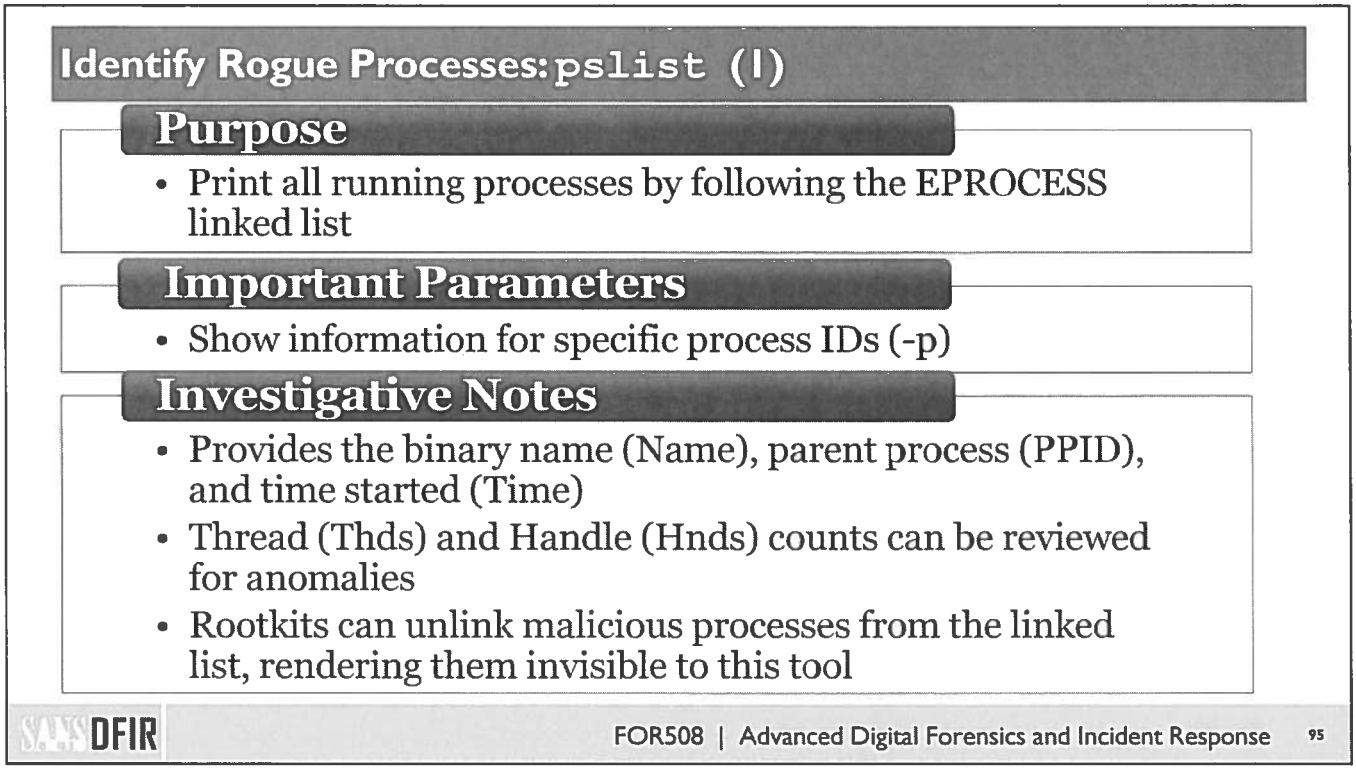
This page intentionally left blank.

Identify Rogue Processes: Plugins

pslist	Print all running processes within the EPROCESS doubly linked list
psscan	Scan physical memory for EPROCESS pool allocations
pstree	Print process list as a tree showing parent relationships (using EPROCESS linked list)
pstotal	Comparison of `psscan` and `pslist` results. Also produces output in graphics format.

Identify Rogue Processes: Plugins

Processes make up the major building blocks of a memory image and are a logical place to start our memory analysis. Unlike the pre-processing time required in Redline, Volatility takes a different approach. For each item or action we want to examine and test, there will be a specialized plugin. The plugins listed here help us analyze information in the first step of our analysis checklist: Identify Rogue Processes. Each one provides slightly different information.



Identify Rogue Processes: `pslist` (1)

Identifying the running processes is one of the first steps you will regularly perform when examining memory. The **`pslist`** plugin is a fast and easy means to get this information. The plugin works by finding the EPROCESS list in the kernel, following the doubly linked list, and parsing information about each process. For each found process, the plugin provides:

- **Virtual offset of EPROCESS block**
- **Process name**
- **Process Identifier (PID)**
- **Parent Process Identifier (PPID)**
- **Number of threads**
- **Number of handles**
- **Process start time**

Each of these process attributes should be reviewed for anomalies. You might notice Volatility provides a bit more information than Redline, namely the thread and handle count for each process. This information is probably overkill for most of our memory analysis purposes, but there might be times where comparisons with like processes lead you to believe one is anomalous. Also, keep in mind that a terminated process should have zero handles and zero threads. If you see such a process in your listing, it probably means the process has ended and was not yet moved out of the doubly linked list.

Keep in mind that the **`pslist`** plugin is the most basic of all of the process plugins. It might not find processes hidden by rootkits or other means.

Identify Rogue Processes: `pslist` (2)

`winppr32.exe` (Sobig worm) was started much later, indicating it was not part of the boot cycle

```
root@SIFT-Workstation:/# vol.py -f /memory/sobig.img pslist

 Offset(V)  Name                 PID    PPID   Thds   Hnds   Time
---------- -------------------- ------ ------ ------ ------ -----------------
0x823c89c8 System                 4      0     50    259 1970-01-01 00:00:00
0x81e02da0 smss.exe             524      4      3     19 2009-07-20 23:43:26
0x82273020 csrss.exe            580    524     11    362 2009-07-20 23:43:31
0x82147cf0 winlogon.exe         604    524     16    429 2009-07-20 23:43:33
0x81dd6c70 services.exe         648    604     16    254 2009-07-20 23:43:36
0x81dd53a0 lsass.exe            660    604     21    334 2009-07-20 23:43:36
0x8212f658 svchost.exe          944    648     11    233 2009-07-20 23:43:41
                                                          :42
 winppr32.exe  1624   1636  2  55 2009-07-27 23:27:44     :42
                                                          :48
0x81d6e798 alg.exe              436    648      6    106 2009-07-20 23:44:18
0x81db0530 msiexec.exe          708    648      8    226 2009-07-27 20:24:43
0x82128020 cmd.exe             1236   1560      1     33 2009-07-27 20:25:58
           6020 winppr32.exe    1624   1636      2     55 2009-07-27 23:27:44
```


Identify Rogue Processes: `pslist` (2)

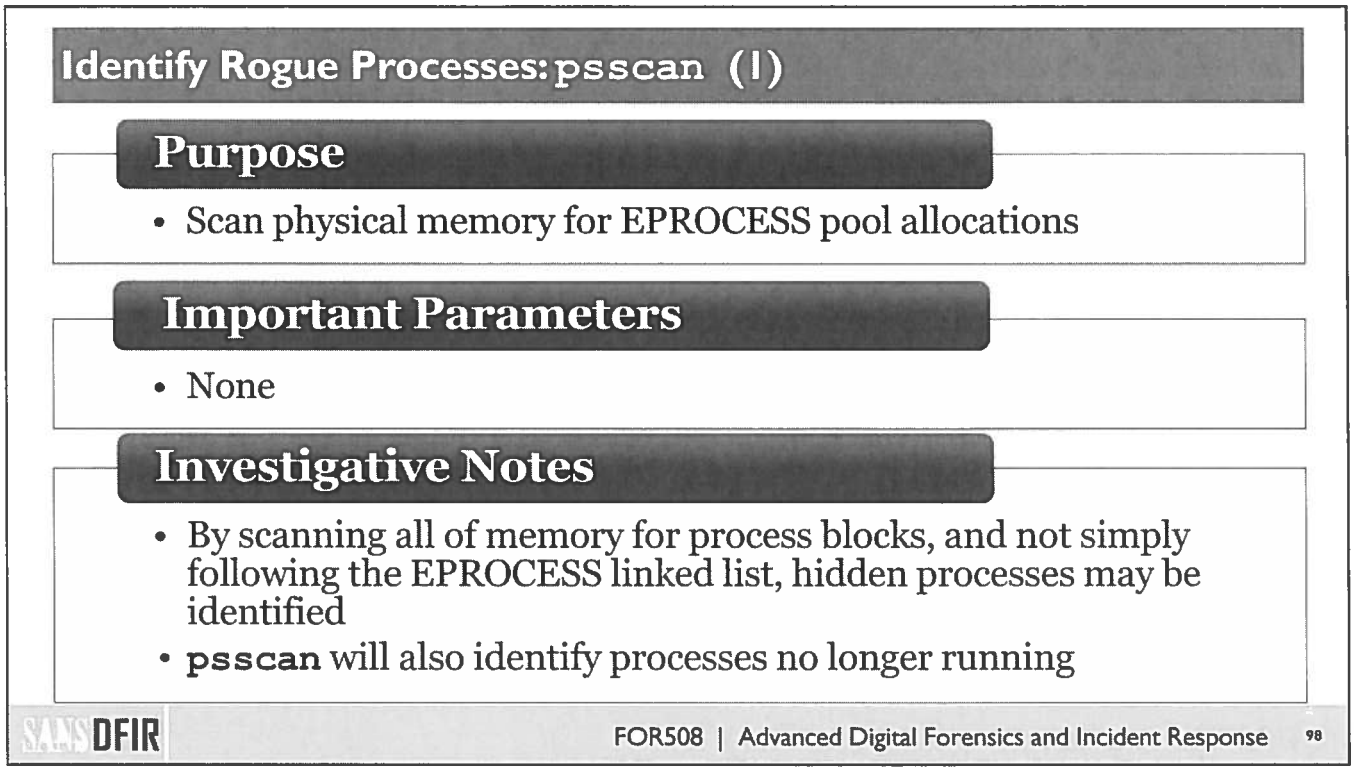
When analyzing a memory image infected with the Sobig worm, we first run **`pslist`** to see whether anything anomalous could be identified. A review of running processes by an experienced incident responder might lead to winppr32.exe being judged to be suspicious. Although it has a legitimate sounding name, you likely have never seen such a process running on other systems (if you have, they were probably infected, too). A less experienced investigator might need to compare this process listing with a similar but uninfected test machine, or perform Internet searches on any unfamiliar process names.

Looking a little closer at winppr32.exe, we see its parent process (PID 1636) is not present in the process list. That likely means that it is terminated (which is the case here), but it could also mean that the process is hidden. This is not enough information to definitely assume this process is suspicious, but it is a data point because very few system processes have a terminated parent process. A review of the process start time column shows the process was started almost a week later than the standard boot processes. This is an additional clue that this might not be a real system process even though its name might suggest it. The start time also tells us that the process was likely not loaded with a persistence mechanism, at least not one that occurs upon system boot.

Command line example:

root@SIFT-Workstation:/# **vol.py -f /memory/sobig.img pslist**

Offset(V)	Name	PID	PPID	Thds	Hnds	Time
0x823c89c8	System 0:00:00	4	0	50	259	1/1/1970
0x81e02da0	smss.exe 23:43:26	524	4	3	19	7/20/2009
0x82273020	csrss.exe 23:43:31	580	524	11	362	7/20/2009
0x82147cf0	winlogon.exe 23:43:33	604	524	16	429	7/20/2009
0x81dd6c70	services.exe 23:43:36	648	604	16	254	7/20/2009
0x81dd53a0	lsass.exe 23:43:36	660	604	21	334	7/20/2009
0x81fb5a28	vmacthlp.exe 23:43:38	812	648	1	25	7/20/2009
0x821331f8	svchost.exe 23:43:40	856	648	18	191	7/20/2009
0x8212f658	svchost.exe 23:43:41	944	648	11	233	7/20/2009
0x81db9020	svchost.exe 23:43:42	1036	648	60	1126	7/20/2009
0x81db5a58	svchost.exe 23:43:42	1128	648	12	120	7/20/2009
0x81dba768	svchost.exe 23:43:44	1268	648	14	188	7/20/2009
0x81f8d020	spoolsv.exe 23:43:47	1468	648	13	120	7/20/2009
0x81f8b540	explorer.exe 23:43:48	1560	1544	17	535	7/20/2009
0x81d6e798	alg.exe 23:44:18	436	648	6	106	7/20/2009
…SNIP…						
0x81db0530	msiexec.exe 20:24:43	708	648	8	226	7/27/2009
0x82128020	cmd.exe 20:25:58	1236	1560	1	33	7/27/2009
0x81f56020	winppr32.exe 23:27:44	1624	1636	2	55	7/27/2009
…SNIP…						



The **psscan** plugin collects process data in a very different way than the **pslist** plugin. Notice the word "scan" as part of its name. This is an interesting distinction that we will see often in Volatility. Scanning plugins do not follow the normal set of procedures to identify their objects, such as following the doubly linked list of all processes. Instead, it scans all of the memory looking for its objects, similar to performing a data carve on a disk. Thus, it provides an interesting counter-point to information provided with other plugins that can enhance our analysis. In the case of **psscan**, it has the capability to identify terminated processes within the "unallocated" part of memory as well as a better ability to find processes hiding due to rootkit techniques. The scanning process causes this plugin to be slower than its **pslist** analog. The plugin returns the following information:

- **Physical Offset of EPROCESS block**
- **Process name**
- **Process Identifier (PID)**
- **Parent Process Identifier (PPID)**
- **Page directory base offset (PDB)**
- **Process start time**
- **Process exit time**

We get a little different information with **psscan** than we did with **pslist**. One addition is the page directory base offset (PDB). The PDB allows the virtual address space for a process to be converted to its physical RAM addresses—not incredibly useful for our standard analysis purposes. The process exit time addition does give us a little more information; if you see this field populated, it means you are looking at a terminated process. That being said, you might encounter terminated processes that have not yet been updated with an exit time.

It is worth spending a moment talking about the differences between physical memory offsets and virtual memory offsets. Physical offsets are the actual location of the entity (EPROCESS block, in this case) in the memory image. You could go to that offset in a hex editor and find that object. Virtual memory offsets (like what the **`pslist`** plugin provides by default) are what are actually used by the running system. They are the location of the object within Virtual memory—meaning physical memory plus all of its virtual pages. A 32-bit virtual address can be broken down as 10 bits for the page directory index, 10 bits for the page table index, and 12 bits for the byte index. The reason why this all matters is that some Volatility plugin options require either a physical or virtual address. If you do not provide the correct one, you will not receive the correct results.

Identify Rogue Processes: psscan (2)

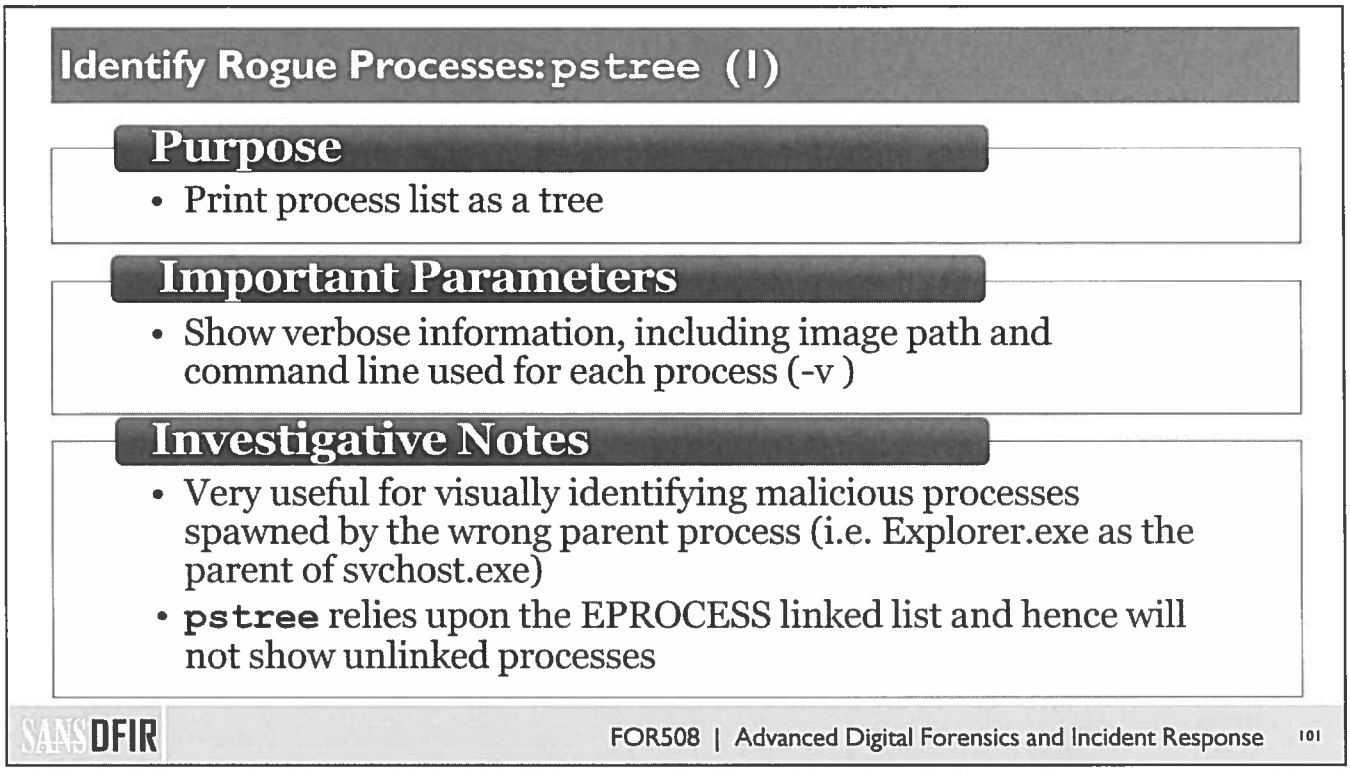
psscan finds additional processes not found by **pslist**
(a.exe is an exited process and not in the pslist output)

```
# vol.py -f win7-32-nromanoff-memory-raw.001 psscan --profile=Win7SP1x86

Offset(P)          Name            PID   PPID PDB        Time created                   Time exited
------------------ ------------ ---- ------ ---------- ------------------------------ ------------------------------
0x0000000004fa0958 System            4      0 0x00185000 2012-04-04 11:47:29 UTC+0000
0x000000007d62a4b8 svchost.exe    2980    564 0x7ecce420 2012-04-04 11:50:42 UTC+0000
0x000000007da0e178 LogonUI.exe     880    520 0x7ecce1a0 2012-04-04 11:47:51 UTC+0000
0x000000007da308e8 svchost.exe     920    564 0x7ecce1c0 2012-04-04 11:47:51 UTC+0000
0x000000007da462b8 svchost.exe    1032    564 0x7ecce200 2012-04-04 11:47:52 UTC+0000
0x000000007da47a58 svchost.exe     944    564 0x7ecce1e0 2012-04-04 11:47:52 UTC+0000
0x000000007df8fd40 winlogon.exe    520    456 0x7ecce040 2012-04-04 11:47:44 UTC+0000
0x000000007ec7f030 csrss.exe       472    456 0x7ecce060 2012-04-04 11:47:44 UTC+0000
0x000000007eccaa70 smss.exe        280      4 0x7ecce020 2012-04-04 11:47:29 UTC+0000
0x000000007eceb430 f-response-ent 7776    564 0x7ecce760 2012-04-06 20:34:42 UTC+0000
0x000000007eef[illegible]540 csrss.exe     412    404 0x7ecce080 2012-04-04 11:47:41 UTC+0000
0x0000000[illegible]  a.exe       5008   4212 0x7ecce960 2012-04-06 13:19:34 UTC+0000   2012-04-06 16:58:26 UTC+0000
0x000000007f01[illegible]8b8 conhost.exe   3408    412 0x7ecce9e0 2012-04-06 14:03:11 UTC+0000

a.exe          5008   4212 0x7ecce960 2012-04-06 13:19:34 UTC+0000   2012-04-06 16:58:26 UTC+0000
```


If we run the **psscan** plugin on our memory image infected with the Sobig worm, we see most of the same processes **pslist** found, with the exception of one, dllhost.exe. If you refer back to our **pslist** slide notes, you will see that this process does not exist in that output. This is most likely because it was terminated and removed from the doubly linked list, but still lingering in unallocated memory space. Though it could also mean that the process was unlinked due to rootkit activity. We came to the conclusion that it was terminated based on running additional plugins to try to enumerate process objects like loaded DLLs and handles, all of which returned no results. By analyzing process start times, we can see that the dllhost.exe process was started before all of the current system processes (including lsass.exe and services.exe). This probably means it survived a "soft" reboot and was carved out of unallocated memory by the **psscan** plugin. Think about the significance of that statement—we can sometimes see processes from previous times the operating system was running! RAM is not nearly as volatile as we once thought.

If you find a suspicious process in the **psscan** output, make note of the physical offset listed for that process. Many of the Volatility plugins will allow you to specify a process by PID ("-p") or by physical offset ("-o"). The latter might work in some cases when the PID is unrecognized.

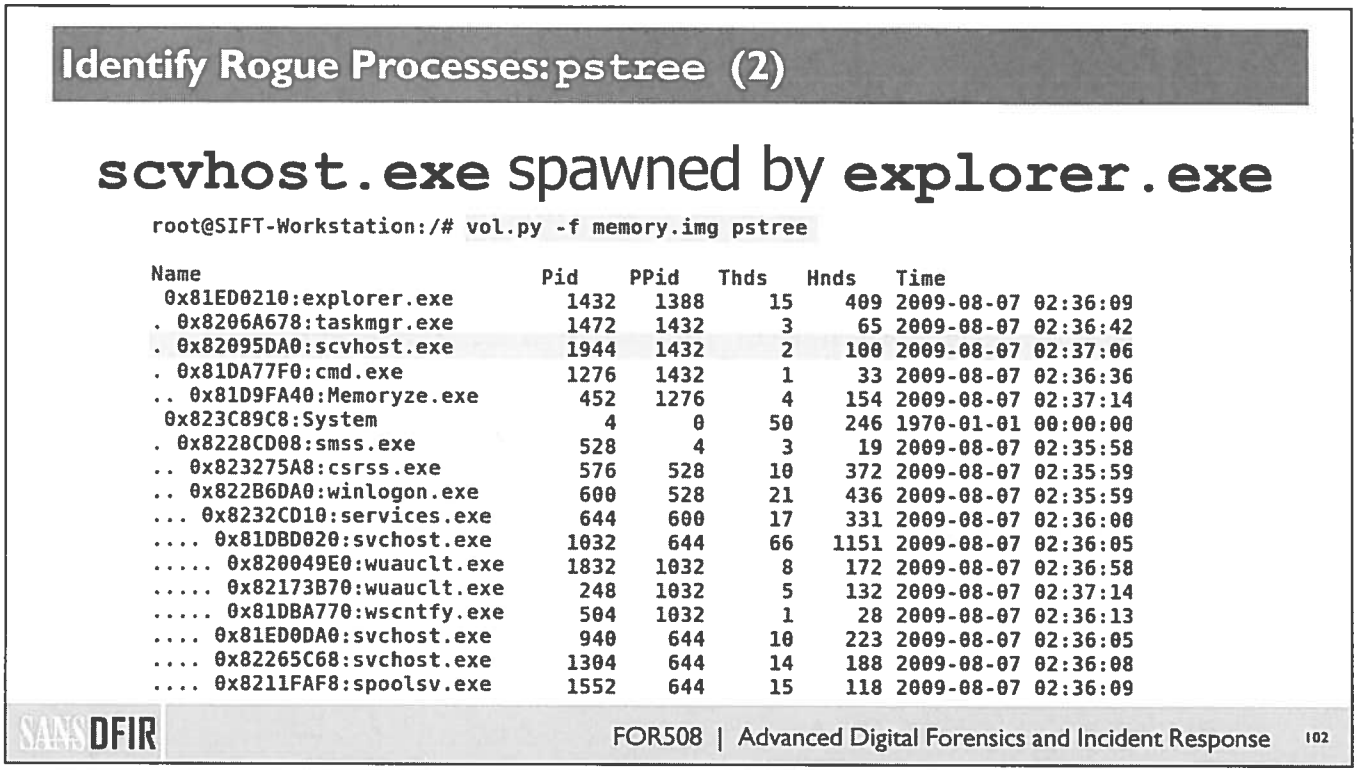
Another item of note for **psscan** is that it does not sort results by their creation times as **pslist** does. This is likely because the list is created as processes are found in memory. You might also see duplicate processes identified due to them being moved around in memory.



Identify Rogue Processes: `pstree` (1)

We previously saw in Redline how viewing parent-process relationships visually can sometimes be very helpful in identifying anomalies. The **`pstree`** plugin gives us this capability in the Volatility framework. It reads the EPROCESS doubly linked list in the kernel and outputs the results as a process tree. Because it gathers information similar to **`pslist`**, it does not have the capability to identify terminated or hidden processes. The information provided by **`pstree`** is the same as in **`pslist`**:

- **Virtual offset of EPROCESS block**
- **Process name**
- **Process Identifier (PID)**
- **Parent Process Identifier (PPID)**
- **Number of threads**
- **Number of handles**
- **Process start time**



Identify Rogue Processes: `pstree` (2)

In this example, we are looking for a process hiding in plain sight. The malicious process was named "**scv**host.exe" in an effort to blend in with the multiple legitimate svchost.exe processes on the system. The human eye might very well have missed the naming distinction when looking at a standard process list. However, when viewed according to the parent-process relationships in the memory image, it becomes quite clear that scvhost.exe was not spawned by the same process (services.exe) as the other svchost.exe processes. The malicious file was spawned by the explorer.exe process in the context of a user shell.

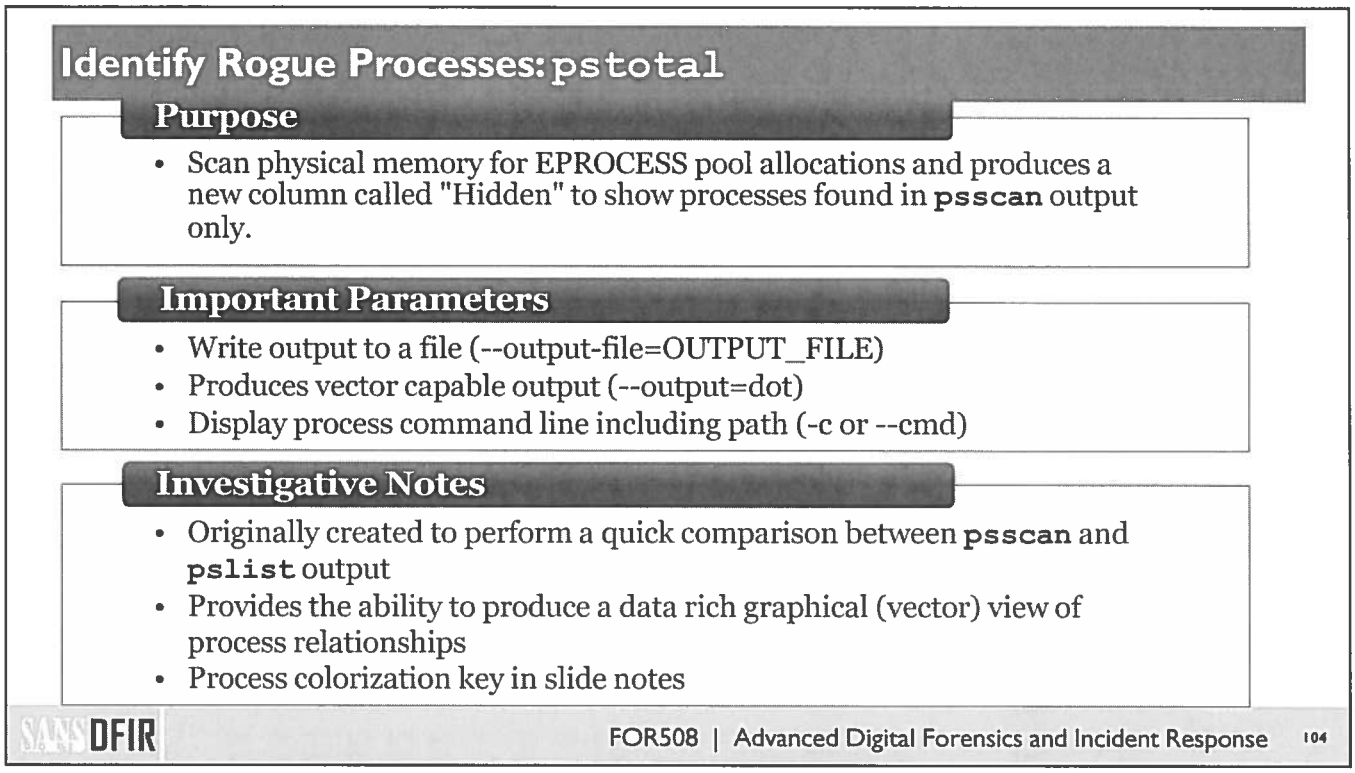
Command line example:

root@SIFT-Workstation:/# **vol.py -f memory.img pstree**

Name	Pid	PPid	Thds	Hnds	Time
0x81ED0210:explorer.exe	1432	1388	15	409	8/7/2009 2:36:09
. 0x8206A678:taskmgr.exe	1472	1432	3	65	8/7/2009 2:36:42
. 0x82095DA0:scvhost.exe	1944	1432	2	100	8/7/2009 2:37:06
. 0x82161918:VMwareTray.exe	1708	1432	1	29	8/7/2009 2:36:10

...SNIP...

```
root@SIFT-Workstation:/# vol.py -f memory.img pstree

Name                                Pid    PPid   Thds   Hnds   Time
 0x81ED0210:explorer.exe           1432   1388     15    409 2009-08-07 02:36:09
. 0x8206A678:taskmgr.exe           1472   1432      3     65 2009-08-07 02:36:42
. 0x82095DA0:scvhost.exe           1944   1432      2    100 2009-08-07 02:37:06
. 0x81DA77F0:cmd.exe               1276   1432      1     33 2009-08-07 02:36:36
.. 0x81D9FA40:Memoryze.exe          452   1276      4    154 2009-08-07 02:37:14
 0x823C89C8:System                    4      0     50    246 1970-01-01 00:00:00
. 0x8228CD08:smss.exe               528      4      3     19 2009-08-07 02:35:58
.. 0x823275A8:csrss.exe             576    528     10    372 2009-08-07 02:35:59
.. 0x822B6DA0:winlogon.exe          600    528     21    436 2009-08-07 02:35:59
... 0x8232CD10:services.exe         644    600     17    331 2009-08-07 02:36:00
.... 0x81DBD020:svchost.exe        1032    644     66   1151 2009-08-07 02:36:05
..... 0x820049E0:wuauclt.exe       1832   1032      8    172 2009-08-07 02:36:58
..... 0x82173B70:wuauclt.exe        248   1032      5    132 2009-08-07 02:37:14
..... 0x81DBA770:wscntfy.exe        504   1032      1     28 2009-08-07 02:36:13
.... 0x81ED0DA0:svchost.exe         940    644     10    223 2009-08-07 02:36:05
.... 0x82265C68:svchost.exe        1304    644     14    188 2009-08-07 02:36:08
.... 0x8211FAF8:spoolsv.exe        1552    644     15    118 2009-08-07 02:36:09
```

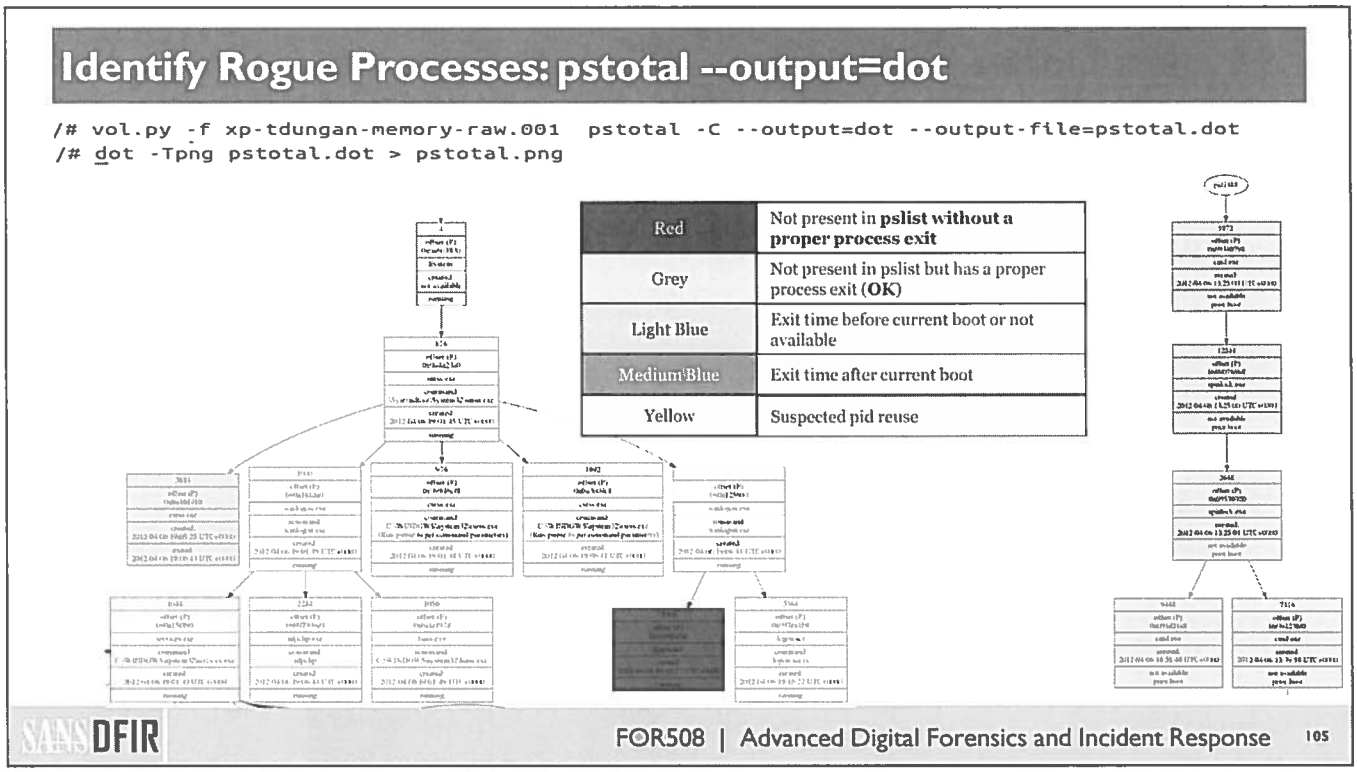


Identify Rogue Processes: `pstotal`

As far as new plugins go, **`pstotal`** provides features not seen in any other plugin. Sue Stirrup modified the original **`pstotal`** plugin written by Jesse Kornblum, and it does two things for us that help us analyze processes in memory. First, it provides a comparison between **`pslist`** and **`psscan`** outputs to help identify which processes were not present in the standard EPROCESS list. If a process is not present in the list, it can be an indicator of an exited process or perhaps a hidden one. **`Pstotal`** can create a vector drawing of the process parent-child relationships, with the **`--output=dot`** option. This can be useful to visualize relationships to see exactly which process spawned another and is more data rich than **`pstree`** output.

The vector output colorizes process containers to represent information about that process. The current color scheme is as follows (code comments in the plugin also document this list in case you ever need another reference):

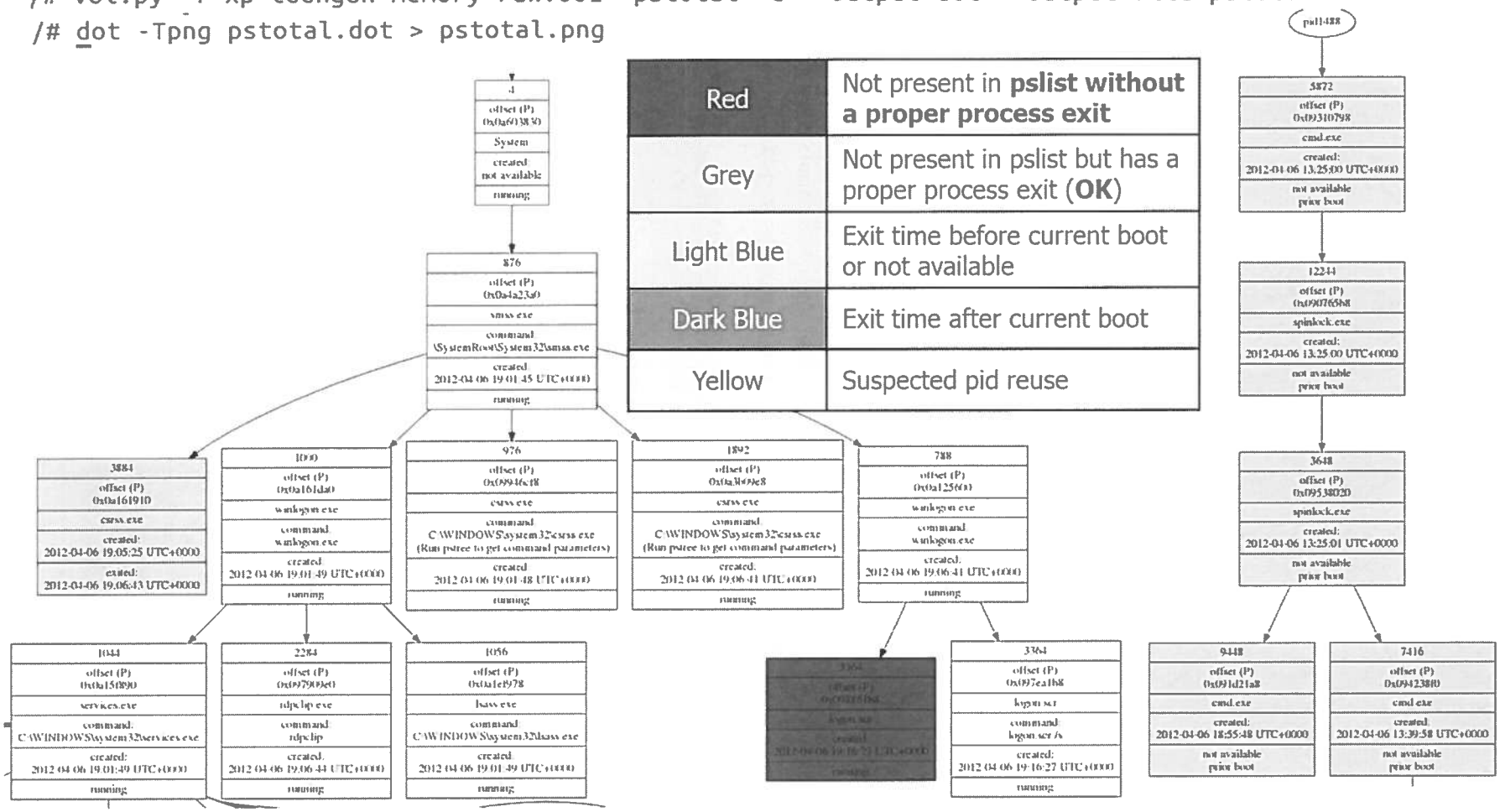
- **Red**: The process is not present in **`pslist`** output and does not have an exit timestamp (worth further analysis if a rootkit is suspected)
- **Grey**: The process is not present in **`pslist`** output but does have an exit timestamp (likely exited)
- **Light Blue**: The exit time of this process was before the most recent boot time
- **Medium Blue**: The exit time was after most recent boot time
- **Yellow**: Indication of potential PID reuse

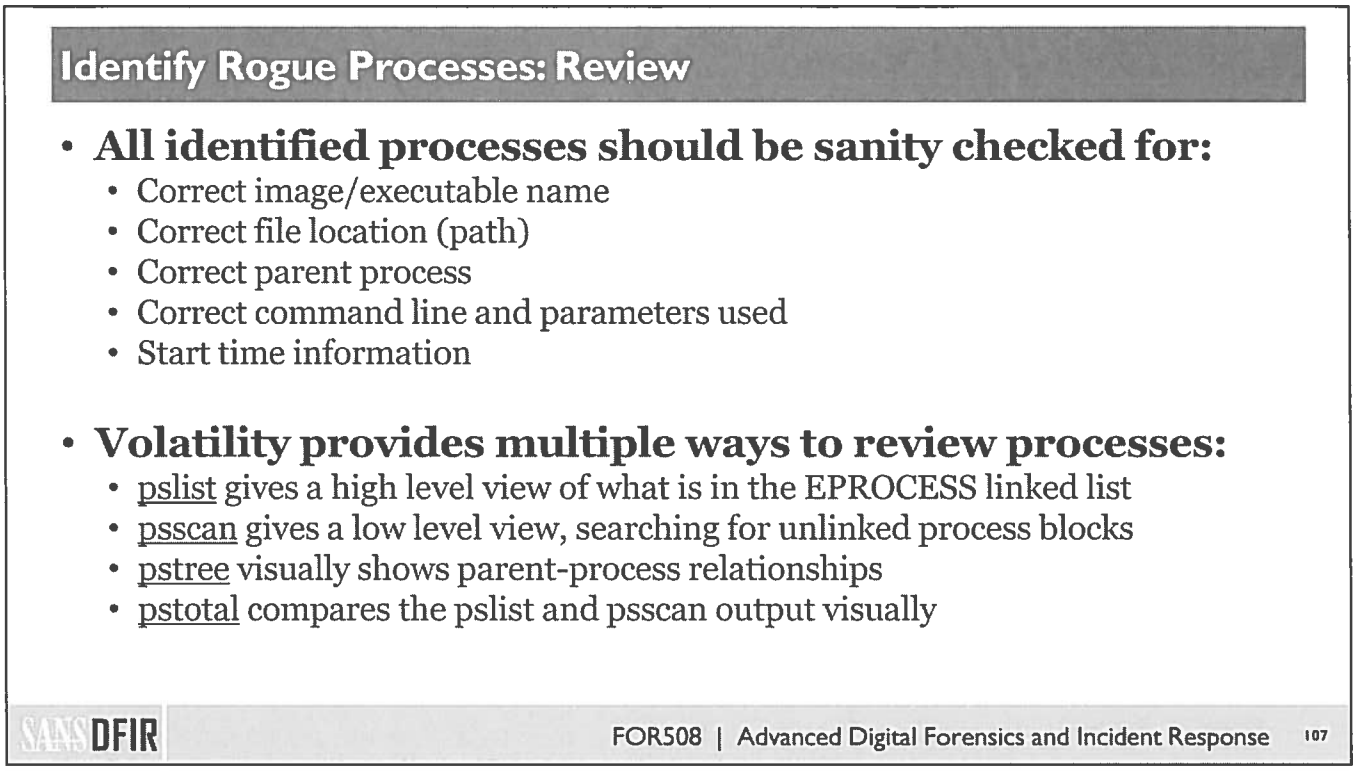


Identify Rogue Processes: `pstotal --output=dot`

We can also use **`pstotal`** to create a graphical output file with the –output=dot option. Using the **`dot`** program inside the SIFT workstation, the vector graph info can be converted into a PNG file (-Tpng option). The SIFT workstation also includes the graphical **`xdot`** tool, which can display the native dot format.

```
/# vol.py -f xp-tdungan-memory-raw.001  pstotal -C --output=dot --output-file=pstotal.dot
/# dot -Tpng pstotal.dot > pstotal.png
```





Identify Rogue Processes: Review

Reviewing running processes is one of the most basic activities accomplished during memory analysis. By reviewing the attributes of each process, you might be able to identify anomalies that lead you to suspect certain processes. Over the next several sections, we will learn how to use Volatility to gather additional information about a process in order to confirm or dispel our suspicions.

We covered three different mechanisms within Volatility to gather and present process information. **`Pslist`** is fast and simple, but might miss hidden processes. **`Psscan`** doesn't trust the standard kernel mechanisms for tracking processes and instead scans the entire memory image, giving it the ability to find hidden and terminated processes. Finally, **`pstree`** and **`pstotal`** display information visually, showing the parent-process relationships for each process.

Step 2:

Analyzing Process Objects

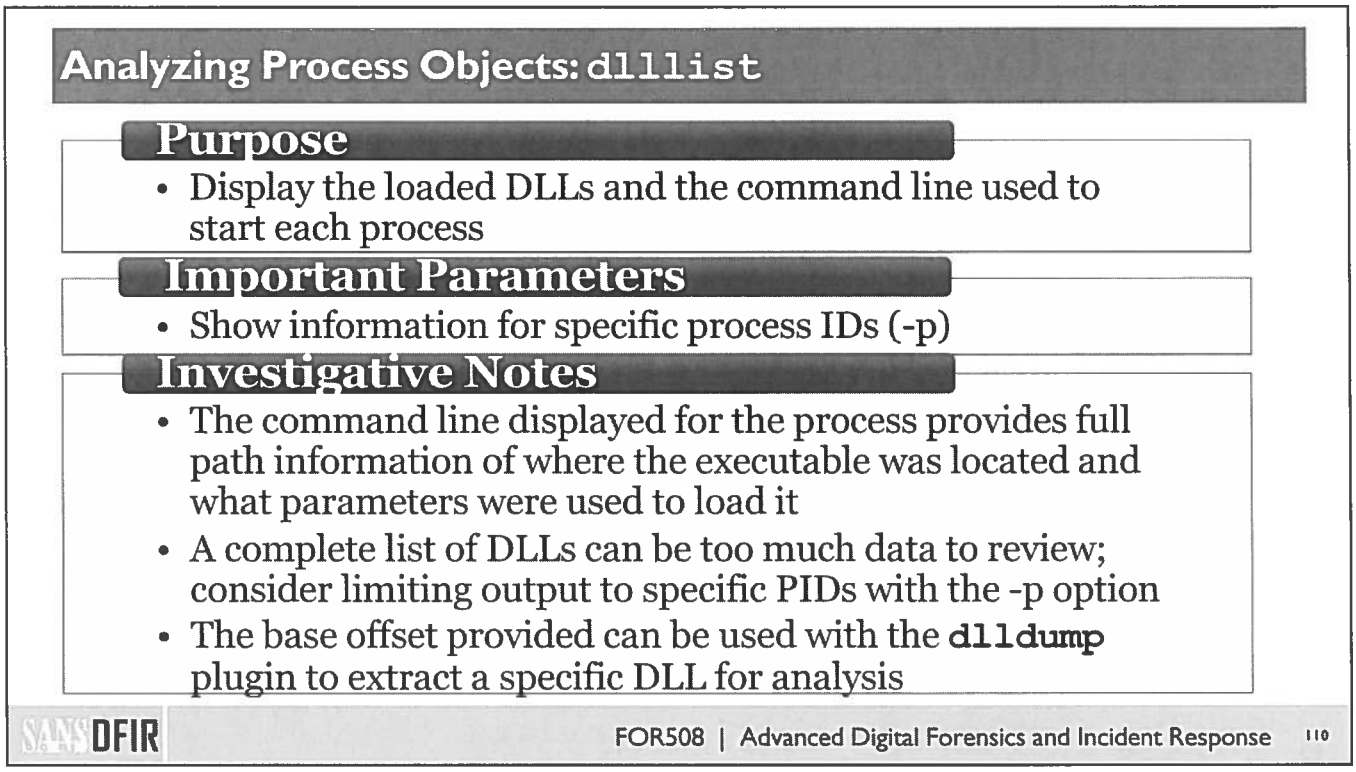
This page intentionally left blank.

Analyzing Process Objects Plugins

dlllist	Print list of loaded dlls for each process
cmdline	Display command-line args for each process
getsids	Print the ownership SIDs for each process
handles	Print list of open handles for each process
filescan	Scan memory for FILE_OBJECTs
mutantscan	Scan memory for mutant objects (KMUTANT)
svcscan	Scan memory for Windows Service information
cmdscan	Scan for COMMAND_HISTORY buffers
consoles	Scan for CONSOLE_INFORMATION output

Analyzing Process Objects Plugins

Upon reaching the "Analyze Process Objects" step, we have likely identified some suspicious processes. Although we might have a feeling that a process is bad, there is much, much more for us to base our decision on than image name, parent process, and start time. When analyzing process objects, we are delving deep into each process to identify and assess their component parts. Volatility has a wealth of plugins allowing us to do this, and we have chosen the most useful for this purpose. The **dlllist** plugin provides loaded DLLs and command-line information, **getsids** shows the security identifiers belonging to each process, **handles** outputs information on the many different handles a process can own, and finally, **svcscan** gives Windows Service information. The **cmdscan** and **consoles** plugins give tremendous new capabilities for carving out command input and output within console applications like cmd.exe or even PowerShell.exe. A few extra tools are listed (not bolded on the slide) and although certainly useful, will be left as homework for the reader. **Cmdline** is a very simple plugin designed to output command lines for each process. **Filescan** and **mutantscan** are specialized plugins that go beyond the output provided by **handles** by scanning memory for file and mutant handles.



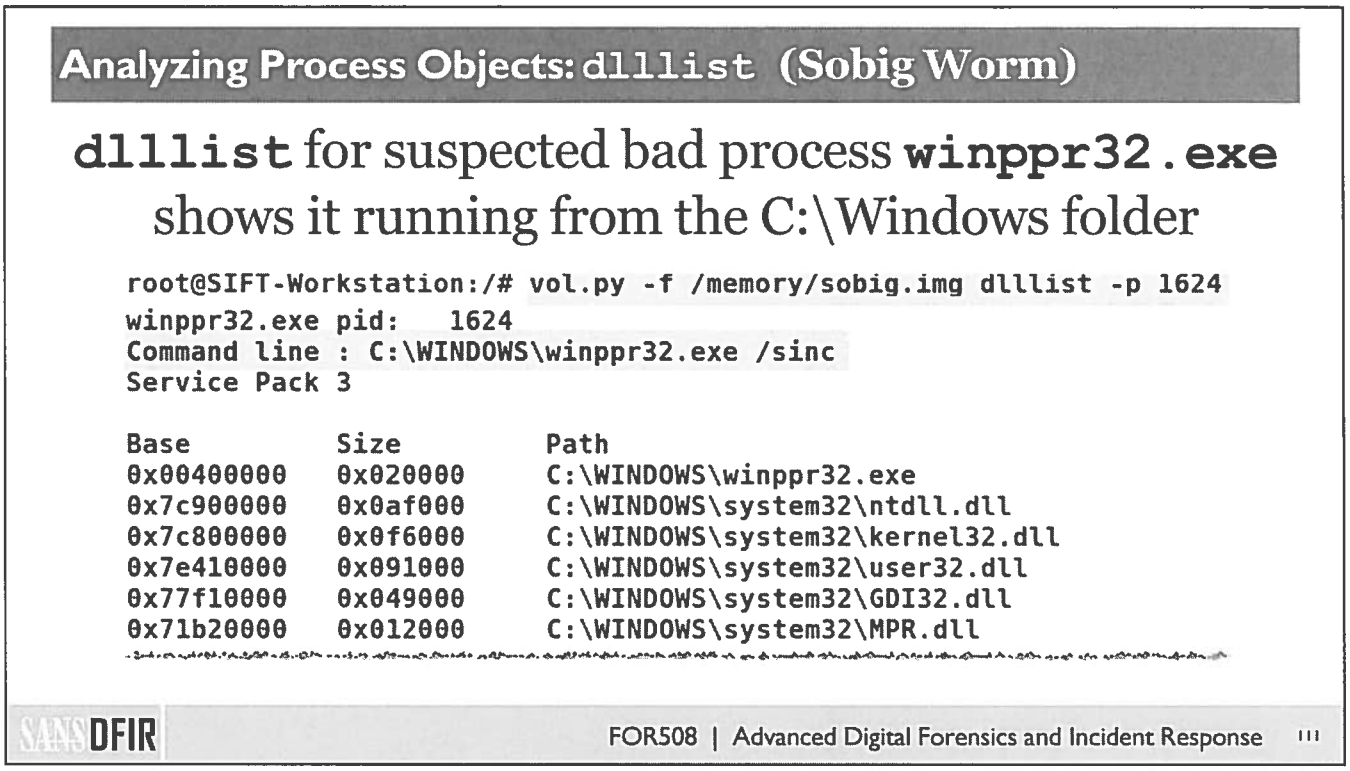
Analyzing Process Objects: `dlllist`

Windows processes rely extensively on loaded libraries to accomplish their functions. Knowing what libraries were loaded can give insight into what the process was doing. It is also an excellent way to detect malicious DLLs that have been loaded. The Process Environment Block (PEB) for each process tracks loaded DLLs. Interestingly, the EPROCESS block allows only 16 bytes for an image name, so to get the full name, you must look in the PEB where it is stored along with the command line (plus parameters) used to start the process.

There is one important option with **`dlllist`**, and that is the ability to limit the information to a comma-separated list of processes using "-p." The DLLs loaded for every process number in the hundreds, so a better strategy is to review them for a smaller set of processes you might be unsure about or for which you desire more information. The plugin provides the following information for each loaded DLL:

- **Base offset**
- **DLL size**
- **DLL file path**

The base offset of the DLL can be useful, because it can be used with another plugin named **`dlldump`** to extract individual DLLs for further analysis. This plugin reports only information found within the PEB. Thus, it is not successful at identifying attacks like reflective DLL injection, which circumvent updating the PEB lists. Another plugin, **`ldrmodules`**, provides additional information about loaded DLLs, helping to identify more advanced attacks. We will cover both of these plugins later in the course.



Analyzing Process Objects: `dlllist` (Sobig Worm)

During our analysis of the memory image infected with the Sobig Worm, we found a suspicious process named **`winppr32.exe`** with a process ID of 1624. To gather more information about that process, we ran the following:

vol.py –f /memory/sobig.img --profile=WinXPSP3x86 dlllist –p 1624

Notice the use of the "-p" flag to limit the results to just the process we are currently analyzing. The **`dlllist`** output gives us the full path used to execute the process, so we see:

Full path: C:\Windows\winppr32.exe

Executable parameters: /sinc

The fact that the process is running from the C:\Windows folder might be an important clue. We also see the various loaded DLLs for the process. Assuming the process has been subverted, we might be able to identify malicious DLL files that can help us better understand the attack. Pay close attention to the name of each DLL as well as where the DLL is located. As an example, Stuxnet malware loaded a malicious DLL into one of its code-injected processes named **`C:\Windows\System32\kernel32.dll.aslr.<random>`**, which sticks out very clearly in the **`dlllist`** plugin output.

Command-line example:

root@SIFT-Workstation:/# **vol.py -f /memory/Stuxnet.img dlllist -p 1928**

****

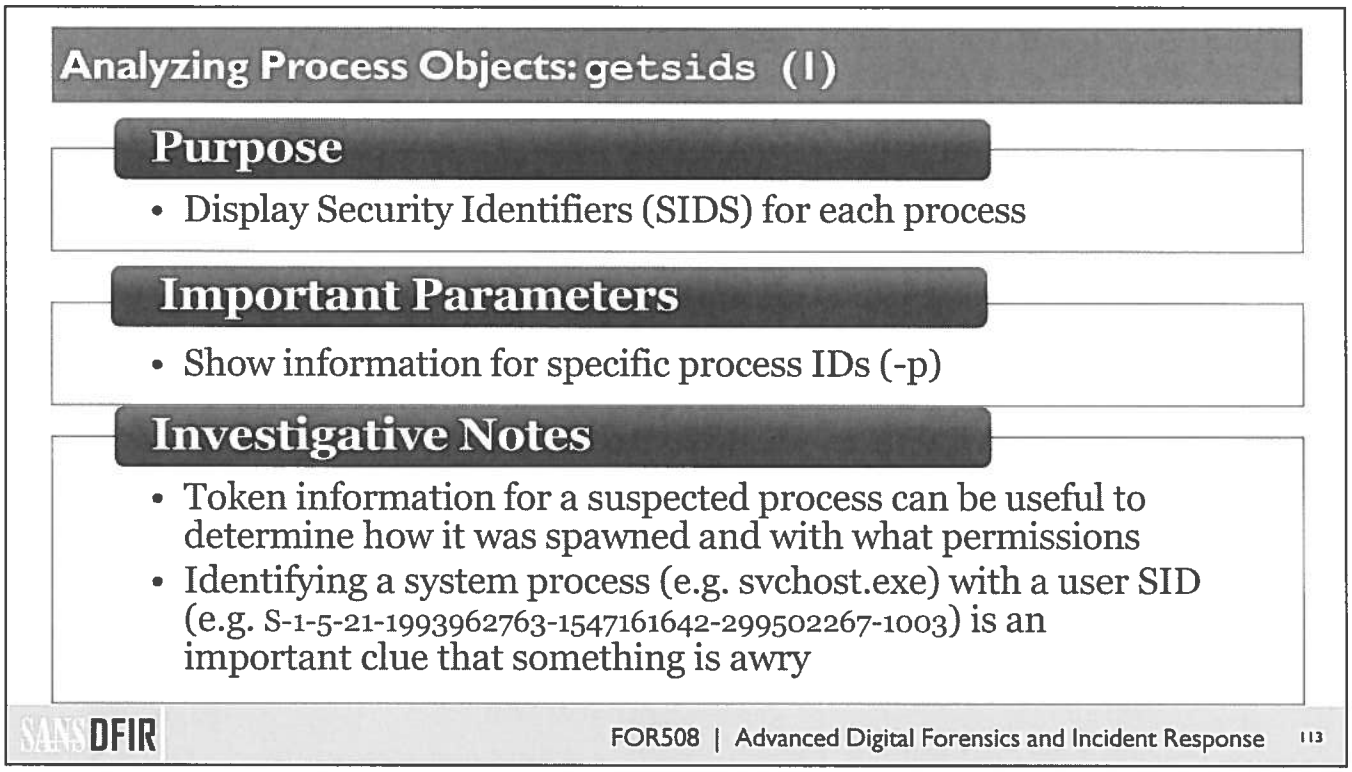
lsass.exe pid: 1928

Command line : "C:\WINDOWS\\system32\\lsass.exe"

Service Pack 3

Base	Size	Path
0x01000000	0x006000	C:\WINDOWS\system32\lsass.exe
0x7c900000	0x0af000	C:\WINDOWS\system32\ntdll.dll
0x7c800000	0x0f6000	C:\WINDOWS\system32\kernel32.dll
0x77dd0000	0x09b000	C:\WINDOWS\system32\ADVAPI32.dll
0x77e70000	0x092000	C:\WINDOWS\system32\RPCRT4.dll
0x77fe0000	0x011000	C:\WINDOWS\system32\Secur32.dll
0x7e410000	0x091000	C:\WINDOWS\system32\USER32.dll
0x77f10000	0x049000	C:\WINDOWS\system32\GDI32.dll
0x00870000	**0x138000**	**C:\WINDOWS\system32\KERNEL32.DLL.ASLR.0360b7ab**
0x76f20000	0x027000	C:\WINDOWS\system32\DNSAPI.dll
0x77c10000	0x058000	C:\WINDOWS\system32\msvcrt.dll
0x71ab0000	0x017000	C:\WINDOWS\system32\WS2_32.dll
0x71aa0000	0x008000	C:\WINDOWS\system32\WS2HELP.dll
0x76d60000	0x019000	C:\WINDOWS\system32\IPHLPAPI.DLL
0x5b860000	0x055000	C:\WINDOWS\system32\NETAPI32.dll
0x774e0000	0x13d000	C:\WINDOWS\system32\ole32.dll
0x77120000	0x08b000	C:\WINDOWS\system32\OLEAUT32.dll
0x76bf0000	0x00b000	C:\WINDOWS\system32\PSAPI.DLL

…SNIP…



Analyzing Process Objects: `getsids` (1)

Every process inherits an access token from the account spawning it. A token is a collection of security identifiers (SIDs) that describe user permissions and user groups. A review of process tokens can help identify malicious processes. Most system processes use well-known system accounts and security identifiers to perform their tasks. Microsoft provides a reference to these on its support site.[1] Example SIDs from a legitimate "lsass.exe" process are:

lsass.exe (556): S-1-5-18 (Local System)
lsass.exe (556): S-1-5-32-544 (Administrators)
lsass.exe (556): S-1-1-0 (Everyone)
lsass.exe (556): S-1-5-11 (Authenticated Users)
lsass.exe (556): S-1-16-16384 (System Mandatory Level)

This process (PID 556) was apparently spawned using the Local System account. However, if you were to see this same process running with user privileges (indicated by a user SID being present), that might be an important clue that something is amiss:

lsass.exe (556): S-1-5-21-4251235867-3156790139-409172211-1001

User SIDs are assigned to processes started under a user context, which would never be the case for the lsass.exe process.

