SANS

www.sans.org

FORENSICS 526

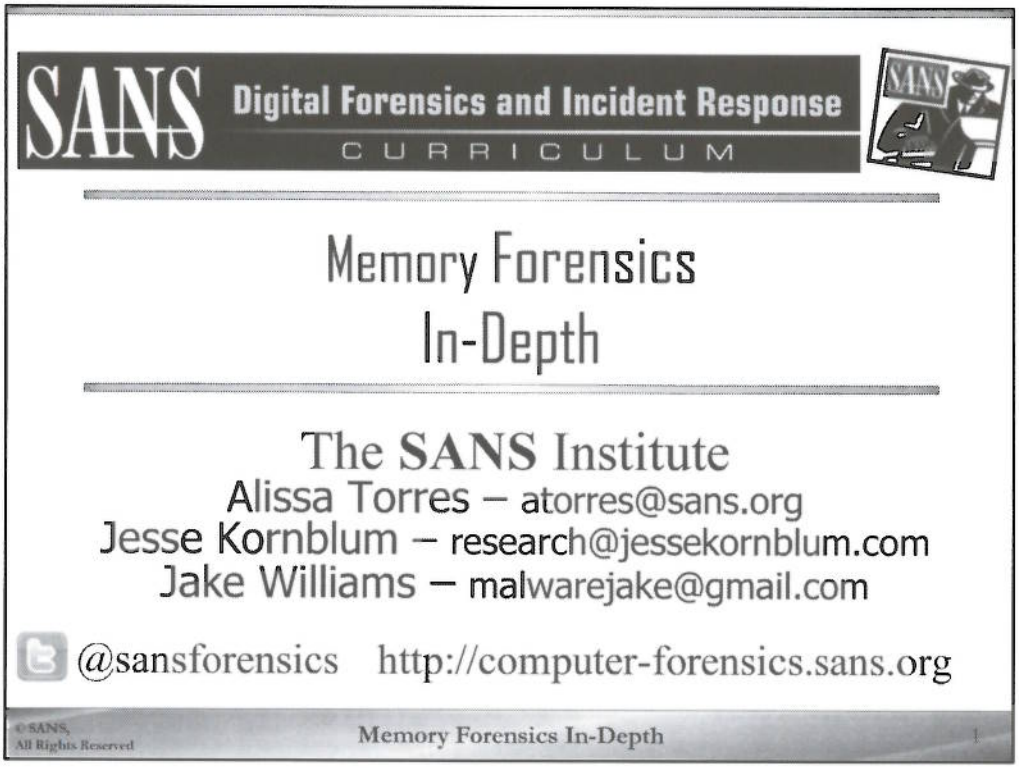
MEMORY FORENSICS IN-DEPTH

526.1

Foundations in Memory Analysis and Acquisition

The right security training for your staff, at the right time, in the right location.

For526_1_A06_01



Welcome to Memory Analysis In-depth, a course that like its subject matter is constantly evolving and being updated. The course authors welcome feedback and future questions you may come upon as you apply the practical skills you gain throughout the duration of the course. Below is the contact information for these course authors.

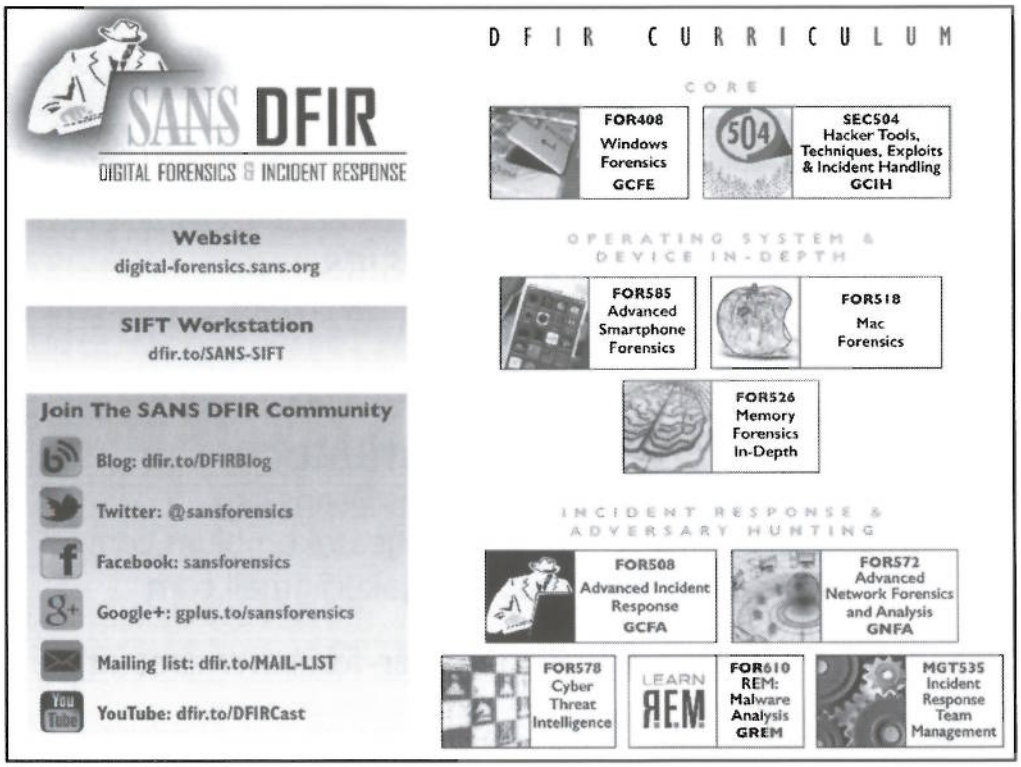
Alissa Torres – atorres@sans.org
http://twitter.com/sibertor

Jesse Kornblum – research@jessekornblum.com
http://twitter.com/jessekornblum

Jake Williams - malwarejake@gmail.com
http://twitter.com/malwarejake

http://twitter.com/sansforensics

We would like to thank the SANS course production team for their support in helping refine this course material. In addition, we would like to thank Rob Lee, Chad Tilbury, Michael Cohen, Matonis (name here), Benjamin Delpy, and Francesco Picasso for the contributions to the forensics community and inclusion of their tools in the FOR526 course materials and hands-on exercises.



This page intentionally left blank.

SANS DFIR

DIGITAL FORENSICS & INCIDENT RESPONSE

Website
digital-forensics.sans.org

SIFT Workstation
dfir.to/SANS-SIFT

Join The SANS DFIR Community

- Blog: dfir.to/DFIRBlog
- Twitter: @sansforensics
- Facebook: sansforensics
- Google+: gplus.to/sansforensics
- Mailing list: dfir.to/MAIL-LIST
- YouTube: dfir.to/DFIRCast

DFIR CURRICULUM

CORE

FOR408
Windows Forensics
GCFE

SEC504
Hacker Techniques, Exploits, and Incident Handling
GCIH

IN-DEPTH INCIDENT RESPONSE

FOR508
Advanced Incident Response
GCFA

FOR572
Advanced Network Forensics and Analysis
GNFA

LEARN REM

FOR610
REM: Malware Analysis
GREM

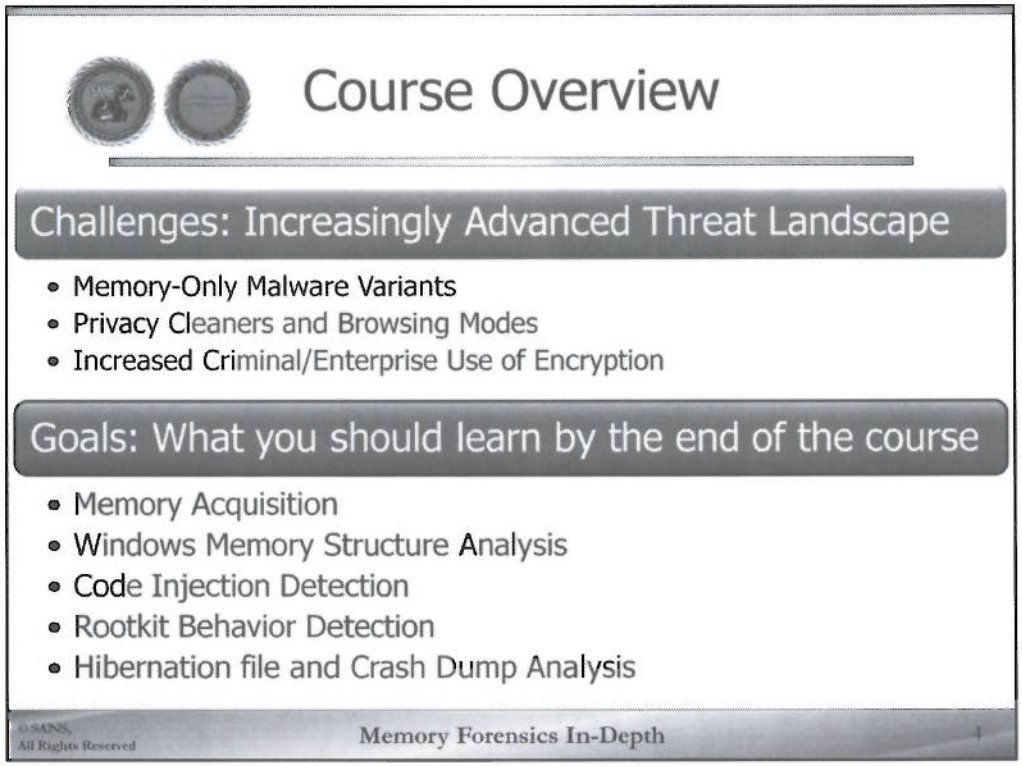
SPECIALIZATION

FOR518
Mac Forensics

FOR526
Memory Forensics In-Depth

MGT535
Incident Response Team Management

FOR585
Advanced Smartphone Forensics



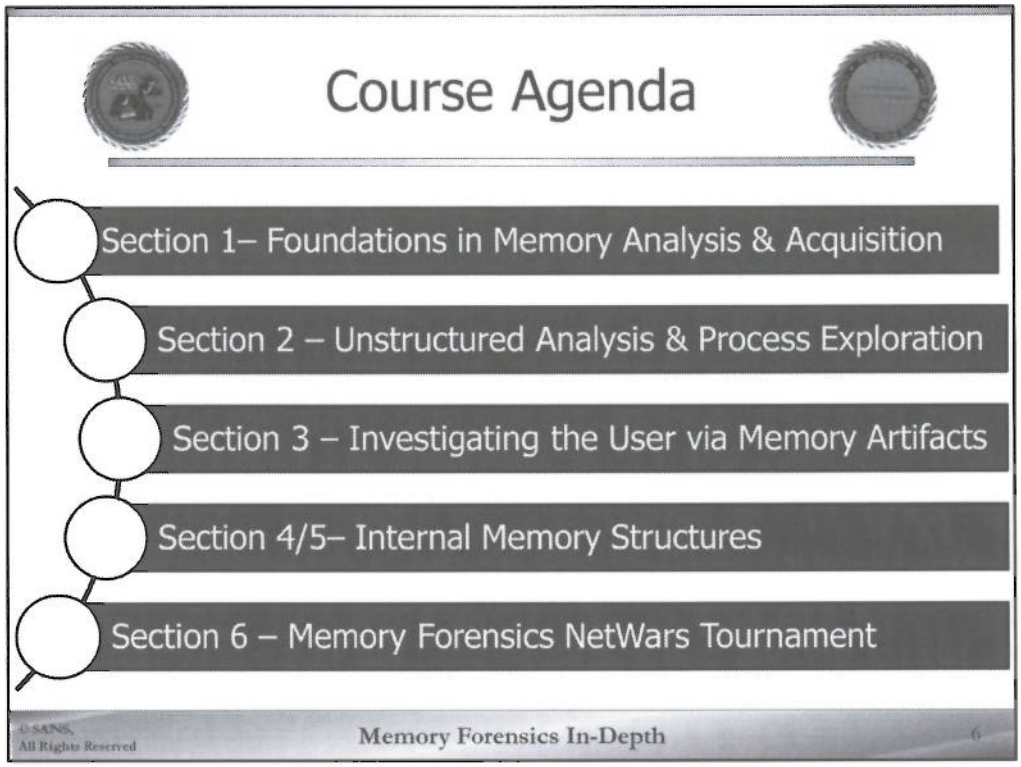
Forensic investigations of all types have grown progressively more complex. Modern malware possesses capabilities that can easily evade file system forensics through executing in system memory only or exhibiting advanced hiding techniques that obscure its presence from today's traditional analytic techniques. Additionally, the prevalence of encryption and memory-only user applications, such a privacy-mode browsers and instant messaging clients, points to the increasing sophistication of today's average user and raises the bar for investigators charged with working Acceptable Use Policy (AUP) employee cases or criminal investigations.

Understandably then, it can be said that if an investigator is not capturing memory as part of a forensic examination, they are "leaving evidence on the table". Whether it be the malicious code that exists only in memory that could have possibly acquitted a suspected DDOS (distributed denial of service) participant or the Skype conversations from a missing teen's personal laptop, capturing the contents of system memory provides a window into real-time system state that can have critical implications for digital forensics cases. In some instances, this window can present key artifacts from memory that are the "smoking gun" for an investigator.

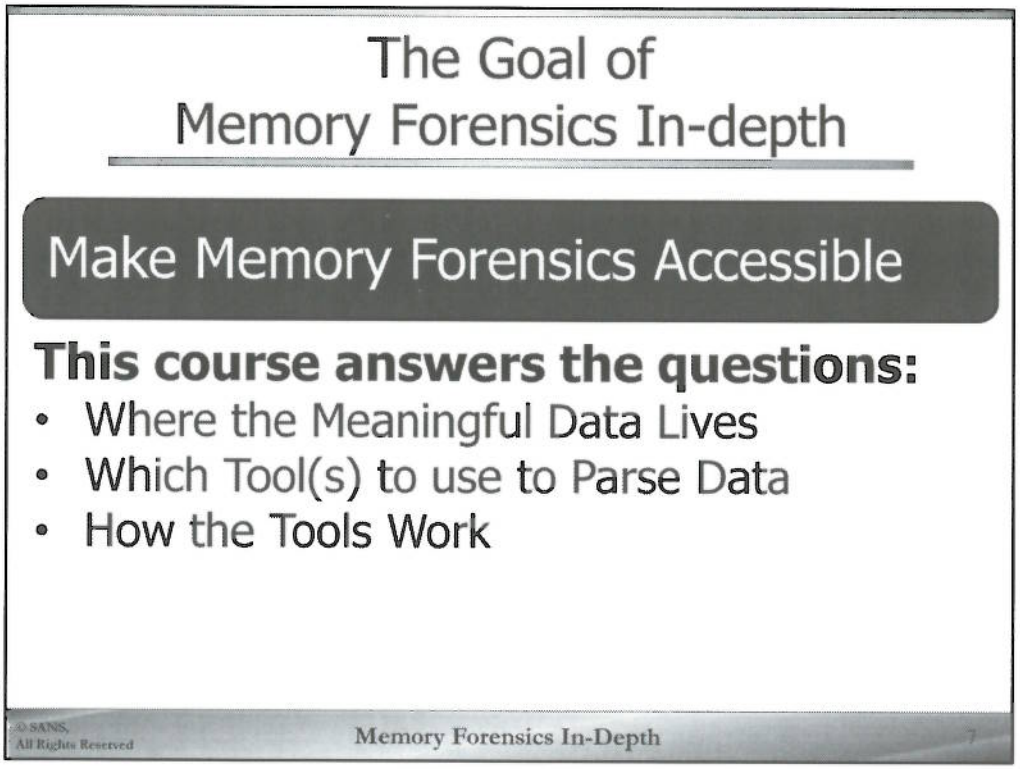
The skills we have found essential to digital forensic examiners and incident responders structures this course content.

- Memory Acquisition - Acquiring system memory is simple enough, but the resultant image varies widely depending upon the method and tool employed. In this course, we will go deeper into understanding how acquisition techniques can affect an investigation.
- Windows Memory Structure Analysis - Memory analysis tools are slowly becoming more intuitive and easier to use. Yet, with the constant changes of memory structures, "go to" tools may fail. By understanding how a tool works "under the hood", forensic examiners can both, spot errors in tool output and configure these tools for their specific investigational needs. The course provides the foundational knowledge of Windows Internals and Memory Structures necessary for intelligent analysis.

- Code Injection Detection - Not all malicious code is going to run as its own process. Commonly, malware will inject dlls and malicious shell code into the address space of a legitimate process, obfuscating its presence from traditional live system triage. This course provides practitioners will methods for effective code injection detection.
- Rootkit Behavior Detection - Subversion of the operating system through the use of rootkit functionality allows malicious processes, network connections, kernel modules and the like to hide from most traditional live system analysis techniques. You will go beyond this cursory view of system state and learn techniques allowing for rootkit identification.
- Hibernation file and Crash Dump Analysis - Many times, while working an investigation, an examiner will not have the opportunity to collect system memory or live system audits from the target system. In theses instances, we can achieve insight into historic system state through the analysis of hibernation files and/or crash dump files. We will cover techniques effectiveness in the identification and analysis of these files.



This page intentionally left blank.



This course will teach you what your GUI tools are doing behind the scenes. We will be using a variety of tools throughout this class - many of them command line, allowing for very granular analysis of Windows memory artifacts. The draw of using CLI (command line interface) tools is there is a tendency to provide far less assistance to the examiner and to require a higher threshold of entry for their use. You will have to do some deep dive analysis over the next five days that you won't be required to perform in the field because your tools have simplified these steps. This is a feature of this class, not a bug.

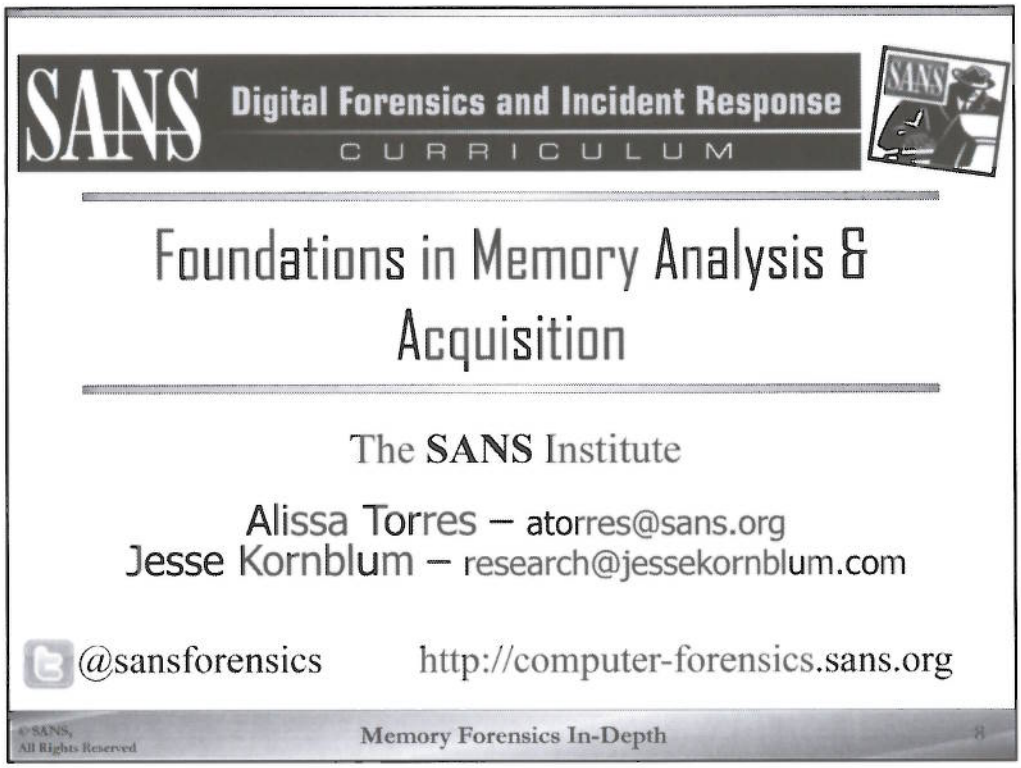
Why is it essential to understand the HOW and WHY of what our memory parsing tools are doing?

We can relate this to file system forensics. When you learned about file systems, you probably had to look at the NTFS Master File Table. Hopefully you had to do some math involving converting a data run for a non-resident data attribute in order to locate the file in the clusters of the volume. In the real world, you don't do that conversion by hand. You have a tool which does it for you. But you have to do the data run conversion at least once to understand how the evidence is put together.

Understanding what your tools are doing will pay dividends when you have to explain where and how you found the evidence in your next big case. You didn't need to parse the MFT by hand in that case. But you need to be able to explain the concept of data streams before you can explain to the jury how there is an alternate data stream. It's not enough to get the data out. You also have to be credible in explaining how you got it out.

What if my tool fails?

With each new version of Windows comes subsequent changes to Windows memory structures. Memory forensics is a moving target - with each service pack, there is another opportunity for Microsoft to change how Windows manages memory. Our tools and their researchers and developers do a pretty decent job of keeping up, but sometimes *tools fail*. By understanding how a tool works, you, as the investigator, may be able to pick out discrepancies and fix them - without having errors affect your investigations.

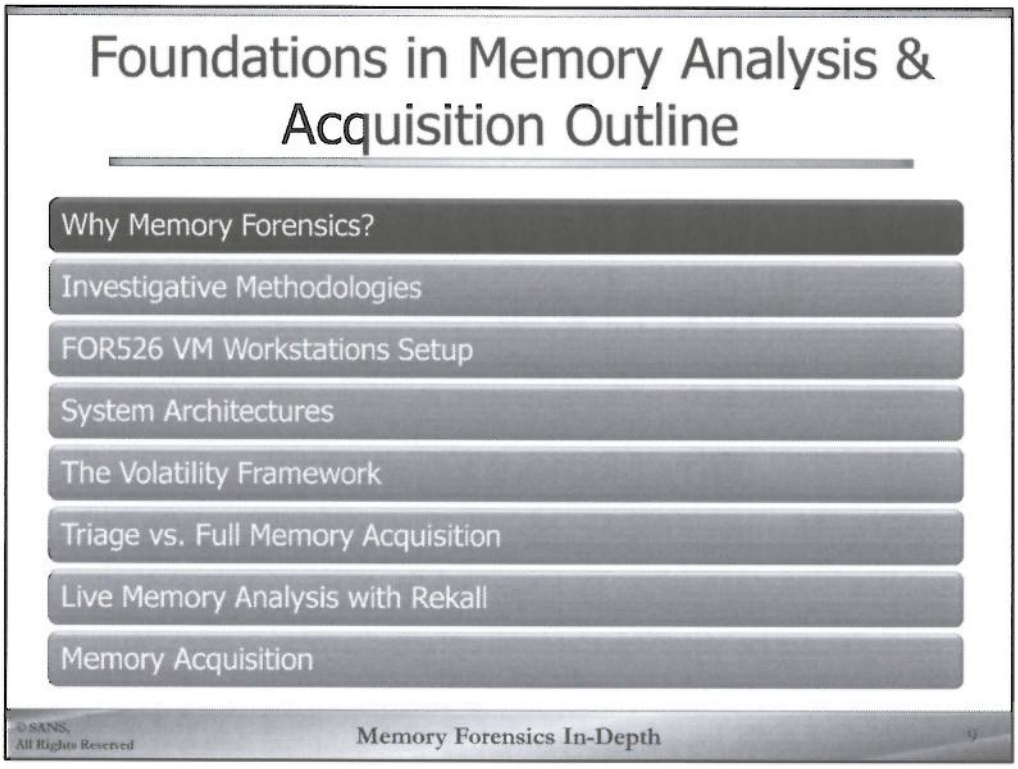


Welcome to Memory Acquisition and Unstructured Analysis.

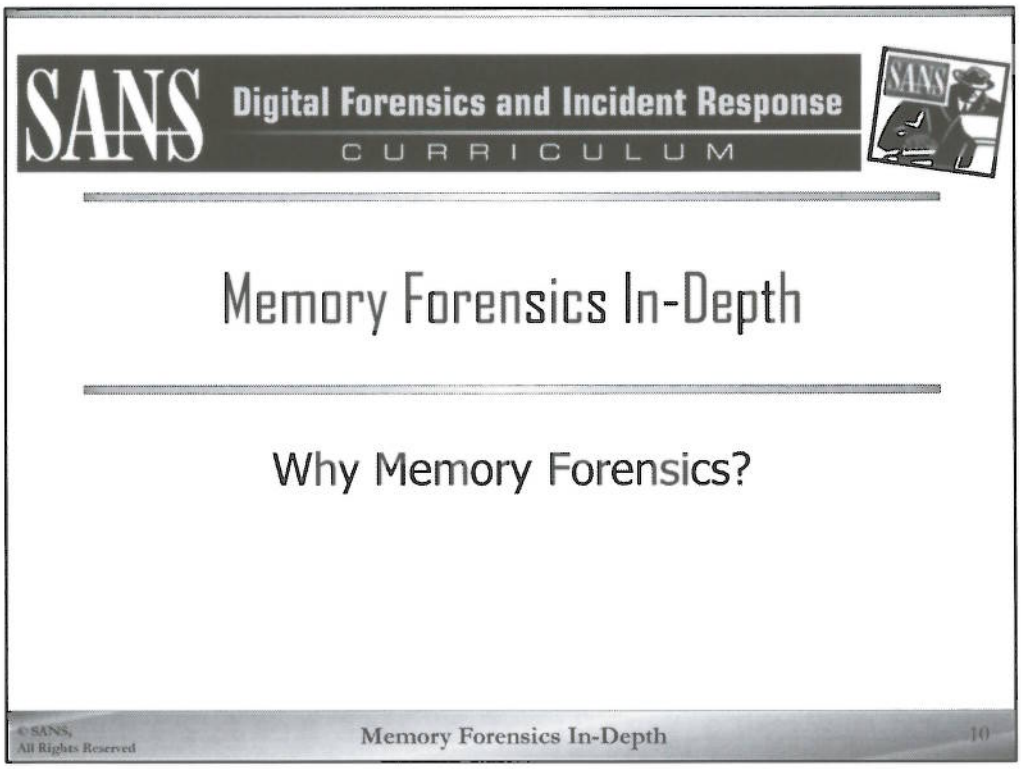
Authors:
Alissa Torres – atorres@sans.org
http://twitter.com/sibertor

Jesse Kornblum – research@jessekornblum.com
http://twitter.com/jessekornblum

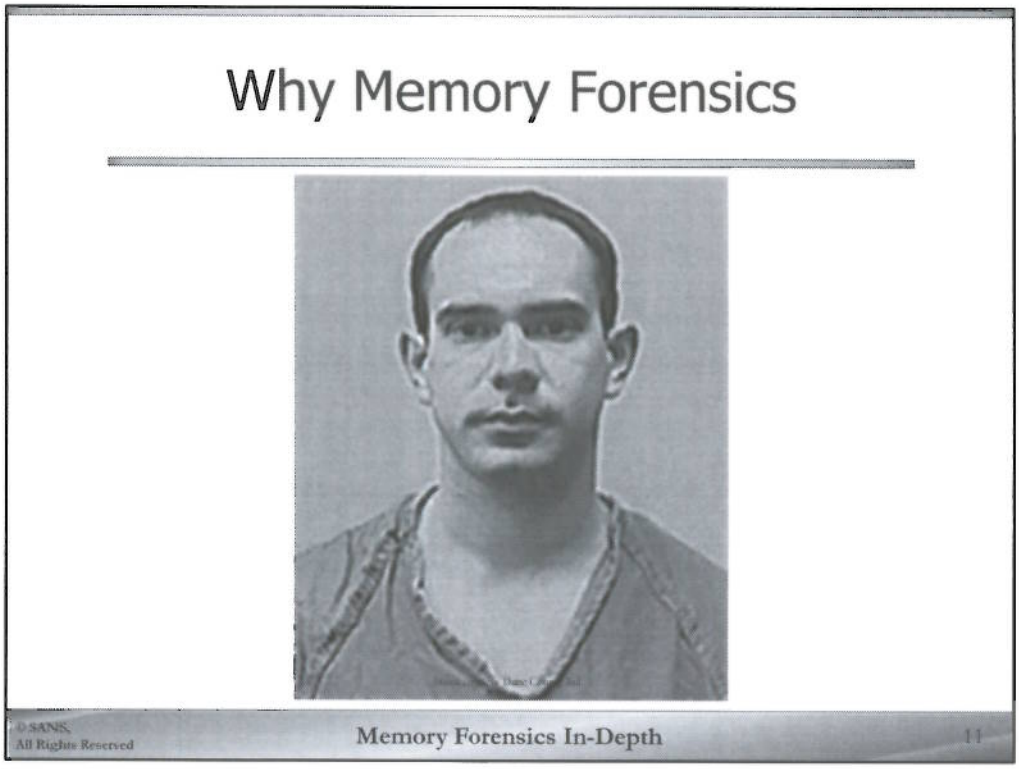
http://twitter.com/sansforensics



This page intentionally left blank.



This page intentionally left blank.

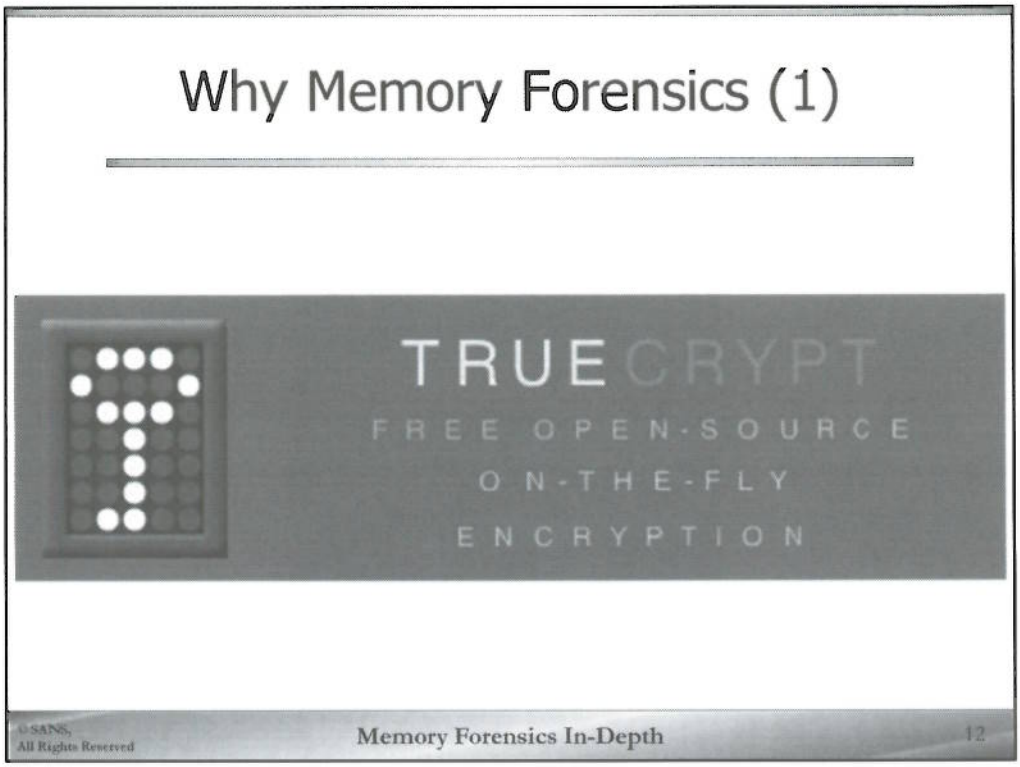


This man was Rajib K. Mitra. He was originally arrested for interfering with police radio communications in Madison, WI and sentenced to several years in prison. While he was incarcerated, he sued the Madison Police Department and individual officers of the department. He demanded the return of his personal computer. Mitra claimed that the computer, which was seized when he was arrested, contained his valuable personal property. The contents of the computer's hard drive were encrypted. The examiners could see the filenames, but not the contents of those files. The filenames, however, suggested the presence of child pornography.

While Mr. Mitra was in prison memory forensic investigative techniques grew in sophistication. It turned out Mr. Mitra's computer contained a hibernation file, a compressed memory image stored on the machine. This file was parsed and found to contain the encryption key for the protected filesystem. Using the key, detectives were able to access the files on the hard drive and present them as evidence at trial. Mr. Mitra was charged with both creation and possession of child pornography, found guilty and sentenced to another six years in prison.

This is why we do memory forensics. Evidence which would have previously escaped the grasp of a forensic examiner was put within reach thanks to this technology.

Source: http://host.madison.com/news/local/crime_and_courts/article_06b3f2e0-71ec-11e0-97ea-001cc4c002e0.html

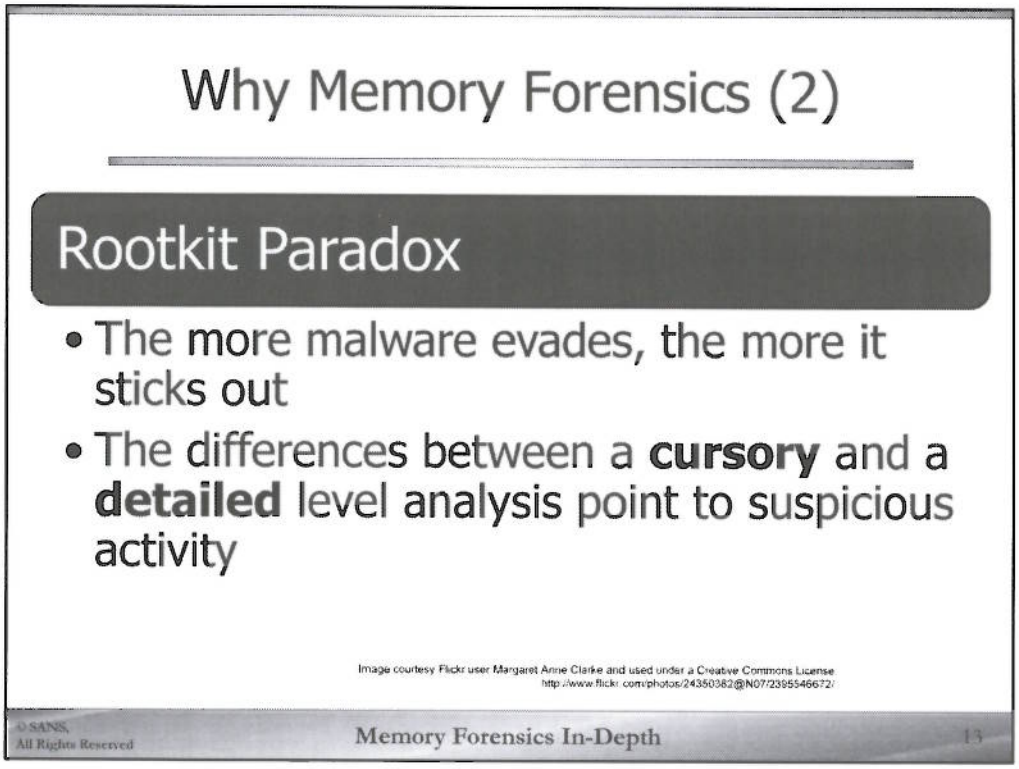


Programs like TrueCrypt are hard targets to analyze. Their protected data appear to be random junk on the disk. Although such files can be detected with an entropy check, if you don't have the passphrase to decrypt the data, you are out of luck. The data is encrypted with AES256, an algorithm which has withstood years of rigorous testing.

Even worse for forensic examiners, programs like TrueCrypt are free, easy to install, and offer full disk encryption (FDE) for Windows, OS X, and Linux. These programs are out there and you will see them in your cases if you haven't already.

Although it used to be possible to recover the passphrase from memory images, TrueCrypt fixed the bug which allowed that years ago.

But don't despair! You will be introduced to tools during this class that will allow the detection and extraction of the AES keys from a memory image, allowing the decryption of a protected volume.



We will also use memory forensics to demonstrate and exploit the rootkit paradox.

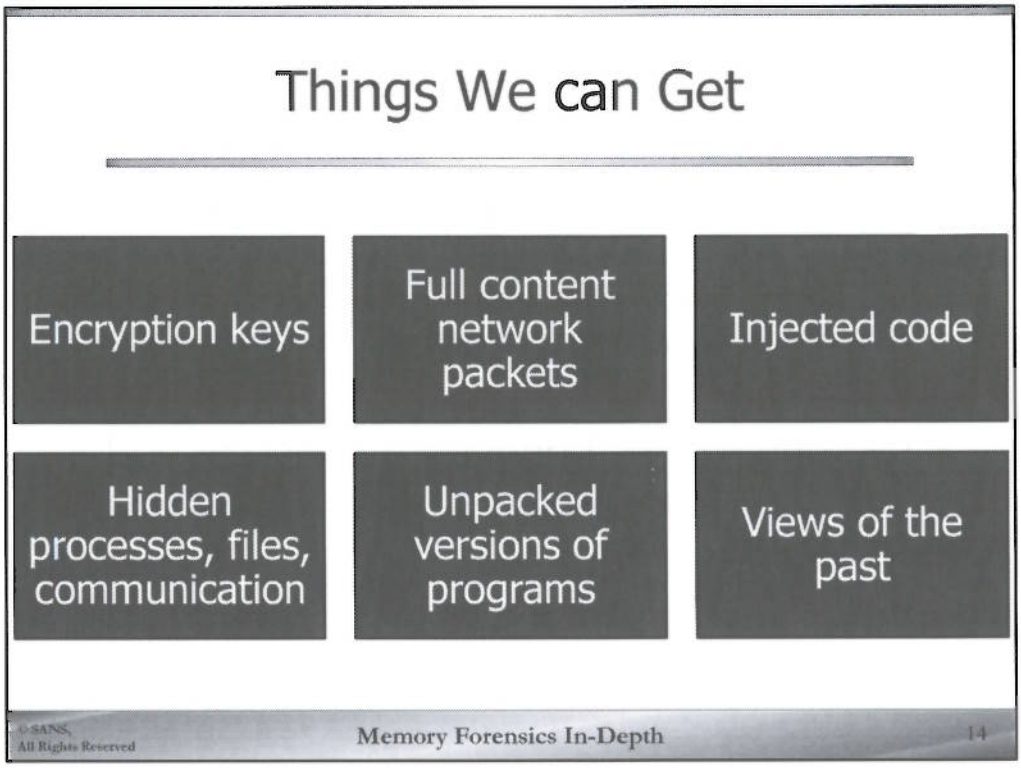
The rootkit paradox is that malicious software wants to run but doesn't want to be seen. It has to be seen by the operating system in order to run. A program needs to allocate memory to hold its code and data. It needs processor time to execute instructions and it needs to access the network to send and receive data. All of those resources are controlled by the operating system. Without the OSs help, malware is useless.

To get around these limitations, malware attempts to cover its tracks. This is different from normal programs, which make no effort to hide their activity. The differences between these two styles of behavior can be detected by doing a high-low analysis of the system. For any kind of resource, such as memory, the processor, or the network, compare a high-level view of the resource use and a low-level view of the resource use. The high-level view should be data which the operating system provides. The low-level view should be a detailed accounting without the operating system's help. Because malware is modifying what the OS sees to hide its tracks, the differences should be obvious. Any differences between the two views are likely the result of malicious activity.

As examiners, we don't care how these differences occur or which processes were involved. We are only trying to isolate the differences and thus the programs which are trying to hide. It is precisely these programs which we are most interested in. This is the paradox: The attempts made by malicious software to hide itself are precisely what lead the examiner straight to it. It is the rootkit paradox, and is fundamental to this course.

The great thing about memory forensics is that it gives us both the low-level view and the high-level view! We can examine the structures the operating system uses and get the same data it would have displayed. We can examine the actual usage of system resources without the operating system getting in the way.

You can read the full Rootkit Paradox paper written by Jesse D. Kornblum at:
http://www.utica.edu/academic/institutes/ecii/publications/articles/EFE2FC4D-0B11-BC08-AD2958256F5E68F1.pdf



We can recover many kinds of data from memory images. We've already mentioned encryption keys and TrueCrypt, but are not limited to data from that program. We can detect and recover encryption keys being used by any program. In fact, we may discover an encryption key being used by a program we didn't even know used cryptography!

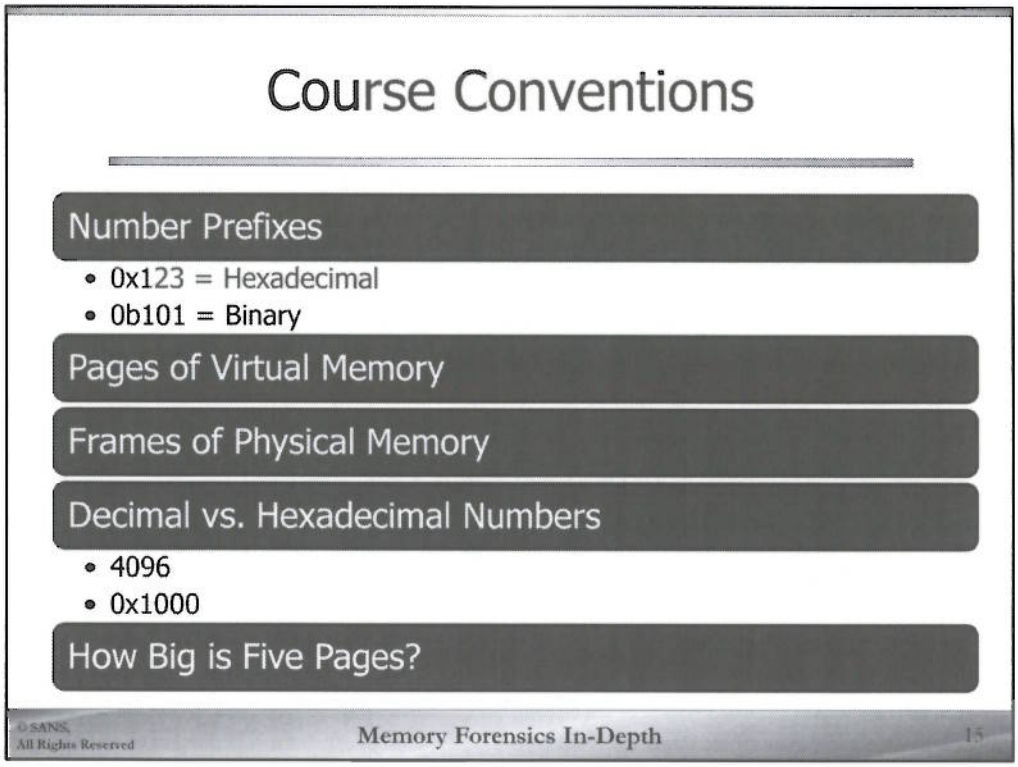
The full content of network communications is stored in RAM. When we capture a memory image, we also capture a snippet of the machine's communication history too. With hibernation files, this snippet could be from months before the memory image was acquired.

Some kinds of malware never write their code to the disk. They are memory resident, and do not leave a footprint for traditional forensics. Because those programs have to run, they have to be in RAM, and therefore we can capture them in a memory image. Memory forensics will enable us to extract these pieces of injected code and analyze them.

The rootkit paradox doesn't just apply to programs, of course. We can detect hidden processes, but many rootkits are designed to hide files and network connections. We can see right through their mechanisms. If anything, the more they try to hide their activity, the easier it is for us to see.

Malware authors frequently pack or encrypt their programs, making them difficult to reverse engineer. Memory forensics allows the examiner to recover unpacked versions of these programs, which can then be reverse engineered with traditional means. In this course we will use rudimentary reverse engineering techniques to match programs back to known software and to detect malicious behavior.

Memory images offer a view into the past. They hold data about terminated processes and communications. Did the user quit the program "KiddiePr0nView.exe" when your agent knocked on the door five minutes ago? We may be able to tell that! We may even be able to determine such activity if they rebooted the machine.

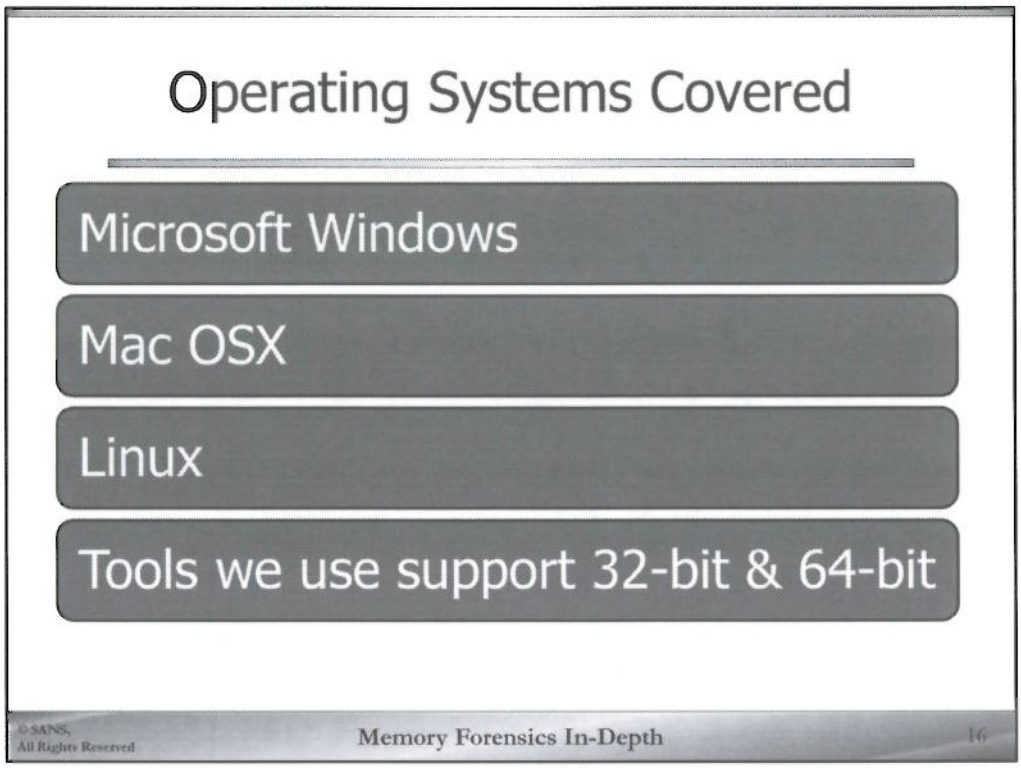


There are some conventions we are going to use in this course.

The numbers in this course will be presented in the base most appropriate for the topic. Most numbers will be in decimal, or base 10, or what you're probably used to dealing with in the real world. Some numbers will be presented in hexadecimal or base 16. These numbers will have a prefix of "0x". Other numbers will be presented in binary or base 2. These numbers will have the prefix "0b".

Computer memory is divided into addressable units. Much like sectors and clusters on a disk, we need terms to organize these units. In virtual memory, or in the view the operating system sees, the smallest unit of memory is the page. In physical memory, or in the view the processor sees, the smallest unit is a frame. These units are the same size, but we use the different terms to keep track of which party is referencing the memory in question and where that data is stored. The term 'frame' also applies to data being stored in a paging file on the disk.

Pages are either normal or large, depending on what the operating system requests for each page. The size of normal and large pages varies depending on the computer's architecture and configuration. Under most circumstances a normal memory page is 4,096 bytes, or 4KB. Computing multiples of 4,096 gets messy quickly. For simplicity, most references to the size of memory regions or offsets in memory use hexadecimal, or base 16 numbers. The decimal number 4,096 is 0x1000 in hexadecimal. Thus it's easy to say how many bytes are five memory pages. (Okay, yes, the number depends on the size of a page. But let's assume 4KB pages for now.) 0x1000 * 5 = 0x5000. There would be 0x5000 bytes in five memory pages.



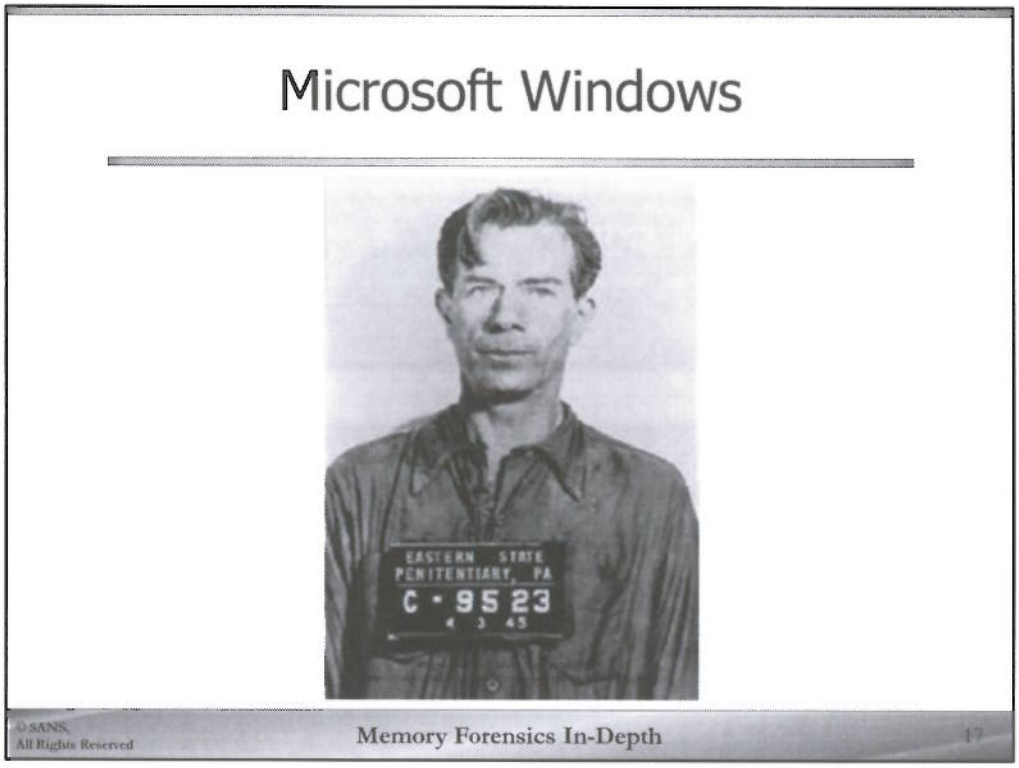
The material in this course is applicable to the following Microsoft operating systems:

Windows 2000
Windows XP
Windows 2003
Windows Vista
Windows 2008
Windows 7
And Windows 8/8.1

In addition to the Windows OS, we will perform analysis of physical memory on Linux and Mac OSX systems during this course.

Much of what we're covering in this course will apply directly to 64-bit operating systems. All of the concepts and most of the specifics will translate directly from 32-bit systems to 64-bit systems. Don't worry too much about the differences right now. There are some additional security restrictions on 64-bit systems and a 32-bit compatibility mode, but for most purposes they are the same. The only difference is the width of addresses and instructions.

We will be using a variety of forensics tools throughout this course, some that are specifically tailored to parse memory structures. Two of our primary "go to" tools are the Volatility and the Rekall Memory Forensic Frameworks, collections of memory parsing plugins that support 32-bit and 64-bit system memory image analysis. The unique feature of both of these frameworks is the granular view into memory structures they both provide, allowing the investigator to move manually step-by-step through their analysis.



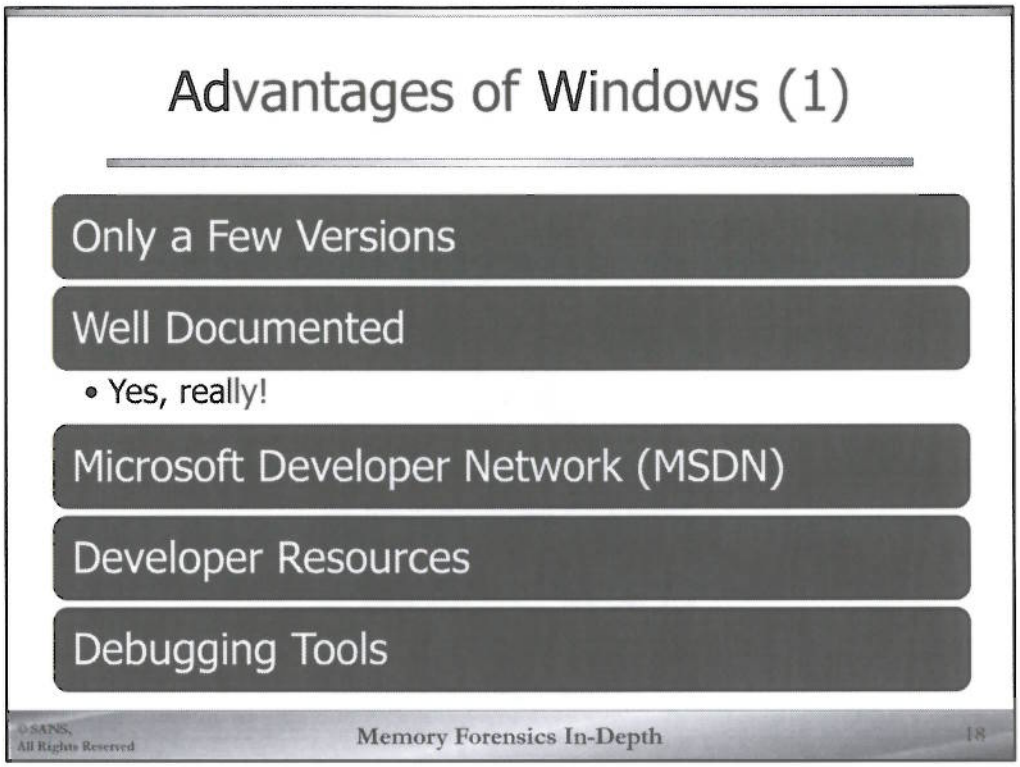
Why do we focus on Microsoft Windows for the majority of this course?

To explain that, remember the Willie Sutton principle. Willie Sutton was a bank robber who's famous for answering a question, "Why do you rob banks?" He replied, "Because that's where the money is."

Why do we study Microsoft Windows? Because that's where the money is. Windows is the dominant platform in use today. Microsoft enjoys more than 91% desktop market share [1] and most systems you will analyze will be Windows.

On Day 5 of this course, we will spend time discussing some of the differences in memory forensics techniques on a Linux/Unix (1.34% of the current desktop market share) platform and OSX (7.21%). You will have the opportunity to work through some memory images from these other platforms in hands-on exercises. For smartphone device forensics, check out SANS 585 Advanced Smartphone Forensics, sans.org/for585.

[1] NetMarketShare Desktop OS as of December 2014, http://www.netmarketshare.com/

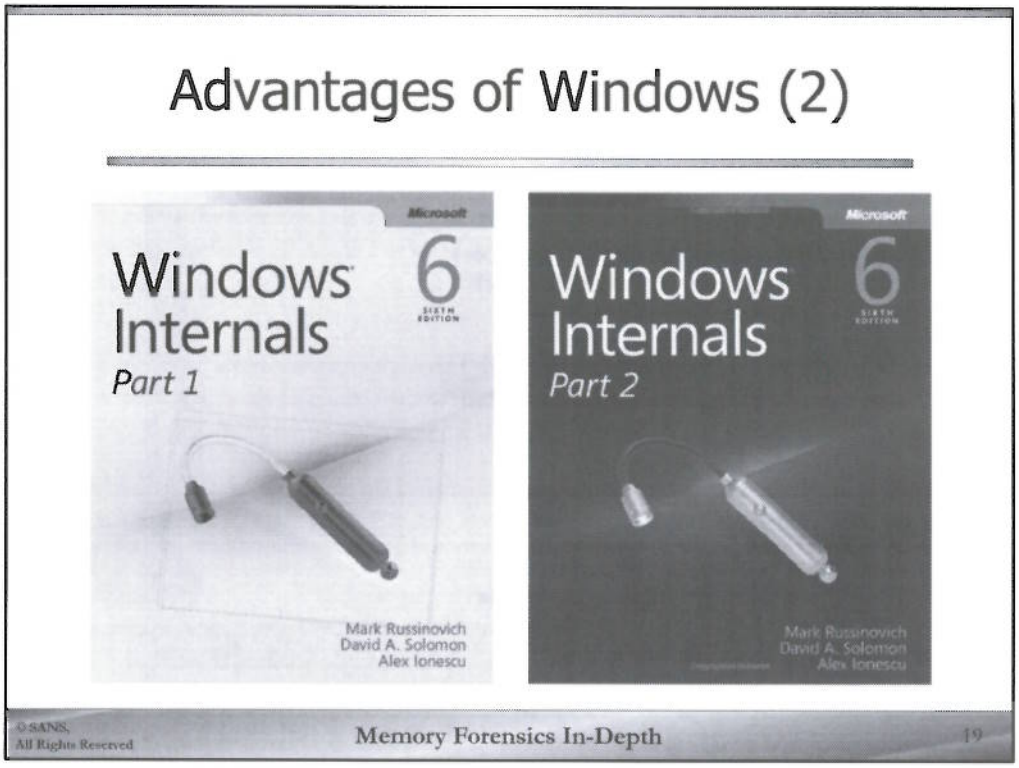


There are many advantages for doing memory forensics on Microsoft Windows. First, there are only a few major revisions of the operating system. Windows 2000 stands by itself. Windows XP and Windows 2003 are based on the same internals. Windows Vista is on its own, and Windows 2008R2 and Windows 7 share the same version of the Windows kernel. Even with those revisions, each new OS builds on the previous. Microsoft has taken great pains to make Windows backwards compatible.