As explained by Microsoft: "An *access token* is an object that describes the *security context* of a *process* or thread. The information in a token includes the identity and privileges of the user account associated with the process or thread. When a user logs on, the system verifies the user's password by comparing it with information

stored in a security database. If the password is *authenticated*, the system produces an access token. Every process executed on behalf of this user has a copy of this access token."[2]

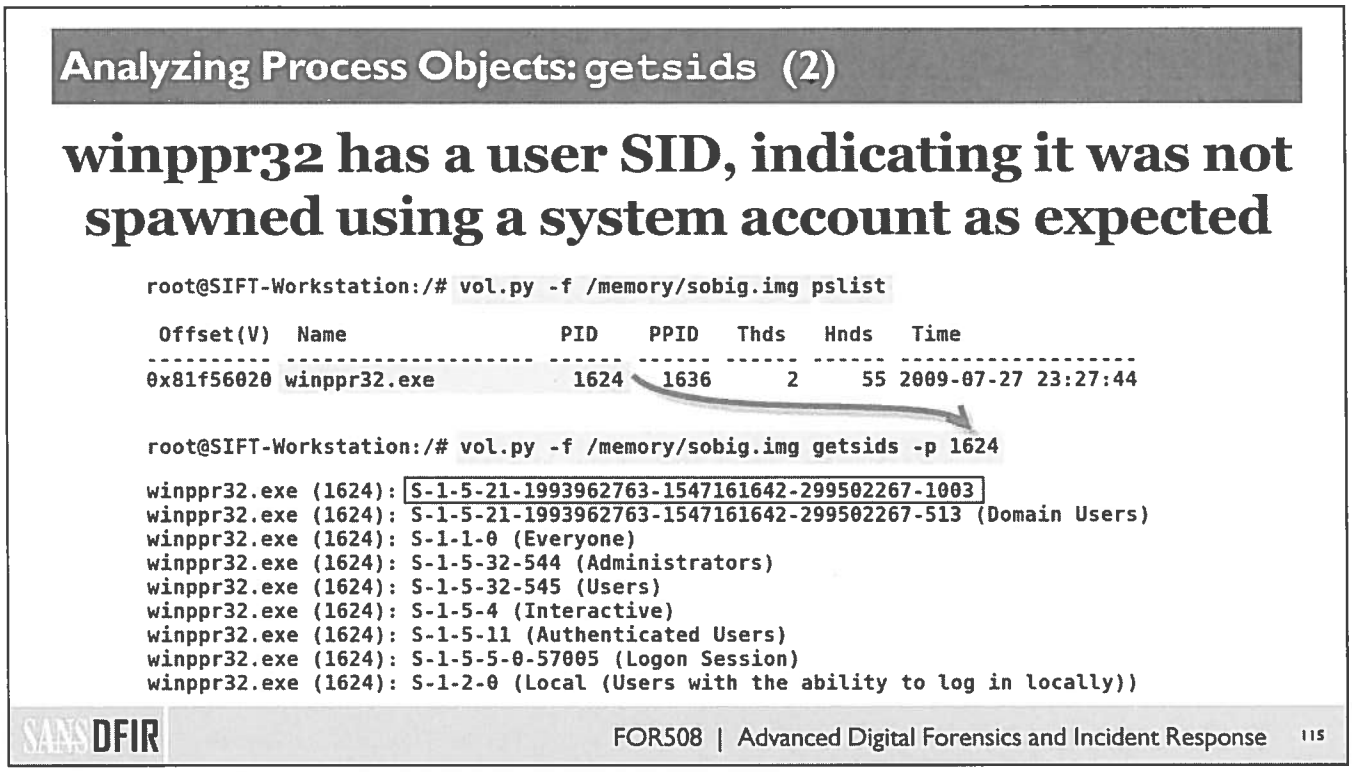
The Microsoft article, "How Access Tokens Work," is also an excellent reference on the subject.[3]

Now taking things a step further, in this example you could identify exactly which user account might have been compromised by matching up the user SID with account information stored in the SAM or SOFTWARE registry hive!

[1] http://support.microsoft.com/kb/243330

[2] http://msdn.microsoft.com/en-us/library/windows/desktop/aa374909(v=vs.85).aspx

[3] http://technet.microsoft.com/en-us/library/cc783557(v=ws.10).aspx



Analyzing Process Objects: `getsids` (2)

In this example, we are diving deeper into a suspicious process named "winppr32.exe," previously identified as suspicious. The process is named to blend in as a legitimate Windows system process. A review of its security identifiers (SIDs) with the **`getsids`** plugin shows a user SID associated with the process. Although this does not necessarily mean the process is malicious, it does tell us that the process was likely spawned from a user context and hence is unlikely to be a true system process. The other SIDs, shown below the user SID, are group SIDs. For instance, we can tell that the user account that spawned winppr32.exe was likely a local admin, but not a domain admin (the domain admin group RID is 512, not 513 as shown in this example). A great way to learn more about SIDs and group information is to run the command **`whoami /all`** from a Windows command prompt. It will nicely show the currently logged in user's SIDs, groups, and privileges assigned.

Command-line example:

root@SIFT-Workstation:/# **vol.py -f /memory/sobig.img pslist**

Offset(V)	Name	PID	PPID	Thds	Hnds	Time
0x81f56020	**winppr32.exe**	**1624**	1636	2	55	2009-07-27 23:27:44

root@SIFT-Workstation:/# vol.py -f /memory/sobig.img getsids -p 1624

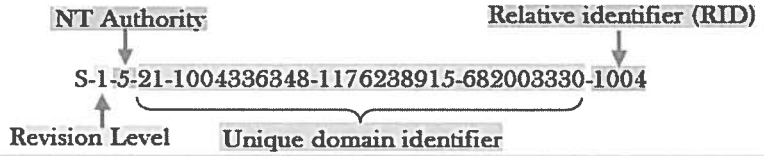
winppr32.exe (1624): S-1-5-21-1993962763-1547161642-299502267-1003
winppr32.exe (1624): S-1-5-21-1993962763-1547161642-299502267-513 (Domain Users)
winppr32.exe (1624): S-1-1-0 (Everyone)
winppr32.exe (1624): S-1-5-32-544 (Administrators)
winppr32.exe (1624): S-1-5-32-545 (Users)
winppr32.exe (1624): S-1-5-4 (Interactive)
winppr32.exe (1624): S-1-5-11 (Authenticated Users)
winppr32.exe (1624): S-1-5-5-0-57005 (Logon Session)
winppr32.exe (1624): S-1-2-0 (Local (Users with the ability to log in locally))

Understanding Security Identifiers (SID)

- A SID identifies a user or group
  - Also represents a privilege level similar to Unix UID/GID

Well-Known Account SIDs		Well-Known Group SIDs	
LocalSystem	S-1-5-18	Administrators	S-1-5-32-544
LocalService	S-1-5-19	Users	S-1-5-32-545
NetworkService	S-1-5-20	Guests	S-1-5-32-546

- A user account consists of a unique domain identifier and relative identifier for that account



SANS DFIR FOR508 | Advanced Digital Forensics and Incident Response 116

Understanding Security Identifiers (SID)

Security Identifiers (SIDs) are unique identifiers assigned to objects by Windows systems and domain controllers. Behind the scenes, Windows uses these SIDs to reference specific accounts and to check security privileges for those accounts. SIDs will be unique within a Windows instance, and domain SIDs will be unique throughout the enterprise.

In memory forensics, we often review SID information in the context of a process. Each running process is assigned an access token that contains the user account SID, SIDs for every group that the user is a member (including the primary group), and a list of privileges and ACLs related to the user account.[1] Most importantly, this information tells us what user account spawned each process. The group information stored by the process gives us additional clues as to the permissions afforded that particular account (for example, if the account is a Domain Administrator, or member of the Remote Desktop Users group).

As you might expect, nearly all Windows system processes are spawned by built-in Windows accounts. These built-in accounts have well-known SIDs associated with them that look much different than a user account SID. The three most popular SIDs used are listed on this slide (LocalSystem, LocalService, and NetworkService). Groups also have well-known SIDs that are well documented by Microsoft. [2] If you see a longer SID assigned to a process, this means that a user account was used to spawn that process. If that process happens to be Chrome.exe, that is normal because Google Chrome is a userland application. On the other hand, if the process is Svchost.exe, that would be abnormal.

Breaking a user SID into its component parts can be instructive:

```
S-1-5-21-1004336348-1176238915-682003330-1004
```

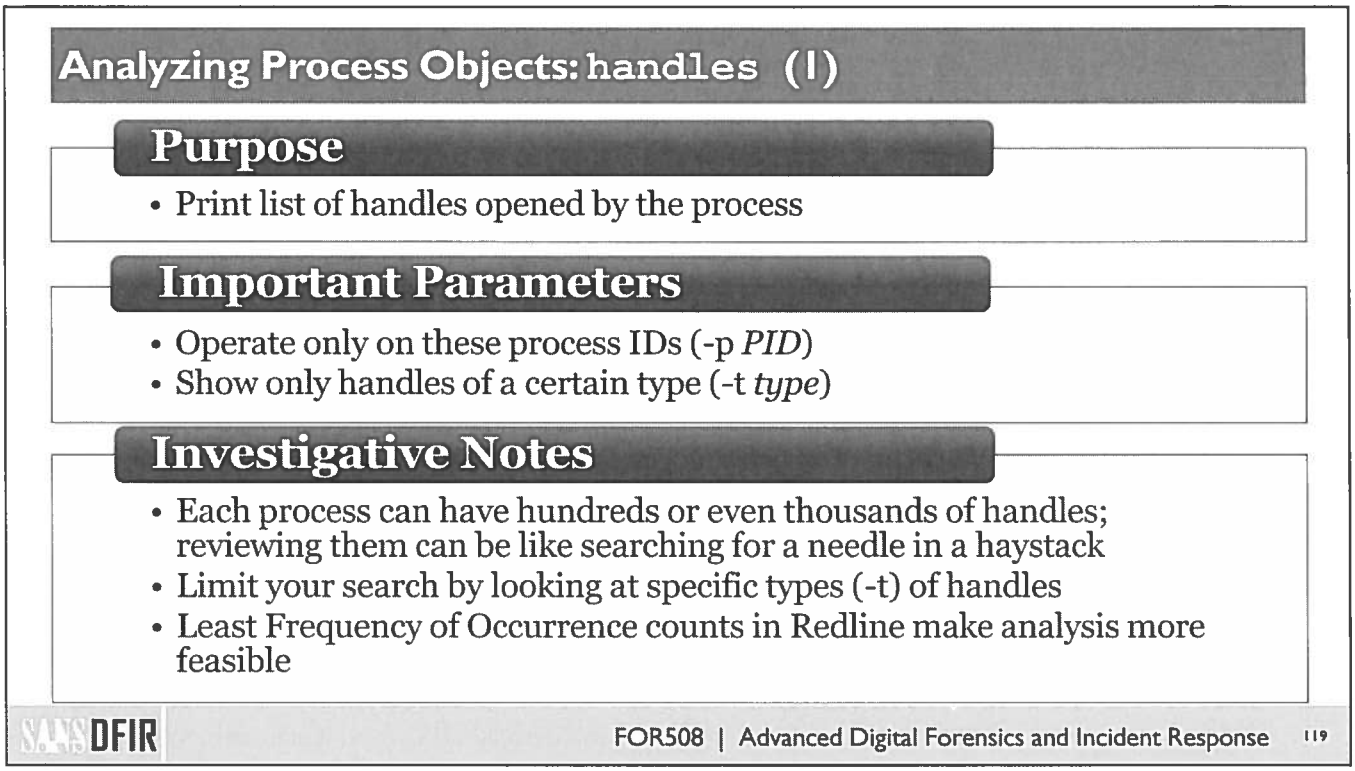
- `S`: Indicates that the string that follows is a SID.
- `1`: Revision Level for the type of SID (we are still on the first revision).
- `5`: Identifier Authority (the most typical issuing authority—in rare cases, you might see others like World Authority (1) and Creator Authority (3)).
- `21-1004336348-1176238915-682003330`: A unique identifier for a specific domain that the account belongs to. Each domain in an enterprise will be assigned a unique identifier. Each domain account will contain this domain identifier as part of its SID.
- `1004`: The relative identifier (RID) for the account within that domain. Every account will have a unique RID within the domain.

The complete SID+RID is the value Windows uses to reference a particular user account. SIDs are everywhere you look in Windows. We commonly encounter them during forensic investigations in the Registry, event logs, and, of course, in process tokens.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/aa374909(v=vs.85).aspx

[2] http://msdn.microsoft.com/en-us/library/cc980032.aspx

```
root@SIFT-Workstation:/# vol.py -f /memory/sobig.img pslist

 Offset(V)  Name                    PID    PPID   Thds   Hnds   Time
---------- -------------------- ------ ------ ------ ------ -------------------
0x81f56020 winppr32.exe            1624   1636      2     55 2009-07-27 23:27:44


root@SIFT-Workstation:/# vol.py -f /memory/sobig.img getsids -p 1624

winppr32.exe (1624): S-1-5-21-1993962763-1547161642-299502267-1003
winppr32.exe (1624): S-1-5-21-1993962763-1547161642-299502267-513 (Domain Users)
winppr32.exe (1624): S-1-1-0 (Everyone)
winppr32.exe (1624): S-1-5-32-544 (Administrators)
winppr32.exe (1624): S-1-5-32-545 (Users)
winppr32.exe (1624): S-1-5-4 (Interactive)
winppr32.exe (1624): S-1-5-11 (Authenticated Users)
winppr32.exe (1624): S-1-5-5-0-57005 (Logon Session)
winppr32.exe (1624): S-1-2-0 (Local (Users with the ability to log in locally))
```

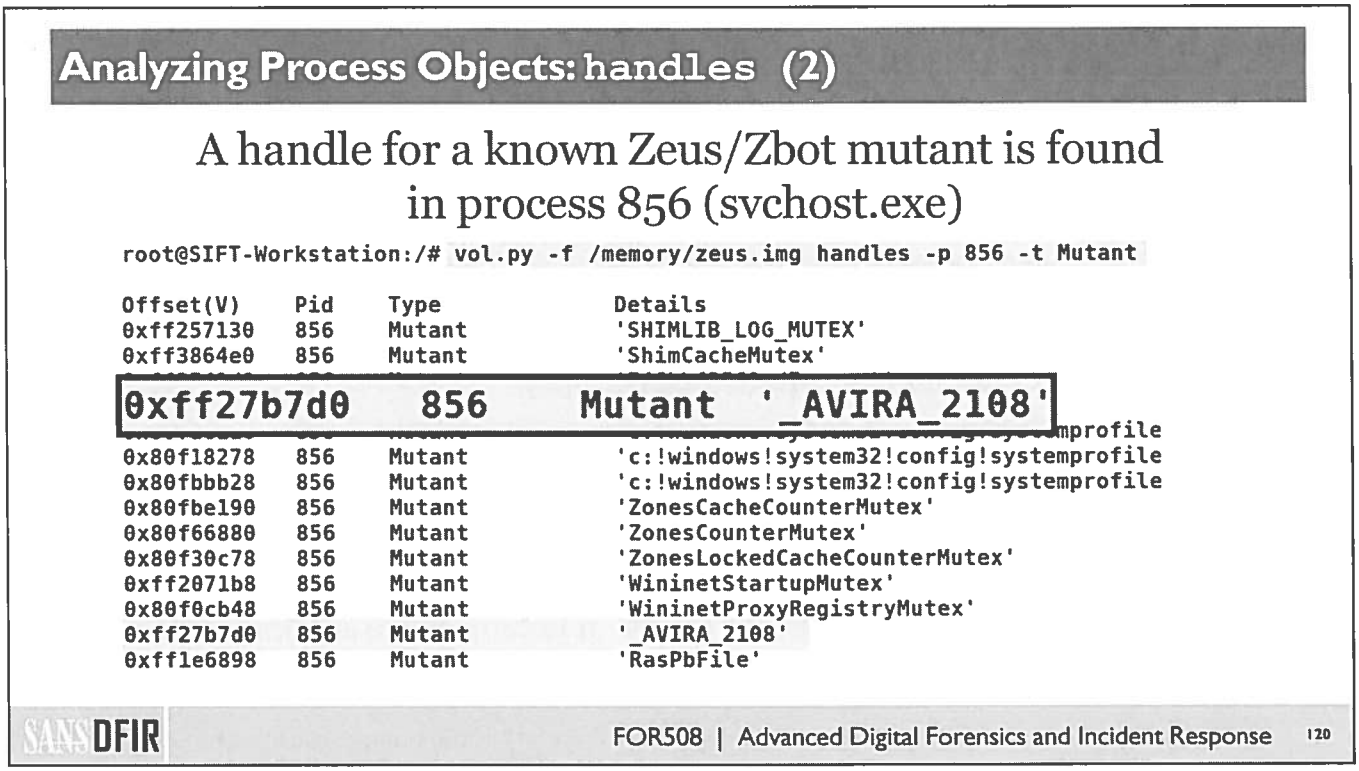


Analyzing Process Objects: `handles` (1)

Handles are important process objects for our analysis, but they can be difficult to analyze because they are so numerous. Unless the malware is well known, or you are lucky, they rarely are the first item noticed pointing to a suspicious process. More commonly, they are reviewed for specific processes that are already suspected of being malicious. Used in this manner, they can be very helpful for confirming suspicions or for identifying a specific malware variant. Knowing this, there are two important parameters for the Volatility handles plugin. The "-p" flag allows us to search a subset of processes (indicated through a list of comma-separated process IDs). We can also request specific handle types, weeding out those that are not interesting or too numerous. This is done via the "-t" flag. This flag requires the handle type to be specified (multiple handle types can be presented as a comma-separated list). When using the "-t" option, make sure to maintain the same capitalization of the handle type as you see on this list. The option is case sensitive. The available handle types are:

- Process
- Thread
- Key
- Event
- File
- Mutant
- Semaphore
- Token
- WmiGuid
- Port
- Thread
- Directory
- WindowStation
- IOCompletion
- Timer

Two additional Volatility plugins exist for more rigorous searching of file handles and mutants; **`filescan`** and **`mutantscan`** search for markers indicating FILE_OBJECTS and _KMUTANT objects and return their respective results.



Analyzing Process Objects: handles (2)

During analysis of a system infected with the Zeus bot, a process named "svchost.exe" with PID 856 was identified as containing injected memory sections. The handles plugin for Volatility was used to retrieve handles for PID 856. Further, the results were limited using the "-t" parameter to show only Mutant handles. A review of mutants used by the suspicious process showed a documented mutant for the Zeus bot, further indicating the process was subverted and ultimately that the system was indeed infected.[1]

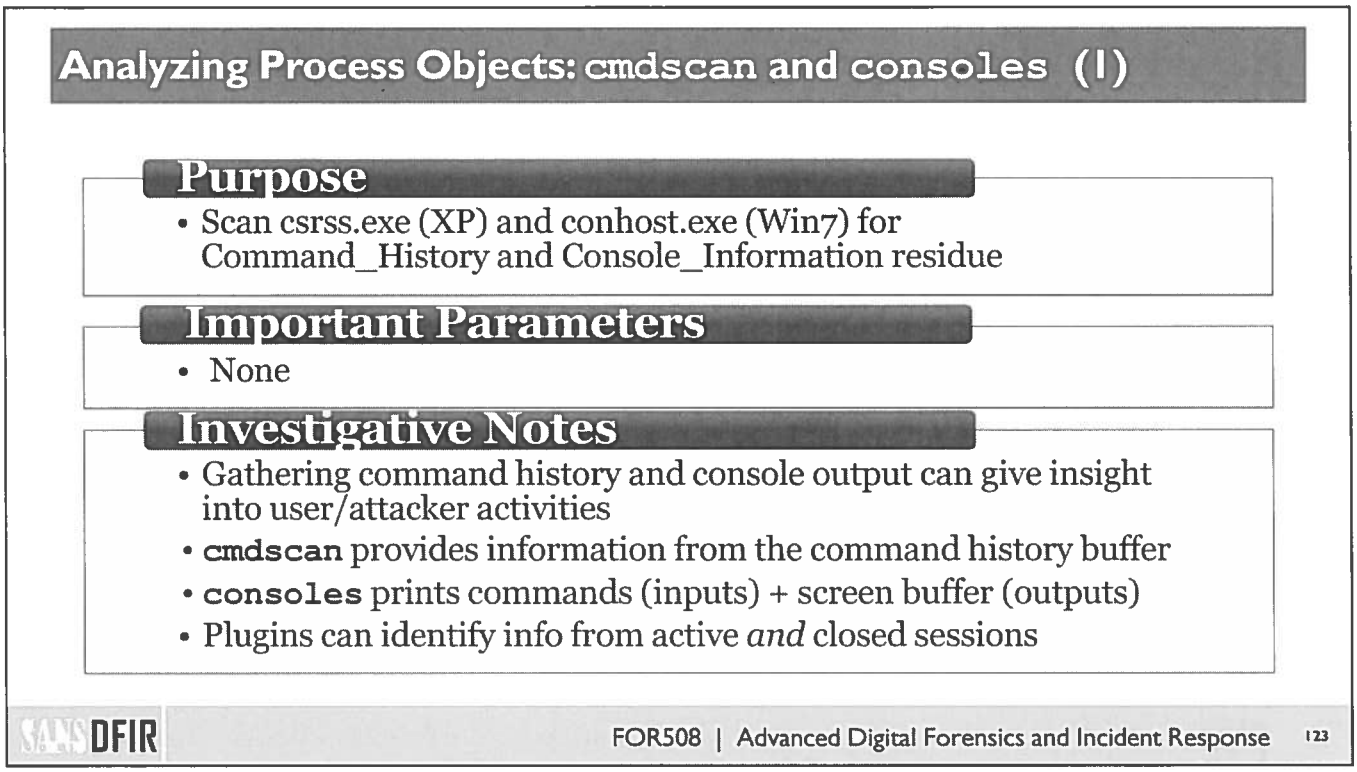
root@SIFT-Workstation:/# **vol.py -f /memory/zeus.img handles -p 856 -t Mutant**

Offset(V)	Pid	Type	Details
0xff257130	856	Mutant	'SHIMLIB_LOG_MUTEX'
0xff149860	856	Mutant	''
0xff2342d0	856	Mutant	''
0xff3864e0	856	Mutant	'ShimCacheMutex'
0xff21e0c8	856	Mutant	''
0xff22f0c8	856	Mutant	''
0xff2232d0	856	Mutant	''
0xff2741d8	856	Mutant	'746bbf3569adEncrypt'
0xff15a2a8	856	Mutant	''
0x80fca0c8	856	Mutant	''
0x80ef7a20	856	Mutant	'_!MSFTHISTORY!_'
0x80fdc1a0	856	Mutant	'c:!windows!system32!config!systemprofile!local settings!temporary internet files!content.ie5!'

0x80f18278 856 Mutant 'c:!windows!system32!config!systemprofile!cookies!'
0x80fbbb28 856 Mutant 'c:!windows!system32!config!systemprofile!local settings!history!history.ie5!'
0x80fbe190 856 Mutant 'ZonesCacheCounterMutex'
0x80f66880 856 Mutant 'ZonesCounterMutex'
0x80f30c78 856 Mutant 'ZonesLockedCacheCounterMutex'
0xff2071b8 856 Mutant 'WininetStartupMutex'
0xff1e3d30 856 Mutant ''
0x80f27f48 856 Mutant ''
0x80f0cb48 856 Mutant 'WininetProxyRegistryMutex'
0xff27b7d0 856 Mutant '_AVIRA_2108'
0x80f191e8 856 Mutant ''
0xff1e6898 856 Mutant 'RasPbFile'

[1] http://www.sans.org/reading_room/whitepapers/malicious/clash-titans-zeus-spyeye_33393

```
root@SIFT-Workstation:/# vol.py -f /memory/zeus.img handles -p 856 -t Mutant

Offset(V)     Pid     Type              Details
0xff257130    856     Mutant            'SHIMLIB_LOG_MUTEX'
0xff3864e0    856     Mutant            'ShimCacheMutex'
0xff2741d8    856     Mutant            '746bbf3569adEncrypt'
0x80ef7a20    856     Mutant            '_!MSFTHISTORY!_'
0x80fdc1a0    856     Mutant            'c:!windows!system32!config!systemprofile
0x80f18278    856     Mutant            'c:!windows!system32!config!systemprofile
0x80fbbb28    856     Mutant            'c:!windows!system32!config!systemprofile
0x80fbe190    856     Mutant            'ZonesCacheCounterMutex'
0x80f66880    856     Mutant            'ZonesCounterMutex'
0x80f30c78    856     Mutant            'ZonesLockedCacheCounterMutex'
0xff2071b8    856     Mutant            'WininetStartupMutex'
0x80f0cb48    856     Mutant            'WininetProxyRegistryMutex'
0xff27b7d0    856     Mutant            '_AVIRA_2108'
0xff1e6898    856     Mutant            'RasPbFile'
```



Analyzing Process Objects: `cmdscan` and `consoles` (1)

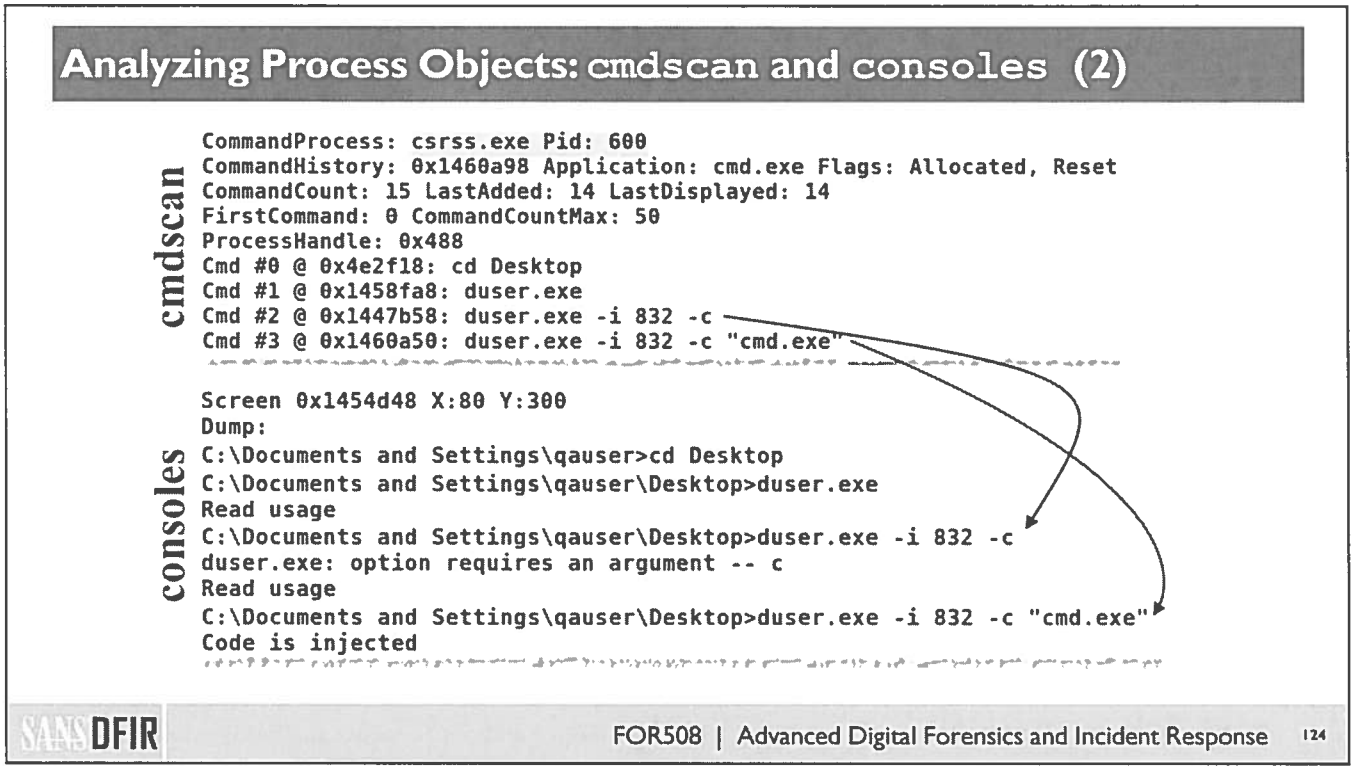
The **`cmdscan`** and **`consoles`** plugins provide a wonderful new capability to carve out full command histories and text console output from memory. Prior to these plugins, investigators were largely relegated to searching through "strings" output for the related processes.

The idea of carving out command history dates back to a seminal paper by Stevens and Casey, *Extracting Windows command line details from physical memory*.[1] The basic idea is that the csrss.exe (XP) and conhost.exe (Win7) processes are responsible for drawing and maintaining text consoles like cmd.exe (or other text consoles like powershell.exe). As such, elements of those consoles are maintained within their assigned memory pages. Searching through the VAD tree of csrss.exe and conhost.exe can yield valuable information related to those console applications. In particular, we can search for the DOSKEY command history buffer kept by cmd.exe. By default, cmd.exe keeps 50 entries within its history buffer. The **`cmdscan`** plugin searches for signatures related to this buffer and outputs the results.

The **`consoles`** plugin takes things one step further. Instead of searching for the command history buffer, it searches for a structure reverse engineered by Michael Hale Ligh called CONSOLE_INFORMATION. This structure keeps a record of the entire console buffer, showing both the input (commands typed) and the output those commands generated. It is like seeing both sides of the conversation! This is a capability that did not exist until Mr. Ligh wrote the **`consoles`** plugin.

The naming scheme for these plugins gives a clue as to the data they can provide. Because the cmdscan plugin scans for signatures, it can recover current and old remnants of command history buffers (hence, it might also return some anomalous data). The consoles plugin will return only input/output buffer information for consoles currently active when memory was dumped.

[1] http://www.dfrws.org/2010/proceedings/2010-307.pdf



Analyzing Process Objects: `cmdscan` and `consoles` (2)

In this example, we see complementary **`cmdscan`** and **`consoles`** output displayed together. From the **`cmdscan`** output, an examiner might start to get the feeling that something is amiss. What type of program is duser.exe? Why do the parameters for "-i" appear to be eerily similar to process IDs (PIDs) on the system? What is the purpose of the "-c" parameter? Although we might be able to identify something bad™ happened here, it isn't until we review the **`consoles`** output that it becomes clear what the purpose of duser.exe is. In this case, it appears to be a code injection tool. Of additional interest is that the user doesn't appear to have a strong familiarity with the tool.

The **`cmdscan`** plugin is a scanning plugin and hence can identify both active and old, closed console application sessions. For each COMMAND_HISTORY signature that it finds, it provides the following information:

- CommandProcess (where the history information was found)
- PID (process ID of CommandProcess)
- Command History (offset where history structure found)
- Application (what console application was running within CommandProcess)
- Flags (any flags set for the structure)
- First Command (location in buffer of first command entry)
- CommandCountMax (size of history buffer)
- LastAdded (entry last added to buffer)
- LastDisplayed (entry last displayed on the screen)
- Process Handle (the application handle number)
- Command list (Cmd #n—the elements stored within the command history buffer)

Similarly, the **consoles** plugin provides the following information:

ConsoleProcess (process containing CONSOLE_INFORMATION structure)
PID (process ID of Process)
Console (offset of process in memory)
Console offset (offset where the console was found)
CommandHistorySize (default number of commands kept in history buffers)
HistoryBufferCount (number of history buffers within console—there can be multiple)
HistoryBufferMax (max number of history buffers kept)
OriginalTitle (first title of console window)
Title (current title of console window)
AttachedProcess (processes run within console by name, PID, and handle ID)
Command list (Cmd #n—the elements stored within the command history buffer)
Screen (coordinates of console window)
Dump (console screen output)

cmdscan

```
CommandProcess: csrss.exe Pid: 600
CommandHistory: 0x1460a98 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 15 LastAdded: 14 LastDisplayed: 14
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x488
Cmd #0 @ 0x4e2f18: cd Desktop
Cmd #1 @ 0x1458fa8: duser.exe
Cmd #2 @ 0x1447b58: duser.exe -i 832 -c
Cmd #3 @ 0x1460a50: duser.exe -i 832 -c "cmd.exe"
```

consoles

```
Screen 0x1454d48 X:80 Y:300
Dump:
C:\Documents and Settings\qauser>cd Desktop
C:\Documents and Settings\qauser\Desktop>duser.exe
Read usage
C:\Documents and Settings\qauser\Desktop>duser.exe -i 832 -c
duser.exe: option requires an argument -- c
Read usage
C:\Documents and Settings\qauser\Desktop>duser.exe -i 832 -c "cmd.exe"
Code is injected
```

Analyzing Process Objects: Review

- **In many cases, the objects that make up a process will provide a clue that something is amiss**
  - DLLs
  - Handles
  - Services
- **There can be thousands of objects associated with a given process**
  - Narrow the focus to suspect processes or those known to be often subverted (e.g. svchost.exe, services.exe, lsass.exe)
  - Check process SIDS and file paths, look for strange services, and use handles when necessary to provide additional confirmation

Analyzing Process Objects: Review

Although process objects can be very useful for identifying suspicious processes, they can also be very challenging to review due to their sheer numbers. By narrowing your focus to specific processes, you can greatly reduce the work. Handles are by far the most voluminous and output can be further limited to show only the specific handle types you might be interested in such as Process, (registry) Key, File, and Mutant. Don't worry about missing something—our analysis process is designed to be layered so even if you miss something in this step, there is a good chance another clue will be discovered in a later step.

Step 3:

Network Artifacts

This page intentionally left blank.

Network Artifact Plugins

`connections`	Print list of active, open TCP connections [XP/2003]
`connscan`	Scan memory for TCP connections, including those closed or unlinked [XP/2003]
`sockets`	Print list of active, available sockets (any protocol) [XP/2003]
`sockscan`	Scan memory for sockets, including those closed or unlinked (any protocol) [XP/2003]
`netscan`	All of the above—scan for both connections and sockets [Vista+]

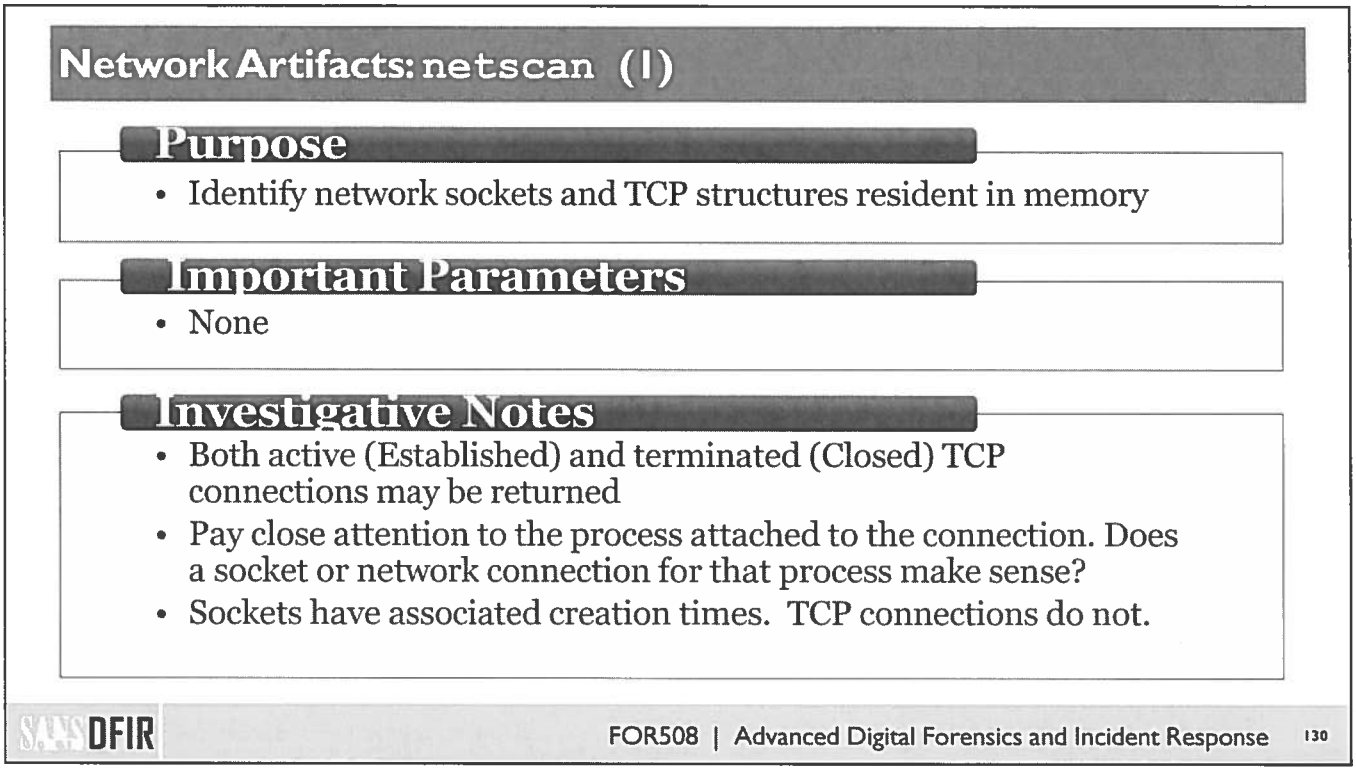
Network Artifact Plugins

There are five primary Volatility plugins to provide information on network artifacts found in a memory image. The network artifact plugins are the only set of Volatility plugins that are operating system dependent. Major changes were made to the TCP/IP stack in Microsoft Vista, changing the underlying data structures in memory and making it much more difficult to find the linked lists pointing to those structures. Instead of updating the existing plugins to work on newer systems, the Volatility team opted to create a new plugin, **`netscan`**. The good news is that it combines both socket and TCP connection data into one output, making review that much faster and similar to well-known tools like netstat.

Network analysis in XP/2003

Volatility is the best memory analysis tool we have found for finding comprehensive information on network connections and sockets within memory. It often finds connections that other tools do not. Its power lies in the ability to search memory two different ways. Similar to what we saw with the process plugins, XP-based network plugins have two different methods: the high-level method of following the standard linked lists and the brute-force method of scanning all of the memory for any references to the network artifacts The **`connections`** plugin follows the standard path: walk the TCP connections single-linked list and parse any structures it finds. This information gives us the network connections that were open when the memory image was acquired. The **`connscan`** plugin takes a more brute-force approach. Instead of finding the connections linked list, it simply searches memory for anything resembling a _TCPT_OBJECT and attempts to parse it. This approach can provide much more information, including old connection structures and any structures that were purposely unlinked by malware trying to hide. The only downside is that these remnant structures can be partially overwritten, occasionally providing unreliable information. A combination of both plugins gives the best of both worlds, allowing the analyst to discern which connections were alive and active (from the **`connections`** plugin) and which were previously terminated or unlinked (**`connscan`**).

Similar to the plugins for network connections, we also have two plugins to fully enumerate network sockets in XP/2003 systems. The **`sockets`** plugin walks the socket single-linked list and reports information from referenced socket structures. The **`sockscan`** plugin scans all of memory looking for _ADDRESS_OBJECT objects.



Network Artifacts: `netscan`

The **`netscan`** plugin combines both socket and TCP connection data into one output, making review that much faster and similar to well-known tools like netstat.

For each network artifact the following information is provided:

- Memory Offset
- Protocol
- Local IP Address
- Remote IP Address
- State (TCP only)
- Process ID (PID)
- Owner Process Name
- Creation Time (Sockets only)

Netscan can identify both established and closed TCP connections. The former indicate connections that were ongoing when the memory snapshot was taken. The latter can identify older terminated connections, including important finds like command and control (C2) communications.

When performing analysis, pay close attention to the process that owns the connection. Every socket and TCP connection is tied directly to a process. If you notice strange network activity from processes that should be only communicating in the local network (or not communicating at all), it can be powerful evidence that the process has been compromised.

One of the wonderful properties of sockets is they also store a creation time. Finding a process with a passive socket listening on a suspicious port can give us a reference time to search for other malicious activity around the time that socket was created. Also notice that unlike the connection structures, socket structures record information about any protocol (not just TCP).

Network Artifacts: netscan (2)

A process named spinlock.exe appears to be communicating to an external IP over port 443 (SSL)

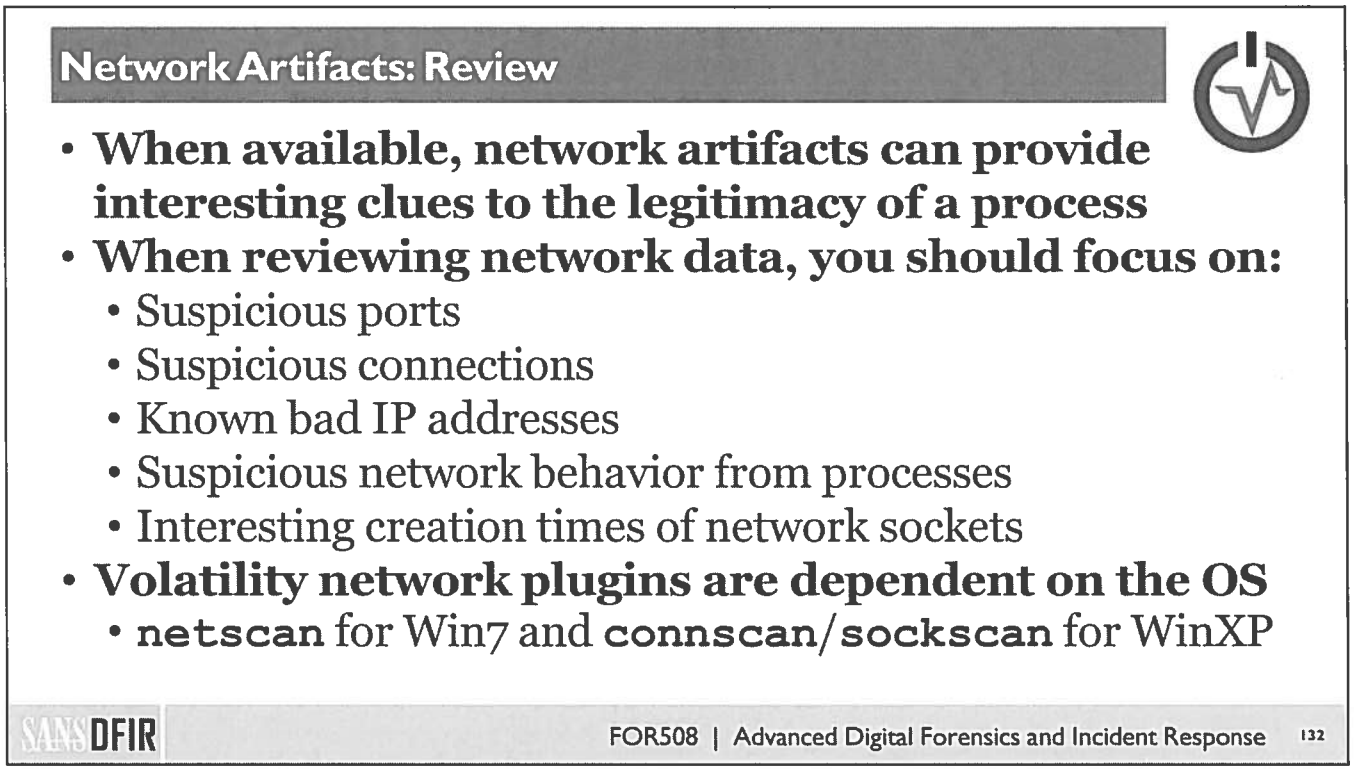
```
# vol.py -f win7-32-nromanoff-memory-raw.001 --profile=Win7SP1x86 netscan

Offset(P)     Proto    Local Address      Foreign Address      State          Pid      Owner          Created
0x4fb90e0     UDPv4    0.0.0.0:0          *:*                                 456      mfefire.exe    2012-04-04
0x4fb90e0     UDPv6    :::0               *:*                                 456      mfefire.exe    2012-04-04

0x7f837580    TCPv4    10.3.58.5:49805    10.3.58.9:445        ESTABLISHED    4        System
0x7f89a1d0    TCPv4    10.3.58.5:50817    199.73.28.114:443    CLOSED         1328     spinlock.exe
0x7f8c8008    TCPv4    10.3.58.5:62421    10.3.58.9:135        CLOSED         592      lsass.exe
0x7f8f8008    TCPv4    10.3.58.5:62333    10.3.16.5:443        CLOSED         7816     Skype.exe
0x7fa47008    TCPv4    -:62334            184.51.253.195:443   CLOSED         7816     Skype.exe
0x7fa559f8    TCPv4    -:62335            184.51.255.240:443   CLOSED         7816     Skype.exe
0x7fc6e318    TCPv4    10.3.58.5:3260     10.3.16.5:48351      ESTABLISHED    7776     f-response-ent
0x7fe9b278    TCPv4    10.3.58.5:62380    10.3.58.9:445        CLOSED         4        System
```


In this example we see some truncated output from the netscan plugin. Following our standard analysis process, we look for anomalous open sockets, suspicious TCP connections, and any processes that should not be communicating on the network.

Notice the spinlock.exe process is identified on the slide. There might be multiple reasons an analyst were to target that process. Its name alone might be suspicious as it is not a common executable to be found running on Windows systems. Further, it is communicating to an external IP over port 443 (SSL). This port is very commonly used for malicious outbound communication channels (in addition to plenty of legitimate traffic!) Those two facts together would cause a good analyst to dig a bit further to at least try and find more information on spinlock.exe. Another excellent next step would be to perform a WHOIS lookup on the external IP to see if that gives any clues to whether the network connection is legitimate.

There is much more to analyze in this output. As an example, is it normal for Skype to communicate via port 443? Should Skype be running on this system? Is that process really Skype, or malware named to blend in as Skype? The better you know normal network activity in the environment being investigated, the faster you can answer many of these questions.

Finally, notice that the TCP connections do not have "Created" timestamps, but the sockets listed (mfefire.exe) do. This is unfortunately to be expected. Sockets get creation timestamps, but the TCP data structure in memory does not record creation time.



Network Artifacts: Review

Volatility is extremely good at identifying network artifacts within memory images. This alone can be an excellent reason to learn Volatility because network artifacts are some of the easiest data points to interpret and use to find evil activity on a system. When reviewing network artifacts, you should be looking for:

- Suspicious ports
- Suspicious connections
- Known bad IP addresses
- Suspicious network behavior from processes
- Interesting creation times of network sockets

Remember that unlike almost every other Volatility plugin, the network collection is Windows operating system dependent. Worse, the plugins do not provide any error messaging when run using the wrong profile.

Bonus: Automating Analysis

malsysproc	Automatically identify suspicious system processes
openioc_scan	Scan memory objects using OpenIOC signature files
baseline	Provides three plugins for baseline comparisons:
processbl	Compare processes and loaded DLLs with a baseline image
servicebl	Compare services with a baseline image
driverbl	Compare drivers with a baseline image

Bonus: Automating Analysis

After teaching memory forensics in the 508 class for many years and seeing firsthand how painful sorting through the mountain of data that memory forensics provides the new analyst, we are thrilled to announce that there is a light at the end of the tunnel. There have recently been several third-party plugins for the framework that can greatly help automate analysis.

Malsysproc was written by Jared Atkinson and attempts to identify malware hiding as legitimate system processes.

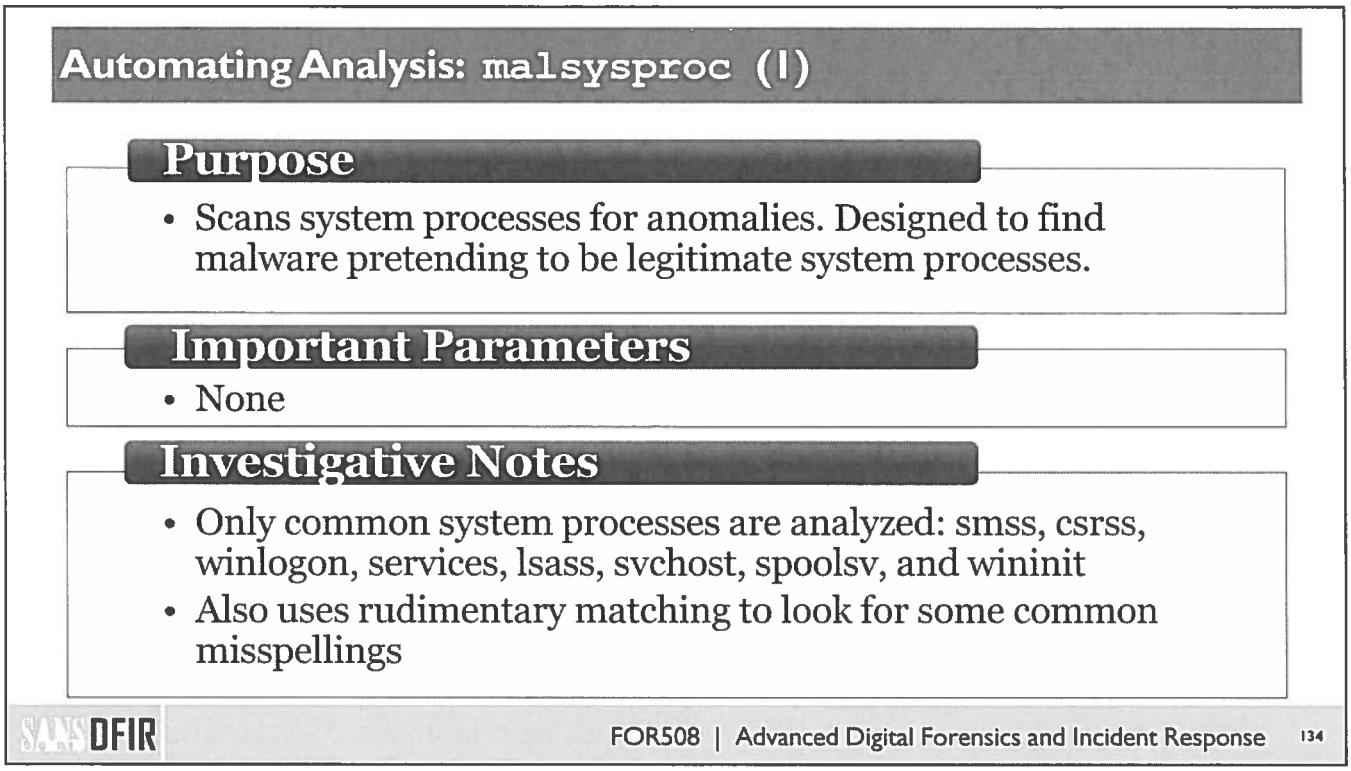
Openioc_scan was written by Takihiro Haruyama and uses the OpenIOC 1.1 format and is capable of performing more complicated indicator matching than even Redline! It supports a wide variety of different checks for processes, registry items, services, drivers, hooks, and file items.[1]

Finally, the baseline suite by Csaba Barta provides the unique capability to compare the current image with a memory image from a clean system of similar type. It provides an incredibly detailed comparison of processes and loaded DLLs (**processbl**), services (**servicebl**), and loaded modules (**driverbl**). [2]

The most exciting thing about these plugins is that they are only the start. We should expect to see even more powerful automation capabilities as developers use these plugins for inspiration.

[1] http://takahiroharuyama.github.io/blog/2014/08/15/fast-malware-triage-using-openioc-scan-volatility-plugin/

[2] https://github.com/csababarta/volatility_plugins/blob/master/baseline.py



Automating Analysis: `malsysproc (1)`

The **`malsysproc`** plugin was written by Jared Atkinson in an attempt to replicate Redline-style Malware Rating Index features in Volatility. The idea is simple: If analysts can identify very common things they look for in every memory image, those checks should be automated. In its first version, this plugin analyzes several common system processes (smss, csrss, winlogon, services, lsass, svchost, spoolsv, and wininit) for a variety of anomalies. Proper parent process, path, command line, execution time (at boot?), process priority, and proper number of processes with that name are all checked. Additionally, the plugin contains code to perform rudimentary matching looking for common misspellings of many of the processes. Results are printed in table form providing a visual way to detect suspicious processes. Remember to validate your findings! Windows is very complicated and some of things analysts look for are not necessarily absolute indicators of evil. For example, it is quite possible to see a svchost process started well after the system boot.

Because Volatilty plugins are written in Python, this script would be very easy to build upon in the future (should you do this, consider submitting your updated to the Github repository!)[1] One such fork of the plugin is named **`malprocfind`** and can be found in Csaba Barta's repository and in the SIFT.[2] **`Malprocfind`** adds useful checks for anomalous process SIDs and potential process hollowing.

[1] https://github.com/Invoke-IR/Volatility/blob/master/malsysproc.py

[2] https://github.com/csababarta/volatility_plugins/blob/master/malprocfind.py

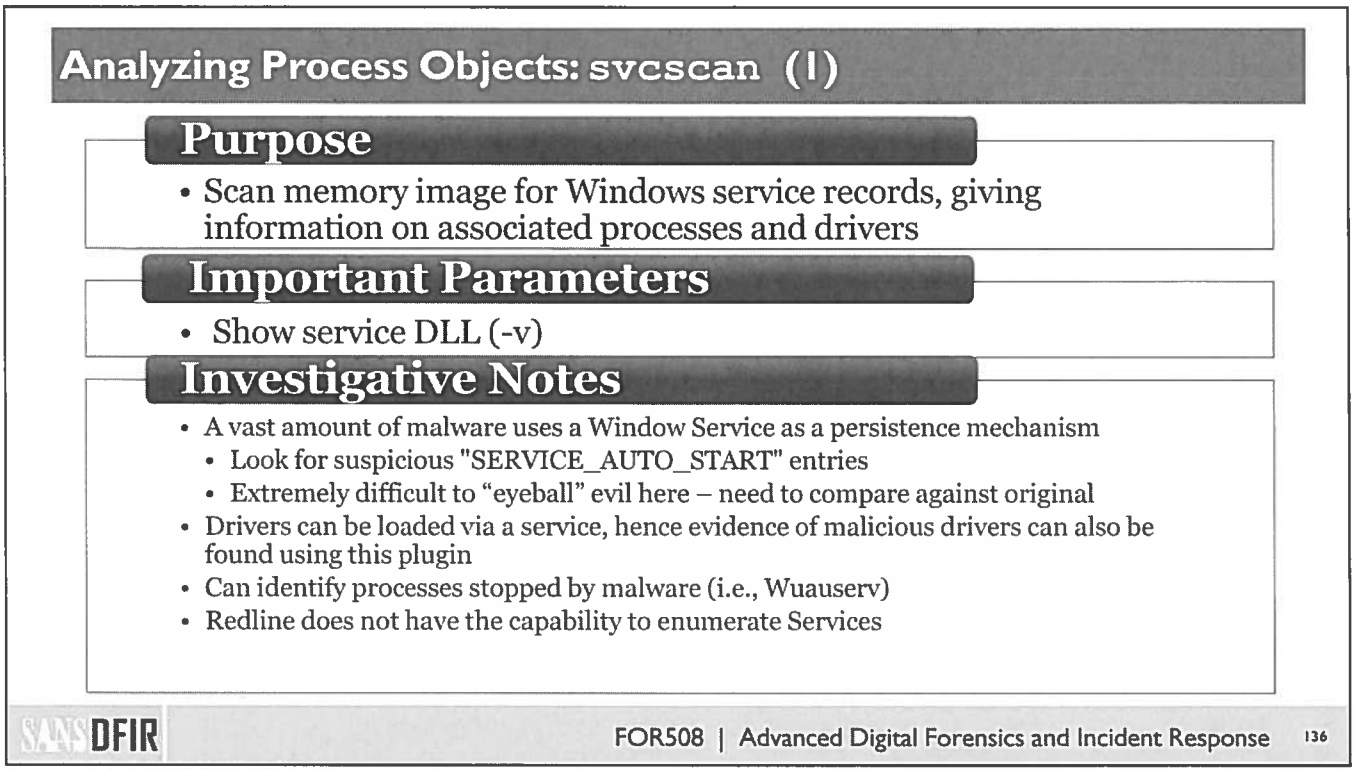
Automating Analysis: malsysproc (2)

```
root@siftworkstation:/# vol.py -f memory.img malsysproc

Offset     ProcessName  PID   Name  Path  PPId  Time  Priority  Cmdline Count
---------- ------------ ----- ----- ----- ----- ----- --------- ------- -----
0x8228cd08 smss.exe       528 True  True  True  True  True      True    True
0x823275a8 csrss.exe      576 True  True  True  True  True      True    True
0x822b6da0 winlogon.exe   600 True  True  True  True  True      True    True
0x8232cd10 services.exe   644 True  True  True  True  True      True    True
0x8232eb68 lsass.exe      656 True  True  True  True  True      True    True
0x8230bb58 svchost.exe    860 True  True  True  True  True      True    True
0x81ed0da0 svchost.exe    940 True  True  True  True  True      True    True
0x81dbd020 svchost.exe   1032 True  True  True  True  True      True    True
0x8211faf8 spoolsv.exe   1552 True  True  True  True  True      True    True
0x82095da0 scvhost.exe   1944 False False False False True      False   True
...Cmdline               : svchost
...Expected Parent Time  : 2009-08-07 02:36:00 UTC+0000
...Create Time           : 2009-08-07 02:37:06 UTC+0000
```

scvhost.exe is compared to legitimate svchost.exe

Automating Analysis: `malsysproc (2)`

Here, we see a memory image from a system infected with APT-like malware. The malware was named "scvhost" in an attempt to hide in plain sight. Luckily, this name variation was searched for by the plugin and identified as similar in name to a standard svchost process. Unfortunately for the malware, although it might be easy to miss the spelling anomaly, almost none of its properties match with what you would expect of a real svchost process (hence all of the False entries in the various columns).