Second, even with the revisions, many of the details under the hood have stayed the same. The look and feel gets updated with each release. But enhancements to the user experience do not necessarily reflect systematic changes to the operating system internals. Even with 32 and 64-bit versions of the operating systems, service packs, and hotfixes, there is still an easily countable number of products to keep track of.

Third, lots of people write about Microsoft Windows. There are literally bales of books on the market about Windows, how it works, and how it's put together. The Microsoft Developer Network (MSDN) is a fantastic source of technical information, straight from Microsoft. They also publish lots of articles and guides on how to interact with Windows. Why do they do that? Developers! Microsoft wants developers to write software for their operating system. Without programs to run, the OS is useless.

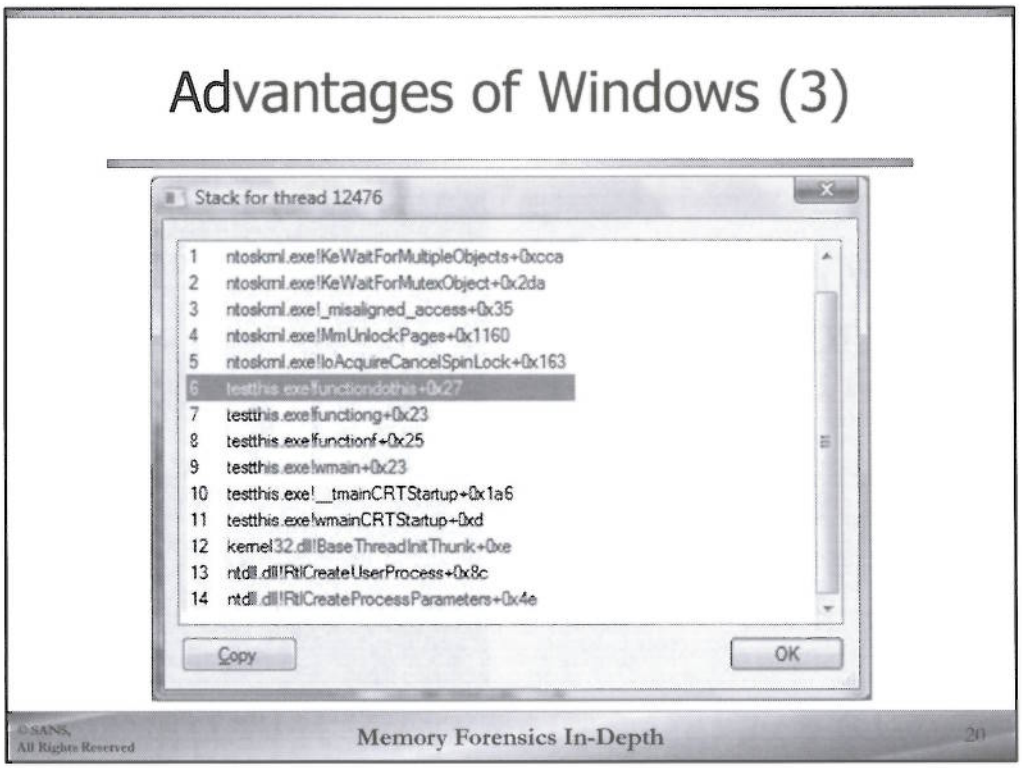
Finally, to make those programs useful, Microsoft wants to ensure developers can fix the bugs in their code. The free debugging tools Microsoft provides are designed to help developers debug their drivers. This is important, as drivers can easily crash the operating system, so Microsoft has a vested interest in helping them write bug free code. We've used these same programs to help us understand how Windows works.



Far and away, the best published resource, in terms of the sheer density of useful information in the book *Windows Internals* by Mark Russinovich, David Solomon and Alex Ionescu. This book details absolutely everything you would want to know about how Windows is designed and supposed to work. It has examples and exercises for you to try. Much of this course was developed from the information in this book.

If you'd like to try before you buy, take a look at the sample chapter available at http://technet.microsoft.com/en-us/sysinternals/bb963901.

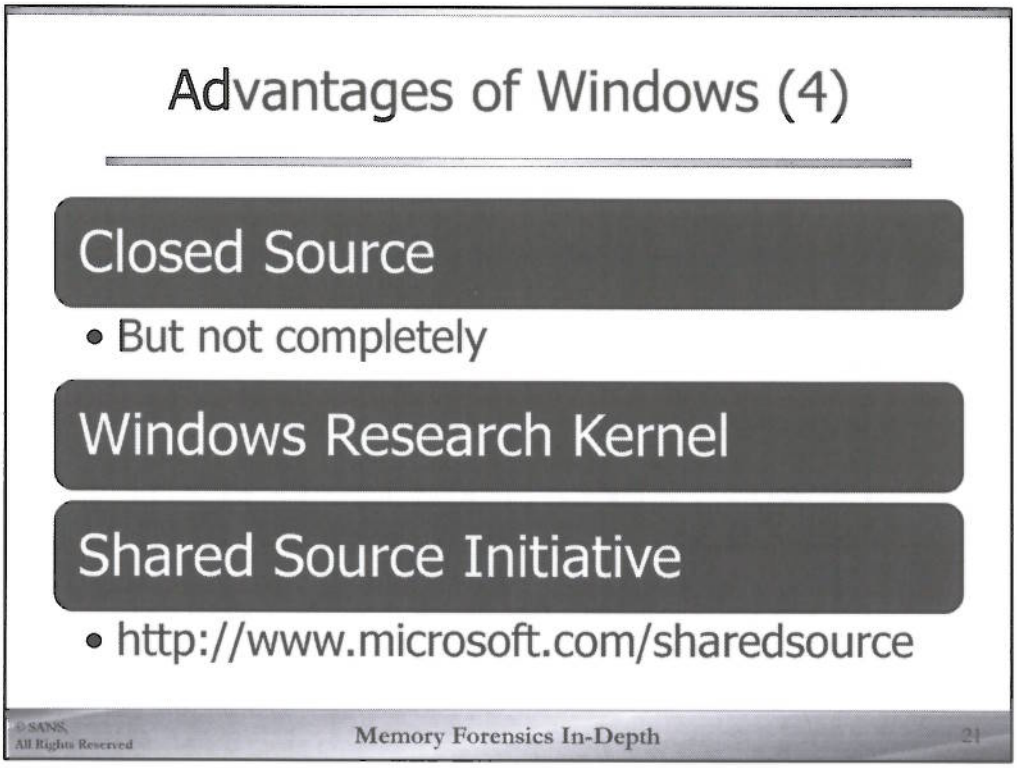
Now, nobody is perfect. Especially when trying to write a two-volume thousand page technical book, mistakes happen. There are errata pages for every edition of Windows Internals. You should definitely check in from time to time and update your copy! The link above also contains the errata for the 6th edition.



Microsoft publishes symbols for all of their products. These symbols give the virtual memory addresses for all kinds of values. In the picture above, we are using the addresses of functions while examining a call stack. A call stack is a history of which functions have called which other functions. Microsoft provides these symbols so that programmers can debug their code. We can use them to see what parts of the operating system are being used. A worthy goal for forensics!

With the symbols we can see the program has called the function KeWaitForMultipleObjects. Without them the function name would just be an address like 0x80485076; not very useful. But we can look up the function name on MSDN or in the Windows Internals book. What does it do? Why is it called? What arguments does it take? We can see all of these, and more, and figure out their impact on our case.

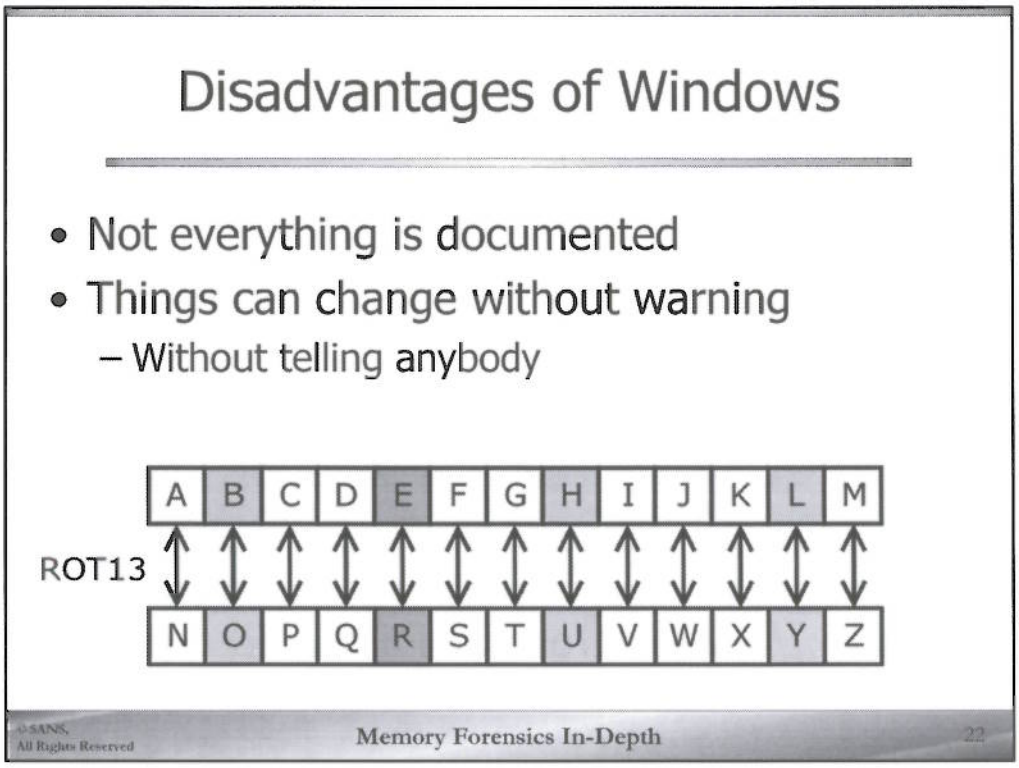
Not only do symbols provide the names of functions, but also the addresses of kernel variables and the composition of operating system structures. These addresses and definitions have saved a lot time in reverse engineering. We don't have to figure out how Windows is put together. Instead we can just compare the data we see to the definitions and instantly understand what's going on.



Microsoft Windows is a closed-source program. You can't download the source code to Windows. Well, unless you can.

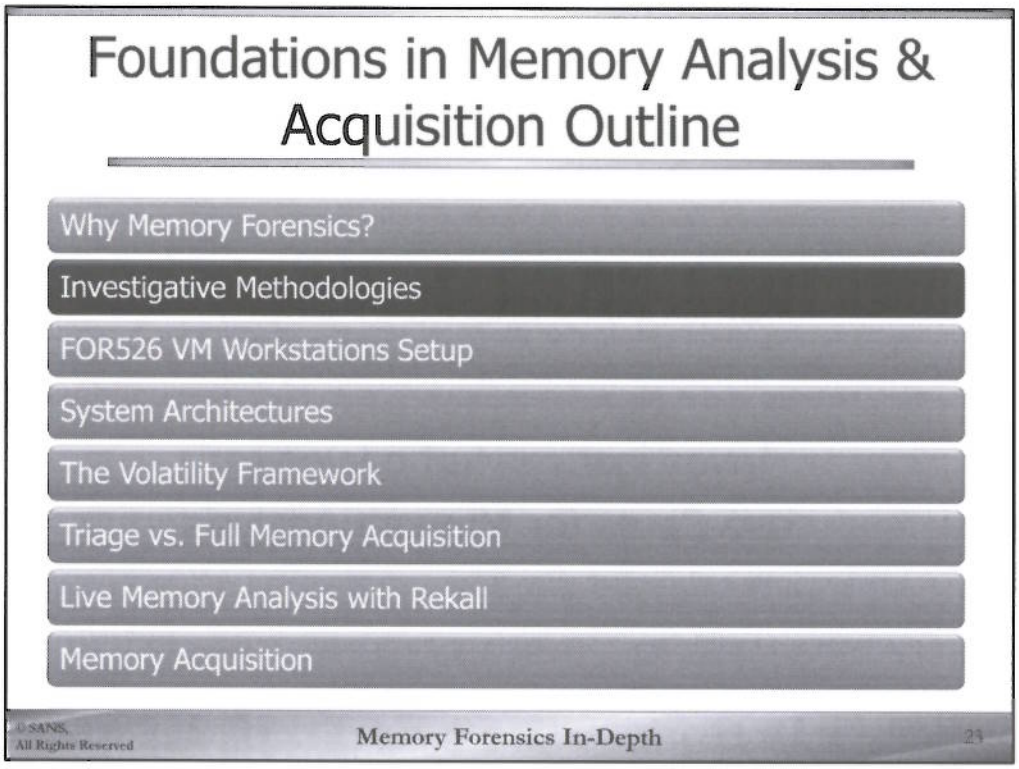
Microsoft does not publish the source code to Windows, but you can buy access to it. Only certain groups are eligible to buy that access. Academics can get a copy of the Windows Research Kernel. The WRK is a stripped down version of the Windows 2003 kernel. Some parts of the kernel are distributed as binary only, but much of it is source code. The kit is designed to allow computer science students to learn how operating systems work and experiment with kernel development. The kit can be compiled, and the resulting kernels can be used to boot and run a computer. You can find copies of the WRK on the Internet from time to time.

But the WRK is only one part of Microsoft's shared source initiative (SSI). The SSI also offers source code access to enterprises, OEMs, Governments, and developers. There are different packages depending on which category you are and how much you're willing to pay.

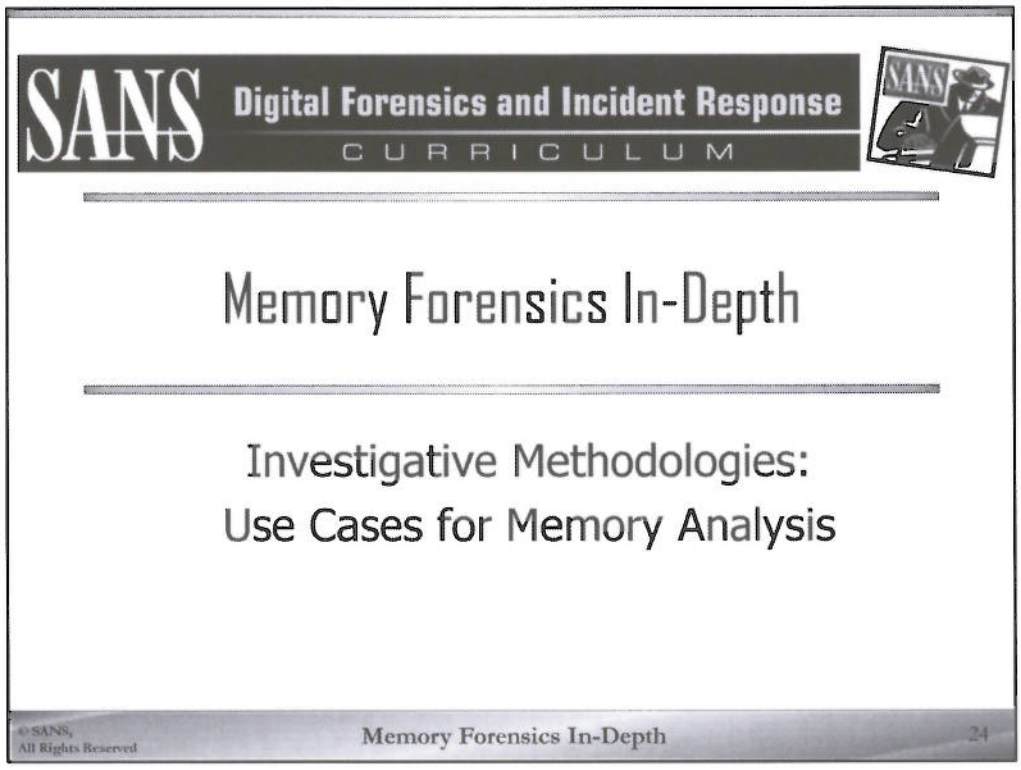


Not everything about Windows is perfect, however. There are many parts of the operating system which are simply not documented anywhere. For example, though most forensic examiners have wondered, no one truly knows why parts of the Windows Registry are encoded in ROT13. Perhaps Microsoft has documentation filed away somewhere. They may not want to publish such information as it's a company trade secret. Of course, there are also proprietary pieces of their operating system which they can't give away for fear of piracy or users stealing the operating system.

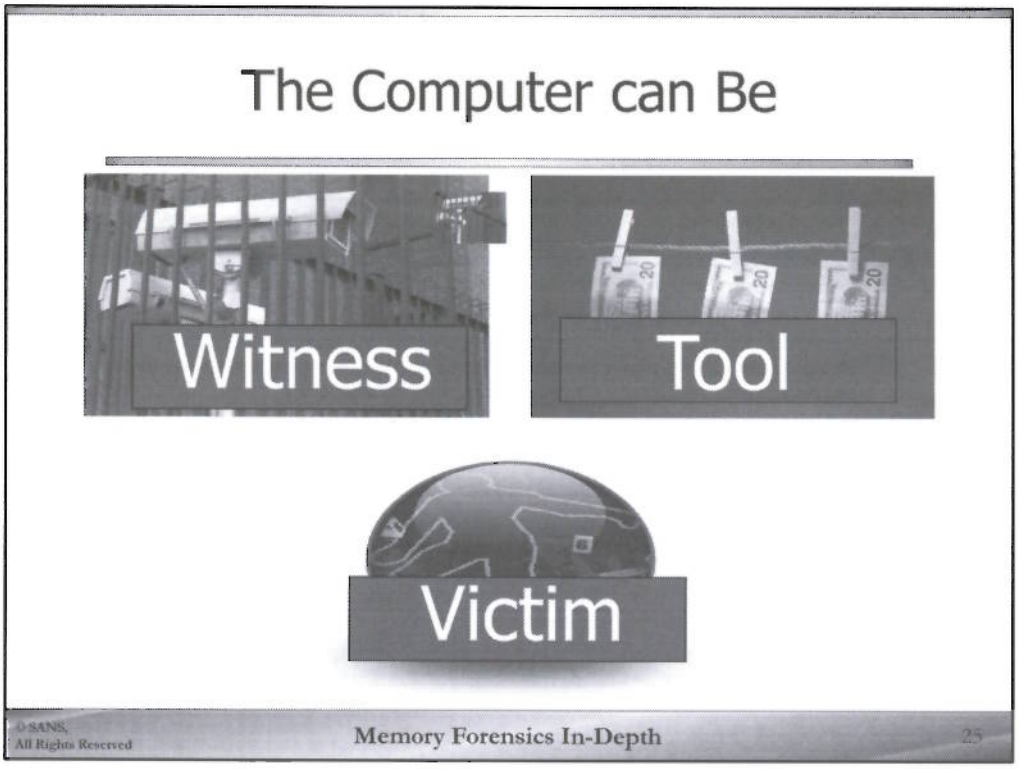
But more often documentation just gets out of date. Microsoft makes changes to Windows all the time, and they don't always tell people about it. Although they are usually good at communicating changes with developers, changes to the truly internal parts of Windows can happen at any time and without any warning. Be careful out there!



This page intentionally left blank.



This page intentionally left blank.

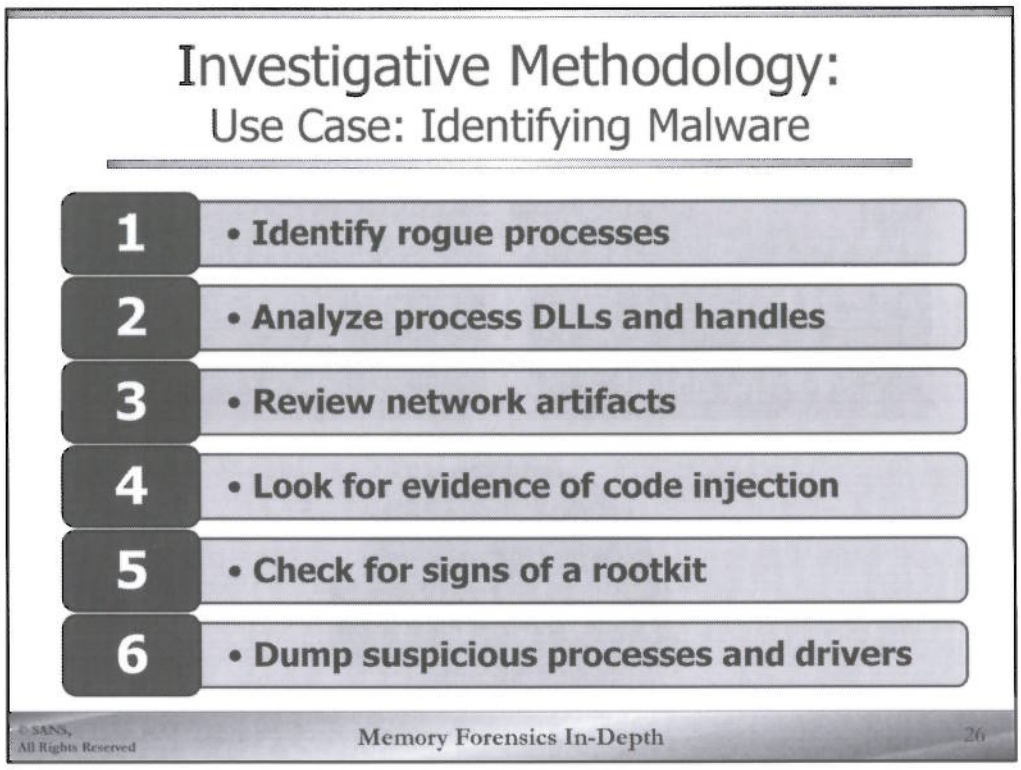


Let's talk about the kinds of data you can pull from memory images and how they fit into your investigations. In general, the data recovered from memory images, and the computer as a whole, will fit into one of three categories.

Witness of a Crime - The crime was committed elsewhere. No criminal act was committed by user or by other people on their computers. But the computer contains information about the criminal activity. Like a video camera, the computer saw something relevant happening.

Tool of a Crime - The computer was used in the commission of a criminal act or is being used to hide the criminal act. The data you are looking for is in the computer somewhere. Finding it will uncover the criminal activity.

Victim of a Crime - The integrity of the computer has been compromised as part of a criminal act. There could be malicious software running on it which will modify, steal, or destroy data.



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

Step 1. Processes: Start with a review of processes because they are the most important objects in memory. What was running on the system? Are there any processes which aren't supposed to be there? Any which are indicative of the attack?

Step 2. Objects: Continue your analysis by scrutinizing all of the various helper objects assigned to each process. Digging into the dlls, threads, handles and memory sections of a process yields significant information during an investigation where the computer is a victim.

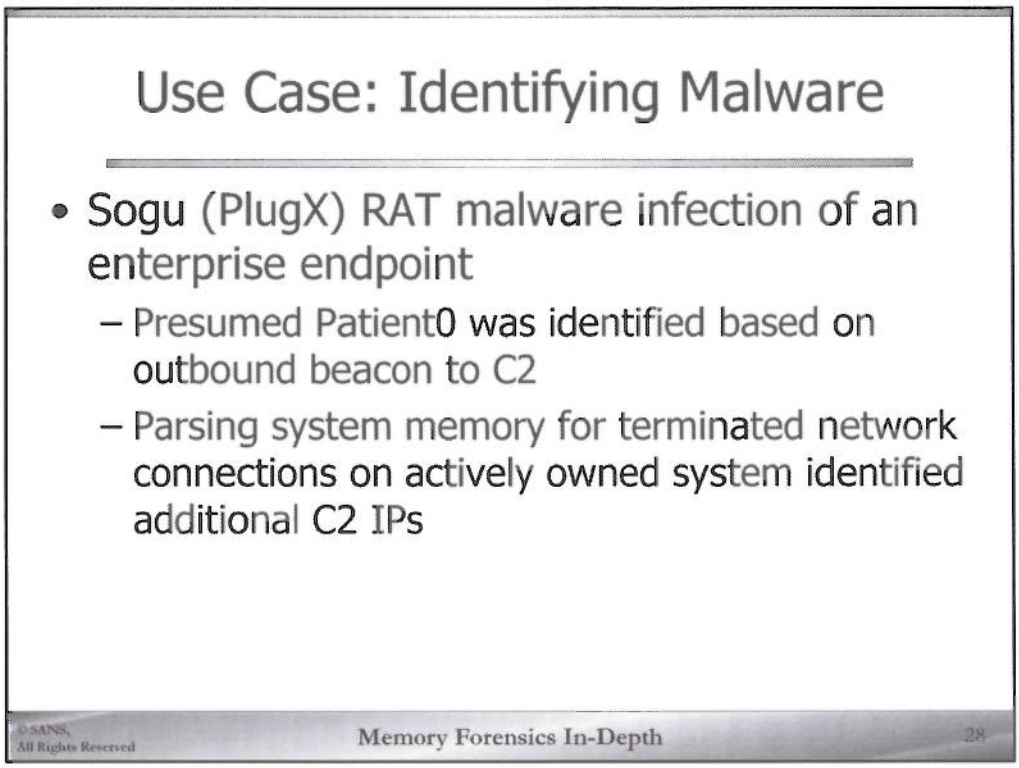
Step 3. Network Artifacts: Whatever malware is running is probably communicating with the outside world. Finding these communications and identify the owning processes can be fruitful in pinpointing malicious code on a compromised system.

Step 4. Code Injection: If you do not yet find anything out of the ordinary in the first three steps, consider searching for signs of code injection, looking to see if malware may have taken over a legitimate process.

Step 5. Rootkit Behaviors: Rootkits may be difficult to spot on a file system, but are often glaringly obvious when looked at through the lens of memory. In the operating system hiding any processes? They aren't normally hidden, so anything which is hidden is almost certainly bad. As the rootkit paradox implies, the more obfuscation and evasion techniques implemented by malware, the easier it is to detect.

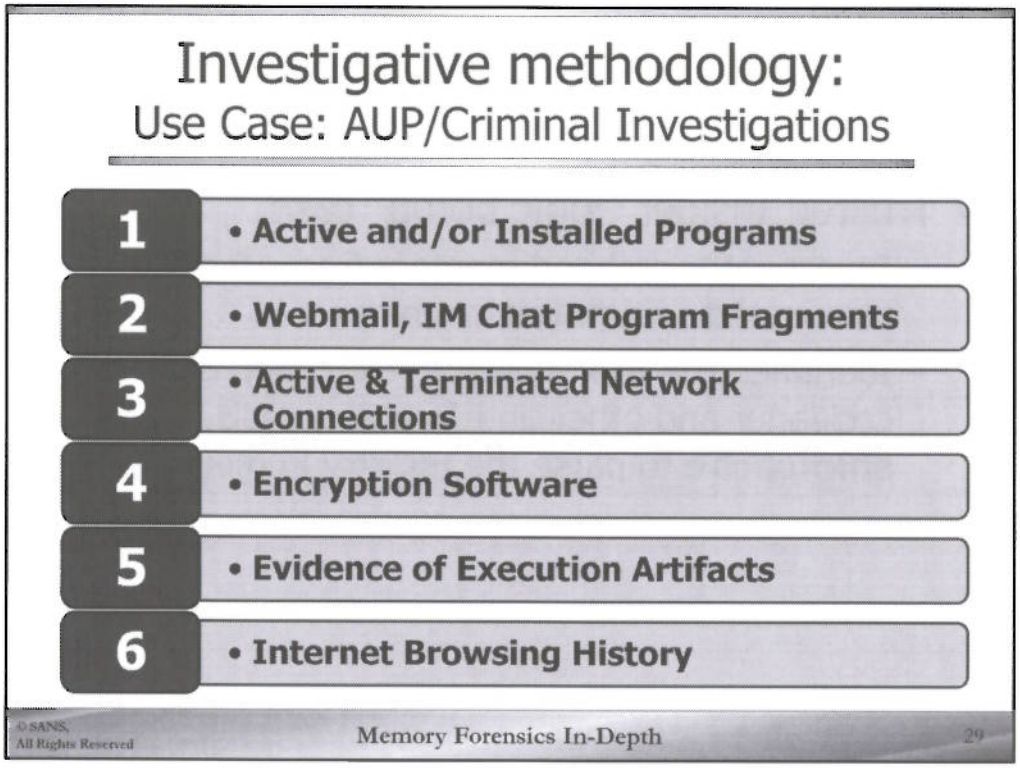
Step 6. Binary Extraction: Finally, extract any suspicious processes, drivers, and memory pages and continue your analysis outside of the memory image via malware reverse engineering techniques or simple anti-virus scans.

Over the next few days, we will examine each of these steps in great detail. Before you leave here, you will be familiar with all of them, what's normal, and have some idea of what looks out of place.



Some of the most exciting memory forensics use cases come under the guise of malware investigations.

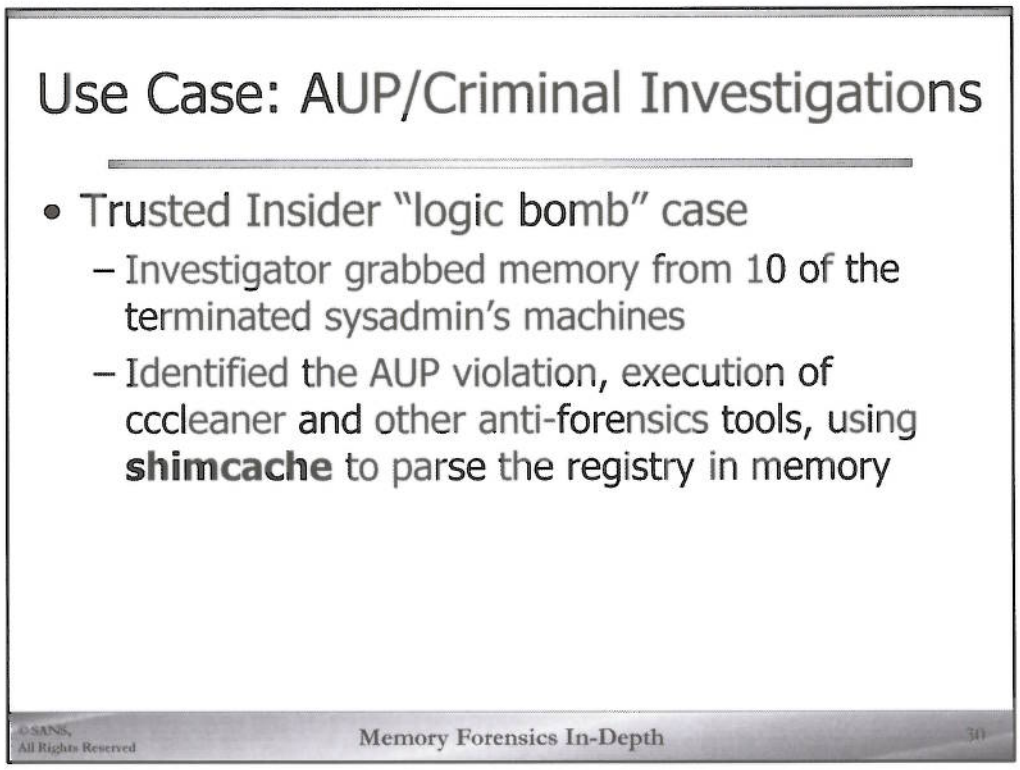
One specific case that clearly highlights the value of having low level access to memory of a potentially compromised system involved a Sogu remote access trojan infection. Based on SNORT alerts, an endpoint on a commercial enterprise was identified as beaconing out to a known bad IP address (previously associated with past SOGU malware infections). Upon investigation, system memory was parsed via a live audit tool and active and terminated network connections were enumerated. The connection that created the alert was identified, but in addition, several other remote IPs from the same region were seen communicating with this endpoint. Using this additional threat intelligence, the newly identified IPs were added to the SNORT blacklist, resulting in a widening scope of the investigation to include additional endpoints.



Many professionals in the digital forensics community work on "cases other than malware" (COTM). This category of investigations includes a wide range of scenarios, from criminal cases, trusted insider cases to acceptable use policy (AUP) violations. Even in intrusion investigations that involve what the adversary did on the network without using malicious code (think stolen credentials here), memory analysis can be extremely fruitful.

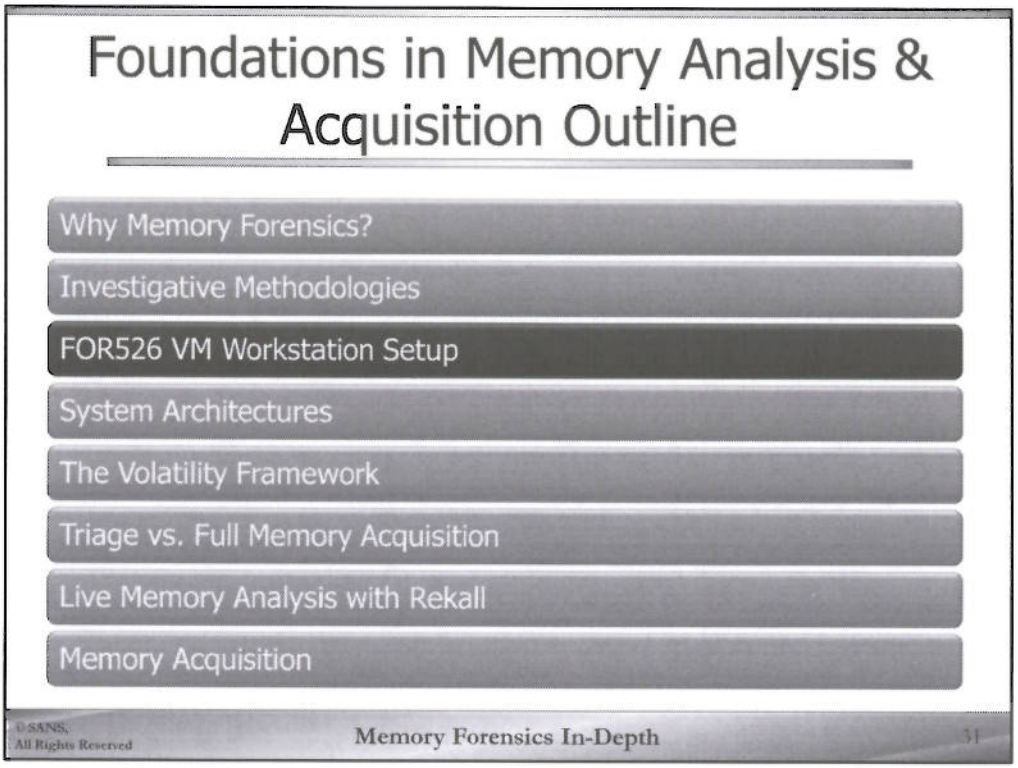
In this course, we will wield memory parsing tools in the pursuit of achieving a clear picture of what a user did (or is actively doing) on a system. We will introduce specific plugins in Volatility and powerful stream based parsing tools that target user artifacts to include browsing artifacts and IM chat conversations that can only be found in physical memory. Many of today's AUP cases involve the installation of rogue applications. Proof of the current or past existence of these applications can be found by parsing registry artifacts found in memory, as well as with traditional file system forensics. Evidence of execution, be it from registry keys or from terminated/active processes, can be the smoking gun needed to prove a suspect's deliberate activity on a system. And finally, due to the increasing prevalence of encryption software, even by the average user on a corporate network, a digital forensic examiner may need to rely on AES keys extracted from a memory image (or hibernation file) in order to gain access to protected, case relevant data.

Clearly, memory forensics has an enormous impact in the outcome of today's typical user investigation. Several of this course's exercises will focus on extracting user artifacts from memory images or live system audits.

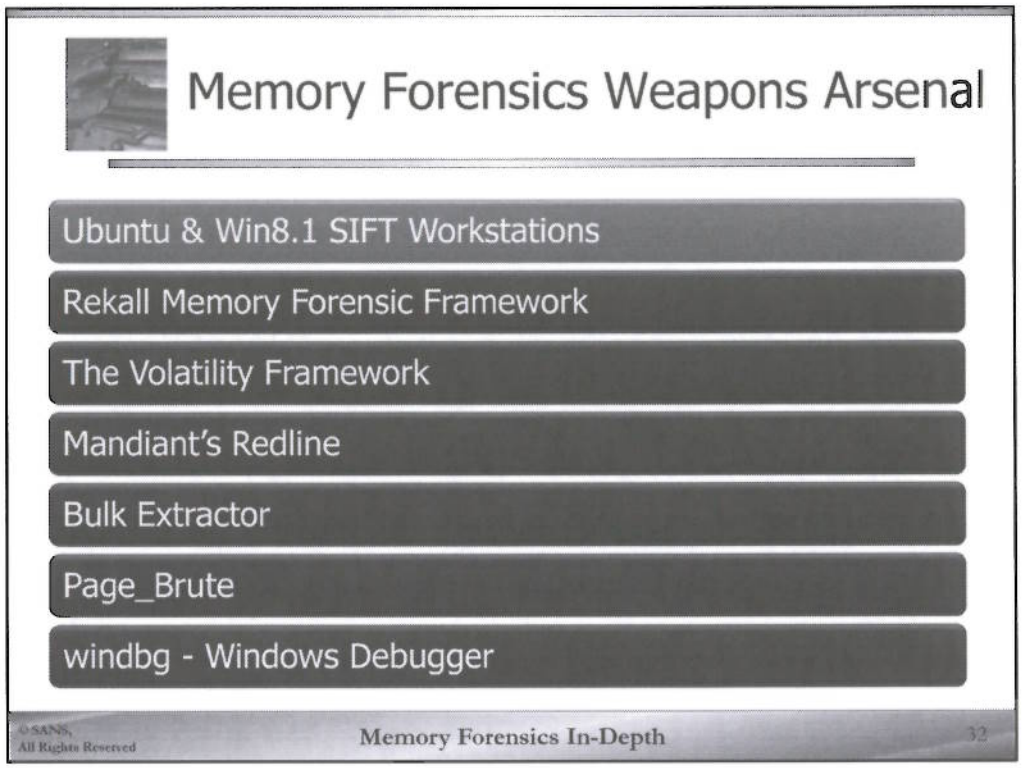


Here is a great example of how memory can be used to substantiate an AUP violation against an employee.

An independent digital forensics consultant was retained to investigate a case of a trusted insider, a recently departed, disgruntled system administrator, suspected of planting a "logic bomb" on several of the network's systems. The investigator decided to grab memory from all 10 identified systems and focus his efforts here due to its smaller search space. Using the **shimcache** plugin from Volatility to pull application compatibility cache[1] from the system registry hive, he located evidence of an AUP violation, the use of cccleaner and other anti-forensics tools on 8 of these systems. Huge win for memory forensics and this investigator (since he didn't have to image 10 hard drives)!!



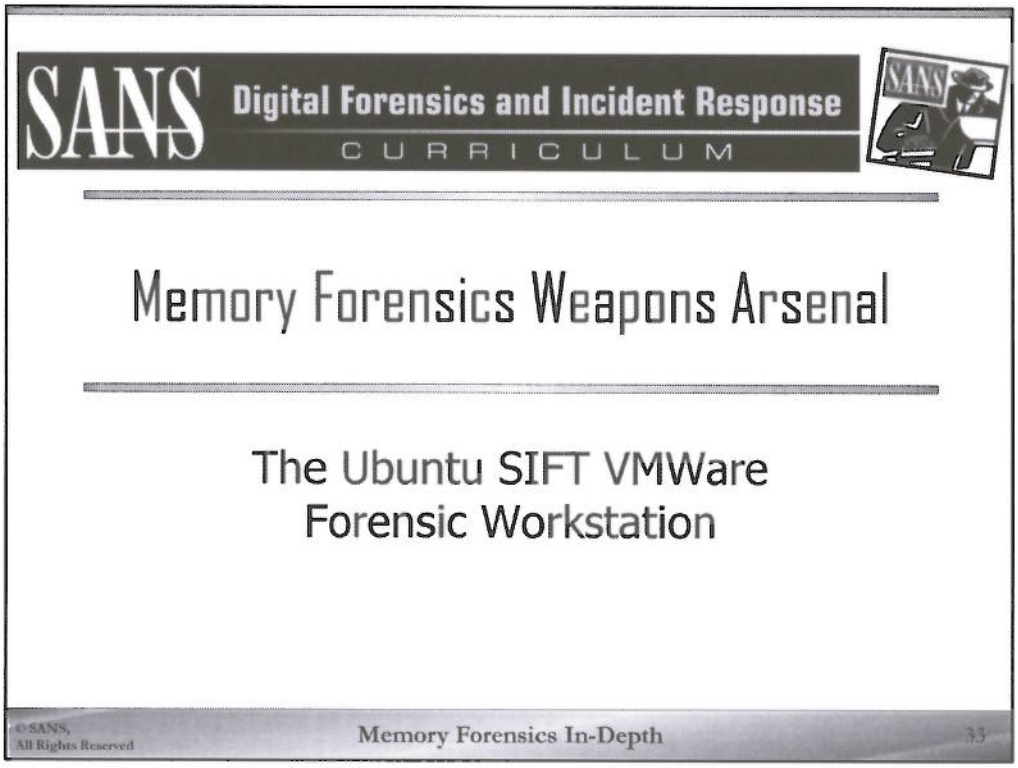
This page intentionally left blank.



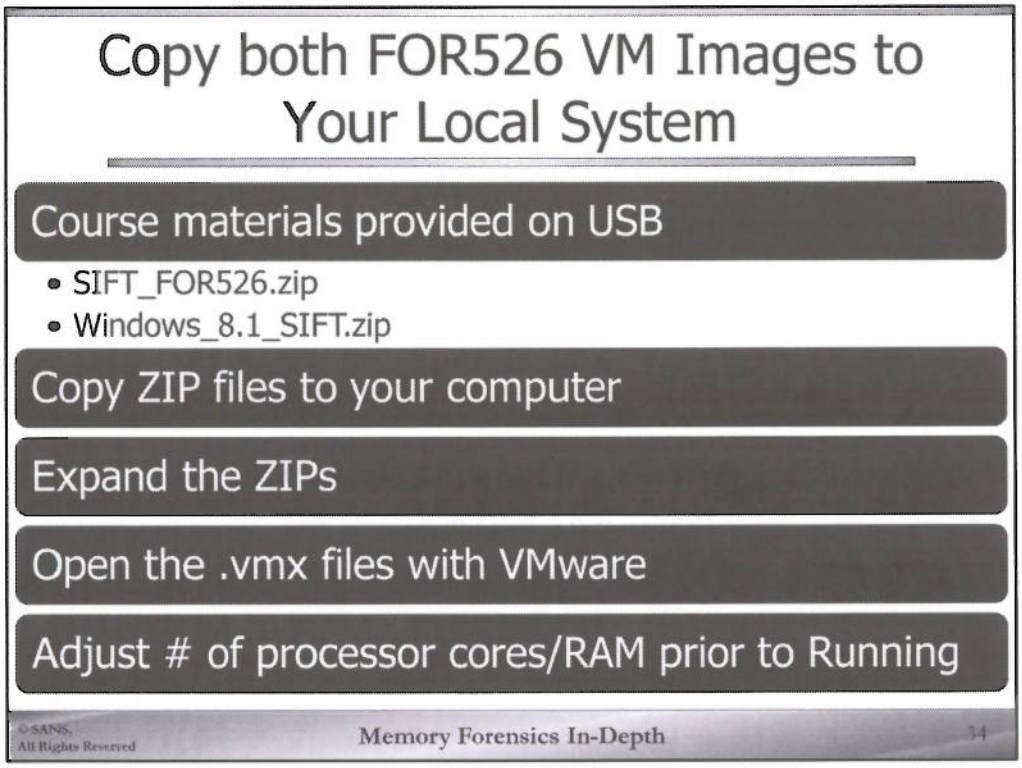
During this course, we will be using various memory and data parsing tools. You may have heard the common question "Which platform should I use for my forensic workstation, Windows or Linux?" Clearly, for this class we have chosen to use both - each one giving examiners access to different analysis tools with varying functionality and capabilities.

Our primary forensics platform will be the Ubuntu SIFT Workstation, customized specifically for the needs of the memory forensics ninja examiner. Included in this SIFT virtual machine is the latest publicly released version of the Volatility Framework, a cutting edge memory parsing toolset, Bulk Extractor, a stream based parsing workhorse, and Page_Brute, a tool specializing in page file parsing with YARA rules inclusion.

In addition, we are distributing a Windows 8.1 virtual machine that is stocked with many open-source/free forensics tools to include Mandiant's latest version of **Redline**, a GUI based live audit and memory image parsing tool, **Windbg**, the standalone Windows **Volatility** and **Rekall** frameworks and **winpmem** acquisition tool.



This page intentionally left blank.



For this course we will be using a number of memory images. Because these files are so large, we are distributing them to you pre-installed on two versions of the SIFT, SIFT 3.0 Ubuntu and Windows8.1 SIFT. The virtual machine files have been compressed into two separate ZIP files and stored on your FOR526 USB under the "**`FOR526_vms`**" directory.

To install the SIFTs, copy the ZIP files from the USB to your computer. Decompress the files on your computer. It is recommended that you use 7zip, or Keka for MAC users, to unzip these files as the native Windows extractor may fail on large files. Move the decompressed folders to the folder where you store your virtual machines.

Before you start the VMs for the first time, don't forget to adjust the number of processor cores allocated to each VM. Some of the exercises will run much faster with more cores! If you have questions about how to do that, ask your instructor.

SIFT 3.0 Ubuntu VM

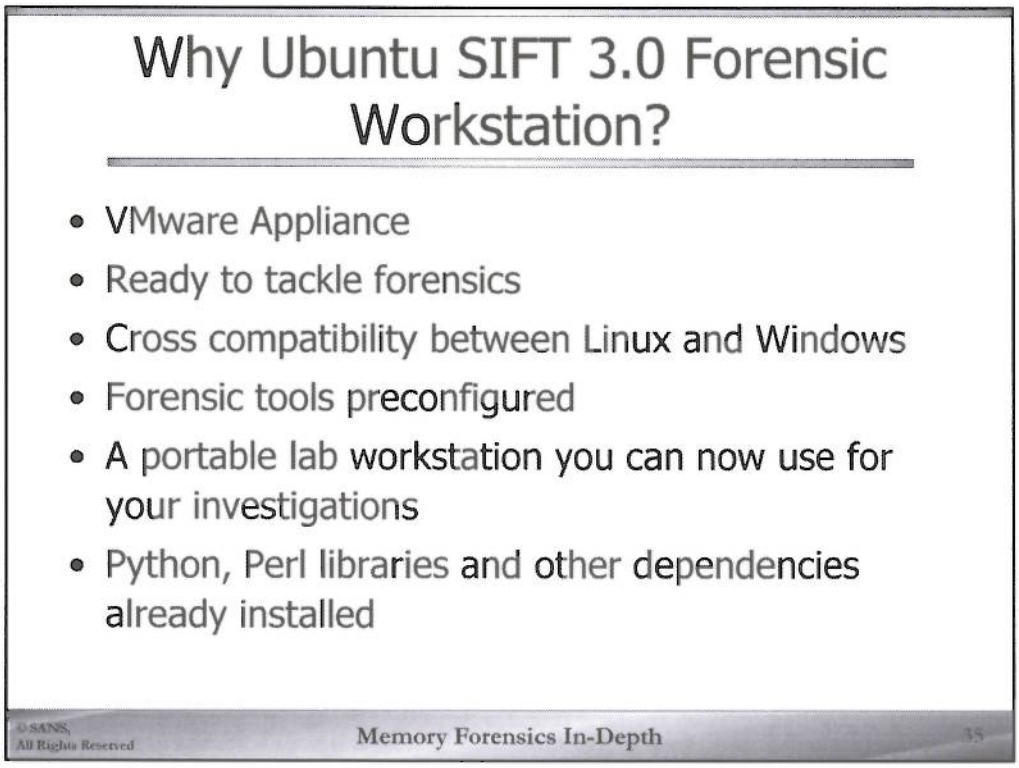
When the SIFT 3.0 Ubuntu boots, login using the credentials below:
Login "**`sansforensics`**"
Password "**`forensics`**"

You should find the memory images in the /cases directory in the **bootcamp.rar** file. Decompress the image files by running the following command:

$cd /cases
$ rar x bootcamp.rar

SIFT Windows8.1 VM

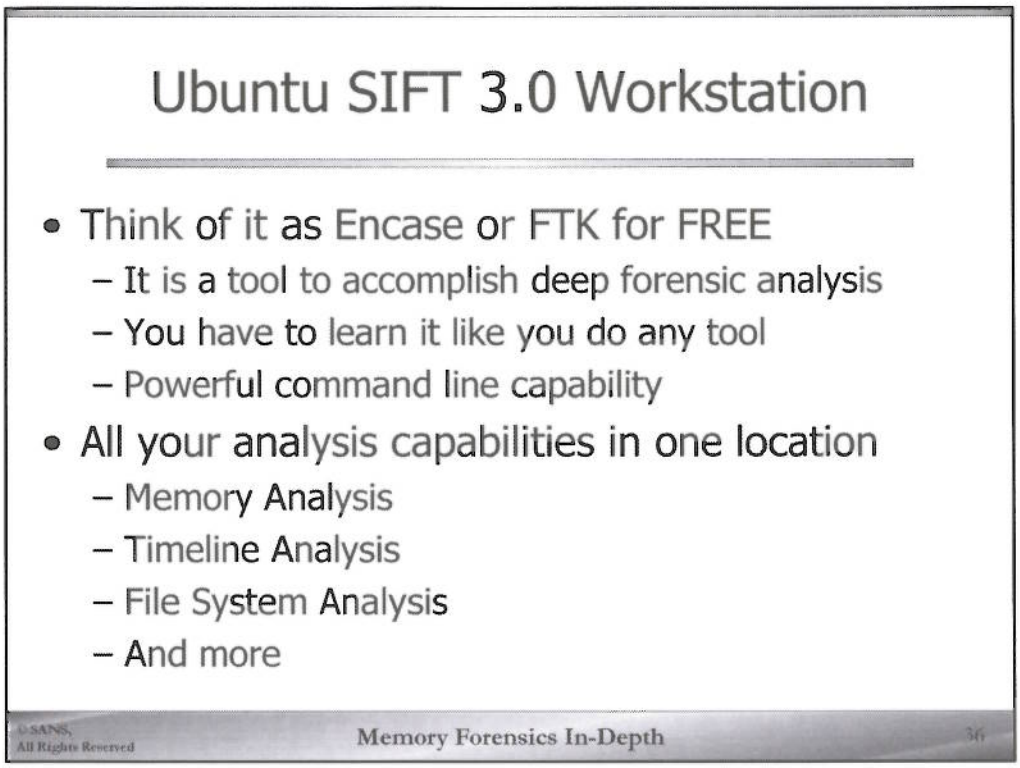
Follow the instructions in Exercise1 in order to properly boot up and activate your Windows 8.1 virtual machine. Once you have set up your "dummy" account, you will log off and log in using the "sansforensics" user account.



Windows forensic tools have a lot of capabilities, but in many cases, you need something with a little more versatility and compatibility. By default, the SIFT workstation comes pre-installed with many tools that allow you to perform in-depth forensic analysis of multiple operating system types.

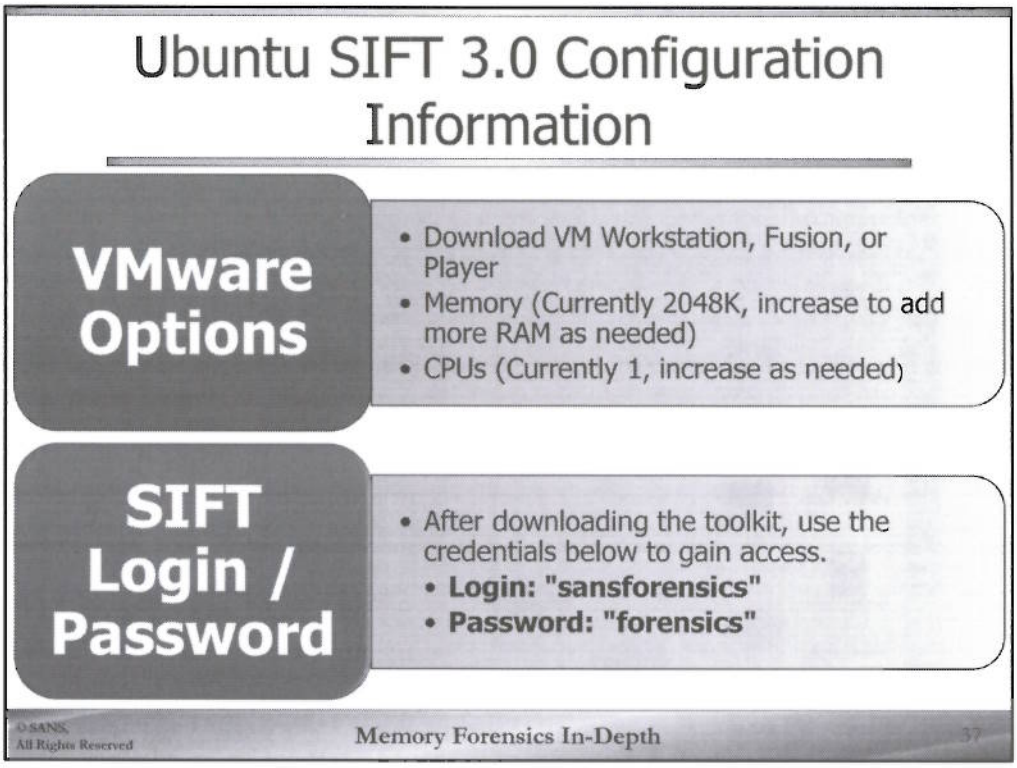
Most of your forensic examination beyond evidence collection can be performed using your SIFT workstation. If you work mainly in a Windows environment, you should still consider using SIFT to examine your compromised Windows platforms. Most seasoned investigators use both Linux and Windows at the same time to ensure evidence is not missed. You will find your job much easier to perform once you get a feel for the powerful forensic options provided to you by a Linux workstation.

Again, it should be noted, that while you may start out in the Windows side collecting your evidence using FTK Imager, winpmem, pstools, regedit etc., you could then examine your collected evidence on your Linux system, or use your Linux system as your anchor to ensure the evidence doesn't change while during the exam.



An international team of forensics experts, led by SANS Faculty Fellow Rob Lee, created the SANS Investigative Forensic Toolkit (SIFT) Workstation and made it available to the whole community as a public service. The free SIFT toolkit, that can match any modern forensic tool suite, is also featured in SANS' Advanced Computer Forensic Analysis and Incident Response course (FOR 508). It demonstrates that advanced investigations and responding to intrusions can be accomplished using cutting-edge open-source tools that are freely available and frequently updated.

The SIFT Workstation is a VMware appliance, pre-configured with the necessary tools to perform detailed digital forensic examination in a variety of settings. It is compatible with Expert Witness Format (E01), Advanced Forensic Format (AFF), and raw (dd) evidence formats. The brand new version has been completely rebuilt on an Ubuntu base with many new capabilities and tools, such as log2timeline, that provides a timeline that can be of enormous value to investigators.