Analyzing Process Objects `svcscan` (1)

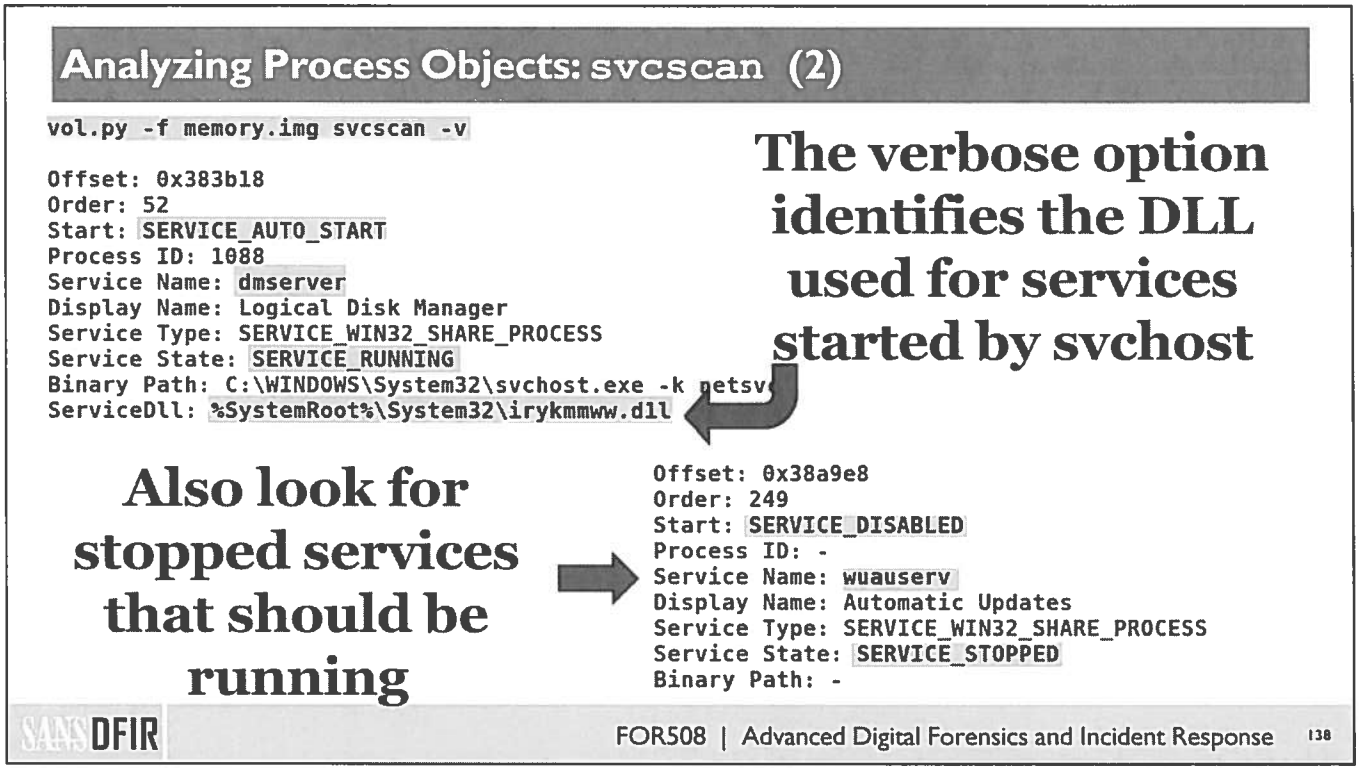
A Windows Service is a special type of process that is intended to be run in the background without user input. A large number of critical system processes are started (and stopped) via services. Services can load both process executables as well as system drivers. They are often used by malware to achieve persistence, or the ability to survive a reboot. Service information can be found in the Registry and in the Event Logs, but malware has been known to clean these areas. Thus, having an ability to review the status of services within a memory image is valuable. Volatility is currently the only memory analysis tool with this capability.

The **`svcscan`** plugin scans the memory image looking for service records. When it finds one, it parses the record data to provide the following:

- Offset
- Order
- Start method (Disabled | System_Start | Boot_Start | Auto_Start | Demand_Start)
- Process ID
- Service Name
- Display Name
- Type (Process | Driver)
- State (Running | Stopped)
- Full Path

If the verbose option (-v) is set, Volatility will find and parse the SYSTEM\CurrentControlSet\Services\<*servicename*>\Parameters\ServiceDLL registry key to identify what DLL was used to start the service. This can be very helpful for identifying malware attempting to use svchost as camouflage.

When reviewing the output, keep in mind that there might be other useful information other than the discovery of a malicious service. Malware often attempts to shut down critical system protection mechanisms in order to prevent things like Windows OS updating or new anti-virus definitions. Looking at the "Stopped" services could provide clues as to whether this has occurred.



Analyzing Process Objects: svcscan (2)

In the top example, a memory image was analyzed that was infected with malware using a service as its persistence mechanism. This particular malware used svchost to load a malicious DLL, further camouflaging the bad activity. While looking through the list of services, it might be difficult to pick out that a service named dmserver and started by svchost was evil. However, with the extra information provided by the verbose (-v) option of **svcscan**, we get more information on the service and can more easily recognize that something is amiss. Notice the file extension for the ServiceDLL is not "dll," but instead is d-one-l. Also note the Start value of SERVICE_AUTO_START. This indicates that the malware will run upon system boot.[1]

Looking at a different **svcscan** output on the bottom of the slide, we see that the wuauserv service was stopped when this memory image was created. Notice that because the service was not running, there is no path information for the loaded binary (because nothing was loaded). Although this is not an absolute indicator of malicious activity, it is unusual. Most modern Windows installations have automatic updates on by default. Some malware will attempt to turn off auto-updates to make sure that future updates do not patch whatever system vulnerabilities they are reliant upon. The takeaway is that as an analyst, you should look at both what was running and what wasn't running to find clues to abnormal system behavior.

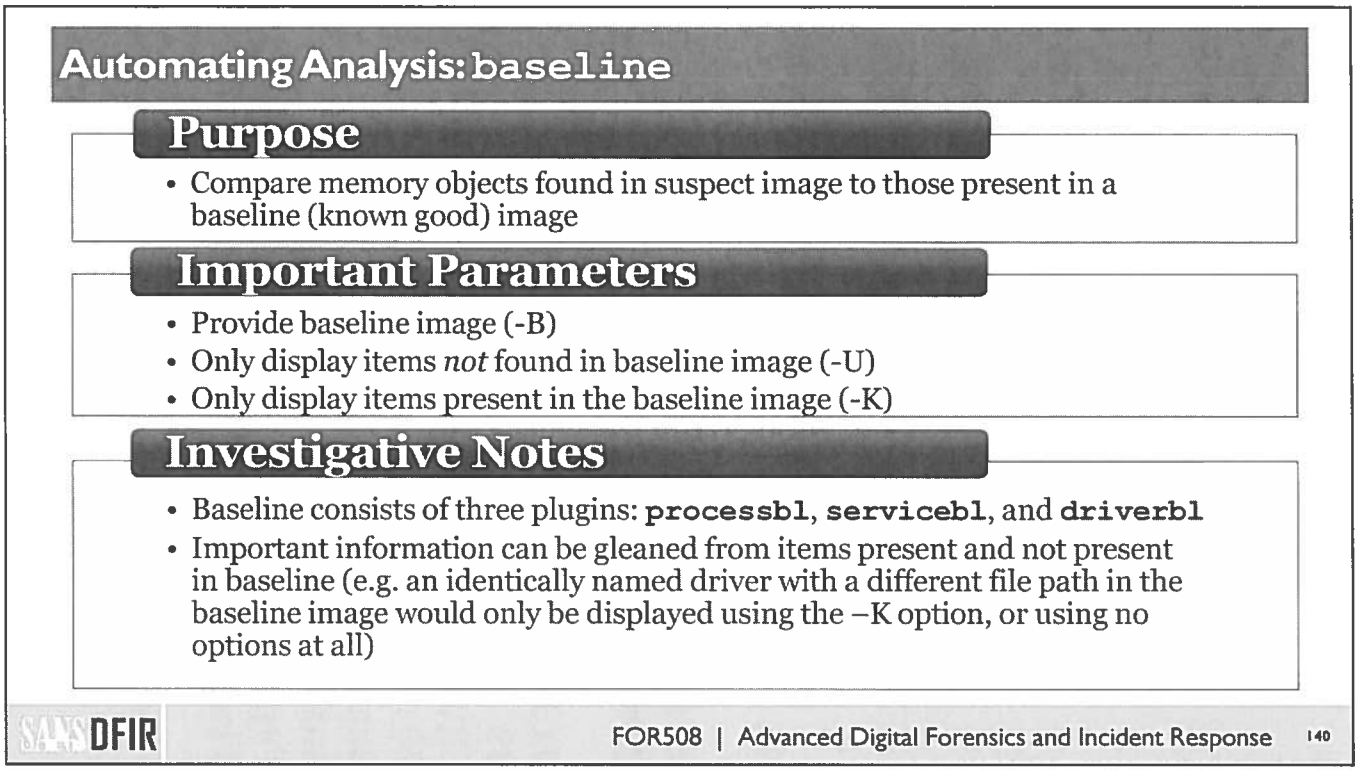
[1] CreateService function - http://msdn.microsoft.com/en-us/library/windows/desktop/ms682450.aspx

```
vol.py -f memory.img svcscan -v

Offset: 0x383b18
Order: 52
Start: SERVICE_AUTO_START
Process ID: 1088
Service Name: dmserver
Display Name: Logical Disk Manager
Service Type: SERVICE_WIN32_SHARE_PROCESS
Service State: SERVICE_RUNNING
Binary Path: C:\WINDOWS\System32\svchost.exe -k netsvcs
ServiceDll: %SystemRoot%\System32\irykmmww.dll
```

The verbose option identifies the DLL used for services started by svchost

Also look for stopped services that should be running

```
Offset: 0x38a9e8
Order: 249
Start: SERVICE_DEMAND_START
Process ID: 1088
Service Name: wuauserv
Display Name: Automatic Updates
Service Type: SERVICE_WIN32_SHARE_PROCESS
Service State: SERVICE_STOPPED
```



Automating Analysis: `baseline`

The baseline plugin was written by Csaba Barta and provides a very exciting new capability.[1] Csaba took the idea of comparing data with a baseline and extended it to the field of memory forensics. As we have learned, memory can contain tens of processes each with hundreds of process objects along with hundreds of drivers and services. Even a seasoned pro will often miss malware hiding amidst the thousands of items that must be reviewed. However, if the analyst has a baseline from a clean system of a similar type, many of the "known good" items can be quickly filtered. To accomplish this feat, Csaba created a capability to feed Volatility two memory images: your suspect image and a baseline image to compare. Further, he wrote a total of three initial plugins to perform comparisons. **`Processbl`** compares processes and their loaded DLLs (using data objects similar to **`pslist`** and **`ldrmodules`**). **`Servicebl`** compares services using the **`svcscan`** components, and **`driverbl`** compares loaded drivers with **`driverscan`** functionality.

Each plugin attempts to match similarly named objects across both images and then display their details in table form that can be quickly compared for anomalies. **`Processbl`** matches Image Name, **`servicebl`** matches Service Name, and **`driverbl`** matches Service Key Name. If a match is made, that item is grouped in the "known" category (only known entries can be displayed with the –K option). If no match is found, the object is placed in the "unknown" category (provided using the –U option). Both categories can provide valuable information and one way to make sure you don't miss anything is to not use either option, effectively showing both knowns and unknowns in the same view (or running the plugin once with each option). As an example, although it is certainly useful to known that a driver is present in the suspect image that was not in the baseline (unknown category), it might be equally interesting to know that a driver with the same name was found in both images, but it has a different path in the suspect image (known category). The plugins provide a lot of data and it takes practice to use it effectively!

[1] https://github.com/csababarta/volatility_plugins/blob/master/baseline.py

Automating Analysis: servicebl

```
root@siftworkstation:/# vol.py -f memory.img svcscan | grep "Service Name" | wc -l
252

root@siftworkstation:/# vol.py servicebl -f memory.img -B baseline.img -U

Offset     Serice Name                    PID   Found DName Path  Type  State
---------- ------------------------------ ----- ----- ----- ----- ----- -----
0x00383d58 EapHost                        ----- False False False False False
0x00384680 hkmsvc                         ----- False False False False False
0x00384a60 idsvc                          ----- False False False False False
0x00385eb0 napagent                       ----- False False False False False
0x00386568 NetTcpPortSharing              ----- False False False False False
0x00389a10 usbehci                        ----- False False False False False
0x0038ab98 irykmmww                       ----- False False False False False
```

Out of 252 services identified, seven were not in the baseline image (one service was malicious)

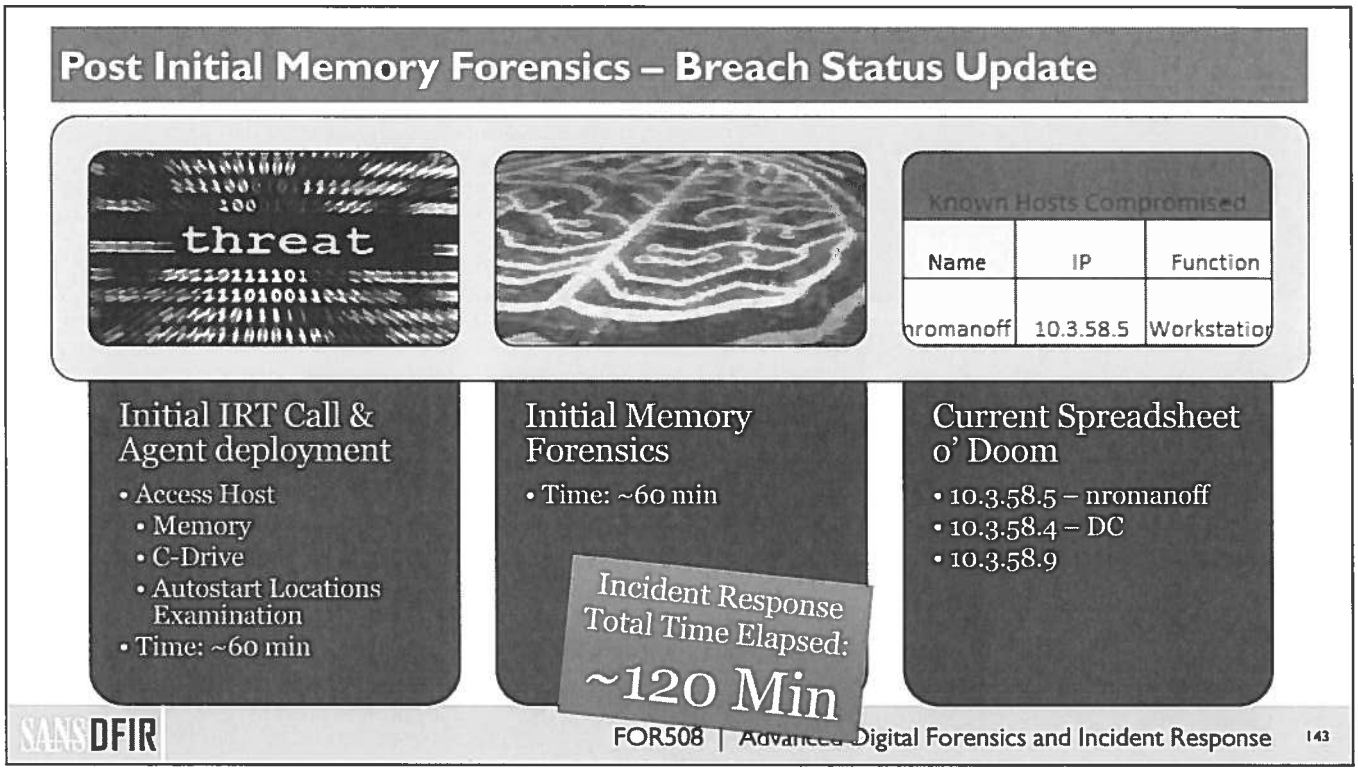
Automating Analysis: `servicebl`

In this example, a quick count of the total services found in the memory image was taken using the **`svcscan`** plugin and the **`wc`** command (the result was 252). Subsequently, **`servicebl`** was run and provided the suspect image (memory.img) and the known good image (baseline.img). In this case, we were looking for services that existed in the suspect image but not in the baseline image so the –U option was provided to only show unknown services. Of the seven services identified as "unknown," one turned out to be malicious—irykmmww. Keep in mind that although using a baseline image can greatly cut down on the amount of data necessary to analyze, there will still be false positives that must be eliminated! In this case, we ran **`svcscan`** on the suspect image and looked at each of the seven processes. Interestingly, **`svcscan`** showed that of the seven, only irykmmww was started (running), whereas the remaining six services were stopped (not running). This lead us to dump the irykmmww service dll and further analysis identified it as in fact malicious.

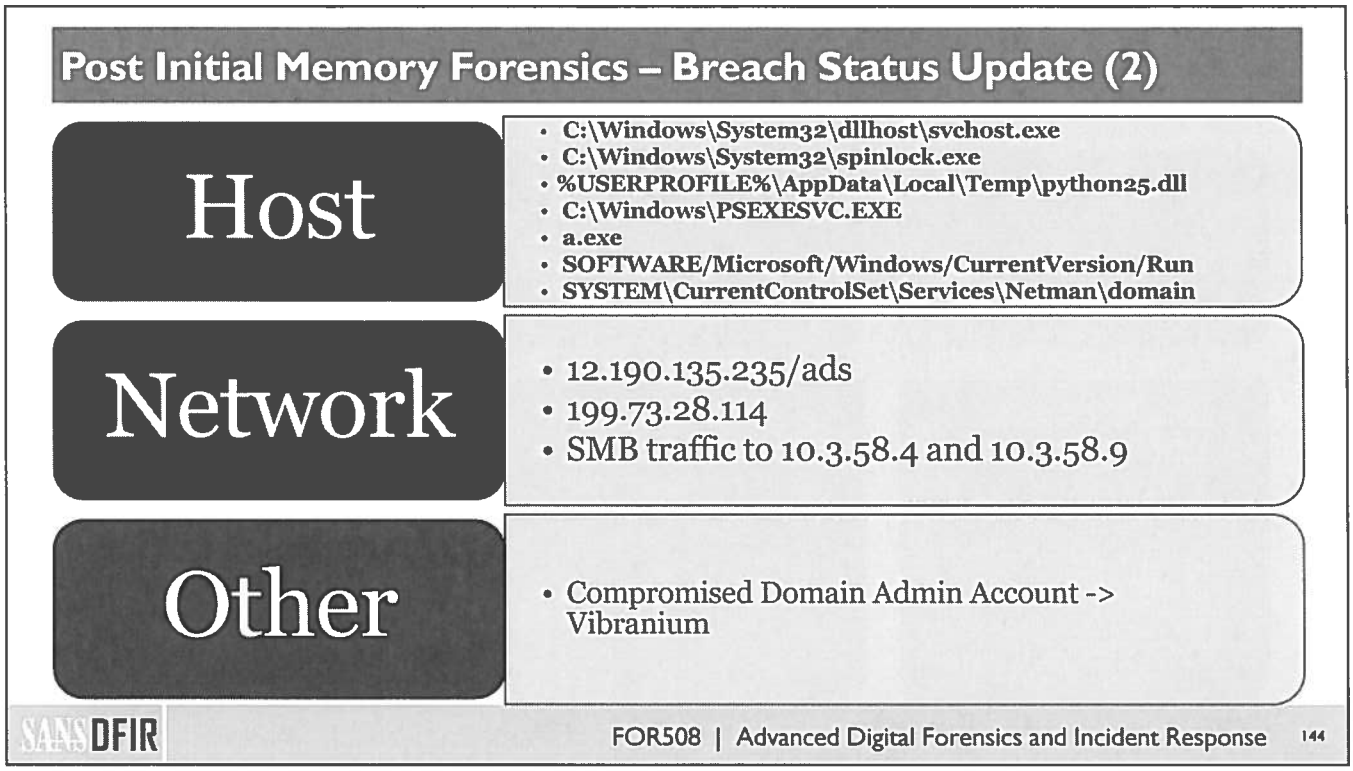
Exercise 2.3

Volatility Memory Analysis

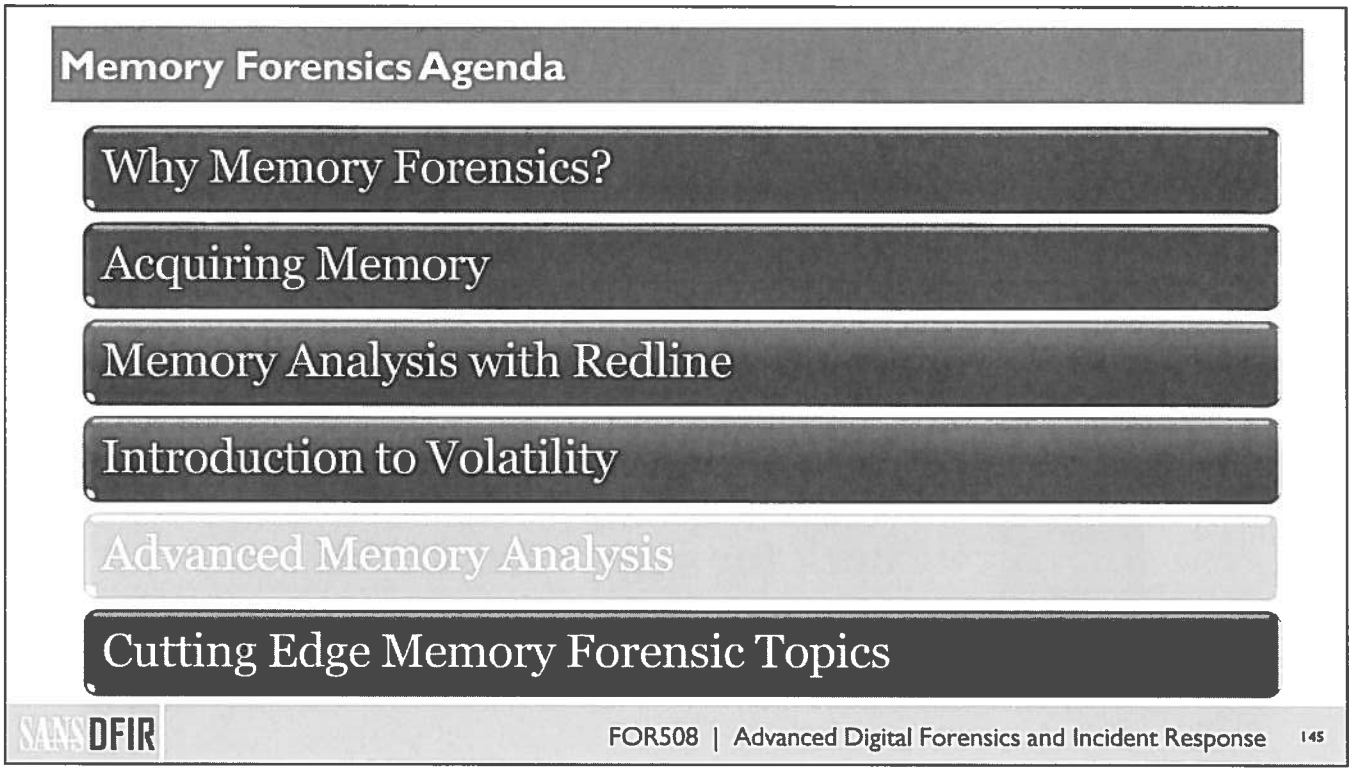
This page intentionally left blank.



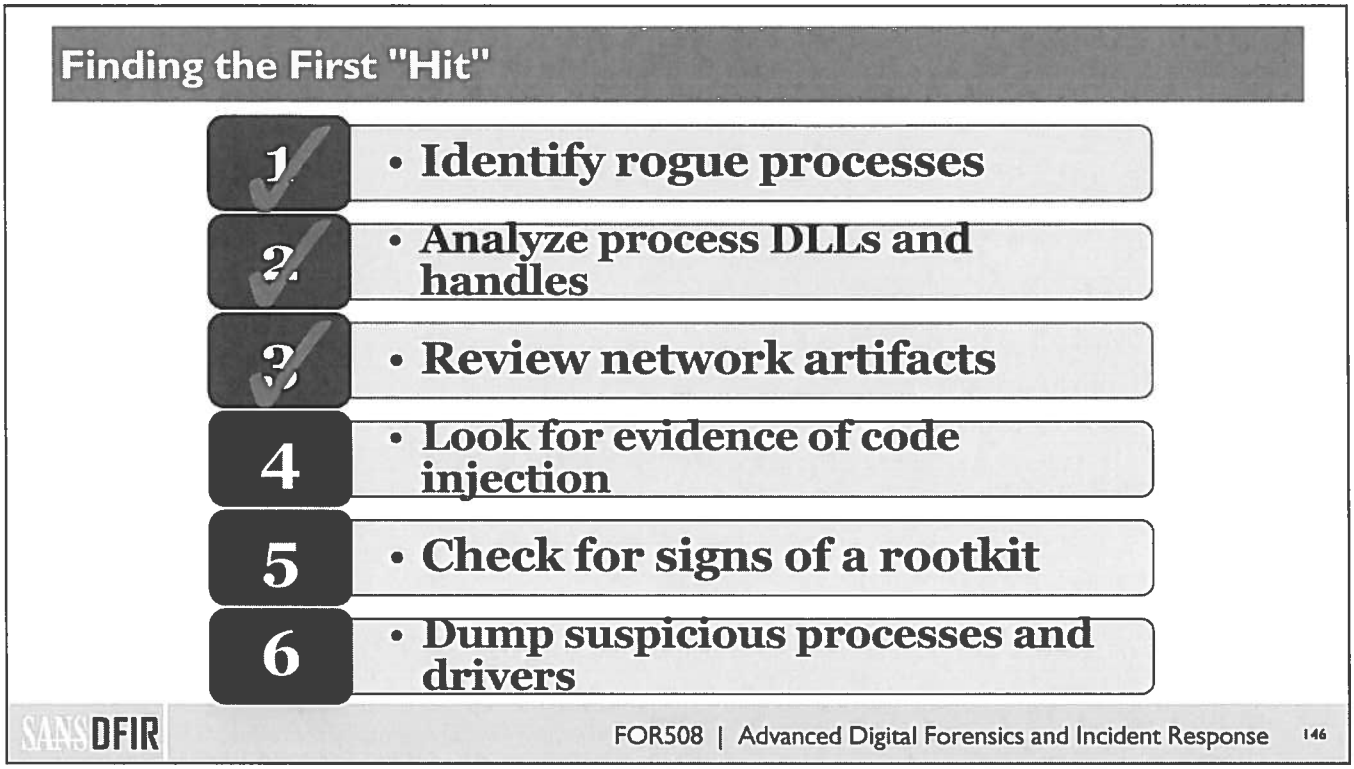
This page intentionally left blank.



This page intentionally left blank.



This page intentionally left blank.



Finding the First "Hit"

So far, we have covered the first three steps of our process in both Redline and Volatility. Now, it's time to dig deeper into more advanced artifacts like evidence of code injection, rootkit detection, and file extraction.

Step 4:

Look for Code Injection

This page intentionally left blank.

Detecting Injection

- DLL injection is *very* common with modern malware
  - VirtualAllocEx( ) and CreateRemoteThread( )
  - SetWindowsHookEx( )
- Process hollowing is another injection technique
  - Malware starts a new instance of legitimate process
  - Original process code de-allocated and replaced
  - Retains DLLs, handles, data, etc., from original process
- **Code injection is relatively easy to detect**
  - Review memory sections marked as `Page_Execute_ReadWrite` and having no memory-mapped file present
    - Scan for DLLs (PE files) and shellcode
  - Process image not backed with file on disk = process hollowing

Detecting Injection

Code injection has become a staple for modern malware. Some of the most current and famous malware around make heavy use of code injection. It is one of the more advanced ways that malware tries to hide. Although code injection is a really good way to hide on a running system, it is quite easy to identify when doing memory analysis.

The two most popular forms of code injections are DLL injection and process hollowing. Unfortunately, the Windows architecture makes DLL injection relatively trivial. The only hurdle is having administrator (or debug privileges) on the system. There are many forms of DLL injection. The attacking process can allocate space in a running process, shove the DLL file into it, and then create a new thread to load the DLL into the running process using the Windows VirtualAllocEx() and CreateRemoteThread() function calls. Alternatively, the attacking process can force a running process to load a malicious DLL by hooking its filter functions using the SetWindowsHookEx() function. Regardless of the technique, during memory analysis, they all look the same: An unnamed memory section containing executable code is attached to victim process.

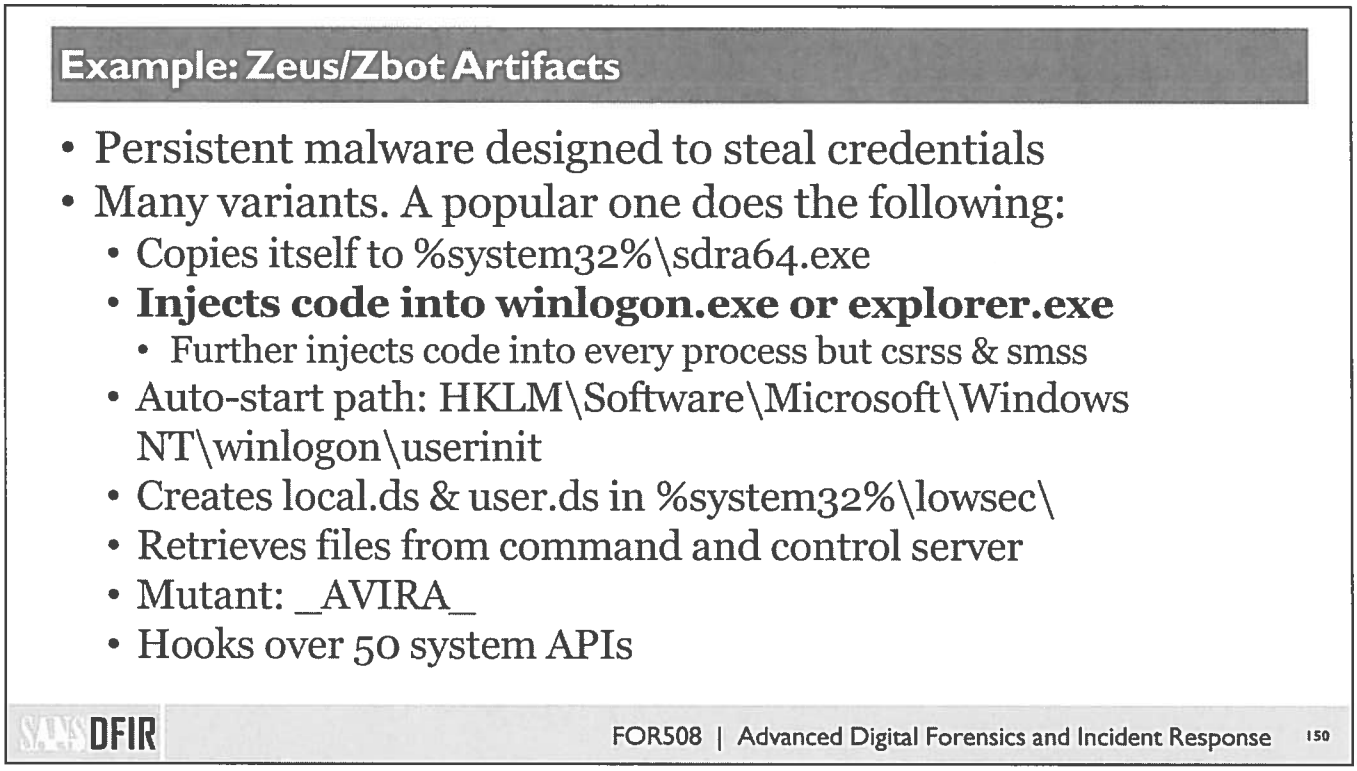
Process hollowing is a little bit different, but looks very similar during memory analysis. In this case, the malware starts another copy of a legitimate system process. It pauses the process, de-allocates some of the original code and replaces it with malicious code. Items like the process image name, path, and command lines remain unchanged and serve to camouflage the now malicious process.[1]

Because these methods circumvent the standard mechanisms for processes to load code, they can be easily detected. The memory analysis tool simply has to follow the VAD tree of the process and review all of the memory sections belonging to that process. If it finds memory pages marked as executable (Page_Execute_ReadWrite) and if the file is unmapped, (for example, not backed with a file on disk), then there is a very good chance that it is an injected page. An additional sanity check is to determine whether a

Portable Executable (DLL) file is present in that memory page. Process hollowing is even easier to detect. If the image file (process binary) is unmapped (not backed with a file on disk), then it is a strong indication that process hollowing has occurred.

This might all sound very complicated, but luckily our memory analysis tools do these checks for us and nicely present their results.

[1] http://blog.spiderlabs.com/2011/05/analyzing-malware-hollow-processes.html



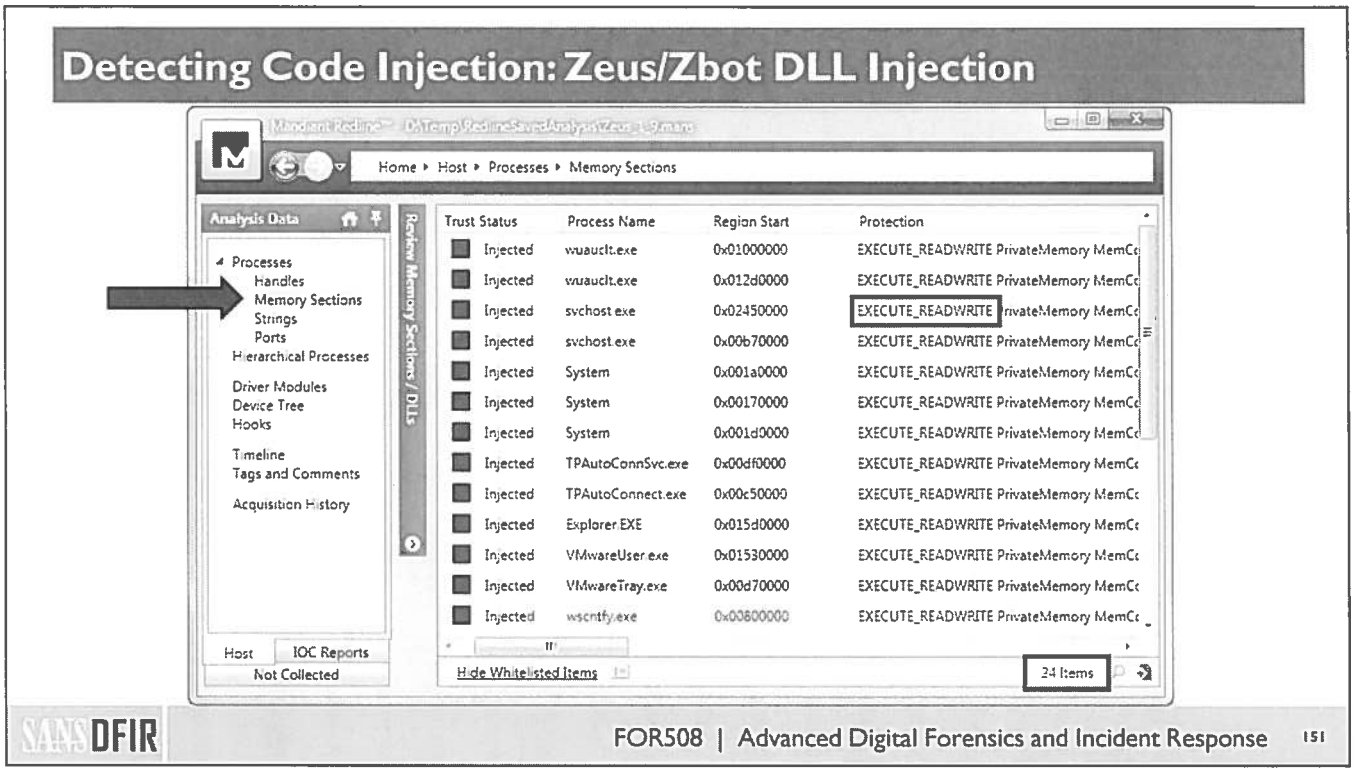
Example: Zeus/Zbot Artifacts

Zeus, or Zbot, malware provides a great opportunity to see code injection in progress. It is primarily a credential stealer, collecting things like protected storage, FTP, POP3, online credentials, and credit card and banking information. Zeus has proven to be exceedingly successful at maintaining a foothold on victim systems, but by doing so, it leaves a variety of artifacts on the system:

- Makes a copy of itself to %system32%\sdra64.exe
- **Injects code into winlogon.exe or explorer.exe**
  - Further injects code into every process but csrss and smss
- Auto-start path: HKLM\Software\Microsoft\Windows NT\winlogon\userinit
- Creates local.ds & user.ds in %sytem32%\lowsec\
- Retrieves files from command and control server
- Mutant: _AVIRA_
- Hooks over 50 system APIs

Zeus is anything but quiet! To a user, it appears virtually invisible, but when doing memory analysis it lights up like a Christmas tree. Now, we will take a look at how we would detect Zeus code injection.

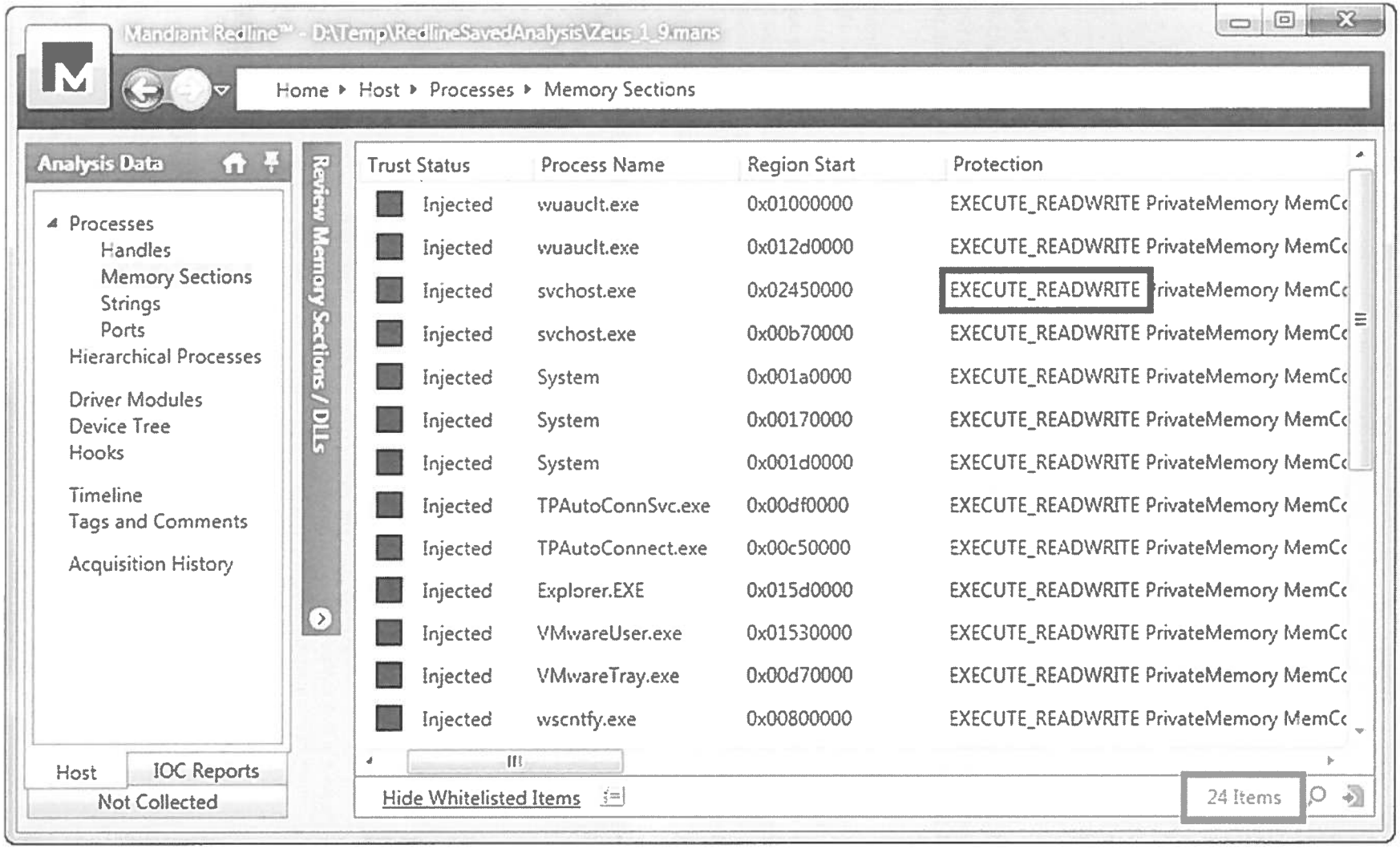
* Note: The artifacts discussed here are for a specific variant of Zeus. Many variants exist with different host-based artifacts.

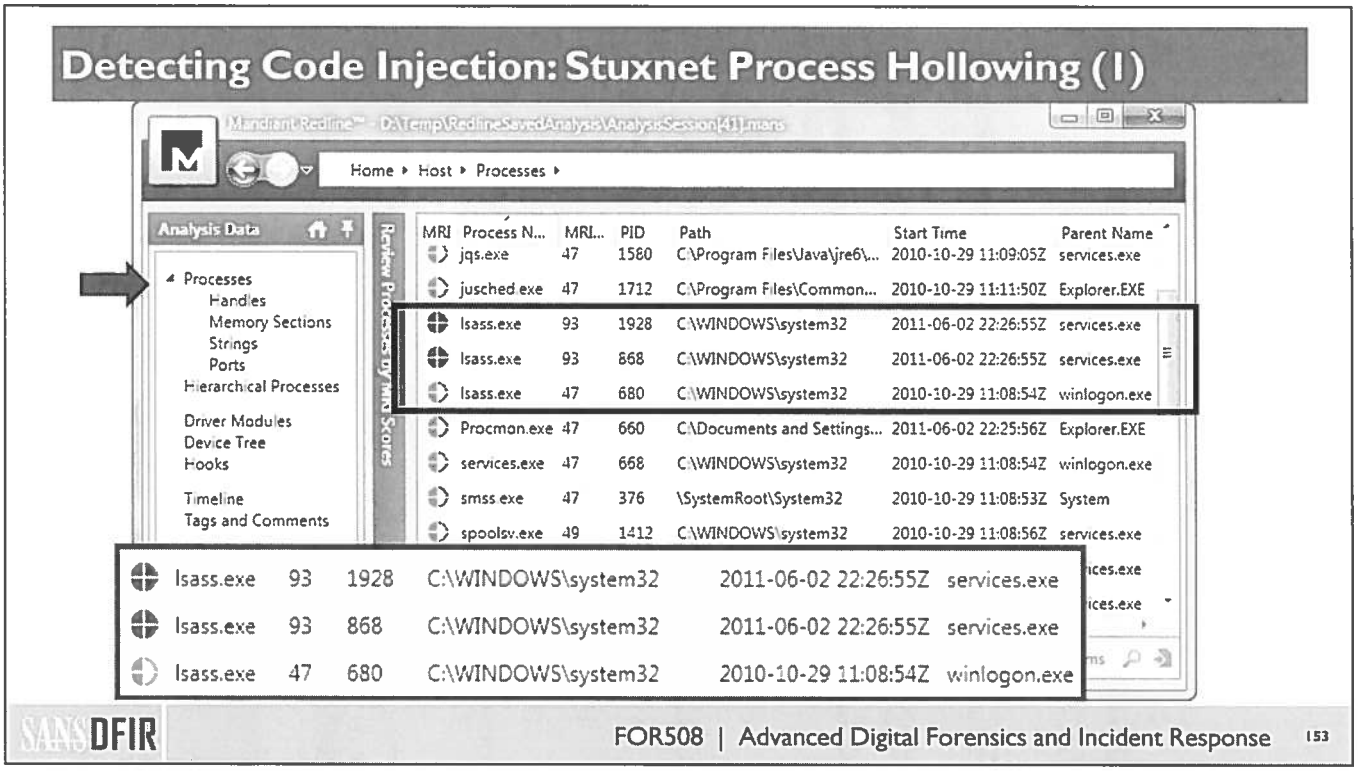


Detecting Code Injection: Zeus/Zbot DLL Injection

After loading a memory image from a system infected with Zeus into Redline, we navigate to the Memory Sections dialog of the Analysis Data panel. This view will show us memory sections for all processes and allow us to sort them by their "injected" status. As expected, we see a large number of memory sections that Redline believes to be injected. Redline will mark processes as injected if there is a memory section within the process that has no name yet begins with a standard MZ/PE header. Additionally, the 24 memory sections Redline identified all have Execute_ReadWrite privileges, which is a very strong sign that they are not legitimate.

If your aim was only to determine whether the system was compromised or not, finding 24 different memory sections with injected code in a variety of different processes should be very convincing! To look further into an injected memory section, double-click on that section, or click **show details** at the bottom right of the interface.


Mandiant Redline™ - D:\Temp\RedlineSavedAnalysis\Zeus_1_9.mans
Home ▸ Host ▸ Processes ▸ Memory Sections
Analysis Data
Processes
Handles
Memory Sections
Strings
Ports
Hierarchical Processes
Driver Modules
Device Tree
Hooks
Timeline
Tags and Comments
Acquisition History
Review Memory Sections / DLLs
Host
IOC Reports
Not Collected
Trust Status
Process Name
Region Start
Protection
Injected wuauclt.exe 0x01000000 EXECUTE_READWRITE PrivateMemory MemC
Injected wuauclt.exe 0x012d0000 EXECUTE_READWRITE PrivateMemory MemC
Injected svchost.exe 0x02450000 EXECUTE_READWRITE rivateMemory MemC
Injected svchost.exe 0x00b70000 EXECUTE_READWRITE PrivateMemory MemC
Injected System 0x001a0000 EXECUTE_READWRITE PrivateMemory MemC
Injected System 0x00170000 EXECUTE_READWRITE PrivateMemory MemC
Injected System 0x001d0000 EXECUTE_READWRITE PrivateMemory MemC
Injected TPAutoConnSvc.exe 0x00df0000 EXECUTE_READWRITE PrivateMemory MemC
Injected TPAutoConnect.exe 0x00c50000 EXECUTE_READWRITE PrivateMemory MemC
Injected Explorer.EXE 0x015d0000 EXECUTE_READWRITE PrivateMemory MemC
Injected VMwareUser.exe 0x01530000 EXECUTE_READWRITE PrivateMemory MemC
Injected VMwareTray.exe 0x00d70000 EXECUTE_READWRITE PrivateMemory MemC
Injected wscntfy.exe 0x00800000 EXECUTE_READWRITE PrivateMemory MemC
Hide Whitelisted Items
24 Items

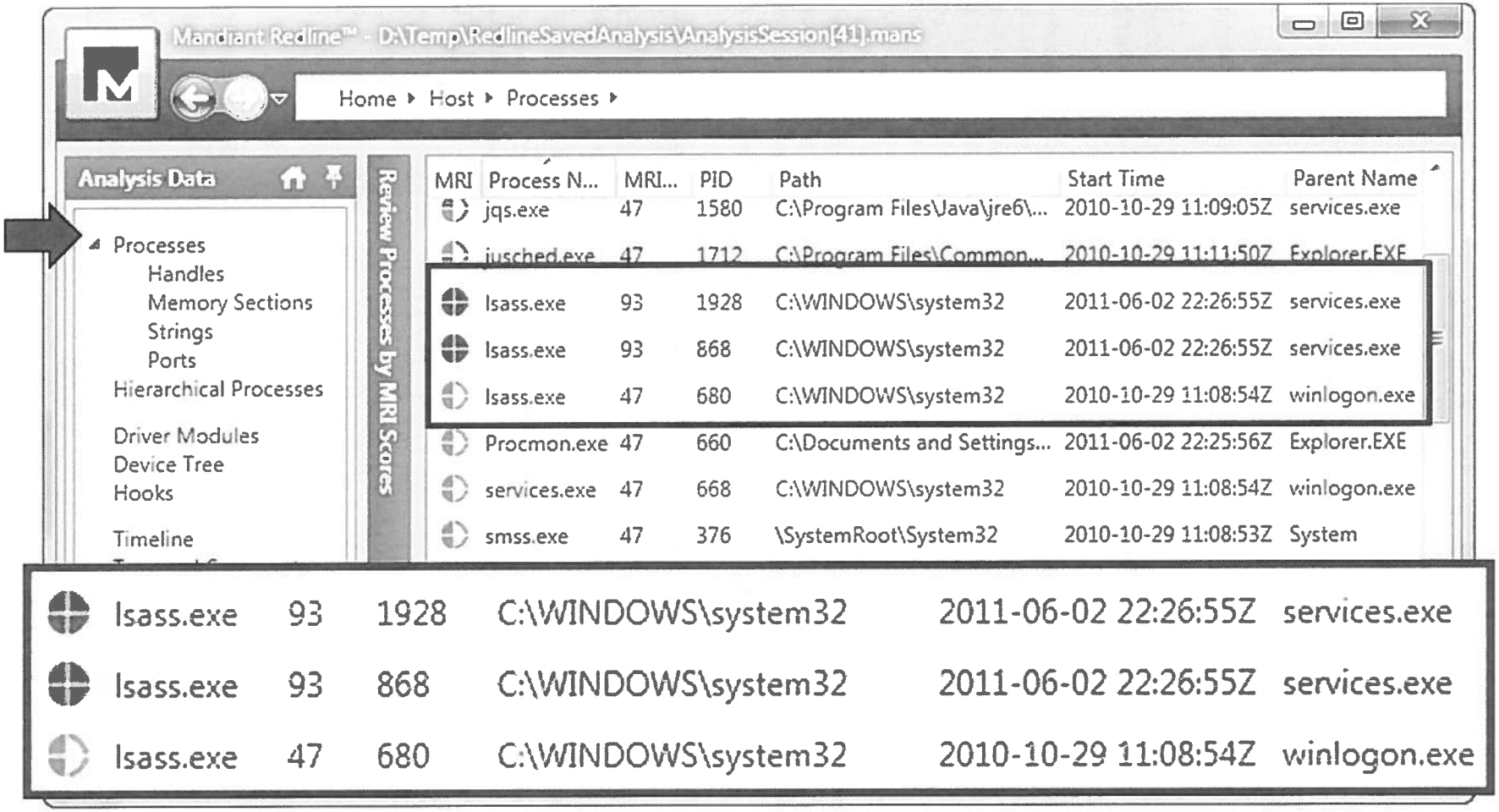


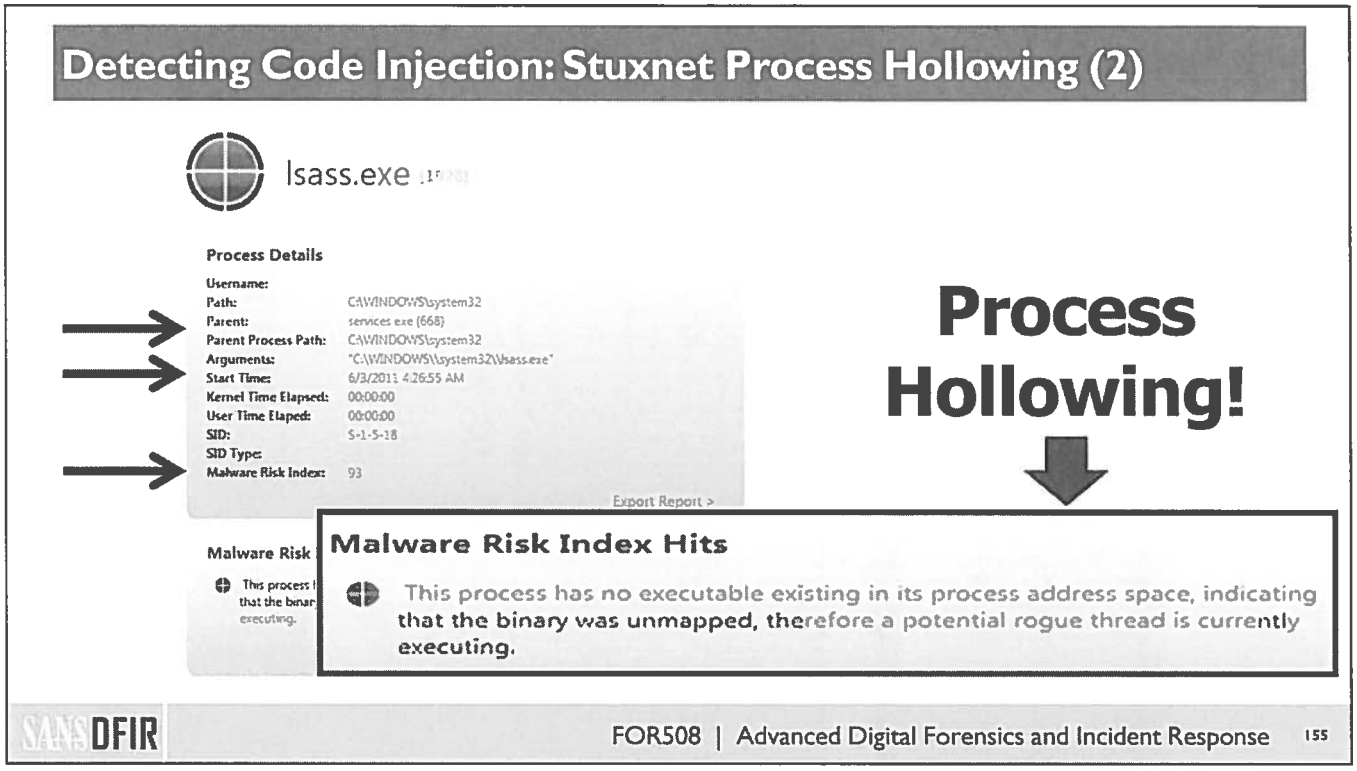
Detecting Code Injection: Stuxnet Process Hollowing (1)

Stuxnet is a little more subtle than Zeus, but still quite blatant. In this example, we will use Stuxnet to demonstrate how process hollowing looks within Redline. Similar to DLL injection, process hollowing is relatively simple to detect during memory analysis. The most obvious clue we see here is three lsass.exe processes running. The Local Security Authority Subsystem Service (lsass.exe) is a critical system process, but there should be only one instance running. Redline makes it easy to determine which of the three lsass.exe processes are illegitimate—it assigns a Malware Rating Index value of 93 to two of them. Looking deeper, you might also notice some other differences between the processes that can help you make that determination. The parent of the legitimate lsass.exe process is winlogon.exe (you could check this on an uninfected system using something like SysInternals Process Explorer).[1] Notice the other lsass.exe processes have services.exe listed as their parents. The start times of the processes are also an excellent clue. Lsass.exe should be started very near boot time, whereas the two outliers were started a very long time after the original copy.

The Stuxnet memory image used comes from a public sample hosted at https://code.google.com/p/volatility/wiki/SampleMemoryImages. It is provided on your course USB drive.

[1] http://technet.microsoft.com/en-us/sysinternals/bb896653


Mandiant Redline™ - D:\Temp\RedlineSavedAnalysis\AnalysisSession[41].mans
Home ▸ Host ▸ Processes ▸
Analysis Data
Processes
Handles
Memory Sections
Strings
Ports
Hierarchical Processes
Driver Modules
Device Tree
Hooks
Timeline
Review Processes by MRI Scores
MRI Process N... MRI... PID Path Start Time Parent Name
jqs.exe 47 1580 C:\Program Files\Java\jre6\... 2010-10-29 11:09:05Z services.exe
jusched.exe 47 1712 C:\Program Files\Common... 2010-10-29 11:11:50Z Explorer.EXE
lsass.exe 93 1928 C:\WINDOWS\system32 2011-06-02 22:26:55Z services.exe
lsass.exe 93 868 C:\WINDOWS\system32 2011-06-02 22:26:55Z services.exe
lsass.exe 47 680 C:\WINDOWS\system32 2010-10-29 11:08:54Z winlogon.exe
Procmon.exe 47 660 C:\Documents and Settings... 2011-06-02 22:25:56Z Explorer.EXE
services.exe 47 668 C:\WINDOWS\system32 2010-10-29 11:08:54Z winlogon.exe
smss.exe 47 376 \SystemRoot\System32 2010-10-29 11:08:53Z System
lsass.exe 93 1928 C:\WINDOWS\system32 2011-06-02 22:26:55Z services.exe
lsass.exe 93 868 C:\WINDOWS\system32 2011-06-02 22:26:55Z services.exe
lsass.exe 47 680 C:\WINDOWS\system32 2010-10-29 11:08:54Z winlogon.exe



Detecting Code Injection: Stuxnet Process Hollowing (2)

Taking a deeper look at one of the malicious lsass.exe processes, we again see suspicious parent and start time information in addition to that very high (93/100) MRI rating. Redline provides the following justification for the rating:

"This process has no executable existing in its process address space, indicating that the binary was unmapped, therefore a potential rogue thread is currently executing."

That sounds extremely similar to our definition of process hollowing! And in fact, reverse engineers have confirmed that the Stuxnet worm does in fact employ process hollowing of lsass.exe processes.[1]

[1] http://blogs.technet.com/b/markrussinovich/archive/2011/03/30/3416253.aspx

lsass.exe (1928)

Process Details

Username:	
Path:	C:\WINDOWS\system32
Parent:	services.exe (668)
Parent Process Path:	C:\WINDOWS\system32
Arguments:	"C:\WINDOWS\\system32\\lsass.exe"
Start Time:	6/3/2011 4:26:55 AM
Kernel Time Elapsed:	00:00:00
User Time Elaped:	00:00:00
SID:	S-1-5-18
SID Type:	
Malware Risk Index:	93

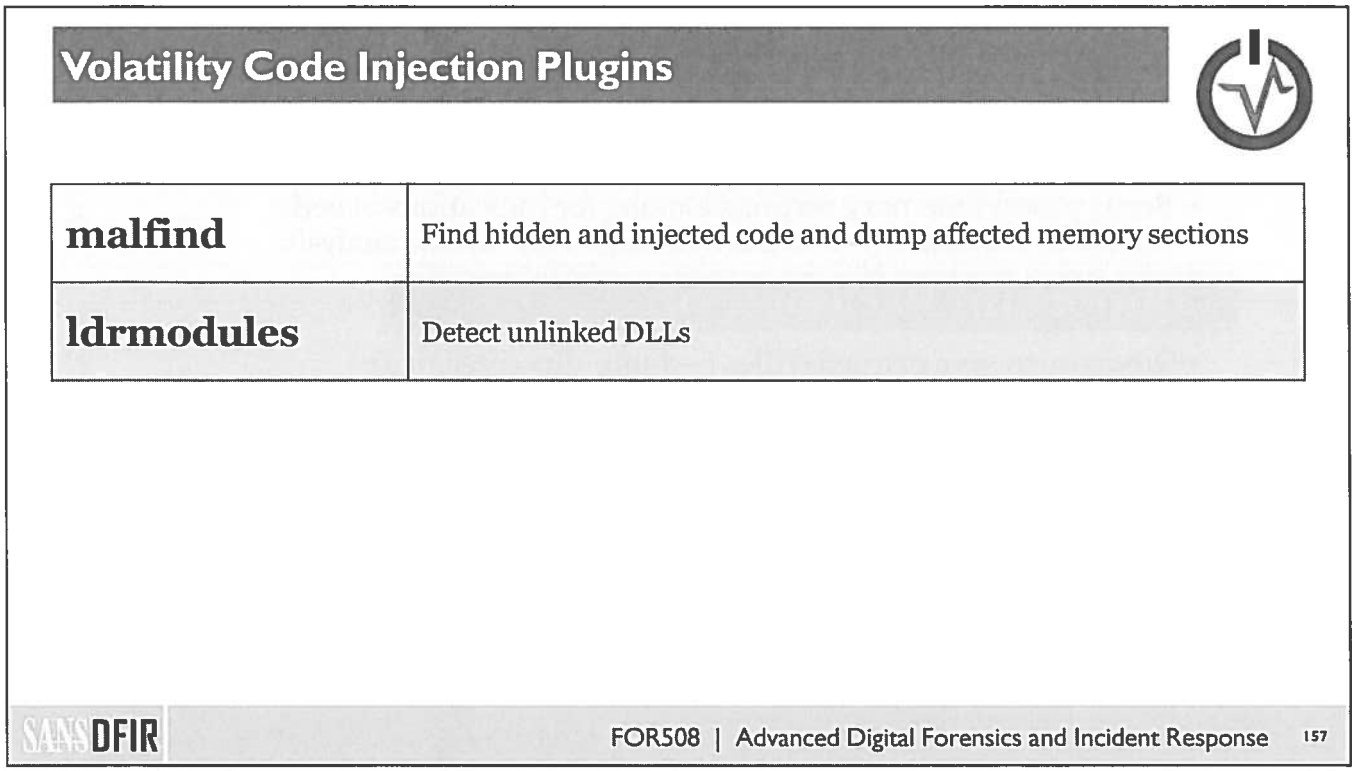
Malware Risk I

This process h
that the binary
executing.

Malware Risk Index Hits

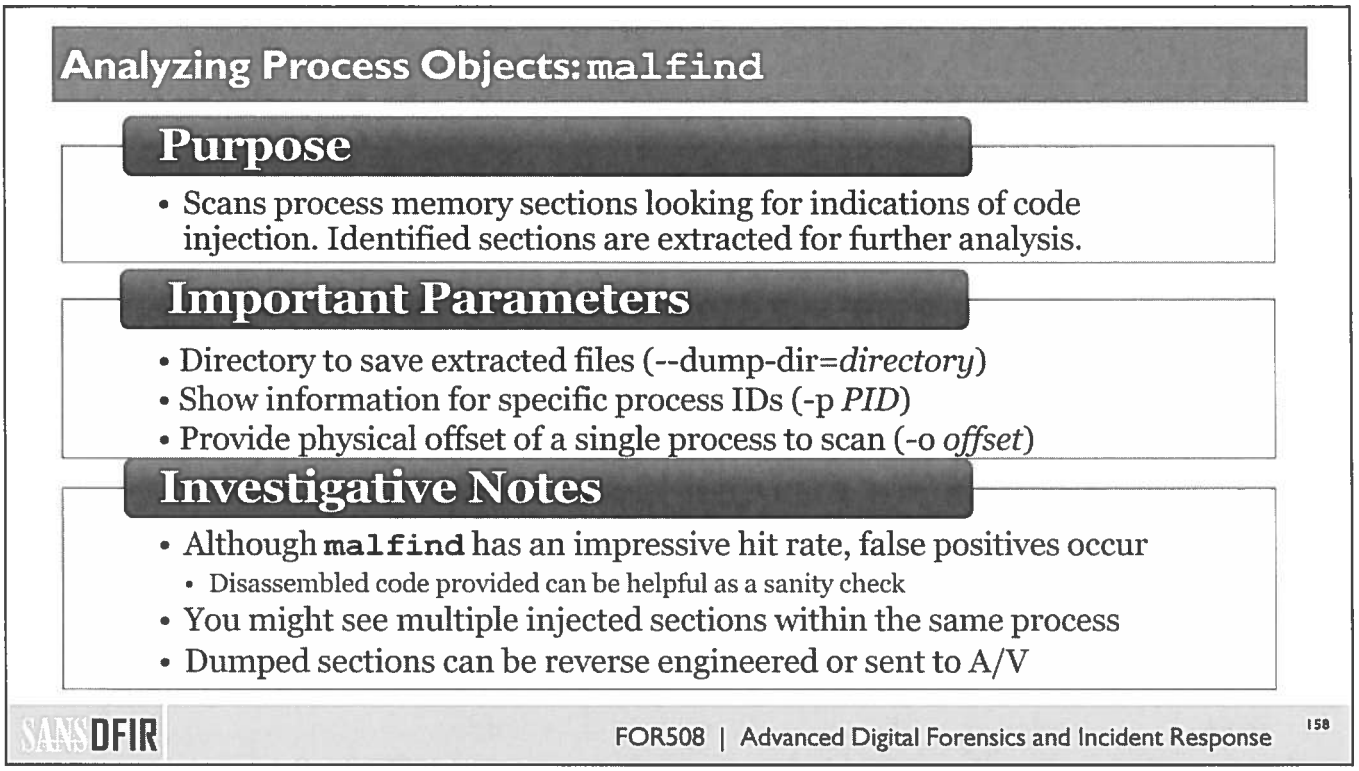
This process has no executable existing in its process address space, indicating that the binary was unmapped, therefore a potential rogue thread is currently executing.

Process Hollowing!



Volatility Code Injection Plugins

The number of Volatility plugins we will cover for code injection in no way indicates how important this step is during memory analysis. It turns out that one super-plugin, **`malfind`**, performs nearly all of the functions typically required to look for injection attacks. It is arguably one of the most important plugins in the Volatility arsenal. In addition to this, we add the plugin **`ldrmodules`** to detect unlinked DLLs, a common occurrence with more advanced malware.

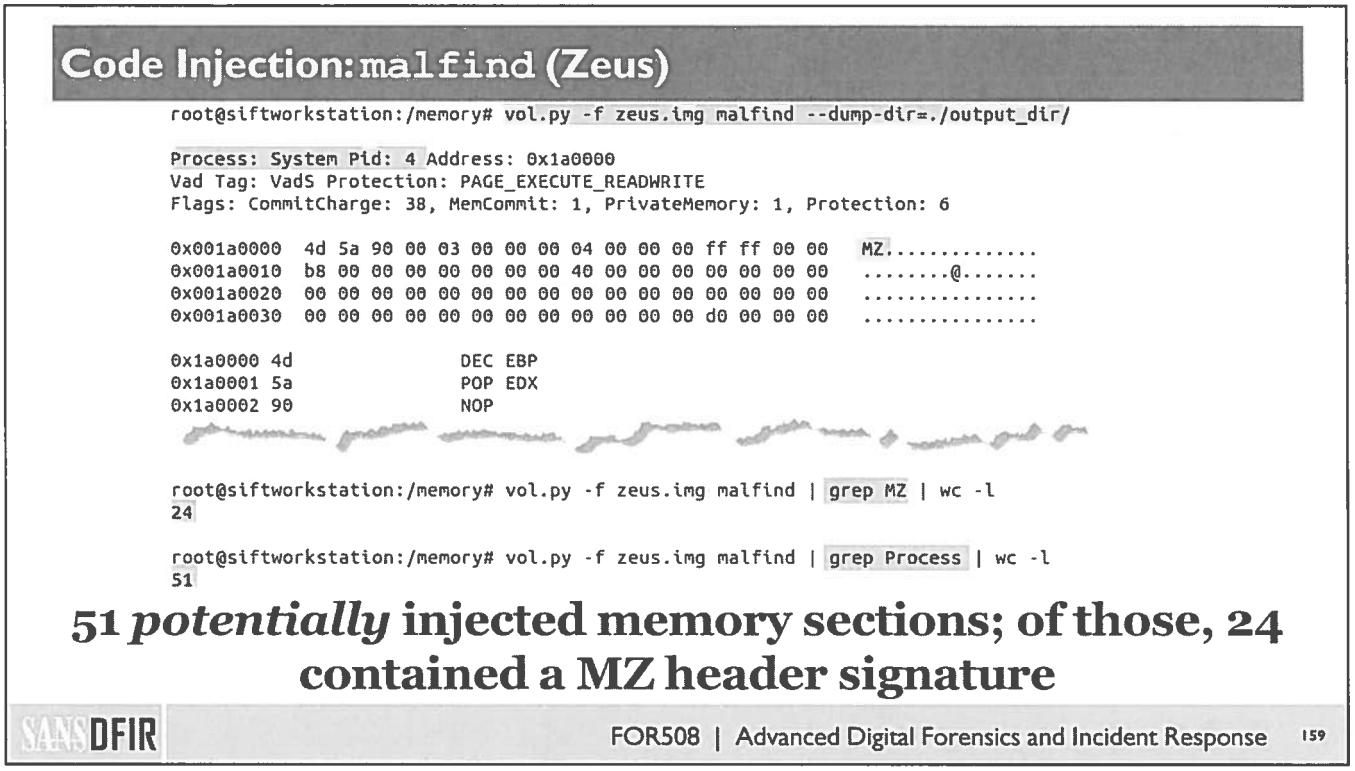


Analyzing Process Objects: `malfind`

`Malfind` is one of the most impressive plugins in the Volatility suite. Its creator, Michael Hale Ligh of Malware Analyst's Cookbook fame, realized that injected code tends to follow some very consistent rules when stored in memory—namely, memory sections marked as "executable" and not including an associated mapped file on disk. A further check is that these sections must contain legitimate code—**`malfind`** leaves this final check up to the analyst. For each process, **`malfind`** scans the assigned memory sections and subjects them to these tests. When a hit is found, it is output to standard out and the section is written to the "dump" directory specified by the user. The sections are named according to the parent process name, PID, and starting virtual address offset. These sections can then be loaded into a disassembler like IDA Pro, or scanned with anti-malware signatures. For each finding, the following information is presented:

- Name (Process Name)
- PID (Process ID)
- Start (Starting Offset)
- End (Ending Offset)
- Tag (Pool tag indicating type of memory section)
- Hits (Number of hits from YARA signatures)
- Protect (Memory section permissions)

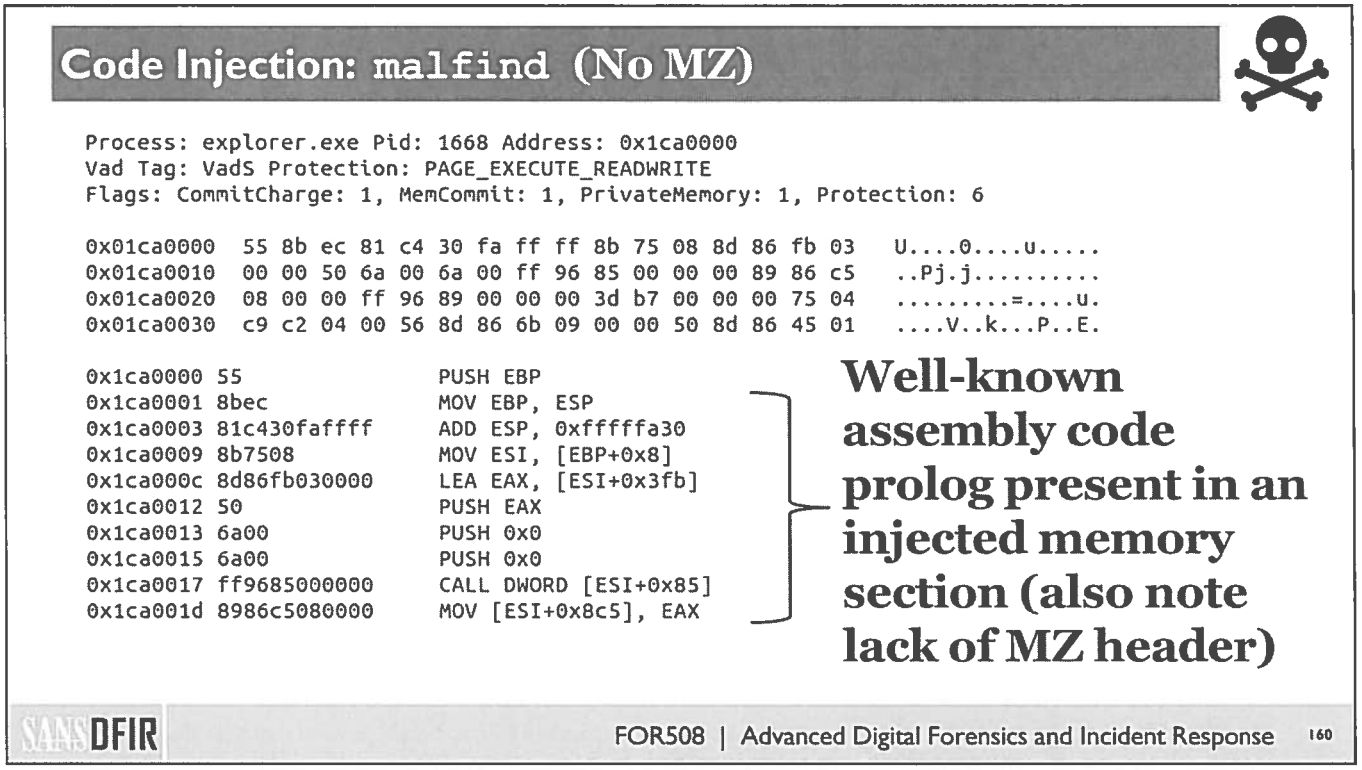
`Malfind` has an impressive success rate, but false positives happen. One nice feature of the output is that it displays the initial part of the memory section in hex and as assembly code, making it easy to identify the telltale "MZ" header for Portable Executable files or whether the data appears to be something else like shellcode.



Code Injection: `malfind` (Zeus)

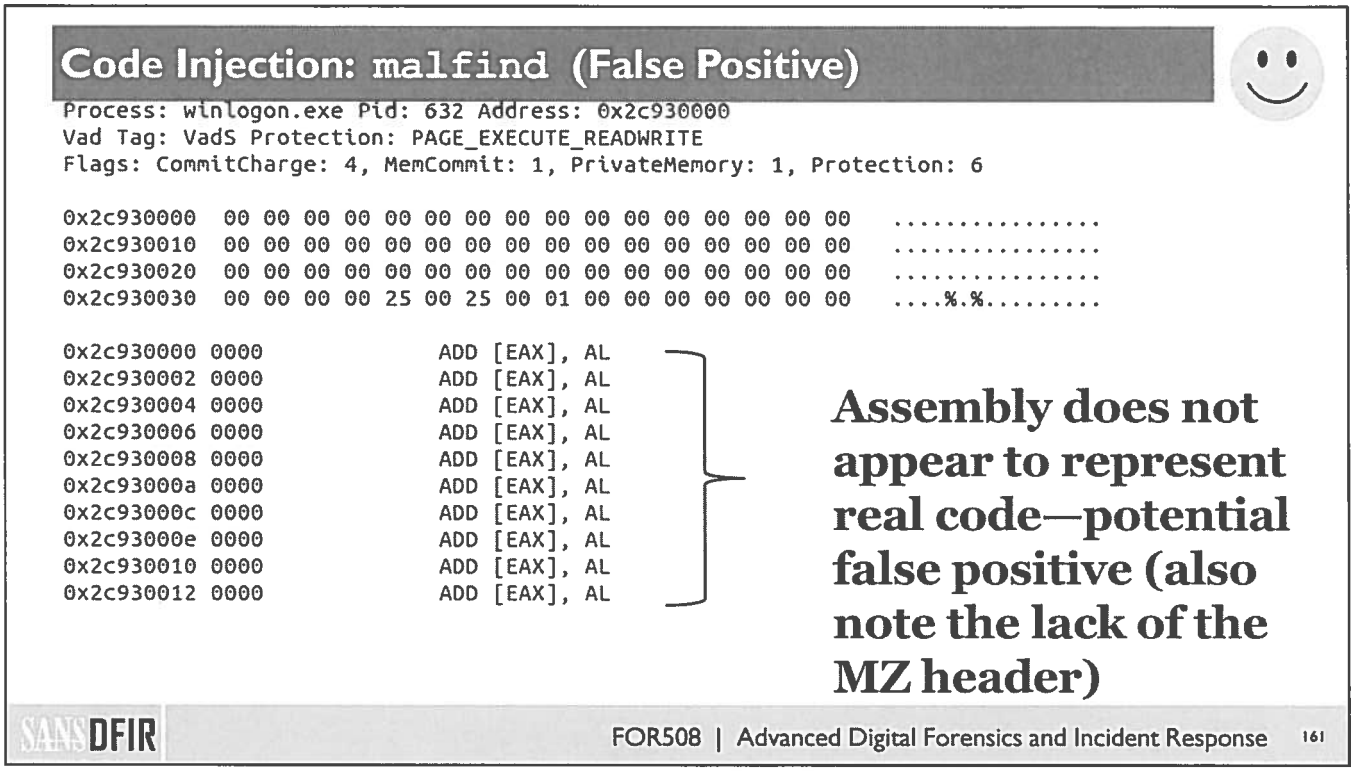
As expected, **`malfind`** run on memory from a system infected by Zeus identifies a large number of injected sections (recall that Zeus often injects into nearly every running process). This slide shows one such section located at memory offset 0x1a0000 owned by the System process (PID 4). Note that this memory section has PAGE_EXECUTE_READWRITE privileges (a known indicator for injection) and is not mapped to a file on disk (unseen here, but a necessary precondition observed by **`malfind`**). The only remaining check is whether there is actual code located in this memory section. In this case, we notice the four-byte Windows Portable Executable signature 4d5a9000 (or MZ signature) at the start of the memory section. This is a pretty good indicator that this in fact is an executable that has been loaded through non-standard methods into a memory section with abnormal (and dangerous) permissions. Thus, it fulfills all three of the checks typically done for injection, and we can mark this process as injected. Also note that we gave the **`malfind`** command a "dump-dir" parameter, so in that folder, we should see an extracted copy of each memory section identified. Our next step might be to perform further analysis on this section, like running it through Anti-Virus or reverse engineering the assembly.

At the bottom of the slide, we see two commands parsing the **`malfind`** output. The first is recording the number of hits for MZ found. Notice that the value returned, 24, matches with what Redline identified in the same image. The second command counts the total number of process memory sections identified by **`malfind`** (it hits on any line containing "Process," which is at the beginning of each malfind hit). Notice that over 50% of the identified sections did not contain an MZ header in this image. Keep in mind that some of them might be false positives. Our next step will be to look at each hit and analyze the disassembly to try to determine whether the hit is a false positive.



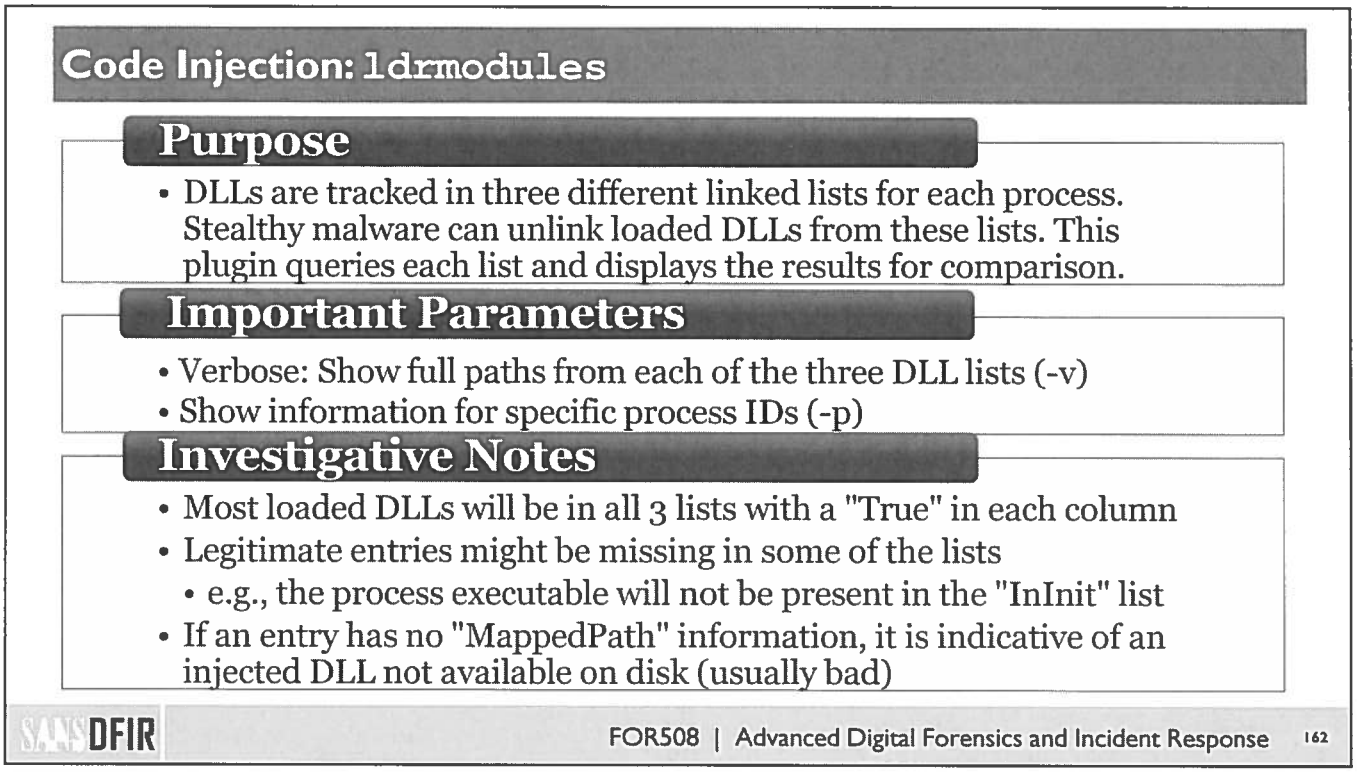
Code Injection: `malfind` (No MZ)

In this **`malfind`** example, we see an identified section in the explorer.exe process with PAGE_EXECUTE_READWRITE privileges that is not mapped to a file on disk (unseen here, but a necessary precondition observed by **`malfind`**). A short snippet of the contents of the page is provided for analyst review. In this example, the contents appear to be legitimate code because the section contains the standard PUSH EBP - MOV EPB, ESP assembly language prolog that denotes a function. The presence of real code in the memory section confirms that the process was likely injected and is an excellent example of code injection that does not utilize the portable executable format (for example, no MZ header). This injected memory section was found on a system infected by Conficker.



Code Injection: `malfind` (False Positive)

Here, we see a section identified by **`malfind`** in the winlogon.exe process with PAGE_EXECUTE_READWRITE privileges that is not mapped to a file on disk (unseen here, but a necessary precondition observed by **`malfind`**). A short snippet of the contents of the section is provided for analyst review. Notice that the disassembly looks very repetitive, with the same instruction being repeated over and over. This is because the large number of hex "00 00" bytes at the beginning of the memory section are being interpreted as "ADD [EAX], AL." By definition, the assembly opcode for this instruction is "00 00," and the disassembler is just trying to interpret what it is being fed. In this case, this looks like garbage and is probably not real code. Hence, this memory section fails one of the three checks for injection (code is not present in the section) and is a likely false positive.



Code Injection: `ldrmodules`

Recall that each process is represented by an EPROCESS block. This block has a link to a Process Environment Block (PEB), which among other things, contains three doubly linked lists for tracking a processes' loaded DLLs. In most cases, these lists contain the same data, just ordered in different ways. The three lists are:

- InLoadOrderModule List
- InInitializationOrderModule List
- InMemoryOrderModule List

Unlinking a DLL from one or more of these lists is a simple means for malware to hide injected DLLs. There is no disruption to the process execution, but any tools used to view the loaded DLLs (like the Volatility plugin **`dlllist`**) will not show the unlinked DLL. Interestingly, unlinking does not require System or Kernel privileges. Thus a user-mode rootkit can accomplish this using only Administrator credentials.

Instead of just reporting what is listed in the process DLL linked lists, **`ldrmodules`** works by scanning all of a processes' memory sections for DLLs. It then compares what was found in the scan with the information present in the standard lists, displaying the results. Additionally, the DLL path information is retrieved from the process VAD tree for further comparison (available in the "MappedPath" column). A "True" within a column means the DLL was present, and a "False" means the DLL was not present in the list. By comparing the results, we can visually determine which DLLs might have been unlinked, and hence malicious. The following is provided by this plugin:

- Process ID
- Process Name
- Base Offset (location in memory image)

- InLoadOrderModule List ('InLoad')
- InInitializationOrderModule List ('InInit')
- InMemoryOrderModule List ('InMem')
- MappedPath

Keep in mind that a small percentage of legitimate DLLs will not be present in some (or all) of the lists. As an example, each process executable (for example, lsass.exe) will be missing from the InInitializedOrder list by definition. If you do not see any information in the "MappedPath" column, that is an indication that the DLL is not present on the disk (hence, there is no full path information). When this is encountered, it is usually a sign of DLL injection, even if the unnamed DLL is present in all three of the linked lists. Use the "-v" flag to try to get additional information or plan to dump the memory section for further analysis.

Code Injection: `ldrmodules` (Stuxnet)

```
root@SIFT-Workstation:/# vol.py -f /memory/stuxnet.img ldrmodules -p 868

Pid   Process    Base       InLoad InInit InMem MappedPath
----  ---------- ---------- ------ ------ ----- ----------
 868  lsass.exe  0x00080000 False  False  False -
 868  lsass.exe  0x7c900000 True   True   True  \WINDOWS\system32\ntdll.dll
 868  lsass.exe  0x77e70000 True   True   True  \WINDOWS\system32\rpcrt4.dll
 868  lsass.exe  0x7c800000 True   True   True  \WINDOWS\system32\kernel32.dll
 868  lsass.exe  0x77fe0000 True   True   True  \WINDOWS\system32\secur32.dll
 868  lsass.exe  0x7e410000 True   True   True  \WINDOWS\system32\user32.dll
 868  lsass.exe  0x01000000 True   False  True  -
 868  lsass.exe  0x77f10000 True   True   True  \WINDOWS\system32\gdi32.dll
 868  lsass.exe  0x77dd0000 True   True   True  \WINDOWS\system32\advapi32.dll
```

Two suspicious regions identified within a Stuxnet injected process (lsass.exe)

- Two DLLs not present on the disk (sign of injection)
- Offsets match injection findings in Redline and **`malfind`**

Code Injection: `ldrmodules` (Stuxnet)

Continuing on our Stuxnet example (from the **`malfind`** slides), we ran **`ldrmodules`** on one of the potentially injected processes to uncover additional information (lsass.exe PID: 868). Interestingly, two entries were returned with no path information. This commonly means that the DLLs do not exist on disk and were injected from memory into the process. By not writing to disk, malware can attempt to evade anti-virus and forensic analysis.

Having no MappedPath is a strong enough clue, but we also see that the DLLs are missing from multiple linked lists. Section 0x80000 is not present in any lists and is a good example of an unlinked DLL. Section 0x1000000 is present in all but the InInit list. Running the "-v" verbose parameter for this process shows that this section is actually the lsass.exe binary. The process executable is not present in the InInit list by definition. However, the fact that it does not have a path on disk is very suspicious. It turns out that this is an artifact of process hollowing: The legitimate executable for this process (lsass.exe) was unmapped so that new code could take its place. Because this all took place in memory, there is no corresponding file on disk representing the new contents of the process executable.

If you match the findings on this slide with what **`malfind`** and Redline returned regarding injected sections by using the Base address, you will see that both tools correctly identified these memory sections as suspicious.

Command-line example:

```
root@SIFT-Workstation:/# vol.py -f /memory/stuxnet.img ldrmodules -p 868

Pid      Process         Base              InLoad InInit  InMem MappedPath
-------- --------------- ----------        ------ ------  -----  ----------
   868 lsass.exe   0x00080000  False  False False -
   868 lsass.exe    0x7c900000    True   True   True  \WINDOWS\system32\ntdll.dll
   868 lsass.exe    0x77e70000    True   True   True  \WINDOWS\system32\rpcrt4.dll
   868 lsass.exe    0x7c800000    True   True   True  \WINDOWS\system32\kernel32.dll
   868 lsass.exe    0x77fe0000    True   True   True  \WINDOWS\system32\secur32.dll
   868 lsass.exe    0x7e410000    True   True   True  \WINDOWS\system32\user32.dll
   868 lsass.exe   0x01000000  True  False True  -
   868 lsass.exe    0x77f10000    True   True   True  \WINDOWS\system32\gdi32.dll
   868 lsass.exe    0x77dd0000    True   True   True  \WINDOWS\system32\advapi32.dll
```

```
root@SIFT-Workstation:/# vol.py -f /memory/stuxnet.img ldrmodules -p 868

Pid   Process    Base       InLoad InInit InMem MappedPath
----  ---------- ---------- ------ ------ ----- ----------
 868  lsass.exe  0x00080000 False  False  False -
 868  lsass.exe  0x7c900000 True   True   True  \WINDOWS\system32\ntdll.dll
 868  lsass.exe  0x77e70000 True   True   True  \WINDOWS\system32\rpcrt4.dll
 868  lsass.exe  0x7c800000 True   True   True  \WINDOWS\system32\kernel32.dll
 868  lsass.exe  0x77fe0000 True   True   True  \WINDOWS\system32\secur32.dll
 868  lsass.exe  0x7e410000 True   True   True  \WINDOWS\system32\user32.dll
 868  lsass.exe  0x01000000 True   False  True  -
 868  lsass.exe  0x77f10000 True   True   True  \WINDOWS\system32\gdi32.dll
 868  lsass.exe  0x77dd0000 True   True   True  \WINDOWS\system32\advapi32.dll
```

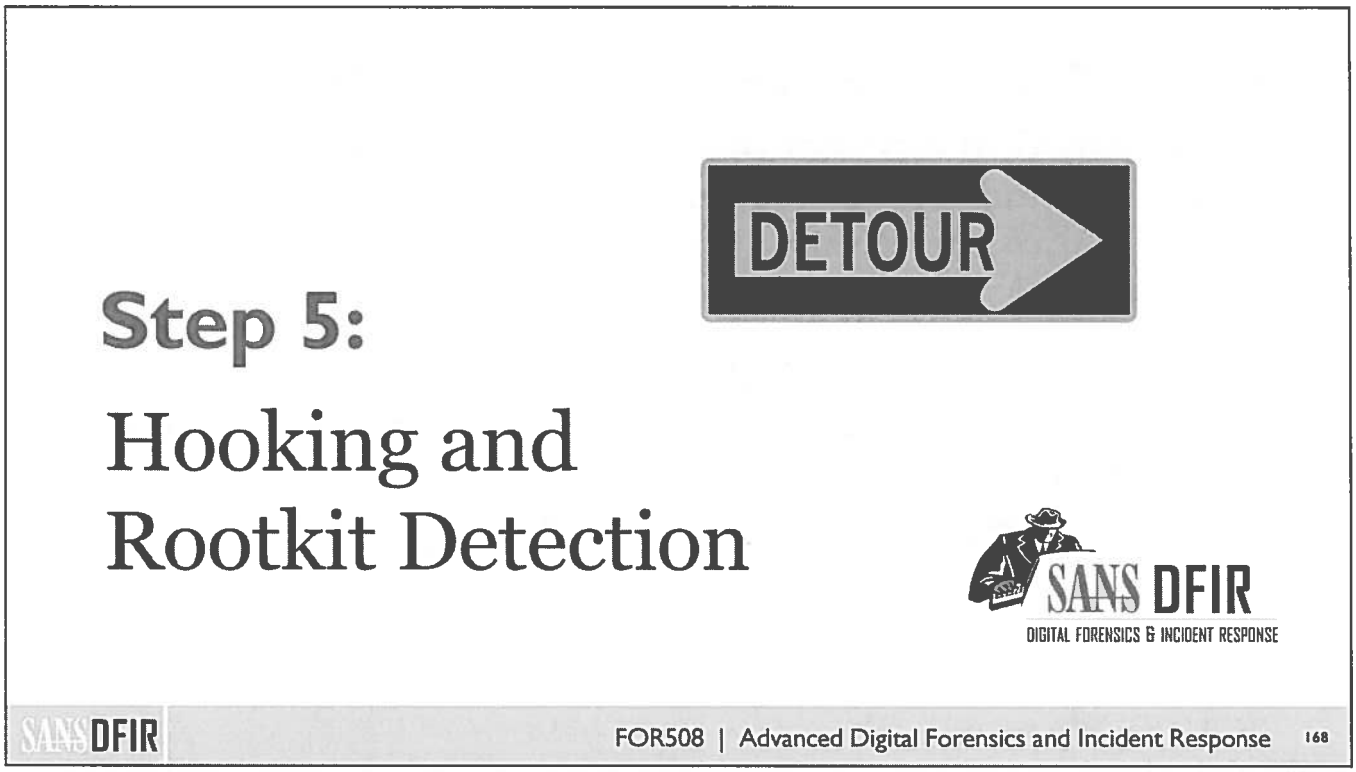
Detecting Code Injection: Review

- **Code injection is a very popular means for malware to hide and launder its activities**
- **The two most common injection methods are:**
  - DLL Injection
  - Process Hollowing
- **Identifying processes with injected memory sections is difficult in disk-based forensics, but fairly easy during memory analysis**
  - Redline identifies injected Portable Executables and unmapped binaries
  - `malfind` identifies and dumps suspicious memory sections and processes, even if there is no Portable Executable header
  - `ldrmodules` shows evidence of unlinked/unmapped DLLs

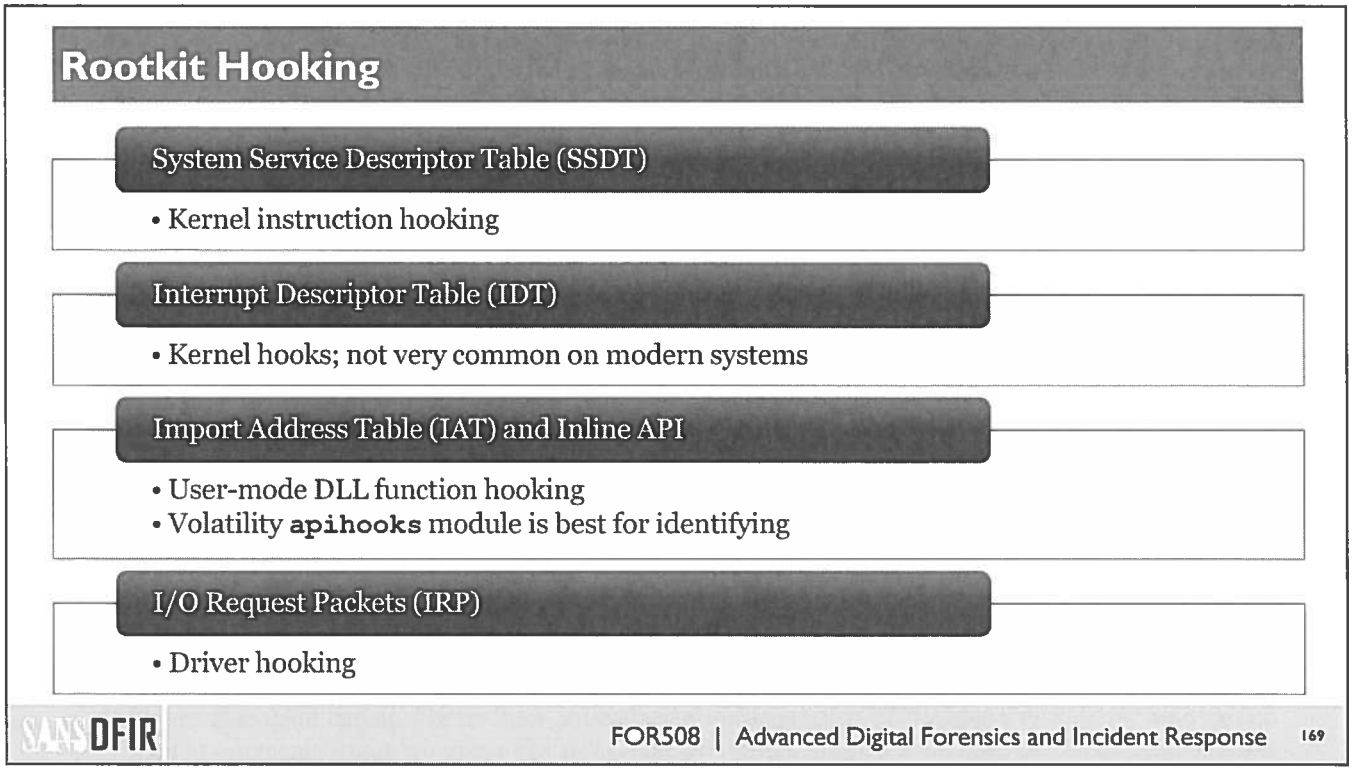
Detecting Code Injection: Review

Although code injection is an advanced malware topic, it is actually easier to spot during memory analysis than many other hiding techniques employed by attackers. This is one reason advanced attackers have resorted to techniques like hiding in plain sight. Redline makes detection of these techniques easy. The presence of injected memory sections is factored into its Malware Rating Index and clearly shown in the Review Memory Sections dialog box. Process hollowing can be detected by identifying strange or duplicate processes running and by looking for evidence of unmapped process binaries, which are easily flagged by Redline.

In Volatility, these techniques can be identified using a combination of the `malfind` and `ldrmodules` plugins. `Malfind` is an advanced plugin that finds suspicious memory sections anywhere within the memory image, dumps them, and provides a preview of disassembly of some of the injected code. `Ldrmodules` focuses on process DLLs, showing evidence of them being unlinked, or not backed by a file on disk.



This page intentionally left blank.



Rootkit Hooking

Rootkits have long been feared by incident responders because they can subvert the kernel to hide processes, files, registry entries, and network connections from live response tools. However, once we get a memory image of the system, finding rootkits becomes much more feasible.

Hooking legitimate system functions and redirecting their output is the most common means for a rootkit to hide itself. A simple way to think about hooking is as a code detour. A malicious process or driver redirects the logical code flow in order to manipulate user input or output. Unfortunately, there are a wide variety of ways to accomplish this in Windows[1]:

System Service Descriptor Table (SSDT)

Hooking the SSDT is a big win for malware. It is easy and highly effective, but also easily detectable. The Kernel uses the SSDT as a lookup table for system functions, and each table entry points to function code. A SSDT hook patches one or more of these pointers to redirect it to a location the rootkit controls. The advantage is that it is based in the kernel and hence is a global or system-wide hook. There can be nearly 1000 instructions available between the two default SSDT tables in a generic Windows XP system. Luckily, there is an easy way for our memory analysis tools to determine which instructions (if any) have been hooked. On a standard Windows install, every SSDT entry will point to instructions in either the system kernel (ntoskrnl.exe) or the GUI driver (win32k.sys). Thus, our tools simply need to identify the address ranges for those two legitimate system files, and filter for any entries that do not point within them. Because SSDT hooks are so popular (at least on WinXP and before), anything listed in the Redline output should be investigated.

Interrupt Descriptor Table (IDT)

The IDT maintains a table of addresses to functions handling interrupts and exceptions. Interrupts occur constantly during system processing, happening whenever a process needs access to the kernel. Thus, if malware controls the IDT, they can manipulate any kernel-level input. However, starting with Windows XP, the use of interrupts has

largely been deprecated and hence they are not very useful for malware to hook. If you see an IDT hook today, it might be due to legacy code in the malware attempting to do something that is no longer viable. There might be legitimate IDT hooks in the table, mostly attributed to ntoskrnl.exe and hal.dll. Redline pre-filters out these well-known exceptions, so if you see an IDT hook in the Redline output, there is a very strong chance that it is malicious.

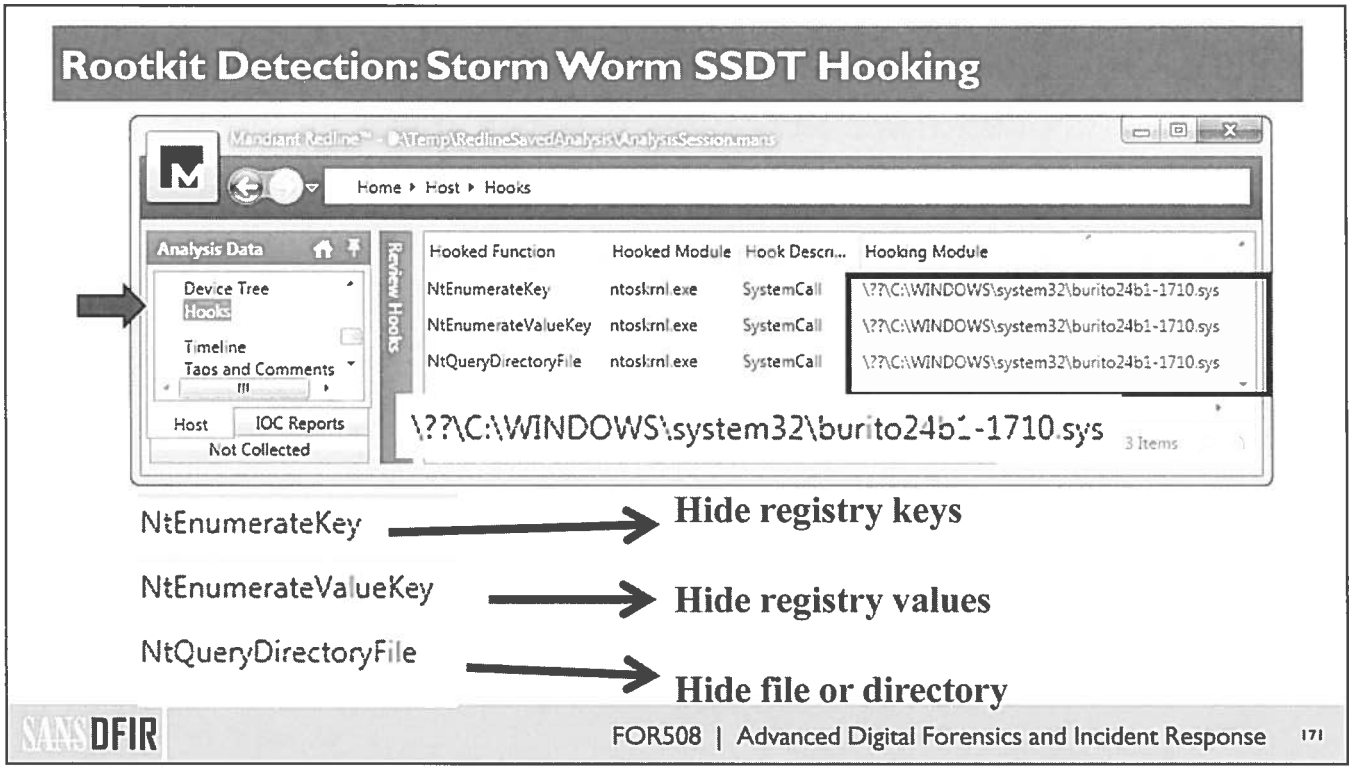
Import Address Table (IAT)

Each process has an import table listing what Windows API functions it uses from specific DLLs and the addresses where they can be found. If malware can get access to this IAT and overwrite addresses, it can redirect certain functions of the process to use malicious code instead. Hooking the IAT of a process takes more work for the attacker, but can be harder to identify. IAT hooks are accomplished in user-mode, not within the kernel. As such, their effects are not global. You might also see legitimate applications creating IAT hooks, making malware identification difficult. Redline does not currently have a means to display IAT hooks. We will see that Volatility has a plugin named **`apihooks`** for this purpose.

I/O Request Packets (IRP)

IRPs are how operating system processes interact with hardware drivers. They control any data being sent or received by hardware, so hooking these functions gives the ability to manipulate things like network traffic, disk reads, and keyboard entries. A large number of legitimate system drivers and processes hook various IRP functions. This makes separating the good from the bad very difficult. Look for suspicious driver names, or drivers with very low occurrences. To make analysis more feasible, Redline will attempt to identify trusted and untrusted hooks and even more helpful, during live analysis, it will verify the digital signatures of the hooking drivers.

[1] http://www.symantec.com/connect/articles/windows-rootkits-2005-part-one



Rootkit Detection: Storm Worm SSDT Hooking

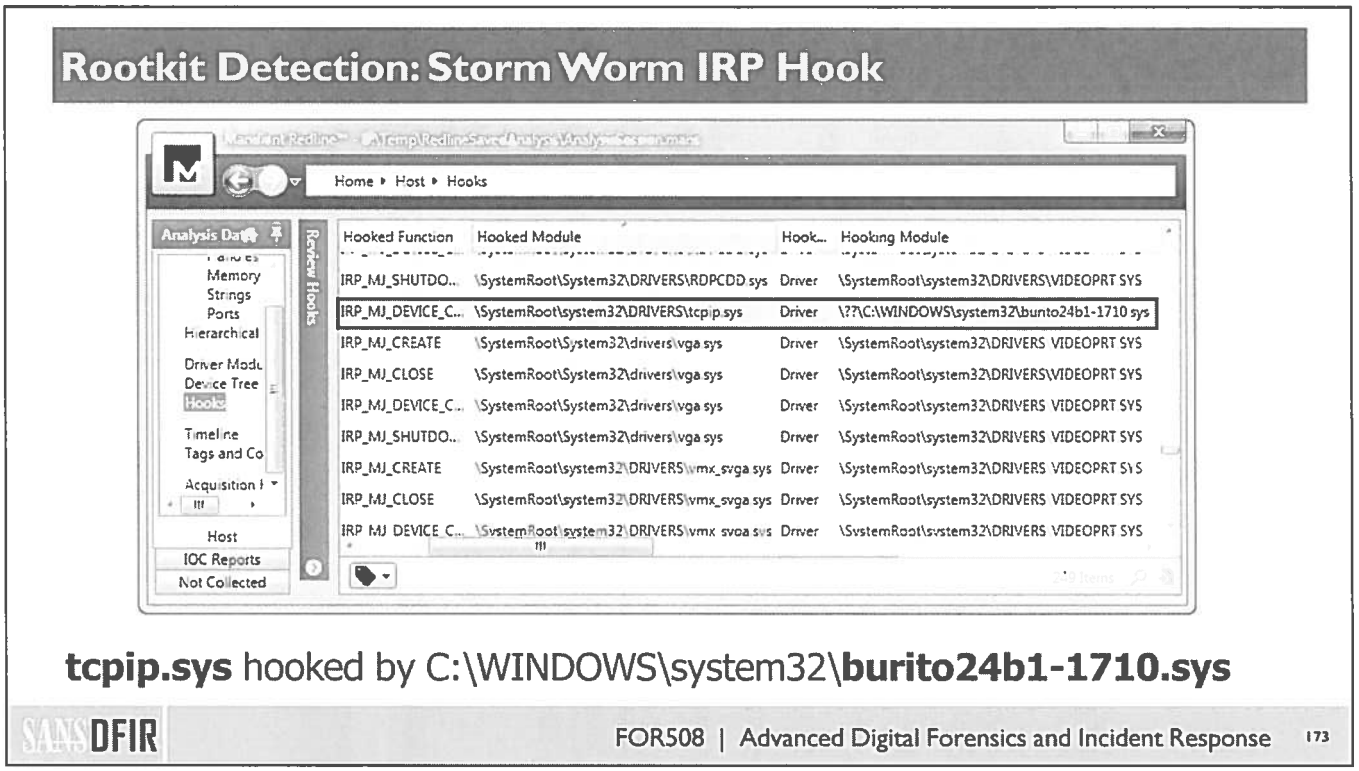
In this example, we have opened the Review Hooks dialog box within Redline and selected **SSDT**. If you see SSDT hooks referenced in Redline, there is a good chance that they are malicious. The SSDT table maintains a list of hundreds of functions and where those functions can be found. On a standard system, all of those functions will be found in one of two places: ntoskrnl.exe or win32k.sys. Redline automatically filters these two locations out, so anything you see in this dialog box is outside the scope of "normal." Keep in mind that some legitimate security applications might still use hooking. As an example, the SysInternals Process Monitor application hooks multiple functions using its Procmon20.sys driver. But in the example shown on this slide, burito24b1-1710.sys does not appear to be a legitimate driver. Looking a little closer, we see three different system calls being hooked by the same burito24b1-1710.sys driver located in the C:\Windows\system32 directory. It turns out that the reason for the strange name of this driver is that the Storm worm chooses a random name for its malware. Because of the random naming scheme, we might not be able to immediately use Google to identify this as being the Storm worm, but at a minimum we know that the system has likely been compromised by a rootkit.

Looking closer at the hooked functions can give us a better idea about what the malware was trying to hide, or help us make a better determination on whether something is malicious. If you were to look up these functions on Microsoft TechNet, you would be able to quickly determine their benefits.

- **NtEnumerateKey:** Allows an application to identify and interact with registry keys. This allows the malware to insert itself between any registry key requests and filter out any registry keys it might want to hide.
- **NtEnumerateValueKey:** Allows an application to identify and interact with registry values. This allows the malware to insert itself between any registry value requests and filter out any registry keys it might want to hide.
- **NtQueryDirectoryFile:** Gives the application the ability to perform a directory listing. By hooking this function, malware can affect what files or directories are returned back to an application, including some anti-virus products, cloaking the files from analysis. Remember that because SSDT hooks are kernel based and global, these changes would be valid for ALL running applications.

The Storm worm hooks these functions to hide registry entries detailing a service started as a persistence mechanism and to hide its malicious files within the file system.[1]

[1] http://securitylabs.websense.com/content/Assets/Storm_Worm_Botnet_Analysis_-_June_2008.pdf



Rootkit Detection: Storm Worm IRP Hook

Malicious I/O Request Packet (IRP) hooks are difficult to identify because there are so many legitimate hooks you must first eliminate. Most systems have a large collection of third-party drivers for various hardware and as such, IRP hooks are very common to override default functions and have them point to the proper driver. A good rule of thumb is to look for outliers. Similar to Least Frequency of Occurrence, most malware will hook sparingly and might hook functions that no other legitimate modules have hooked. When looking for IRP hooks, you might find several modules that might be suspicious. Your next step might be to extract those drivers from the memory image to run anti-virus on or perform more detailed reverse engineering.

Continuing our Storm Worm example, we were given a big head start when looking for IRP hooks because we had already found some SSDT hooks and identified the hooking driver, burito24b1-1710.sys. As seen on this slide, that same suspicious driver hooked a function within the tcpip.sys driver. By hooking the tcpip.sys, the rootkit can manipulate data returned about network connections and sockets (hooking the disk.sys driver would allow it to manipulate directory listing results). The Storm Worm is a SPAM bot, so it will usually have multiple network connections open. By hooking the IRP_MJ_DEVICE_CONTROL function within tcpip.sys, it can hide all of its network activity from live response tools like netstat or openports.

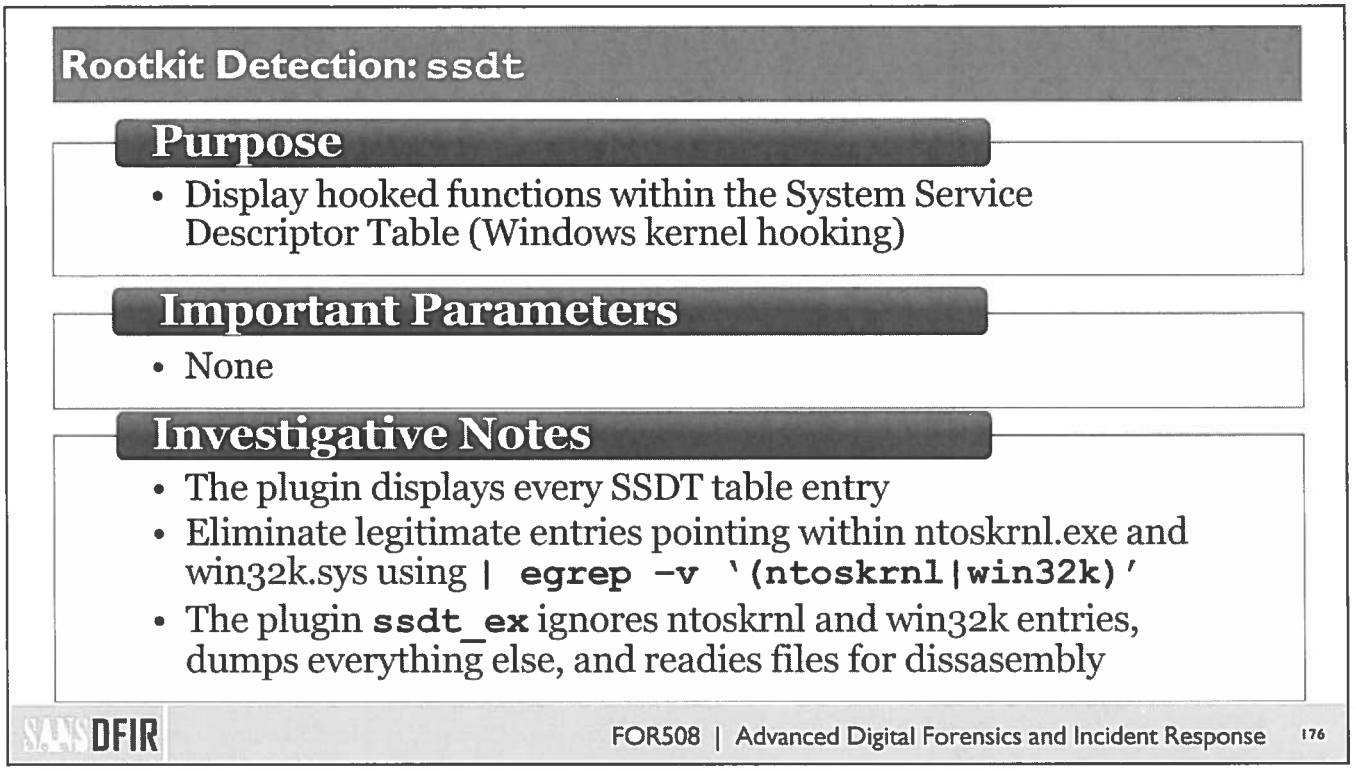
Mandiant Redline™ - DATemp\RedlineSaved\AnalysisSession\mans
Home ▸ Host ▸ Hooks
Analysis Da
Memory
Strings
Ports
Hierarchical
Driver Modu
Device Tree
Hooks
Timeline
Tags and Co
Acquisition I
Host
IOC Reports
Not Collected
Review Hooks
Hooked Function
Hooked Module
Hook...
Hooking Module
IRP_MJ_SHUTDO... \SystemRoot\System32\DRIVERS\RDPCDD.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
IRP_MJ_DEVICE_C... \SystemRoot\system32\DRIVERS\tcpip.sys Driver \??\C:\WINDOWS\system32\bunito24b1-1710.sys
IRP_MJ_CREATE \SystemRoot\System32\drivers\vga.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
IRP_MJ_CLOSE \SystemRoot\System32\drivers\vga.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
IRP_MJ_DEVICE_C... \SystemRoot\System32\drivers\vga.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
IRP_MJ_SHUTDO... \SystemRoot\System32\drivers\vga.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
IRP_MJ_CREATE \SystemRoot\system32\DRIVERS\vmx_svga.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
IRP_MJ_CLOSE \SystemRoot\system32\DRIVERS\vmx_svga.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
IRP_MJ_DEVICE_C... \SystemRoot\system32\DRIVERS\vmx svoa.sys Driver \SystemRoot\system32\DRIVERS\VIDEOPRT.SYS
249 Items

Volatility Rootkit Detection Plugins

ssdt	Display System Service Descriptor Table entries
psxview	Find hidden processes via cross-view techniques
modscan	Find modules via pool tag scanning
apihooks	Find DLL function (inline and trampoline) hooks
driverirp	Identify I/O Request Packets (IRP) hooks
idt	Display Interrupt Descriptor Table hooks

Volatility Rootkit Detection Plugins

Volatility provides some interesting tools for use in detecting rootkits. The **`psxview`** plugin takes advantage of several other plugins that query process information and displays their results within a table so inconsistencies can be visually found. This is a very powerful technique for finding hidden processes. **`Modscan`** identifies loaded kernel modules allowing us to search for malicious drivers commonly used to take control of the operating system. The **`apihooks`** plugin provides us additional hook detection capabilities, including inline function hooks, which we did not see in Redline. Finally, the **`ssdt`**, **`driverirp`**, and **`idt`** plugins provide information on each type of hook. Because the latter two operate identically to the **`ssdt`** plugin, we present them here but will not have a separate slide for them.



Rootkit Detection: `ssdt`

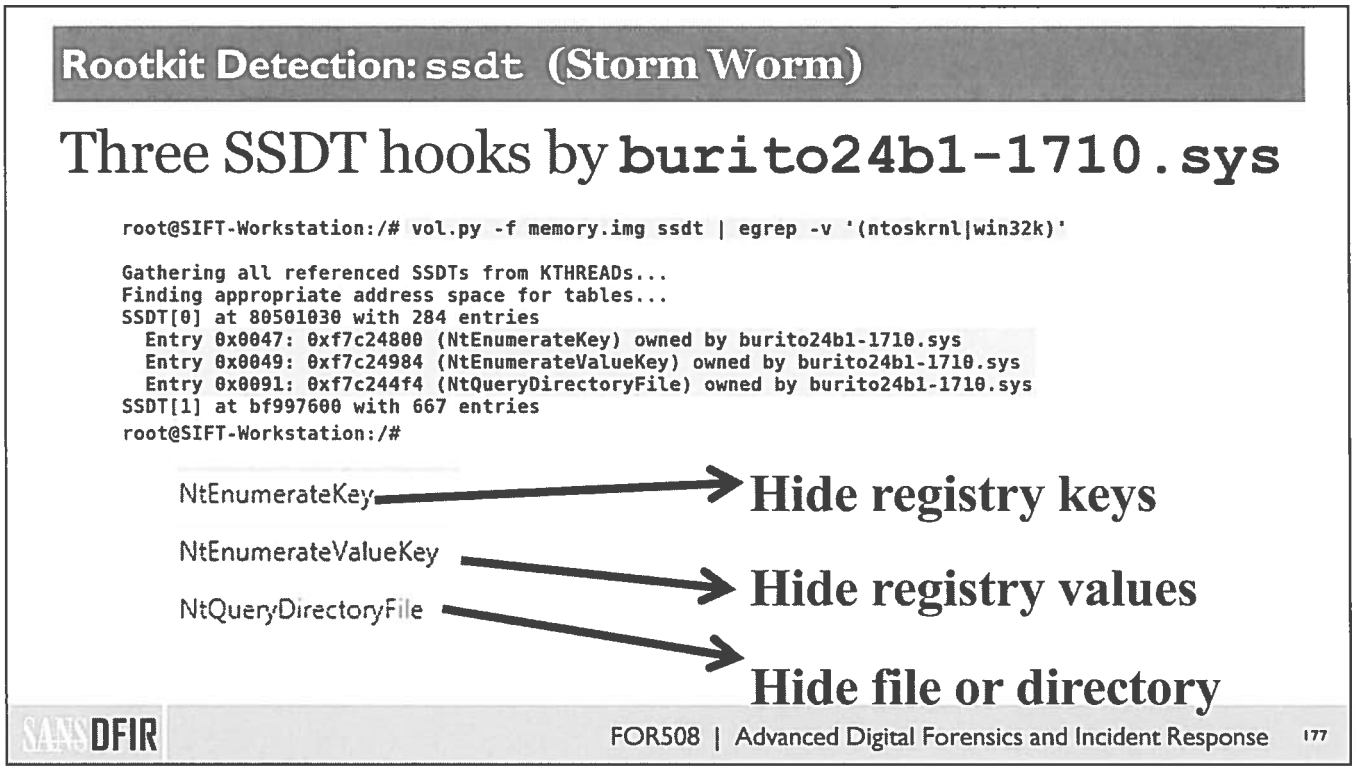
Hooking the System Service Descriptor Table (SSDT) is like invoking the nuclear option. It is highly effective, but also easily noticed. The SSDT holds pointers to the various kernel functions that power Windows. The advantage of hooking these functions is that they are instantly global: The hooks will be valid for any process on the system. The disadvantage is that it requires changes to the Kernel, which can risk the stability of the running system. In fact, as of Vista, Kernel Patch Protection (known as PatchGuard) will now preemptively crash the system when anomalous changes are identified in the SSDT and other critical kernel components. Although this doesn't eliminate the possibility of finding these hooks in Vista and above (there are always counter-measures to every protection), it does greatly reduce their likelihood.