Host Machine Connectivity

Enable SHARED FOLDERS

`VM -> SETTINGS -> OPTIONS -> Shared Folders -> Always Enabled` (Check)

Access to Host System Found on Desktop

`VMware-Shared-Drive`

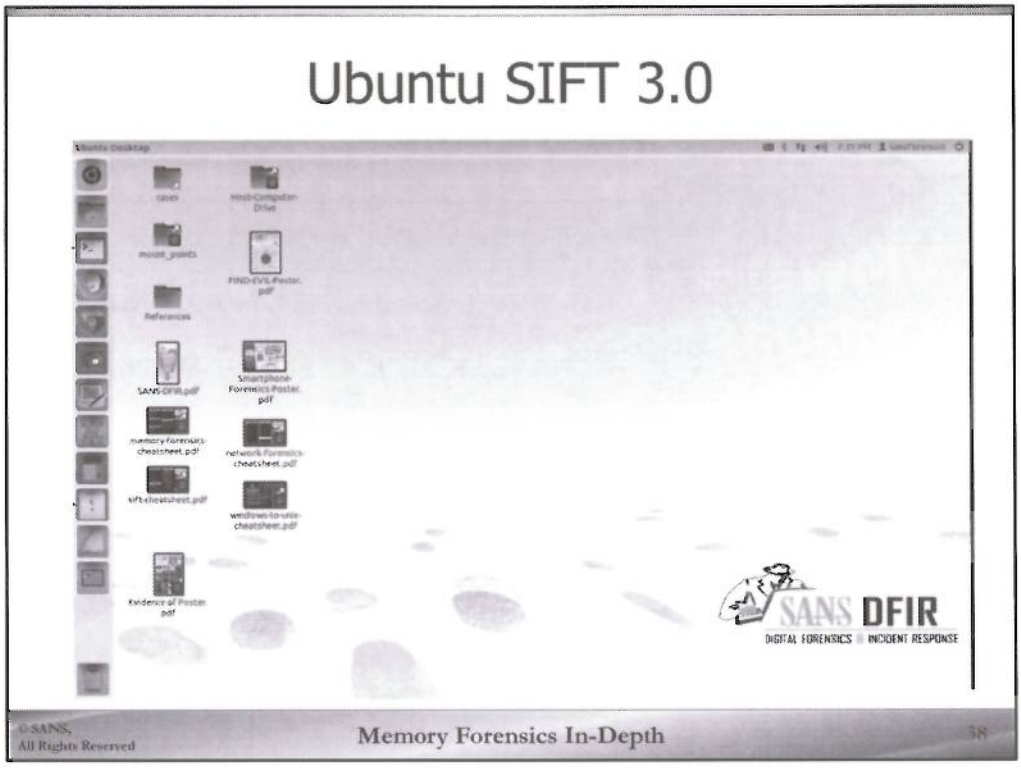
Access from a Windows Machine

Filesystem Shares \\SIFTWORKSTATION

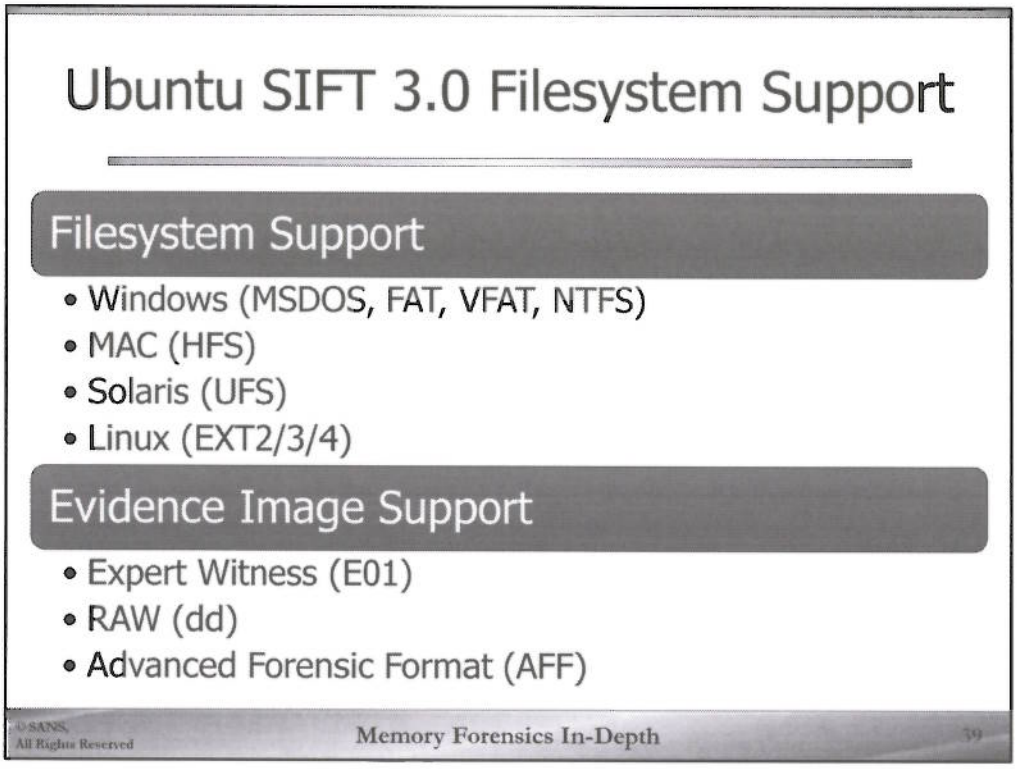
or use ifconfig and connect to eth0 IP Address listed

/mnt - Mount point for read-only examination of digital forensic evidence

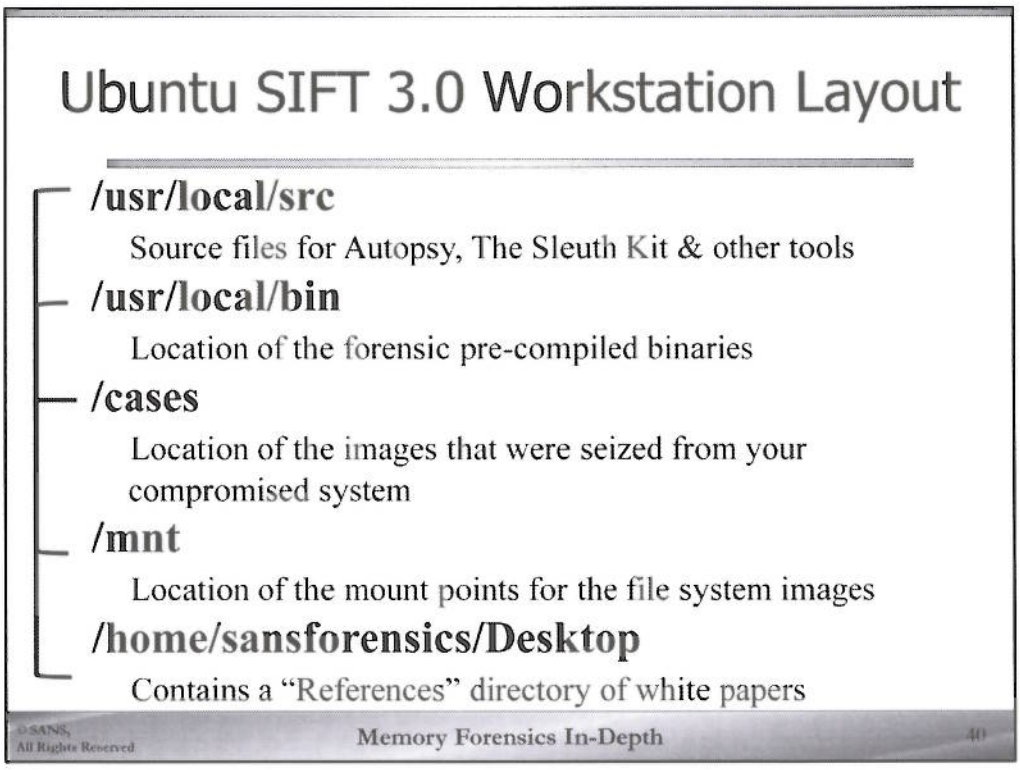
/cases - Directory to store evidence



The newest version of the Ubuntu SIFT is installed on Ubuntu 12.04.4 LTS. Check out the SANS DFIR poster files, cheatsheets and white papers in the References directory located on **sansforensics** Desktop for deeper research on topics covered in this course.



Linux has the natural ability, unlike Windows, to have support for many file systems. This enables the investigator to examine any type of file system encountered. An example of how this may benefit the memory forensics examiner is when EnCase is used to collect a physical memory image from a target system. The output of the resultant memory dump may be .E01 and many analysis tools operate more efficiently with a raw memory image. With the raw image emulation tool, ewfmount included in the Ubuntu SIFT, we can present our image as raw to our memory analysis tools for parsing. Due to this expansive support for various image and filesystem types, the Linux-based operating system is typically a good investigator's tool of choice if no other option is available.

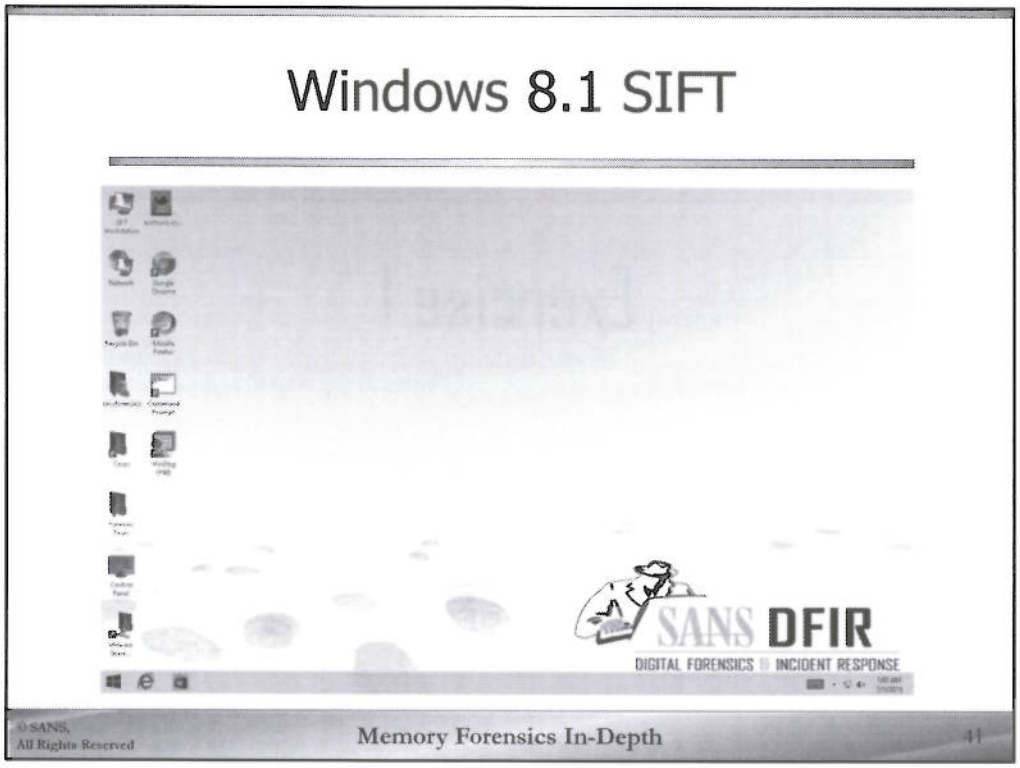


These are the common directories that a new user of the SIFT should be familiar with while using the workstation. The /usr/local/src/ and /usr/local/bin/ directories are used for source and executable files respectively.

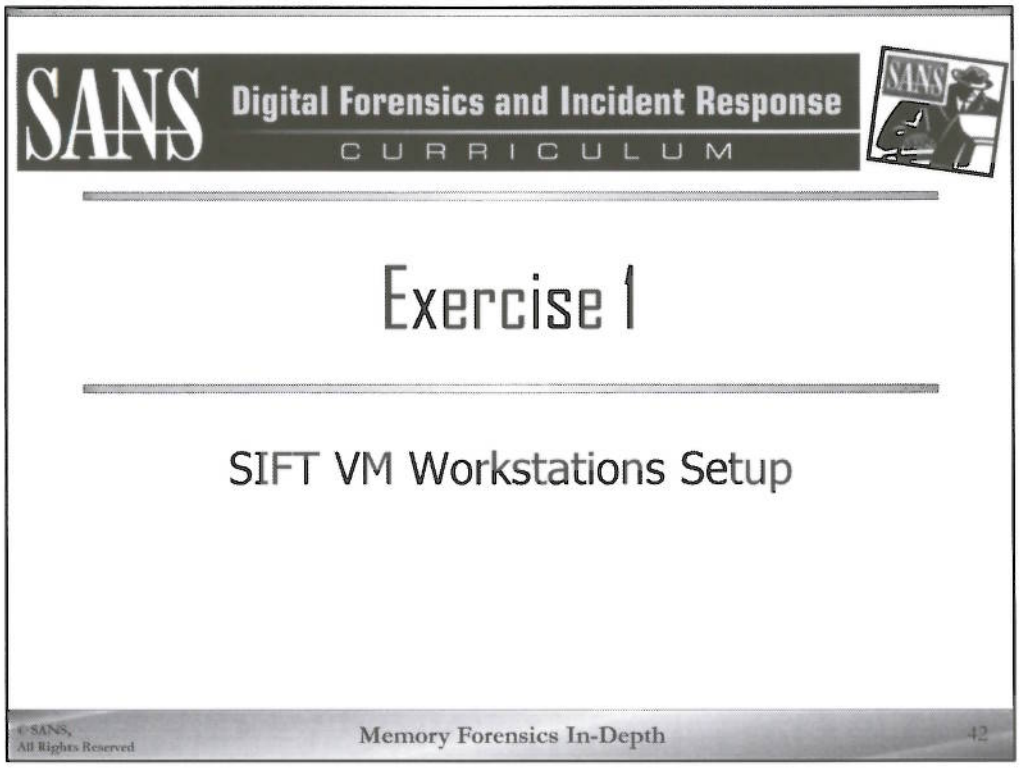
The /cases directory is where your case files will be located during your investigation. Typically, the /cases/YYYYMMDD-casename format is useful to identify data surrounding a specific case. I typically create sub-directories based on the type of data I have collected. (e.g., Image, Carved Data, Live Response Output)

The /cases directory is a perfect place to create a case for each of the images for every case you encounter. For this course, we will be examining several sets of images. All images for this course will be in the **`/cases/`** directory. Each subdirectory will be the name of your exercise or case (i.e., **`exercise1`**, **`netwars`**)

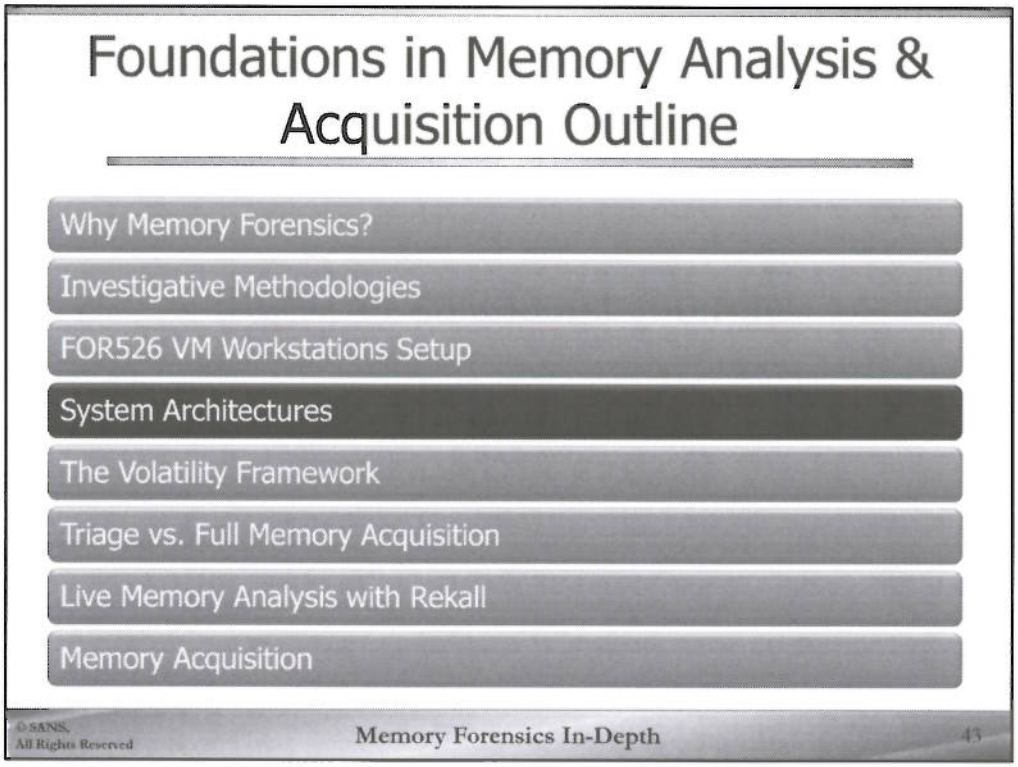
The /mnt directory is the location in the SIFT Workstation where you will mount your image files so you can browse the directory structure of the disk image you have collected of your victim system.



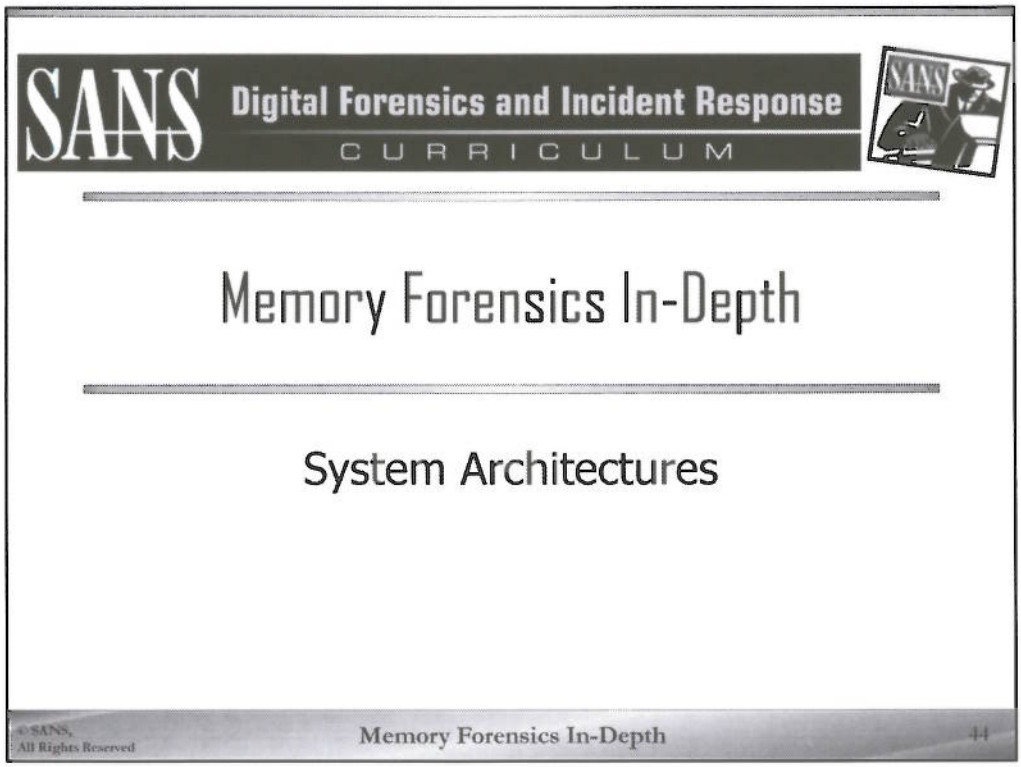
This Windows 8.1 SIFT is loaded with open source and free forensics tools ready for use. Two directories to check for these forensic tools are "C:\Forensic Program Files" and "C:\Users\sansforensics\Desktop\Forensics Tools", but others can be found in "C:\Program Files". The SysInternals suite is one of the Windows-specific tools loaded into the Windows 8.1 virtual machine at "C:\Forensic Program Files\SysInternalsSuite:. We will be using several of the SysInternals tools throughout the course. Other notable tools included here are FTK Imager, NirSoft Browser parsers and various Windows artifact parsers.



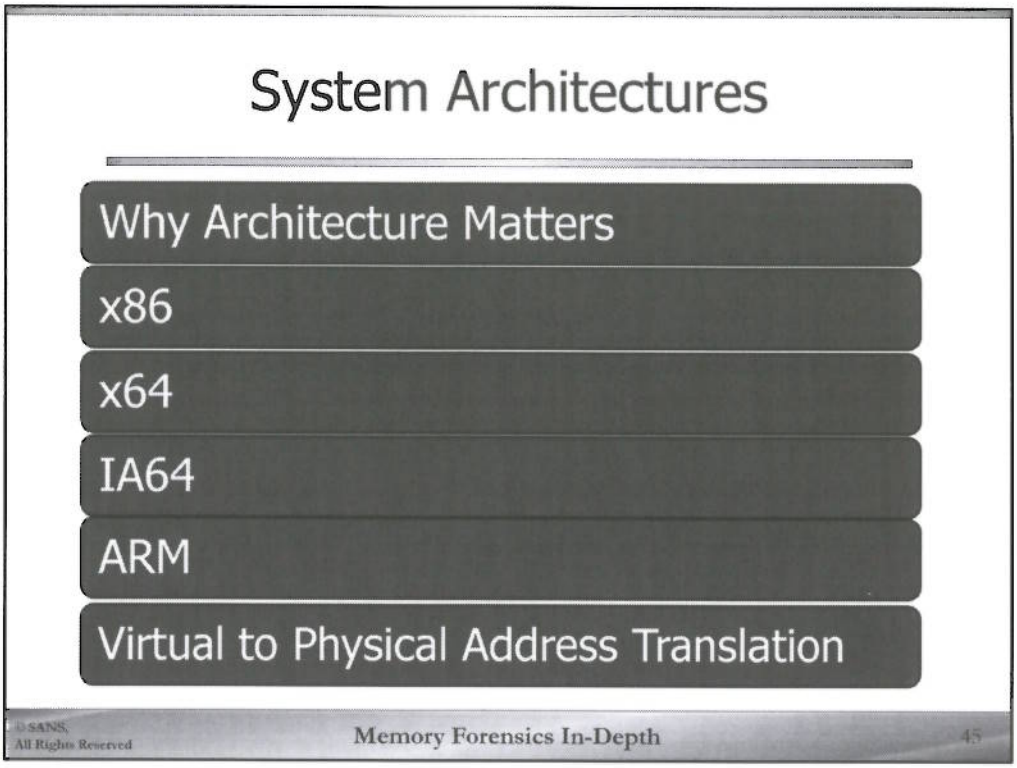
This page intentionally left blank.



This page intentionally left blank.



This page intentionally left blank.



In this section we will introduce the three architectures supported by Microsoft Windows. This course will be focusing largely on x86, but includes x64 system architectures as well. With 64-bit systems becoming increasingly more common due to rising memory demands of modern applications, it is imperative that digital forensics examiners are familiar with analyzing memory images from these two most prevalent architectures. We also briefly mention IA64, which although still supported by Microsoft, is waning in popularity.

If you have any questions about an architecture, you can consult the manuals. Both AMD and Intel publish free guides to their chips. These guides, which come in multivolume sets, are insanely detailed. Both companies want operating system developers to write bug-free programs for their products. As such they have done everything possible to encourage developers to write for their platforms.

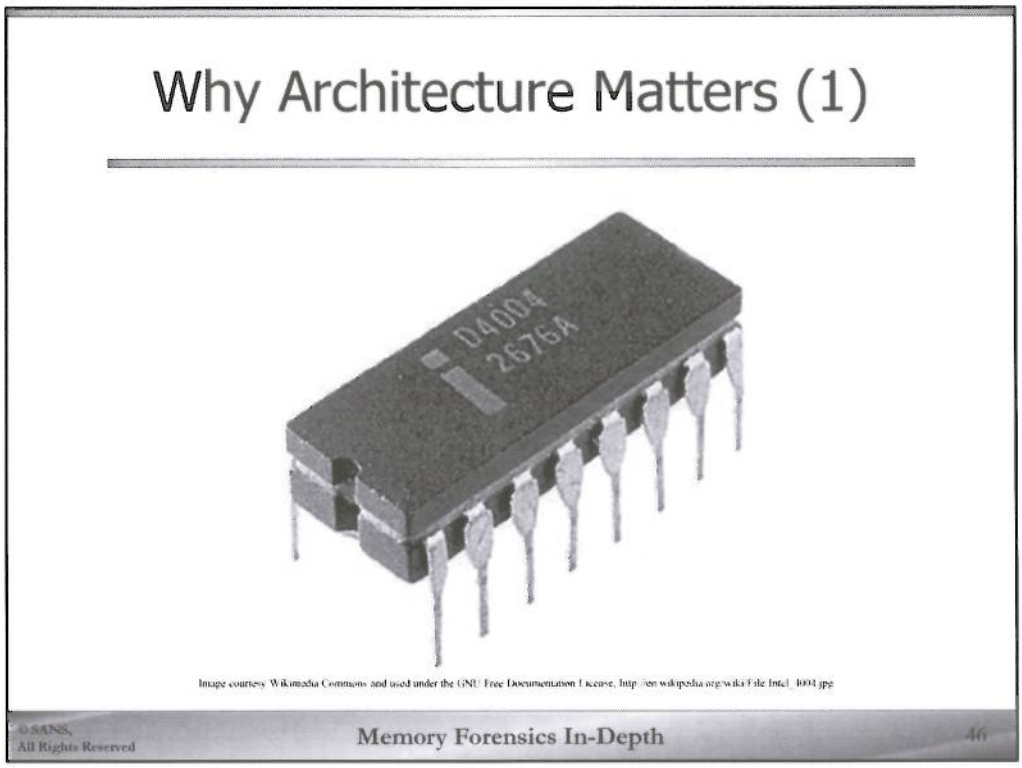
Intel Architectures Software Developer Manuals

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

AMD

http://developer.amd.com/resources/documentation-library/

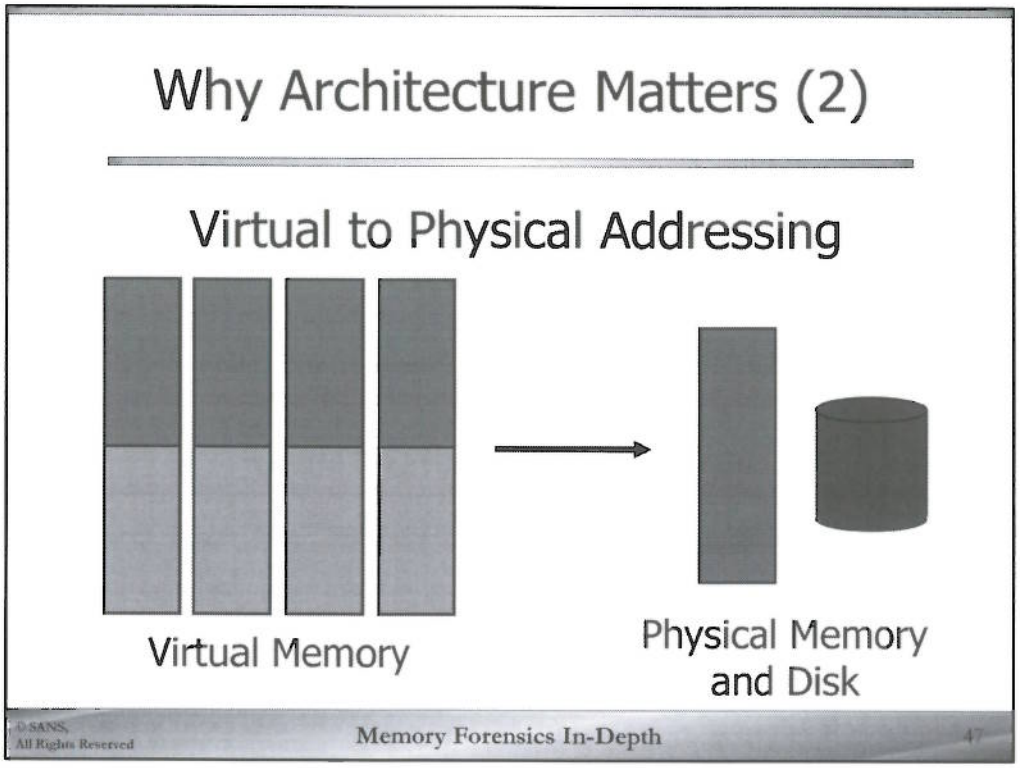
The manuals are updated frequently, even now. Features are added to processors, bugs are corrected (both on the chips and in the documentation!) and confusing items are clarified.



The processor architecture describes the layout and operation of the Central Processing Unit (CPU) —it's the heart of the computer. It dictates how every component must interact with the CPU. How memory is addressed and modified, and which instructions are legal and how they are called is based on system architecture. Every component of the system must speak the CPU's language, which means they must be familiar with the CPU's architecture.

Because so much of the computer's operation depends on the CPU and thus the architecture, architectures almost never remove support for features. The processor architecture is probably the oldest part of your computer. It contains obtuse specifications which most users never see. Except you, of course.

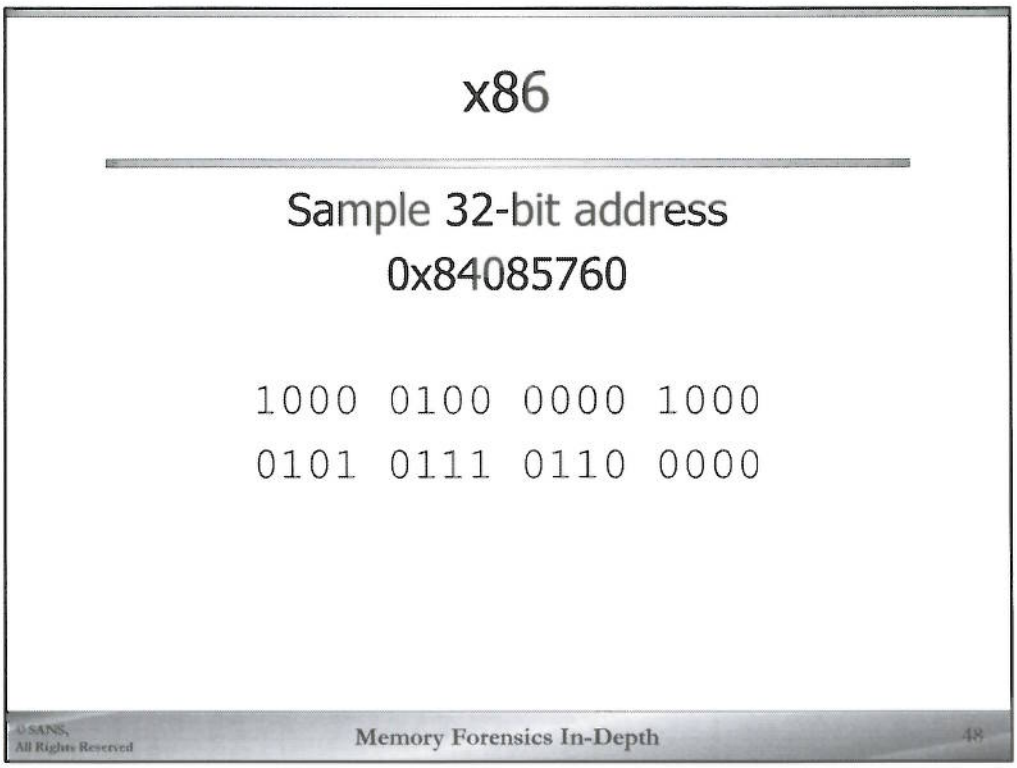
The picture above shows the Intel 4004 processor, the great grandfather of the x86 architecture. The x86 architecture was originally developed in the 1970s. Although many things have been added on since then, almost nothing has been removed. This accumulation of cruft and hacks, while preserving backwards compatibility, means that an Intel 8088 assembly program, written in 1981, would run on today's top of the line x86 processor without *any* modifications.



Memory Management and address translation are dependent upon system architecture. How active processes address memory locations depends on the size, in bits, of the system bus, 32 bit or 64 bit. Of course, processes do not address physical memory directly, but make use of a memory manager to translate their virtual address ranges to physical locations in RAM. Essentially, when an application accesses memory using a virtual address, the operating system translates the virtual address to the physical memory address where the requested data actually resides. Not only does the operating system decide where in physical memory to place process data, it may also decide to move process data out of physical memory entirely. The paging file is the portion of the hard drive that is set aside to act as additional virtual memory storage.

There are numerous benefits to this virtual to physical translation process. Virtual memory is an abstraction between the operating system and the physical system. Physical memory is often in a fragmented state and virtual memory allows the operating system to present a consistent, contiguous view of addressing space to each process. The application programmers don't need to worry about how much RAM is in the system, or how it's allocated, or really anything to do with the memory outside of their process.

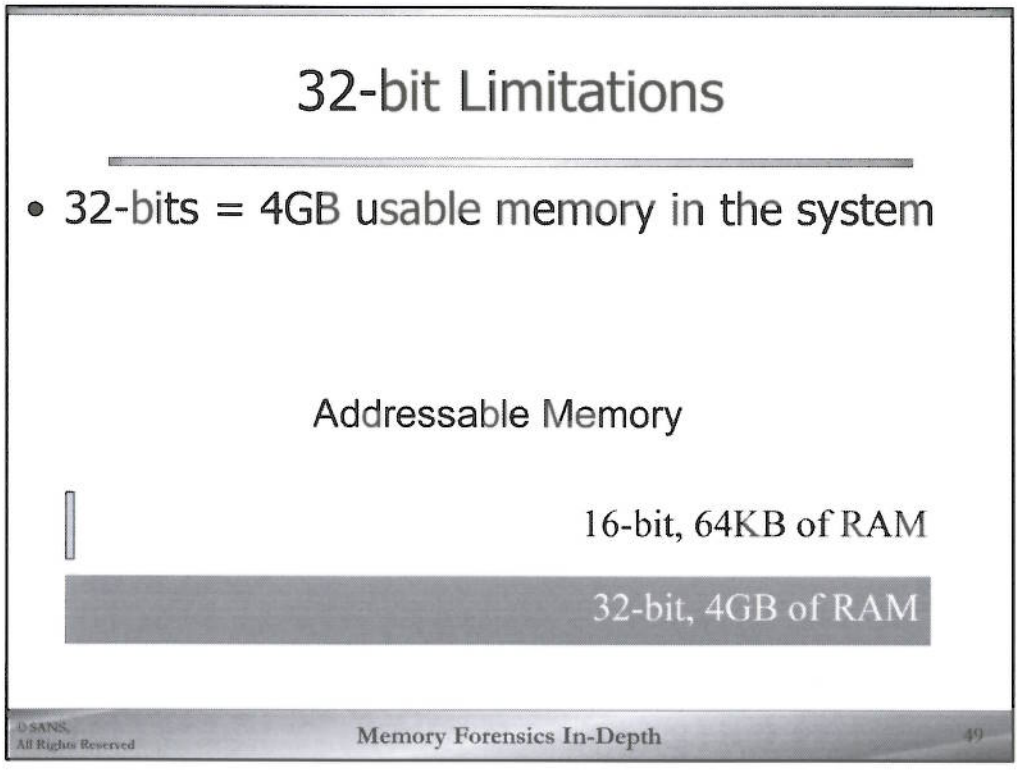
In addition, having a memory manager track the use of reserved physical memory prevents more than one process writing to the same frame. And, the greatest benefit to virtual addressing is the efficiency gain realized by physical memory only being allocated as needed. Processes utilize contiguous virtual address ranges with the memory manager making the backend decisions about whether data should be in physical memory, sent to the paging file or left on disk and loaded on-demand.



The x86 architecture is the most common 32-bit architecture in use today. Chips which support x86 are made by many vendors, but the standard is set and maintained by Intel.

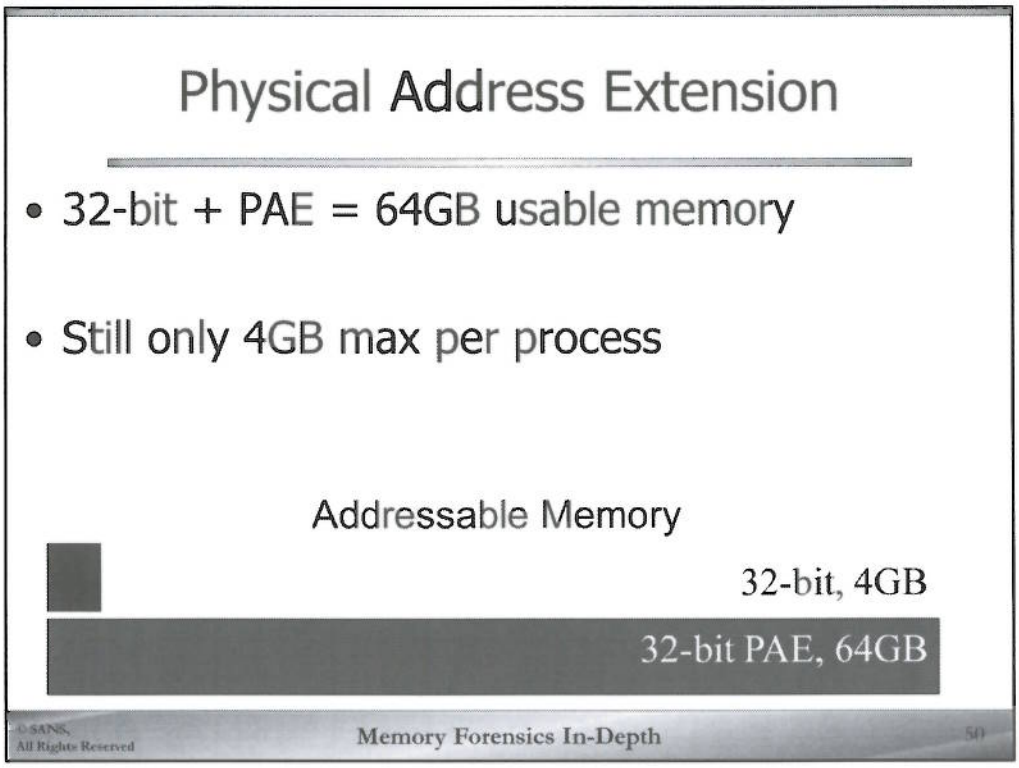
The primary feature of x86, for our purposes, is the 32-bit addresses (and hence 32-bit architecture). Thirty two bit addresses are eight hex digits, or 32 binary digits, as shown above.

As noted, the architecture still has backwards compatibility for 8-bit and 16-bit architectures too. On boot, the processor uses the 16-bit mode to start the operating system. This means that when the computer is turned on, it can only use 16-bit addresses and accept 16-bit commands. The boot sector code is thus limited to addressing 2^16-1 bytes of memory, or just 64KB. The operating system, as part of its boot sequence, must manually tell the processor to switch to 32-bit mode, which allows for 32-bit addressing and 32-bit instructions.



Using 32-bit addresses means the system can use up to 4GB of RAM. 2^32 = about four billion, so 4GB of RAM. You can boot a 32-bit machine with more RAM in the system, but it only uses 4GB. The extra memory will be ignored.

The bar at the bottom of this picture represents the size of the 16-bit and 32-bit address spaces. The 32-bit address space is 65,000 times larger than the 16-bit address space! Over the next few slides we will display the sizes of other address spaces in comparison to the 4GB bar. The 16-bit address space bar will be too small to see on the next slide.

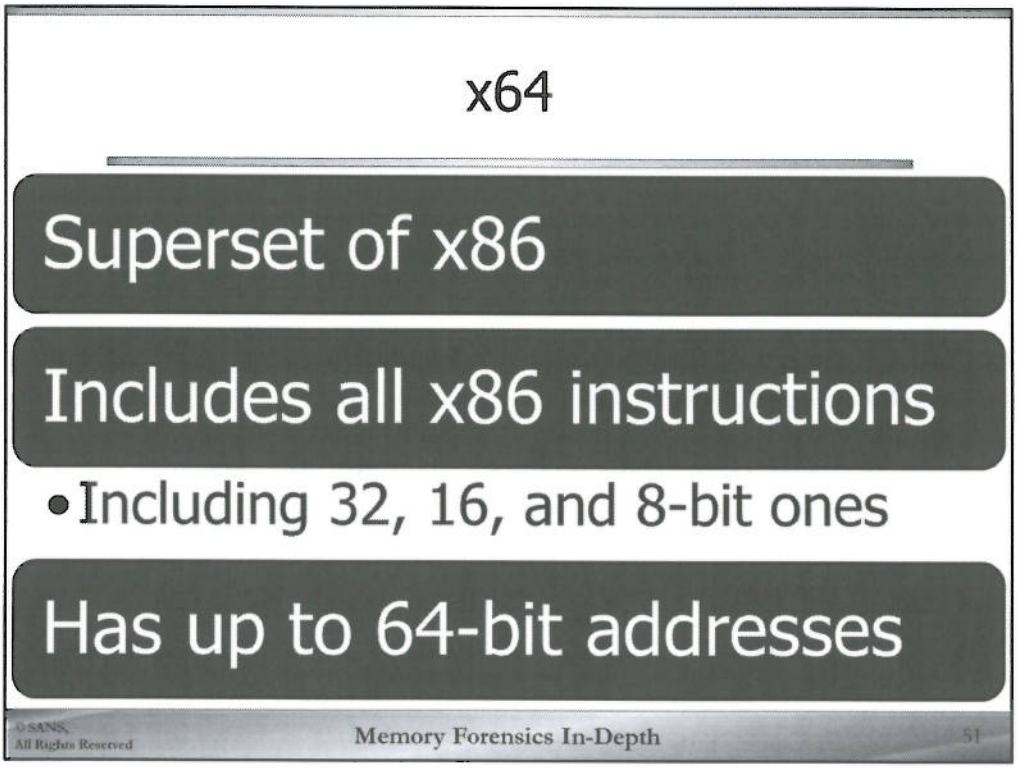


Physical Address Extension (PAE), is a processor mode which extends the addressable memory limit to 64GB. That is, using PAE, a 32-bit system can support a total of 64GB of RAM. It works through the use of an "architectural patch" by expanding the size of structures related to translating virtual address into physical addresses.

The picture above gives a roughly-to-scale representation of the size of addressable memory between a system not running PAE and a system running PAE.

Even with PAE enabled, however, each process is still limited to 4GB of RAM. Virtual addresses are still limited to 32-bits, so each process can only address 4GB of memory. Expanding the size of virtual addresses requires either an advanced trick like Address Windowing Extensions, or expanding the width of the addresses.

PAE was created in 1995 by Intel. It was first supported by Windows Server 2000. The first home operating system support was in Windows XP, and it has been a part of Microsoft operating systems since. These days it's usually enabled by default on 32-bit systems. You have probably been using PAE for some time now. It doesn't affect your day to day experience of the operating system, but you will be able to access more RAM.

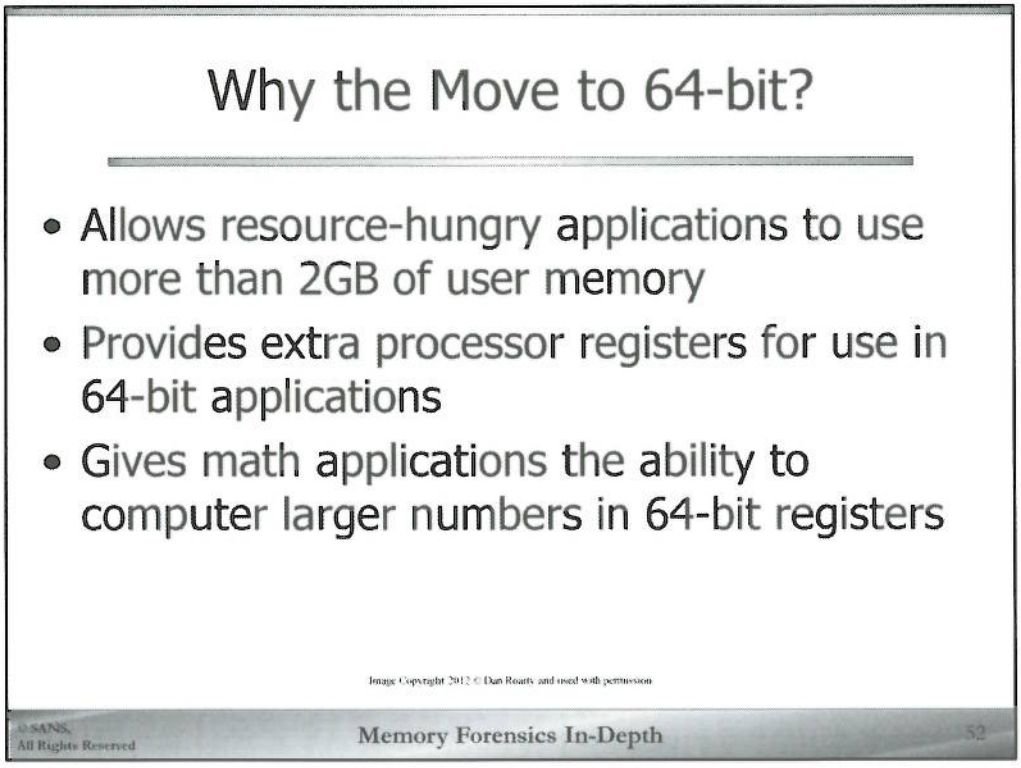


x64 is a 64-bit architecture developed by AMD around the turn of the century. It goes by several names, including x64, x86_64, and AMD64. Previously it's been called IA-32e, EM64T, and Intel64, but those names aren't used anymore. AMD still holds the license to x64 and licenses it to Intel and other chip makers.

The x64 architecture is a superset of x86. That is, the x64 instruction set contains all of the same instructions as the x86 instruction set did, and more. Running x86 programs on x64 systems is a complicated matter, however. When an x86 or x64 processor is powered on, it starts out in "real mode", where it can only address a very limited amount of memory and execute 16-bit instructions. The processor has to be manually moved into "protected mode" where it can run 32-bit instructions. X64 chips, however, have another mode, "long mode," in which they can run 64-bit instructions.

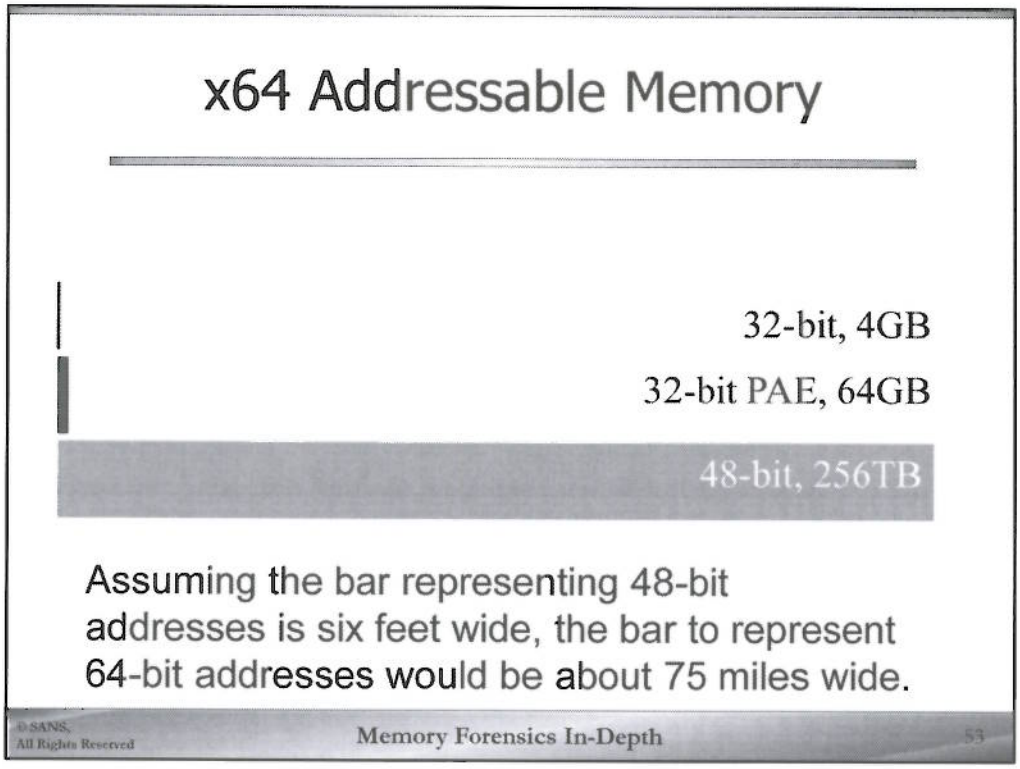
The upshot of all this is that an x64 processor can run 32-bit operating systems which use protected mode. The OS simply sets the processor to use protected mode. The OS never realizes there *is* another mode on the chip. In fact, a real-mode, or 16-bit operating system could use an x64 chip. It would just never move the processor out of real mode! Thanks to backwards compatibility, today's modern processors can run an operating system from 30 years ago.

The power of the 64-bit architecture isn't just 64-bit instructions, but also being able to use 64-bit addresses. The next slide shows the scale of memory which can be used with 64-bits.



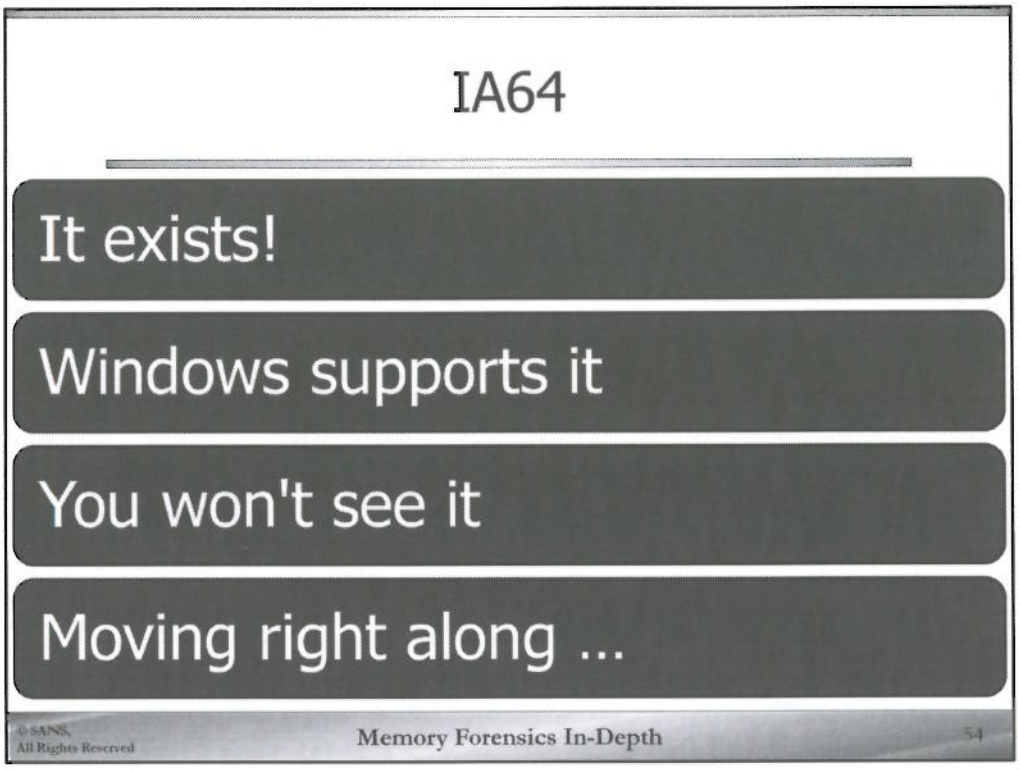
The move to 64 bit processors addresses several problems. While PAE allows the OS to address more than 4GB of main memory, each process can only address 4GB. Because 2GB of this memory is normally reserved for the kernel, this leaves only 2GB for user space. The Windows /3G switch and Linux 4G/4G split attempt to address this, but other solutions were still needed. Some processes simply need more than 2GB of memory. Unless Intel released a 64-bit processor, applications that needed more memory would continue to operate on other architectures (e.g. SPARC).

Applications that do not require additional memory can also benefit from 64 bit processors. The 64 bit processor adds a number of additional registers that can be used for fast access and computations on data. Speaking of computations, some math operations must use numbers larger than those that can be represented using only 32 bits. These applications also benefit from 64 bit processors. Additionally, Intel added relative instruction pointer addressing modes in 64 bit that make position independent code (PIC) easier to write and compile. This results in better performance and fewer errors (better for security).



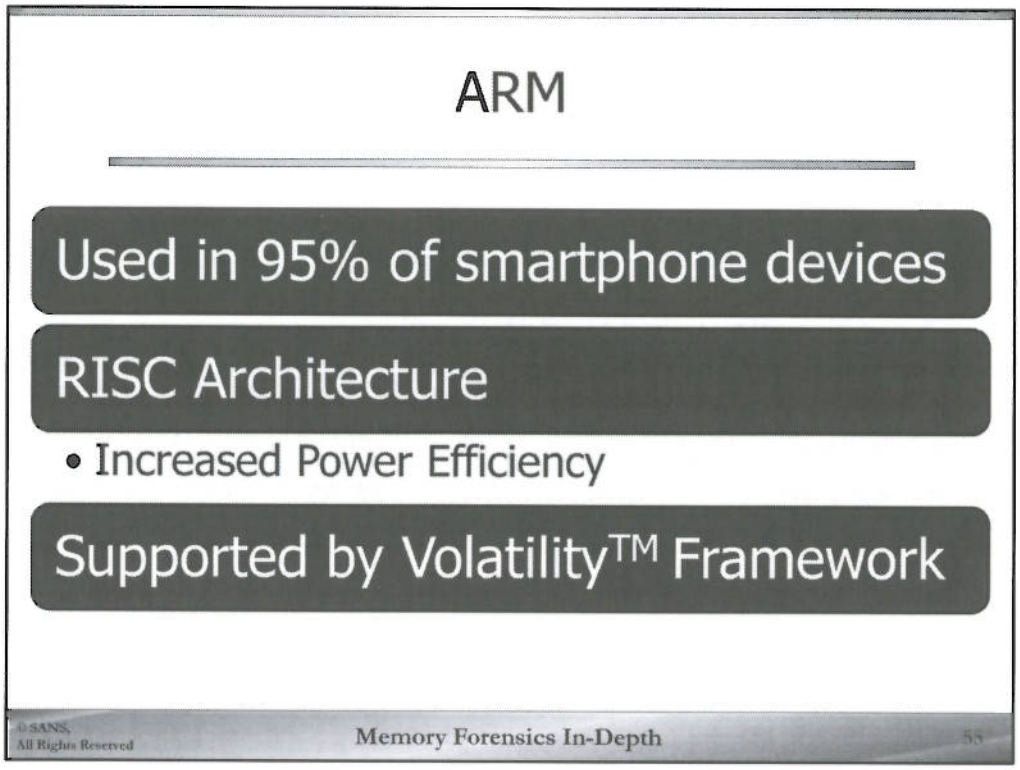
Far and away, however, the advantage of the 64-bit architecture is the enormous address space. Sixty four bit addressing means theoretically 16EB of physical memory. 1EB, or Exabyte, is 1,024 TB, or more than a million GB.

In versions of Windows prior to 8.1, x64 system architecture limits the address space to 48-bit addresses. This allows for greater physical memory usage, making lookups both faster and easier. A 48-bit address space can use 256TB of RAM. With Windows 8.1, the most significant advantage in the additional bits available for addressing is the increase in the number of bits used in entropy of ASLR (Address Space Layout Randomization). This enhances security on a Windows system by randomizing the base memory address of an application and other structures when initially loaded This implementation makes it hard to attackers to guess where they are located, increasing the difficulty for malware to make successful use of Return Oriented Programming.

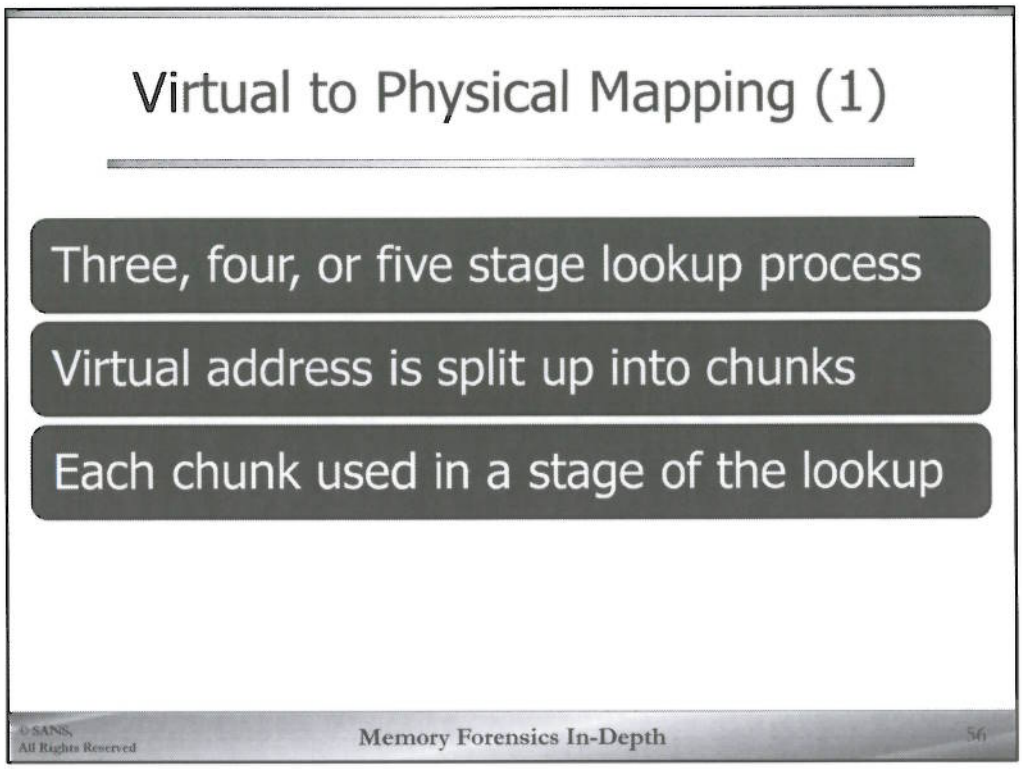


Microsoft Windows supports another 64-bit architecture called IA64. This architecture is made by Intel, and is also known as Intel Itanium. It is most commonly seen in high-end enterprise servers, but its use is fading in popularity. Microsoft Windows 2003 and 2008 support the Intel Itanium architecture, but Windows Server 2008 R2 will be the last version of Windows Server to do so, with extended support offered until July 10, 2018. For the purposes of Windows forensics, Itanium is on the way out. We mention it in this course so that you can recognize it, but will not be spending any time analyzing it.

For further reference, http://www.intel.com/content/www/us/en/processors/itanium/itanium-processor.html



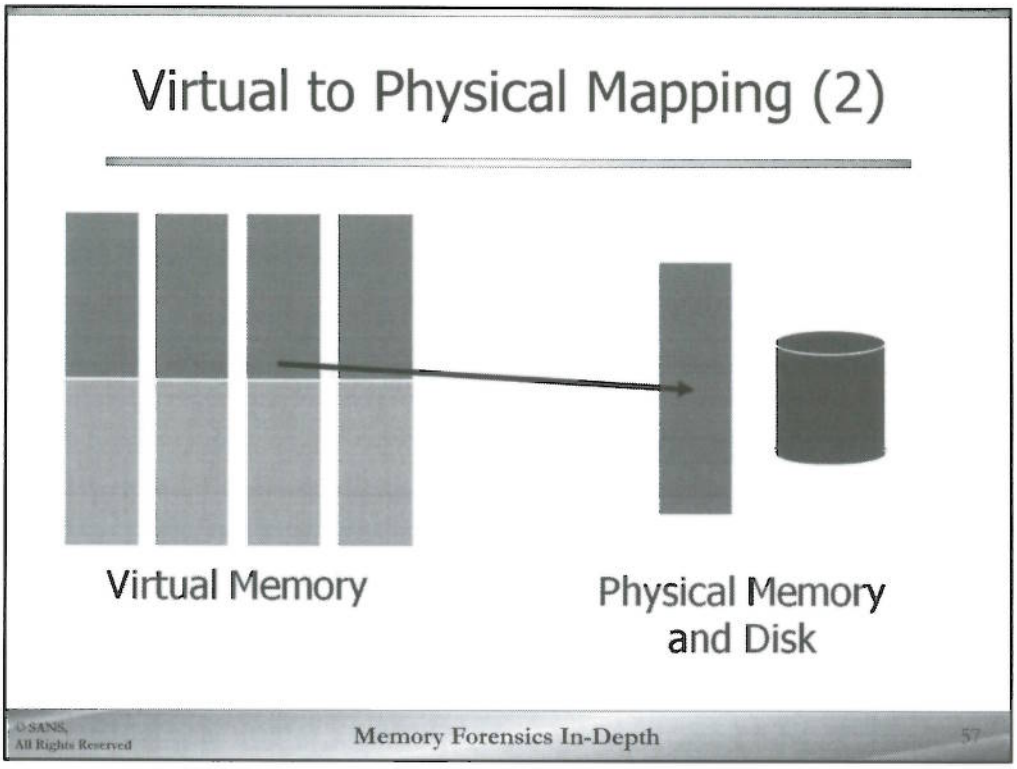
ARM architecture is popular in mobile devices and smartphones in particular due its increased processing and power efficiency. It is a RISC, Reduced Instruction Set Computing, platform, enabling faster instruction execution in contrast to the other CISC, Complex Instruction Set Computing, architectures mentioned in this section. The Volatility™ framework, one of the tools we will be using in this class for memory analysis, now supports 32-bit ARM address space with its recent addition of Android plug-ins. In the future, forensic analysis of ARM memory images will be increasing more important, as mobile devices, to include smartphones, become ubiquitous in forensic investigations.



We have discussed how system architecture affects virtual to physical address translation. Let's take a deeper look at this translation process to achieve an understanding of the complexity of what our memory parsing tools make look exceedingly easy. The Virtual to Physical mapping process, or vtop (pronounced vee-too-pee) translates the virtual addresses programs use into location in physical memory (i.e. RAM chips) where data is stored. Depending on the architecture and operating system configuration, vtop is a three, four, or five stage process. Some of the mechanisms for doing vtop are in the processor, while some are in the operating system.

Let's start with the processor structures. Although these data structures are read and manipulated by the operating system, they refer to values which the processor maintains.

We will discuss the differences between architectures in the vtop process in the upcoming slides. Now, you won't have to do this math as any decent memory forensics program will do it for you. But you should know what's going on under the hood in the same way that you should understand what an NTFS stream is and how they are parsed. You never have to do it manually, but when asked about Alternate Data Streams, you need to be able to explain the former to comprehend the latter.



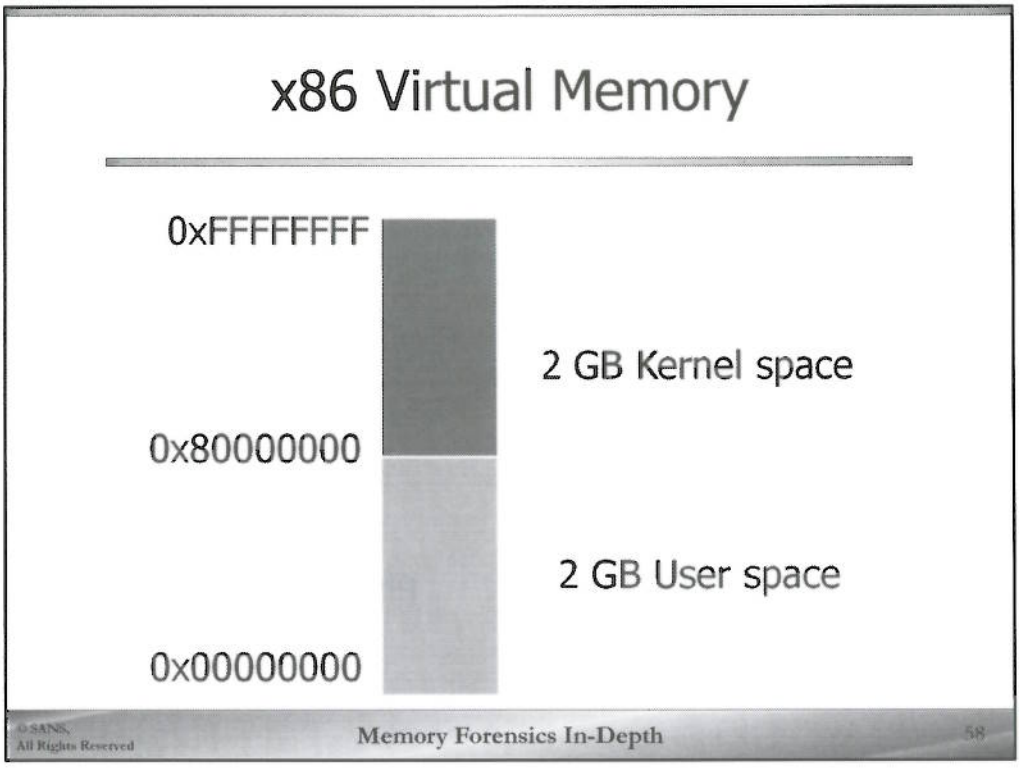
We are going to start with a virtual address and end up with an offset in physical memory. That is, we'll start with the address being used in a process and end up with the location in the memory image.

This example will use 32-bit virtual to physical address translation without PAE. (We'll get to PAE in a few pages, don't worry.)

Of course, the virtual memory has to be mapped to physical memory at some level. This process is called Virtual to Physical mapping, or vtop (pronounced vee-too-pee). This process translates the virtual addresses programs use into location in physical memory (i.e. RAM chips) where data is stored. Depending on the architecture and operating system configuration, vtop is a three, four, or five stage process. Some of the mechanisms for doing vtop are in the processor, while some are in the operating system.

Let's start with the processor structures. Although these data structures are read and manipulated by the operating system, they refer to values which the processor maintains.

We will do an example to demonstrate vtop. Now, you won't have to do this math as any decent memory forensics program will do it for you. But you should know what's going on under the hood. In the same way that you should understand what an NTFS stream is and how they are parsed. You never have to do it manually, but when asked about Alternate Data Streams, you need to be able to explain the former to comprehend the latter.



On 32-bit systems, each process sees its own 4GB address space. This is a virtual address space. Not every system has as much, or as little, as 4GB of RAM. And certainly very few computers would allocate an actual 4GB of physical RAM for each process. But having 4GB of RAM for each process is the view that Microsoft Windows presents to each program. The abstraction saves work for the programmers and allows the operating system to conserve resources.

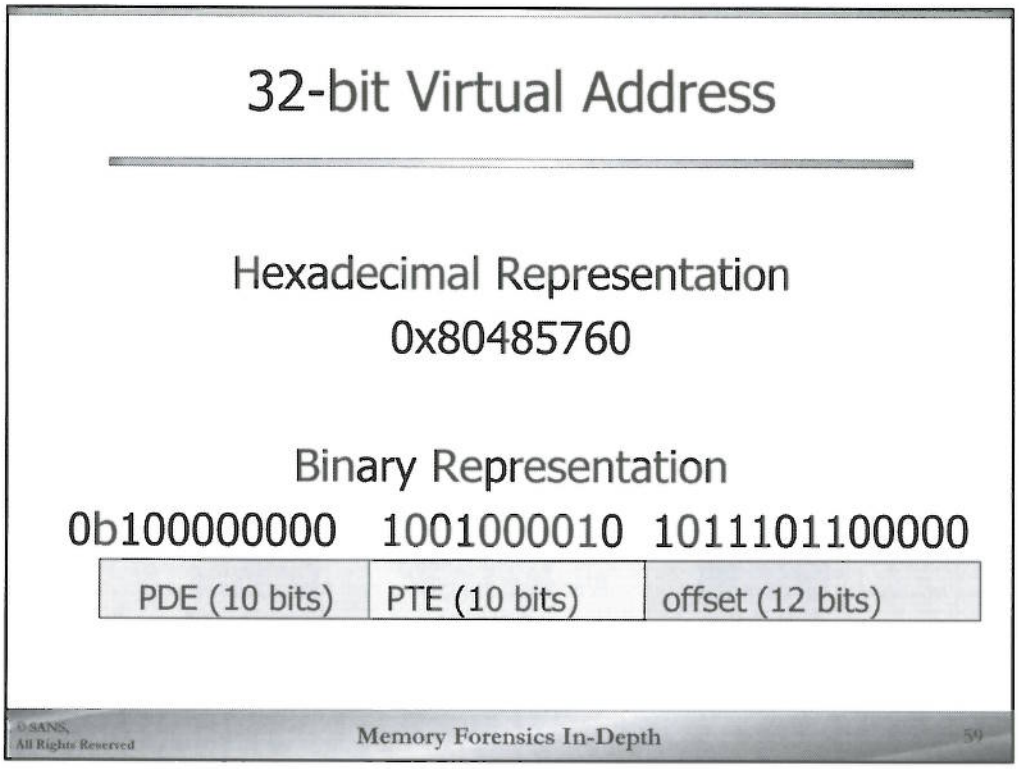
The virtual address space is 4GB on 32-bit systems because that's the number of addresses which can be specified using 32 bits. Each address, like every other number a 32-bit processor can handle, is exactly 32-bits wide. The virtual addresses range from 0 to $2^{\wedge 32}$-1.

Of the 4GB virtual address space, the lower 2GB is reserved for the process memory. This area is also called user space. The upper 2GB is reserved for the operating system and is also called kernel space. Again, this doesn't mean that the system has 2GB of physical RAM which is reserved for either the OS or a process. But it's the view presented to the programs.

On 64-bit systems the sizes of these pieces are different, but the concept is the same.

Note that kernel space begins at offset 0x80000000. The '8' in that address denotes the high bit in the address, bit 31, is set. You can look at a virtual address' high bit and immediately know* if the address is in user space or kernel space.

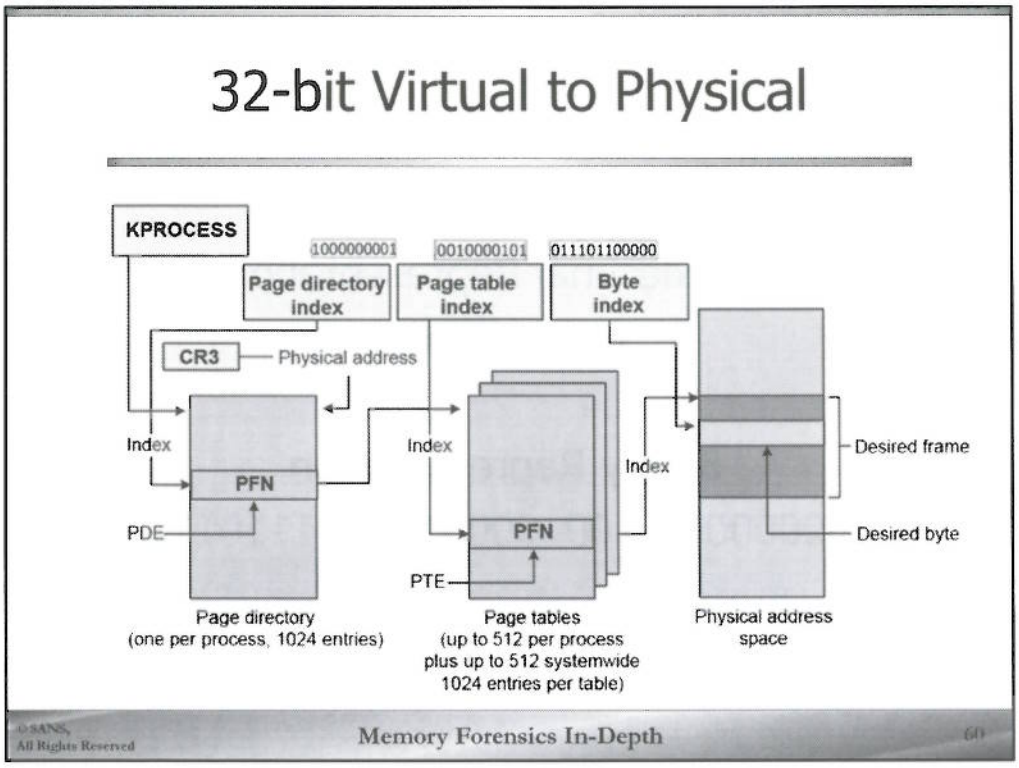
* Knowing, of course, is a strong term. There is a boot option which reserves 3GB for user space and 1GB for the operating system. It was created for processes which need lots of RAM, like database or Microsoft Exchange servers, but isn't common on home systems. You probably won't see it in your cases. On the other hand, it's possible you *could* see it as part of an anti-forensics technique. Most forensics programs assume the standard 2GB divisions of virtual memory. If that assumption is invalidated, it could throw a monkey wrench into your tools!



Here's a hypothetical virtual address, 0x80485760. Note that the high bit is set. Is this address in kernel space or user space? [Answer: kernel space]

The virtual address is going to be divided up into three pieces for the lookups. Those divisions are based on bits, so to save ourselves time later we'll also write out the address in binary.

We are going to divide the original virtual address into three pieces: the PDE index, PTE index, and an offset. Those three values will be used as lookup values during the vtop process.



Here's the full picture, based on the one portrayed in the Windows Internals book [1], representing the translation process of a 32-bit virtual address to an offset in physical memory when PAE is not enabled.

The process starts with the Directory Table Base or DTB. The DTB is stored in a structure called KPROCESS, which we'll talk about later. When the process is active, the DTB is also stored in the CR3 register on the processor.

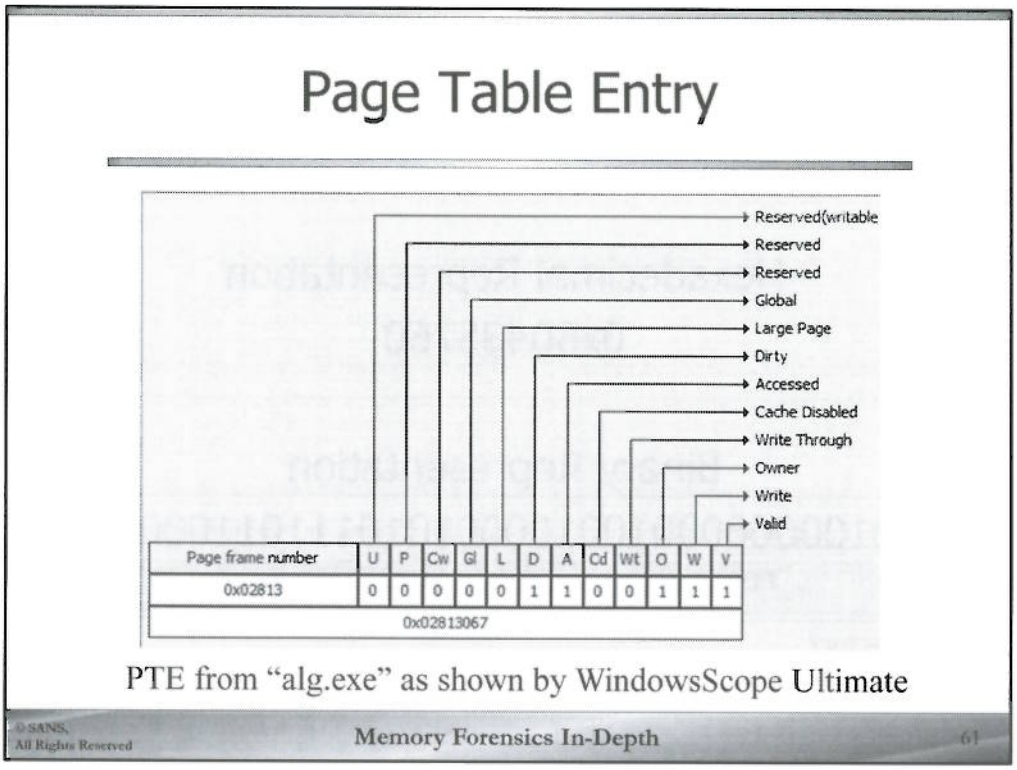
The DTB points to a frame of memory which contains a table of Page Directory Entries (PDE). The DTB is the *physical* offset of this table. (Hence 'frame' of memory, not 'page'.) That is, it denotes where to find the table of PDEs in the memory image. The table contains values which we will need for the next stage of the lookup. We use bits 22-31 of the virtual address as an index into this table.

Combining those bits with the address from CR3, we get a value which contains some flags and the physical offset of the table Page Table Entries or PTEs. We use bits 12-21 of the virtual address as index into this table. The value we get from the PTE table is the frame in physical memory where the data is stored.

The last 12 bits of the virtual address are the offset in this frame.

Again, you don't have to do this math by hand. Don't panic.

[1] Russinovich, M., Solomon, D., and Ionescu, A. Windows Internals: Part 1. Sixth Edition.



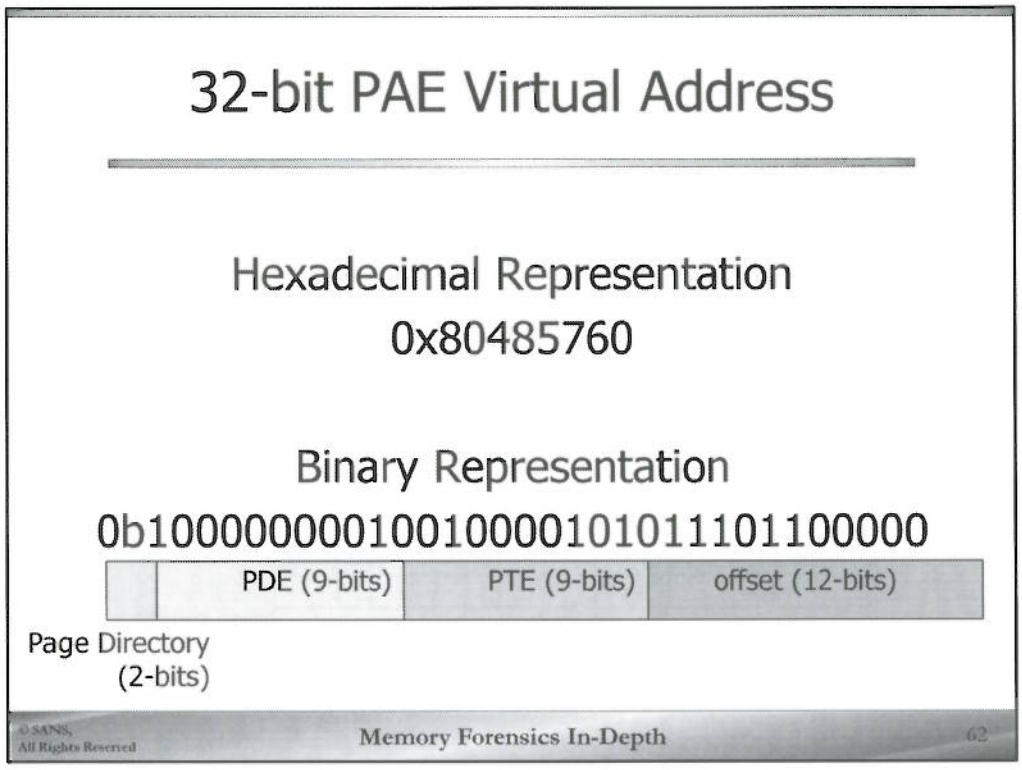
Shown above is a screenshot from the WindowsSCOPE Ultimate tool depicting a page table entry from the process "alg.exe". WindowsSCOPE Ultimate is a GUI memory forensics tool that allows for the analysis of saved memory images as well as the virtual and physical memory of a live system. [1] This tool shows in-depth kernel structures, running processes, dlls and drivers in addition to open files, network sockets and registry keys.

As you can see, the flags allow us to determine that the data referenced by this PTE is "valid", actively loaded into physical memory.

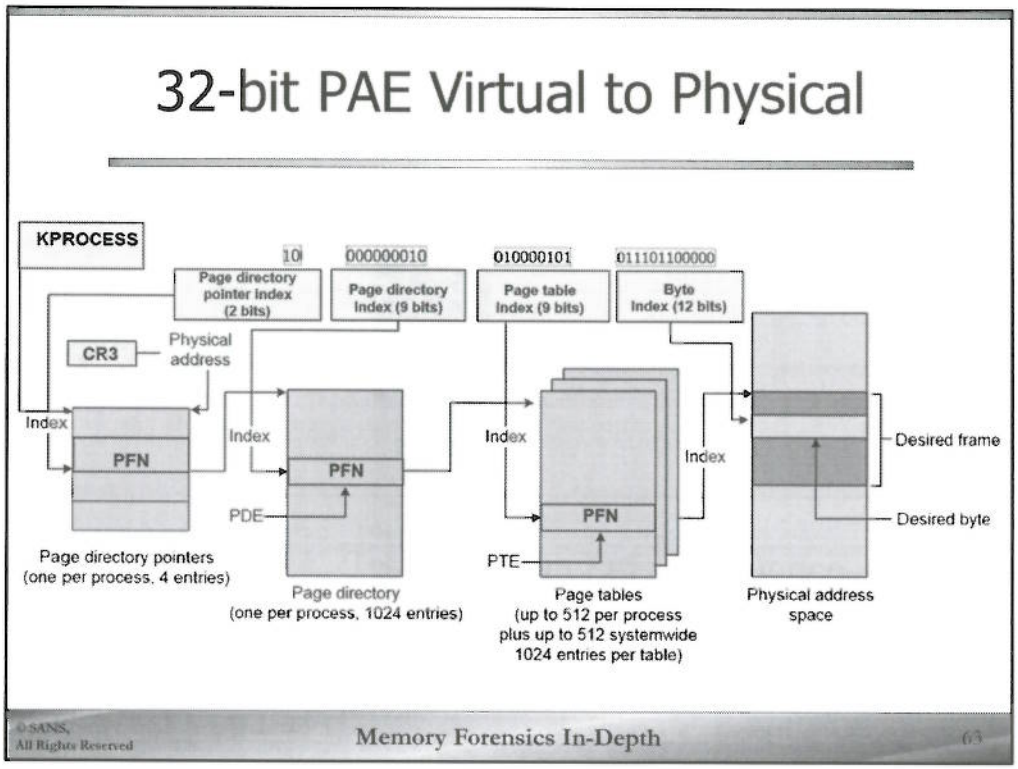
Some other notable flags that are specific to the PTE include offset 8. This flag is used to indicate that the data referenced by this entry is global. That is, it belongs to something in kernel space, which should not be cleared from the cache when this process is paged out. Offset 6 is the "dirty" flag, tracking whether the memory been accessed before (has some process read from it or written to it?)

See Chapter 10 on Memory Management in *Microsoft Windows Internals* for the best reference. There are also references on the web, such as http://blogs.msdn.com/b/ntdebugging/archive/2010/04/14/understanding-pte-part2-flags-and-large-pages.aspx, which cover these flags in some detail.

[1] http://windowsscope.com



The x86 PAE address translation process involves chunking the 32-bit address into 4 separate pieces. The top two bits are used as an offset into the Page Directory Index, an additional layer added to the PAE lookup process. The PDE and PTE index values are only 9-bits in size each, leaving the lower 12-bits for the offset into the physical frame.

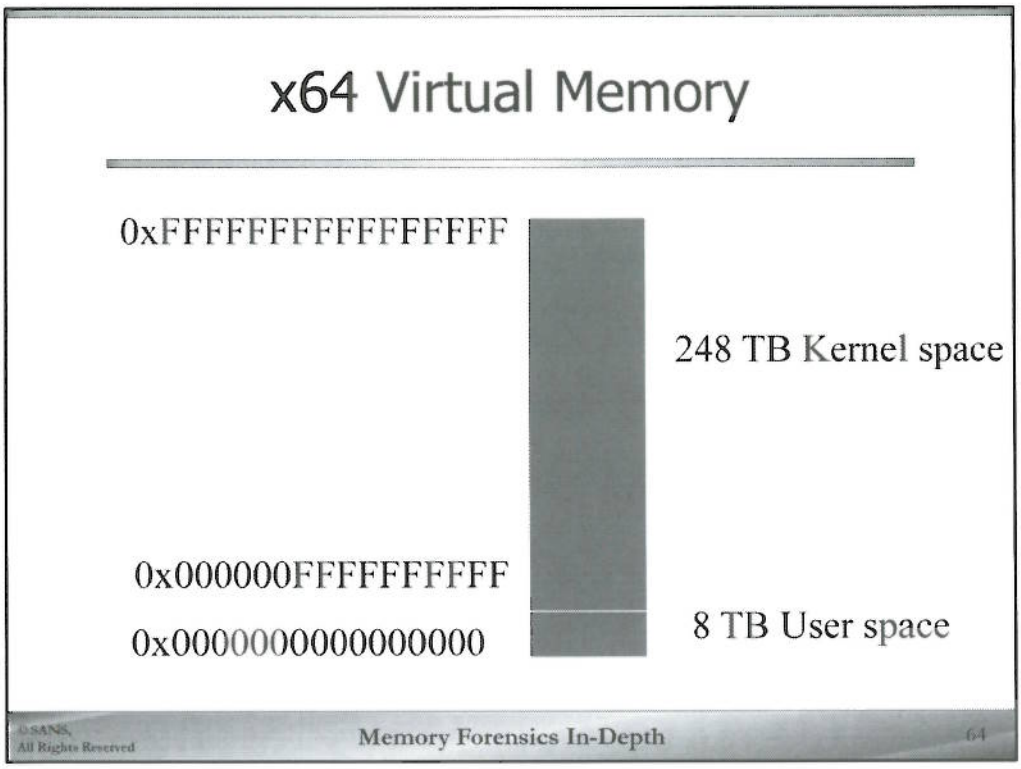


Here's a diagram of the virtual to physical address translation process on 32-bit systems using PAE.

The virtual to physical address translation process is slightly different on 32-bit systems using Physical Address Extension (PAE). Remember that PAE allows 32-bit systems to use more than 4GB of physical RAM using a hardware extension on the processor. Each *process* can still only use 4GB, but the system as a whole can use more than 4GB. That is, virtual addresses are still 32-bits, so each process is limited to 4GB.

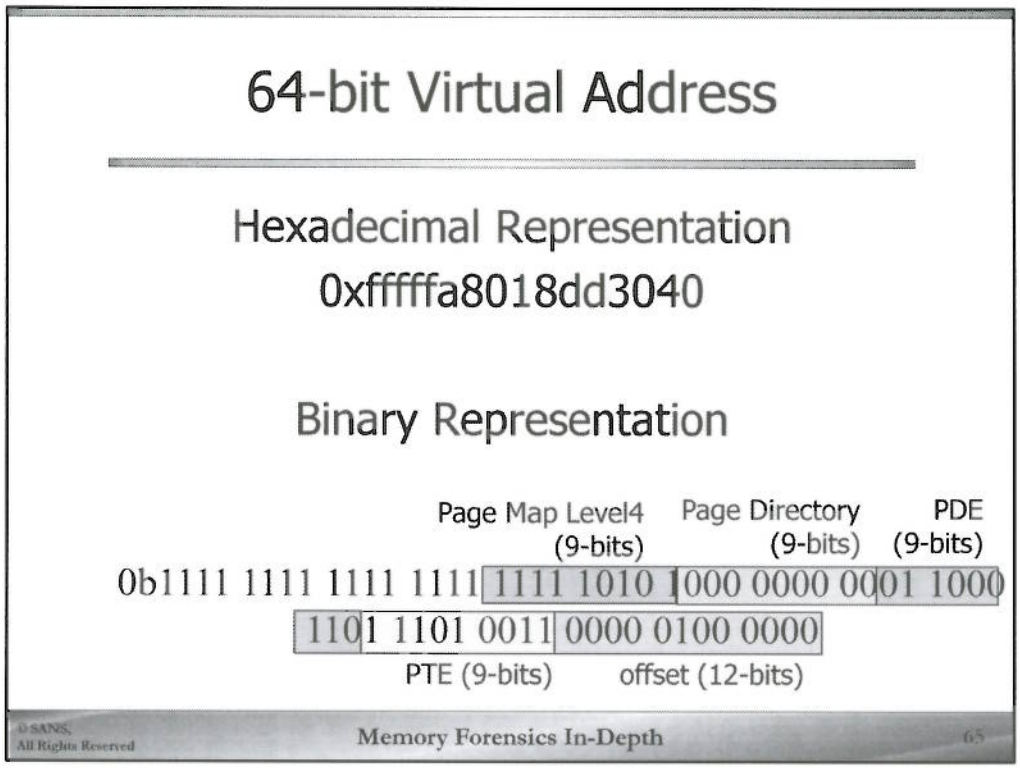
PAE allows the system to use more than 4GB of RAM by extending the size of the PDE and PTE entries from four bytes to eight bytes. It also adds a fourth step, the Page Directory Pointers Table (PDPT), to the lookup process. The ability to address extra RAM is *not* due to the fourth lookup step. The ability to address the extra memory comes from the wider PDE and PTE entries. It's those wider values which can hold the wider offsets in physical memory. Without PAE, the four byte PTE entries, for example, could only go as high as 0xffffffff, or still just 4GB. With an eight byte PTE, however, the PTE can reference a much wider address, and therefore more memory.

Microsoft limits 32-bit operating systems running under PAE to 64GB of RAM—it's not a limit of the architecture. Even with PAE enabled, Windows still uses the 4GB virtual address space, of which only 2GB is reserved for the kernel. If more than 64GB of RAM was addressable, the Page Frame Number database, which keeps track of which frames of physical memory are mapped to which virtual pages, would grow so large that it would take up most of the 2GB virtual address space!

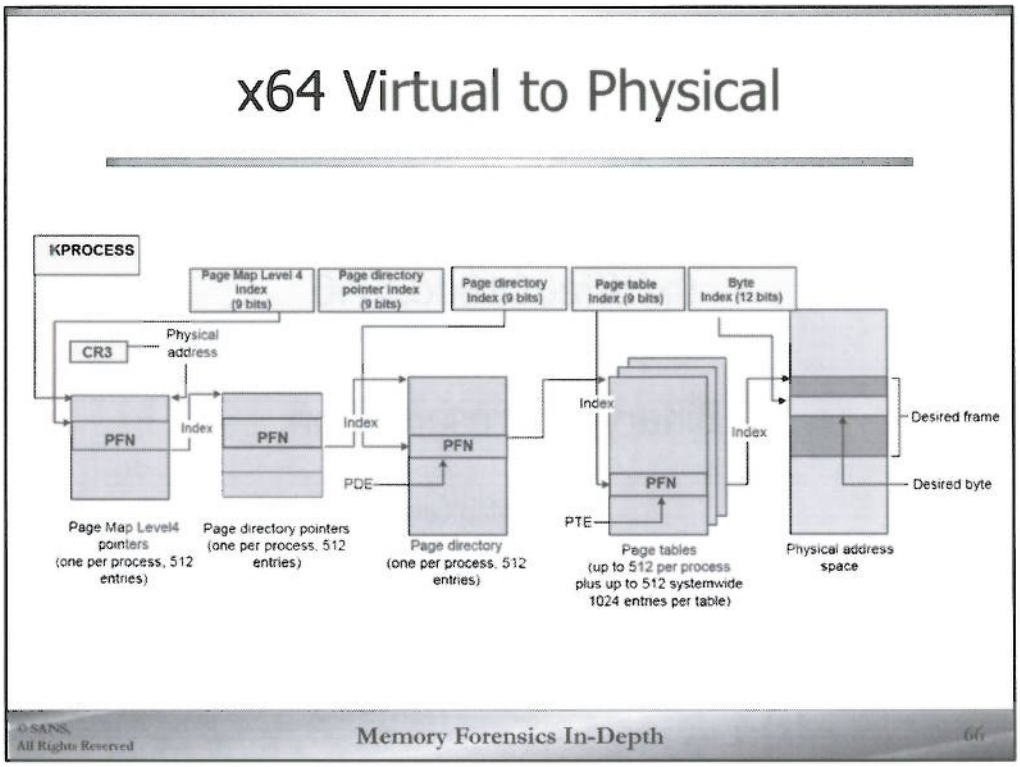


On 64-bit systems, each process sees its own 256TB virtual address space regardless of the amount of system's physical memory. The virtual address space is 256TB on 64-bit systems because 48 bits are used for addressing. Of the 256TB virtual address space, the lower 8TB is reserved for the process memory or "user space". The kernel space occupies the upper 248TB virtual address range.

https://msdn.microsoft.com/en-us/library/windows/hardware/hh439648%28v=vs.85%29.aspx



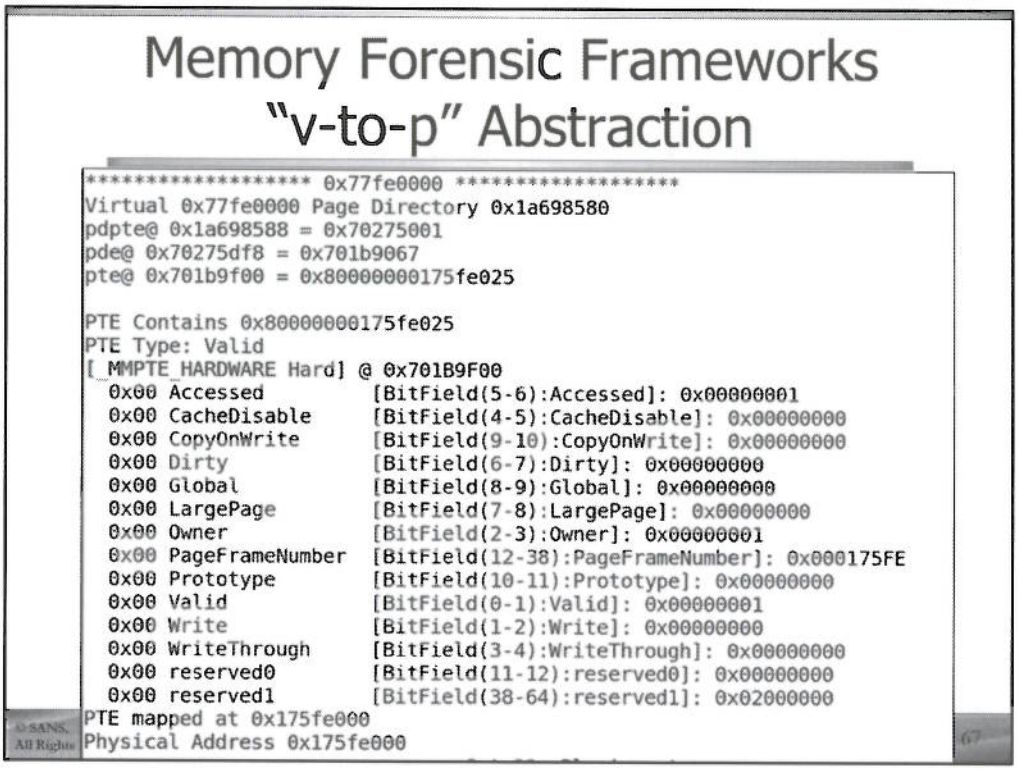
As stated earlier, 64-bit system architectures only use 48-bit for addressing. You can see above, the hexadecimal representation of 64-bit address appears strange to those examiners accustomed to only parsing 32-bit memory images.



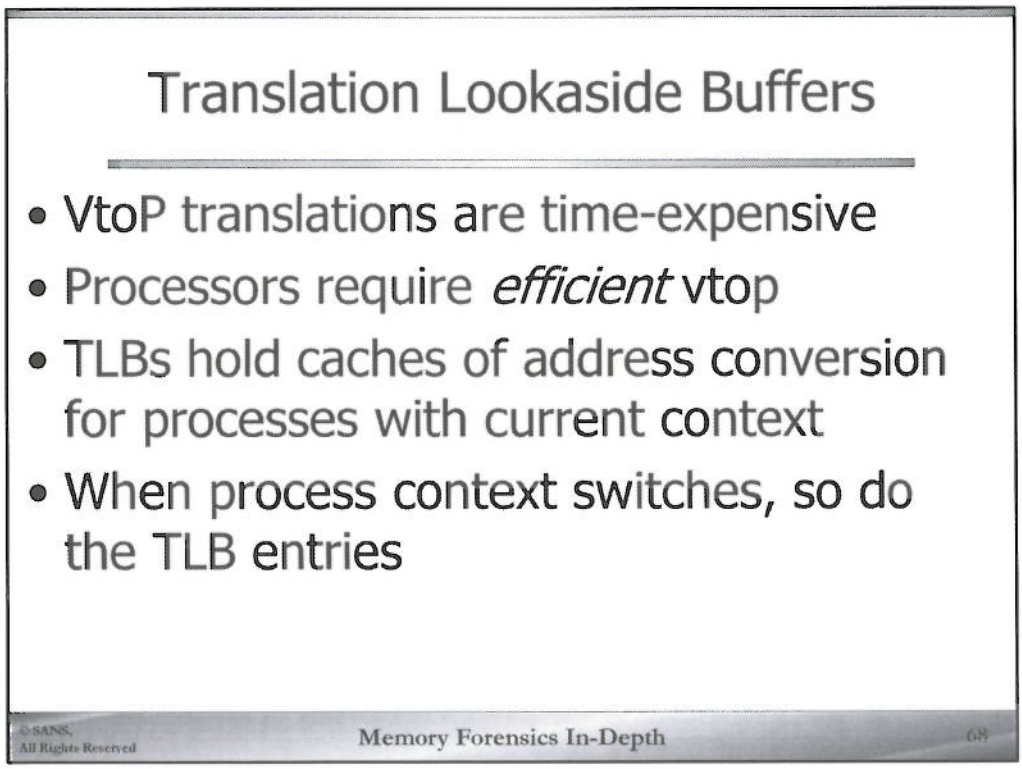
The virtual to physical address translation process is effectively the same on x64 systems, but it's a five stage lookup process. The fifth stage is called "Page map level 4", or PML4. Perhaps even the hardware designers were getting confused by this point!

But as with the previous vtop translation systems, the original virtual address is divided into parts and used as lookups into tables. The first table is pointed to by the DTB for the process.

Although we're not covering it, the address translation process for IA-64 is slightly different. It's important to note that this is the vtop process for x64, but not all 64-bit systems.



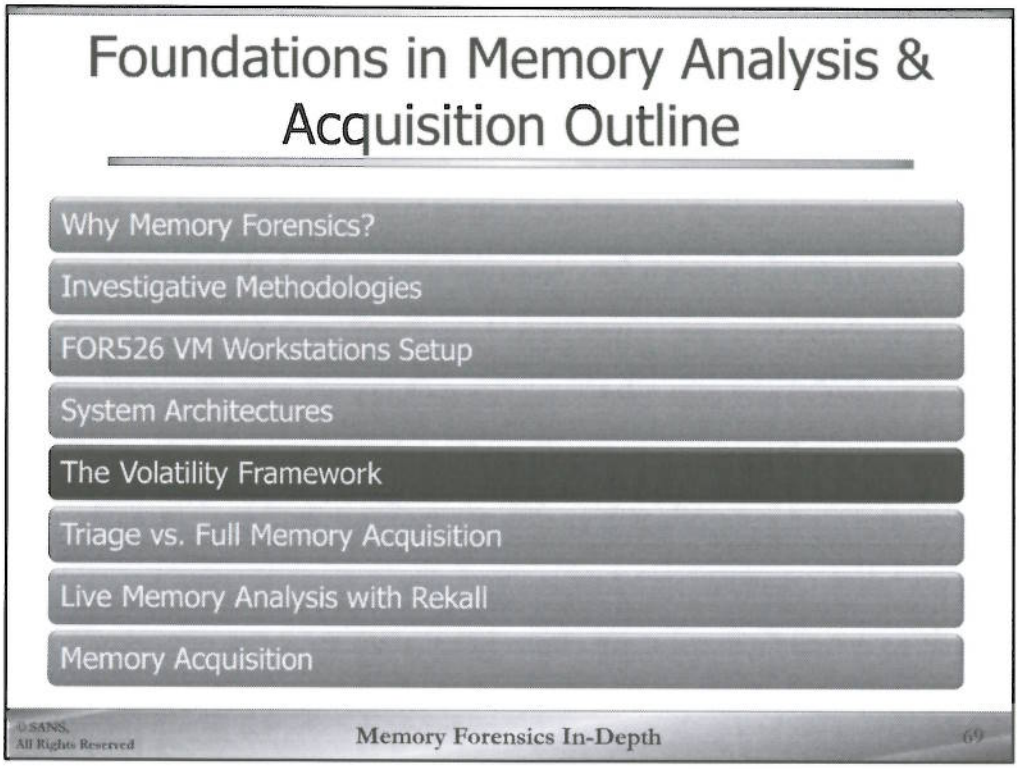
After manually walking through the virtual to physical address translation process, we have a healthy appreciation for the automated fashion with which this translation occurs when using one of the common memory forensic frameworks. Shown above is truncated output from the Rekall Memory Forensic Framework's "vtop" plugin, which converts a given virtual address and displays whether it successfully translates to a physical location. As we will discover, Rekall can incorporate the page file(s) into its memory analysis and with "vtop" output, you can determine whether the data referenced by a specific virtual address has been paged to disk.



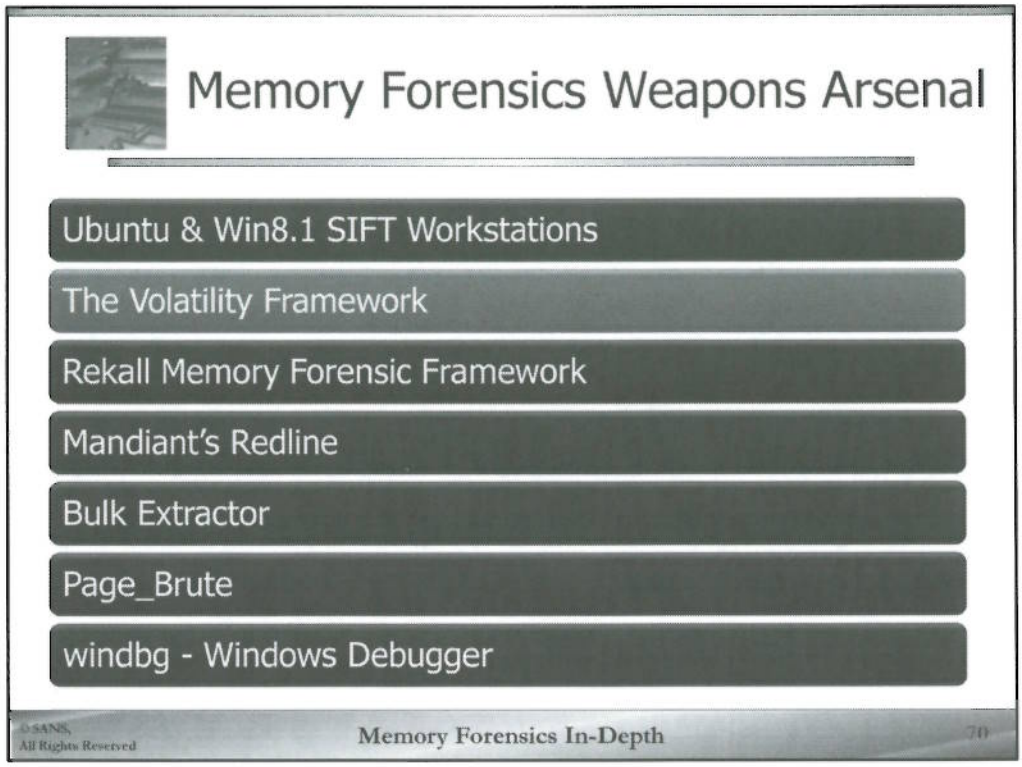
These three, four, and five stage look processes take a lot of time. In computer science terms, they are "expensive". To be more efficient, the processor maintains a cache of virtual to physical mappings for each process. This cache is called the Translation Lookaside Buffer (TLB) and prevents the processor from having to do these expensive lookups every time it wants to translate a virtual address into a physical one. A TLB has a fixed number of entries and is stored in the processor, inaccessible to the operating system. This means it's also inaccessible to a forensic examiner.

In x86 system architecture, the TLB cache is made up of at least two tables (per processor), the *ITLB*, Instruction Translation Lookaside Buffer, and the *DTLB*, Data Translation Lookaside Buffer. The ITLB hold virtual to physical translations for code and the DTLB hold virtual to physical translations for data. Normally these two buffers are synchronized and translate code and data memory accesses to the same physical frame.

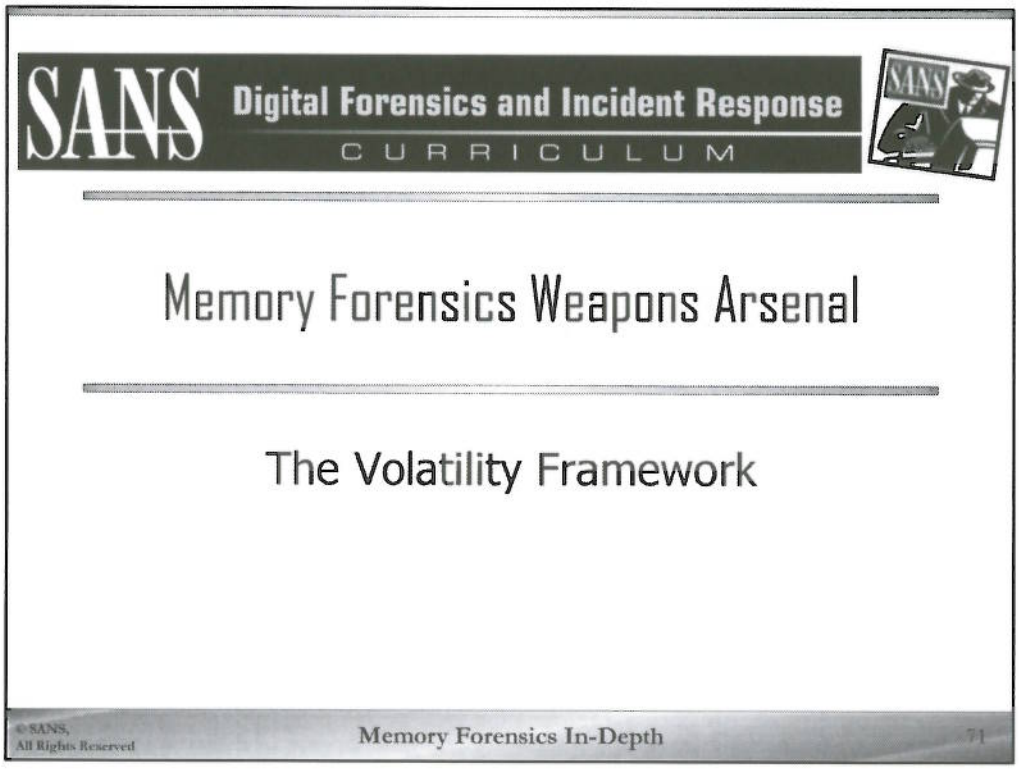
Because the virtual to physical mappings are different for each process, the processor has to clear the TLBs every time it switches between processes. There is a flag in each PTE which indicates if the memory is "global". That is, if the memory being referenced is in kernel space. Because kernel memory is the same for all processes, it does not need to be cleared when the processor switches to a new process. Hence the memory is "global", and does not need to be cleared.



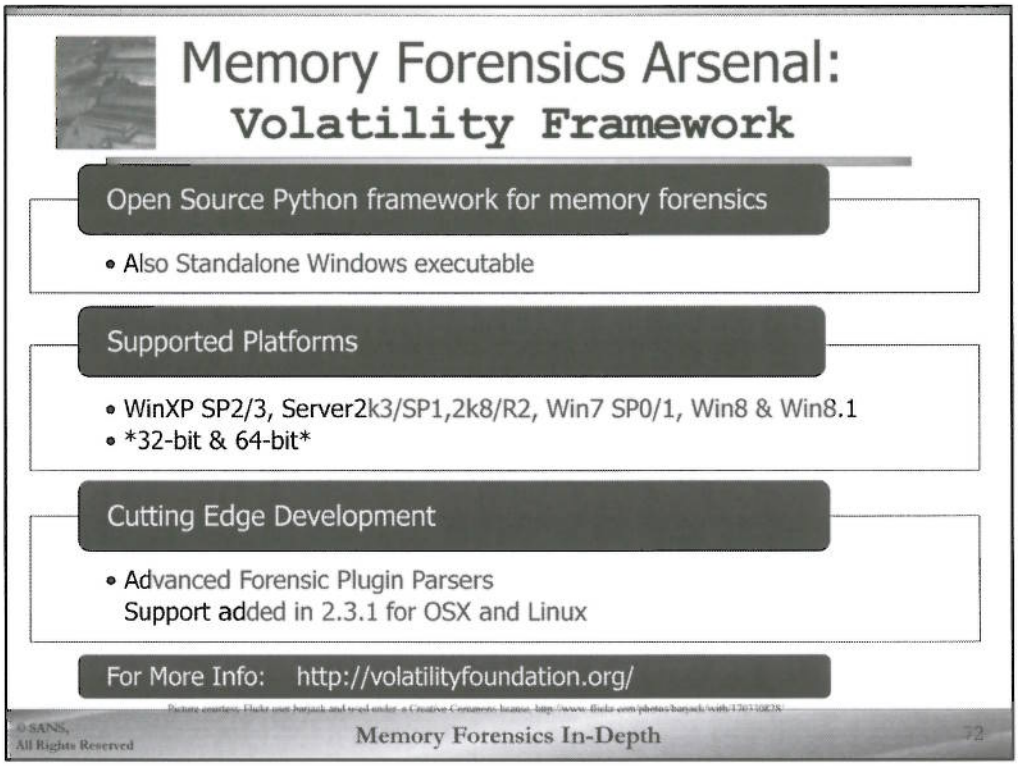
This page intentionally left blank.



We will now introduce the Volatility Framework, a collection of memory forensics parsing tools. We have installed the framework on both of the virtual machines distributed in this course, the Windows 8.1 and the Ubuntu SIFT. In utilizing various plug-ins, or specific parsing tools, from the framework, we will explore the Windows memory structures and the significance of their presence in a system memory image. Remember, the end goal of learning how to use each of the tools listed above in the Memory Forensics Weapons Arsenal is to be able to apply their use to our forensics investigations, be it incident response or employee investigations.



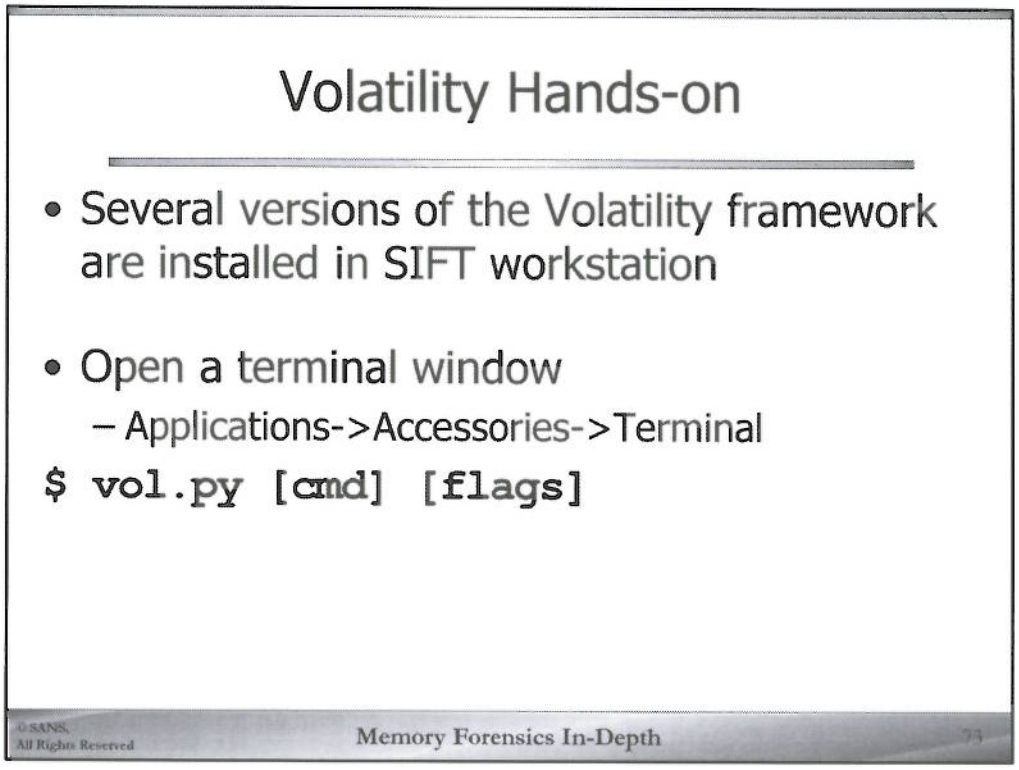
This page intentionally left blank.