We can identify SSDT hooks using the Volatility plugin **`ssdt`**. This plugin works a little differently than what we saw in Redline. Instead of only showing suspicious SSDT entries, the plugin will print every entry in the various SSDT tables available. For each table entry, the following is provided:

- Table Entry
- Function Offset
- Function
- Function Owner (the module that the SSDT entry is sending the request to)

To determine which SSDT instructions may be hooked, we will need to filter out references within ntoskrnl.exe and win32k.sys, which own the vast majority of legitimate functions that the SSDT tables track. If we ignore any SSDT entries pointing into those two modules, any malicious drivers become readily apparent. Filtering can be easily accomplished by piping the plugin results into the extended grep tool: **`egrep -v '(ntoskrnl | win32k)'`** . For those familiar with Redline, it does this filtering automatically.

You should be aware that a similar plugin exists with additional functionality. The **`ssdt_ex`** plugin was built to automate the process of SSDT analysis. It automatically ignores any functions within ntoskrnl and win32k, dumps any other drivers found to be hooking functions within the SSDT, and pre-builds files for import into the IDA Pro disassembler (used for reverse engineering).



Rootkit Detection: `ssdt` (Storm Worm)

The Storm Worm includes rootkit features to hide its processes, files, and Windows Registry entries. Looking at an image infected with the Storm Worm, we ran the following:

root@SIFT-Workstation:/memory# **vol.py -f memory.img** ssdt | egrep -v '(ntoskrnl|win32k)'

Gathering all referenced SSDTs from KTHREADs...

Finding appropriate address space for tables...

SSDT[0] at 80501030 with 284 entries

Entry 0x0047: 0xf7c24800 (NtEnumerateKey) owned by **burito24b1-1710.sys**

Entry 0x0049: 0xf7c24984 (NtEnumerateValueKey) owned by **burito24b1-1710.sys**

Entry 0x0091: 0xf7c244f4 (NtQueryDirectoryFile) owned by **burito24b1-1710.sys**

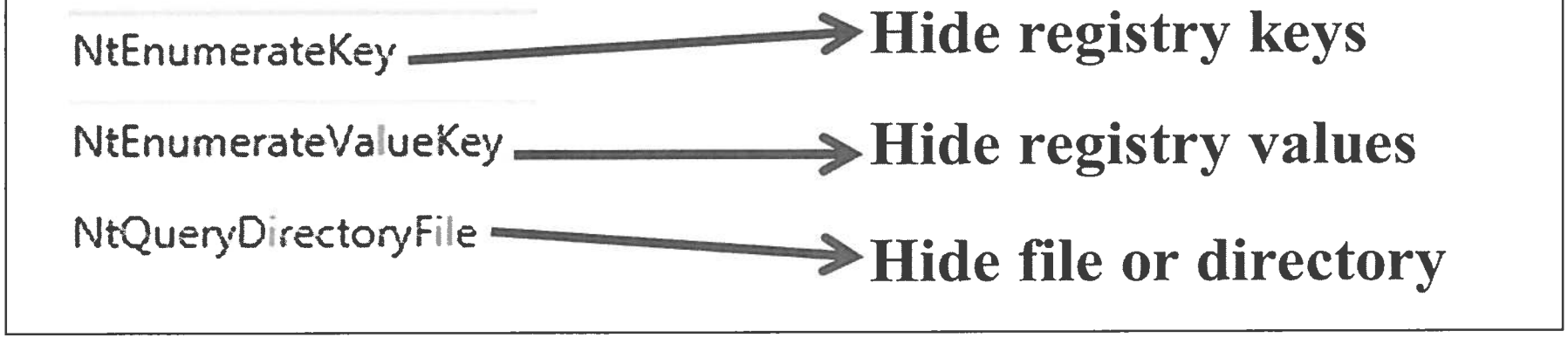
SSDT[1] at bf997600 with 667 entries

This command runs the Volatility ssdt plugin and pipes the results to a grep search function set up to ignore (-v flag) any lines containing ntoskrnl or win32k. Looking at the results, it is clear how many lines of output grep saves us: in two different SSDT locations, there are a combined total of 284+667= 951 entries. Of those 951 System Service Descriptor Table (SSDT) entries, three pointed to functions in modules other than ntoskrnl.exe and win32k.sys. Those three were hooked by a driver named "burito24b1-1710.sys." Luckily, that driver name is highly suspicious; the Storm Worm names the driver using random characters. Looking a little closer into the kernel functions hooked by this driver gives helpful information as to what the malware was attempting to accomplish:

- **NtEnumerateKey:** Allows an application to identify and interact with registry keys. This would allow the malware to insert itself between any registry key requests and filter out any registry keys it might want to hide.
- **NtEnumerateValueKey:** Allows an application to identify and interact with registry values. This would allow the malware to insert itself between any registry value requests and filter out any registry keys it might want to hide.
- **NtQueryDirectoryFile:** Gives an application the ability to perform a directory listing. By hooking this function, malware can affect what files or directories are returned back to an application, including some anti-virus products, cloaking the malware from analysis. Remember that because SSDT hooks are kernel based and global, this would be valid for ALL running applications.

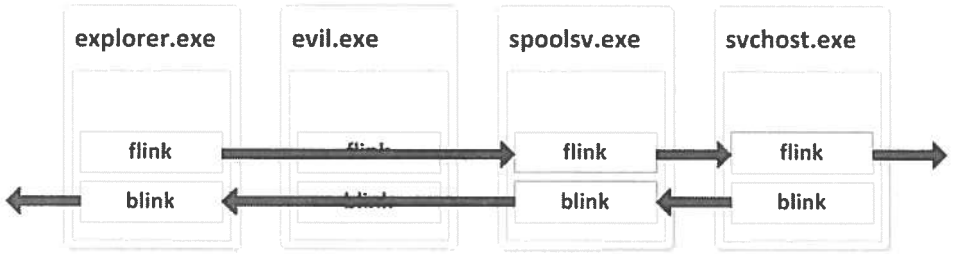
If you find a driver of interest, the next step would be to export that driver from the memory image for further review. This can be accomplished using the Volatility plugin **moddump**.

```
root@SIFT-Workstation:/# vol.py -f memory.img ssdt | egrep -v '(ntoskrnl|win32k)'

Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 80501030 with 284 entries
  Entry 0x0047: 0xf7c24800 (NtEnumerateKey) owned by burito24b1-1710.sys
  Entry 0x0049: 0xf7c24984 (NtEnumerateValueKey) owned by burito24b1-1710.sys
  Entry 0x0091: 0xf7c244f4 (NtQueryDirectoryFile) owned by burito24b1-1710.sys
SSDT[1] at bf997600 with 667 entries
root@SIFT-Workstation:/#
```



Rootkit Detection: Direct Kernel Object Manipulation

- DKOM is an advanced process hiding technique
  - Unlink an EPROCESS from the doubly linked list

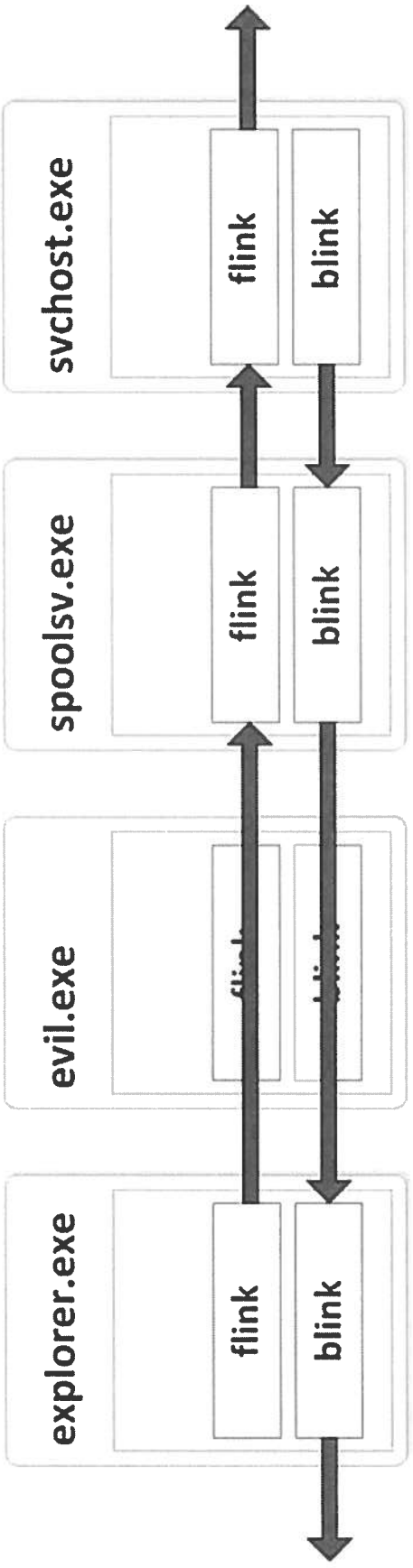


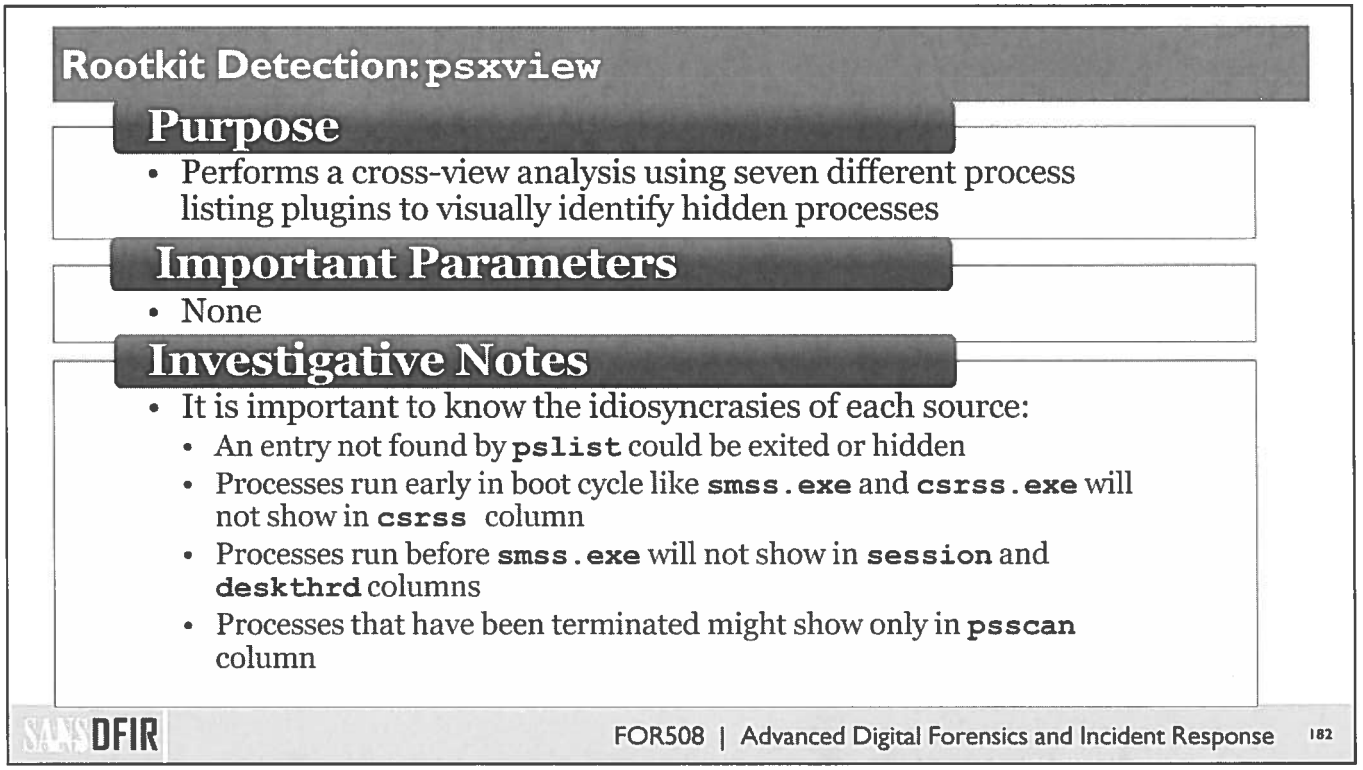
- Several examples exist in the wild:
  - FU, Myfip.H worm, Fanbot.A worm, Prolaco

Rootkit Detection: Direct Kernel Object Manipulation

Direct Kernel Object Manipulation (DKOM) is one of the newest and more advanced rootkit techniques. As the name indicates, DKOM allows malware to make changes to kernel objects directly within memory. The changes do not get written to disk, giving a very small footprint and making it very difficult for protection mechanisms like anti-virus to detect. DKOM can be used to change nearly any kernel object, but popular attacks include unlinking objects from their standard kernel lists. We see such an attack on this slide where DKOM has been used to unlink a process, evil.exe from the EPROCESS doubly linked list. Amazingly, the unlinked process will continue to run, but any standard tools for displaying running processes will fail to identify it. Standard live response tools like `tasklist.exe` and SysInternals' `pslist.exe` on a live system or even the Volatility `pslist` plugin are defeated by DKOM. To defeat this technique, we will need to use a plugin like Volatility's `psscan`, which does not just rely on what appears in the various linked lists.

Although some rootkit techniques are largely theoretical, this one is not. There are several documented cases in the wild including the FU rootkit, Myfip.H worm, Fanbot.A worm, and the Prolaco virus.


svchost.exe
flink
blink
spoolsv.exe
flink
blink
evil.exe
explorer.exe
flink
blink

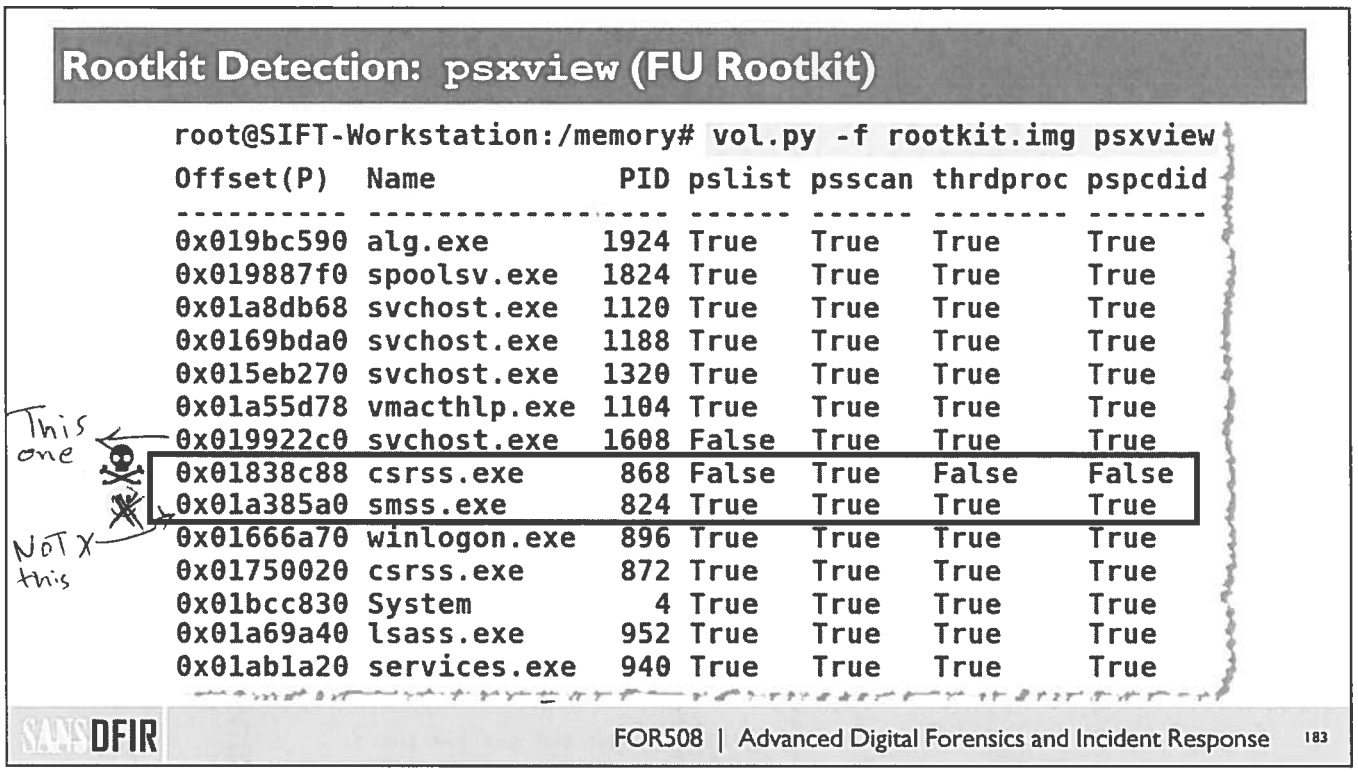


Rootkit Detection: `psxview`

The psxview plugin is a compound plugin that helps the analyst visually identify anomalies between plugins. Because there are several different ways to identify processes within memory, the authors decided to make it easy to compare all of their output for easy cross-view analysis. Cross-view analysis is a simple but powerful technique for identifying rootkits. You start a cross-view analysis by querying the system at a high level, say using the Windows API on a live system or reading the linked lists within a memory image. Then, you compare those results with low-level data gathering, such as manually reading the Master File Table from a raw disk or scanning all of the memory for errant process structures. If your two results differ, then that can be a sign that a rootkit is hiding something from the high-level request. The psxview plugin cuts across many different types of memory objects, assuming that even if a hiding technique can conceal itself from some of them, it would be very difficult to conceal itself from all of them. The plugins run include:

- **pslist**: Read the EPROCESS doubly linked list.
- **psscan**: Scan for EPROCESS structures throughout memory.
- **thrdproc**: Review all threads found in the memory image and collect processes using the thread parent process identifier.
- **pspcid**: The PspCid table is yet another kernel object that keeps track of process and thread PIDs.
- **csrss**: The csrss.exe process keeps a handle to each process started after it (so there will be no entries for smss.exe, the System process, and csrss.exe).
- **session**: List of processes belonging to each logon session.
- **deskthrd**: Identify processes via threads attached to each Windows desktop.

The psxview table has a column for each of these sources, and displays a "False" for the process if it is not found in that source or a "True" if is present. Keep in mind that various legitimate factors might cause some processes to not be present. For instance, the csrss information is only good for processes started after csrss.exe. Thus, early system processes like System, smss.exe, and csrss.exe process will all show as "False" in this list. Also, session and deskthrd information is not available for smss.exe and anything started before it.

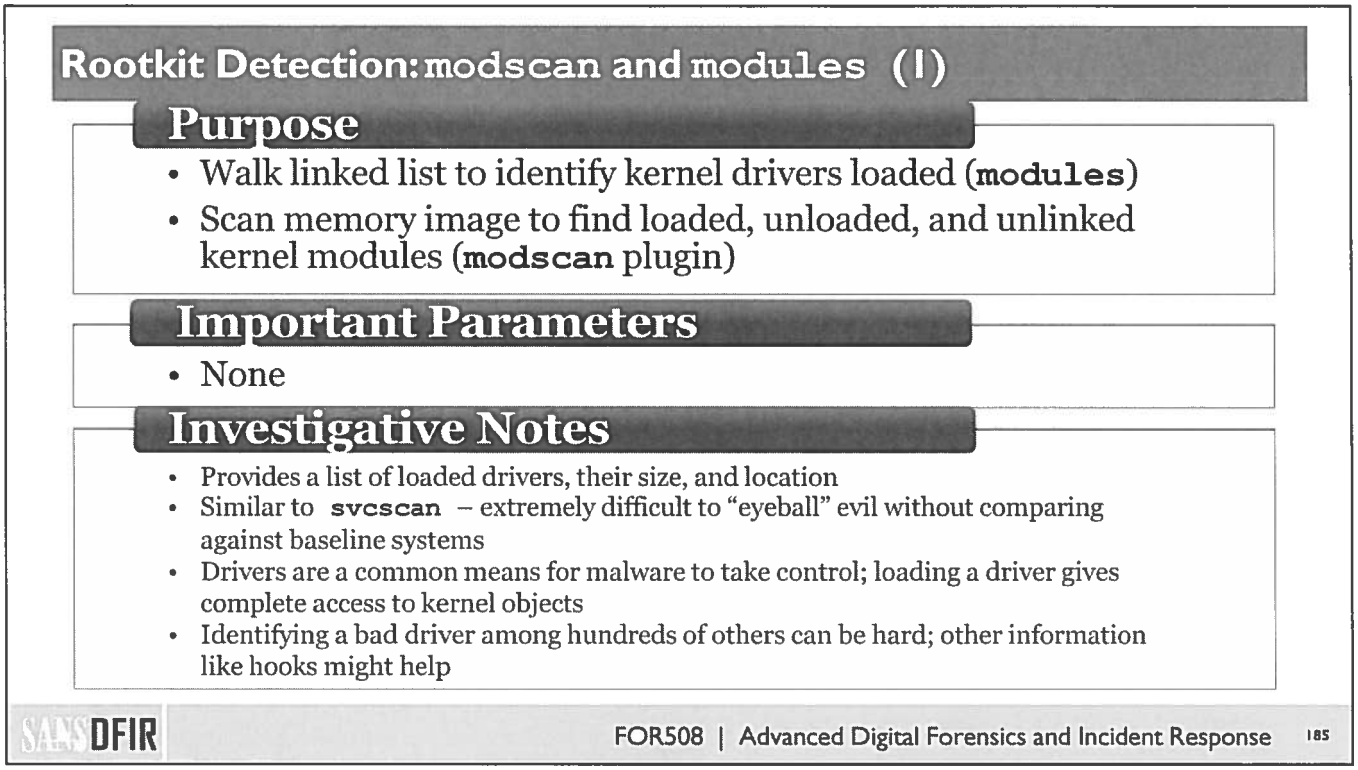


Rootkit Detection: `psxview` (FU Rootkit)

The FU rootkit employs Direct Kernel Object Manipulation (DKOM) techniques to hide processes and drivers. In this example, FU was used to hide a malicious executable named "svchost.exe" using PID 1608. Running the **`psxview`** plugin performs comparisons similar to what a root-kit detector would do. As we see in the example, **`pslist`** does not show PID 1608 whereas **`psscan`**, **`thrdproc`**, and **`pspcdid`** do show entries (the **`csr_`** columns have been cut off). The **`pslist`** plugin should be considered the high-level request and as such whenever you see something not listed there, but registered elsewhere, it is worth looking into. It might be an indication of rootkit activity or it could just be an old terminated process that **`psscan`** picks up, but neither **`pslist`** nor the **`csr_hnds`** and **`csr_list`** structures have pointers remaining for. An example of this behavior is seen in the csrss.exe process with PID 868. Notice that **`psscan`** found an EPROCESS structure for that process, but none of the other lists have any references to it. This is a classic example of a terminated process (in this case, it could be a csrss process from a previous boot).

A final question to ponder is that if **`psscan`** finds the hidden process, why bother with **`psxview`**? The answer is that without comparing and noticing that PID 1608 was present in the **`psscan`** output and not in the **`pslist`** output, it would be easy to assume that the process was legitimate. Knowing it didn't show up in **`pslist`** is important! This is another area that Volatility does better than Redline. Redline's default view operates similar to **`psscan`**. Thus, it finds this DKOM'd process, but does not flag it for not being present in the EPROCESS doubly linked list.

```
root@SIFT-Workstation:/memory# vol.py -f rootkit.img psxview
Offset(P)  Name             PID pslist psscan thrdproc pspcdid
---------- ---------------- ---- ------ ------ -------- -------
0x019bc590 alg.exe         1924 True   True   True     True
0x019887f0 spoolsv.exe     1824 True   True   True     True
0x01a8db68 svchost.exe     1120 True   True   True     True
0x0169bda0 svchost.exe     1188 True   True   True     True
0x015eb270 svchost.exe     1320 True   True   True     True
0x01a55d78 vmacthlp.exe    1104 True   True   True     True
0x019922c0 svchost.exe     1608 False  True   True     True
0x01838c88 csrss.exe        868 False  True   False    False
0x01a385a0 smss.exe         824 True   True   True     True
0x01666a70 winlogon.exe     896 True   True   True     True
0x01750020 csrss.exe        872 True   True   True     True
0x01bcc830 System             4 True   True   True     True
0x01a69a40 lsass.exe        952 True   True   True     True
0x01ab1a20 services.exe     940 True   True   True     True
```

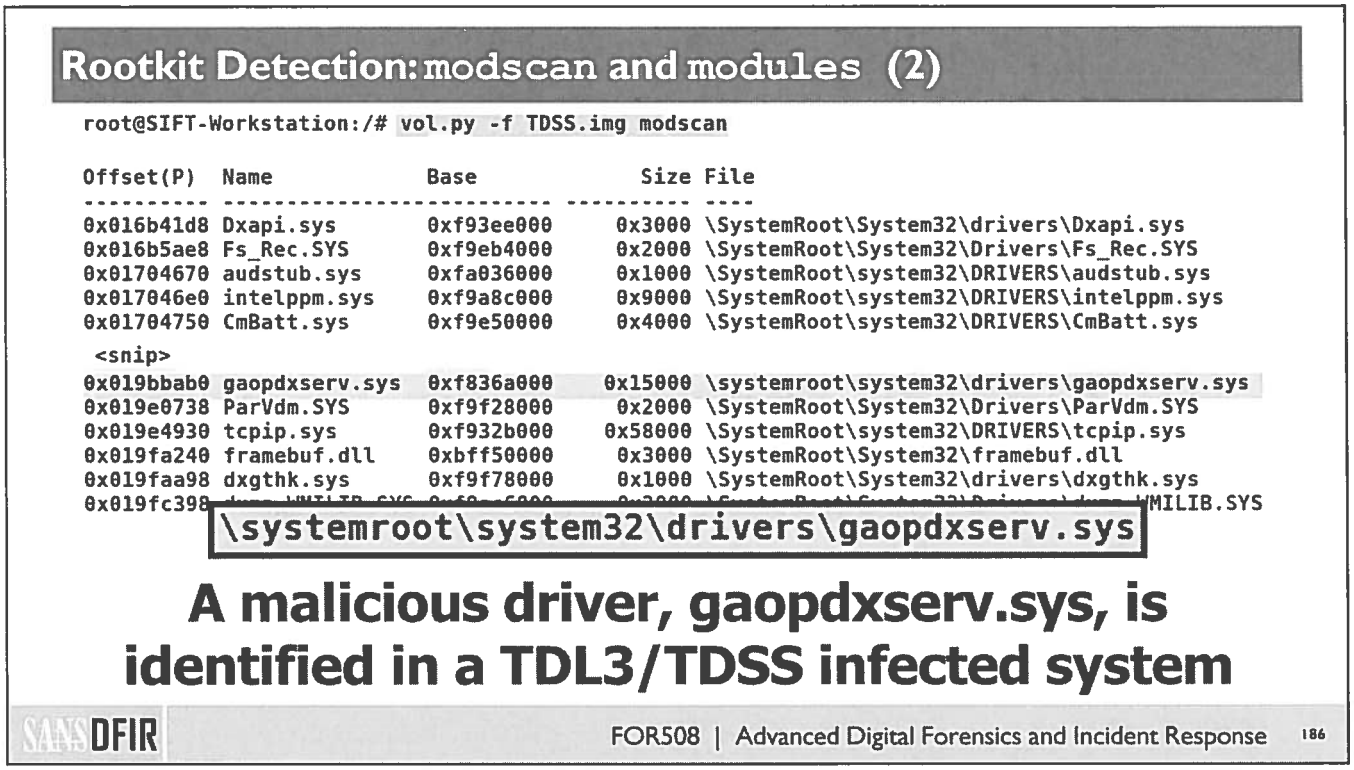


Rootkit Detection: `modscan` and `modules` (1)

Drivers provide the operating system kernel a means to extend its functionality. We see this most often with hardware device drivers, which allow us to interact with various hardware like disks, wireless network cards, or printers. Malware can also utilize drivers to introduce code into the kernel and hence take control of the system. This is a particularly common way that rootkits are installed.

One of the benefits of studying a GUI tool like Redline and command-line tool like Volatility is that the GUI tool helps you visualize the data, whereas the command-line tool helps you understand where the data is coming from. In Redline, we saw device drivers nicely displayed in a table and visually as a relationship tree. The **`modscan`** plugin gives us an idea of how this information is collected. The plugin scans the memory for any instances of pool tags associated with memory pages containing drivers (also referred to as modules). It includes relevant data for each driver including name, size, and location. Sometimes, evil drivers can be recognized immediately using this information, but often they do an excellent job of blending in with the hundreds of other legitimate driver modules loaded on the system. Running another plugin named **`devicetree`** might give a different view as it provides visual information about the chaining, or layering of drivers. Another alternative is to analyze the loaded modules in conjunction with the information gathered on hooks on the system. The two regularly go hand in hand, so comparing information from multiple plugins might be your best bet to finding well-hidden kernel modules.

Should you find a suspicious driver, the Volatility plugin **`moddump`** can be used to extract the driver for more detailed analysis.



Rootkit Detection: `modscan` and `modules` (2)

We use the **`modscan`** plugin in this example to identify suspicious drivers in our memory image infected with the TDL3/TDSS rootkit. Similar to other rootkit detection techniques, distinguishing bad drivers from good drivers can be a very difficult task. A modern Windows system can have hundreds of modules (drivers) loaded. Experienced (or lucky) incident responders might immediately pick out gaopdxserv.sys as a malicious driver, but for the rest of us, we might pick out three or four drivers from this slide that we are unfamiliar or uncomfortable with. This is where we use our standard investigative techniques to narrow things down. One help could be using web search engines (only paying attention to well-known, reliable sites like Microsoft TechNet or Symantec). We could also compare these results against drivers loaded on a similar system in our environment that we do not believe was compromised. This kind of "known goods" analysis can be very powerful. At some point, we might even need to dump any remaining drivers and do reverse engineering. Overall, this is a great example of how much harder bad drivers are to find with this tool versus using the extra capabilities Redline live analysis gives with data like file hash and digital signature checks to help us narrow our focus. But we don't always have a live analysis to work with, so sometimes we just have to do things the hard way….

It turns out that TDSS has a documented malicious driver named "gaopdxserv.sys" (via Google).[1] Notice it still maintains its ".sys" extension. Evil drivers aren't always this hard to spot, because very commonly malware authors don't intend for them to be seen due to their rootkit cloaking capabilities. Upon finding a suspicious driver, our next step would be to identify areas where it might have been used. Searching known hooking locations and device/process strings for the driver name can sometimes help identify its scope and capabilities.

Command-line example:

root@SIFT-Workstation:/mnt/hgfs/SANS/Memory Images# **vol.py -f TDSS.img modscan**

```
Offset(P)  Name                 Base         Size         File
---------- -------------------- ----------   ----------   ----
0x016b41d8 Dxapi.sys        0xf93ee000    0x3000 \SystemRoot\System32\drivers\Dxapi.sys
0x016b5ae8 Fs_Rec.SYS       0xf9eb4000    0x2000 \SystemRoot\System32\Drivers\Fs_Rec.SYS
0x01704670 audstub.sys      0xfa036000    0x1000 \SystemRoot\system32\DRIVERS\audstub.sys
0x017046e0 intelppm.sys     0xf9a8c000    0x9000 \SystemRoot\system32\DRIVERS\intelppm.sys
0x01704750 CmBatt.sys       0xf9e50000    0x4000 \SystemRoot\system32\DRIVERS\CmBatt.sys
<snip>
0x019bbab0 gaopdxserv.sys  0xf836a000    0x15000 \systemroot\system32\drivers\gaopdxserv.sys
0x019e0738 ParVdm.SYS     0xf9f28000   0x2000   \SystemRoot\System32\Drivers\ParVdm.SYS
0x019e4930 tcpip.sys          0xf932b000  0x58000 \SystemRoot\system32\DRIVERS\tcpip.sys
0x019fa240 framebuf.dll       0xbff50000   0x3000    \SystemRoot\System32\framebuf.dll
0x019faa98 dxgthk.sys         0xf9f78000   0x1000    \SystemRoot\System32\drivers\dxgthk.sys
0x019fc398 dump_WMILIB.SYS   0xf9ec6000     0x2000 \SystemRoot\System32\Drivers\dump_WMILIB.SYS
```

Note that the **`modules`** plugin will not identify gaopdxserv.sys.

[1] http://www.f-secure.com/v-descs/rootkit_w32_tdss_gen!a.shtml

```
root@SIFT-Workstation:/# vol.py -f TDSS.img modscan

Offset(P)  Name             Base             Size File
---------- ---------------- ---------- ------ ----
0x016b41d8 Dxapi.sys        0xf93ee000     0x3000 \SystemRoot\System32\drivers\Dxapi.sys
0x016b5ae8 Fs_Rec.SYS       0xf9eb4000     0x2000 \SystemRoot\System32\Drivers\Fs_Rec.SYS
0x01704670 audstub.sys      0xfa036000     0x1000 \SystemRoot\system32\DRIVERS\audstub.sys
0x017046e0 intelppm.sys     0xf9a8c000     0x9000 \SystemRoot\system32\DRIVERS\intelppm.sys
0x01704750 CmBatt.sys       0xf9e50000     0x4000 \SystemRoot\system32\DRIVERS\CmBatt.sys
 <snip>
0x019bbab0 gaopdxserv.sys   0xf836a000    0x15000 \systemroot\system32\drivers\gaopdxserv.sys
0x019e0738 ParVdm.SYS       0xf9f28000     0x2000 \SystemRoot\System32\Drivers\ParVdm.SYS
0x019e4930 tcpip.sys        0xf932b000    0x58000 \SystemRoot\system32\DRIVERS\tcpip.sys
0x019fa240 framebuf.dll     0xbff50000     0x3000 \SystemRoot\System32\framebuf.dll
0x019faa98 dxgthk.sys       0xf9f78000     0x1000 \SystemRoot\System32\drivers\dxgthk.sys
0x019fc398 dump_WMILIB.SYS  0xf9ec6000     0x2000 \SystemRoot\System32\Drivers\dump_WMILIB.SYS
```

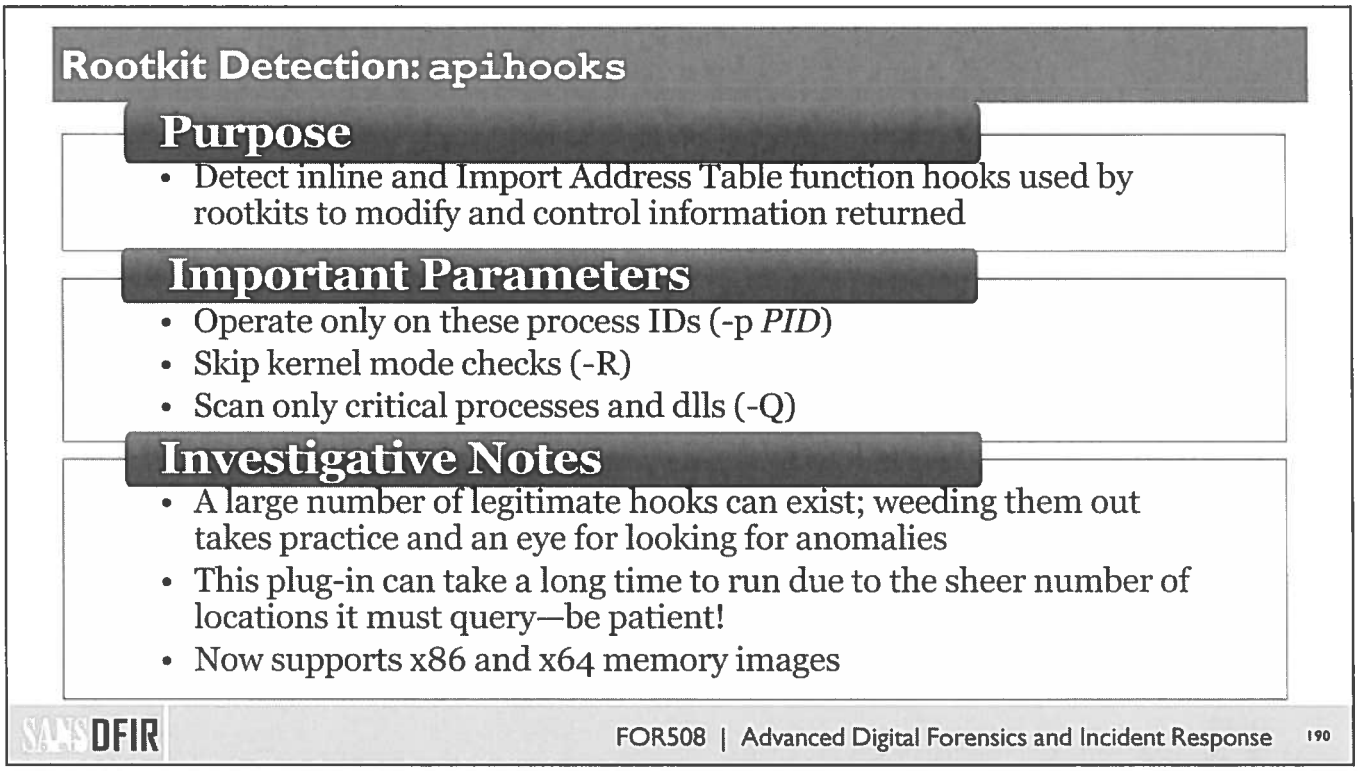
Automating Analysis: `driverbl`

```
root@siftworkstation:/# vol.py -f TDSS.img modules | wc -l
113

root@siftworkstation:/# vol.py driverbl -f TDSS.img -B baseline.img -U

Offset(P)  Service Key      Found Name  DName Module Size  IRPs  Path
---------- ---------------- ----- ----- ----- ------ ----- ----- ----
0x016be3f0 gameenum         False False False False  False False \systemroot\system32\drive
0x01813898 intelppm         False False False False  False False \systemroot\system32\drive
0x01813ed0 usbehci          False False False False  False False \systemroot\system32\drive
0x01872c70 sysaudio         False False False False  False False \systemroot\system32\drive
0x018735f0 wdmaud           False False False False  False False \systemroot\system32\drive
0x01ac3388 es1371           False False False False  False False \systemroot\system32\drive
0x01ac3788 PCnet            False False False False  False False \systemroot\system32\drive
0x01ac8570 dmboot           False False False False  False False dmboot.sys
0x01af6670 gaopdxserv.sys   False False False False  False False None
```

Out of ~113 drivers identified, nine were not in the baseline image (one service was malicious)

Automating Analysis: `driverbl`

In this example, an approximate count of the total drivers found by the **`modules`** plugin was taken using the **`wc`** command (the result was 113). Subsequently, **`driverbl`** was run and provided the suspect image (TDSS.img) and the known good image (baseline.img). In this case, we were looking for drivers that existed in the suspect image but not in the baseline image, so the –U option was provided to show only unknown services. Of the nine drivers identified as "unknown," one turned out to be malicious—gaopdxserv.sys. Although using a baseline image can greatly cut down on the amount of data necessary to analyze, there will still be false positives that must be eliminated! In this case, gaopdxserv.sys and dmboot immediately were interesting because their paths were different than the others. Dmboot was legitimate, whereas gaopdxserv turned out to be a known driver associated with the TDSS rootkit.



Rootkit Detection: `apihooks`

As techniques like IDT and SSDT hooks are increasingly scanned for and blocked by system protection applications, attackers have turned to new methods. We discussed Import Address Table (IAT) hooks in the Redline rootkits section. Inline/trampoline hooks are similar to IAT hooks in that they both intercept process function calls. The primary difference is that instead of manipulating the import table address in the process, which can be easily identified and has some limitations, inline hooks modify the functions themselves, adding a jump (or equivalent) to malicious code. For truly devious malware, these changes can be made only on the copies in memory, never touching the disk.

Similar to what we saw with IRP hooks in Redline, the output from the **`apihooks`** plugin can be difficult to manage. A large number of legitimate hooks exist. Examples of legitimate hooking applications are svchost, vmtools, anti-virus, and codemeter/hasp copy protection mechanisms. Focusing your analysis only on suspected processes using the "-p" option can help limit the data set. Also, Brian Milliron compiled a list of Microsoft DLLs known to legitimately hook other functions in his SANS SCORE guide to investigating rootkits.[1] Keeping these "legitimate" hooks in mind can help identify the many false positives reported :

- setupapi.dll
- mswsock.dll
- sfc_os.dll
- adsldpc.dll
- advapi32.dll
- secur32.dll
- ws2_32.dll
- iphlpapi.dll
- ntdll.dll
- kernel32.dll
- user32.dll
- gdi32.dll

This plugin is an example of functionality present within Volatility that does not exist in Redline. A combination of both tools is a good best practice.

[1] https://www.sans.org/score/checklists/rootkits-investigation-procedures

Rootkit Detection: Zeus Inline DLL Hooking

root@SIFT-Workstation:/# **vol.py -f /memory/zeus.img apihooks**
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 676 (services.exe)
Victim module: USER32.dll (0x77d40000 - 0x77dd0000)
Function: USER32.dll!GetClipboardData at 0x77d6fcb2
Hook address: 0x7f4fd5
Hooking module: <unknown>

Disassembly(0):
0x77d6fcb2 e91e53a888 JMP 0x7f4fd5
0x77d6fcb7 83ec2c SUB ESP, 0x2c
0x77d6fcba 56 PUSH ESI
...SNIP... **Zbot hooks nearly 400 DLL functions!**

Rootkit Detection: Zeus Inline DLL Hooking

Zeus malware was not designed to be stealthy—at least not to a forensic examiner armed with memory analysis capabilities! When analyzing this Zeus infected image with the **`apihooks`** plugin, the sheer number of hooks (nearly 400) would be a big clue that something is amiss. Also, the large number of "unknown" values for "Hooking module" is concerning. "Hooking module" should indicate the DLL that the "JMP" instruction is jumping into. "Unknown" means that it is jumping into a memory section that is not mapped (backed with a file on disk), indicating it might be an injected memory section. Also notice which functions are being hooked. Although it is of great advantage to be a Windows internals guru when looking at output like this, you don't have to be an expert to get an idea that a lot of very important functions have been modified. Why does the "services.exe" process need hooks for HTTPQueryInfo and HTTPSendRequest? Or how about GetClipboardData? With experience looking at other systems, these will start to look suspicious. The latest version of apihooks includes multiple-hop disassembly, meaning it will follow jumps in the code and provide sample assembly code at each step. This is not shown on the slide due to space limitations, but can be seen in the following example. The code for the GetClipboardData function within USER32.dll has been patched to include an immediate jump to offset 0x7f4fd5, which is not mapped as a library in memory and hence listed as <unknown> [Disassembly(0)]. Code execution continues after the jump in Disassembly(1).

Example:

root@SIFT-Workstation:/# **vol.py -f /memory/zeus.img apihooks**

..SNIP..

****

Hook mode: Usermode

Hook type: Inline/Trampoline

Process: 676 (services.exe)

Victim module: USER32.dll (0x77d40000 - 0x77dd0000)

Function: USER32.dll!GetClipboardData at 0x77d6fcb2

Hook address: 0x7f4fd5

Hooking module: <unknown>

Disassembly(0):

0x77d6fcb2 e91e53a888 JMP 0x7f4fd5

0x77d6fcb7 83ec2c SUB ESP, 0x2c

0x77d6fcba 56 PUSH ESI

0x77d6fcbb 57 PUSH EDI

0x77d6fcbc 8d45d4 LEA EAX, [EBP-0x2c]

0x77d6fcbf 50 PUSH EAX

0x77d6fcc0 ff7508 PUSH DWORD [EBP+0x8]

0x77d6fcc3 e8e8000000 CALL 0x77d6fdb0

0x77d6fcc8 8bf0 MOV ESI, EAX

Disassembly(1):

0x7f4fd5 55 PUSH EBP

0x7f4fd6 8bec MOV EBP, ESP

0x7f4fd8 53 PUSH EBX

0x7f4fd9 56 PUSH ESI

0x7f4fda e859ebfeff CALL 0x7e3b38

0x7f4fdf 8b7508 MOV ESI, [EBP+0x8]

0x7f4fe2 56 PUSH ESI

0x7f4fe3 ff1560137e00 CALL DWORD [0x7e1360]

0x7f4fe9 8bd8 MOV EBX, EAX

0x7f4feb 85db TEST EBX, EBX

…SNIP…

Rootkit Detection: Review

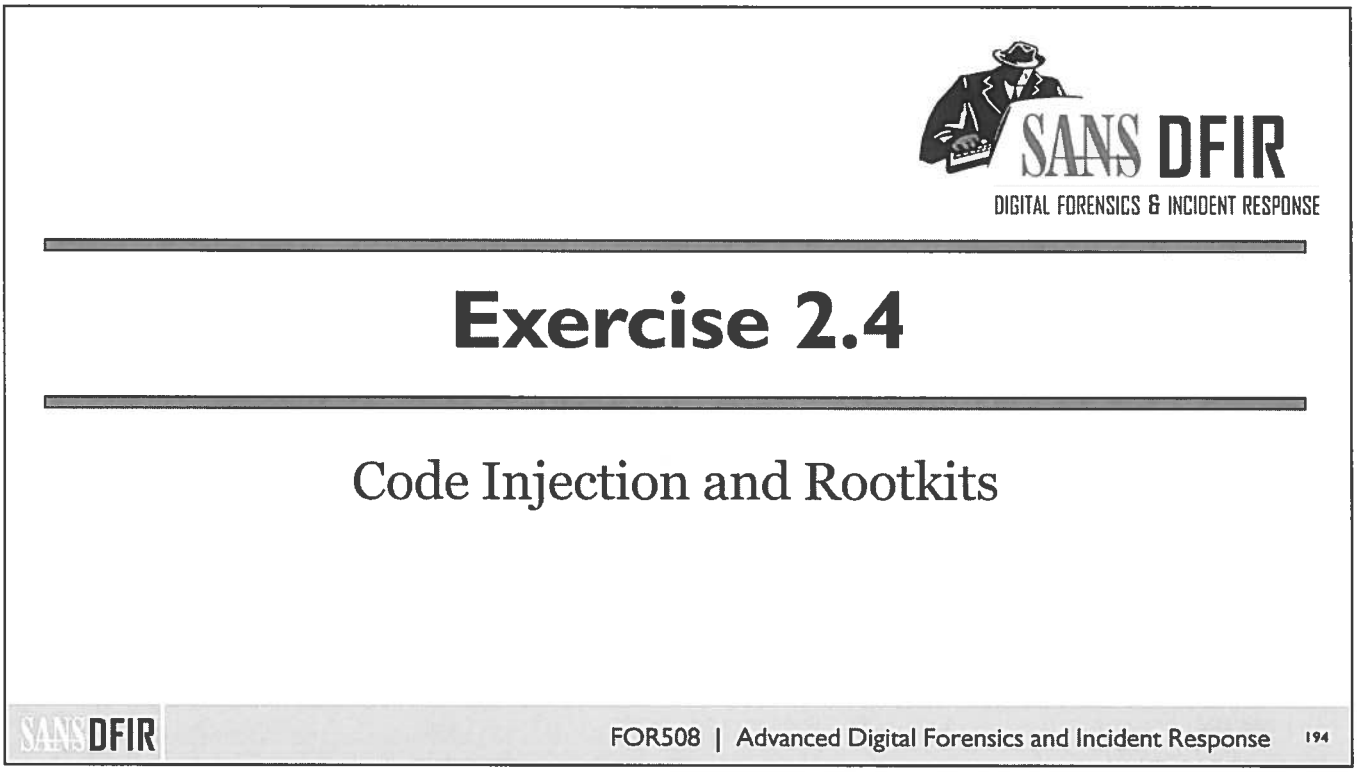
- **Rootkits hide the existence of system objects like processes, files, registry keys, and network artifacts**
- **A variety of system locations can be hooked:**
  - System Service Descriptor Table (SSDT)
  - Interrupt Descriptor Table (IDT)
  - Function Import Address Table (IAT)
  - I/O Request Packets (IRP)
- **Redline can identify SSDT, IDT, and IRP hooks**
- **Volatility is more capable at identifying hooking and unlinking:**
  - `ssdt` and `ssdt_ex`
  - `psxview`
  - `modscan` and `modules`
  - `apihooks`
  - `idt` and `driverirp`

Rootkit Detection: Review

Rootkits aim to hide process, files, registry keys, and network artifacts from users and incident responders. Although not as prevalent as code injection, malware rootkits can be more difficult to identify and diagnose. Redline attempts to identify malicious SSDT, IDT, and IRP hooks. Volatility has more rootkit detection features than available in Redline. We discussed several Volatility plugins to help identify hooking and unlinking:

- **`ssdt`:** Display System Service Descriptor Table entries.
- **`ssdt_ex`:** Identify and dump System Service Descriptor Table entries.
- **`psxview`:** Find hidden processes via cross-view techniques.
- **`modscan`:** Scan for loaded kernel drivers.
- **`modules`:** `List loaded kernel drivers.`
- **`apihooks`:** Find DLL function (inline and trampoline) hooks.
- **`Idt`:** Display Interrupt Descriptor Table hooks.
- **`driverirp`:** Identify I/O Request Packets (IRP) hooks.

Although we have the tools to identify rootkit hooking within our memory images, it is one of the harder analysis steps to complete. This is because hooking is not solely a malicious activity. A host of legitimate system processes and even protection applications like anti-virus or process monitors require hooking to perform their roles. Also, a small percentage of malware in the wild employ rootkits. For these reasons, we don't start our memory analysis process with hook detection. Often by the time you reach this step, you have already discovered some bad processes and are just trying to identify additional information about the attack.

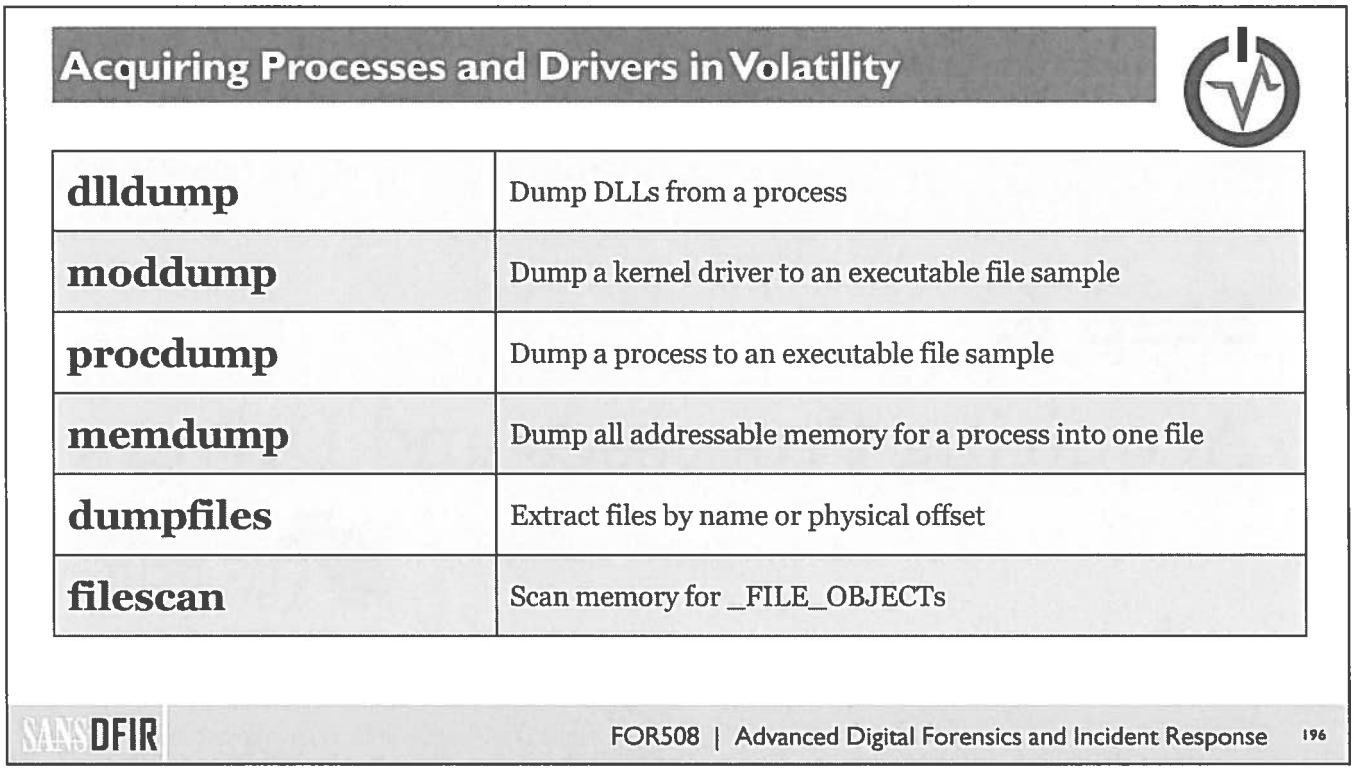


This page intentionally left blank.

Step 6:

Acquiring Processes and Drivers

This page intentionally left blank.

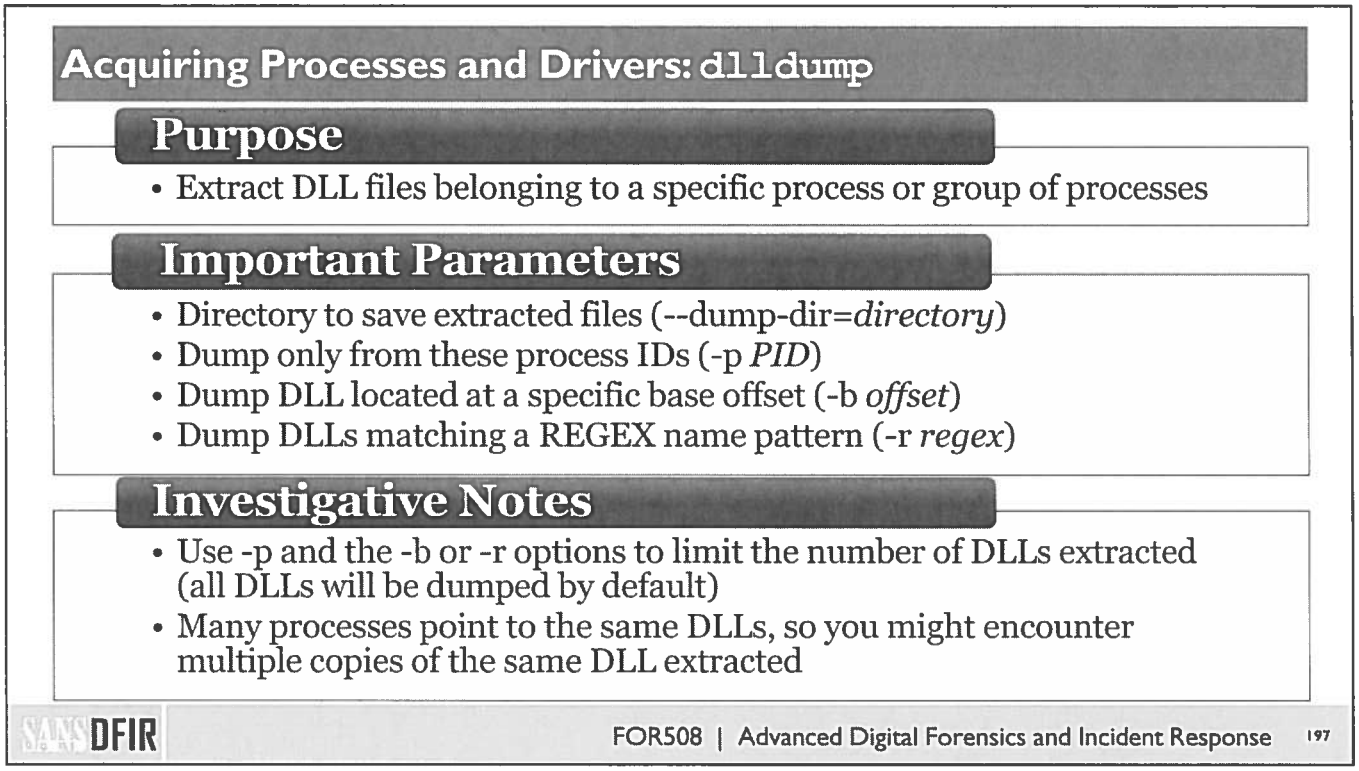


Acquiring Processes and Drivers in Volatility

The final step in our memory analysis process is to extract the suspicious processes and drivers from the memory image so we can perform additional analysis. This is a key step and a reminder that we can't always wrap up our memory analysis by using just one tool. Memory forensics can usually get us very close to understanding what was compromised and how the malware works, but a complete understanding of the malware and all of its indicators usually requires dynamic and static malware analysis. So once we have narrowed down our items for additional analysis, we will need to dump them. There are a lot of different ways this could be done and many possible results. Do we just want image executable so we can try dynamic analysis in a sandbox? Do we want the mapped DLLs and other files? Do we want all of the memory sections so we can look at those believed to have been injected? All these options and more are available in the Volatility framework.

Volatility has the most complete set of extraction features, so we will focus on it in the completion of this final step. There are many plugins to choose from and although a bit cumbersome, the single-function plugins provide a great deal of flexibility. The most important acquisition plugins for you to know are:

- `dlldump`: Dump DLLs from a process.
- `moddump`: Dump a kernel driver to an executable file sample.
- `procdump`: Dump a process to an executable file sample.
- `memdump`: Dump all addressable memory for a process into one file.
- `dumpfiles`: Extract cached files from memory.
- `filescan`: Search for file objects present in memory.