Volatilty is an open source framework, meaning we were free to modify it for our needs. Next, Since the software is licensed under the General Public License (GPL), we are able to freely redistribute the original framework and any subsequent modified versions. The Framework's command line interface produces text output. We believe there is great value in using the command line interface when teaching. There is no need to learn a GUI, and you are presented with all of the output at once. It forces students to determine which data are important.

Volatility can do analysis on 32-bit and 64-bit versions of Microsoft Windows XP Service Pack 2 or 3, Windows 2003, Windows Vista, Windows Server 2008/R2, Windows 7, Windows 8/2012 and Windows 8.1. Service pack zero, as Volatility refers to them, is also known as the Release to Manufacturing (RTM) version of the operating system. There was no RTM of Windows Server 2008. That product started with Service Pack 1.

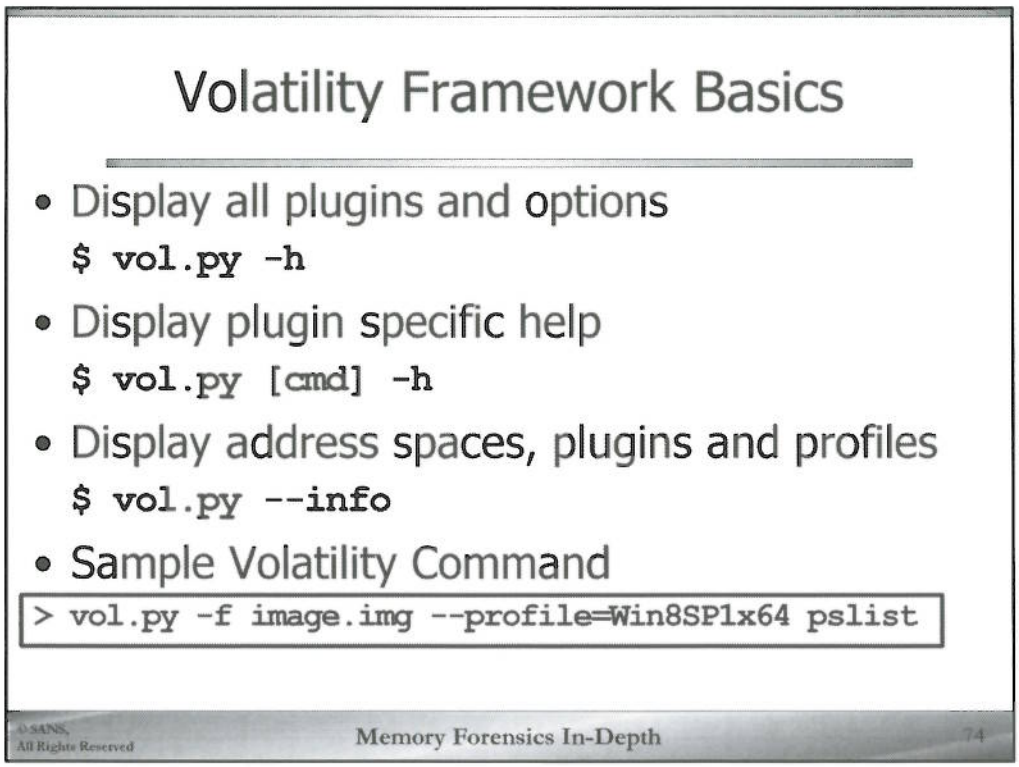
Volatility is not the only free memory forensics tool available. We will also introduced a number of other free tools to include a fork of Volatility called Rekall. This program, which offers much of the same memory parsing functionality as Volatility but in a more efficient fashion. We will be covering the unique live auditing capabilities of Rekall as well in the upcoming "Triage" section of this class.



We have already installed Volatility 2.4 with all of the required dependencies in your SIFT workstation.

Volatility is all command line based, so to use it you will need to open another command prompt. You can open a command prompt by going to the "Dash Home" menu option (top icon on the left toolbar), and typing "Terminal" or simply by clicking the terminal window icon on the left toolbar.

The general syntax for running Volatility is **$ vol.py [cmd] [flags]**. On many systems you have to call the Python interpreter and then the Volatility command, vol.py. On the SIFT workstation we have set it up so that you just type vol.py. This invocation is followed by a command to run and the flags to use on that command. You will always need at least one flag "-f" to specify the memory image you want to process with Volatility. Unlike many command line programs, which use the arguments after all of the command line options as the input files, Volatility requires you to specify the input file using a flag.



To get help from Volatility, invoke the framework using the -h flag. When invoked in this manner, the framework displays the list of available plugins and the applicable command line options for those plugins.

Many plugins have options specific to them. For example, the pstree plugin has a verbose mode which is not found in every plugin. This means, to find out which options apply to the plugin you are calling, you should first run the command using the -h flag. For example, take a look at the output you got when running the framework as:

```
$ vol.py -h
```

Now compare that to using the -h flag with pstree:

```
$ vol.py pstree -h
```

Notice the --verbose or -v flag which is now available as a default options, but the output will vary widely across plugins.

```
$ vol.py --info
```

Sometimes you actually know what version of Window your target system was running when the memory was acquired. Remembering the exact case-sensitive profile syntax can prove to be a challenge. The "--info"

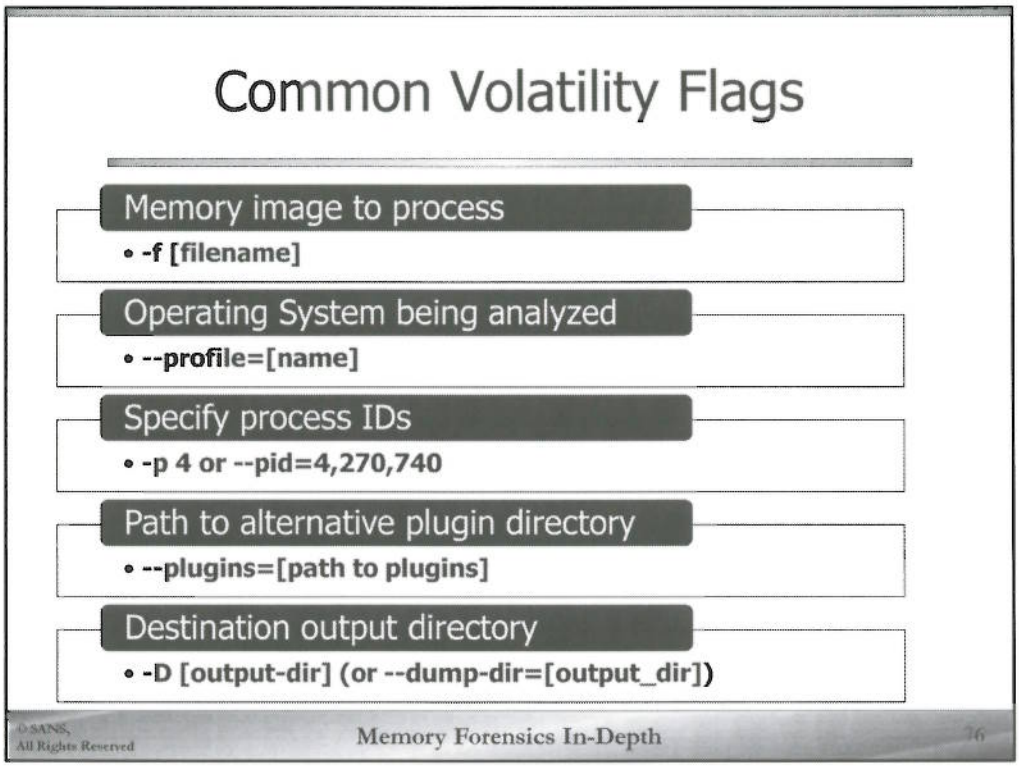
If you're going to be running a lot of commands on the same memory images, you can save yourself some typing by setting some environment variables. The volatility framework will check these variables when it runs and eliminates the need for you to put these values on the command line. You can specify the filename and profile to process with the variables VOLATILITY_LOCATION and VOLATILITY_PROFILE, respectively.

As an example, here's how to set these environment variables for the xp-laptop memory image we'll be using throughout the course.

```
$ export VOLATILITY_LOCATION=file:///home/sansforensics/Desktop/cases/xp-laptop-2005-07-04-1430.vmem
$ export VOLATILITY_PROFILE=Win7SP2x86
```

You can, of course, still put something on the command line and override them, but be careful. If you forget to override something, you could end up with unexpected results or an unexpected error.

To avoid confusion we are not going to use these environment variables in this class, but we wanted to show them to you.



There are many Volatility command line options, but here are ones you will use most often.

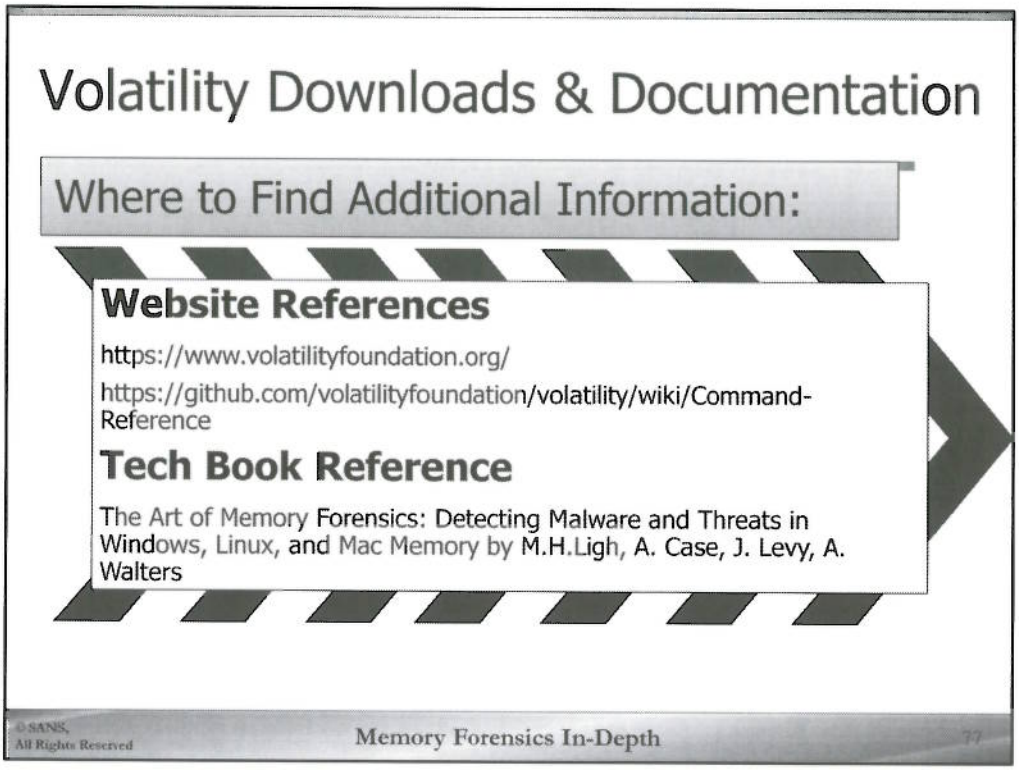
Every plugin requires you to specify the memory image to be processed using the -f flag. Note that if the file you specify does not exist, Volatility will present you with an error about that even if you have just asked for help with the -h flag.

Many plugins require you to specify the operating system found in the memory image using an operating system profile. You can choose a profile from a set included with Volatility. There will be a list of available profiles on a later slide. They are stored in volatility\plugins\overlays\windows. (Because, you know, they're profiles, and thus stored in the directory 'overlays'. Did we mention that Volatility is complex?)

Several plugins allow you to specify which process or processes you want information. You can specify a single process using the --pid flag, or even a comma separated list of processes.

Many examiners created their plugins or use those that have been contributed by other members of the forensics community. If you have a separate plugin repository that you would like to reference, you must specify the plugin directory location as the first variable in the Volatility command with the "--plugins=[path to plugin directory]

In addition, many of the plugins will require a destination directory (or "dump directory") to be specified upon their invocation. In most cases, the destination directory can be specified using "--dump-dir=[destination directory]" or the shorthand "-D [destination directory]".



The tagged releases of Volatility, the stable and tested builds, are available from the website of Volatility Foundation, https://www.volatilefoundation.org/. Along with just the Python source code, you can also get a standalone executable for Windows systems and a Windows installer. These are handy if you need to run Volatility on a machine which does not have Python installed or which you cannot install anything, such as when you're doing incident response. If you don't know where to start with Volatility, start with one of these packages.

Along with the Volatility code, there are some dependencies necessary to run parts of the framework. Although you don't need to install any of these to get started, you will get some warning messages about them. The complete list of which dependencies are needed for which plugins is at https://github.com/volatilityfoundation/volatility/wiki/Installation, but in general you will want to install

- Python (http://python.org)
- PyCrypto (pre-compiled for Windows at http://www.voidspace.org.uk/python/modules.shtml#pycrypto)
- Distorm (http://code.google.com/p/distorm/)
- and YARA-Python (http://code.google.com/p/yara-project/downloads/list).

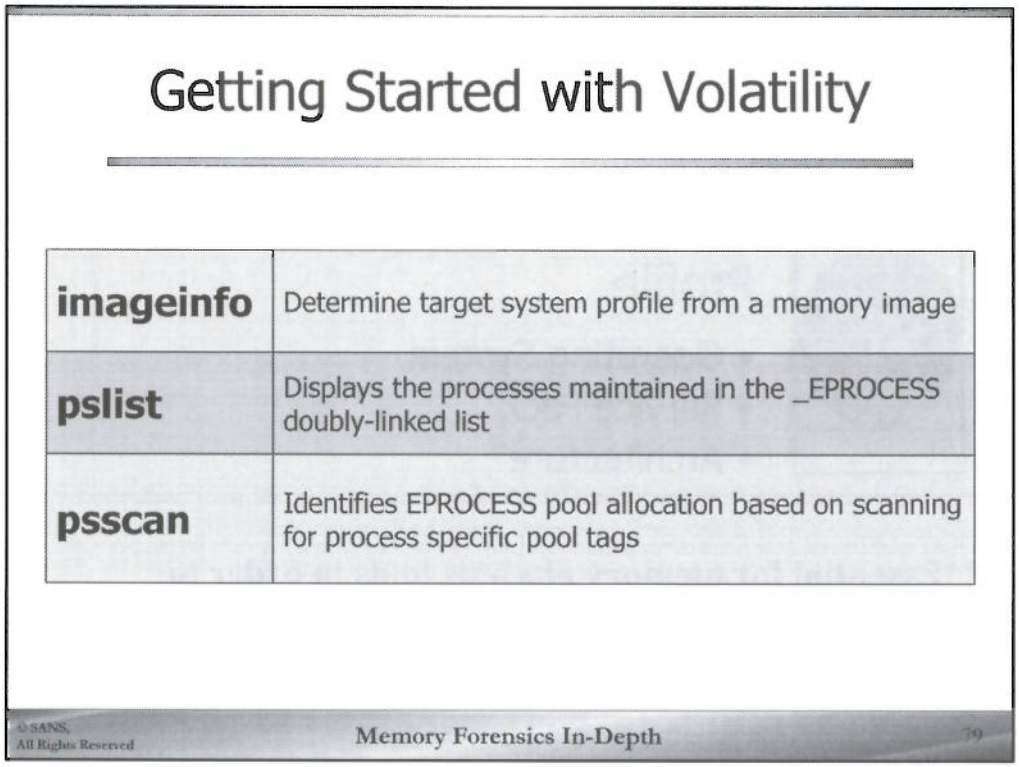
The Volatility website has documentation on the all plugins, both in the main trunk and many of the community contributed plugins in the project. This documentation is supposed to be tracking the trunk, not the tags. When in doubt, the help screen provided by the plugin in the specific version you are using should provide insight into its implementation. But if you have questions which can't be answered there, or just want to know more about what a plugin is supposed to be doing, take a look at the online documentation.

The first link here divides all of the plugins up by categories. This is handy if you are interested in a particular kind of data. For example, plugins which work on processes and DLLs are grouped together. Some context is provided for each plugin and an example is given of its usage. They may also contain links with more information about each plugin.

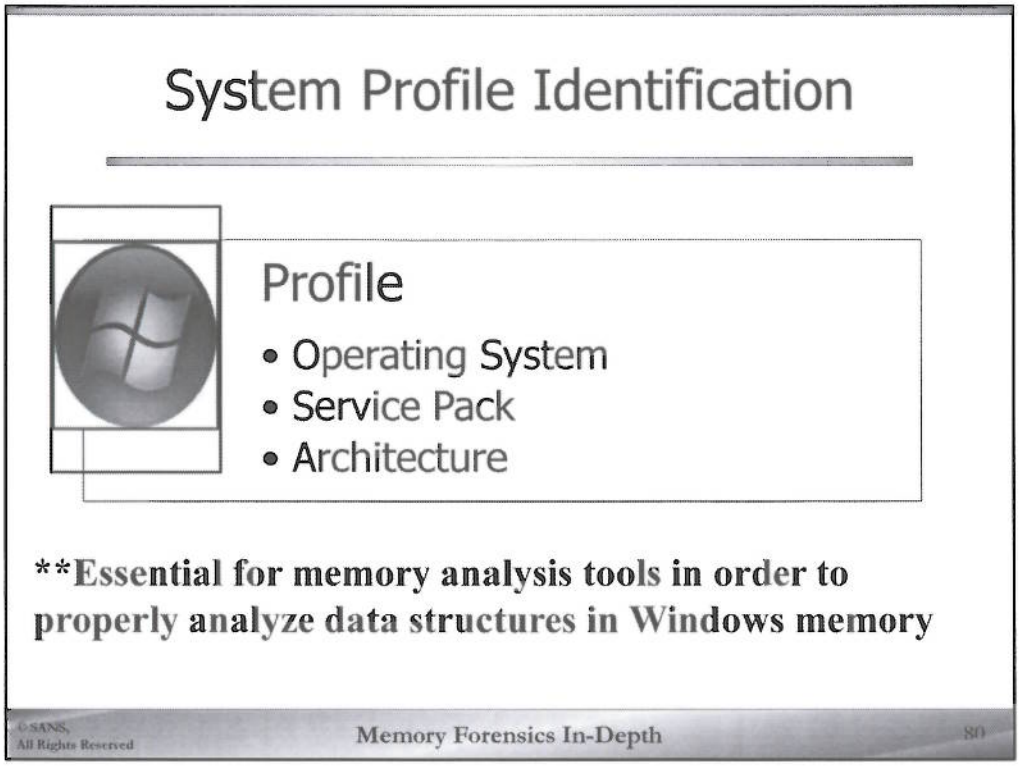
https://github.com/volatilityfoundation/volatility/wiki/Command-Reference

In addition, you may find memory analysis research and tutorials offered by other professionals in the community as well as the members of the Volatility development team on the Framework's wiki. This collection of blog posts, white papers and presentations offers insight into the bleeding edge of the Volatility tool and the practical applications other examiners have found for its use in both forensic investigations and intrusion analysis.

https://github.com/volatilityfoundation/volatility/wiki/Volatility-Documentation-Project



In order to solidify your understanding of the Volatility framework and its required syntax (and build "muscle memory"), we will walking through 3 quick plugins that will get us in the game immediately. These plugins, **imageinfo**, **pslist**, & **psscan**, allow us to first determine the profile of our target system, enumerate actively running processes and identify terminated or hidden processes. You will be amazed, if you aren't already, by how quickly you are able to begin parsing memory images - at the command line, no less!

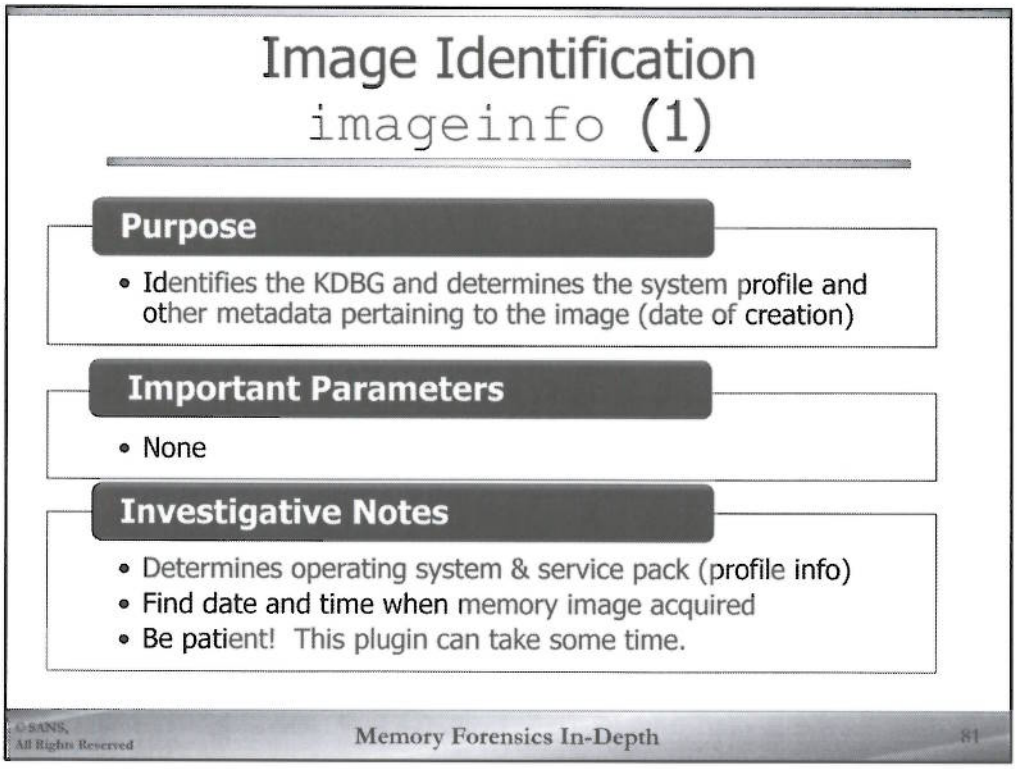


It's time to start preparations for memory analysis. Before even looking at the 6 Steps of Investigative Methodology, we must first determine the *profile* of the system from which the memory image came. The analysis of a 32-bit Windows XP PC is going to be different than looking at a 64-bit Windows 2008R2 server based on structural changes within Windows memory. In order for our tools to parse properly, we either have to feed them this profile information (as is the case with Volatility) or the tool itself must determine the system profile (as seen in Rekall and Redline).

For memory analysis, we need to determine the:

- Architecture
- Operating System
- Service Pack Number

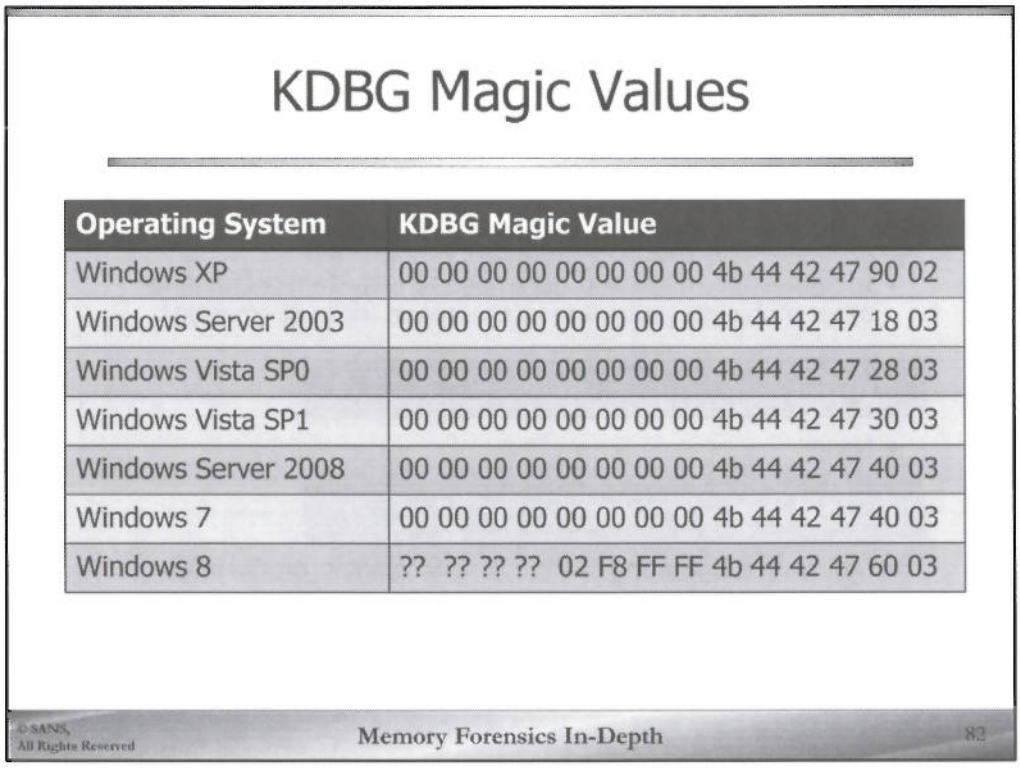
Never assume or guess the system profile type! Users are often confused about what operating system they're running, let alone which Service Pack level. Take the extra time to determine these values for yourself.



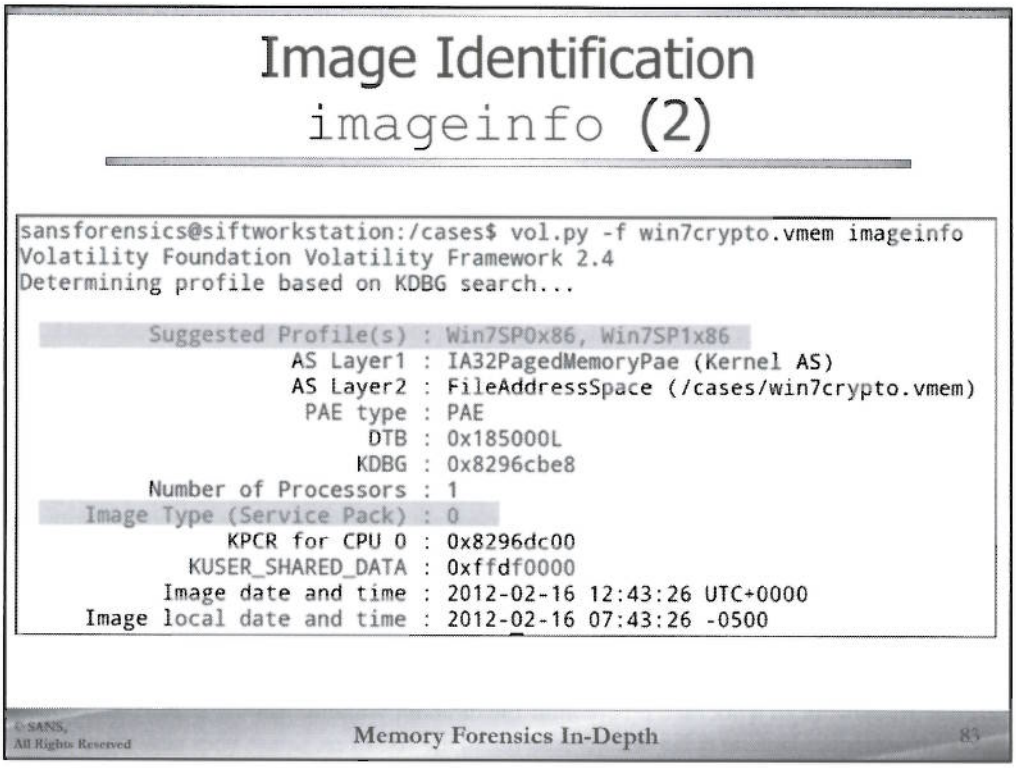
The **imageinfo** plugin identifies the system profile (Windows version, service pack and system architecture) by locating the KDBG, kernel debugging data block, within the memory image. The start of the KDBG can be identified by a unique header specific to OS version so once the imageinfo plugin searches for and locates the "magic number" of the KDBG, it has an initial guess as from what type of system the image came.

The plugin goes one step further and walks the list of active processes, querying a specific value from the Process Environment Block of each process that indicates Service Pack. In a majority rules manner (because these values are not always the same within each process), imageinfo polls each process and then goes with the service pack that most of the processes claim. It presents the Service Pack option in the plugin output along with the time/date of the system at the time of imaging.

Most Volatility plugins that work by scanning for pattern matches take a considerable amount of time to run and imageinfo is no exception. Be patient - at this point, in order to use subsequent Volatility commands, the system profile must be specified.



The KDBG starts with a magic value, which is different for each operating system. Each magic value starts with eight bytes of zeros, the ASCII string KDBG, and then a two-byte size of the KDBG structure (represented in little endian). The KDBG is 0x290 bytes in size in Windows XP, and has grown to 0x340 bytes in Windows 7. (Yes, technically it's a four byte representation of the size, so you could append each of the magic values above with two extra zero bytes.) Windows 8 proves to be a bit different in its presentation of the KDBG, as it is encoded.



Imageinfo may 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 suggest 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:

sansforensics@siftworkstation:/cases$ vol.py -f win7crypto.vmem imageinfo
Volatility Foundation Volatility Framework 2.4
Determining profile based on KDBG search...

Suggested Profile(s) : Win7SP0x86, Win7SP1x86
AS Layer1 : IA32PagedMemoryPae (Kernel AS)
AS Layer2 : FileAddressSpace (/cases/win7crypto.vmem)
PAE type : PAE
DTB : 0x185000L
KDBG : 0x8296cbe8
Number of Processors : 1
Image Type (Service Pack) : 0
KPCR for CPU 0 : 0x8296dc00
KUSER_SHARED_DATA : 0xffdf0000
Image date and time : 2012-02-16 12:43:26 UTC+0000
Image local date and time : 2012-02-16 07:43:26 -0500

After examining your memory image, Volatility will suggest an operating system profile you should use in further analysis. Note that the format for the suggested profiles is OS-ServicePack-Architecture and are case-sensitive!

In the initial lines of output, Volatility suggests more than one possible profile. This is because some data in the profiles in the same as seen in the previous discussion on KDBG Magic Values, and it's not sure which operating system it's looking at. By combining this field with the "Image Type" field, we can determine the correct profile. The data associated with each process contains a string, called CSDVersion, which contains the Service Pack level of the operating system. In RTM releases, or Service Pack 0, to use Volatility's term, this string may blank to indicate SP0. On Service Pack 1 systems, the string is "Service Pack 1", and so on. There is one string for each process, and some of them are always blank. That is, even on a Service Pack 2 system, for some processes the CSDVersion string is blank. (This can be because that part of memory was paged out, but from our perspective the string is still blank.) To deal with this uncertainty, Volatility counts which string occurs most often in all of the processes and reports that as the Image Type.

Using the suggested profiles, and Image Type field in the output, we can surmise that the correct profile to use for this memory image is Win7SP0x86. Therefore in subsequent commands to analyze this memory image, we will use the --profile flag with that label.

Address Space Layers
Another part of the imageinfo output is the stackable address layers or "AS Layer". An address space is a set of functions which act on the input file or a lower-level address space to retrieve information. This allows, for example, for the framework to work on a hibernation file directly, without needing to fully decompress it first. A hibernation file address space does the decompression on the fly. (Yes, this is very slow; but from an academic perspective on computer science, it's wicked cool.)

Most often you will see these two address spaces in the imageinfo output. The first is an address space for reading data from virtual addresses using the vtop methods we discussed earlier. The second, and lower, address space, is a simple wrapper around reading the input file from the disk. In general you probably won't need to worry directly about address spaces during your analysis, unless you require that Volatility apply a specific offset when reading in memory image of a distinct format. For example, VMSS files pulled from a VMWare snapshot require that the 65535 byte header be skipped in order to correct parse the memory image.

The next few pieces of data are not immediately useful, but are needed for later analysis. The first, PAE, indicates whether or not Physical Address Extensions were enabled on the target. Next is the physical offset of the Directory Table Base of the System process. It is followed by the virtual addresses of the Kernel Debugging Data Block and the Kernel Processor Control Region. Finally, the framework gives us the virtual address of the KUSER_SHARED_DATA structure. It's a kernel data structure which holds some useful information. In particular, it holds the system date and time, as shown on the next slide.

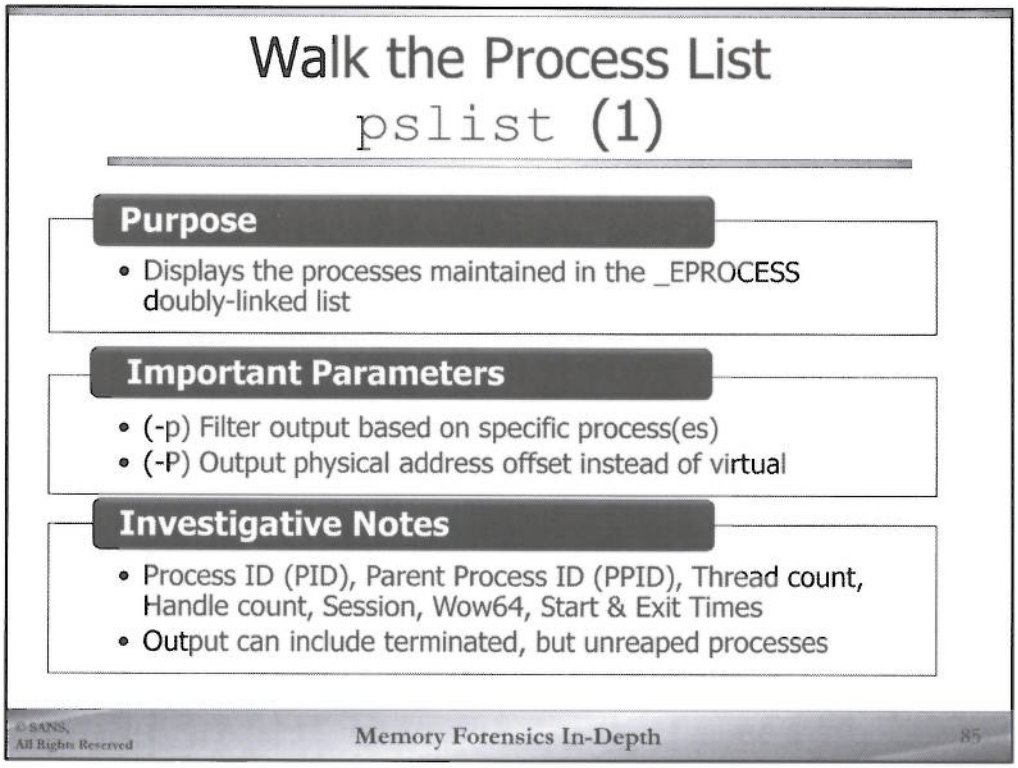
Image date and time
Finally, Volatility displays some immediately useful information. We get two timestamps:

```
        Image date and time : 2012-02-16 12:43:26 UTC+0000
  Image local date and time : 2012-02-16 07:43:26 -0500
```

This was the date and time of the system when the memory image was captured. The first timestamp was the time of the system clock, which is in UTC. The second timestamp is the system clock adjusted for the time zone the machine was configured for. (Remember those report writing skills. This time zone was not necessarily the local time. It's whatever the user set for the time zone!)

Based on the date and time, it looks like this machine was configured for UTC-5, or Eastern Standard Time.

Except where noted *all* timestamps reported by Volatility are in UTC.



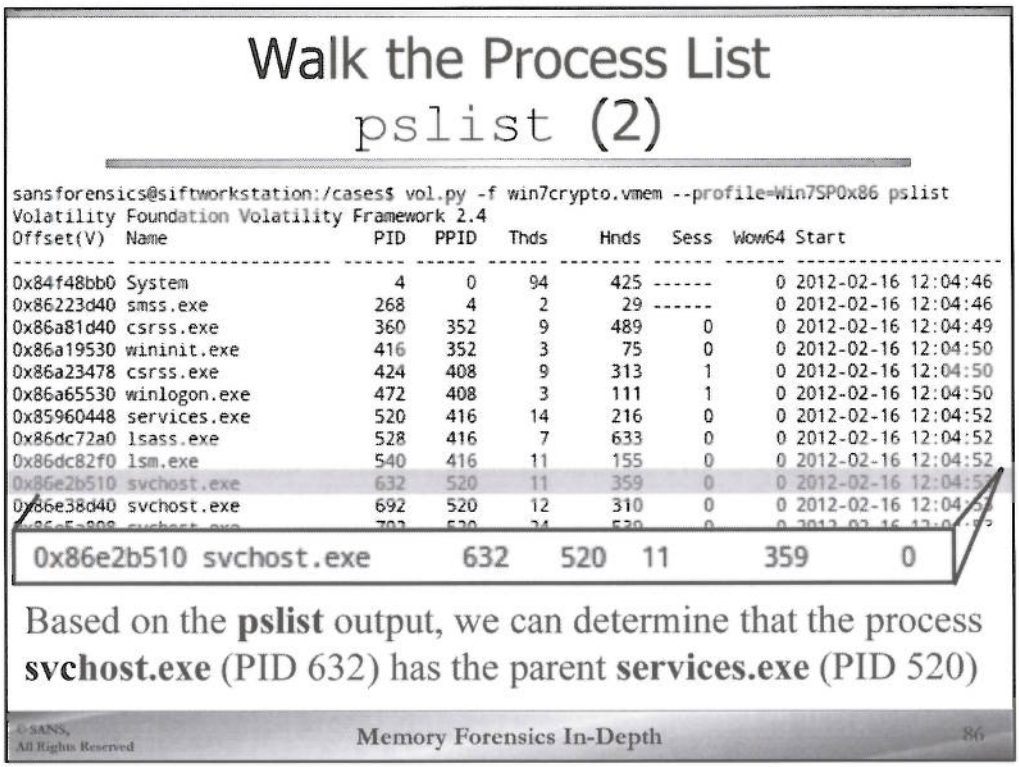
We're now going to use one of the list walking plugins in Volatility. We're going to start with walking the list of processes and the plugin **pslist**. This plugin uses the **KDBG** (Kernel Debugging Data Block) to find the PsActiveProcessHead variable. That variable points to the doubly linked list of processes - the very one that the Windows operating system uses to enumerate running processes when **task manager** or **tasklist** is run. For each process, it takes the virtual address of a pointer to that process, converts it to a physical offset, subtracts back to the start of the EPROCESS block, and reads from that offset in the memory image.

The pslist plugin has two notable options. If you are only seeking to show output for specific processes, you may use the "-p" in order to filter the output. For example, in order to filter on and only show output for specific processes, you may separate them with a comma:

```
# vol.py -f Win7Crypto.vmem --profile=Win7SP0x86 pslist -p 844,1092
```

Another valuable plugin option for pslist, "-P", outputs the EPROCESS structure locations (first column in the output) in physical address offsets instead of the default virtual address offsets. This is helpful if your investigation requires the use of manual parsing of the EPROCESS structure.

We will explore the EPROCESS block in great detail later in the course, but right now, we will simply describe it as a structure in kernel memory that contains process "metadata". This metadata includes information such as Process Identifier, Parent Process Identifier, Process name and the number of threads and handles within each process. There are many pointers maintained within each process' EPROCESS structure, so it is key that we become familiar with the various tools that allow us to extract this information. The pslist plugin is just one of those tools that we will discussing in this class allowing visibility into the EPROCESS structure.



As you study the output of the pslist plugin, note that the process list is presented in chronological order, with the processes with the earliest start times shown at the top of the output. The first process "SYSTEM" is indicative of the most recent boot time of the target system, so its start time correlates with the last time the system was booted, whether a cold boot or a simple reboot. You will discover after doing several investigations, and even here on the output of this slide above, that sometimes the SYSTEM process does not have a start time.

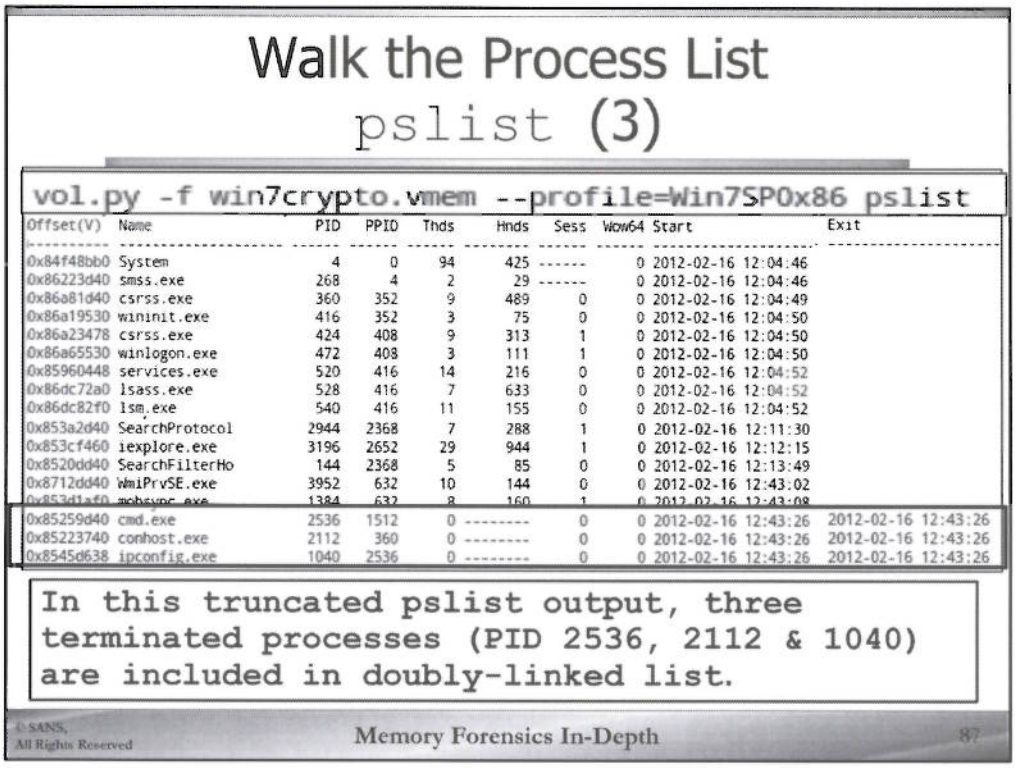
Warning: Earlier versions of the Volatility framework misrepresented this null value as 1970-01-01, Unix Epoch time. Though this has been corrected in most of the plugins in the most recent release of the framework, you will still see this null time/datestamp value represented as the Unix Epoch in volshell, for example.

If the SYSTEM process does not have a start time, the investigator may use the smss.exe process start time to show when the system last booted, as well.

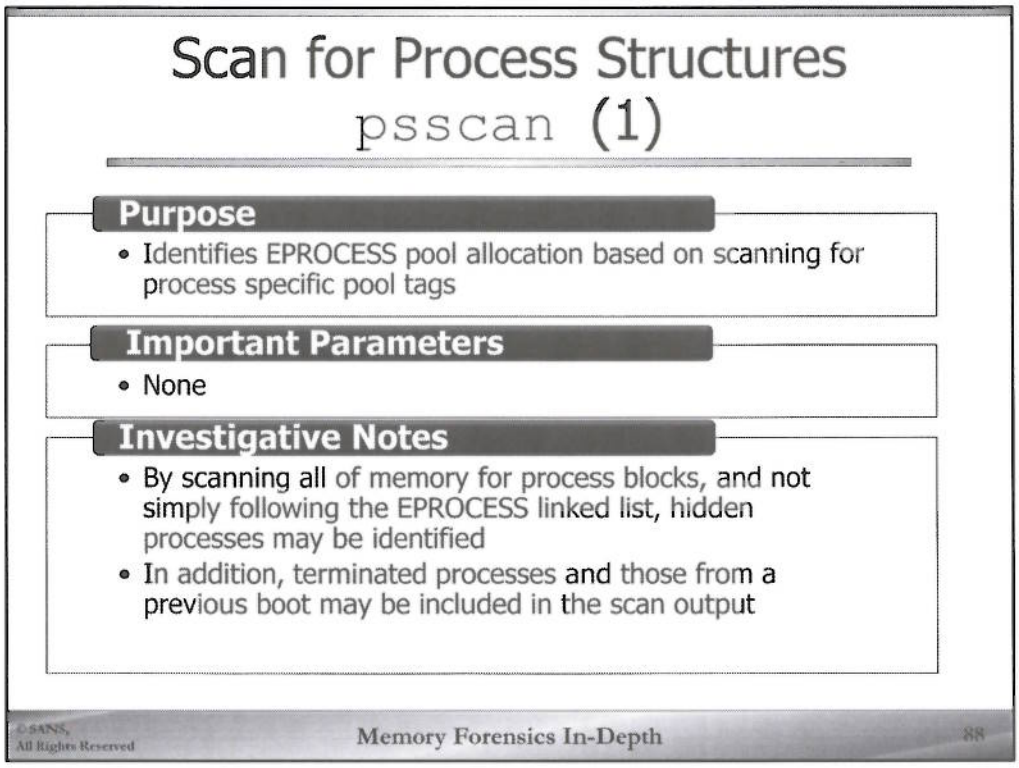
Two additional output columns that are new to **pslist** as of Volatility 2.3 Framework are the **Sessions** and **Wow64** columns.

The WoW64 (Windows on Windows) subsystem allows 32bit processes to run on the 64bit OS. If the memory image is 32-bit, then this field has no meaning. However, if the OS being analyzed is 64 bit, this field allows the analyst to discover whether the processes running are 32 or 64 bit. It is expected that most system processes (csrss.exe, lsass.exe, etc.) will be 64 bit. However, it is not at all unusual for many user (GUI) processes to be 32-bit.

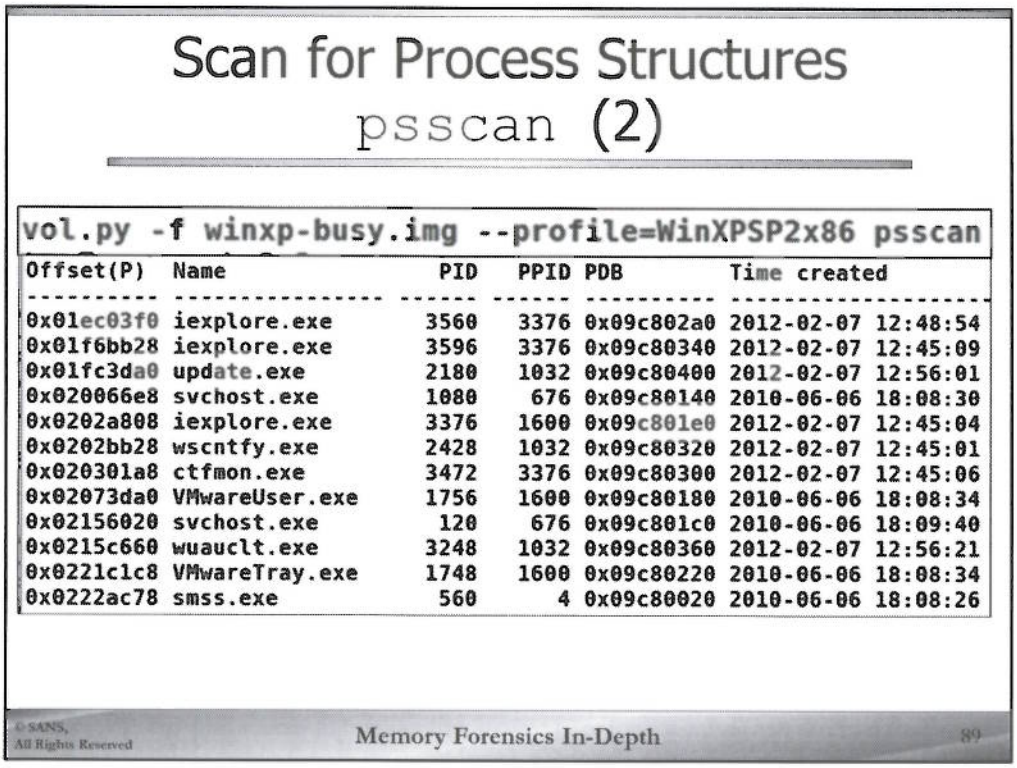
The sessions field used in more modern versions of Windows represents which logon session the process belongs to. Session data can be used to group processes together that might not otherwise be obviously related. Most single user machines will only have session 0 (hosting system processes) and session 1 (hosting the user's desktop). However, when analyzing multi-user terminal servers where many users connect to a machine via RDP, this field has additional relevance.



Just a warning for investigators - some anomalous entries may exist in pslist output that are **not** indicative of malicious processes. Sometimes, terminated processes can stick around in the doubly-linked list even when they clearly have no running threads or open handles due to another process having an open handle to the terminated process. With a handle still open, the kernel will not deallocated the process object. Once all of the handles to a terminated process close, the kernel will destroy the process object.



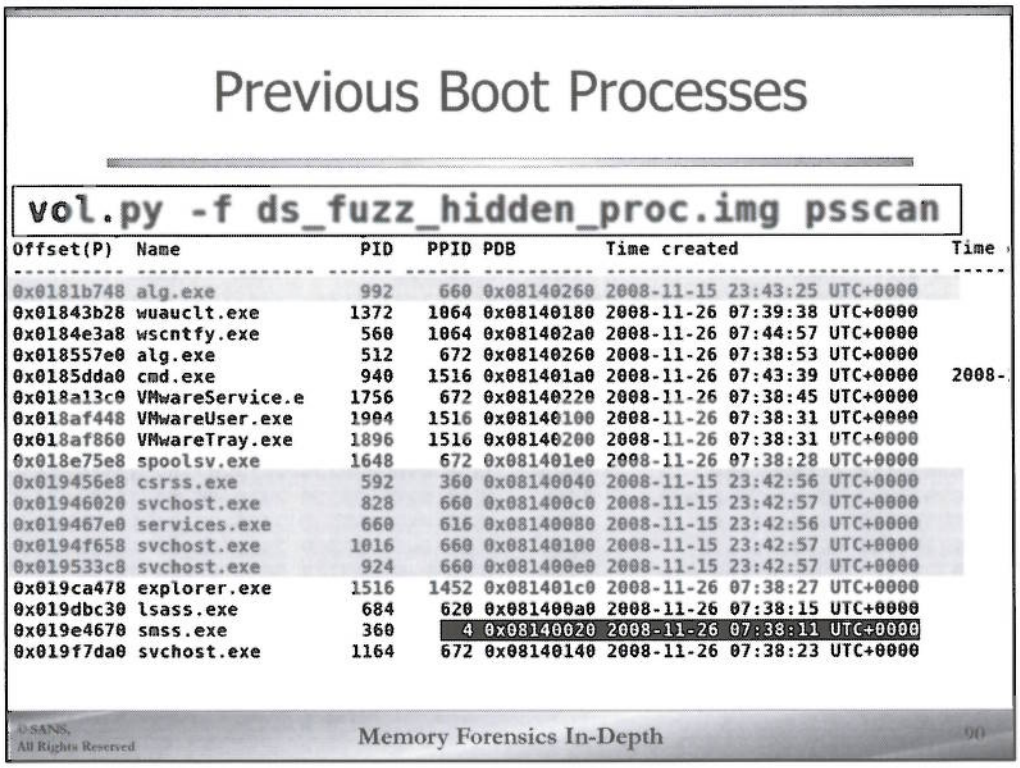
The **psscan** plugin gives investigators another way to enumerate processes within a memory image. Instead of following the doubly-linked list of EPROCESS structures, **psscan** is a brute force scanner, searching the entire physical memory image for potential frames of pool memory that are "tagged" with a specific identifier, or pool tag, that indicates a process pool memory allocation. Once an allocation are identified, based on pattern matching, the **psscan** plugin extracts the process "metadata" from the EPROCESS structure contained within the pool allocation using specific offsets. These offsets vary across different versions of Windows, emphasizing again how important specifying the **--profile** option is when parsing memory with Volatility.



Note that the entries per process in the **psscan** output reference the EPROCESS structures using a physical address offset. This is due to the brute force manner in which the structures are identified, at a physical scanning level of the memory image itself. For the same reason, the entries are not in chronological order, but the order in which they were identified in the scan. Psscan output includes process name, process and parent process identifier, as does pslist. Additionally, each entry shows the physical address of the Page Directory Base (PDB) per process. The PDB is also called the DTB (Directory Table Base), that pointer we referenced earlier when discussing the virtual to physical translation (VtoP) process. If you recall, each process maintains its own indexes that allow for virtual to physical address translation. Maintained in each EPROCESS structure, the PDB is the pointer to the first table used in this lookup process.

The immediate value of running **psscan** in a forensics investigation is that it provides a window into the past, enumerating processes that might have been terminated and de-allocated but not yet overwritten. As we will find in our first exercise, this ability to find evidence that a process once ran on a system is as valuable in some cases as catching it actively running (and therefore in the doubly-linked list) on the system.

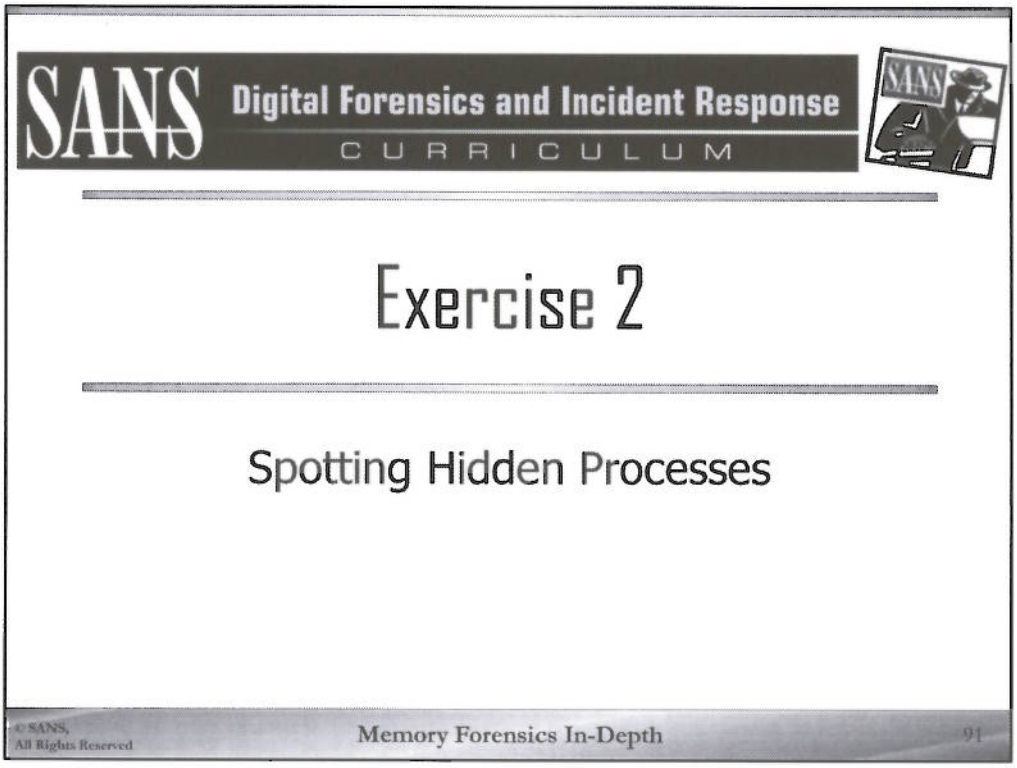
In addition, scanning for process structures in memory, instead of relying on a linked list, gives us the ability to spot an unlinked process. Malicious drivers with access to kernel memory can alter the pointers maintained by EPROCESS blocks, essentially removing a specific process from being enumerated in a process listing because the pointers to it have been altered.