Acquiring Processes and Drivers: `dlldump`

The **`dlldump`** plugin is used to dump DLLs from one or more processes. By default, it will extract all DLLs in the memory image, so limiting it by selecting specific process ID (PIDS) is a good best practice. If you do not want all of the DLLs from a process, you can specify the DLL of interest using its Base Offset and the "-b" flag. The Base Offset can be found using the Volatility plugin **`dlllist`**. The Base Offset is a relative address, and thus this flag should usually be paired with "-p" specifying which process you would like to pull the DLL from. If you omit the "-p" information, it will dump DLLs from any process having one at that Base Offset. Alternatively, the "-r" parameter allows the DLL to be chosen by using a regular expression (REGEX) name pattern. REGEX patterns are case sensitive in Volatility. Examples of these two methods can be found in the following.

Keep in mind that there are no guarantees that all of a processes' DLLs will be contained in memory image. Dumping a DLL that has been paged out of memory at the time of the memory image will return an error.

Examples:

root@SIFT-Workstation:/# **vol.py -f memory.img dlllist -p 1012**

****

svchost.exe pid: 1012

Command line : C:\WINDOWS\System32\svchost.exe -k netsvcs

Service Pack 2

```
Base        Size        Path
0x01000000  0x006000    C:\WINDOWS\System32\svchost.exe
0x7c900000  0x0b0000    C:\WINDOWS\system32\ntdll.dll
...SNIP...
0x6fbd0000  0x03e000    C:\WINDOWS\System32\catsrv.dll
0x10000000  0x017000    C:\WINDOWS\system32\metsrv.dll
0x76bf0000  0x00b000    C:\WINDOWS\System32\PSAPI.DLL
```

root@SIFT-Workstation:/# **vol.py -f memory.img dlldump -p 1012 -b 0x10000000 --dump-dir=/output/**

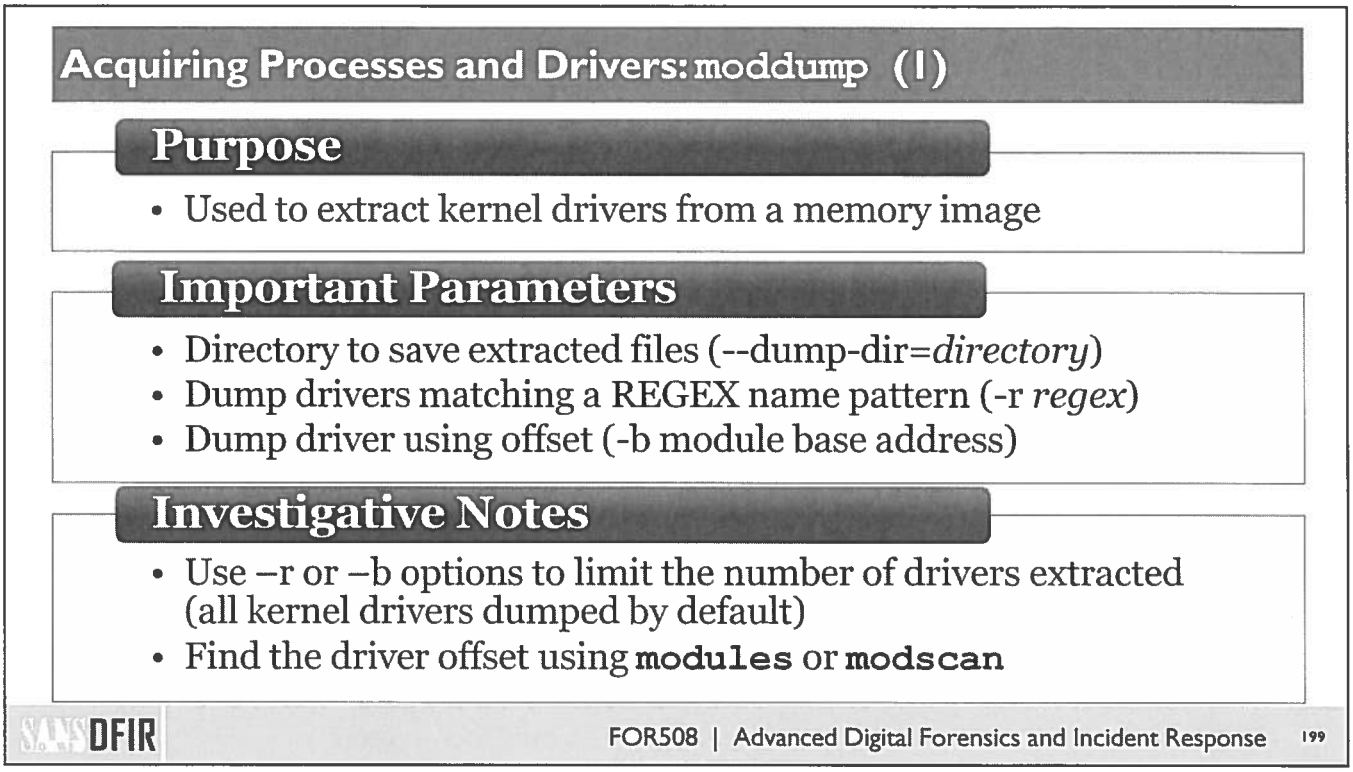
Dumping metsrv.dll, Process: svchost.exe, Base: 10000000 output: module.1012.1916d08.10000000.dll

Alternatively, we could have used the REGEX capability:

root@SIFT-Workstation:/# **vol.py -f memory.img dlldump -p 1012 -r metsrv --dump-dir=/output/**

Dumping metsrv.dll, Process: svchost.exe, Base: 10000000 output: module.1012.1916d08.10000000.dll

Metsrv.dll is a DLL used by the Metasploit Meterpreter.



Acquiring Processes and Drivers: `moddump` (1)

When you have identified potentially malicious drivers within a memory image, the next step would be to extract them from the image for further review and analysis. The plugin **`moddump`** was designed to make this process easy. First, we need to give it our output directory of choice by using the "--dump-dir" parameter. By default, **`moddump`** will dump all of the kernel drivers in the memory image. If you want to be more selective, you have two different means to select which drivers you care to dump. The "-r" flag allows drivers to be selected based on a case-sensitive regular expression matching their name(s). To dump the burito24b1-1710.sys malicious driver we found previously in the Storm Worm example, we might do the following:

root@SIFT-Workstation:/# **vol.py -f memory.img moddump -r burito --dump-dir=/output/**

Dumping burito24b1-1710.sys, Base: f7c24000 output: driver.f7c24000.sys

Notice that the following information is provided upon a successful extraction:

- Driver Name
- Driver Base Offset
- Output File Name

Alternatively, we can select the driver we want to dump using the "-b" parameter. This information might come from the Volatility plugin **`modscan`**, which provides a list of drivers, their base locations, and their size (among other things).

Acquiring Processes and Drivers: moddump (2)

modules used to find the offset address for dumping **burito24b1-1710.sys**

```
# vol.py -f memory.img modules
Offset(V)  Name                 Base             Size File
---------- -------------------- ---------- ---------- ----
0x817ca280 ParVdm.SYS           0xf9f12000     0x2000 \SystemRoot\Syste
0x818460b0 vmdesched.sys        0xf9d3c000     0x5000 \??\C:\WINDOWS\sy
0x818af5e0 vmmemctl.sys         0xf9d44000     0x7000 \??\C:\Program Fi
0x8171d3a0 burito24b1-1710.sys  0xf7c24000    0x20000 \??\C:\WINDOWS\sy
0x818b1330 srv.sys              0xf7ba9000    0x53000 \SystemRoot\syste
0x81891358 HTTP.sys             0xf7988000    0x41000 \SystemRoot\Syste

# vol.py -f memory.img moddump -b 0xf7c24000 --dump-dir=./output
Module Base Module Name          Result
----------- -------------------- ------
0x0f7c24000 burito24b1-1710.sys  OK: driver.f7c24000.sys
```


Acquiring Processes and Drivers moddump (2)

A very common way to use the **moddump** plugin to dump drivers is by using the driver's offset location within memory. When a driver is identified, use the Volatility plugins **modules** or **modscan** to identify the driver offset. For instance, imagine you find several SSDT hooks related to a specific driver. The offset information provided by the **ssdt** plugin is for the SSDT function location, not the driver offset. To get the driver offset, you would run **modules**, and filter the results for the name of the driver found in your **ssdt** results. Finally you would run **moddump** using the base location of the driver (the "Base" offset provided by **modules)**. The resulting driver would be located in your "--dump-dir." For example:

root@SIFT-Workstation:/# **vol.py -f memory.img ssdt | egrep -v '(ntoskrnl|win32k)'**

Gathering all referenced SSDTs from KTHREADs...

Finding appropriate address space for tables...

SSDT[0] at 80501030 with 284 entries

Entry 0x0047: 0xf7c24800 (NtEnumerateKey) owned by burito24b1-1710.sys

Entry 0x0049: 0xf7c24984 (NtEnumerateValueKey) owned by burito24b1-1710.sys

Entry 0x0091: 0xf7c244f4 (NtQueryDirectoryFile) owned by burito24b1-1710.sys

SSDT[1] at bf997600 with 667 entries

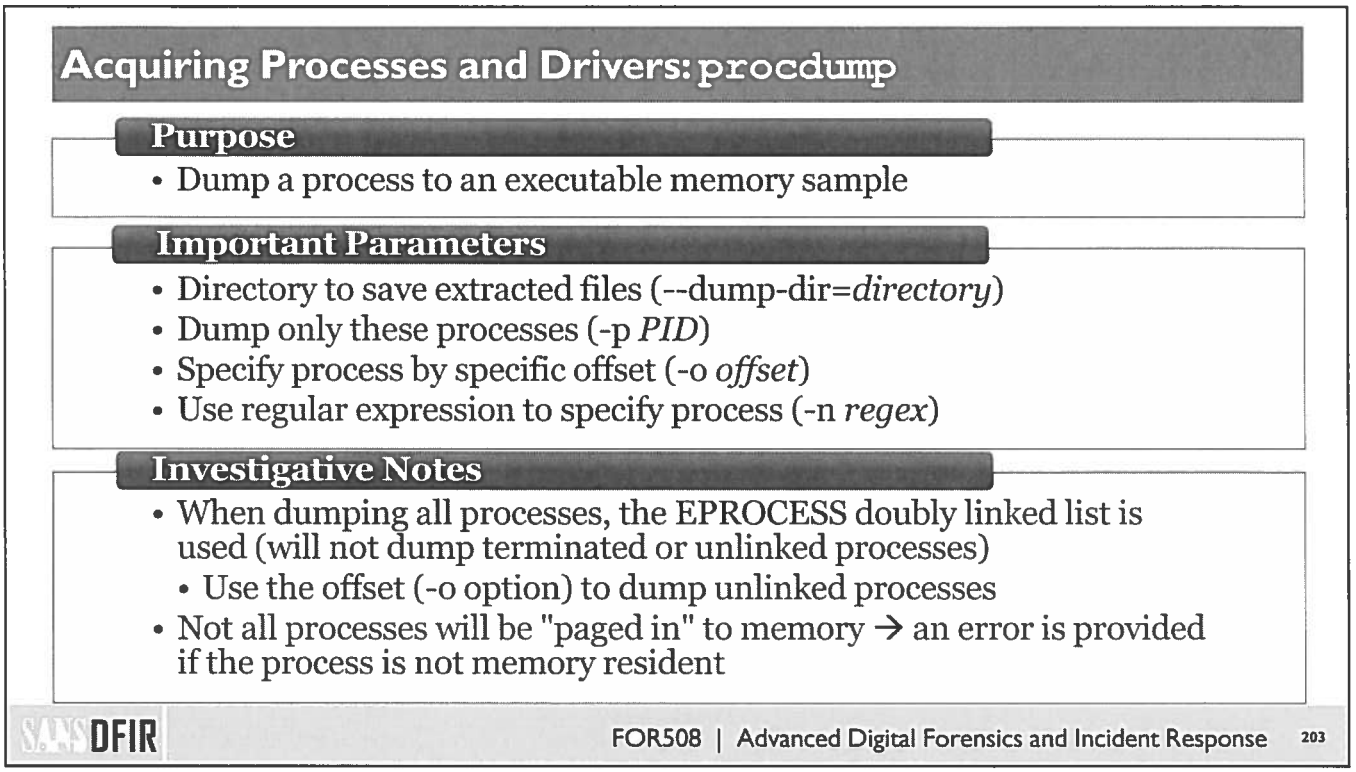
root@SIFT-Workstation:/# **vol.py -f memory.img modules | grep burito**

0x06ac83a0 burito24b1-1710.sys 0xf7c24000 0x20000 \??\C:\WINDOWS\system32\burito24b1-1710.sys

root@SIFT-Workstation:/# **vol.py -f memory.img moddump -b 0xf7c24000 --dump-dir= ./output**

Dumping burito24b1-1710.sys, Base: f7c24000 output: driver.f7c24000.sys

```
# vol.py -f memory.img modules
Offset(V)  Name                    Base             Size File
---------- -------------------- ---------- ---------- ----
0x817ca280 ParVdm.SYS           0xf9f12000     0x2000 \SystemRoot\Syste
0x818460b0 vmdesched.sys        0xf9d3c000     0x5000 \??\C:\WINDOWS\sy
0x818af5e0 vmmemctl.sys         0xf9d44000     0x7000 \??\C:\Program Fi
0x8171d3a0 burito24b1-1710.sys  0xf7c24000    0x20000 \??\C:\WINDOWS\sy
0x818b1330 srv.sys              0xf7ba9000    0x53000 \SystemRoot\syste
0x81891358 HTTP.sys             0xf7988000    0x41000 \SystemRoot\Syste

# vol.py -f memory.img moddump -b 0xf7c24000 --dump-dir=./output
Module Base Module Name          Result
----------- -------------------- ------
0x0f7c24000 burito24b1-1710.sys  OK: driver.f7c24000.sys
```



Acquiring Processes and Drivers: `procdump`

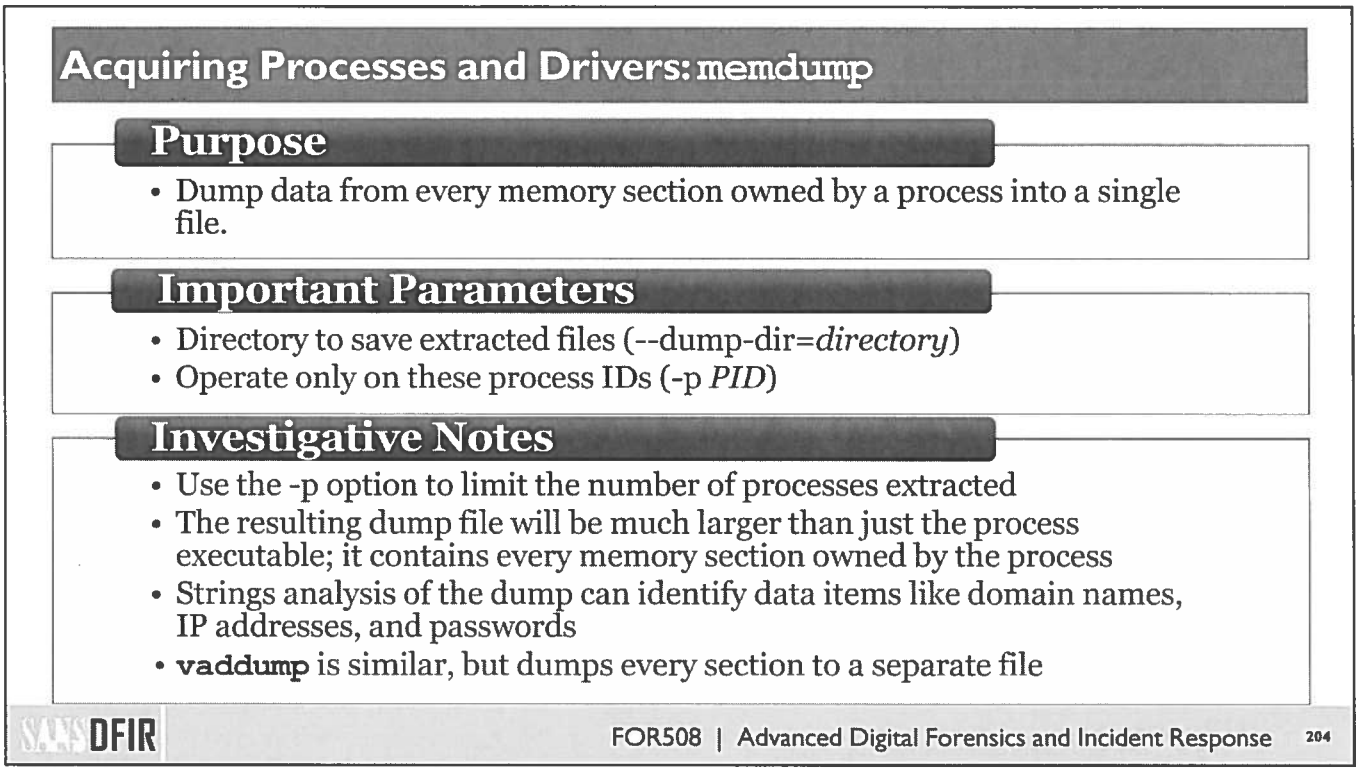
The **`procdump`** plugin facilitates extracting suspicious processes from the memory image for further review. By default, it will dump all of the processes within the EPROCESS doubly linked list. Knowing this, we should expect that it will not dump terminated or unlinked processes like the ones we might identify using the **`psscan`** plugin. To dump these processes, you will want to use the "-o" flag and specify the offset where the process exists in memory.

You might be familiar with the previous Volatility plugins **`procmemdump`** and **`procexedump`**. As of version 2.4, **`procdump`** combines the two into one plugin.

Following is an example of dumping all processes (notice how paging errors are enumerated):

vol.py -f memory.img procdump --dump-dir=/output/

```
Process(V)   ImageBase   Name                 Result
----------   ----------  -------------------- ------
0x8a603830   ----------     System            Error: PEB at 0x0 is unavailable (possibly due to paging)
0x8a4a23a0   0x48580000 smss.exe              OK: executable.876.exe
0x89946cf8    0x4a680000 csrss.exe            OK: executable.976.exe
0x8a161da0   0x01000000 winlogon.exe         OK: executable.1000.exe
0x8a15f890    0x01000000 services.exe         OK: executable.1044.exe
0x8a1ef978    0x01000000 lsass.exe            OK: executable.1056.exe
0x8a4bba08   0x00400000 vmacthlp.exe         OK: executable.1220.exe
0x8a35c3f0    0x01000000 svchost.exe          OK: executable.1236.exe
...SNIP...
```



Acquiring Processes and Drivers: `memdump`

The **`memdump`** plugin can be very useful when attempting to gather more information about a process. Unlike **`procdump`**, which just extracts the process executable, **`memdump`** walks the entire memory (VAD) tree for the process and collects every memory page belonging to that process in a single file. This output file will contain the executable code as well as any loaded DLLs (malicious or otherwise), memory mapped files, global kernel memory pages (shared across all processes), and other data. This file can be a lucrative place to search for things like domain names, IP addresses, user-typed data, and passwords. It can also be quite large. For instance, in the following example, dumping a suspicious svchost.exe process returned a dump file of ~80MB. Knowing this, use the "-p" flag to specify a comma-separated list of just the processes you care to examine outside the memory image.

`Memdump` has a sister plugin named **`vaddump`**. The two are nearly identical, with the only difference being that **`vaddump`** outputs the data by saving a separate file for each memory section owned by the process (as opposed to every section be placed in a single file). This can be helpful if you plan to run anti-virus or YARA signatures against the results.

Command-line example:

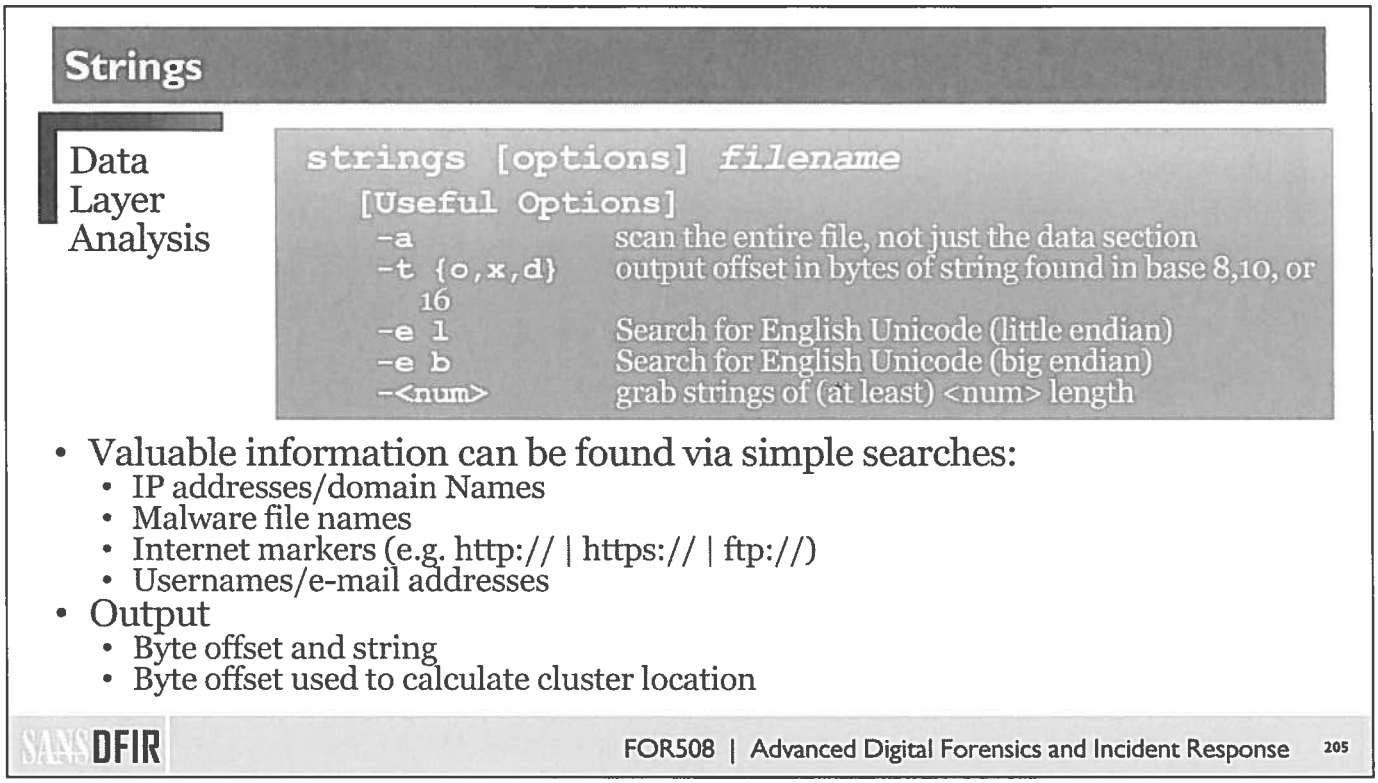
root@SIFT-Workstation:/# **vol.py -f memory.img memdump -p 1012 --dump-dir=/output/**

****

Writing svchost.exe [1012] to 1012.dmp

root@SIFT-Workstation:/# ll /output/1012.dmp

-rw-rw-rw- 1 root root **83574784** 2011-10-25 14:43 **/output/1012.dmp**



Strings

The strings tool is used to extract English ASCII and Unicode strings from a data stream. It is useful for a wide range of investigative efforts, including memory analysis, malware reverse engineering, and searching unallocated space of a disk image. Much of the evidence we are looking for on a system is represented as strings—IP addresses, domain names, file names, Internet communications, malware command and control packets, and even usernames and passwords. The strings output from a memory image would be a good place to start looking for words that one could use to open a locked zip file or Word document. It might even be a good baseline for a dictionary attack.

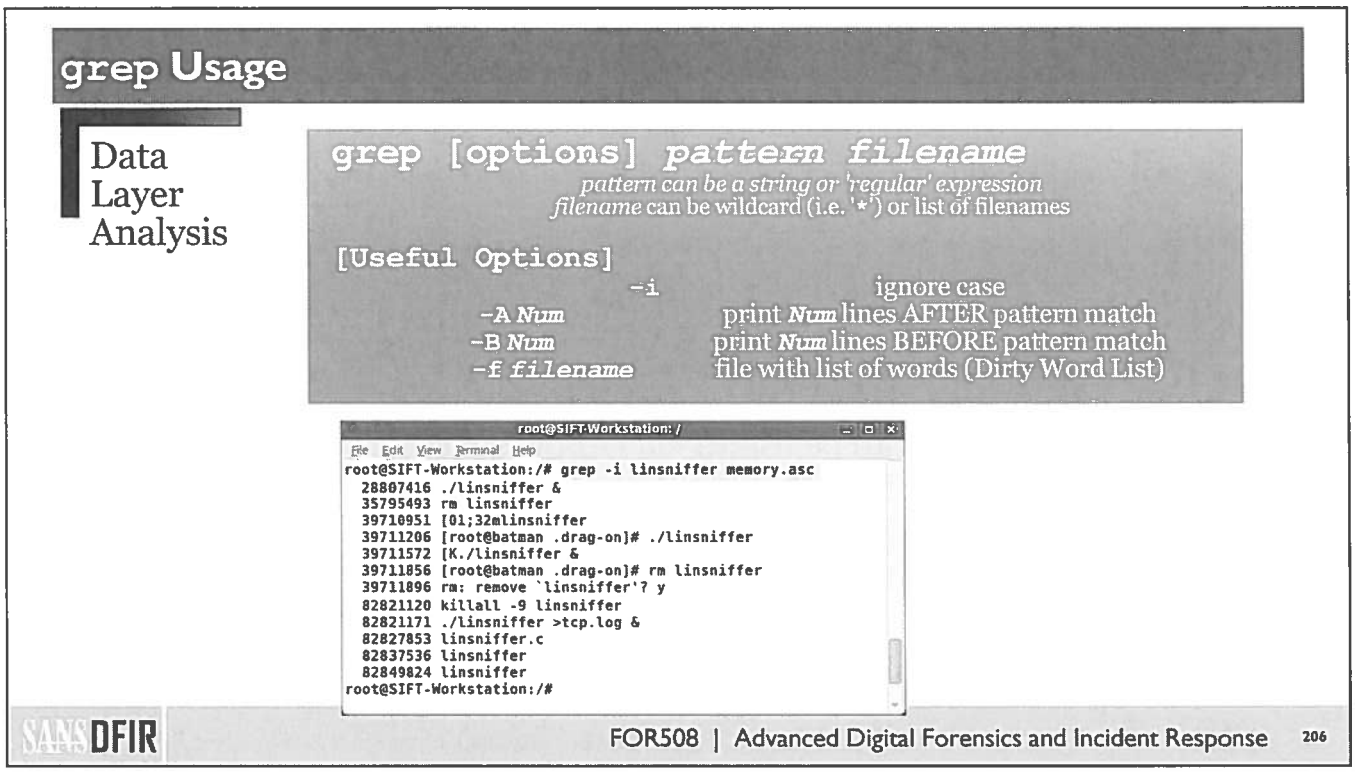
A good best practice is to use the –t d option in order to get the exact byte offsite (location) of the string in question. The byte offset will be the number listed at the beginning followed by the string. If you find a hit of interest, you can go back to that location in the original data stream and look at other information around it to determine context.

Standard practice often has us running strings twice: once for Unicode strings (with the –e l option) and once for ASCII strings. Once generated, these two files can be combined into a single strings file. Some simple command lines follow:

```
strings -a -t d file > strings.asc
strings -a -t d -e l file >> strings.uni
```

or

```
strings -a -t d file > strings.txt
strings -a -t d -e l file >> strings.txt
sort strings.txt > sorted_strings.txt
```



grep Usage

Once you have your strings file created, the only step left is to search. Grep, a tool commonly found natively in *nix environments (like the SIFT Workstation), is a surprisingly efficient search tool. You should first extract all strings from a data set prior to executing the tool grep looking for pattern matches.

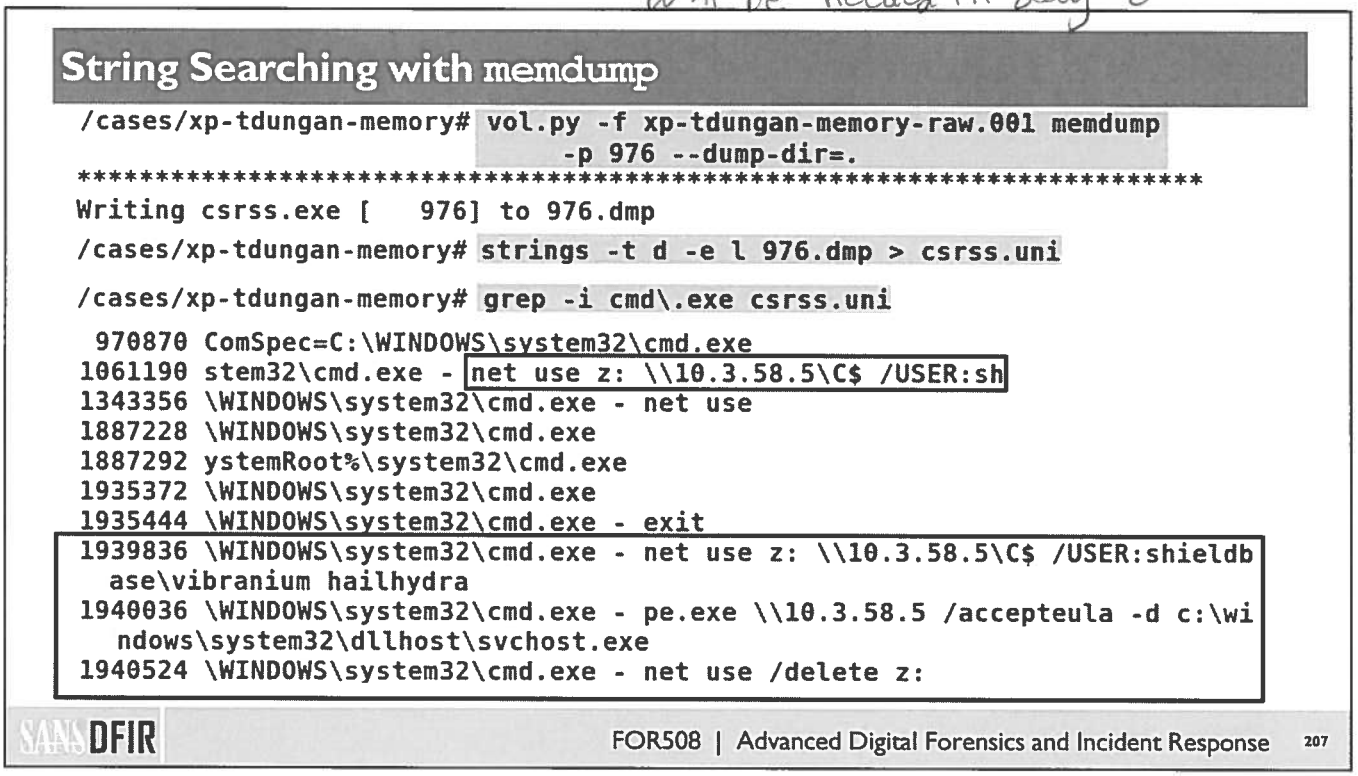
Of course, the real magic is knowing what to search for. Your keyword list should be a living document, growing and contracting as you learn more about whatever you are investigating. You can search for a single string (or regular expression) at a time, or if you have several, the "-f" option allows grep to search against a file containing a list of expressions. For those that are not masters of regular expressions, we use a neat little utility called Regex Coach.

In this example, we were investigating a hacked Linux system with a rootkit installed (before there were memory forensic tools for Linux). Live response actions showed a process running on the system named "linsniffer." A quick search of the ASCII strings of the memory image shows some interesting results. In particular, we see several commands likely run by the attackers. Among other things, these commands lead us to believe:

- An application named "linsniffer" was executed, possibly multiple times.
- A file or directory named linsniffer was deleted.
- An application named "linsniffer" was executed by the "root" account from a folder named ".drag-on."
- A process named "linsniffer" was halted (killall command).
- The output of an application named "linsniffer" was output to a file named "tcp.log."

That is a lot of information about our malware with very little effort!

will be needed in Day 6

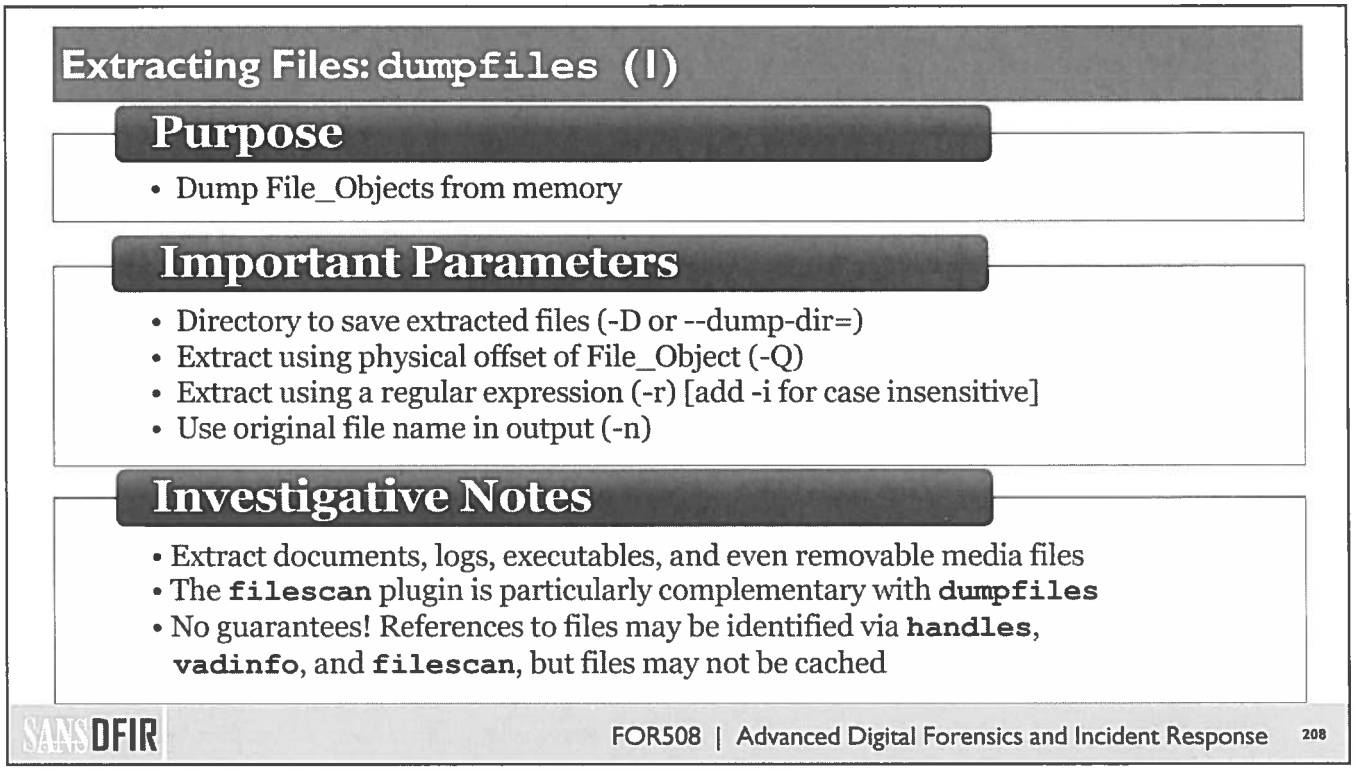


String Searching with memdump

Although there are some very powerful Volatility plugins to automagically pull the command history out of the memory (**cmdscan** and **consoles**), we think it is important to also know how to perform tasks manually. Nothing is guaranteed in memory and as such, that one marker your plugin is looking for might be paged out or somehow corrupted, returning incomplete results.

In this example, we used **memdump** to dump out the csrss process (PID 976) from the xp-tdungan-memory image. We then used a tool called **strings** to pull out English Unicode strings present in the process dump, saving them to a file named "csrss.uni." Finally, we used the **grep** command to start searching our new list of strings (searching for "cmd.exe" in this case). The strings found here should come as no surprise because you have likely already seen them while performing string searching in the Redline exercise. However, now we see the equivalent process using Volatility.

This example shows us looking for command history strings, but you could also use this technique (and regular expressions) to search for domains, IP addresses, and other malware indicators found during reverse engineering attacker's code. Although this slide demonstrates Unicode string searching, the same command can (and should) be used to pull out ASCII strings (just remove the -e l parameter): **strings -a -t -d 976.dmp > csrss.asc**.



Extracting Files: `dumpfiles` (1)

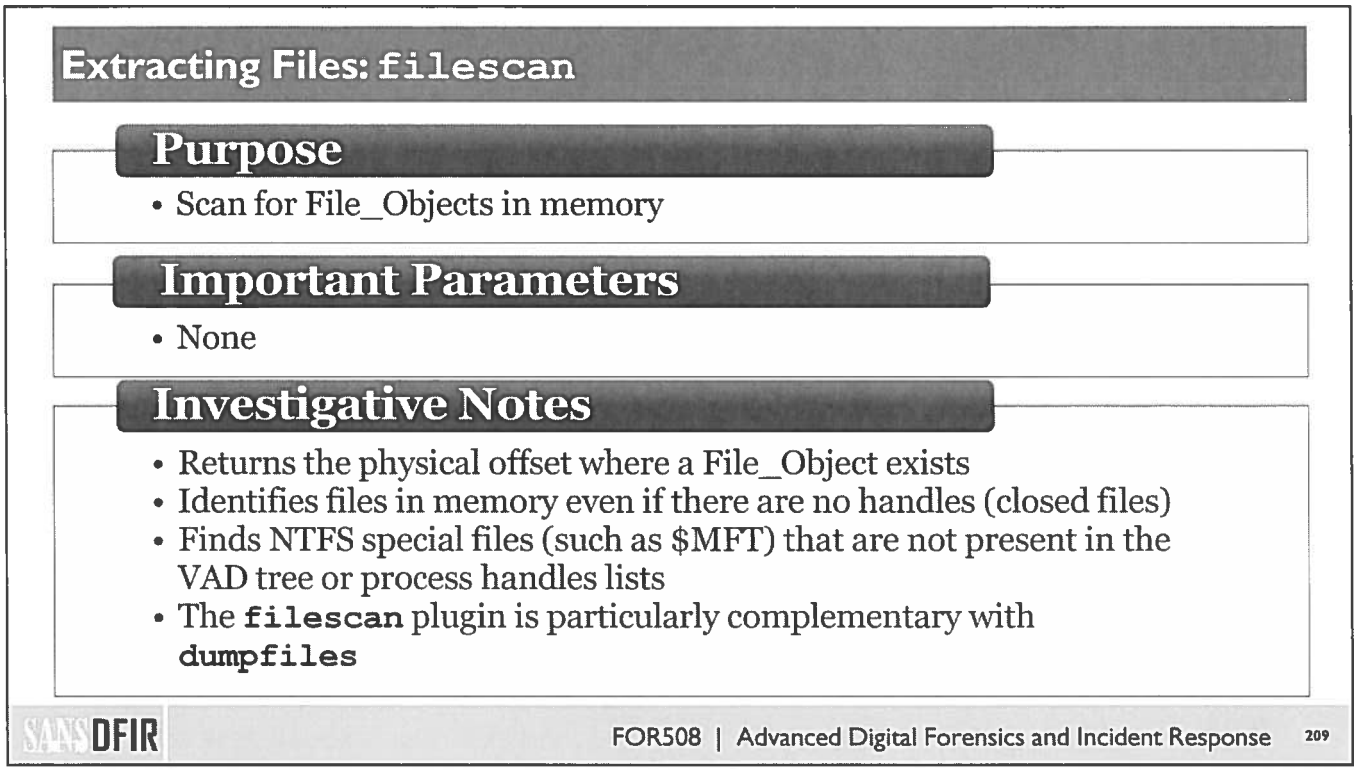
The **`dumpfiles`** plugin is an exciting addition to the Volatility suite. By default, it will attempt to extract all of the files currently mapped within the memory image. This collection will contain executables, DLLs, system files like registry hives and .dat databases, and even data files like text files or PDF and Office documents. This will be a large number of files, so we often use the –Q or –r parameters to limit the extraction to just what we are interested in. Because **`dumpfiles`** takes advantage of the memory structures that track how files are mapped in memory, file recovery using this tool is much more effective than others like file carving.

The **`dumpfiles`** plugin works by querying the process handle table and virtual address descriptor tree for File_Objects. Each File_Object contains three Section_Object_Pointers:

- **ImageSectionObject**: Points to memory mapped binaries (executables)—output will be saved as a .img file
- **DataSectionObject**: Points to memory mapped data files (such as document data stored by a Microsoft Word process)—output will be saved as a .dat file
- **SharedCachedMap**: Pointer to file parts cached by the Windows Cache manager—output will be saved as a .vacb file

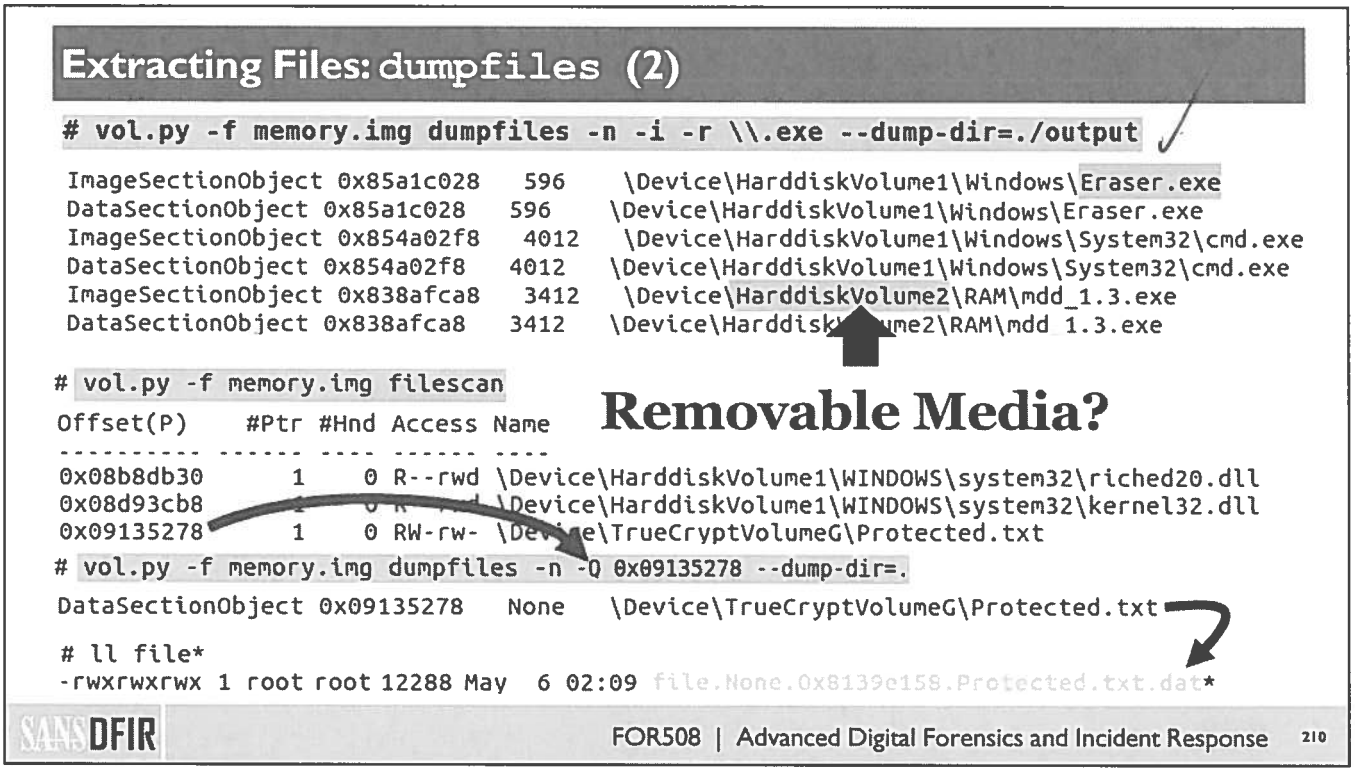
There are no guarantees when dealing with memory, and not every File_Object will have data currently memory mapped or cached. The file might be paged out, or only chunks of the file might be available (the plugin pads unavailable regions with zeroes). Hence, many file extractions result in corrupt or unreadable files. Sadly, some very important files like Prefetch and Registry hives are rarely available in Windows 7 and above (Registry hives can be extracted from XP systems using **`dumpfiles`**).[1] That doesn't mean data from those files isn't possibly in memory (reference the many Volatility Registry plugins that can still retrieve data from individual registry keys). It just means that those files are not able to be extracted using the File_Object Section_Object_Pointers that **`dumpfiles`** relies upon.

[1] https://media.blackhat.com/bh-us-11/Butler/BH_US_11_ButlerMurdock_Physical_Memory_Forensics-WP.pdf



Extracting Files: `filescan` (1)

`filescan` is a specialized plugin used to search for File_Object signatures in memory. It is particularly complementary to **`dumpfiles`** because it provides visibility into objects that might ordinarily be missed. For instance, **`dumpfiles`** identifies only File_Objects within the VAD tree or in process handles lists and hence will not recover closed or maliciously manipulated File_Objects. In addition, NTFS special files such as $MFT and $Logfile are not present in the VAD tree and will not be recovered by **`dumpfiles`** by default. Use **`filescan`** to do a more thorough job of file searching. If you find something of interest, record the physical offset of the File_Object found and use the **`dumpfiles`** –Q option to attempt recovery. As you are certainly aware of by now, there are no guarantees in memory forensics, so do not be too disappointed if your file of interest cannot be extracted! Even if your file cannot be recovered, remember that the existence of a file on a system can sometimes be just as useful. For instance, showing that a known malicious executable was once present on a system allows you to treat the system as compromised and triage accordingly.



Extracting Files: `dumpfiles` (2)

This slide illustrates two different ways to use **`dumpfiles`**. In the top example, we are using the **`dumpfiles`** plugin to extract all files containing ".exe" via the regular expression option (-r). This command will query the handles table and VAD trees to extract any files matching that regular expression to the dump directory (/tmp in this example). The slide shows several .exe files being extracted including Eraser.exe, cmd.exe, and mdd_1.3.exe. Although the standard output displays the full path information for each file, the naming of the resultant files can be quite confusing. There are two options in the plugin to make it easier to find the extracted files that you care about. The –n option includes the original file name in the extracted file name. There is also a –S option (not shown on the slide) that will create a summary text file that can help map extracted files back to their original filenames.

While reviewing the path information, also look at the devices listed. In this example, we see files present on both HarddiskVolume1 and HarddiskVolume2. It turns out that HarddiskVolume 2 was a removable device that was used by the incident responders to run their memory dumping tool (mdd_1.3.exe). Unfortunately, Windows uses the "HarddiskVolume" naming scheme regardless of the type of media, but numbering can identify files from additional volumes.

Pairing with `filescan`

By default, **`dumpfiles`** provides only access to files that are currently in the handle table or in the process VAD trees. However, there might be files that were once part of these objects, but have since been unlinked. The Volatility plugin named **`filescan`** can be used to identify such files. As the name suggests, **`filescan`** carves File_Objects from memory and can identify additional files still resident in the memory image. When paired together, **`filescan`** and **`dumpfiles`** form a very powerful combination.

In the second example on the slide, we see the **`filescan`** output showing an interesting file on a TrueCrypt encrypted volume (mounted as drive letter G). To extract a single file, we use the –Q option and provide the

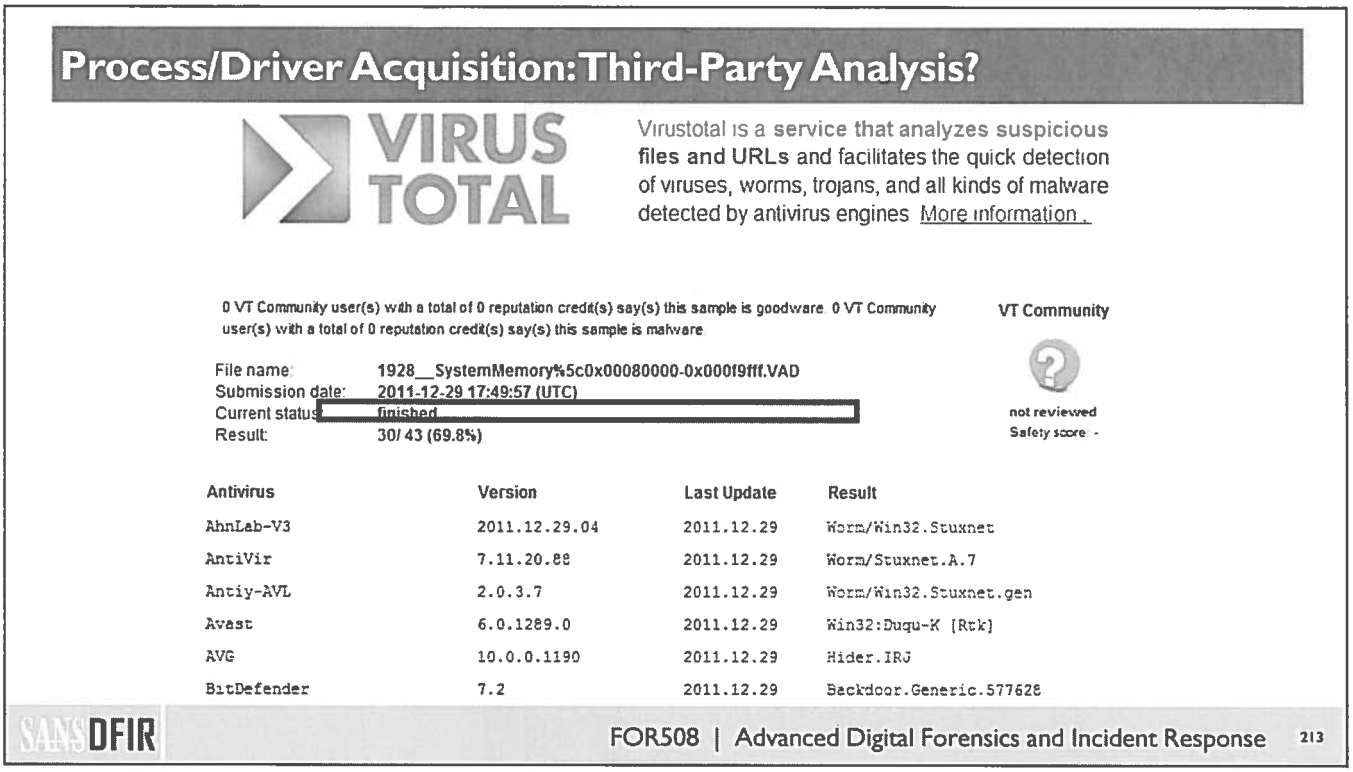
physical memory offset of the File_Object from the **filescan** output. In this case, we get lucky and it appears that the data pointed to by the File_Object Section_Object_Pointers is still available, and the file is successfully dumped from memory (reference the file listing at the bottom of the slide).

In summary, not only do we see interesting and important files being extracted via the **dumpfiles** plugin, we also see access to files that might no longer be present on the system itself! This slide shows evidence of files from both removable media and encrypted volumes. It can also be particularly useful for extracting components of memory-resident malware.

```
# vol.py -f memory.img dumpfiles -n -i -r \\.exe --dump-dir=./output

ImageSectionObject 0x85a1c028   596    \Device\HarddiskVolume1\Windows\Eraser.exe
DataSectionObject 0x85a1c028   596    \Device\HarddiskVolume1\Windows\Eraser.exe
ImageSectionObject 0x854a02f8   4012   \Device\HarddiskVolume1\Windows\System32\cmd.exe
DataSectionObject 0x854a02f8   4012   \Device\HarddiskVolume1\Windows\System32\cmd.exe
ImageSectionObject 0x838afca8   3412   \Device\HarddiskVolume2\RAM\mdd_1.3.exe
DataSectionObject 0x838afca8   3412   \Device\HarddiskVolume2\RAM\mdd_1.3.exe
```

Removable Media

```
# vol.py -f memory.img filescan
Offset(P)     #Ptr #Hnd Access Name
---------- ------ ---- ------ ----
0x08b8db30      1    0 R--rwd \Device\HarddiskVolume1\WINDOWS\system32\riched20.dll
0x08d93cb8      1    0 R--rwd \Device\HarddiskVolume1\WINDOWS\system32\kernel32.dll
0x09135278      1    0 RW-rw- \Device\TrueCryptVolumeG\Protected.txt
# vol.py -f memory.img dumpfiles -n -Q 0x09135278 --dump-dir=.
DataSectionObject 0x09135278   None   \Device\TrueCryptVolumeG\Protected.txt

# ll file*
-rwxrwxrwx 1 root root 12288 May  6 02:09 file.None.0x8139e158.Protected.txt.dat*
```



Process/Driver Acquisition: Third-Party Analysis?
Once you have your evil processes and drivers extracted, the next question is "Now what?" You have a whole host of options from the very low-tech of scanning the files with anti-virus signatures to the high-tech of studying the assembly code to understand instruction by instruction how the malware operates.

If you plan to scan with multiple anti-virus engines during your analysis, VirusTotal.com is an interesting option. In this example, we acquired and uploaded one of the known injected memory sections Redline identified in the Stuxnet memory image. The results were all over the map, from generic trojan to Duqu, to the actual hit on Stuxnet: Backdoor.Generic.577628, Trojan.Win32.Stuxnet, W32/MalwareF.JBBO, Win32:Duqu-K, and Packed.Win32.MUPX.Gen. However, 30/43 anti-virus engines flagged it as malicious, confirming our suspicions that the lsass.exe process is evil. One note of caution: Anything you upload to VirusTotal has now entered the public domain. If you are dealing with advanced attackers, you might want to be careful about tipping them off to malware that you might have found. It is possible that the attackers are using that malware only in your environment and that they could be checking for its appearance to determine whether they have been discovered in your network. A safer option (but still prone to information leakage) in VirusTotal is to select the **Search** tab and see whether anything with the same MD5 hash of your sample has been uploaded.[1]

There are also automated analysis tools that perform dynamic and some static malware analysis on samples. If you do not have reverse-engineering capabilities in your organization, these can be the next best thing. Threat Expert and GFI Sandbox are two of the most popular examples of free automated analysis tools.[2], [3]

Finally, if you are ready to take your analysis skills to the next level, consider the SANS FOR610 Reverse Engineering Malware course.[4]

[1] https://www.virustotal.com/documentation/searching/
[2] http://www.threattrack.com/
[3] http://www.threatexpert.com/submit.aspx
[4] http://digital-forensics.sans.org/courses/description/reverse-engineering-malware-malware-analysis-tools-techniques-54

Virustotal is a service that analyzes suspicious files and URLs and facilitates the quick detection of viruses, worms, trojans, and all kinds of malware detected by antivirus engines. More information...

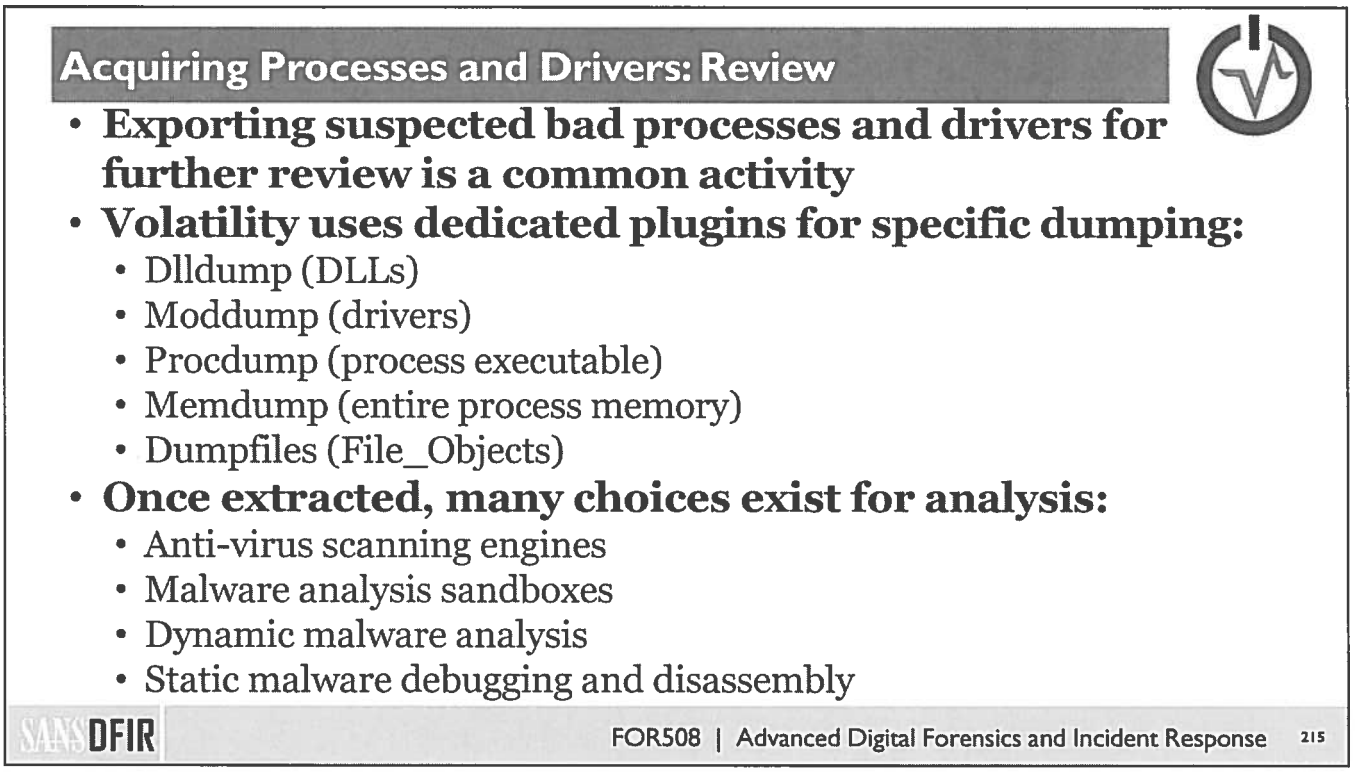
0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

VT Community

not reviewed
Safety score: -

File name: 1928__SystemMemory%5c0x00080000-0x000f9fff.VAD
Submission date: 2011-12-29 17:49:57 (UTC)
Current status: finished
Result: 30/ 43 (69.8%)

Antivirus	Version	Last Update	Result
AhnLab-V3	2011.12.29.04	2011.12.29	Worm/Win32.Stuxnet
AntiVir	7.11.20.88	2011.12.29	Worm/Stuxnet.A.7
Antiy-AVL	2.0.3.7	2011.12.29	Worm/Win32.Stuxnet.gen
Avast	6.0.1289.0	2011.12.29	Win32:Duqu-K [Rtk]
AVG	10.0.0.1190	2011.12.29	Hider.IRJ
BitDefender	7.2	2011.12.29	Backdoor.Generic.577628



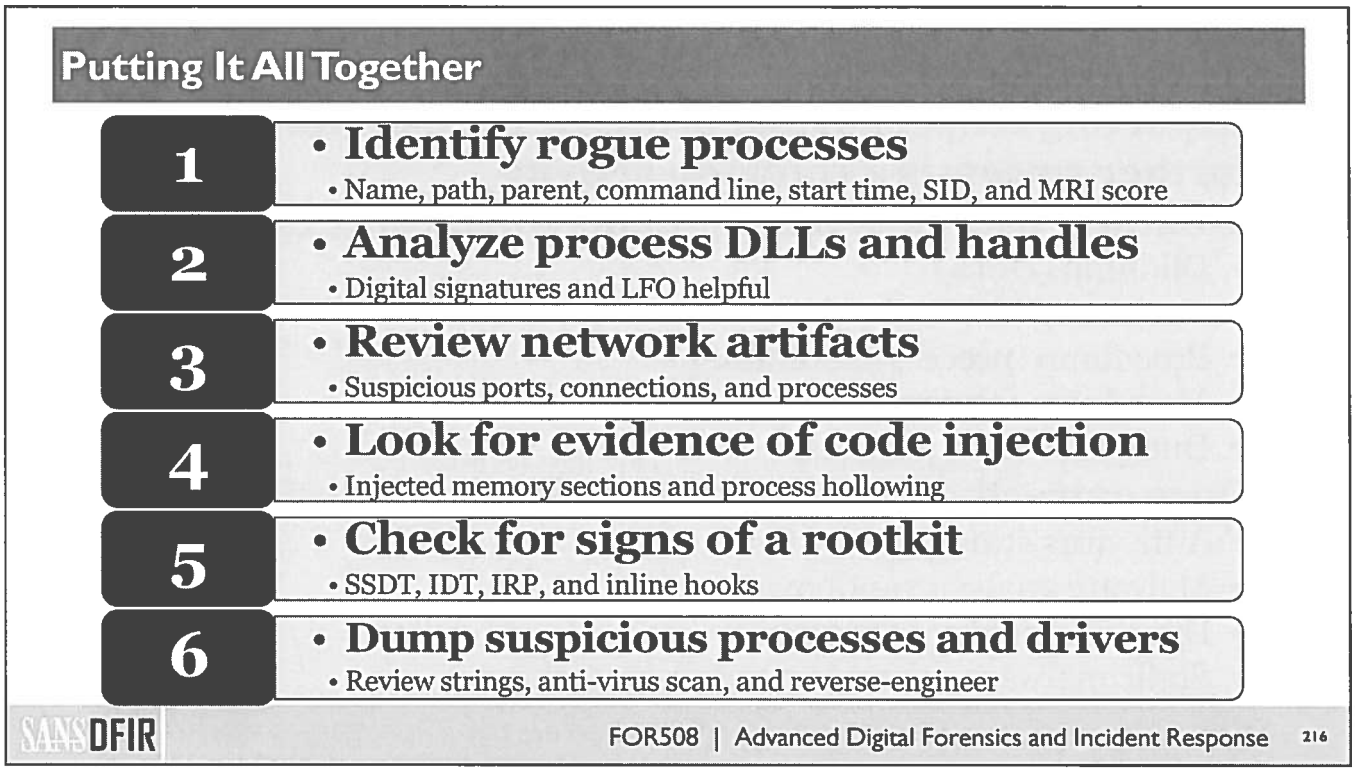
Acquiring Processes and Drivers: Review

Acquiring processes and drivers is the last step in our memory analysis process. Memory analysis can tell us many things, but it can't always tell us everything. Very commonly, we will need to extract processes, drivers, or memory sections out of the image to perform more in-depth analysis. It is very common to find three or four suspicious processes that you just aren't 100% sure about. Extracting them and getting confirmation from an anti-virus scan can help you determine you are on the right track.

Similar to what we have seen in other sections, Volatility has a large set of plugins that perform very specific dumping functions. While giving added power and flexibility, the sheer number of plugins can be overwhelming at first. We chose five Volatility plugins that would allow us to dump everything we have analyzed thus far:

- **dlldump:** Dump DLLs from a process.
- **moddump**: Dump a kernel driver to an executable file sample.
- **procdump**: Dump a process to an executable file sample.
- **memdump**: Dump all addressable memory for a process into one file.
- **dumpfiles**: Extract File_Objects from a memory image.

Once we have our files extracted, a wealth of options is available, ranging from submitting files to an anti-virus scan or automated threat scanning engine to reverse engineering the process or driver to uncover its traits.

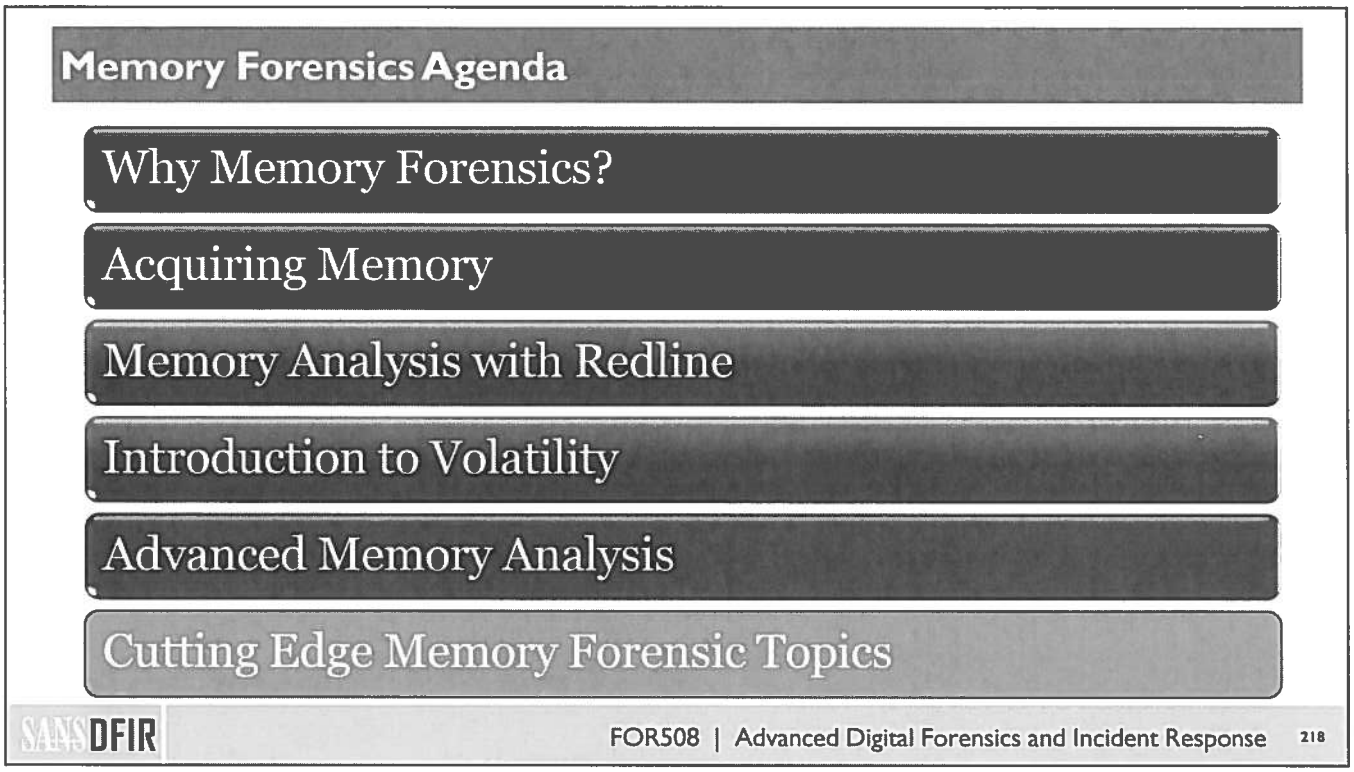


Putting It All Together

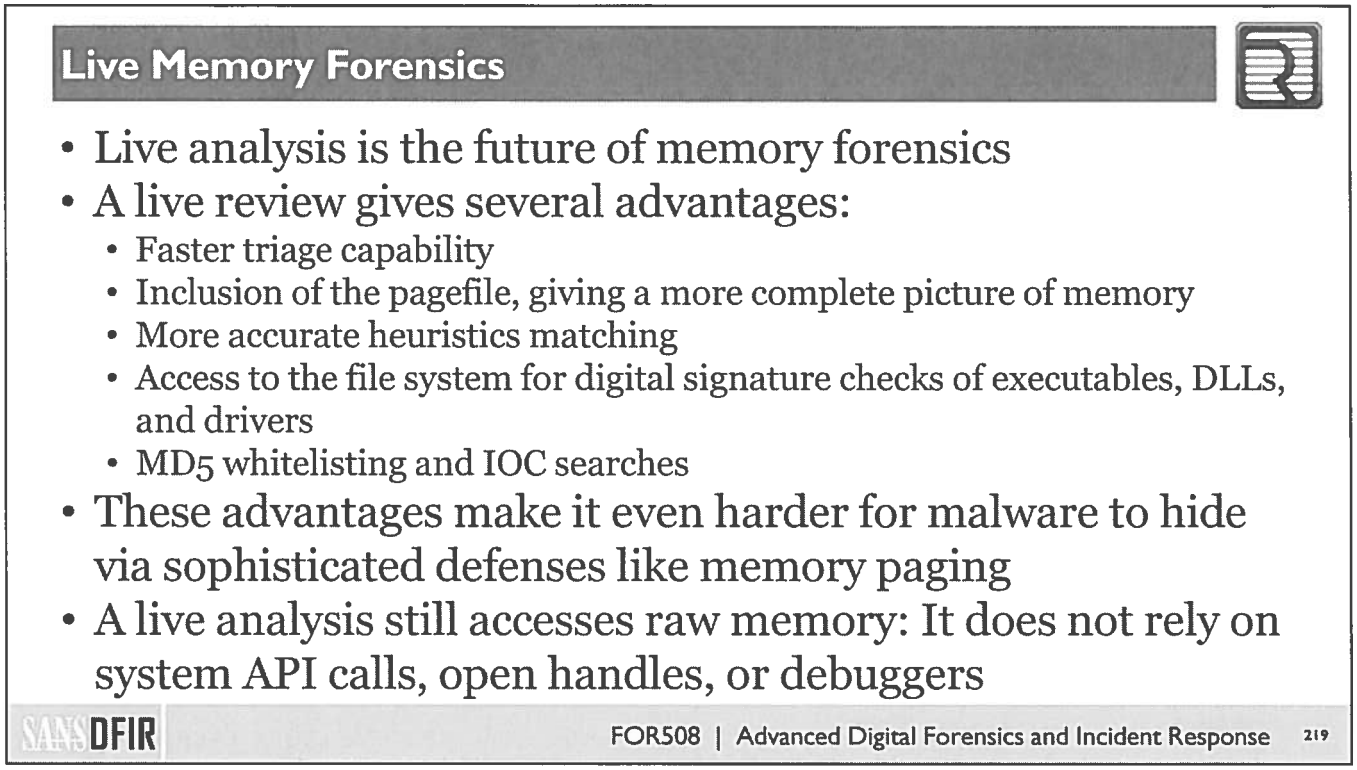
We have now covered all six steps in our malware analysis process. You have hopefully noticed that we take a layered approach, not expecting any one step to solve the case for us, but instead finding clues as we look at the different components that make up a complete memory image.

- **Identify rogue processes**: Reviewing processes is the obvious first step because processes are the most important construct in memory. By scrutinizing the image binary name, full path, parent process, command-line parameters, start time, security identifiers, and Redline MRI score, we have a good chance of finding suspicious processes early.
- **Analyze process DLLs and handles:** Our next step is to dig deeper into any suspicious processes and review the objects that define each process. The multitude of objects makes this a more challenging step, but Redline features like digital signature checks (live memory analysis only) and especially Least Frequency of Occurrence checks can be very helpful focusing our efforts.
- **Review network artifacts:** Network artifacts have long been a great place to identify strange activity on the system, and the best part of working with memory is that unlike on the running system, there is no way for the attacker to hide them. Analyzing behaviors like suspicious ports, network connections, and processes that should not be communicating can help identify evil processes.
- **Look for evidence of code injection:** Process hollowing and especially DLL injections are a very popular means for malware to hide. Luckily, it is currently quite easy for our memory analysis tools to detect these actions.

- **Check for signs of a rootkit:** Rootkits exist to hide processes and process objects and do this via a variety of different methods. The most popular are SSDT, IDT, IRP, and IAT hooks. Although some of these hooks are harder to find than others, memory analysis gives us an easy means to view and analyze hooked functions.
- **Dump suspicious processes and drivers:** The final step in our process is to extract our findings for further review. Once we have a list of suspicious process and drivers, we can dump them from the memory image and analyze via string searching, anti-virus scans, automated malware analysis engines, and manual reverse-engineering techniques.



This page intentionally left blank.



Live Memory Forensics

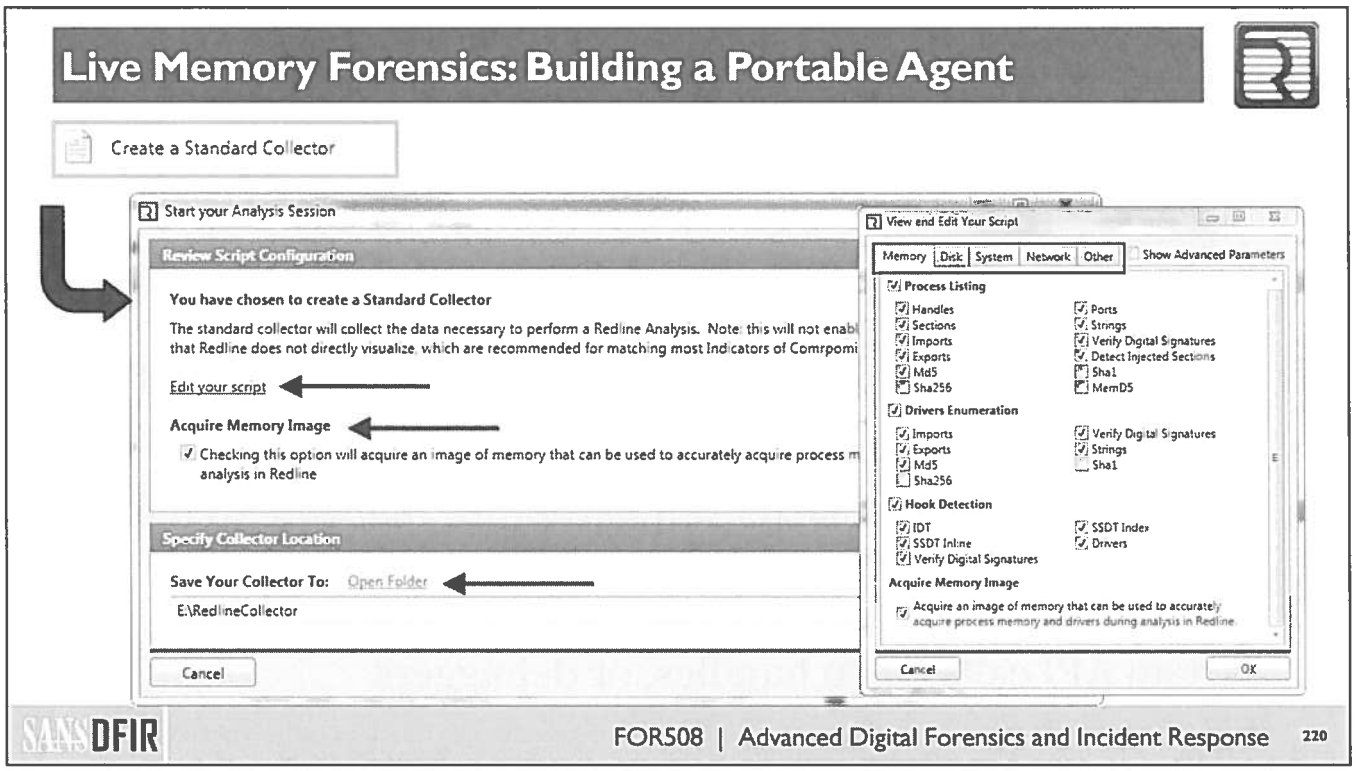
I credit the predecessor of Redline, Mandiant Memoryze, with popularizing the idea of performing live memory analysis, and believe it is a revolutionary change. The idea itself could be as controversial as creating a memory image was just a few years ago. Do you remember the naysayers questioning how our forensic analysis could possibly be valid if we were to run our memory imaging applications on the live system? Shouldn't we still be pulling the plug? What would they say if we now told them we were going to play "Find the Hacker" on that same live system? Luckily, it turns out that the system impact of doing a live analysis versus (or in addition to) taking a memory image is minimal. And the benefits are great:

- Faster triage capability
- Inclusion of the system pagefile, providing a more complete picture of memory
- More accurate heuristics matching
- Digital signature checks of process executables, DLLs, and drivers
- Comparison of process executables, DLLs, and drivers with "known good" MD5 whitelist
- Indicator of Compromise searches using pre-defined IOC files

Keep in mind that live analysis occurs by accessing physical memory, and *not* relying upon API calls, open handles, or debuggers. Thus, it is just as effective at defeating advanced malware and rootkits as analyzing a standard memory image. In fact, proof of concept code like Shadow Walker, which tries to page itself out of memory when a memory acquisition tool is detected, could be defeated through live analysis.[1]

To accomplish live memory analysis, our tool has to be more sophisticated than one used for standard memory acquisition. A memory audit must be conducted to identify all of those processes, drivers, and other artifacts we leverage during memory forensics. Redline allows you to build a configuration script for just this purpose. For the smallest footprint possible on the target system, consider running the audit from a USB device. Once the live analysis is complete, the results can be reviewed on your forensic workstation at your leisure.

[1] https://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf



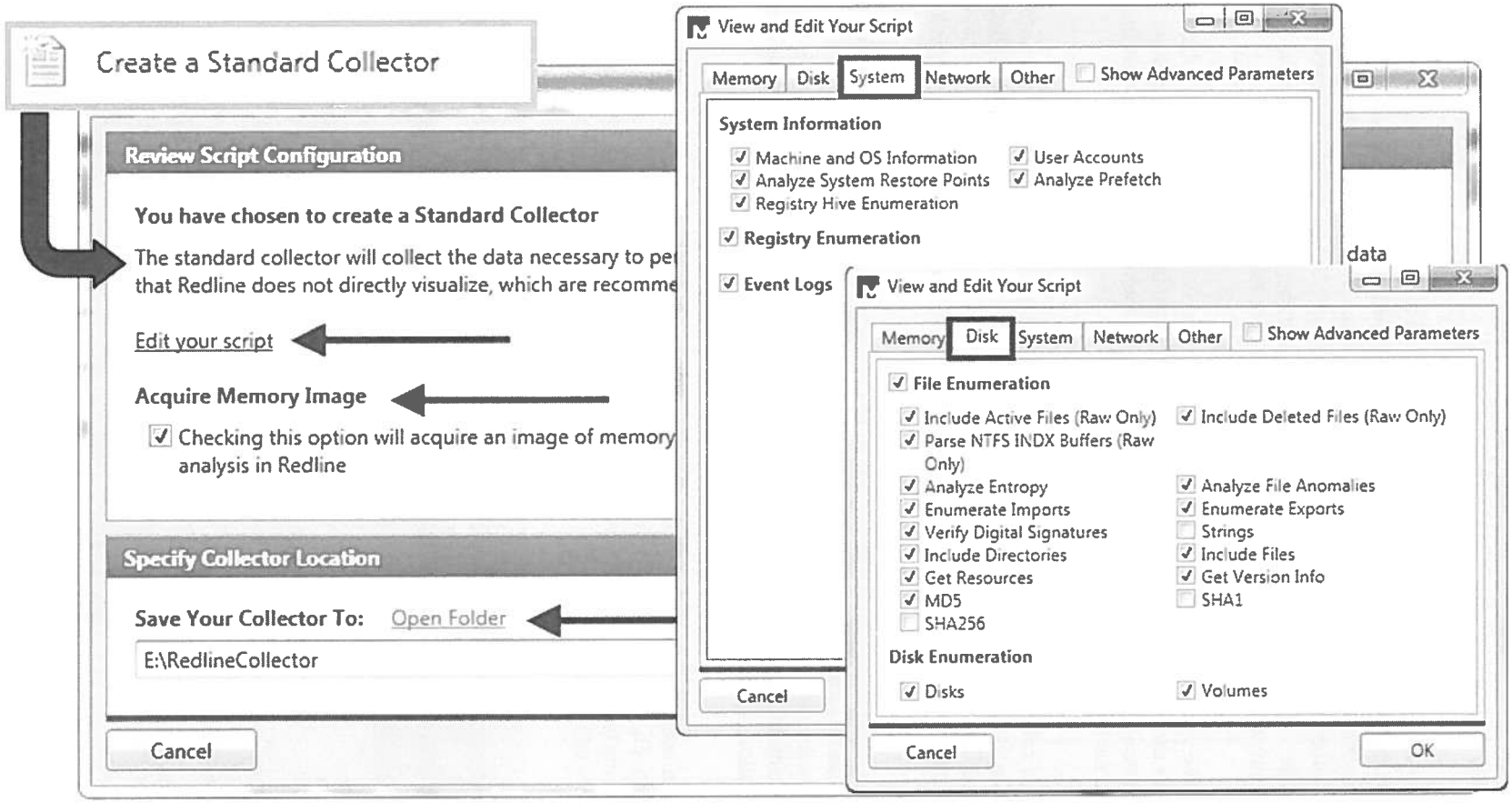
Live Memory Forensics: Building a Portable Agent

Mandiant Redline now makes it simple to build a powerful live response script. A "collector" is a portable agent created to gather the data necessary to perform an assessment of a live system. It is implemented as a batch script and designed to be run from removable media, such as a USB thumbdrive. The results are written back to the removable media and can be imported into Redline via the **From a Collector** option under the Analyze Data options.

When starting Redline, or clicking the Redline logo, you are presented with three different options:

- **Create a Standard Collector:** A standard collector collects the data necessary to conduct a full live memory analysis of the target system.
- **Create a Comprehensive Collector:** A comprehensive collector collects data necessary for memory analysis in addition to host-based artifacts such as file metadata, Windows Registry hives, Event Logs, and Prefetch files. This complete data set can then be used for more comprehensive searches, including system live response and employing indicator of compromise (IOC) searches. Note that this type of collection will take MUCH more time than the other options.
- **Create an IOC Search Collector:** Collect the minimal set of memory and host-based artifacts to perform a scan for JUST the selected indicators of compromise (IOCs) specified—a fast, targeted system probe. Memory analysis cannot be conducted with this minimal set.

In this slide, we have chosen **Create a Standard Collector**. Selecting **Edit your script** provides detailed collection options from Memory, Disk, System, Network, and Other items. Although a full analysis is nice to have (known as Comprehensive Collector), keep in mind that there are time and storage space tradeoffs. A recommended option to consider is **Acquire Memory Image**. This will allow you to gather a full memory image in addition to the pre-built live analysis data. With a full memory image, analysis can be conducted in Volatility and other memory analysis suites outside of Redline.


Create a Standard Collector
Review Script Configuration
You have chosen to create a Standard Collector
The standard collector will collect the data necessary to pe
that Redline does not directly visualize, which are recomme
Edit your script
Acquire Memory Image
Checking this option will acquire an image of memory
analysis in Redline
Specify Collector Location
Save Your Collector To:
Open Folder
E:\RedlineCollector
Cancel
View and Edit Your Script
Memory
Disk
System
Network
Other
Show Advanced Parameters
System Information
Machine and OS Information
User Accounts
Analyze System Restore Points
Analyze Prefetch
Registry Hive Enumeration
Registry Enumeration
Event Logs
Cancel
View and Edit Your Script
Memory
Disk
System
Network
Other
Show Advanced Parameters
File Enumeration
Include Active Files (Raw Only)
Include Deleted Files (Raw Only)
Parse NTFS INDX Buffers (Raw Only)
Analyze Entropy
Analyze File Anomalies
Enumerate Imports
Enumerate Exports
Verify Digital Signatures
Strings
Include Directories
Include Files
Get Resources
Get Version Info
MD5
SHA1
SHA256
Disk Enumeration
Disks
Volumes
Cancel
OK

Upon completion, the tool gives the following instructions:

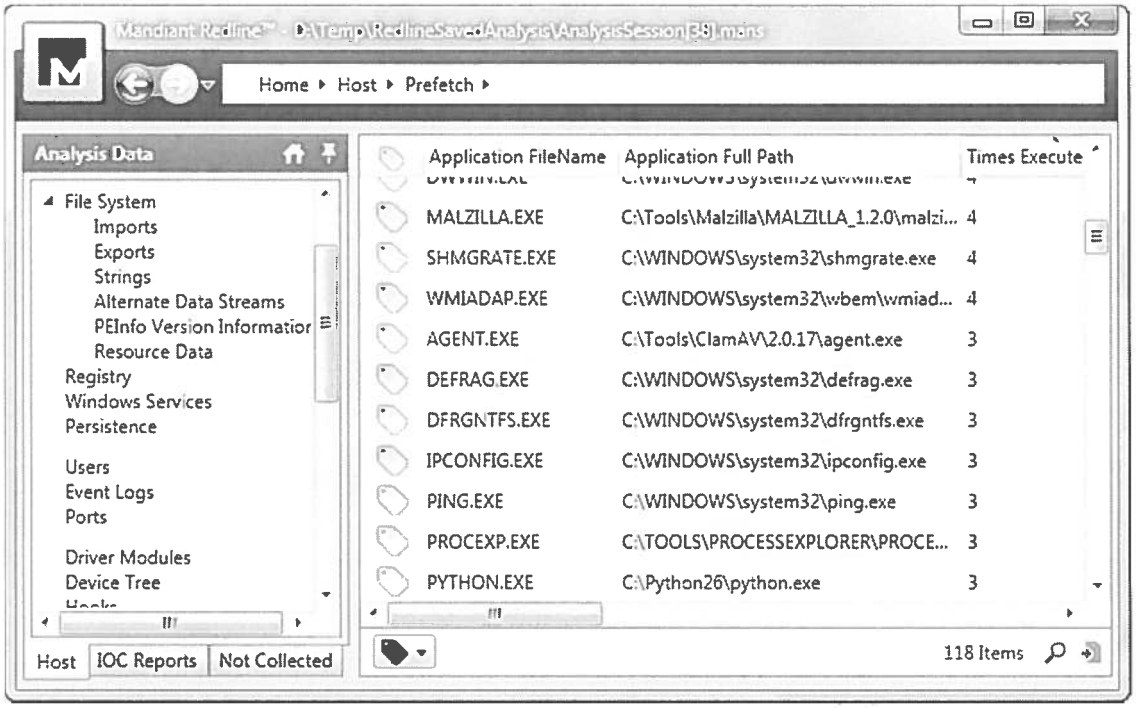
1. Your collector package will be created and saved to the location you specified
2. On the machine you want to audit, run the '**`RunRedlineAudit.bat`**' script, preferably from a removable media (for example, a USB Hard Drive). This script will run the collector as you configured it and save the results to a folder named "Audits" next to the script.
3. When the collection is finished, transfer the results back to your analysis machine, and use the **Analyze Collected Data** option on the start page or main menu.

The results will be stored under the same location as the Redline Portable files within a folder named "Audits." Multiple systems can be analyzed with the same portable agent, with each stored in a separate folder named by hostname and date of acquisition. If you are also collecting full memory images, make sure that you have enough space in the target location to hold the results!

Once the Redline collector has been run on a system, the resulting data can be loaded back into the Redline application for analysis. Simply select the logo in the top-left corner and select **Analyze Collected Data**. You will be prompted for the location of your collector's audit results (point it at the sub-folder named according to date and time) and given the option to also provide Indicators of Compromise to search for (more on these later).

Redline allows a surprisingly robust examination to be done on a system, all using a powerful, consistent interface. Data can be sorted by columns, filtered, and searched using regular expressions. If a "comprehensive" collector was chosen, the analyst will have nearly everything necessary to perform a detailed live response: detailed file system information, full copies of event logs and registry hives, prefetch, Windows Services, network information, live memory analysis data, and even browser history!

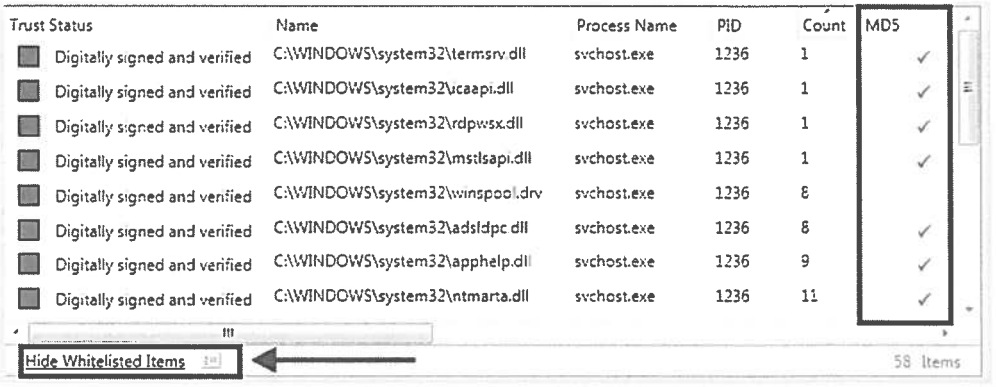
Keep in mind that although you might have full access to things like prefetch data, event logs, and registry hives, the tool does not store files in native form. All data is recorded by Redline in XML. This can be an advantage for some because XML is a very portable format for ingesting data into other analysis tools. Additionally, any table can be exported in .csv format from within the Redline console for review outside of the tool.



Although Redline collectors were really designed to be run on local systems, the resultant batch scripts can be pushed across the network and run remotely. Tom from the c-apt-ture blog has a nice post and script to accomplish this. These are located at http://c-apt-ure.blogspot.co.uk/2014/08/3r4lr-running-redline-remotely-for-live.html.

Whitelisting of Binaries

- ID "known good hashes" from live memory analysis
- **`Options → Whitelist Management`**
- Alert list for "known bad hashes" can be designated via IOC

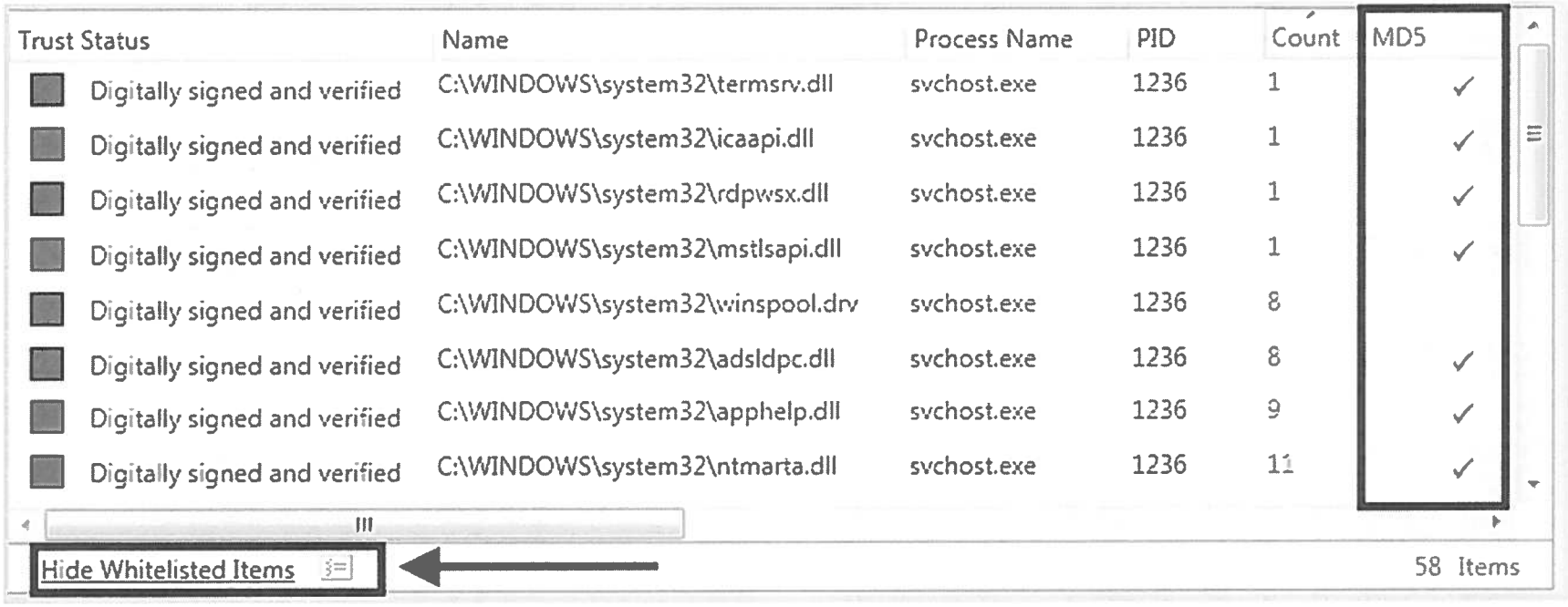


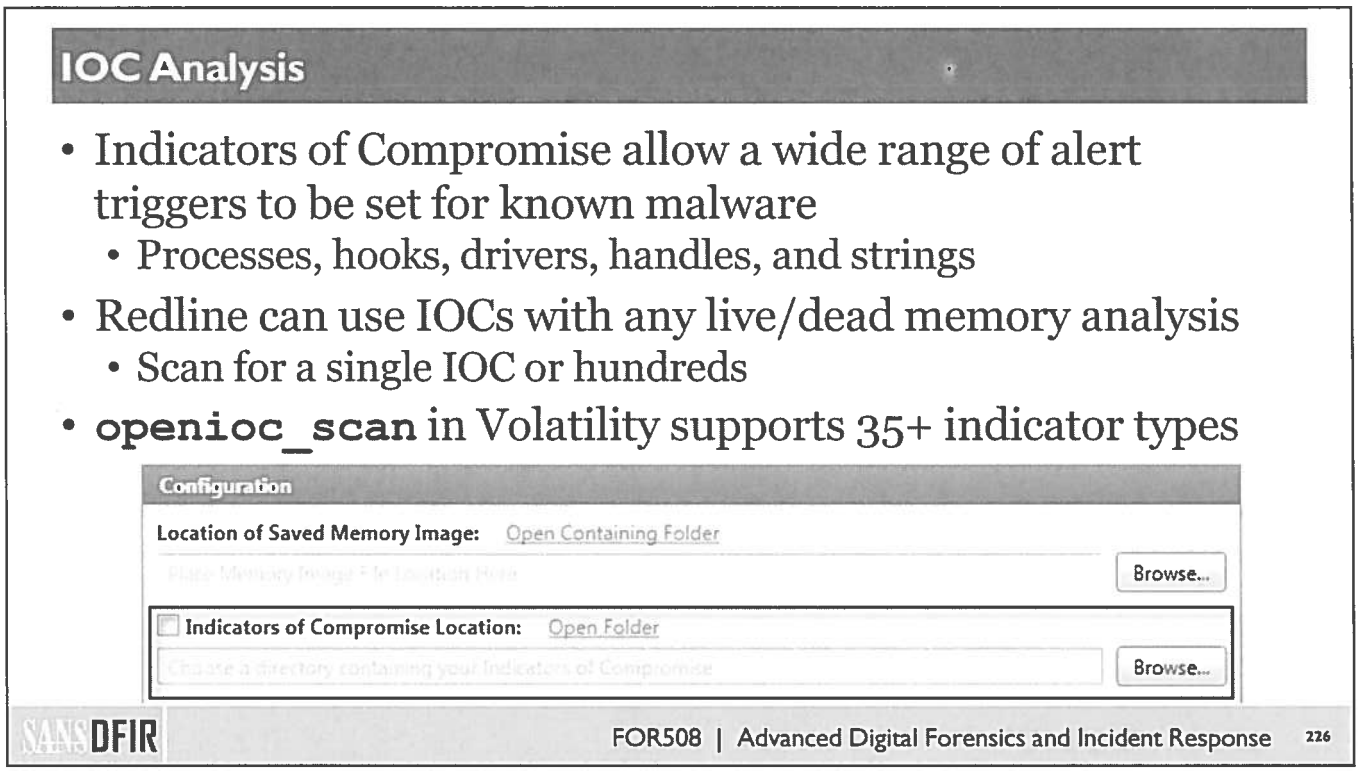
Whitelisting of Binaries

Redline has the ability to maintain a whitelist and include lookup information during memory analysis. This information will be presented only for live memory analyses conducted using the Redline collector scripts (both the Standard Collector and Comprehensive Collector provide the necessary information). The limitation exists because MD5 hash collection must take place using the file system because binaries loaded in memory are modified (and hence do not have a consistent MD5 hash). When the Redline collector output is loaded for analysis, the hashes are compared with the current whitelist and any matches are displayed in the MD5 column where relevant. Similar to digital signature checks, this can be a huge help with reducing the set of suspicious items. If a binary has both a valid digital signature AND is in a good whitelist database, the odds are quite low that it is malicious. As an example, the Stuxnet and Flame malware would pass the digital signature checks (they both were signed with valid certificates), but would fail the hash check against a known good database.

Redline ships with a sample set of hashes for XP systems. Mandiant maintains a separate whitelist available in their downloads section containing hashes for most modern Windows operating systems. This list can be added to, you can use the NIST National Software Reference Library hashes, or create your own database more specific to your environment. Hashes are updated within Redline by visiting the **`Options → Whitelist Management`** screen. Information on the last update and current number of hashes loaded is provided. There is also an option there to hide whitelisted items by default.

If you are looking for Redline to alert you on "known bad" hashes, this can be accomplished by creating an Indicator of Compromise for those hashes and including the IOC during initial analysis.


Trust Status
Name
Process Name
PID
Count
MD5
Digitally signed and verified
C:\WINDOWS\system32\termsrv.dll
svchost.exe
1236
1
Digitally signed and verified
C:\WINDOWS\system32\icaapi.dll
svchost.exe
1236
1
Digitally signed and verified
C:\WINDOWS\system32\rdpwsx.dll
svchost.exe
1236
1
Digitally signed and verified
C:\WINDOWS\system32\mstlsapi.dll
svchost.exe
1236
1
Digitally signed and verified
C:\WINDOWS\system32\winspool.drv
svchost.exe
1236
8
Digitally signed and verified
C:\WINDOWS\system32\adsldpc.dll
svchost.exe
1236
8
Digitally signed and verified
C:\WINDOWS\system32\apphelp.dll
svchost.exe
1236
9
Digitally signed and verified
C:\WINDOWS\system32\ntmarta.dll
svchost.exe
1236
11
Hide Whitelisted Items
58 Items



IOC Analysis

Indicator of Compromise (IOC) use allows automated analysis of systems, or in this case memory images. The idea is simple. Once malicious activity has been identified, an analyst creates an IOC defining a way to detect that activity in very specific terms. In terms of memory, this could be the presence of a specific process name, command line, DLL path, port opened, unlinked DLL, hooked function, etc. An indicator is created and tested, and then can be used to automatically identify the malicious activity in the future. In memory forensics, IOCs can greatly speed analysis because examiners do not need to waste time finding the same malicious artifacts over and over again. If the system you are currently looking at is infected with something seen before, your IOC will alert and identify exactly what artifacts should be reviewed. Building indicators (regardless of what format you choose) is a good best practice for incident response teams. As malware is discovered in your enterprise, it should be analyzed and signatures created. These signatures can then be used to identify similar malware found in the future. This process can save an enormous amount of time, preventing team members from "re-discovering" the same malware over and over again.

Redline has deep integration with the OpenIOC standard of representing malware indicators (it should because Mandiant created both). This means that you can include IOC files during your initial analysis and Redline will alert on any matching indicators. There are donated IOCs available for some of the most common malware and backdoors at the OpenIOC site and on sites like IOC Bucket.[1] You can also create your own using the free IOC Editor tool.[2] Keep in mind that the IOC Editor tool can represent far more triggers to alert on than are present in a standard memory image. Information that can be scanned in memory images are processes, hooks, drivers, handles, and process/driver strings. For IOCs dependent on file properties, registry keys, event logs, etc., you will need to conduct a live analysis using a Comprehensive Collector or a minimal IOC Search Collector created by Redline.

Takahiro Haruyama created a plugin named "openioc_scan" that brings the OpenIOC format to Volatility.[3] It does an excellent job of mapping the various IOC terms to its Volatility equivalents. It can use existing IOCs in

the version 1.1 format, or you can build your own. For the latter, we recommend the python-based editor PyIOCe because it has nice integration with the Volatility-specific naming conventions.[4], [5]

[1] http://www.openioc.org/

[2] https://www.iocbucket.com/

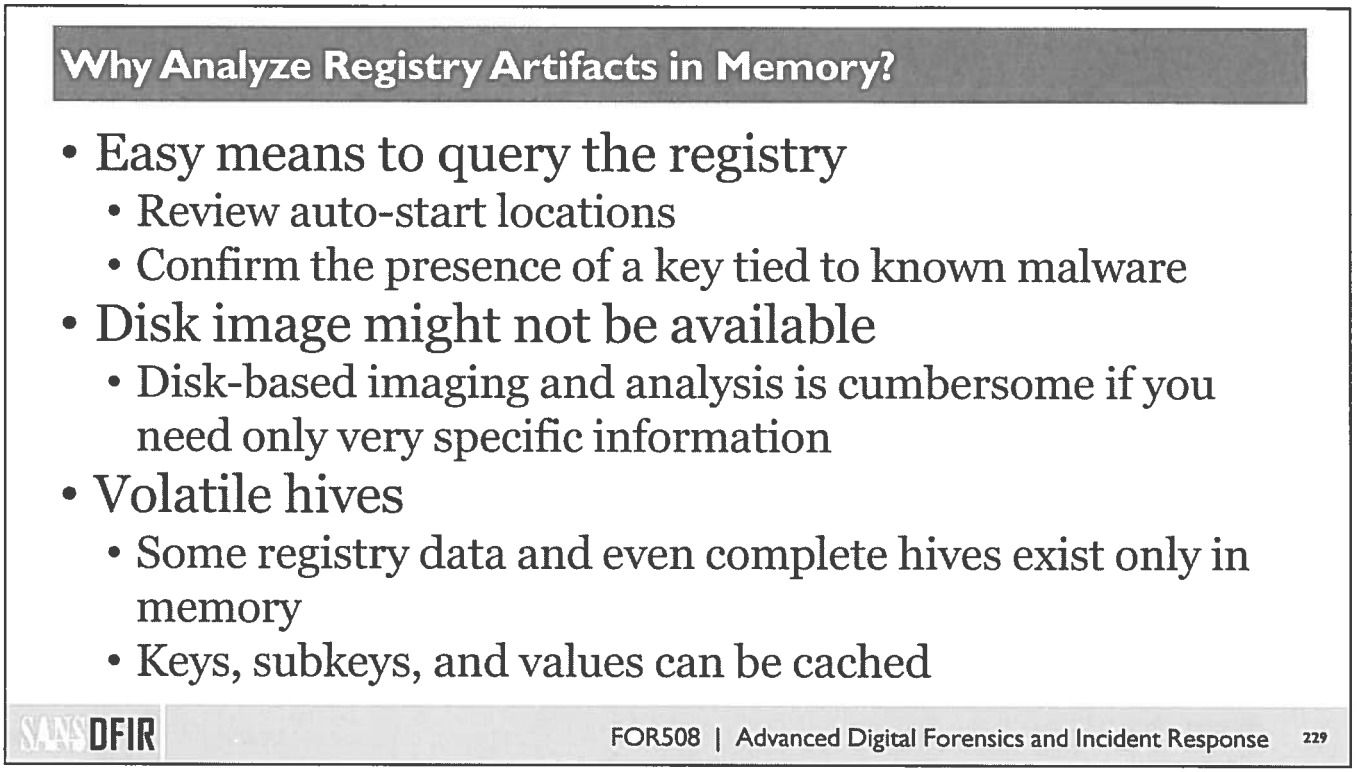
[3] http://takahiroharuyama.github.io/blog/2014/08/15/fast-malware-triage-using-openioc-scan-volatility-plugin/

[4] https://github.com/yahoo/PyIOCe

[5] http://takahiroharuyama.github.io/blog/2014/10/24/openioc-parameters-used-by-openioc-scan/

Registry Analysis

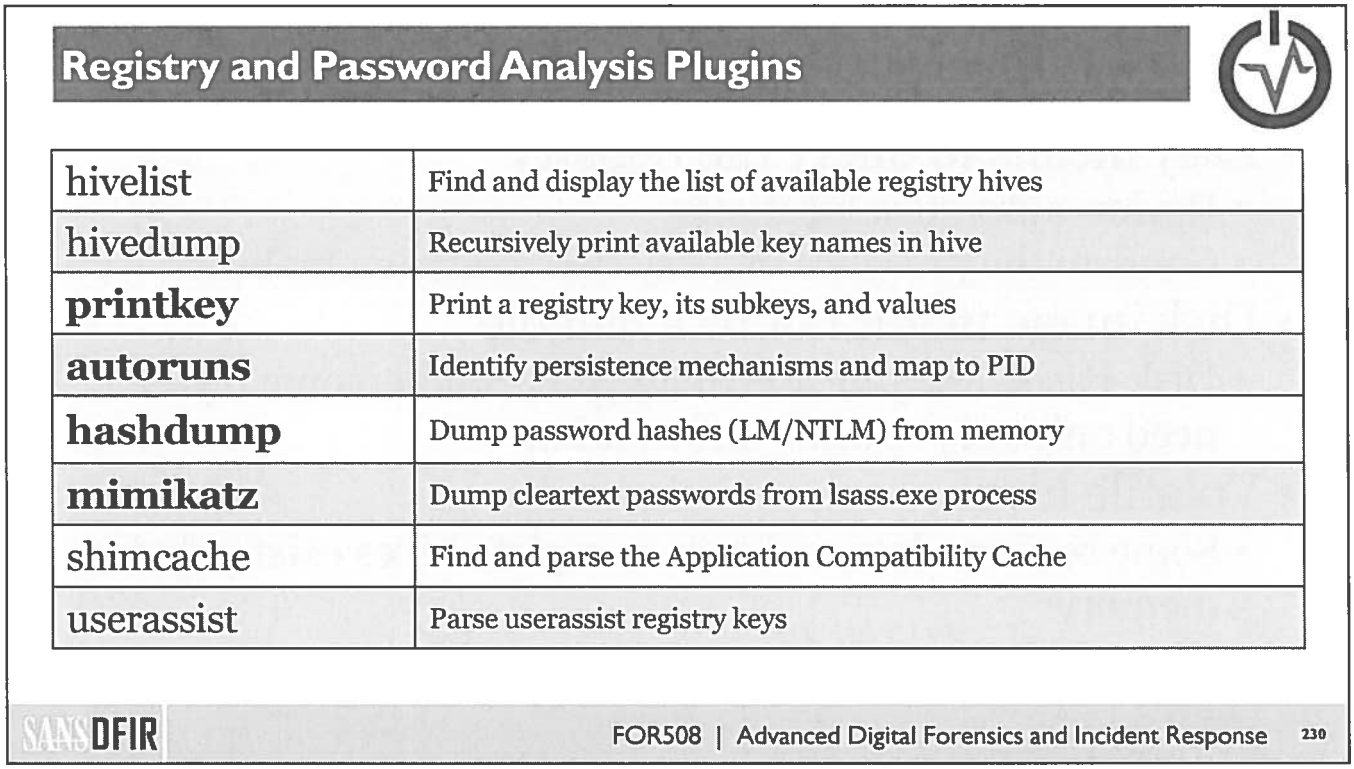
This page intentionally left blank.



Why Analyze Registry Artifacts in Memory?

Everything we have done thus far with memory analysis has revolved around processes and drivers. At this stage in our analysis, we might have identified a potentially malicious process or driver, or perhaps evidence of network connections or URLs of interest, leading us to the stage where we need a more in-depth analysis of the complete computer system. Before immediately diving into the disk image (if it exists), we first will transition to using our memory image to assist with answering questions for which we typically have relied on disk-based forensics. Ordinarily, our next step would be to start looking at the disk image, but before doing that, there is a wealth of Windows registry information we can get directly from the memory image.

In addition to expediency, there are a couple of reasons we might want to perform registry analysis using a memory image. First, registry hives are mapped to memory for the express purpose of speeding up reads and writes of regularly used keys. Data caching is in play, meaning that the copy in memory might be more up to date than the registry hive on disk. There are also memory only (also called "volatile") hives that might hold valuable data. A good example of a volatile hive is HKLM\HARDWARE. Volatile keys might never get written to disk. In fact, registry keys have been modified in memory by an attacker with the express purpose of frustrating disk-based forensics. The changes would not survive a reboot, but maybe they don't have to.



Registry and Password Analysis Plugins

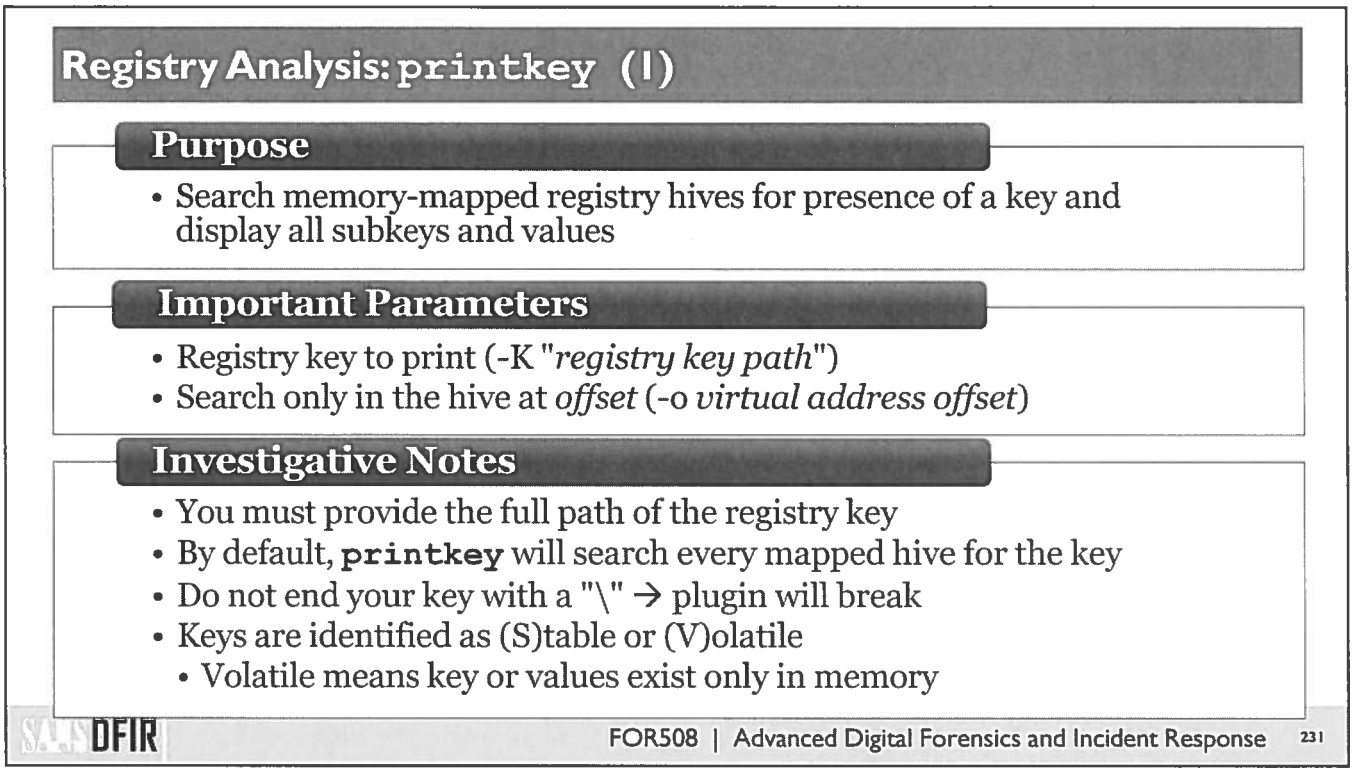
Research of in-memory registry forensics is one of the most exciting areas of focus for the Volatility project. The Windows registry is one of our most important forensic artifacts, and every recent version of Volatility has included wonderful new capabilities for doing registry analysis without the need for a disk image. The focus of this section is to introduce you to some of the registry capabilities of Volatility, but we do not have the time in this course to do an exhaustive overview (if you like memory forensics, consider a course like SANS FOR526 in the future for a week-long deep-dive!) Instead, we have selected the core Volatility registry plugins to demonstrate capabilities and will leave you with your newfound skills to investigate the rest. This is not an exhaustive list, and nor will it ever be due to all of the new plugins currently in development. But it gives us a good overview of what is currently available in the Volatility framework and what the future potential is. We will cover:

- **hivelist**: Find and display a list of available registry hives.
- **printkey**: Print a registry key, its subkeys, and values.
- **autoruns**: Identify a variety of different application-persistence mechanisms and attempt to map to running processes.
- **hashdump**: Dump NTLM and Lanman hashes.
- **mimikatz**: Dump cleartext passwords from lsass.exe.

The other plugins listed on the slide (not in bold) are here to whet your appetite towards some of the more unique registry capabilities of Volatility. As an example, a very specialized plugin for parsing userassist keys was created by Jamie Levy.[1] This is a great proof of concept plugin that shows both the power of registry analysis via a memory image, and the ease of output that can be delivered via the Volatility framework.

The shimcache plugin is one of the newest registry plugins and takes advantage of recent research into the Windows Application Compatibility Cache. This cache provides yet another artifact for application execution and has proven to be an excellent source for identifying malware or tools that were run on a system.

[1] http://gleeda.blogspot.com/2011/04/volatility-14-userassist-plugin.html



Registry Analysis: `printkey` (1)

Many times during a review, the analyst might need to review a few registry keys to do things like confirm a compromise of the system, identify persistence mechanisms, or find out system description information like timezone or hostname. The **`printkey`** plugin gives an easy way to peer within the memory-mapped registry hives available and search for a specific registry key. If the key is found, all of the data you would typically expect is provided:

- Registry hive where found
- Key name
- Key last written time
- Stable (on disk) or Volatile (only in memory)
- Subkeys
- Values
- Value Type
- Value Name
- Value Data

The "-K" option must be provided to tell the plugin what key to search for. Make sure not to leave a trailing "\" on the key or the plugin will stall! Unfortunately, the value provided must be the full path of the key; there is no regular expression capability (yet). By default, the **`printkey`** plugin will run the **`hivelist`** plugin and search every registry hive returned. Thus, you might get multiple hits from multiple hives. If you would like to specify a hive, say a specific user's NTUSER.Dat file, the "-o" parameter can be used with the hive's *virtual* address (found using **`hivelist`**).

An interesting component of this plugin is that it records whether each key, subkey, and value is either Stable or Volatile. A Stable key or value means that the information exists in the registry hives on disk. A Volatile key or

value means the information only *exists* in memory. If you have been doing forensics for a while, this should be an eye-opening moment! How much volatile information have we been missing in our examinations all these years? This is particularly interesting with regard to malware, because a malicious process could change values within the memory-mapped hive that exist only in memory and never get written to disk. A researcher named Brendan Doyle-Gavitt (who also wrote many of the Volatility registry plugins) did a proof of concept where he was able to change the Administrator's password hash in the memory-mapped registry hive and subsequently log on using the new hash. When the system was rebooted, the new hash was purged.

Registry Analysis: printkey (2)

Auto-start value for Sobig worm (winppr32.exe) found in "Owner" hive

```
# vol.py -f /memory/sobig.img printkey
    -K "Software\Microsoft\Windows\CurrentVersion\Run"
Legend: (S) = Stable   (V) = Volatile
----------------------------
Registry: \Device\HarddiskVolume1\Documents and Settings\Owner\NTUSER.DAT
Key name: Run (S)
Last updated: 2009-07-27 23:27:44

Subkeys:

Values:
REG_SZ        TrayX            : (S) C:\WINDOWS\winppr32.exe /sinc
```


Registry Analysis: `printkey`

In this example, we revisit our image infected with the Sobig Worm. One of the persistence mechanisms used by the Sobig Worm is to create a "run" key so that its malicious executable is run every time the infected user logs on. We previously found a suspicious process in this image named "winppr32.exe." To verify that this process is in fact malicious, we searched the memory-mapped registry hives in the memory image using the Volatility plugin **`printkey`**, giving it the key "Software\Microsoft\Windows\CurrentVersion\Run" to search for. As the slide shows, it found an interesting value in the Owner account's NTUSER.DAT hive. The value listed in the key was "C:\Windows\winppr32.exe /sinc." This key and value is listed as being (S)table, meaning the information should exist within the same hive written to disk. The full command line is listed in the following. Notice that **`printkey`** found the key in multiple hives, printing out the results for each in succession.

root@SIFT-Workstation:/# vol.py -f /memory/sobig.img printkey -K "Software\Microsoft\Windows\CurrentVersion\Run"

Legend: (S) = Stable (V) = Volatile

Registry: \Device\HarddiskVolume1\Documents and Settings\NetworkService\NTUSER.DAT

Key name: Run (S)

Last updated: 2009-07-20 23:40:00

Subkeys:

Values:

Registry: \Device\HarddiskVolume1\Documents and Settings\LocalService\NTUSER.DAT
Key name: Run (S)
Last updated: 2009-07-20 23:40:05

Subkeys:

Values:

Registry: \Device\HarddiskVolume1\Documents and Settings\Owner\NTUSER.DAT
Key name: Run (S)
Last updated: 2009-07-27 23:27:44

Subkeys:

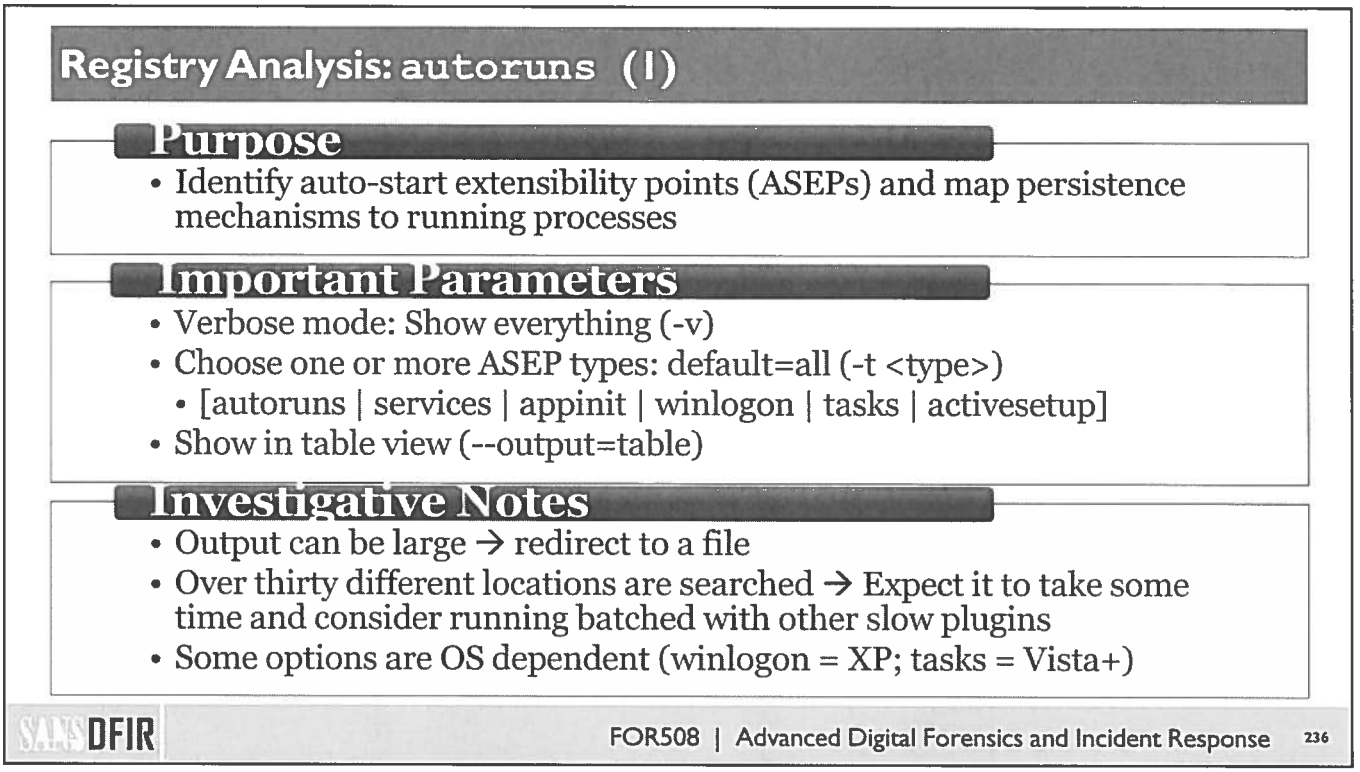
Values:
REG_SZ TrayX : (S) C:\WINDOWS\winppr32.exe /sinc

Registry: \Device\HarddiskVolume1\WINDOWS\system32\config\default
Key name: Run (S)
Last updated: 2009-07-20 19:26:47

Subkeys:

Values:

```
# vol.py -f /memory/sobig.img printkey
    -K "Software\Microsoft\Windows\CurrentVersion\Run"
Legend: (S) = Stable   (V) = Volatile
----------------------------
Registry: \Device\HarddiskVolume1\Documents and Settings\Owner\NTUSER.DAT
Key name: Run (S)
Last updated: 2009-07-27 23:27:44

Subkeys:

Values:
REG_SZ        TrayX            : (S) C:\WINDOWS\winppr32.exe /sinc
```



Registry Analysis: `autoruns` (1)

One of the many challenges of the Windows operating system is the sheer number of auto-start extensibility points (ASEPs) that it harbors. ASEPs can provide applications persistence by executing them at boot or whenever a user logs on. Because malware usually wants to survive a reboot, persistence mechanism locations can be a very fruitful place to hunt. Thomas Chopitea wrote a Volatility plugin to search the most common ASEP locations, extract the data, and output it a human-friendly way.