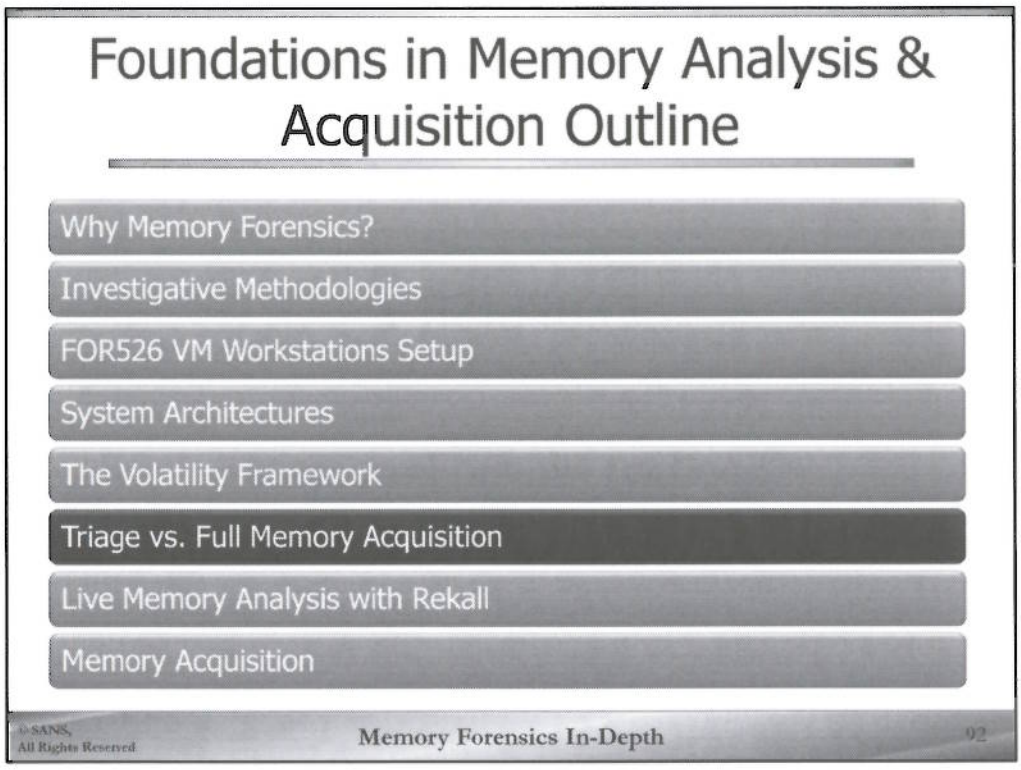
In addition to revealing terminated processes, psscan can reveal EPROCESS block that are leftovers from a previous boot, indicative of processes that were running when the system last rebooted. Although it may sound odd, this happens all the time. Long ago, computer memory was wiped during boot as part of a RAM self test. As users demanded faster boot times, however, the RAM test was generally disabled. As such, when you reboot a computer, nothing is done to wipe its memory. Whatever was in RAM just before rebooting will still be there until it is overwritten. This behavior has been exploited for memory acquisition using what's called the Cold Boot Attack, which we'll get to later.

Two conditions unique to "previous boot" processes that allow an investigator to discern them from terminated or an unlinked, hidden processes are the absence of an "exit time", which is not populated prior to the system rebooting, and the "start time", which will proceed the active **smss.exe** process.

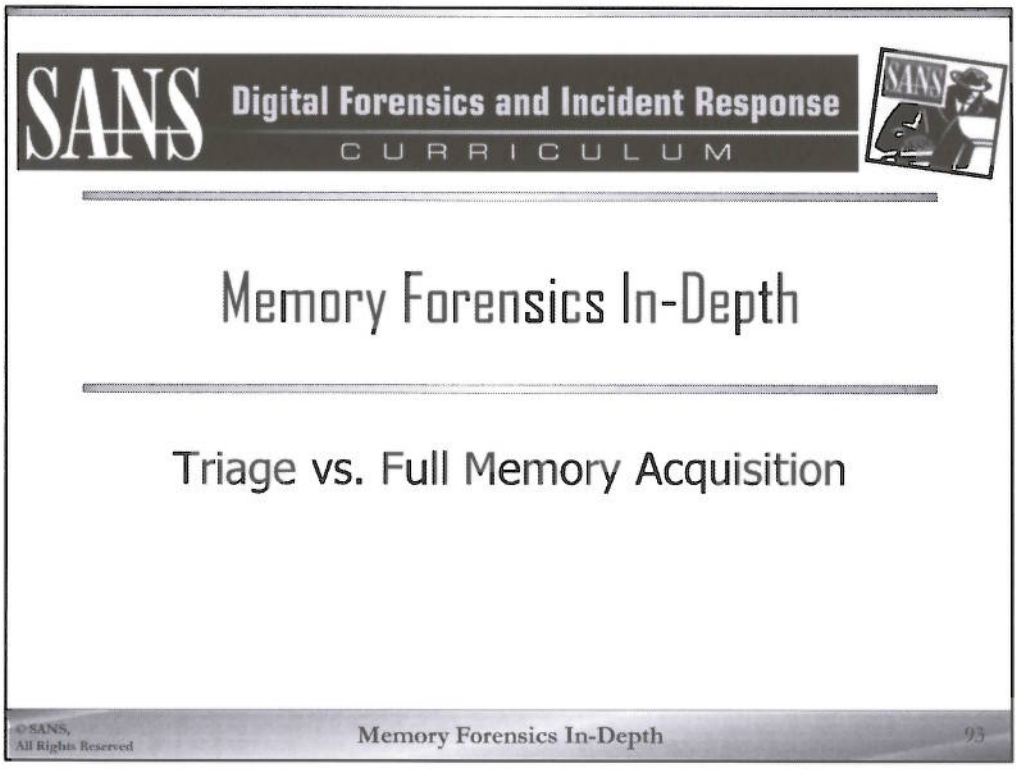
In the output above, you can see that the running **smss.exe** has a start time of 2008-11-26 07:38:11 UTC. There are six processes that psscan found that exist from a previous boots with start time of 2008-11-15. Although they do not have exit times, due to the fact that their start time predates the last boot, they can be considered terminated.



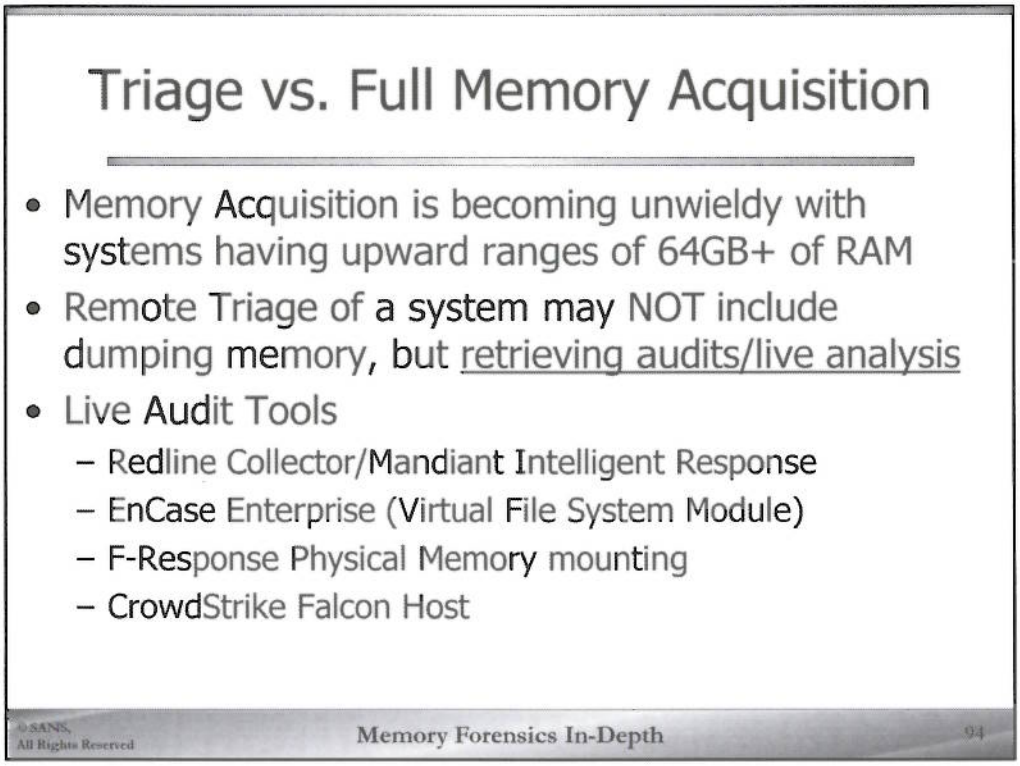
This lab gives us a chance to practice the basic syntax for parsing memory images with the Volatility framework. In addition, we will use the output of **pslist** and **psscan** to spot terminated and potentially hidden, unlinked processes on our target system.



This page intentionally left blank.

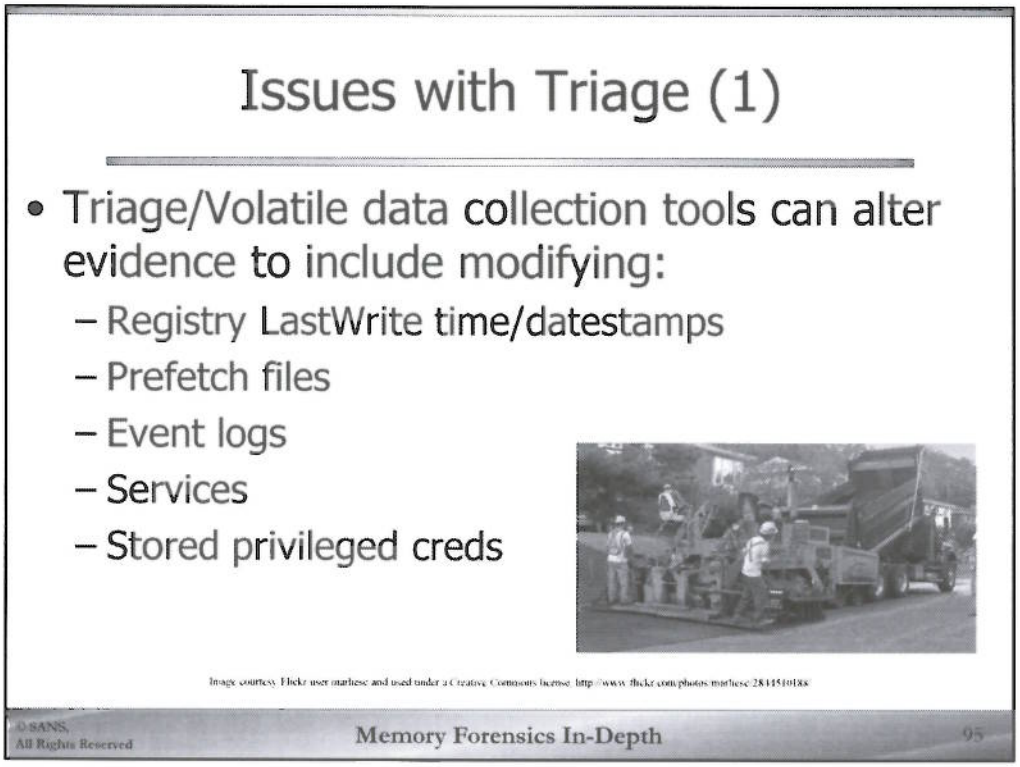


This page intentionally left blank.



In response to the growing use of remote forensic acquisition and analysis tools as well as the increased capacity of RAM on workstations and servers in an enterprise, some response teams are moving towards audit collection instead of capturing physical memory in its entirety. The time required to capture physical memory remotely from a server of interest with over 64 GB of RAM now rivals that of system hard drive acquisitions and may not be the best choice when quick triage/assessment is required. In this case, many security teams will use an auditing tool, typically run on the endpoint as a software agent that reports system state back to a "controller" or collection system. These agents allow for low level access to target systems and allow for live enumeration and analysis of processes, network connections and current system state.

There are several different live response/triage tools currently available in the market. Mandiant's Redline allows for the creation of an audit collector, a one-to-one solution for system triage. Mandiant Intelligent Response is the enterprise solution for remote system assessment and audit collection. EnCase Enterprise allows for remote mounting of a system's physical memory and subsequent analysis of system state. F-Response offers an enterprise tool that uses ISCSI connections to enable a remote mounting of physical memory between an endpoint and an analysis machine. And finally, CrowdStrike now offers an endpoint product, Falcon Host, that monitors system state real-time and reports collection data to a cloud-based controller. With the exception of Redline, all of these live response/triage products are commercial and require licenses for endpoint agent installation.

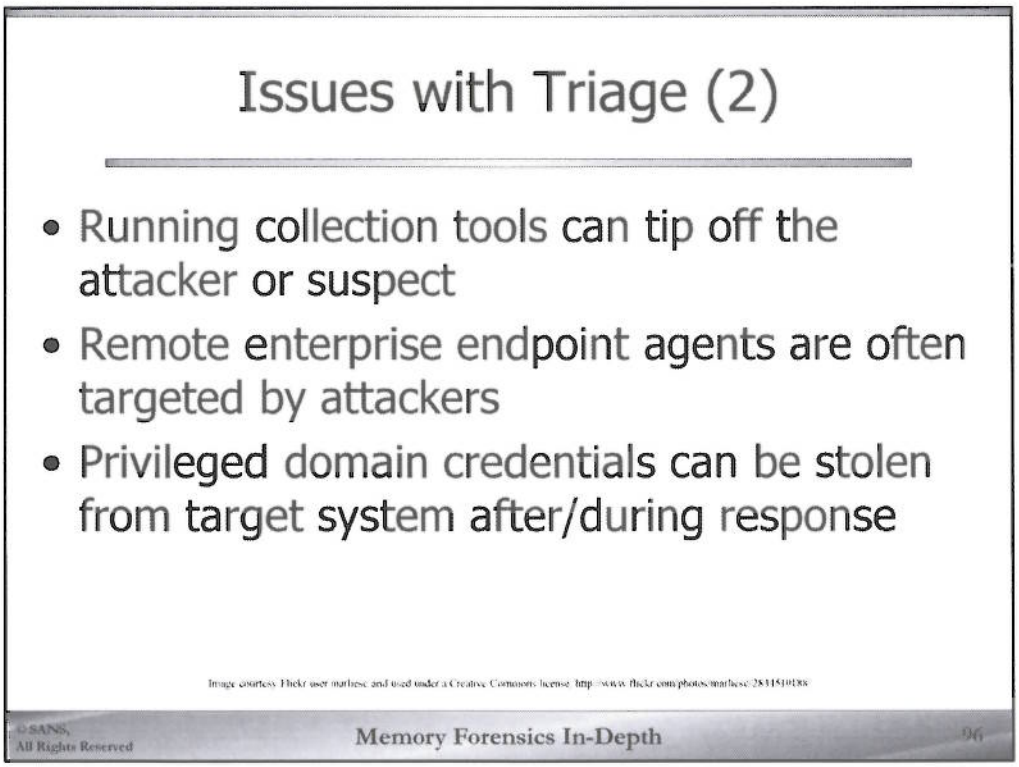


One of the biggest concerns with doing live forensics is the possibility of losing evidence. With the execution of triage/volatile data collection tools, both the memory and the file system of the target system are changed. Some of these changes include the creation of new prefetch files, event logs and services.

This could happen because the investigator's actions overwrite data, a legitimate user's actions overwrite data, or the attacker is alerted to your investigation and deletes the data. The latter could happen automatically or by direct action. In other words, the bad guy could program her tools to automatically delete themselves if they detect you running an incident response tool, or she could, for example, detect your actions in her keystroke capture logs.

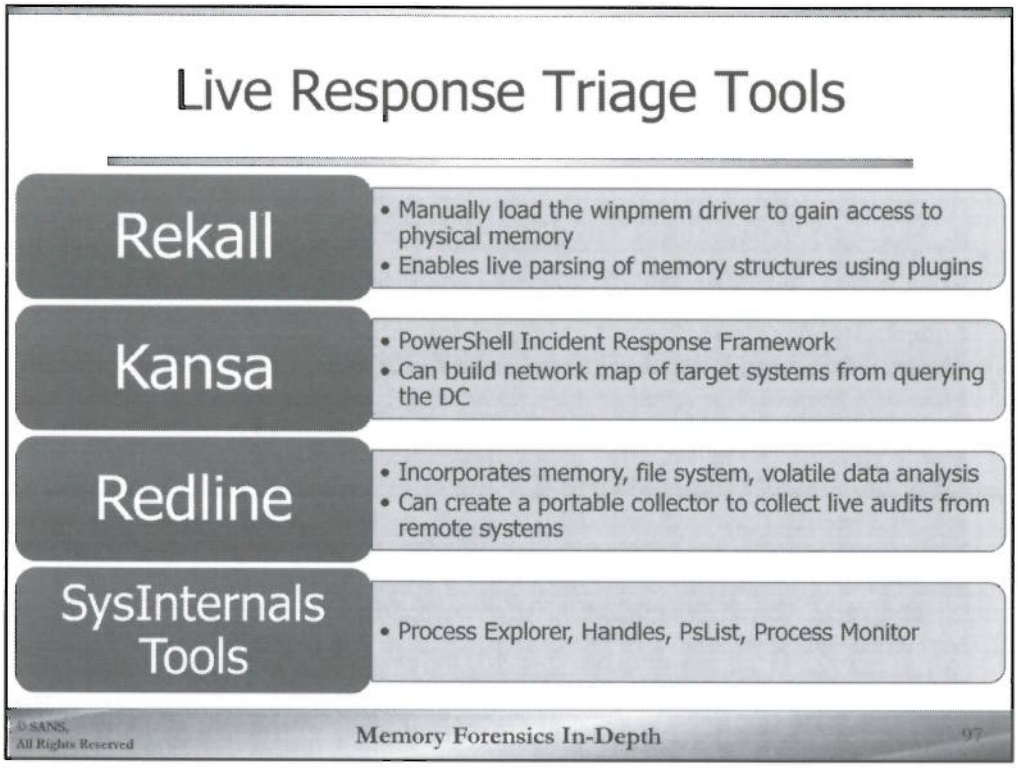
These concerns are mitigated by the fact that without some measure of incident response, there will not be an investigation. That is, until we know which systems have been affected and how they have been affected, there is nothing for us to investigate. We have to do *some* investigation before it's possible to know if a full investigation is warranted.

As such, during live forensics, move slowly and cautiously. Think about and document each command you are going to execute and its effect on the system. It's a good idea to practice live response on your own in a test environment prior to working a real-world incident.



Attackers, especially those exhibiting characteristics of advanced persistent actors, will know a security team's standard operating procedures, to include tools, response time and analysis process, if they have been on the network for some time. Often, the enterprise agents used in many endpoint protection tools are killed or removed from a compromised system. In addition, common volatile data collection tools used by most security teams are easy to identify and can tip an attacker off to the fact that they have been "found out".

Another consideration when performing response concerns privileged credential harvesting. When a privileged user such as an incident responder, makes a remote connection or logs in locally in order to assess a suspect machine, they run the risk having their credentials cached on the system. By default, Windows caches ten domain account credentials on the system, unless otherwise directed through the use of group policy. If these privileged credentials did not exist on the system prior to the response, the attacker may be at a greater advantage by having one of his "owned" systems found out. Privileged domain credentials are commonly used by attackers to laterally move from system to system inside a network.



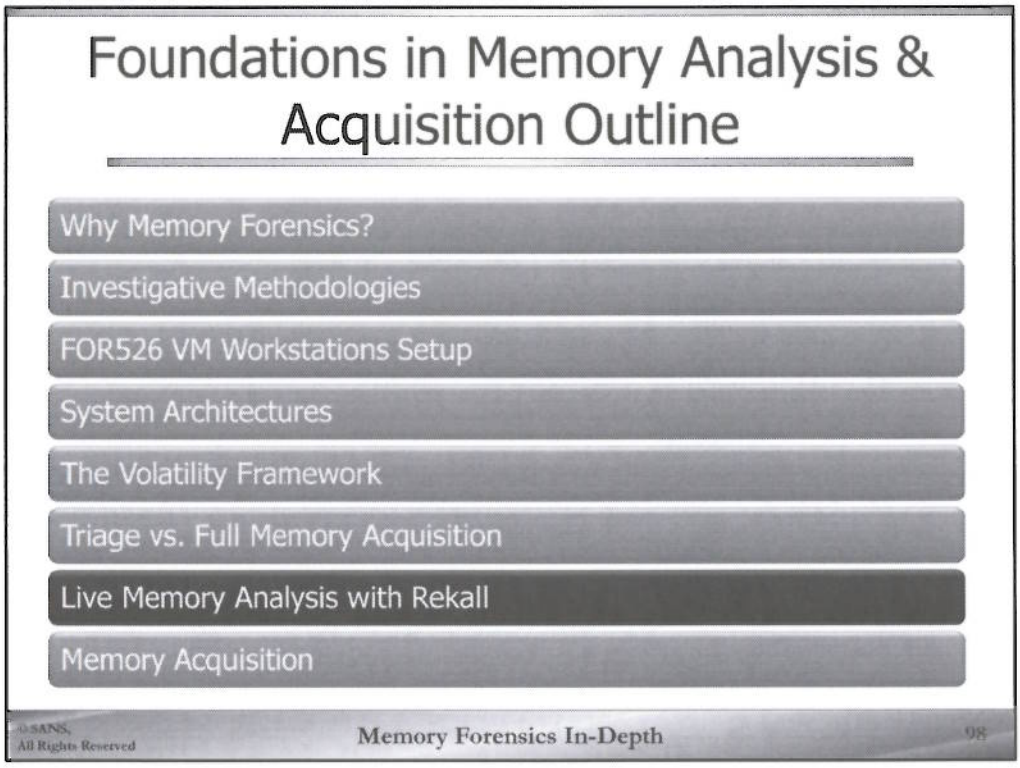
Rekall which will be covered in more depth in the upcoming section is a memory forensic analysis framework. With the use of winpmem, it can perform live analysis and parse Windows/Linux and OSX memory structures on a running system. We are going to be using Rekall as our tool of choice for examining a live system, but it's certainly not the only tool for doing so. In fact there are several tools specifically designed not only for live forensics, but for examining the memory of a running system.

There are other tools which will let you *read* memory. These are just ones which will do some *analysis*. Some other free live memory forensics tools include:

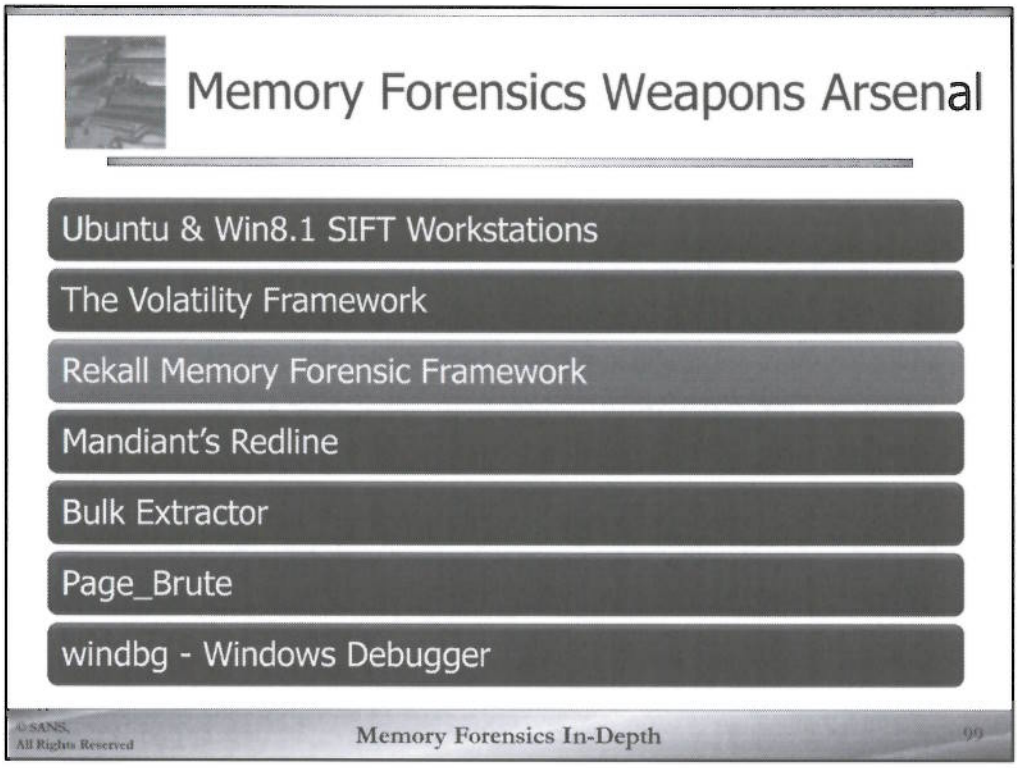
Redline, by Mandiant, conducts examinations of systems and attempts to produce a risk index score for each process. Not only does Redline need to be installed before it can be used, but it requires .NET version 4, which is not installed by default on Windows 7. Installing both programs could overwrite a lot of data.

Kansa developed by Dave Hull (http://trustedsignal.blogspot.com). This framework is a collection of PowerShell commands that interrogate local and/or remote systems, identifying the Domain Controller and building a list of systems in a domain to be queried. It is capable of running examiner-specified modules against up to 32 systems at a time.

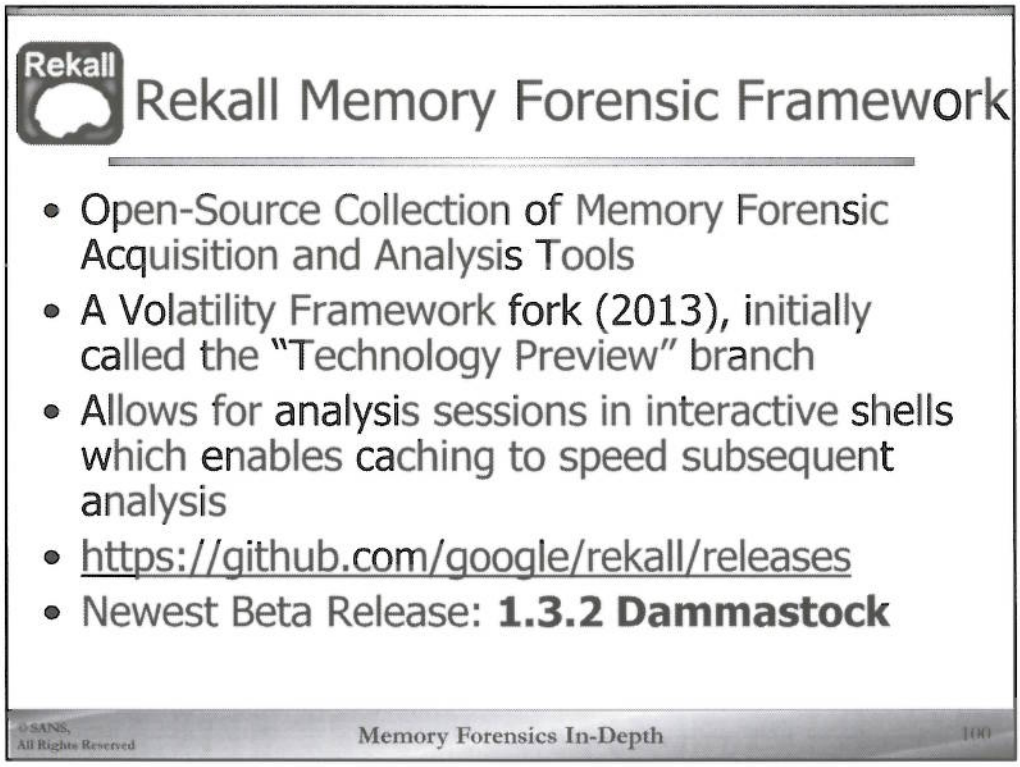
The SysInternals tools suite offers a wide range of tools for live response of a system and behavioral analysis of suspicious binaries. An excellent example of a power SysInternals tool, Process Explorer is much faster than many other live analysis tools. Although it doesn't do as much analysis *automatically*, it can drill down deeply into what a process is doing. The SysInternals blog (http://blogs.technet.com/b/sysinternals/) is an excellent resource as well for walk-throughs of case studies.



This page intentionally left blank.

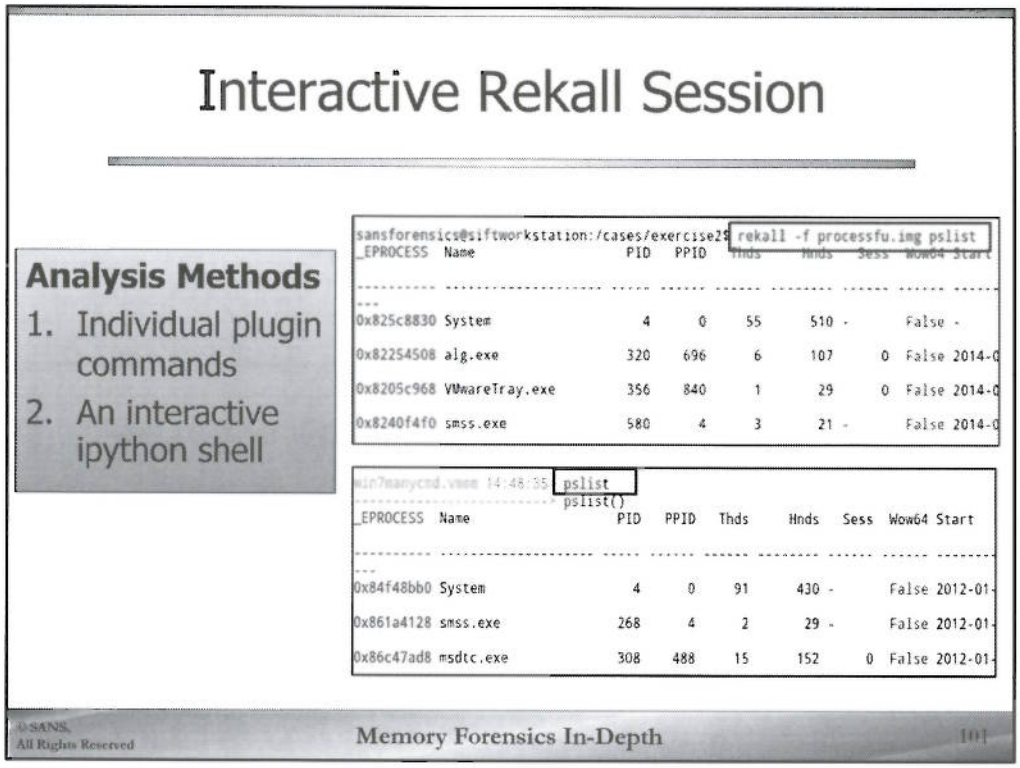


We will now introduce the Rekall Memory Forensic Framework, a fork of the Volatility collection of memory forensics parsing tools. We have installed the framework on both of the virtual machines distributed in this course, the Windows 8.1 and the Ubuntu SIFT. In utilizing various plug-ins, inside of and outside the interactive shell, we will become familiar with the gained efficiency of this tool and its effectiveness as a ***live*** memory analysis tool.



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

Some of the key differences that most analysts notice with Rekall is its ease of use, as it does not require the specification of a target system profile when invoking a plugin. Rekall uses an alternative means of deciphering the profile of the source system other than reading the KDBG (Kernel Debugging Data Block). Rekall uses interactive analysis sessions that caches information in memory, allowing data to remain available for subsequent modules, allowing them to run faster. For example, when running pslist, Rekall caches the processes PIDs, process names, etc. within the image. This allows for fast recall when other plugins referencing process information are run during the same session.



One of the Rekall Memory Forensic Framework's most impressive features is its powerful Interactive shell that takes advantage of caching. Subsequent plugin parsing reuses objects identified by previous plugin runs. This increases speed of analysis for examiners While in the interactive shell, typing "print session" will allow the examiner to see that which has been cached during that specific session.

When running single plugin commands, a cache of a session may be saved off to a file by using the "-s [session_cache_file]" shown here:

```
$ rekal -f fariet1.vmem -s /cases/fariet.cache
```

Keep in mind that for live memory analysis, reuse of objects may not be desirable and plugins can be run individually without caching. . To create a new session inside an interactive shell, type "**snew**" and then "**sswitch [#]**" with [#] being the number of the new session to which you wish to switch.

[1] win7crypto.vmem 13:13:59> **snew**

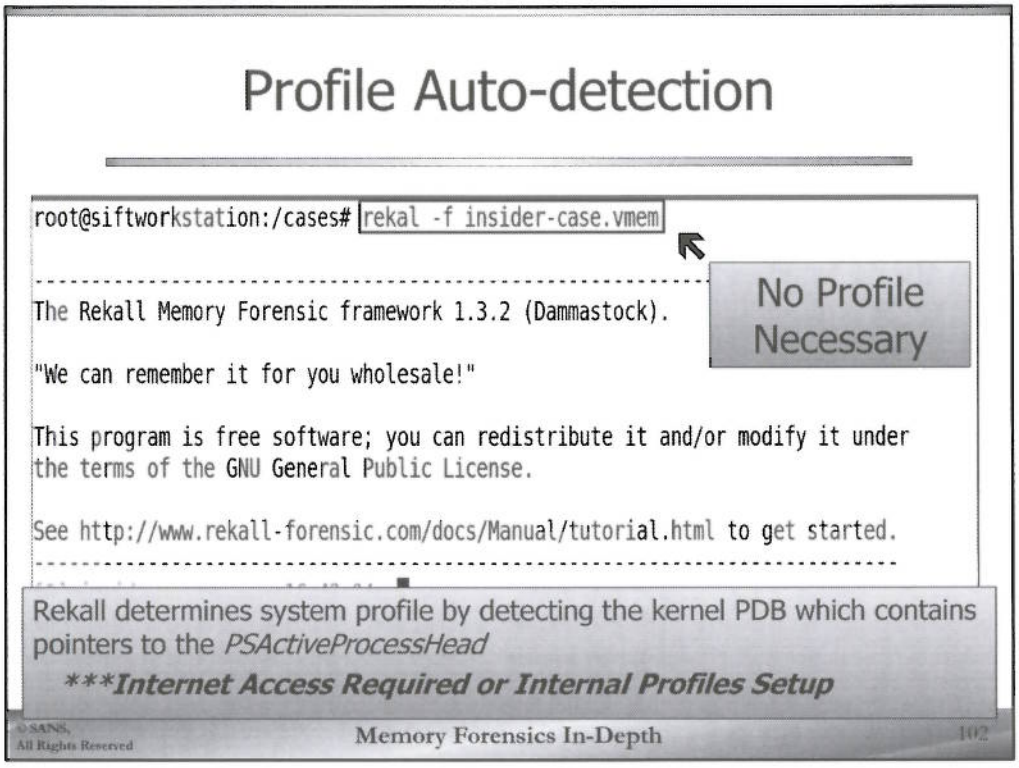
----------> snew()

Created session [2] win7crypto.vmem

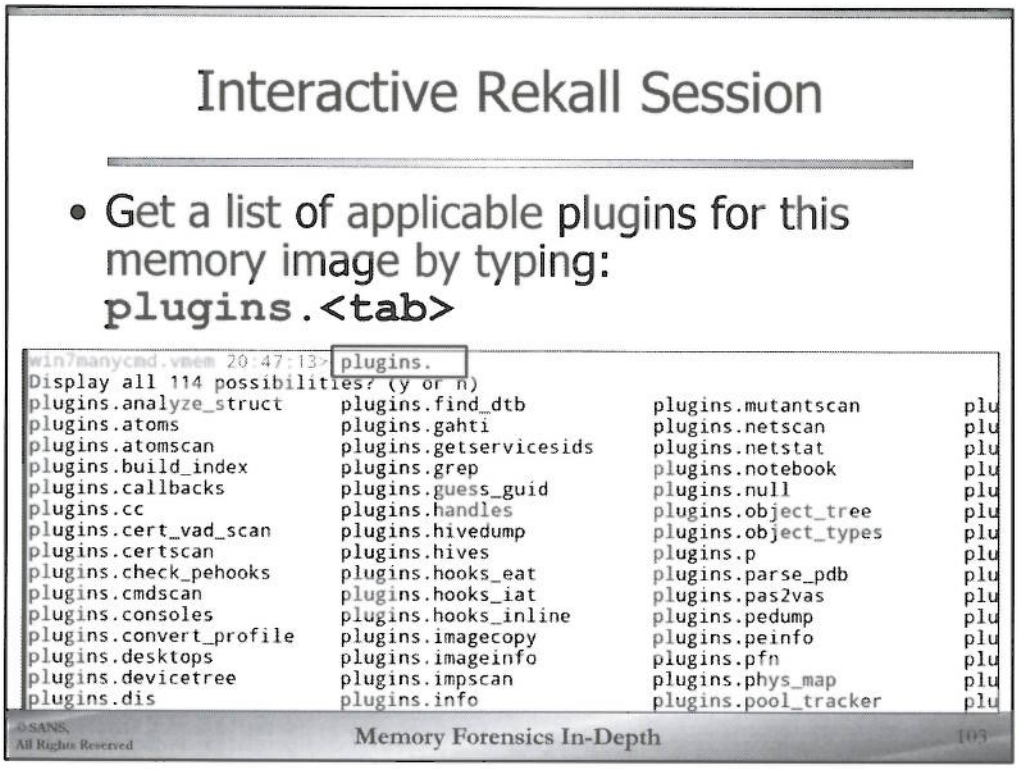
[1] win7crypto.vmem 13:14:57> **sswitch 2**

----------> sswitch(2)

[2] win7crypto.vmem 13:15:20>

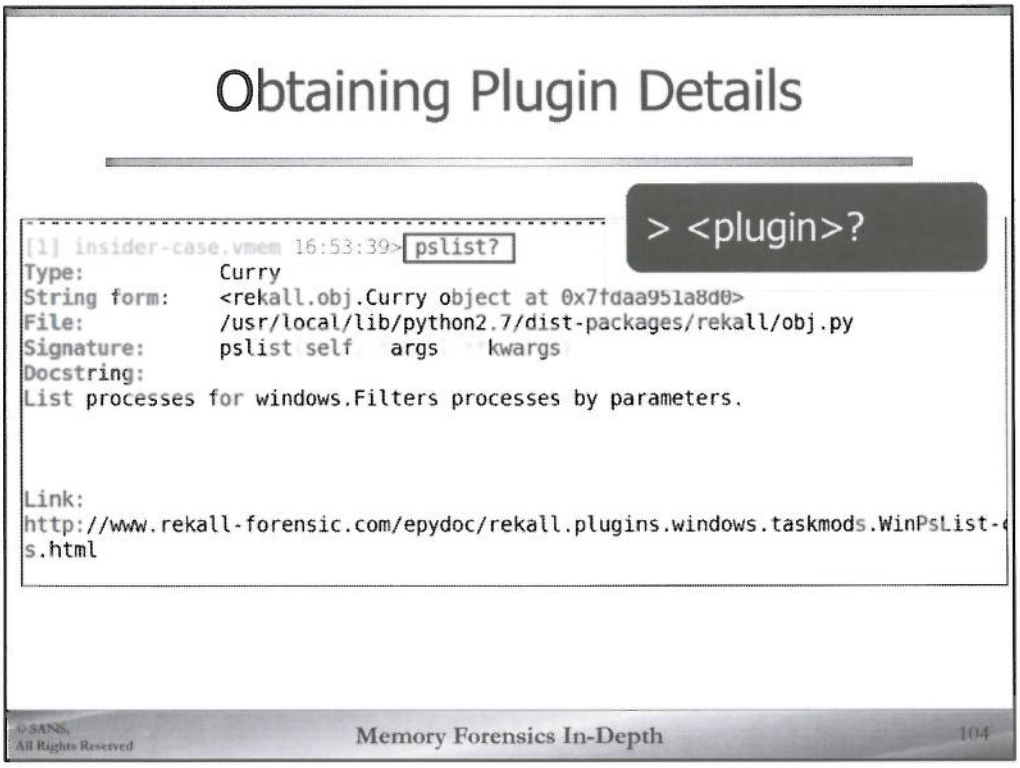


Another notable feature of Rekall is its ability to automatically determine the profile needed in order to parse the target system's memory. After locating the base of the kernel image, Rekall identifies the correct PDB for the loaded kernel using Windows debugging routines, selecting a profile exported from its public profile repository. With the application of the profile specific to the target system, Rekall can then find pointers to PSActiveProcessHead and other structure members without scanning for the KDBG. By default, Rekall references its public repository, located at http://profiles.rekall.googlecode.com/git/. A local profile repository may also be setup on your own system as discussed in this blog post http://www.rekall-forensic.com/posts/2014-02-20-profile-selection.html



When operating in an interactive Shell within Rekall, it is possible to get a list of available plugins that are specific to the system or memory image being analyzed. In order to list applicable plugins, type "plugins." and hit <Tab>.

As you can see in the screenshot above, many of the plugin names are the same of those from Volatility, such as pslist, pscan, and imageinfo. Realize that in most cases, they are using very different methods to parse memory structures than Volatility implements.

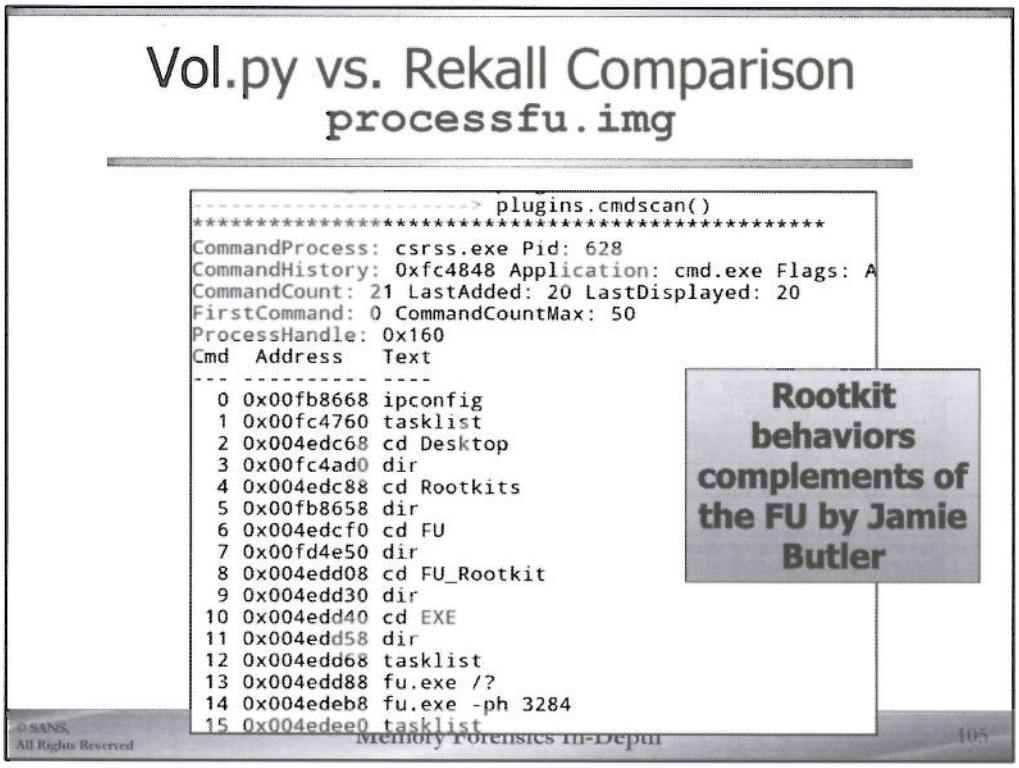


In order to get a brief explanation of a plugin and parameters available for its use, implement one of these two methods for invoking the object description.

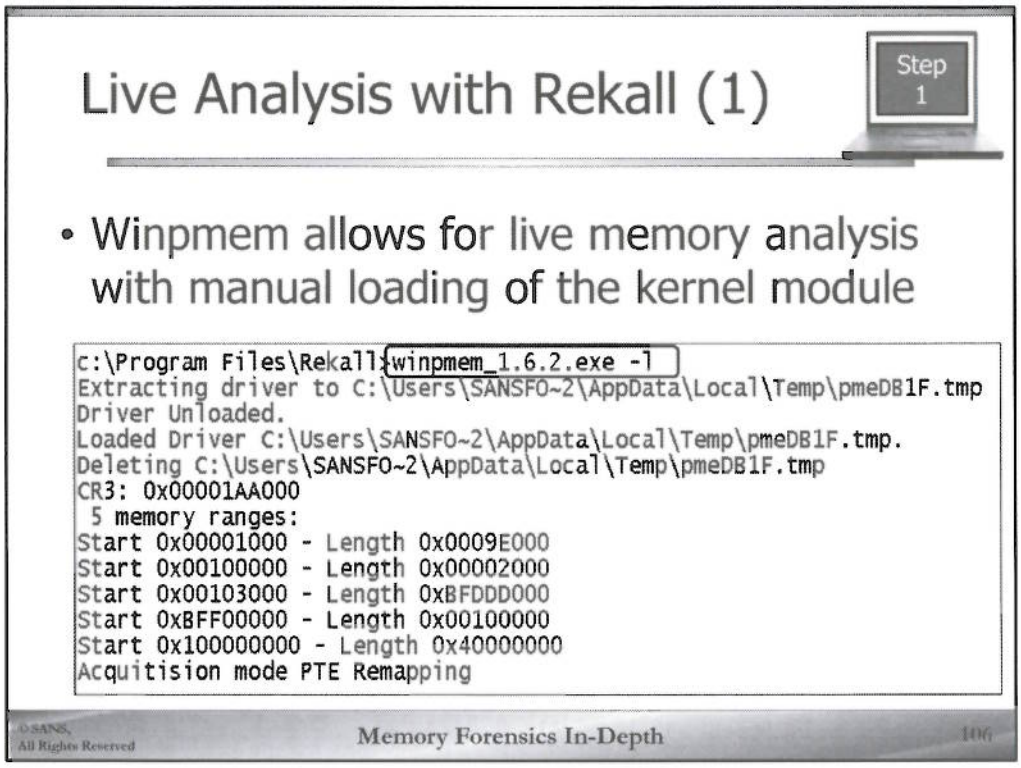
> **<plugin>?**

for the long object description

Most parameters are easily invoked using the following syntax "**dlllist pid 212**" or "**dlllist(pid=212)**" for example.

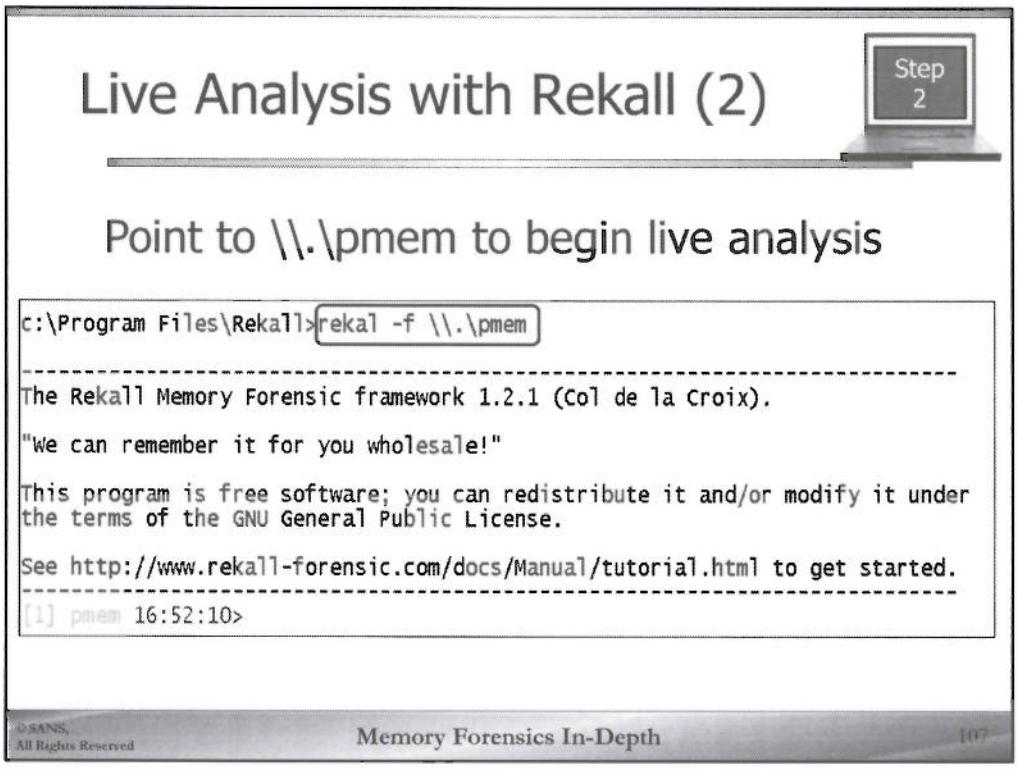


For the hard-charging FOR526 student, we suggest you walk through Exercise 2 using Rekall's pslist (using the default, all 5 methods of process enumeration) and note the differences between tools in detecting the "nc.exe" process. We will be comparing the differing techniques used by these two tools in the next section of this course, but taking the time to experience the differences yourself builds muscle memory for the Day 6 NetWars Tournament.

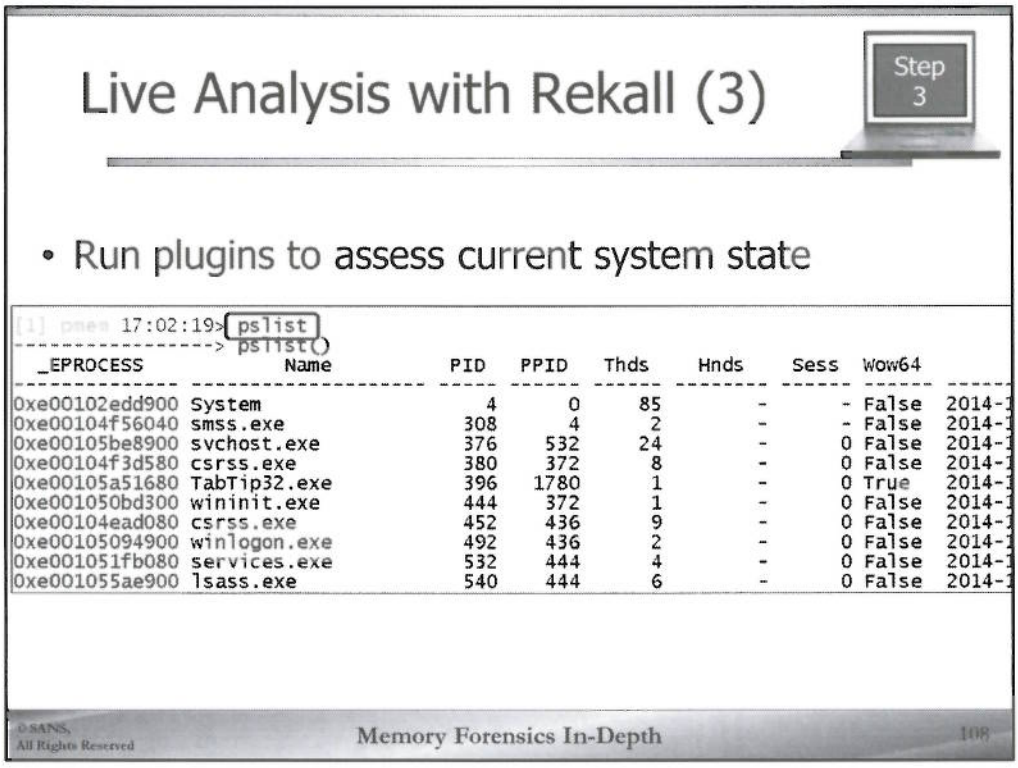


In the next exercise, we will be using winpmem and Rekall to conduct LIVE system memory analysis on our target system (the Windows 8.1 VM). This is very exciting for various reasons - up until the summer of 2014, an examiner's ability to analyze Windows 8/8.1 memory was limited. The primary open-source memory forensic framework was unable to parse any of the linked lists of structures in memory images created from these systems. Another groundbreaking development that we will be enjoying is the ability to analyze live system memory using the same powerful "detailed" view plugins that we just used against the processfu.img memory image. No memory acquisition necessary!

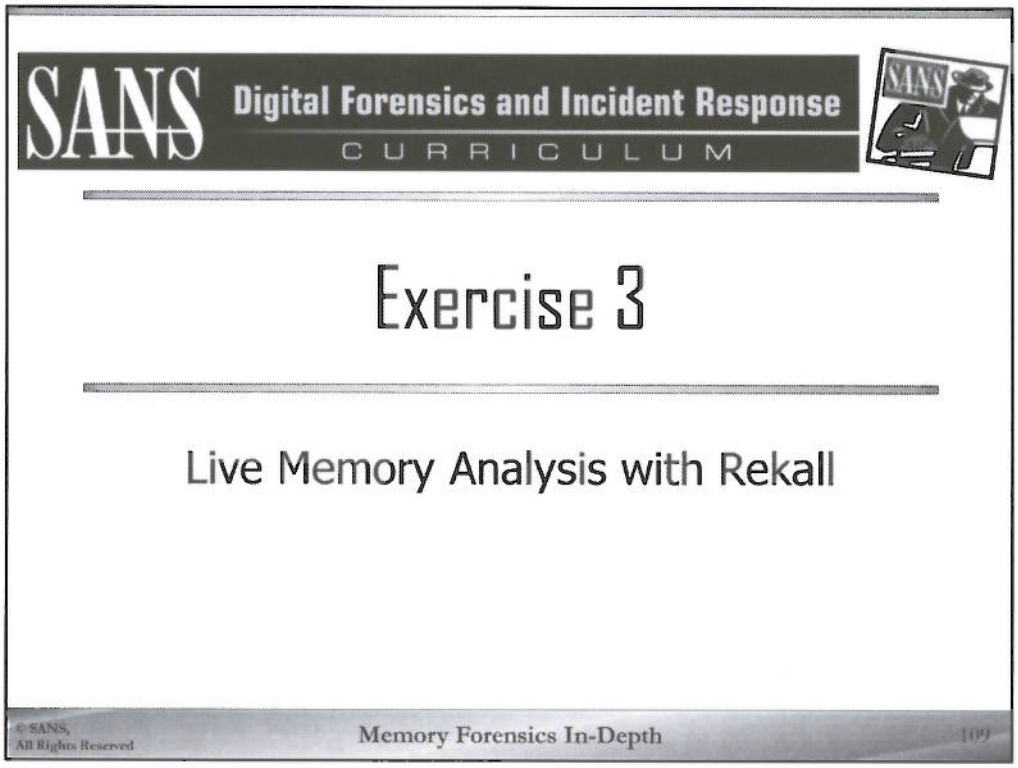
The first step in achieving success in Exercise 3 will be to manually load the winpmem driver on your target Windows system, as shown above.



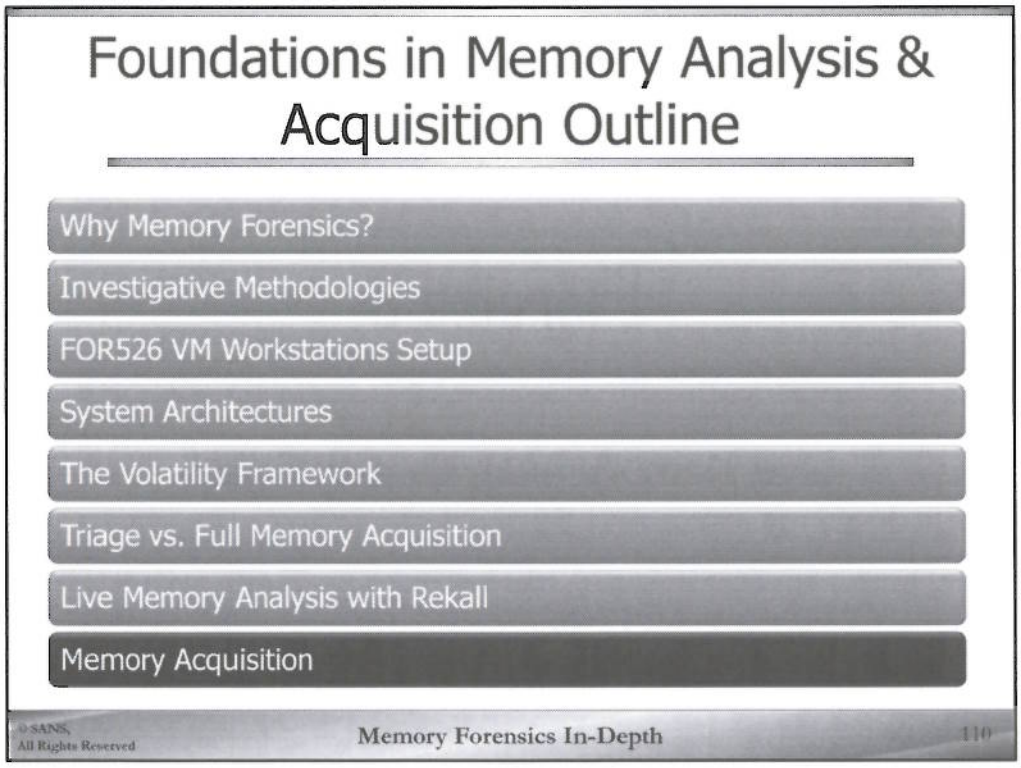
Once the pmem kernel driver is loaded into the memory of our target system, we are able to create an interactive session with Rekall Memory Forensic Framework. By referencing **\\.\pmem**, a device file in the Global Object Table, Rekall gains access to the functionality of the winpmem driver, which subsequently calls Memory Management Windows APIs.