The plugin can run slowly because a majority of its ASEP locations reside in the Windows registry, and multiple registry hives must be located and searched in addition to other post-processing. However, it is worth the wait! Traditionally, we would need to gather this information from a live system or worst case from a disk image. Now, Mr. Chopitea has given us the ability to do it solely from a memory image. Once the plugin has collected the ASEP information, it then goes back through the memory image to try to match ASEPs with running processes. It does this by matching the filename and path found in the ASEP to a process command line or a loaded DLL in the case of a Windows Service.

When performing a full review of ASEP data, it is recommended to add the "verbose (-v)" option because Services started from the C:\Windows\system32 folder will be ignored by default.

The **`autoruns`** plugin searches for and reports data from the following information sources:[1]

Software hive

Microsoft\Windows\CurrentVersion\Run,

Microsoft\Windows\CurrentVersion\RunOnce,

Microsoft\Windows\CurrentVersion\RunServices,

Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,
Wow6432Node\Microsoft\Windows\CurrentVersion\Run,
Wow6432Node\Microsoft\Windows\CurrentVersion\RunOnce,
Wow6432Node\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,
Microsoft\Windows NT\CurrentVersion\Terminal Server\Install\Software\Microsoft\Windows\CurrentVersion\Run,
Microsoft\Windows NT\CurrentVersion\Terminal Server\Install\Software\Microsoft\Windows\CurrentVersion\RunOnce

NTUSER.DAT
Software\Microsoft\Windows\CurrentVersion\Run,
Software\Microsoft\Windows\CurrentVersion\RunOnce,
Software\Microsoft\Windows\CurrentVersion\RunServices,
Software\Microsoft\Windows\CurrentVersion\RunServicesOnce,
Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,
Software\Microsoft\Windows NT\CurrentVersion\Terminal Server\Install\Software\Microsoft\Windows\CurrentVersion\Run,
Software\Microsoft\Windows NT\CurrentVersion\Terminal Server\Install\Software\Microsoft\Windows\CurrentVersion\RunOnce,
Software\Microsoft\Windows NT\CurrentVersion\Run,
Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,
Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run

Winlogon and AppInit
Microsoft\Windows NT\CurrentVersion\Winlogon (value AppInit_DLLs)
Microsoft\Windows NT\CurrentVersion\Winlogon\Notify
Microsoft\Windows NT\CurrentVersion\Winlogon\Notify
Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit
Microsoft\Windows NT\CurrentVersion\Winlogon\VmApplet
Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
Microsoft\Windows NT\CurrentVersion\Winlogon\TaskMan
Microsoft\Windows NT\CurrentVersion\Winlogon\System

Services
CurrentControlSet\Services

Scheduled Tasks
C:\Windows\System32\Tasks\ (Windows Vista and onwards only)

Active Setup
Microsoft\Active Setup\Installed Components

[1] https://github.com/tomchop/volatility-autoruns

Registry Analysis: autoruns (2)

```
# vol.py -f xp-tdungan-memory-raw.001 autoruns -t autoruns
Autoruns ########################################

Hive: \Device\HarddiskVolume1\WINDOWS\system32\config\software
Microsoft\Windows\CurrentVersion\Run (Last modified: 2012-04-03 00:39:24 UTC+0000)

VMware User Process          : "C:\Program Files\VMware\VMware Tools\VMwareUser.exe" (PIDs: 2992)
Adobe Reader Speed Launcher : "C:\Program Files\Adobe\Reader 9.0\Reader\Reader_sl.exe" (PIDs: -)
svchost                      : c:\windows\system32\dllhost\svchost.exe (PIDs: 3296)
VMware Tools                 : "C:\Program Files\VMware\VMware Tools\VMwareTray.exe" (PIDs: 2924)
SunJavaUpdateSched           : "C:\Program Files\Common Files\Java\Java Update\jusched.exe" (PIDs: 3268)
McAfeeUpdaterUI              : "C:\Program Files\McAfee\Common Framework\udaterui.exe" /StartedFromRunKey
McAfee Host Intrusion Prevention Tray : "C:\Program Files\McAfee\Host Intrusion Prevention\FireTray.exe"
APSDaemon                    : "C:\Program Files\Common Files\Apple\Apple Application Support\APSDaemon.e
ShStatEXE                    : "C:\Program Files\McAfee\VirusScan Enterprise\SHSTAT.EXE" /STANDALONE (PID
QuickTime Task               : "C:\Program Files\QuickTime\QTTask.exe" -atboottime (PIDs: -)
```

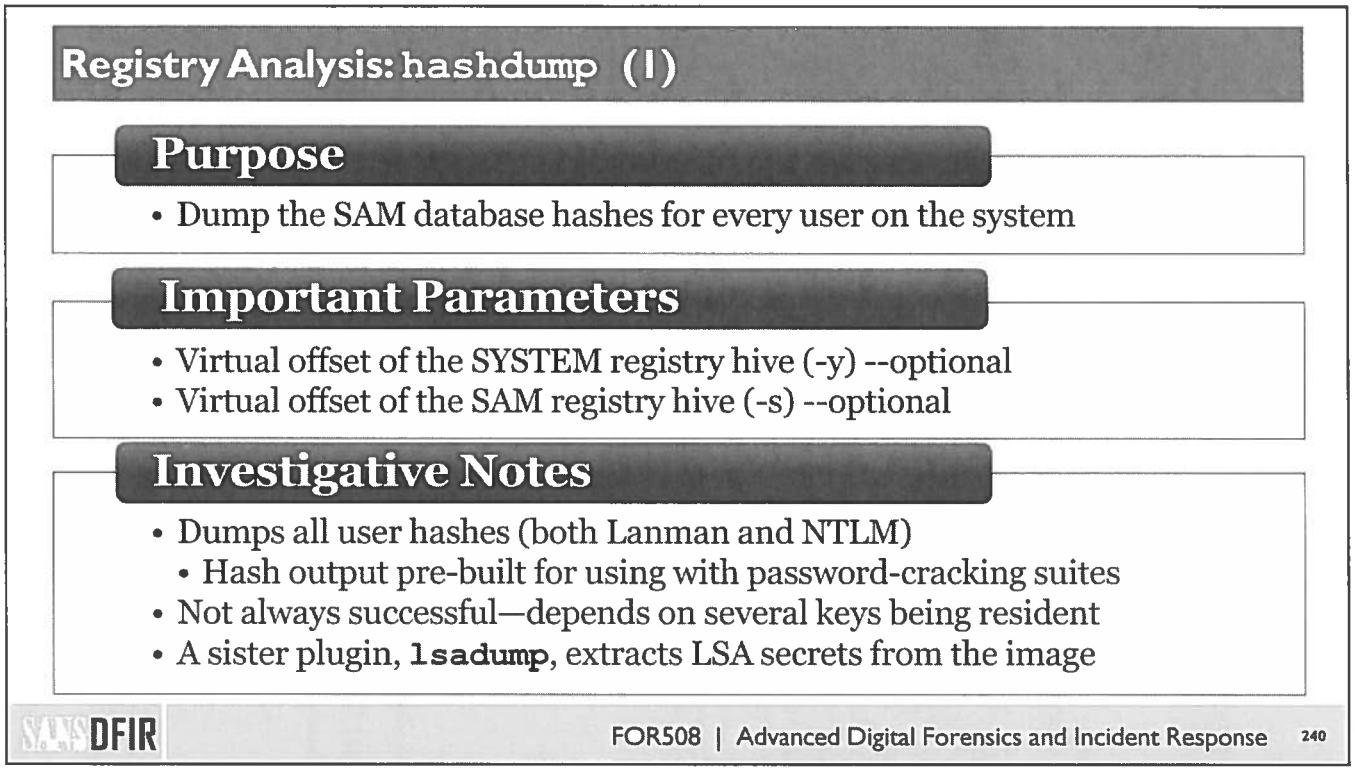
```
svchost     : c:\windows\system32\dllhost\svchost.exe (PIDs: 3296)
```


Registry Analysis: autoruns (2)

In this example, we ran the autoruns plugin requesting data from only the common "run" keys in the registry (these keys are so common for malware persistence that the entire plugin was named for this option). In the Tdungan image, the svchost entry in the SOFTWARE\Microsoft\Windows\CurrentVersion\Run key is very unusual. Further analysis would show that it is pointing to an executable in a strange folder (c:\windows\system32\dllhost). Note that the autoruns plugin went a step further and tried to match each entry to a running process (also note that it was not successful for every entry). The suspicious svchost entry maps to process ID 3296. If this would have been the first plugin you ran on this image, you would be well on your way to unraveling what happened on this system!

```
# vol.py -f xp-tdungan-memory-raw.001 autoruns -t autoruns

Autoruns ######################################

Hive: \Device\HarddiskVolume1\WINDOWS\system32\config\software
Microsoft\Windows\CurrentVersion\Run (Last modified: 2012-04-03 00:39:24 UTC+0000)

VMware User Process          : "C:\Program Files\VMware\VMware Tools\VMwareUser.exe" (PIDs: 2992)
Adobe Reader Speed Launcher : "C:\Program Files\Adobe\Reader 9.0\Reader\Reader_sl.exe" (PIDs: -)
svchost                      : c:\windows\system32\dllhost\svchost.exe (PIDs: 3296)
VMware Tools                 : "C:\Program Files\VMware\VMware Tools\VMwareTray.exe" (PIDs: 2924)
SunJavaUpdateSched           : "C:\Program Files\Common Files\Java\Java Update\jusched.exe" (PIDs: 3268)
McAfeeUpdaterUI              : "C:\Program Files\McAfee\Common Framework\udaterui.exe" /StartedFromRunKey
McAfee Host Intrusion Prevention Tray : "C:\Program Files\McAfee\Host Intrusion Prevention\FireTray.exe"
APSDaemon                    : "C:\Program Files\Common Files\Apple\Apple Application Support\APSDaemon.e
ShStatEXE                    : "C:\Program Files\McAfee\VirusScan Enterprise\SHSTAT.EXE" /STANDALONE (PID
QuickTime Task               : "C:\Program Files\QuickTime\QTTask.exe" -atboottime (PIDs: -)
```

```
svchost      : c:\windows\system32\dllhost\svchost.exe (PIDs: 3296)
```



Registry Analysis: `hashdump` (1)

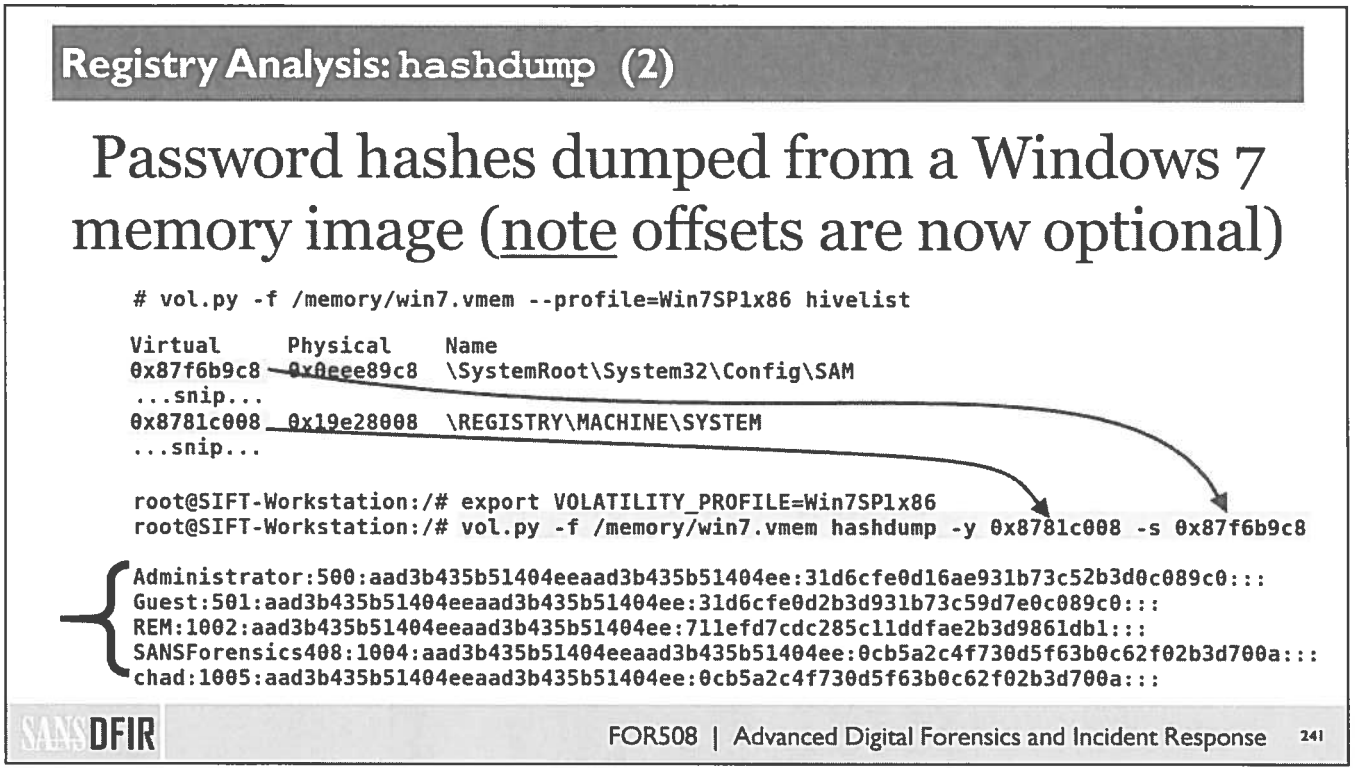
Although not every registry key is present in the memory-mapped registry hives, it makes sense that the account password hashes (and most of the SAM hive for that matter) would often be available; these keys and values are constantly being used by the system for authentication purposes. The **`hashdump`** plugin is a specialized plugin that attempts to pull the SAM database hashes out of the memory image. When successful, the output is nicely formatted to be plugged directly into your favorite password-cracking tool like John the Ripper.

It turns out that pulling the password hashes out of the registry is a little more difficult due to the built-in Windows protection called "SysKey." SysKey protection was implemented by Microsoft in an attempt to thwart offline attacks on the SAM hash database. The SAM database password hashes are encrypted with a different key (called the "bootkey"). This key is itself stored in the registry (though in the SYSTEM hive not the SAM), broken into four pieces, and stored in hidden areas of the hive. Of course, once you know where to look, it is trivial to put the key back together again. Which is exactly what the **`hashdump`** plugin does. Newer versions of the plugin will automatically find the virtual offsets of the SYSTEM hive and SAM hive, but you can optionally specify them. The requirement to find the SYSTEM and SAM hives explains why **`hashdump`** might not grab hashes from every memory image—if any one of these pieces happens to be paged out, no dice.

Interestingly, the same bootkey is also used to "protect" the LSA secrets, containing things like network passwords, service passwords, and cached domain credentials. A sister plugin, named **`lsadump`**, was developed to also extract this information. This plugin works on XP/2003 systems.

The fact that this is possible should give you some insight as to why attackers have such an easy time getting our password hashes (and cached credentials). Malware running with Administrator rights easily get access to the Windows kernel and hence complete access to memory. Anything we can access in memory is also available to an attacker once he or she has a foothold on the system.

As of Volatility 2.4, this runs on memory images from both x86 and x64 systems.



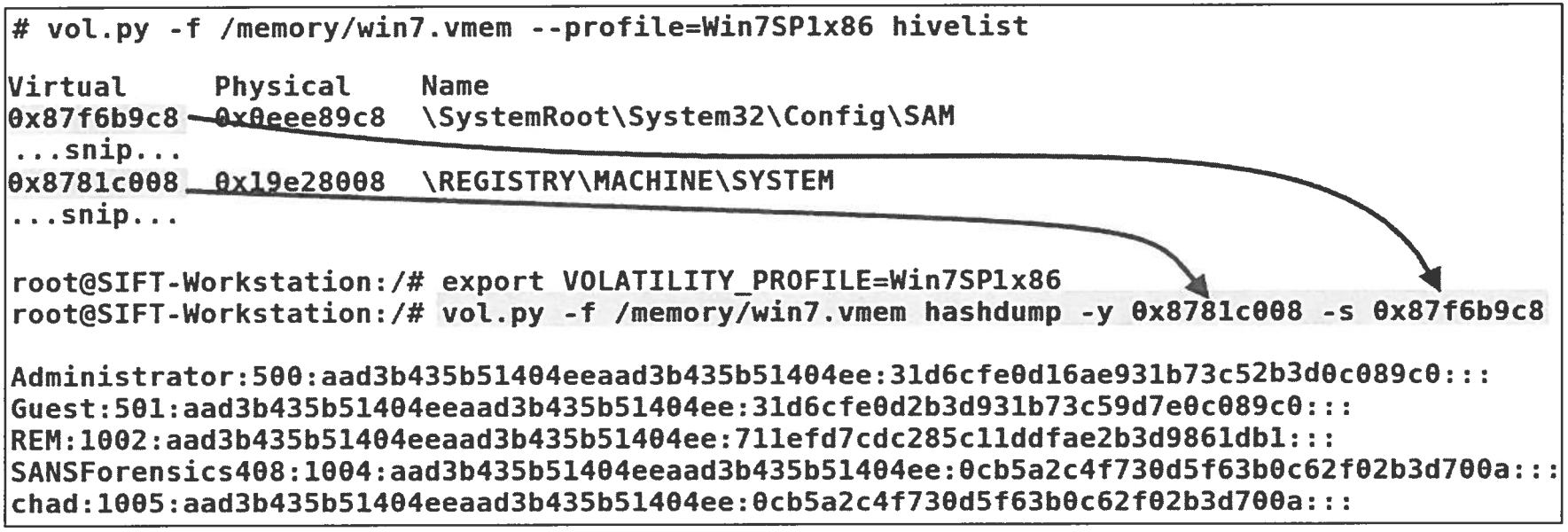
Registry Analysis: `hashdump` (2)

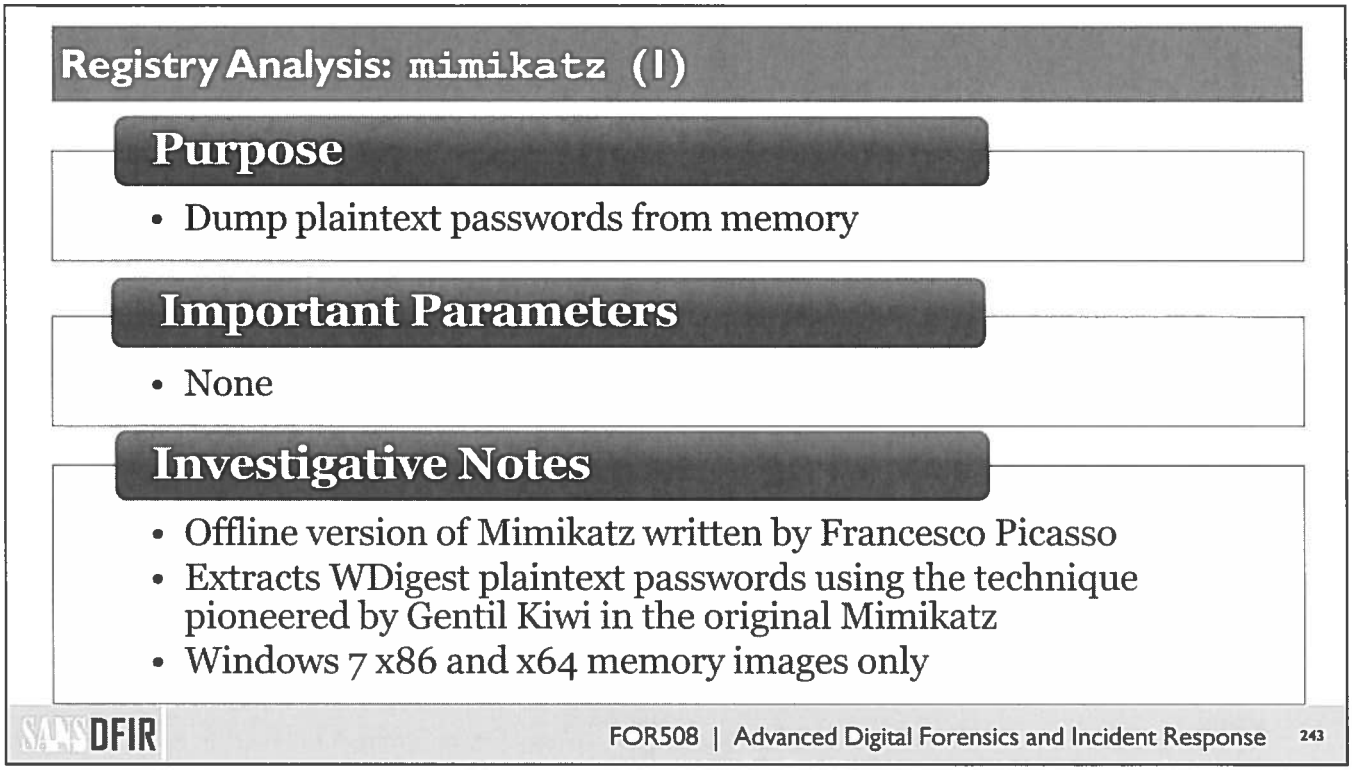
For this example, we decided to try our luck with a Windows 7 memory image. And not just any image, but a VMware memory dump from the old SANS 408 Windows 7 SIFT workstation.

First, we ran the **`hivelist`** plugin to identify the virtual addresses of the SAM and SYSTEM. While the latest version of hashdump doesn't need this information (it will find them itself), it is always nice to know how to do things the hard way.

With the Virtual Offsets for the SAM and SYSTEM hives in hand, we simply feed them to the **`hashdump`** plugin and it goes off to find the hashes and decrypt them with the SysKey bootkey. The five user accounts seen here are a result of this process. Notice that the account names, relative identifiers (RIDS), and two hashes for each account are displayed. The first hash (between the colons) is the Lanman hash for the account, and the second hash is the NTLM v2 hash. As you might know, the Lanman hash is incredibly easy to crack if it exists. Because this is a Windows 7 system, Lanman is turned off by default and hence those hashes equate to "empty." However, the NTLM v2 hashes are viable. Your next step might be to run them through a password cracker or bounce them up against rainbow tables. There are several online rainbow table lookups available. A good one for NTLM hashes is at the Md5Decrypter site.[1] If you are looking for something to do, try to crack the hash for the REM account. This was the pre-built account for the FOR610 Reverse Engineering Malware account.

[1] http://www.md5decrypter.co.uk/ntlm-decrypt.aspx


vol.py -f /memory/win7.vmem --profile=Win7SP1x86 hivelist
Virtual Physical Name
0x87f6b9c8 0x0eee89c8 \SystemRoot\System32\Config\SAM
...snip...
0x8781c008 0x19e28008 \REGISTRY\MACHINE\SYSTEM
...snip...
root@SIFT-Workstation:/# export VOLATILITY_PROFILE=Win7SP1x86
root@SIFT-Workstation:/# vol.py -f /memory/win7.vmem hashdump -y 0x8781c008 -s 0x87f6b9c8
Administrator:500:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c52b3d0c089c0:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d2b3d931b73c59d7e0c089c0:::
REM:1002:aad3b435b51404eeaad3b435b51404ee:711efd7cdc285c11ddfae2b3d9861db1:::
SANSForensics408:1004:aad3b435b51404eeaad3b435b51404ee:0cb5a2c4f730d5f63b0c62f02b3d700a:::
chad:1005:aad3b435b51404eeaad3b435b51404ee:0cb5a2c4f730d5f63b0c62f02b3d700a:::



Registry Analysis: `mimikatz` (1)

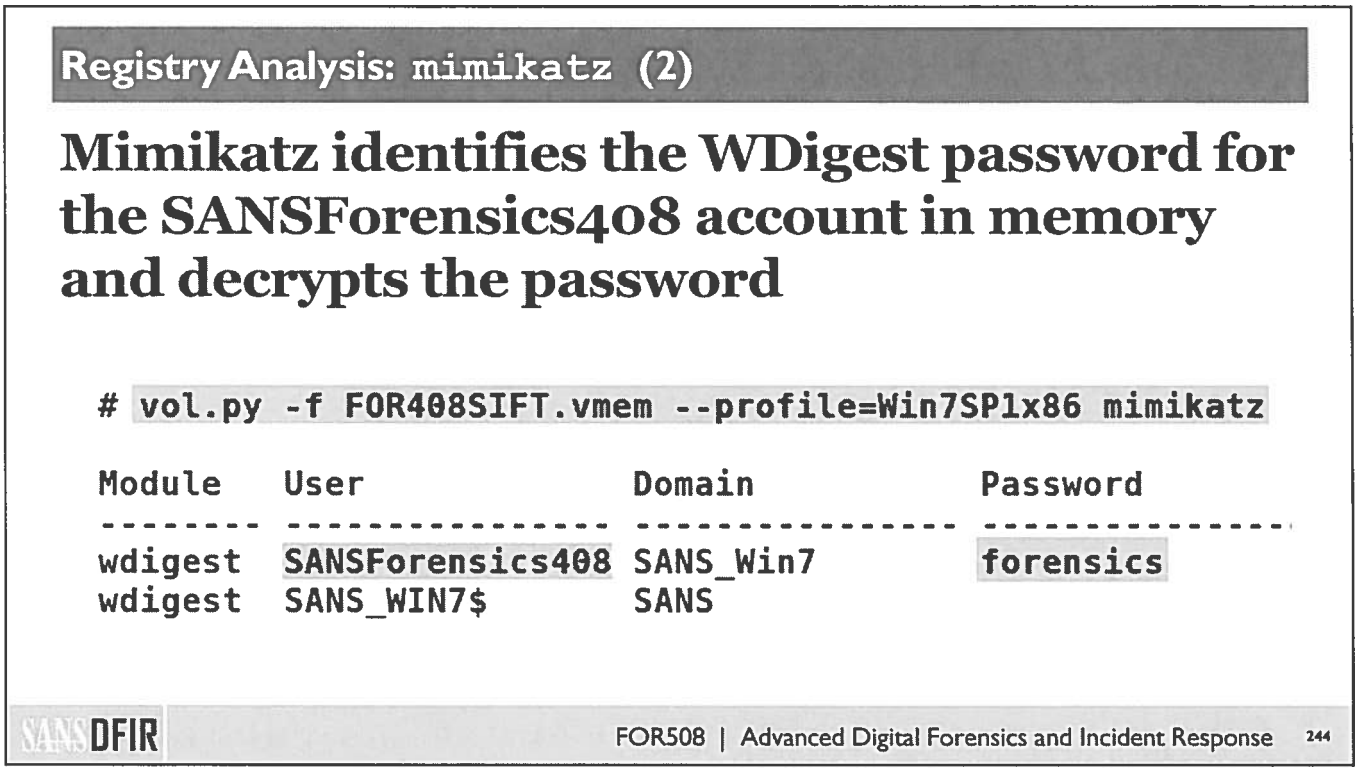
Mimikatz is one of the most devastating hack tools to emerge in recent memory. Although the world is still trying to mitigate pass-the-hash, all of a sudden attackers can now easily retrieve plaintext passwords! The technique has been ported to many tools including Metasploit and PowerShell and lends itself well to a memory forensics plugin because it is really a memory-based attack. Benjamin Delpy (also known as Gentil Kiwi) found user passwords stored using reversible encryption during research into single sign-on in RDP.[1] Along the way, he discovered an even better cache of password data called "WDigest." WDigest is one of the ways that a Web server can negotiate credentials with a Web browser. The protocol hashes the password before sending, but it requires a user's plaintext password to derive the key. An encrypted version of that password is stored in memory and can be easily decrypted.

Although the original version of Mimikatz supports the collection of multiple different types of credentials, this plugin currently focuses on the extraction of WDigest credentials. It was written by Francesco Picasso and has no options—the only requirement is a Windows 7 memory image.[2] Retrieving a user's plaintext password has many uses, including a great way to educate individuals about the need to physically secure their systems as well as taking advantage of password reuse when trying to crack encryption. Imagine a scenario where someone forgot to check for encryption and powered down a Bitlocker-enabled system. If he happened to get a memory image first, and if the user uses some derivation of his Windows password for his Bitlocker key, there is still a very good chance of getting access to the encrypted volume.

Note that this plugin is designed to be run only on memory images from Windows 7 x86 and x64 systems. It is unlikely to be ported to Windows 8 systems because Win8.1 implemented significant mitigations including turning off WDigest by default.

[1] http://blog.gentilkiwi.com/downloads/mimikatz-phdays.pdf

[2] http://blog.digital-forensics.it/2014/03/et-voila-le-mimikatz-offline.html



Registry Analysis: `mimikatz` (2)

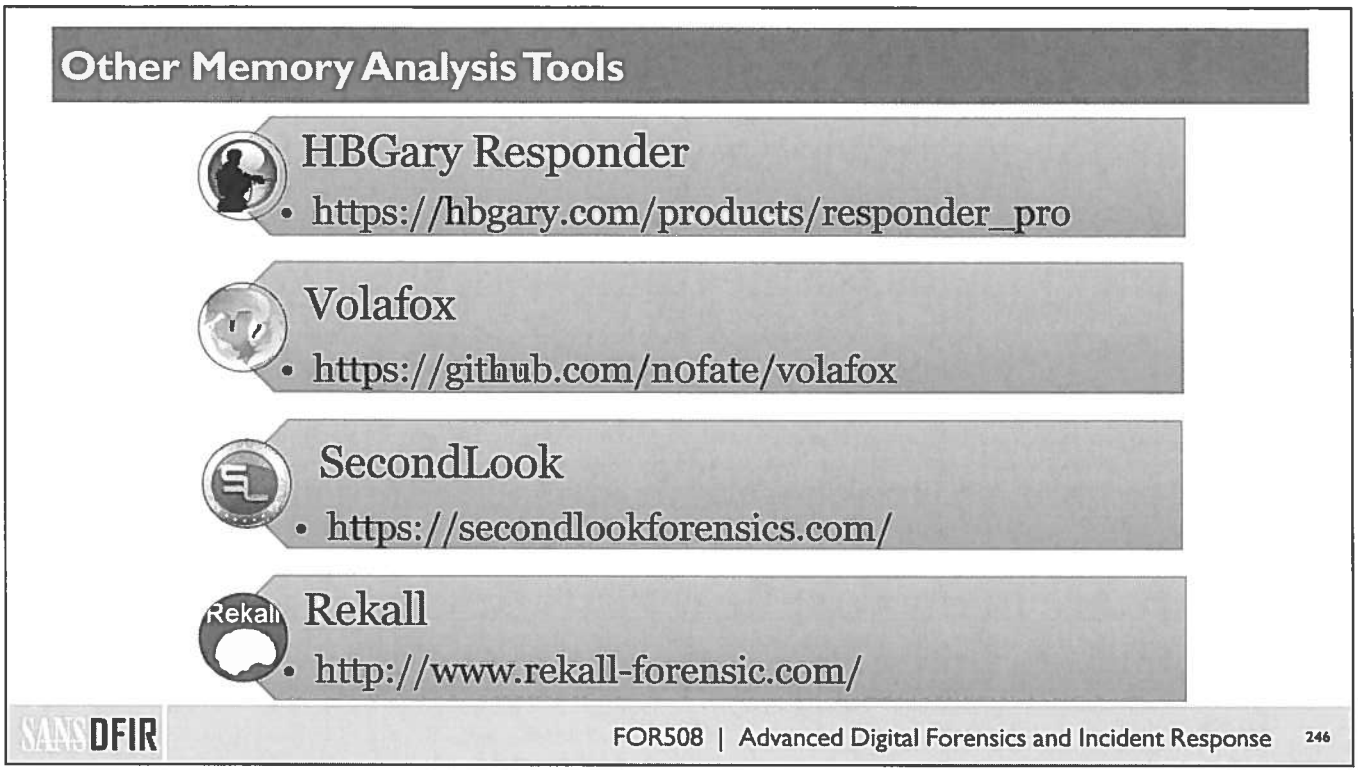
Why dump password hashes and go through all that effort of cracking them when you can just get the plaintext password? Mimikatz is an incredibly powerful tool and should scare the heck out of anyone who witnesses it in action. Here, we use a VMware .vmem file for the old Windows 7 version of the FOR408 SIFT workstation. With a simple command, we can easily extract out the last user's password. Keep in mind that this could be done over the network with F-Response or via other types of on-disk memory captures like hibernation files and crash dumps.

Registry Analysis: Review

- Volatility can provide much more than just process and driver information
- Registry plugins for the framework show great promise
  - Expect to see additional plugins leverage registry artifacts resident in memory
  - **`autoruns`**, **`shimcache`**, and **`userassist`** are just the beginning!
- There are many volatile registry keys that exist only in memory and never get written to disk

Registry Analysis: Review

Although not as fully formed as the process, driver, and malware plugins, the registry plugins for Volatility show great promise. They open up an entire new world to the analyst—allowing some "disk" forensics to be done without an image of the disk! One of the most important takeaways for this section is that information might exist in the memory-mapped registry hives that never gets written to disk. The **`printkey`** plugin is a great place to see this in action. It marks each key, subkey, and value as Stable (on disk) or Volatile (disappears upon shutdown). To our knowledge, no comprehensive review has been done on volatile keys to find which hold important information to an examiner.



Other Memory Analysis Tools

We have covered two of the best and most full-featured Windows memory analysis tools, but the field is growing. HBGary has a popular commercial product named "Responder Pro." It also released a free "community" version. Its limitations are that it will analyze only memory images up to 6GB in size and does not include its well-regarded Digital DNA, meaning its built-in heuristics for malware detection are limited.

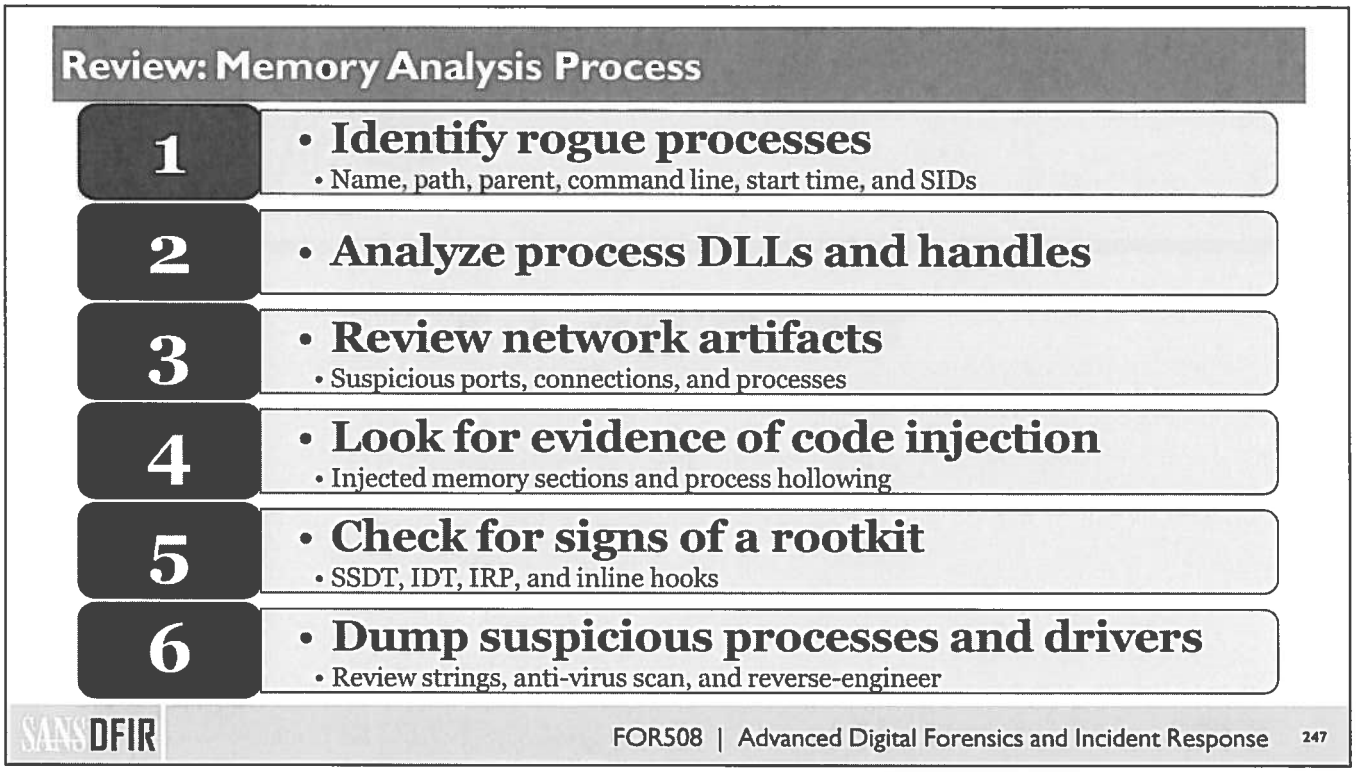
Windows memory analysis tools have long been ahead of those for other operating systems, but that gap has significantly narrowed. Volafox is a project for Mac OS X memory analysis. SecondLook is a commercial tool for Linux memory dumps. Volatility also supports both Mac OS X and Linux memory dumps.

Finally, the Rekall project is a fork of the Volatility codebase by previous members of the Volatility team. In December 2011, a new branch within the Volatility project was created to explore how to make the codebase more modular, improve performance, and increase usability. The modularity allowed Volatility to be used in GRR, making memory analysis a core part of a strategy to enable remote live forensics. As a result, both GRR and Volatility would be able to use each others' strengths.

Over time, this branch has become known as the "scudette" branch or the "Technology Preview" branch. It was always a goal to try to get these changes into the main Volatility codebase. But, after two years of ongoing development, the "Technology Preview" was never accepted into the Volatility trunk version.

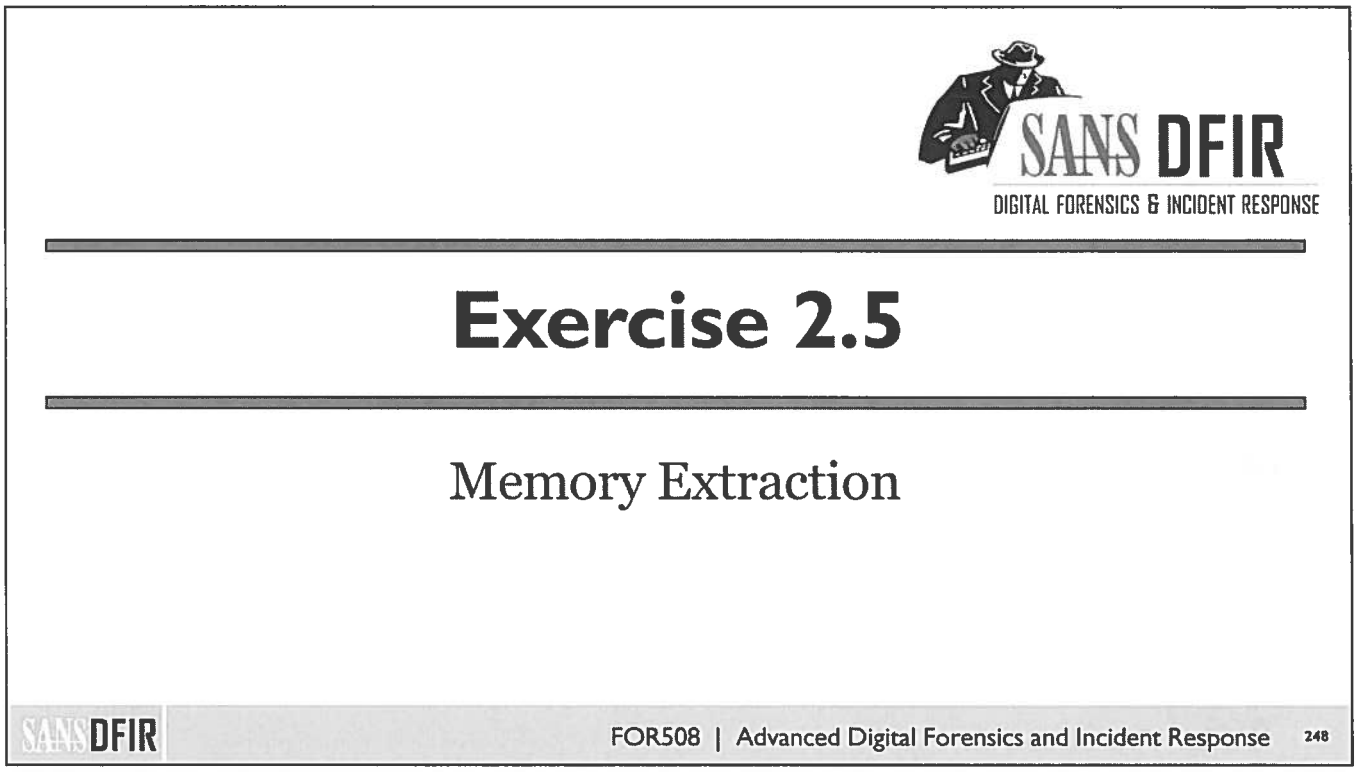
Because it seemed unlikely that these changes would be incorporated in the future, it made sense to develop the Technology Preview branch as a separate project. On December 13, 2013, the former branch was forked to create a new stand-alone project named "Rekall." This new project incorporates changes made to streamline the codebase so that Rekall can be used as a library. Methods for memory acquisition and other outside contributions have also been included that were not in the Volatility codebase.

Rekall strives to advance the state of the art in memory analysis, implementing the best algorithms currently available and a complete memory acquisition and analysis solution for at least Windows, OSX, and Linux.

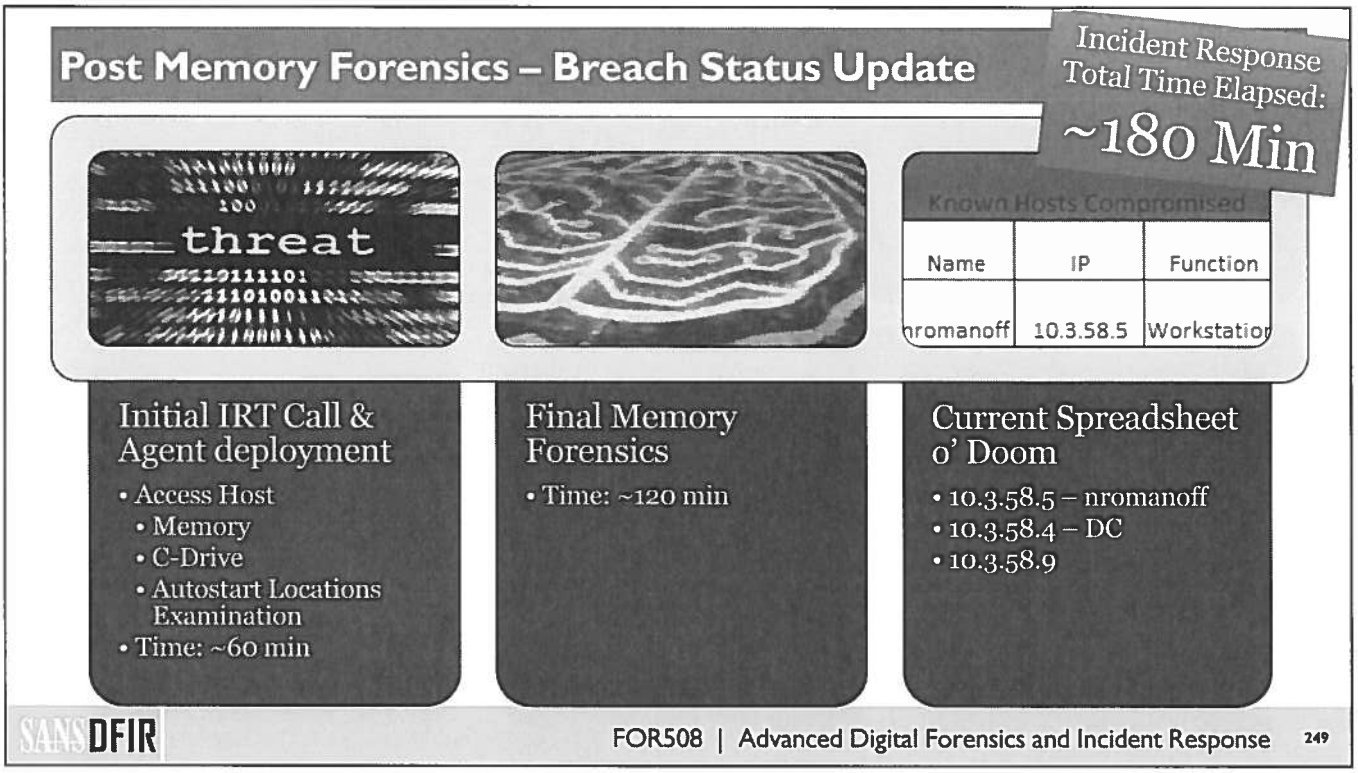


Review: Memory Analysis Process

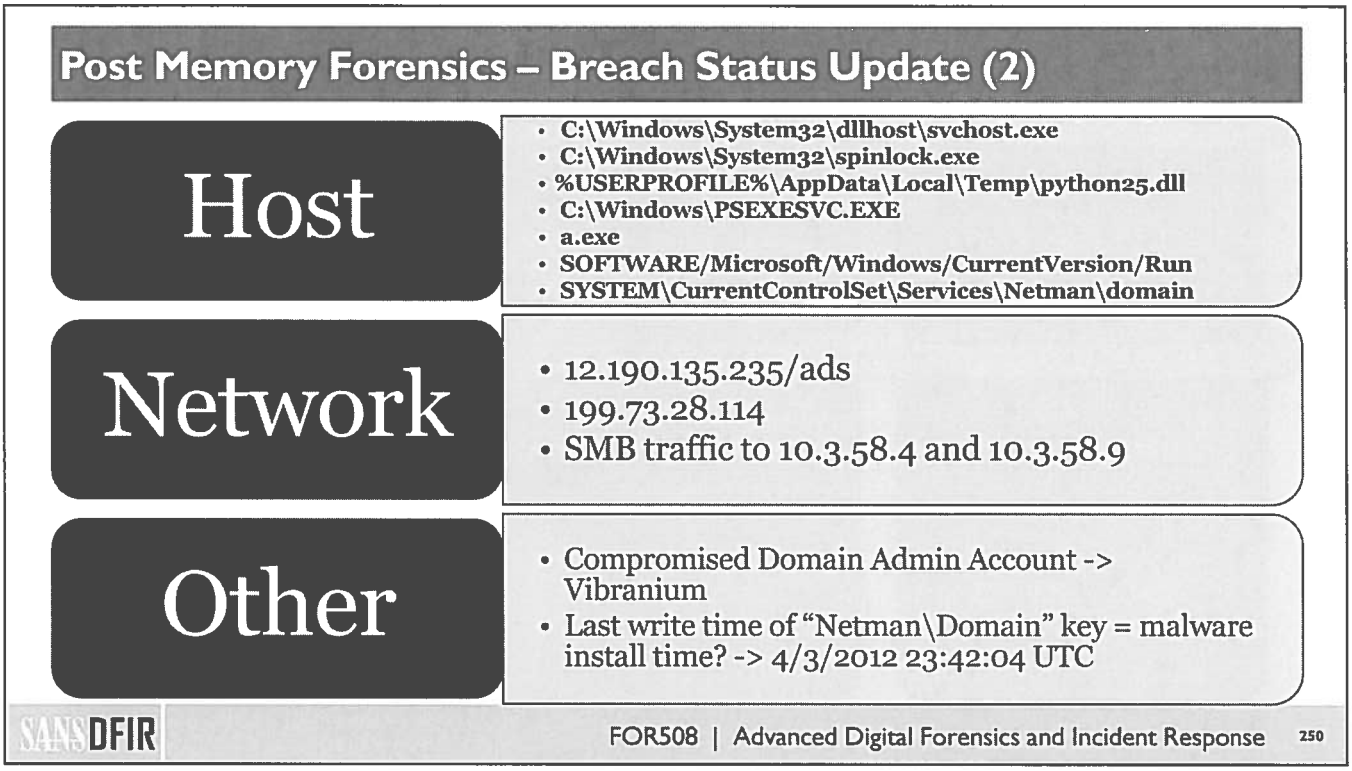
We have now covered an enormous number of tools for performing memory analysis using the Volatility framework. It is time for an exercise!



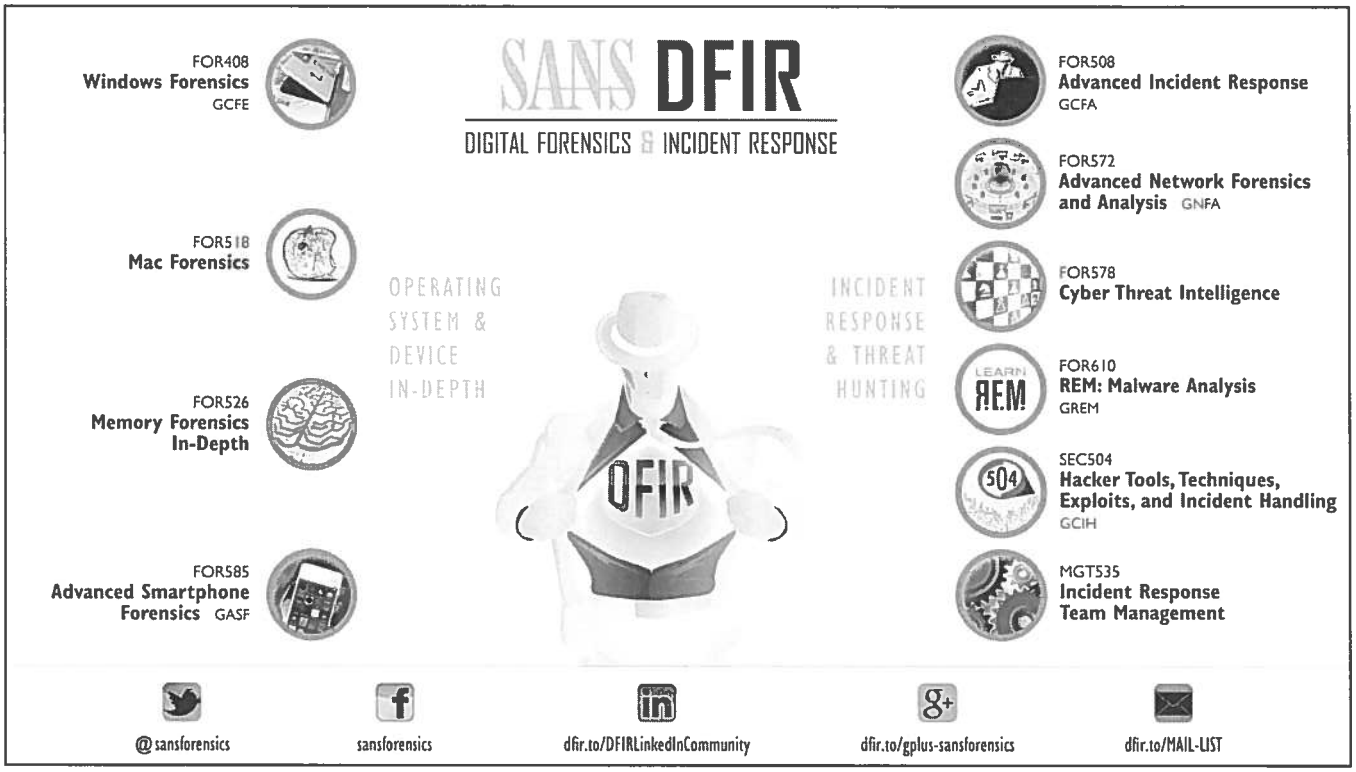
This page intentionally left blank.



This page intentionally left blank.



This page intentionally left blank.



This page intentionally left blank.

SANS DFIR

DIGITAL FORENSICS & INCIDENT RESPONSE

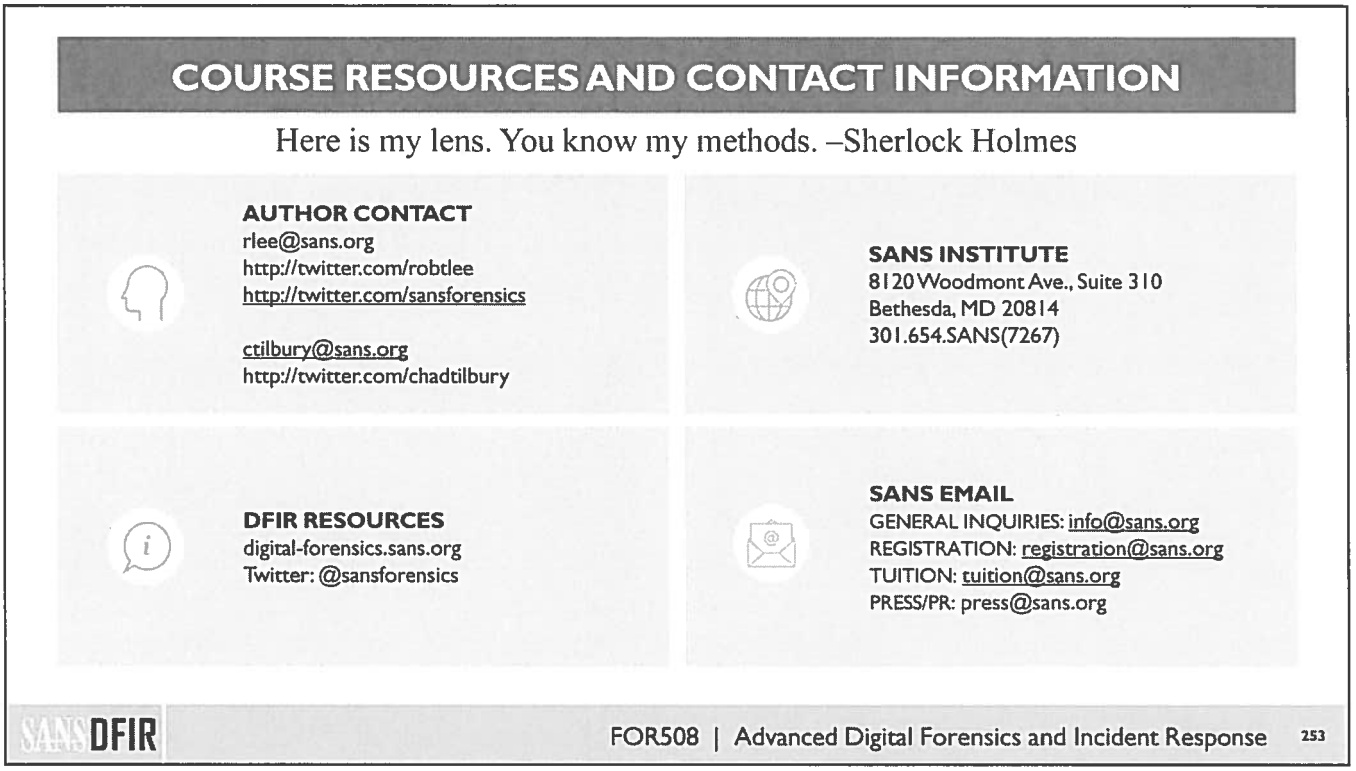
OPERATING SYSTEM & DEVICE IN-DEPTH

- FOR408 **Windows Forensics**
- FOR518 **Mac Forensics**
- FOR526 **Memory Forensics In-Depth**
- FOR585 **Advanced Smartphone Forensics**

INCIDENT RESPONSE & ADVERSARY HUNTING

- FOR508 **Advanced Incident Response**
- FOR572 **Advanced Network Forensics and Analysis**
- FOR578 **Cyber Threat Intelligence**
- FOR610 **REM: Malware Analysis**
- SEC504 **Hacker Tools, Techniques, Exploits, and Incident Handling**
- MGT535 **Incident Response Team Management**

@sansforensics | sansforensics | dfir.to/DFIRLinkedInCommunity | dfir.to/gplus-sansforensics | dfir.to/MAIL-LIST



This page intentionally left blank.