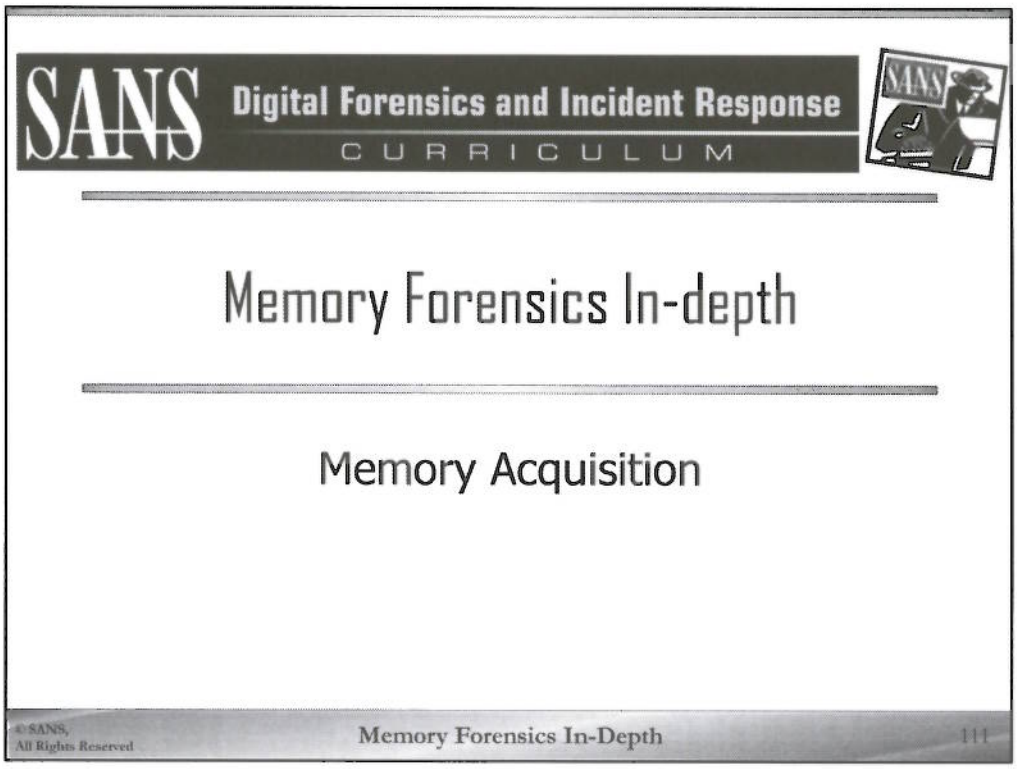
Within our interactive session, we can queries system memory and assess current system state. Again, beware of session caching's ability to "reuse" past plugin output. If a new process starts after the examiner has run a process enumeration plugin, the cached information is likely to be used. A possible solution to this, if a new captured of system state is desired, you may create a new session inside an interactive shell by typing "snew" and then "sswitch [#]" with [#] being the number of the new session to which you wish to switch.



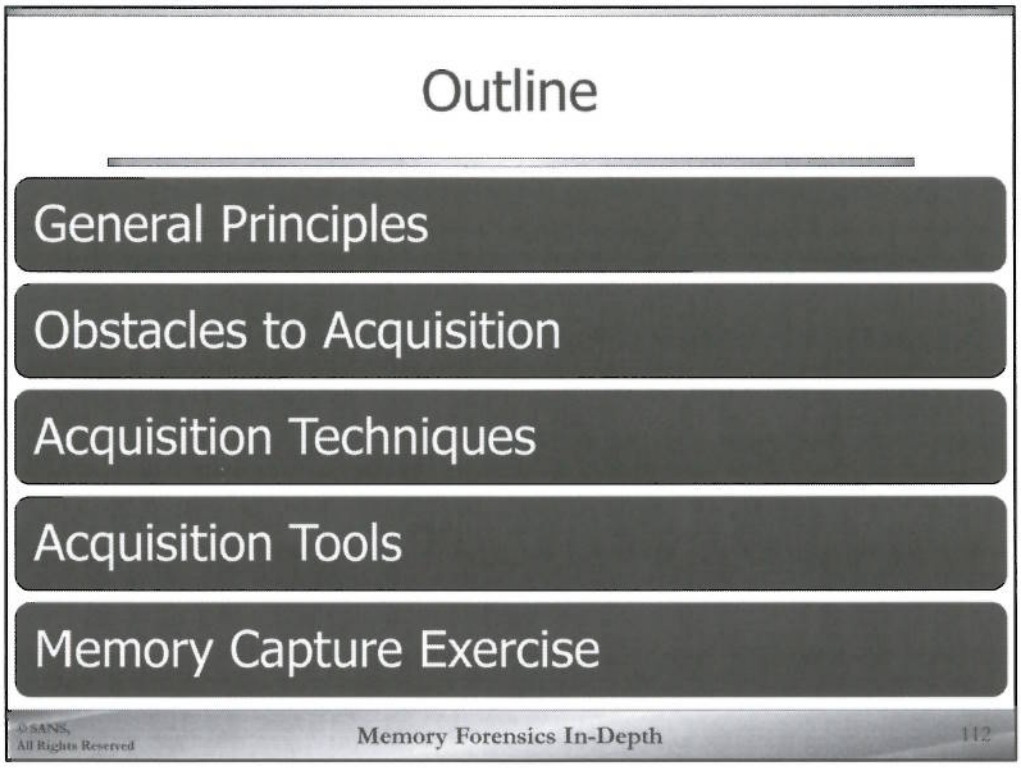
This page intentionally left blank.



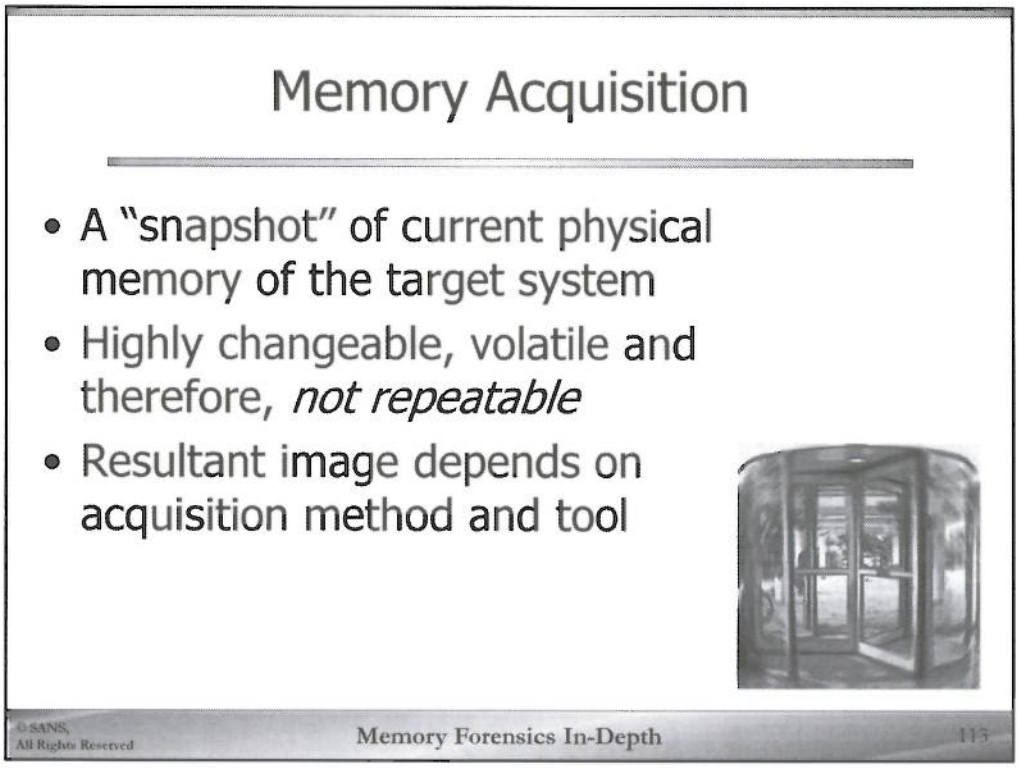
This page intentionally left blank.



This page intentionally left blank.



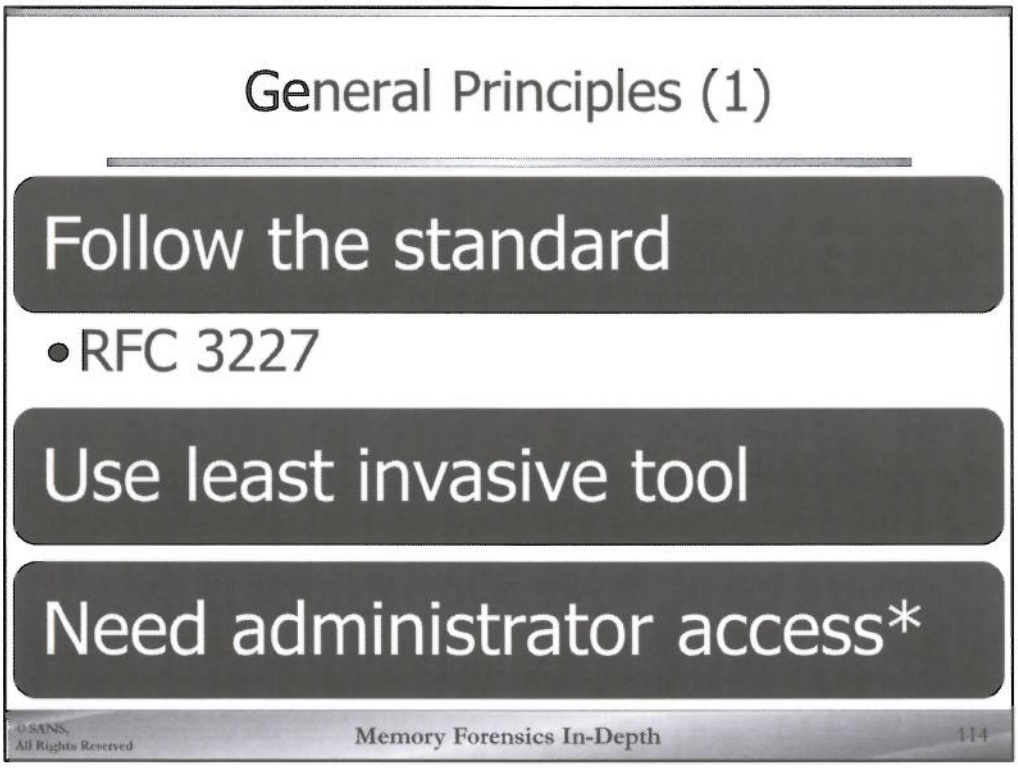
This page intentionally left blank.



Before we can do any memory forensics and analysis techniques, we have to acquire a memory image. That is, we have to have a sample of the computer's memory to work with. Imaging memory is not like imaging an unmounted disk drive. It is not a repeatable process. If you acquire the memory of a system twice, you will get vastly different results. This is because unlike a disk drive, the memory on a system is changing during the imaging process and is changed by the imaging process. Further, because of how some of physical memory is mapped to devices connected to the system, memory imaging can be tricky business. These mapped regions of memory must be avoided. Failure to avoid I/O mapped memory, as it's called, can hang the imager or crash the system.

As a result, there is even a debate about what to call the data which is acquired by such processes. Is it an "image"? A "sample"? There is no consensus in the forensics community. Those who favor the term "sample" feel that because the process is not repeatable the result should not be called an "image".

In this section we will describe some of methods for capturing memory, starting with the simple and moving into the more exotic. Although not useful in every situation, the exotic methods may be the best (or only) method for particular circumstances.



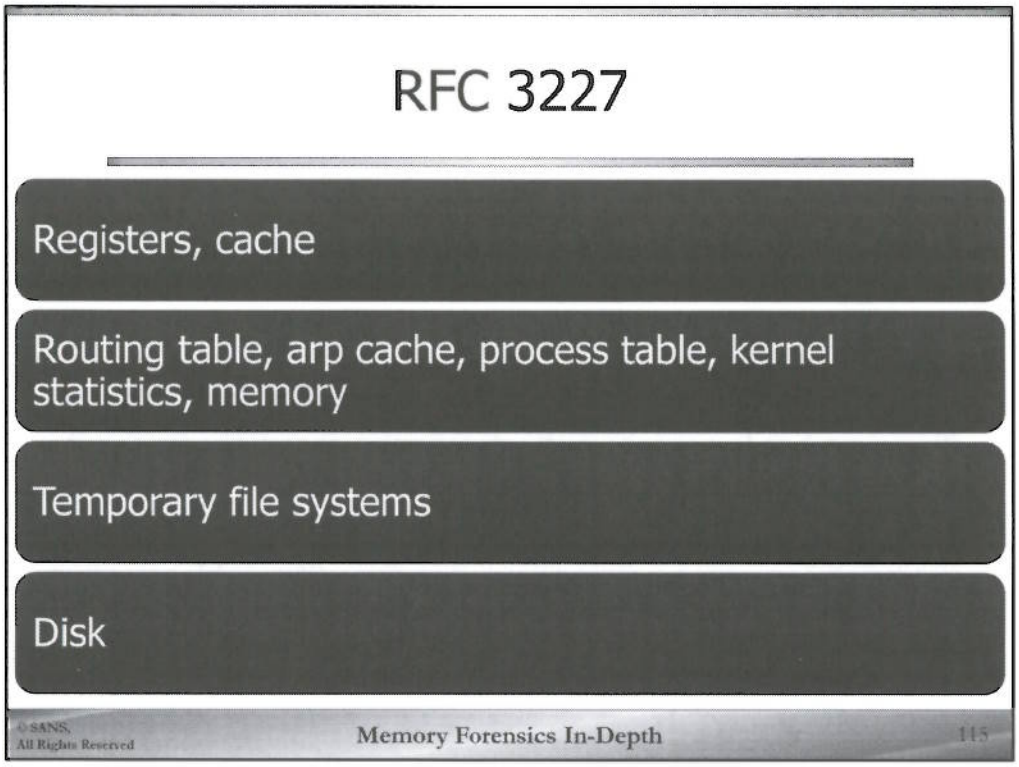
There is a standard for the order in which to gather data during incident responses. RFC 3227, "Guidelines for Evidence Collection and Archiving". If you're going to do something other than what the RFC recommends, you'd better have a good reason and be able to back it up in court. That's not to say there aren't times to violate the RFC, but you must be able to explain why you did so.

Acquiring the memory of a potential victim computer should be the first thing done during incident response. Anything else you are going to do will change the state of memory and potentially damage the very evidence which could help you the most. The programs you run or commands you execute in the name of incident response will change the state of the machine. We want to see the actions of the bad guys (and the good guys who touched the system before you), not evidence of our own actions.

There are many ways to capture the memory. You should always use the least invasive method to get what you need. We are going to discuss several methods, all of which will result in a memory image. But the "best" method will depend on what you are trying to get and how much disruption you can tolerate on the system.

Generally, you will need Administrator access to the machine in question*. Accessing the memory of a system is, thankfully, limited to the Administrator. Microsoft has even further restricted such access to drivers. Userland programs can't open memory at all. (They can read it, but can't open the object necessary to do so.) So memory acquisition tools generally rely on drivers to get their work done. And loading a driver requires Administrative privileges.

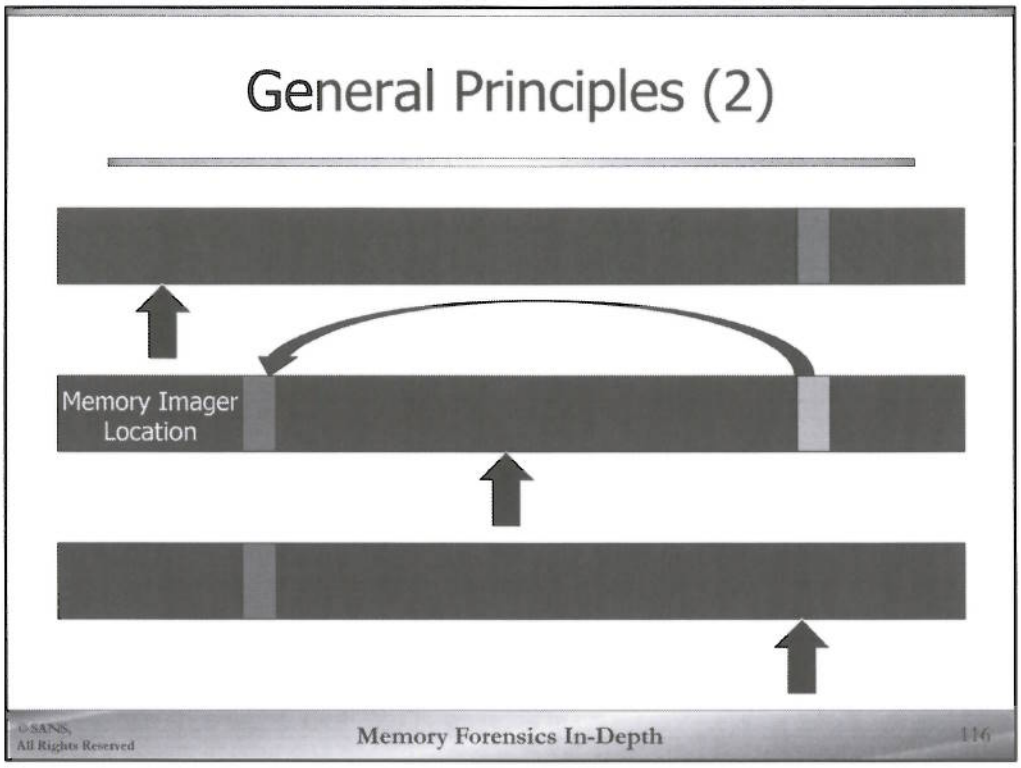
* There are two methods, DMA and the Cold Boot method, which don't require Administrative rights. But these methods aren't perfect and are disruptive in other ways. Again, you can use them in your investigations, but you must be able to justify why you didn't use a less disruptive method.



You can (and should) take a look at the standard for volatile evidence collection, RFC 3227. It's online in lots of places, including http://www.ietf.org/rfc/rfc3227.txt. The standard includes some general guidelines for how to do incident response, and then lists the order in which data should be collected. The first thing to gather would be the state of the processor, namely what's in the register and the cache (e.g., translation look-aside buffer). We generally do not get down to that level in traditional incident response. The current instruction being processed and the exact state of the processor are generally too fine grained to be of significant evidentiary value during an investigation. Whether there was malicious software on the computer or not, the current instruction being processed would have little or no bearing on the matter.

The next category of data specified in the RFC is the volatile data we are most interested in. The standard breaks out these data into categories such as the networking state and process state, but lists the computer's memory too. We have shown so far in this course that grabbing the entirety of memory is sufficient to encompass all of the other pieces of information in this category. In fact, it would probably be more disruptive to the system to worry about grabbing the routing table specifically than it would be to just grab all of memory.

The RFC then specifies the capture of temporary file systems, such as RAM disks, and then actual file systems. After that are data which are more commonly discussed in other training courses, such as remote logs, configuration and topology information, and so on.

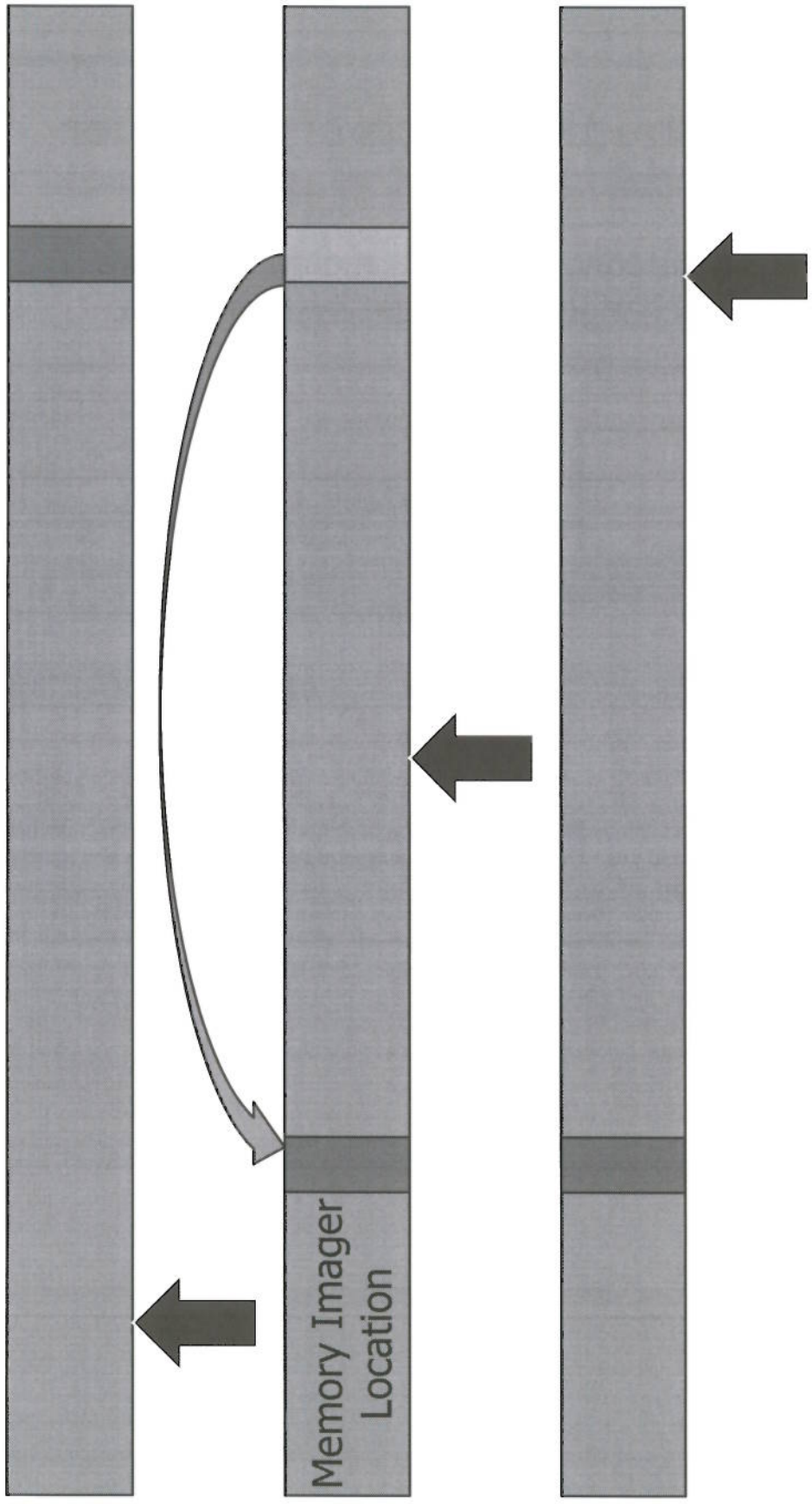


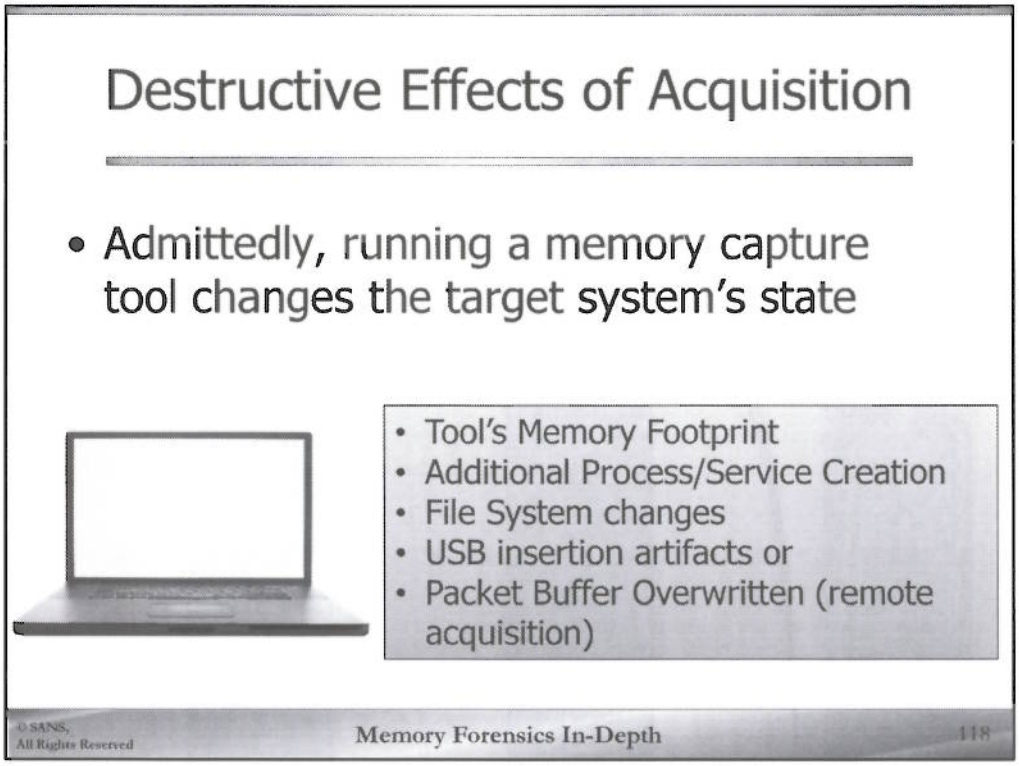
Memory imaging is not the same kind of process as disk imaging. When imaging a disk, the examiner can read every byte of the drive and write those bytes to a file. If desired, they or somebody can repeat this process months or even years later. The drive has been properly stored as evidence; any subsequent imaging will produce exactly the same result as before.

Computer memory, on the other hand, obeys no such constraints. The contents of memory are constantly changing. Because the software to capture a memory image is executing on the system in question, and thus being stored in the very memory it is attempting to image, the memory is being changed during the acquisition process, and by the acquisition process. The result is not repeatable. If you run a memory imager twice, on the same system, in short succession, you will end up with two different images. It is impossible to know if your memory imaging program worked "correctly". Because it's impossible to repeat a memory imaging process with two tools, or even the same tool run twice, it is difficult to evaluate the correctness of such products.

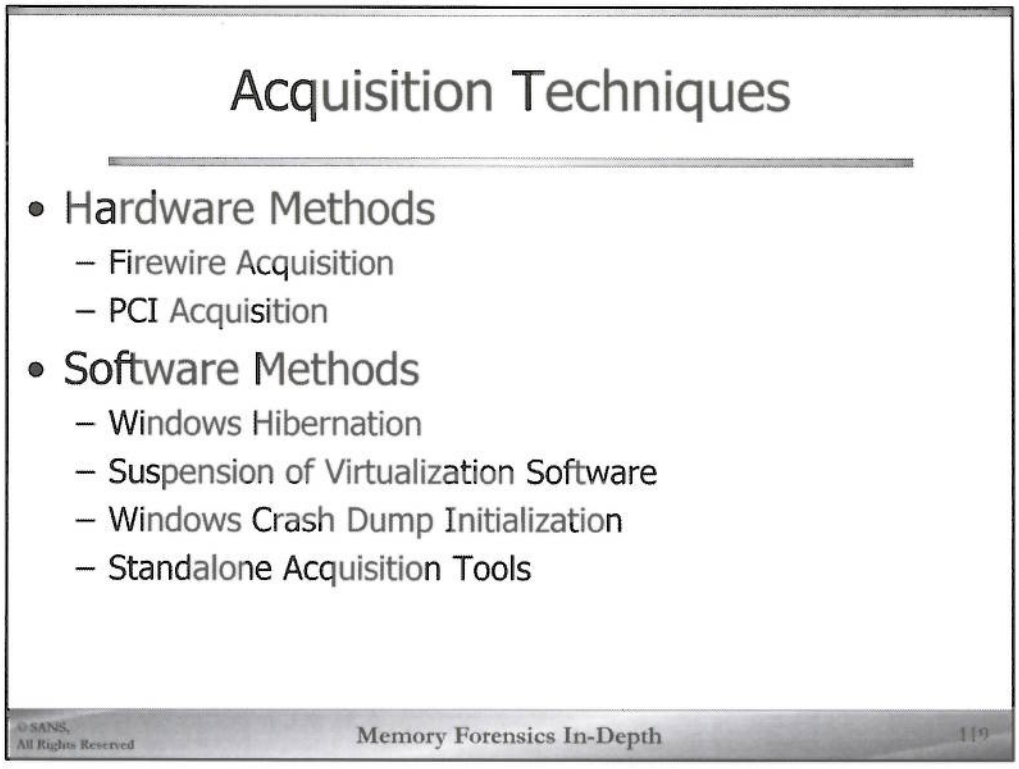
Even the term "memory image" is debated. Some examiners think the term "image" should be reserved for disk images, or any media which can be captured via a repeatable process. Other terms for the result of a memory capture operation include "capture", "sample", and "snapshot".

If anything, the contents of a memory image are a blurry snapshot. Data could quite legitimately not appear in a memory image even though it was stored in memory during the acquisition. For example, let's say there was some datum which was stored in the high end of memory. While the memory imager was running, this datum was moved to a low position in memory—specifically a portion of memory which the imaging tool had already read. The datum in question would never actually be read by the acquisition tool.

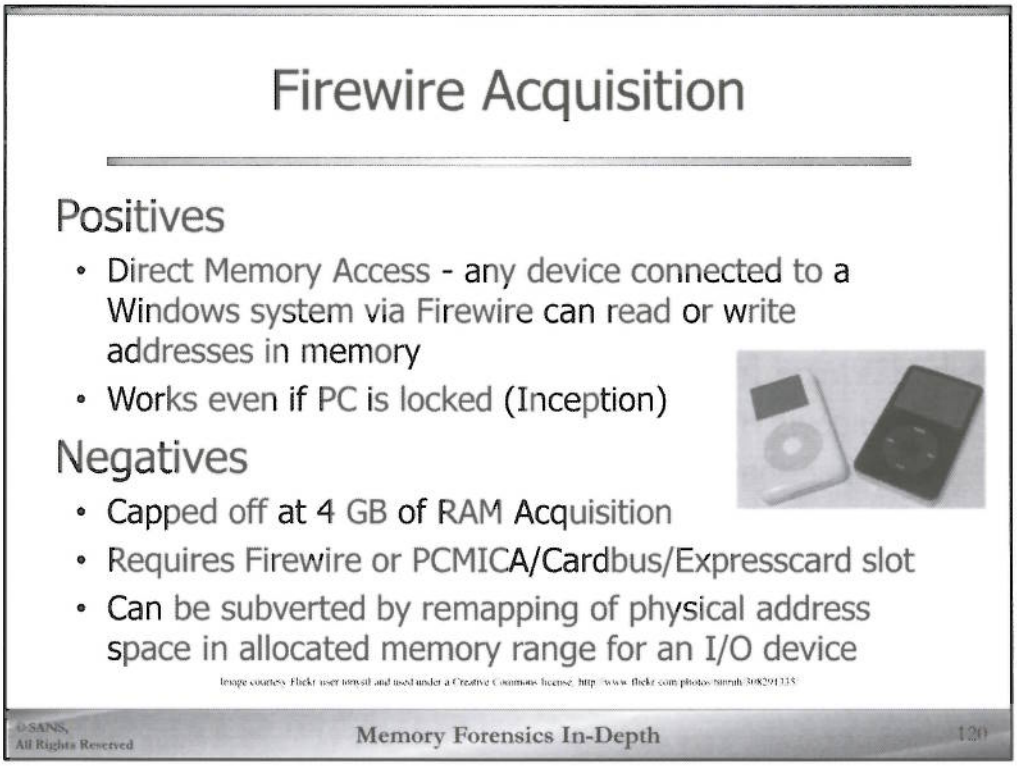

Memory Imager
Location



You should, of course, test out all of your forensics tools before using them in the field; but be especially vigilant with memory acquisition tools. If your lab tool fails to work, you (hopefully) were working on a copy of the original evidence and didn't lose anything. But if your acquisition tool fails, you will not have *any* evidence upon which to build a case! Knowing how your acquisition tool works and how it affects the target system, is essential in proper incident response and well as evidence collection. Of course, the mere act of connecting to a system to conduct triage creates changes on that system in memory and the file system, whether it be a local connection or remotely over the network. Changes to the registry based on USB insertion or to the target's event logs based on account logon event creation can be significant in recreating the story of initial vector of compromise and subsequent attacker activity on a system. Therefore, a team's standard operating procedures for Incident Response and Volatile Evidence Collection should consider the ramifications of such changes and weigh the advantages and disadvantages of collection methods and tools.



This page intentionally left blank.

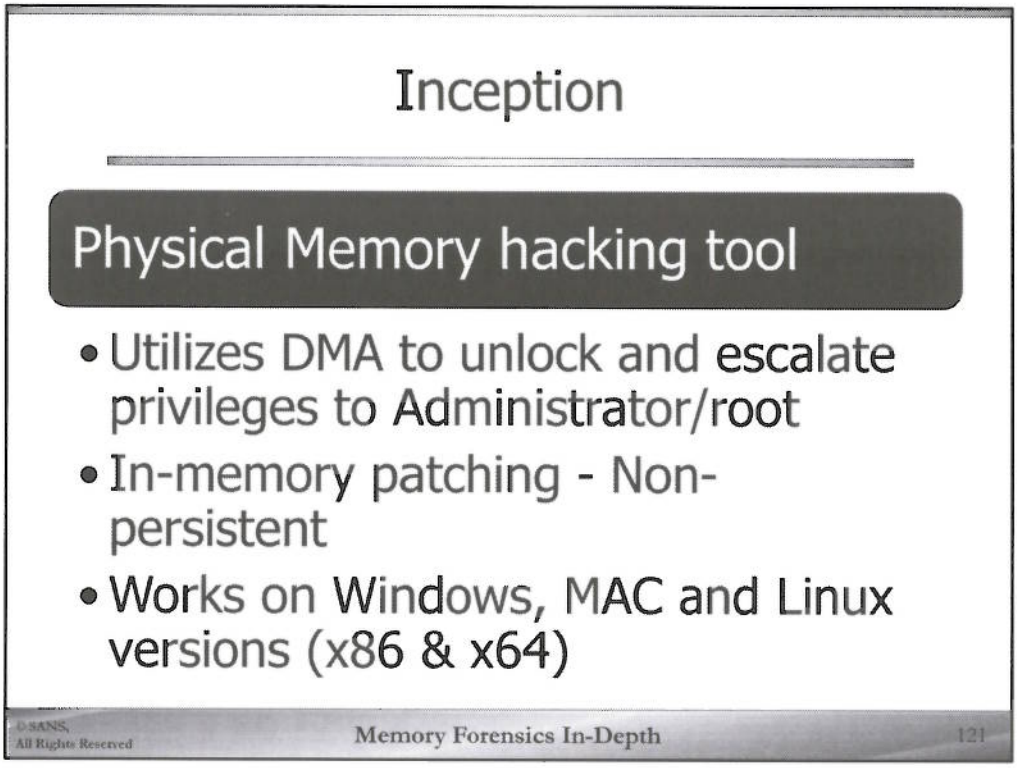


The "fire" in Firewire is Direct Memory Access. When the protocol was developed, the goal was to allow Firewire devices to read and write directly to main memory without going through the operating system. This way, a Firewire disk drive could load data into main memory while the processor works on other tasks. Windows capitalized on this, for example, by allowing some devices unrestricted access to main memory. For example, iPods are allowed to read and write any address in memory for performance reasons. Users who want to sync their iPod to their computer want it to happen as fast as possible.

The upshot of allowing DMA, however, is that any device connected to a Windows system via Firewire can read or write addresses in memory. Although described as far back as 2002, papers and tools were published in 2005 which allowed anybody to access the memory of a Windows computer using a Linux system via Firewire. The system was configured to identify itself to Windows as an iPod. It could then make arbitrary changes to memory or read any part of memory. Along with memory capture, there was also a tool which altered memory to bypass the screen saver protection. When the screensaver is locked, normally access is only granted if the typed password matches the actual password. But a one-byte change via Firewire made it so that when the user entered a password, access was only granted if the typed password didn't match the actual password.

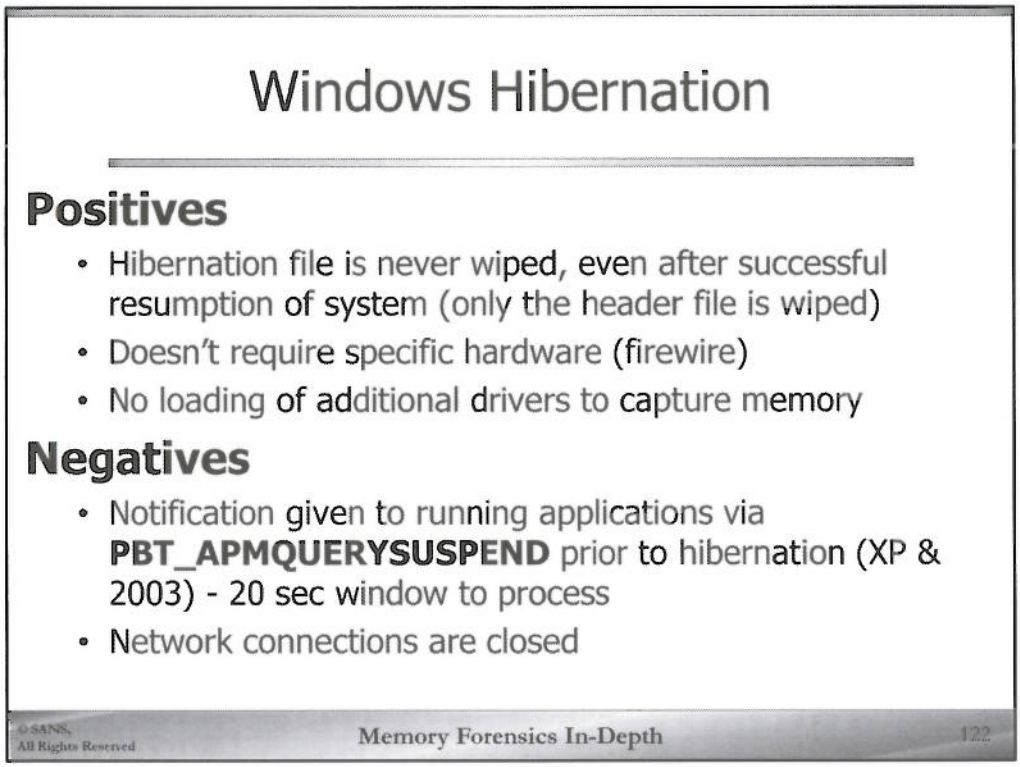
There are some problems with using Firewire for memory acquisition, however. First, it isn't found on many systems today as Firewire has fallen out of fashion largely due to the faster speeds offered by USB 3.0. Second, the Firewire address bus is only 32-bits wide. Thus when a Firewire device requests to read memory from the system, it is limited to 32-bit addresses, or the lower 4GB of physical RAM. Any memory in excess of 4GB is inaccessible by Firewire devices. Third and finally, as noted above with I/O memory, attempting to access an I/O mapped address could crash the system.

You can still buy Firewire based imaging tools. For example, Passware sells one, http://www.lostpassword.com/hdd-decryption.htm#imager.



One tool that makes use of Firewire DMA capabilities is Inception, a physical memory hacking tool. This tool has many use cases to include unlocking and escalate privileges to allow a user Admin/root access to the target system. It can work over FireWire, Thunderbolt, ExpressCard, PC Card and any other PCI/PCIe interfaces. Most notably, this tool can identify password based on signature search and override authentication mechanisms resident in memory to allow ***any*** password to be used to grant access. Inception also has memory dumping capabilities and can be used in "pickpocket" mode that dumps memory upon connection to a locked system, something forensics examiners and incident responders encounter commonly. According to the developer's website, it will run on Windows 8 SP0, Windows 7 SP0-1, Vista SP0 and SP2, Windows XP SP2-3, Mac OS X Snow Leopard, Lion and Mountain Lion, Ubuntu 11.04, 11.10, 12.04, 12.10, Linux Mint 11, 12 and 13 x86 and x64-bit machines. [1]

[1] http://www.breaknenter.org/projects/inception/



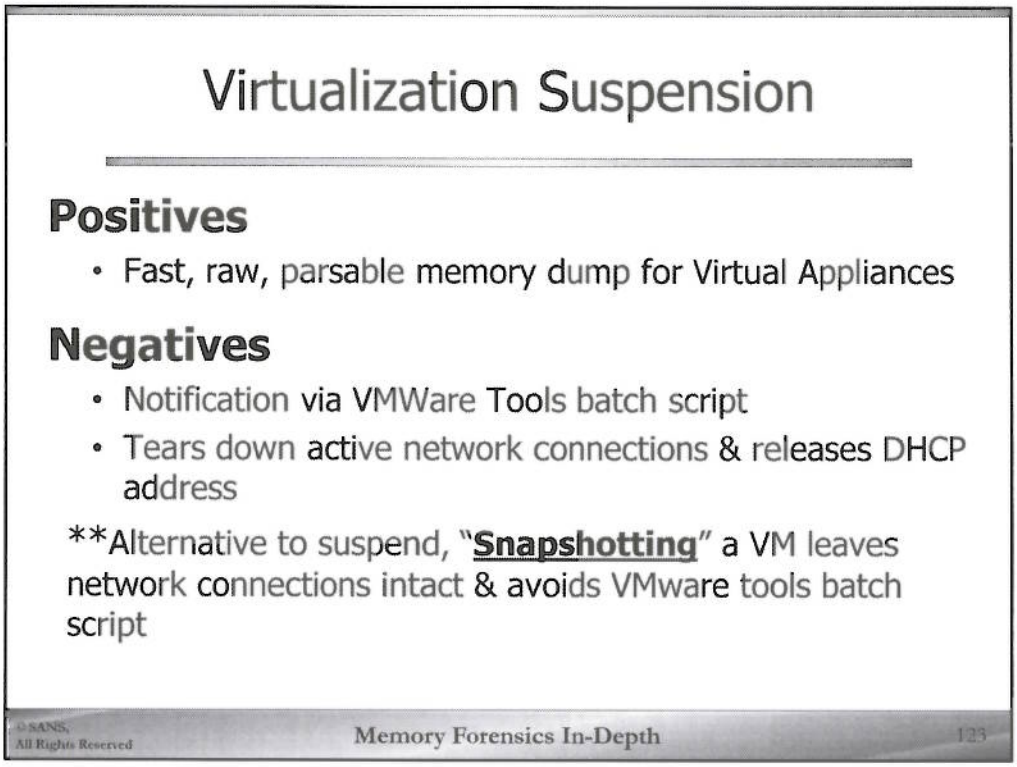
The Windows hibernation feature allows the machine to save the current state to the disk. This allows the user to return to the same state when the power is turned on the next time. Aptly put by an anonymous forum poster, "Hibernate lets you cram your laptop safely into its bag." preserving the state of physical memory, even the state of the processor itself to a file usually called "hiberfil.sys".

Hibernation creates a serialized memory image, one where the full contents of memory have been obtained. Unlike imaging memory with a software program, no data is missed because it was moving while the image was being captured.

One of the possible downsides of hibernation, if you are attempting to capture malicious processes running on the target system is that at the start of the hibernation process, each process is notified and given a chance to do anything it wants. For example, a program could display a dialog box, "Are you sure you want to hibernate now and interrupt your file transfer?". In the case of malicious code, if given the opportunity to react to the hibernation process, subversive behaviors may be implemented such as sending rogue process memory out to the paging file or wiping sensitive data such as encryption keys.

Another notable change that occurs when a system is hibernated is network connections are typically torn down and DHCP addresses are released. This may affect the examiner's ability to enumerate active TCP connections from the hibernation file.

From a forensics perspective, the hibernation file is unmatched. It contains the complete state of the computer from some point in the machine's past. It could be from thirty seconds before the examiner acquired the machine. It could be from three months earlier. Not only does the file give us a window into the state of the machine at that time, but hopefully the user was engaged in some kind of meaningful work at the time in question. If they had their full disk encryption volume mounted, the encryption keys would have been in memory, and thus in the saved state in the hibernation file.

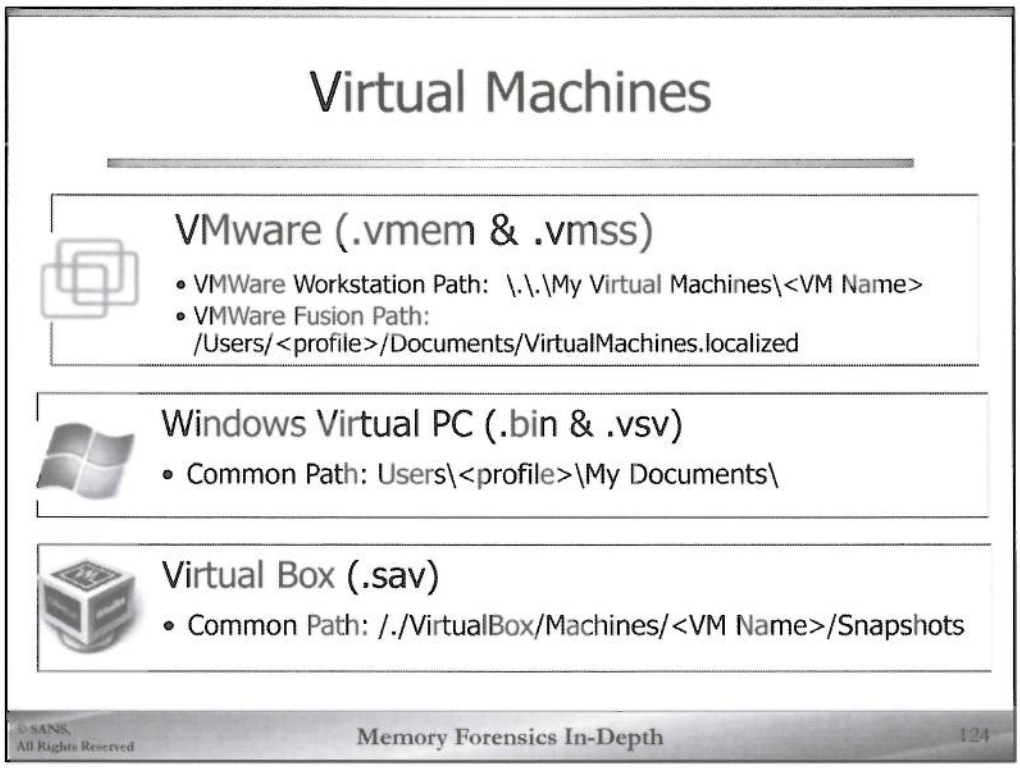


The easiest way to get started with memory forensics is to experiment with a virtual machine. Most virtual machine software can suspend the guest operating system. When the guest is suspended, its memory is normally written to the disk as a file. That file represents a serialized memory image which is *perfect* for analysis. (A serialized image is one where the state of memory is fixed. It does not suffer from the problem noted above where data can be missed because it moved while the imager was working and thus was never seen.)

For example, when a virtual machine is suspended, VMware creates a **.vmem** file which contains a serialized memory image. The serialization process is a kind of clean up routine, creating a sane picture of the computer's operation and making it easier to resume later on. This process has two side effects, however, which could impact your memory forensics.

First, the serialization process notifies each application that it's going to be shutdown. Most legitimate programs don't do anything to interfere. But a malicious program could use this opportunity to delete itself, delete other things on the system, call your ex-boyfriend, or really just about anything. A solution to this type of evasion is to utilize the "snapshot" feature available in most versions of VMware.

Second, the operating system closes all open network connections. This is part of being a responsible party to communication. The computer notifies other hosts that it is going away and should tear down the reliable connections. But this also destroys information about those connections which you may be interested in.

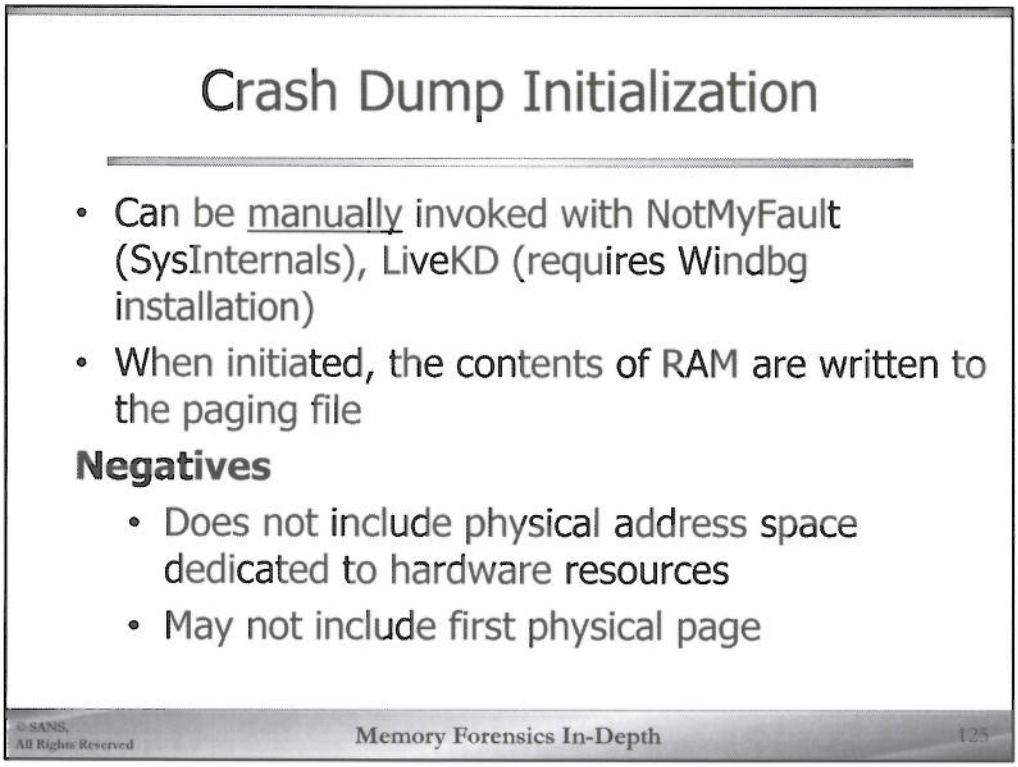


VMware and Microsoft Virtual PC all offer the ability to suspend the guest operating system. VMware creates a file that contains the system's memory and has a .vmem extension. The state of this file can be undefined while the VM is running. (Sometimes it's up-to-date, sometimes it's not.)

VMware has a free player application which can be used to run virtual machines, but not create them. You can download it from http://www.vmware.com/products/player/.

Microsoft's Windows Virtual PC is free and can be obtained from http://www.microsoft.com/windows/virtual-pc/. In addition, Microsoft Hyper-V upon suspension creates a full raw memory image that can be analyzed with Volatility.

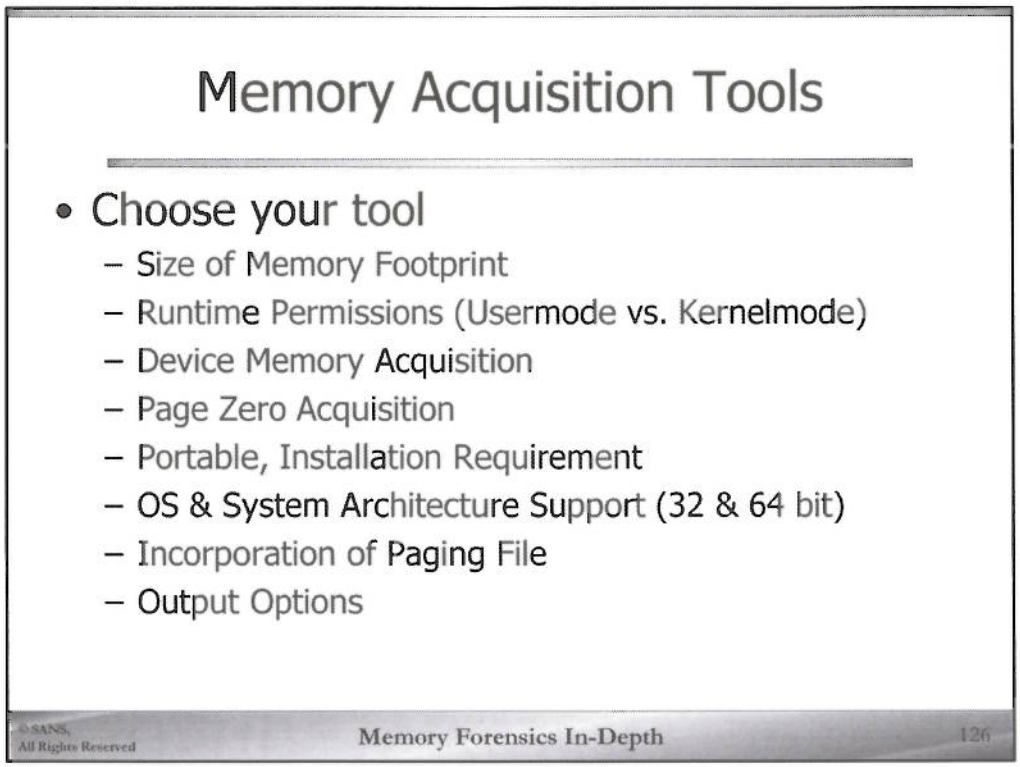
VirtualBox does not automatically save a full memory dump upon suspension. There are ways to configure VirtualBox to save a full raw memory image through various configurations changes described here: http://code.google.com/p/volatility/wiki/VirtualBoxCoreDump. Additional information on VirtualBox can be obtained from https://www.virtualbox.org/.



Crash dumps were created for debugging Windows drivers. When the operating system crashes, it can write out a crash dump file. This file lists what went wrong, details about the problem, and what it was trying to do at the time. Examiners can manually invoke a crash dump through the use of LiveKD or the SysInternals tool, NotMyFault.

Crash dumps can be valuable, because there is no notification process that takes place for running processes on the target system. This gives malicious code less of a chance to react and evade capture. One limitation of a crash dump though is that is does not include an entire capture of physical memory and may miss reserved memory sections containing substantive evidence to an investigation.

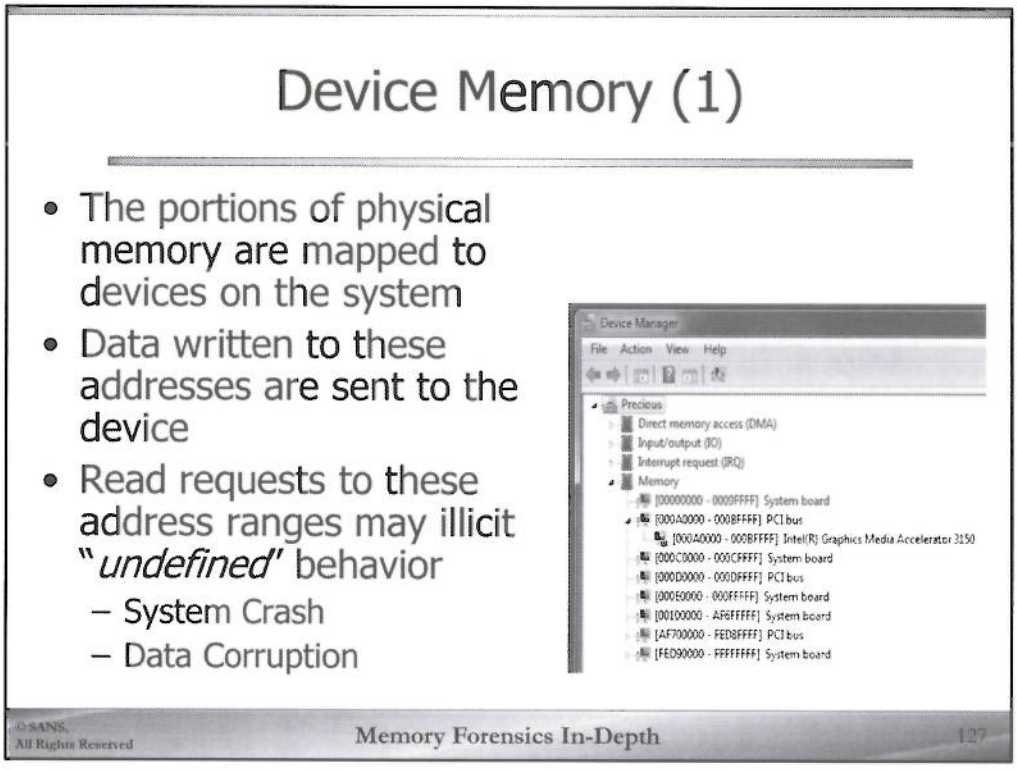
We discuss crash dump files in more detail later in this course but it is important to mention that in certain circumstances, an examiner may desire to create a manual crash dump in order to get a capture of system state whether responding to a potentially compromised system or recreating a malware infection in their testing lab.



Whichever memory capture tool you plan to use during incident response, you should test it out several times, on several systems, before using it in the real world. If memory capture goes wrong, it's easy not only to miss data, but potentially crash the system, and lose everything!

Some questions to ask and things to try out with memory capture tools:

* Does the tool work on Windows 7? Vista? 2003? XP? 2000?
* Does the tool work on 64-bit operating systems? 32-bit systems?
* What privileges does the tool require?
* Is there a way to check the privileges before running the tool?
* Where is the data written? You don't want to write to the hard drive being examined, as you could overwrite evidence. But in some situations, that compromise could speed along the investigation. Whatever choice you make, be prepared to defend it in court!

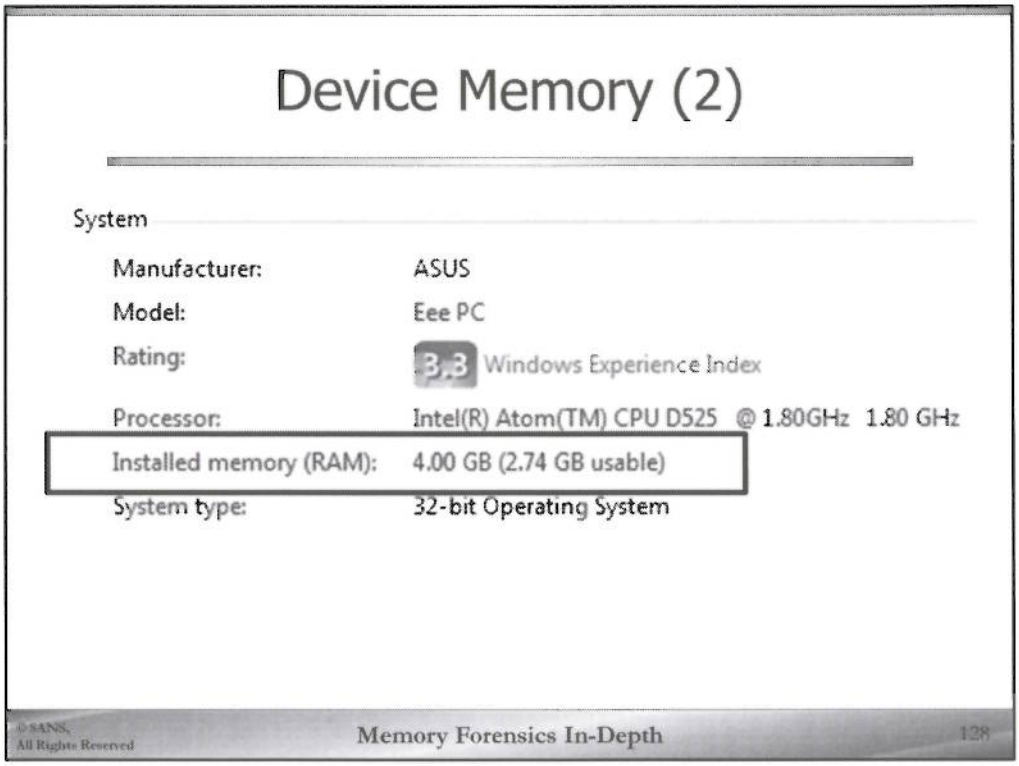


There is a Windows kernel object, \Device\PhyiscalMemory, which allows direct access to the physical memory on the system. Programs can read from this device to get the content of physical memory. Prior to Windows Server 2003 Service Pack 2, this object could be opened by userland programs. After that service pack, a handle to that object could only be opened by code running in the kernel, or a device driver. Userland programs could still read from the device, but required a driver to open it for them.

Reading from \Device\PhysicalMemory is dangerous, however. Device memory is the portion of physical memory which is mapped to other devices on the system. That is, the memory space for devices like video cards is mapped into a portion of physical memory so that the operating system can send data to those devices. Blocks of the physical memory map are reserved for these devices, and data written to those addresses are sent to the device. To handle operating systems that can't address more than 4GB of RAM, the x86 and x64 architectures create these mappings below the 4GB mark.

For example, in the picture above, the range 0xa0000 to 0xbffff has been reserved for the Intel Graphics Media Accelerator. Any data written or requested from those addresses is translated into a request sent to the hardware device. How the device responds is completely dependent on the device. It could respond to a write request by updating the display appropriately. It could respond to a read request with valid data. Or invalid data. Or it could start spewing packets on the network, send your e-mail to your ex-girlfriend, and trash your credit score. It could also hang or crash the computer, destroying your evidence. In computer science parlance, its behavior is "undefined."

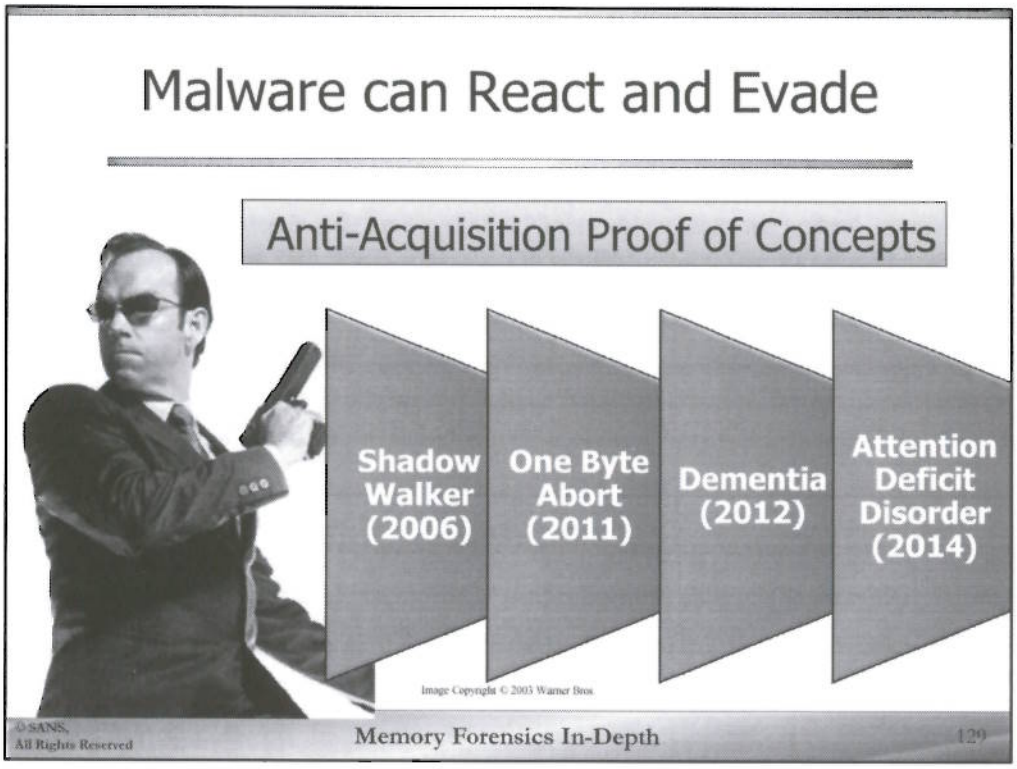
Programs and devices which are attempting to image memory must avoid these areas of mapped memory and not attempt to read from them. Some early memory imaging tools did not avoid these areas, and are thus prone to problems.



The result of these mappings is that some of physical RAM cannot be used by the operating system. Even though the machine pictured here had 4GB of physical RAM, only 2.74GB was usable by the operating system.

Although normally just a performance issue for users, the presence of device memory impacts some memory imaging tools. Tools which attempt to read from the device memory encounter unpredictable behavior. ManTech's memory imaging tool, mdd, for example, can hang the system if it encounters device memory mapped to some video cards, for example. Be careful out there! Test your memory imaging tools before you get to the field!

For more details on device memory, see
http://blogs.technet.com/b/markrussinovich/archive/2008/07/21/3092070.aspx.



Even more insidious than data which was legitimately not captured by a memory imaging program is malware which evades memory imaging programs. Remember that malware is most likely running on the victim system and has hooked operating system functions. It could detect the name of your imaging program as it's being started and interfere with its operation. It could prevent the program from running at all, feed it false data, or just excise those portions of memory which contain the malicious software itself.

This is vastly different from imaging traditional media. When you image a hard drive, the hard drive can't fight back!

Much research has been conducted over the last 8 years in the area of "Anti-Acquisition". Three examples of proof-of-concept tools created by security researchers show the evolution of sophistication in modern malware evasion techniques. The first example, Shadow Walker, developed in 2005 by Sherri Sparks and Jamie Butler, is a rootkit that evades memory analysis and capture by differentiating and filtering execute, read, and write accesses to its own memory sections. A page fault handler hooks and desynchronizes the two Translation Lookaside Buffers (TLBs) used by the processor for cached address translation. When read, the rootkit is able to feed random garbage data to the request and is able to evade analysis in this manner. [1]

The "One Byte Abort" technique was first presented by Takahiro Haruyama and Hiroshi Suzuki at BlackHat Europe 2012. They presented research that identified what signatures and byte offsets the top three memory analysis tools used to identify processes and OS version. With this analysis, they presented a POC driver that manipulated these identified offsets by one byte so that the running analysis tool is unable to correctly analyze the resultant memory image. [2]

And the most recent advance in "Anti-Analysis" research was presented by Luka Milkovic at the 29th Chaos Communication Congress in December 2012. His tool, Dementia, evades memory capture by intercepting

NtWriteFile() calls through the use of inline hooking and a filesystem minifilter.[3] The buffer of a memory acquisition tool is manipulated so that any reference to the target process and its kernel objects is removed and the resultant memory image file has no evidence of this running process.

Our most recent example of an Anti-Memory Acquisition/Analysis POC is ADD (Attention Deficit Disorder), written by Jake Williams[4]. This tool creates fake EPPROCESS, TCP_Endpoint and FILE_OBJECT structures in memory, leading the examiner down rabbit holes where files may have appeared to have been loaded into system memory or network connections to rogue IP/domains may appear to have existed. As with the arms race of malware sophistication and the reversing skills of our ninja malware engineers, anti-analysis techniques will continue to push at the edge of forensic detection.

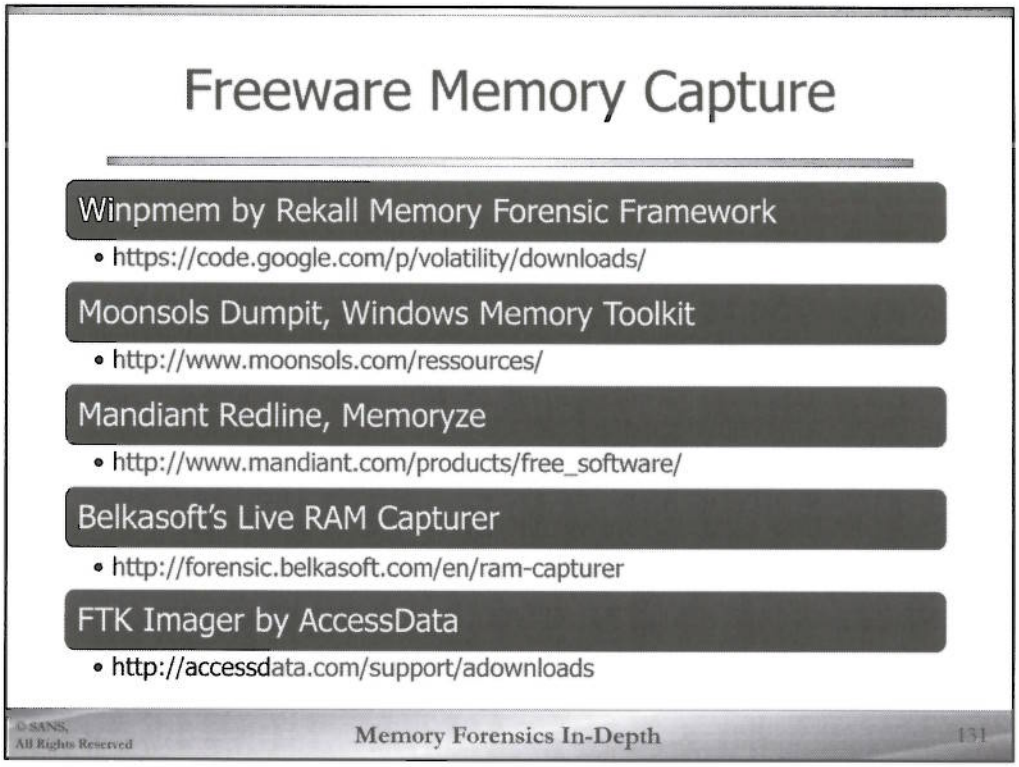
Though these techniques are all "proof of concept" works, we can expect tomorrow's malware variants to implement more of these anti-acquisition behaviors, increasing the difficulty of capturing valid memory images.

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

[2] http://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_Forensic-Slides.pdf

[3] http://dementia-forensics.googlecode.com/files/Defeating%20Windows%20memory%20forensics.pdf

[4] http://malwarejake.blogspot.com/2014/01/analysis-of-add-ref-image-part-1.html



There are a multitude of third-party memory acquisition tools to choose from that support the most current versions of the Windows operating system, Linux and OSX. SANS does not recommend any ONE acquisition tool, but instead strongly suggests that forensic examiners gain familiarity with a few different tools for acquisition. The best tool for an investigation depends on your circumstances: your agency's policies, your budget, the operating system you will be imaging, and so on.

Winpmem from Rekall

One of the most feature rich acquisition tools, winpmem, is currently included in the Rekall Memory Forensic Framework. Winpem is a free, open source acquisition tool, recently developed by Michael Cohen, one of the framework's core developers. This tool supports WinXP SP2 to Windows 8.1 x86 and x64. Some of the exciting features included in winpmem that makes it "best in class" is its live memory analysis capability of the running kernel using Rekall, ability to incorporate the page file in acquisition as well as translate host machine's memory from a host memory acquisition. [1]

Windows Memory Toolkit by Moonsols

The community edition of Windows Memory Toolkit, free for download from www.moonsols.com does not support 64 bit or Windows 8 at this time. Moonsols has provided free tools such as dumpit, an easy-to-use acquisition tool, and hibr2bin, which converts hibernation files to raw memory images. [2]

Redline/Memoryze from Mandiant

The Mandiant Company distributes several free tools for memory capture and analysis. Redline, Memoryze and MacMemoryze can be used for capturing memory images and performing analysis on them.[3] Redline offers live audit collection capabilities that extend well past memory acquisition to include file system, registry and browser artifact analysis. For those who prefer graphical user interfaces to command line tools, Redline is an ideal chose for guided system triage.

Live RAM Capturer by Belkasoft

Live RAM Capturer supports the acquisition of Windows XP to Win8/2012 32 & 64 bit memory and does not require installation.[4] It can run from a USB and distinguishes itself by writing the contents of memory with a kernel mode process, as opposed to user-mode like most other acquisition tools, making it less susceptible to some anti-dumping mechanisms.

FTK Imager from AccessData

AccessData's FTK Imager is one of the most powerful full-featured free forensic analysis and preview tools available today. Not only does this tool allow live preview of a hard drive or volume, and analysis and mounting of forensic images, it also includes a memory acquisition feature. The installed program files can be moved to a removable device and run on a target system.[5]

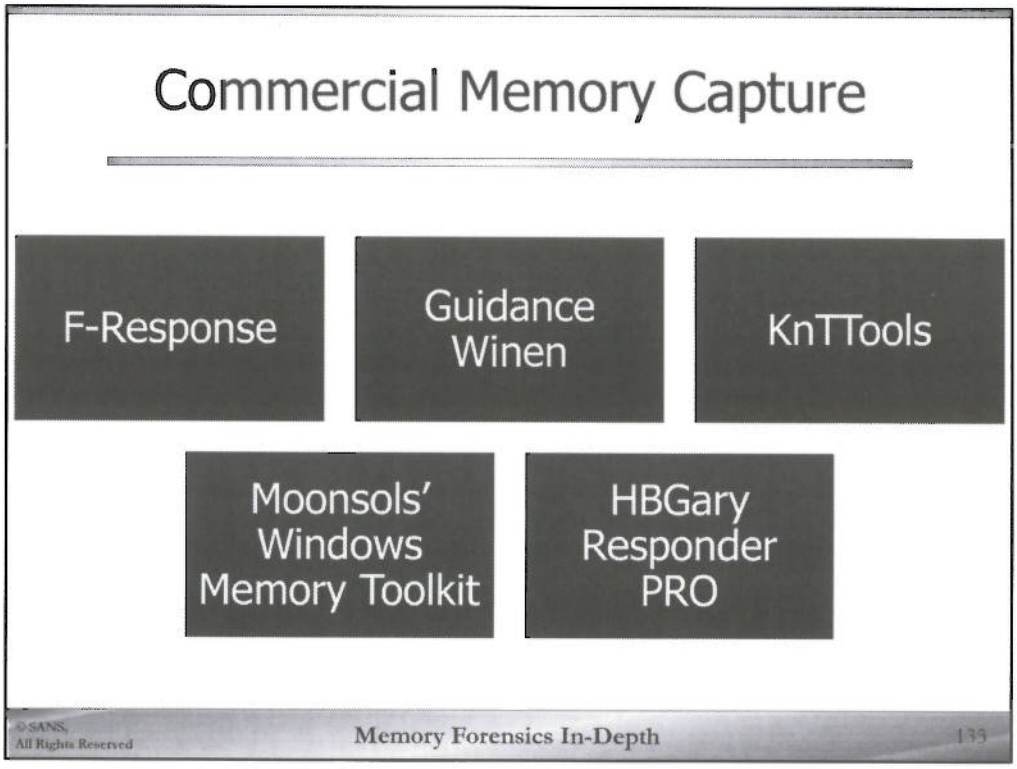
[1] http://www.rekall-forensic.com/downloads.html

[2] http://www.moonsols.com/resources/

[3] https://www.mandiant.com/resources/download/

[4] http://forensic.belkasoft.com/en/ram-capturer

[5] http://accessdata.com/product-download/digital-forensics/ftk-imager-version-3.3.0



In addition to the free tools, there are several supported commercial tools for capturing Windows memory images. These include:

F-Response

F-Response software allows an examiner to obtain read-only access to the full physical disk(s) and *memory* of a networked Windows, Linux, and Apple OSX device. With this remote connection, the examiner can use any analysis platform to interrogate the target, to include the memory forensic frameworks included in the course. For more information on F-Response, visit http://www.f-response.com/

Winen by Guidance Software

Since 2008, Guidance Software has included a command line memory acquisition tool with EnCase, their forensic analysis suite. http://www.guidancesoftware.com/forensic.htm

KnTTools by GMG Systems

Both Basic and Enterprise editions of the KnTdd support compression and encryption of acquired image and include remote acquisition modules. For more information on KnTdd, visit http://www.gmgsystemsinc.com/knttools/

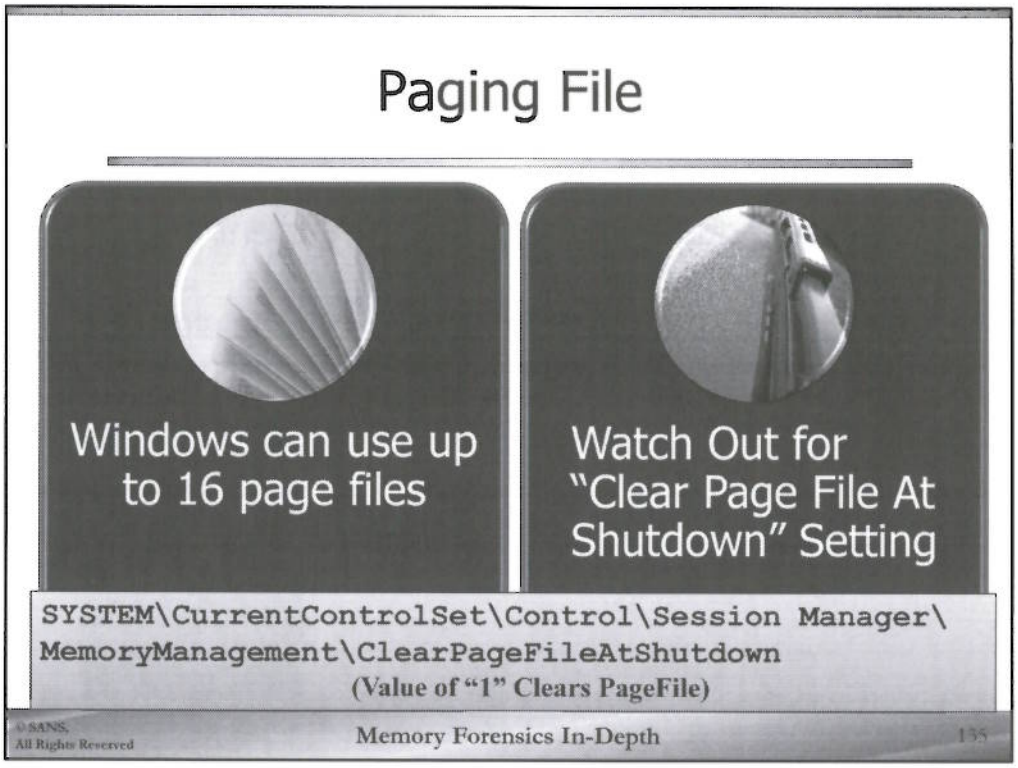
Windows Memory Toolkit

Matthieu Suiche 's company, Moonsols, has published a collection of memory imaging tools known as the Windows Memory Toolkit.[1] This kit comes in a free community version, which is limited in OS support, and paid versions, consultant and enterprise. You can see a complete list of Moonsols products at http://www.moonsols.com/resources/

HBGary Responder Pro

Part of HBGary's Responder Pro memory analysis tool is the FastDump Pro acquisition tool. Fast Dump Pro supports full capture of Windows physical and optional capture of the paging file, making it unique amongst acquisition tools. The resultant physical RAM and page file uses the proprietary **.hpak** format. For more information on Responder Pro, visit the following site:
http://mcsi.mantech.com/products/responder%C2%AE_pro

There are many other products for imaging memory, of course. These are just a sample of what's out there.



Along with capturing the contents of a computer's memory, you should also capture the paging files as well. The paging file contains data which isn't backed by the disk (i.e., is not an executable or DLL). This includes user data, documents, and malicious programs. There could be gold in there!

Windows can use up to sixteen paging files, with one stored on each volume. Typically they are stored in the top level directory as pagefile.sys, but this is configurable.

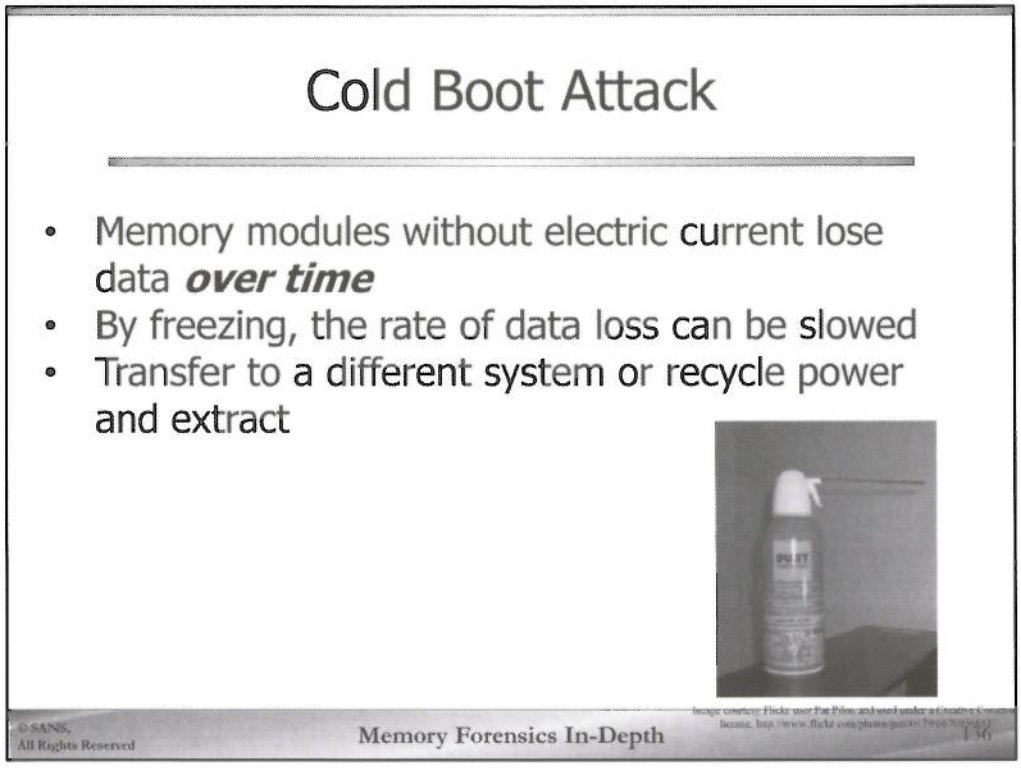
Your memory acquisition tool should capture the paging file as well as main memory. The goal is to capture these two items in as little time as possible. That way the contents of the memory image and the contents of the paging file are as synced as possible. The paging file, like the hibernation file and other operating system files, is always opened by the operating system. You can't copy it from the Windows explorer, for example. You will need a program to either read the raw volume to get the data or some other means.

If you're using VMware, you can mount the VMware virtual disk with their utilities and copy the paging file that way. That method is great in that the memory image and paging file are perfectly synced.

Many analysis programs allow you to include the paging file in your analysis. Unfortunately, however, the paging file cannot be used with Volatility. Using the paging file allows you to recover data which was paged out and not backed by the disk. This could include malicious code, documents, or other key information. You should use the paging file whenever possible.

There is also a "ClearPageFileAtShutdown" option that wipes the contents of the pagefile when a system shuts down. The following setting is maintained at
`SYSTEM\CurrentControlSet\Control\Session Manager\MemoryManagement\ClearPageFileAtShutdown` (Value of "1" Clears PageFile).

This is usually a relevant value to check as it helps in determining whether the contents of the pagefile could contain notable artifacts.

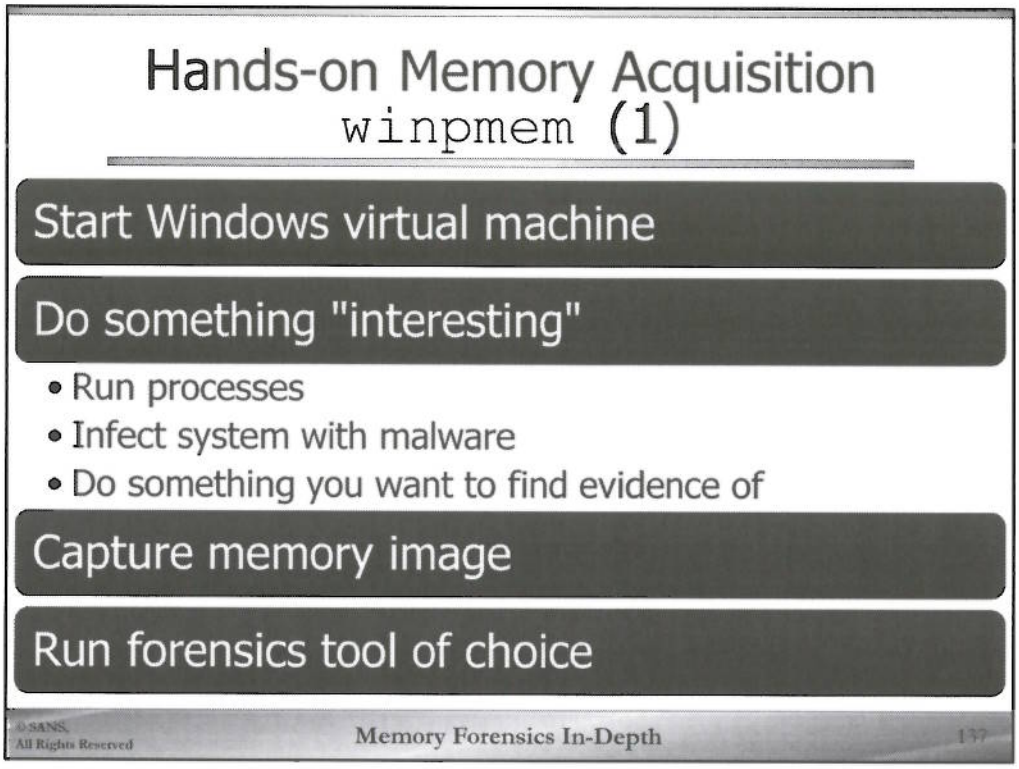


In 2008 a team at Princeton University published a paper and a set of tools for conducting what they called the Cold Boot attack for imaging memory. When the power is disconnected from computer memory, the data in the chips does not disappear immediately. The amount of time it takes for memory to decay to a random state varies based on the type of memory chips and their temperature. The team found that by cooling the chips, such as by using a can of compressed air, they could extend the time the memory remains consistent in the chips. By quickly power-cycling a computer and rebooting into a minimal operating system, they were able to capture the vast majority of what had been in memory. The Princeton team focused on recovering encryption keys from memory, but the resulting memory images can be analyzed for any of the data we've discussed so far.

This method is fantastic in that it does not allow any potential malware to react to the memory imaging process. Programs can't unload themselves or evade memory capture. Only a small amount of data is overwritten by the new operating system. The technique also isn't restricted by the operating system involved. It works equally well for capturing the memory of systems running Microsoft Windows, Linux, OS X, or whatever home-brew operating system the computer was running.

On the other hand, this memory capture method prevents any subsequent investigation of the live system. You can't capture memory this way and then run other commands on it. The process is completely manual, however. If something goes wrong, all of the memory can be lost and the examiner can gather no evidence. If you are going to use this method in the field you must practice it, preferably on the same hardware you will encounter in the field, to ensure you can do it correctly. It's a one-shot deal!

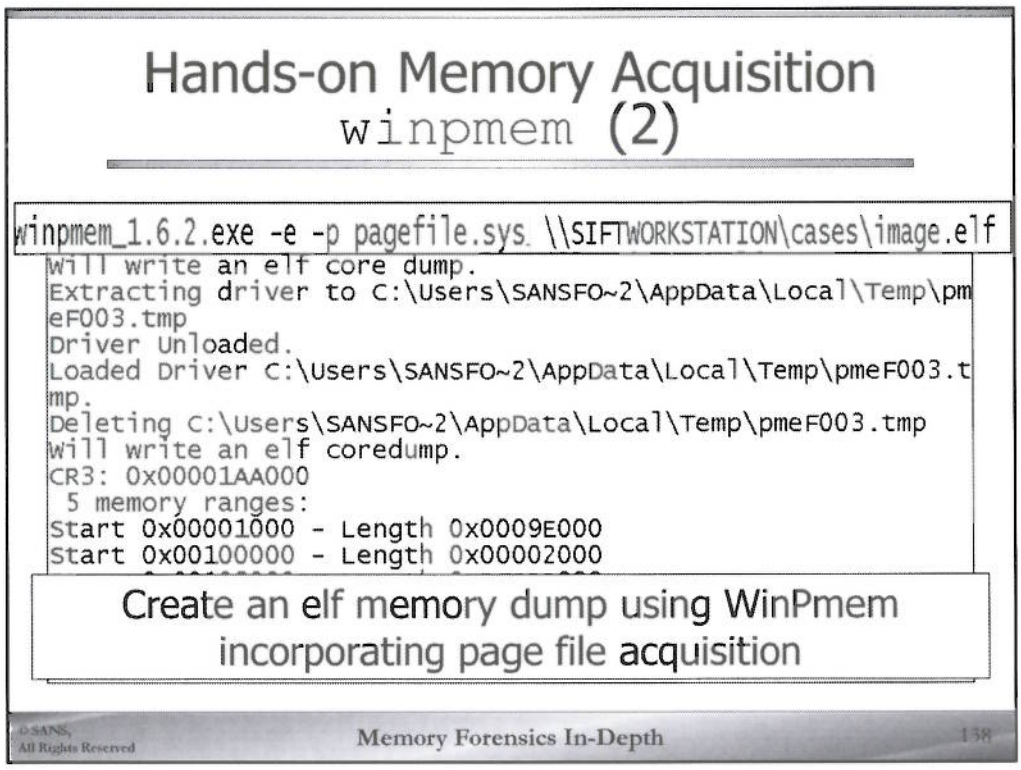
The full details of the Cold Boot paper, including the tools used to recover the data, are at https://citp.princeton.edu/research/memory/.



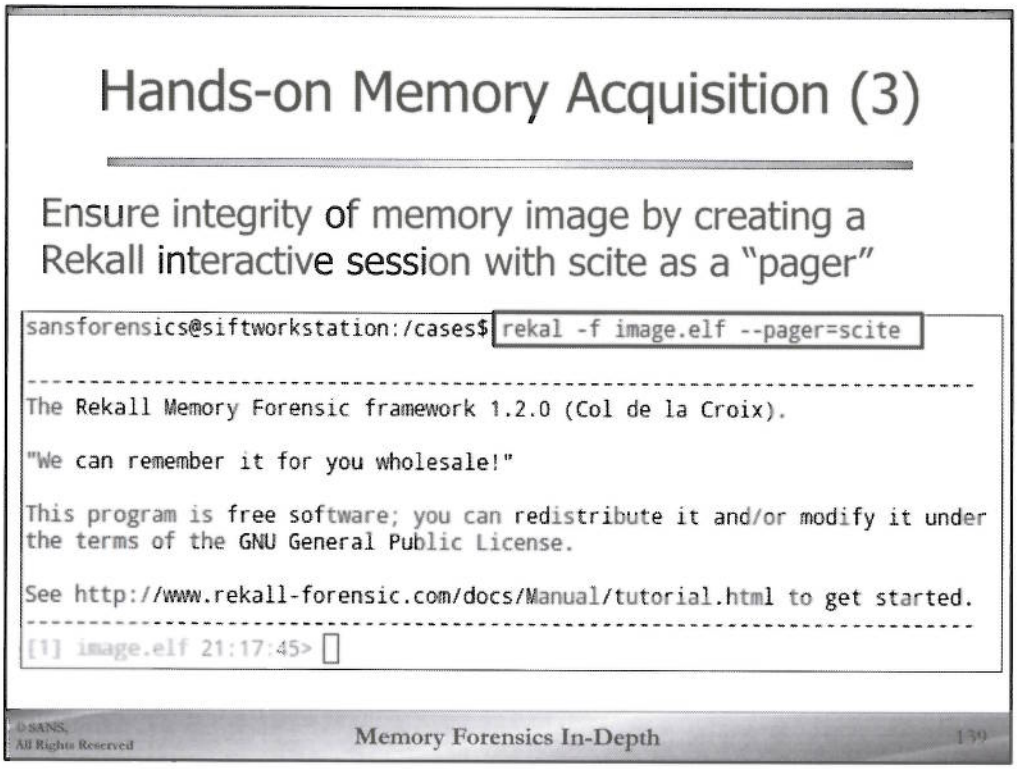
Now that we've discussed the theory, let's try it in practice! In this hands-on exercise we're going to acquire a memory image using the winpmem utility.

To get started, suspend your SIFT VM and start up the Windows VM you were given for this course. (To suspend the VM, look for the button which looks like a "pause" symbol, or choose "Suspend" from the Virtual Machine menu.)

Once the VM is running, launch some processes. Start Notepad, a Command Prompt, visit some websites, go crazy! Do something you'd like to see evidence of in the memory image. You could also infect this VM with malware, but that's not recommended in this course.



Now that there's something interesting to find, we're going to capture a memory image using the Rekall's WinPmem utility. One of the advantages of WinPmem is that it's a one-step memory acquisition tool. It can create a raw memory dump for x86 and x64 architectures and supports Windows XP SP2 to Windows 8.1 systems. By default, WinPmem outputs as a raw memory dump, but can be optioned to output as an elf file or to StdOut to allow for input to another tool. In addition, it can be configured to acquire the page file during acquisition, as shown below.

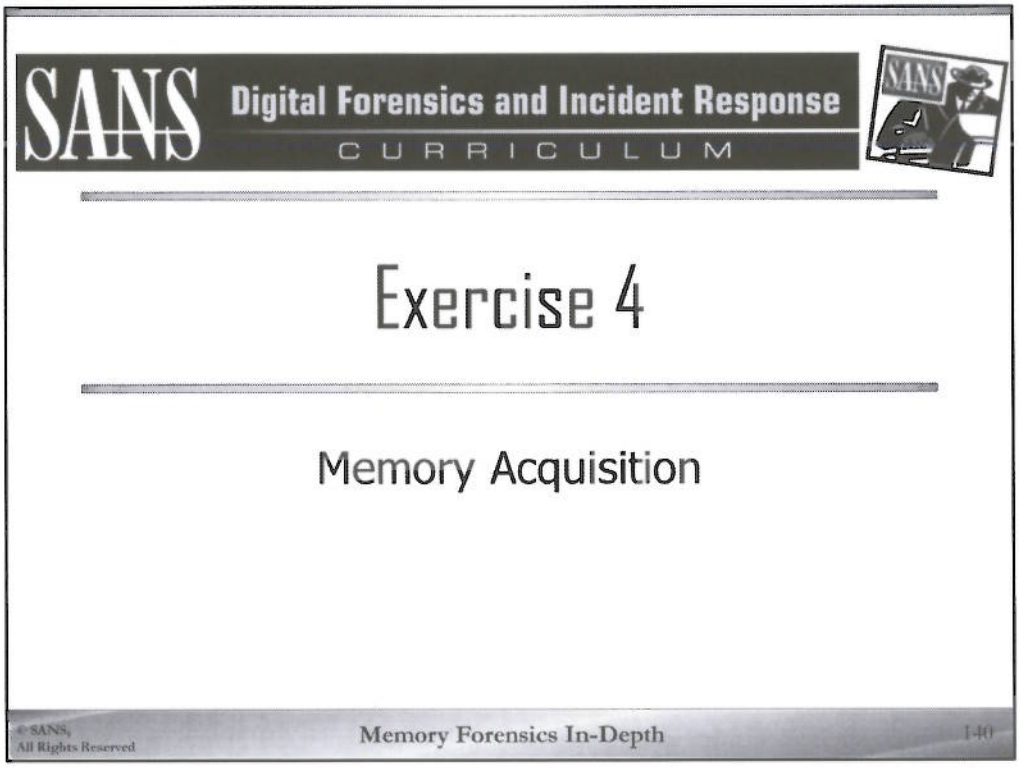


After our memory acquisition has completed, we will validate its integrity by analyzing it with the **imageinfo** plugin. There are many options for doing this to include using the standalone Windows Volatility framework located on your Win8.1 virtual machine at C:\Tools. Additionally, you may drag the path of the image into a terminal window on your Ubuntu SIFT and parse it via a remote share to the target machine. A third option is to copy the resultant memory image to your SIFT workstation's /cases directory, and parse directly from there.

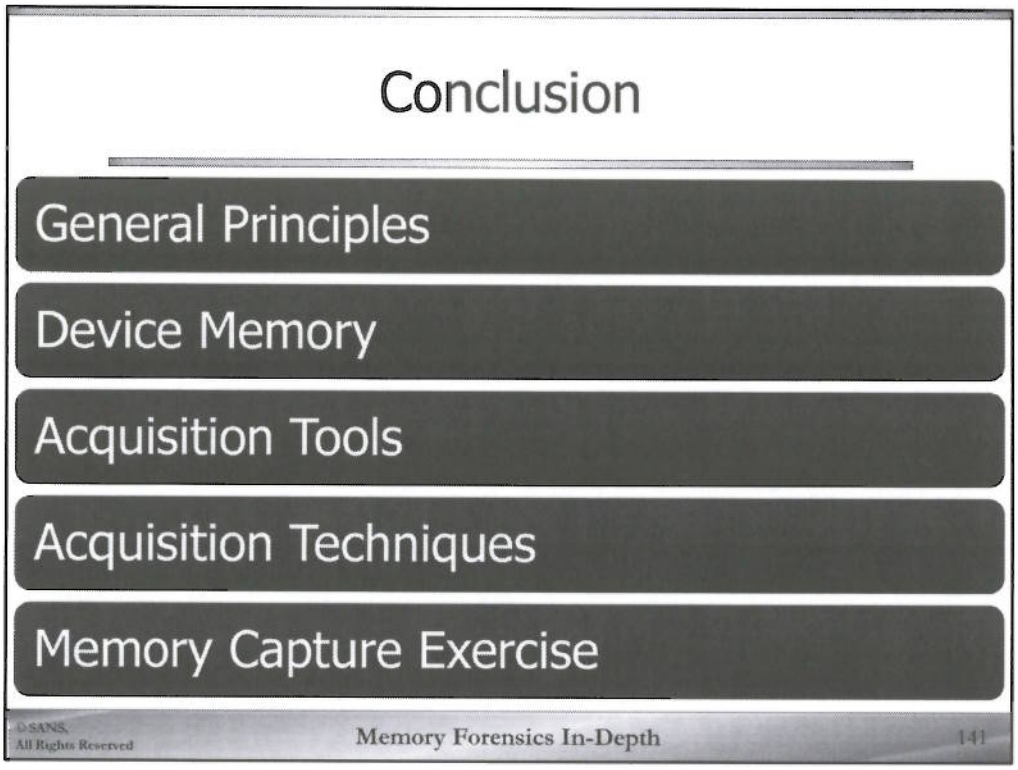
If the memory image is readable, you can expect to see output similar to that shown below.

```
sansforensics:cases$ vol.py -f win7.img imageinfo
Volatile Systems Volatility Framework 2.3
Determining profile based on KDBG search...

          Suggested Profile(s) : Win7SP0x86, Win7SP1x86
                     AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (/cases/win7.img)
                      PAE type : PAE
                           DTB : 0x185000L
                          KDBG : 0x82b78c28L
        Number of Processors : 2
     Image Type (Service Pack) : 1
               KPCR for CPU 0 : 0x82b79c00L
               KPCR for CPU 1 : 0x807c5000L
            KUSER_SHARED_DATA : 0xffdf0000L
          Image date and time : 2013-07-25 14:03:18 UTC+0000
    Image local date and time : 2013-07-25 10:03:18 -0400
```



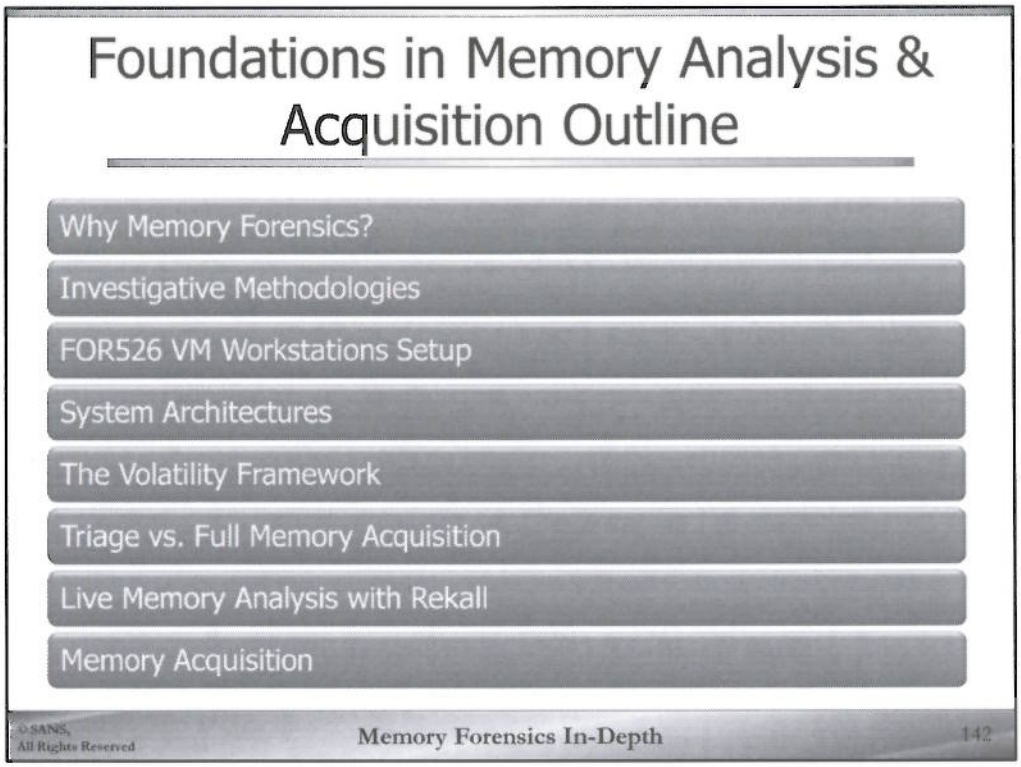
This page intentionally left blank.



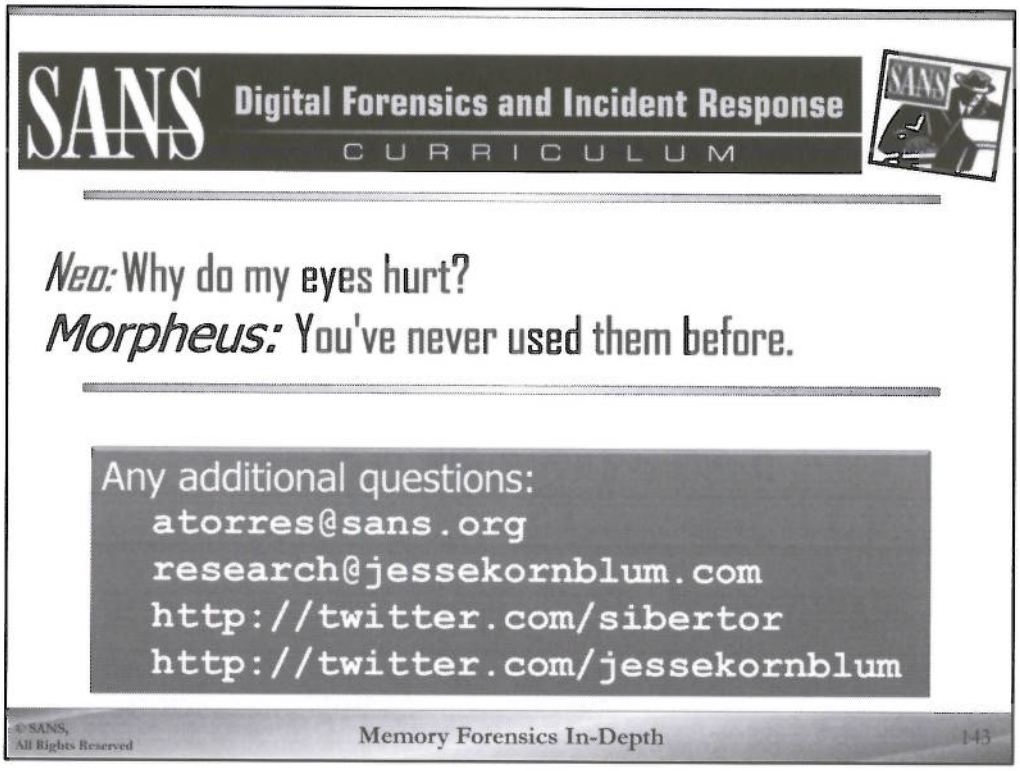
In this section we have discussed how to get started with memory forensics by capturing a memory image. Even though the term "image" is in debate, there has to be some sample, which can be checked into evidence, for forensics. The standard for gathering data in incident response, RFC 3227, dictates collecting memory before anything else. Capture memory first! (Or be prepared to explain why you did not.) But also remember that memory imaging cannot be repeated like disk imaging. Every image will be different.

The volatile nature of memory means that the memory image will be changed during the acquisition process and by the acquisition process. With the sensitive nature of memory, including device memory, it is imperative that you test out your memory imaging tools ahead of time. A crash with such tools could leave you with no evidence for processing.

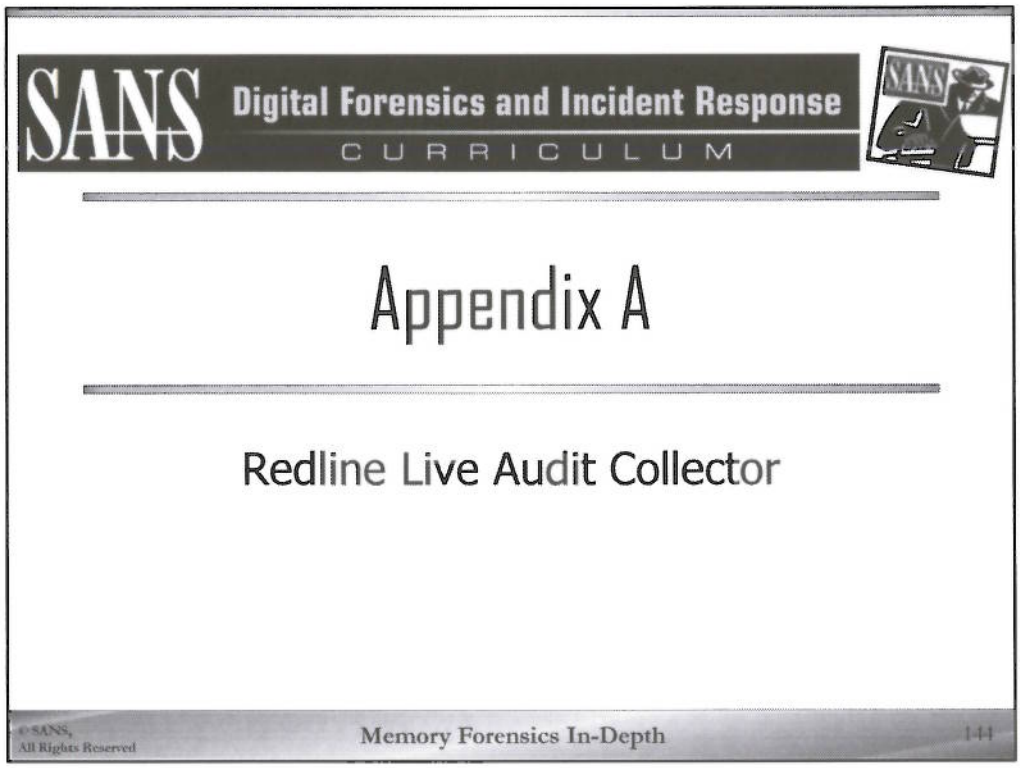
There are both free and paid versions of memory imaging software. Which ones work best for you and your requirements is something you'll have to figure out. Don't forget the paging file! When working in the lab, using a virtual machine such as VMware is an easy way to capture memory images. There are also some exotic tools like Firewire devices, and rebooting into a minimal operating system (the Cold Boot attack).



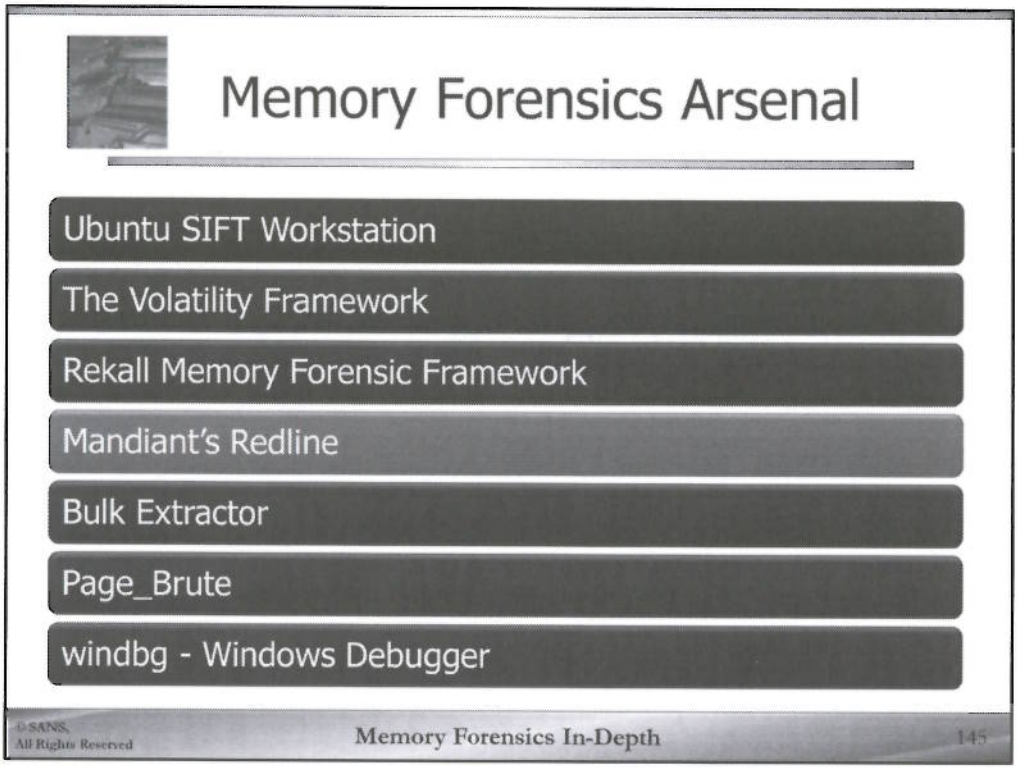
This page intentionally left blank.



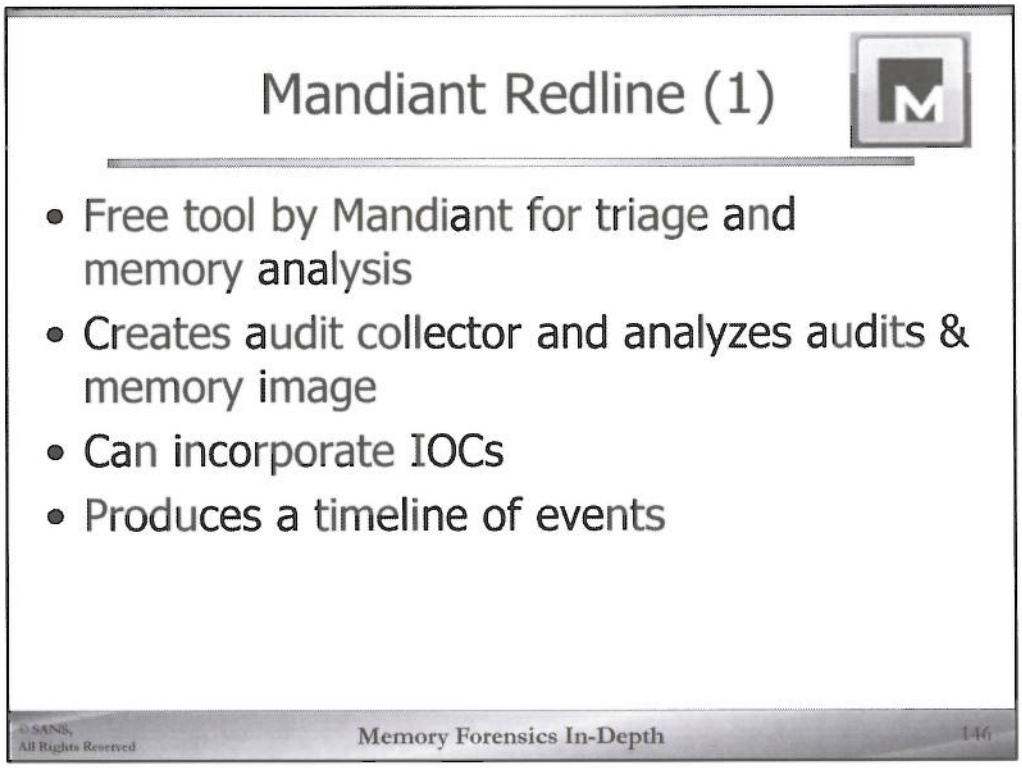
[1] The Matrix. Dir. Andy Wachowski and Larry Wachowski. Warner Bros. Pictures, 1999. DVD.



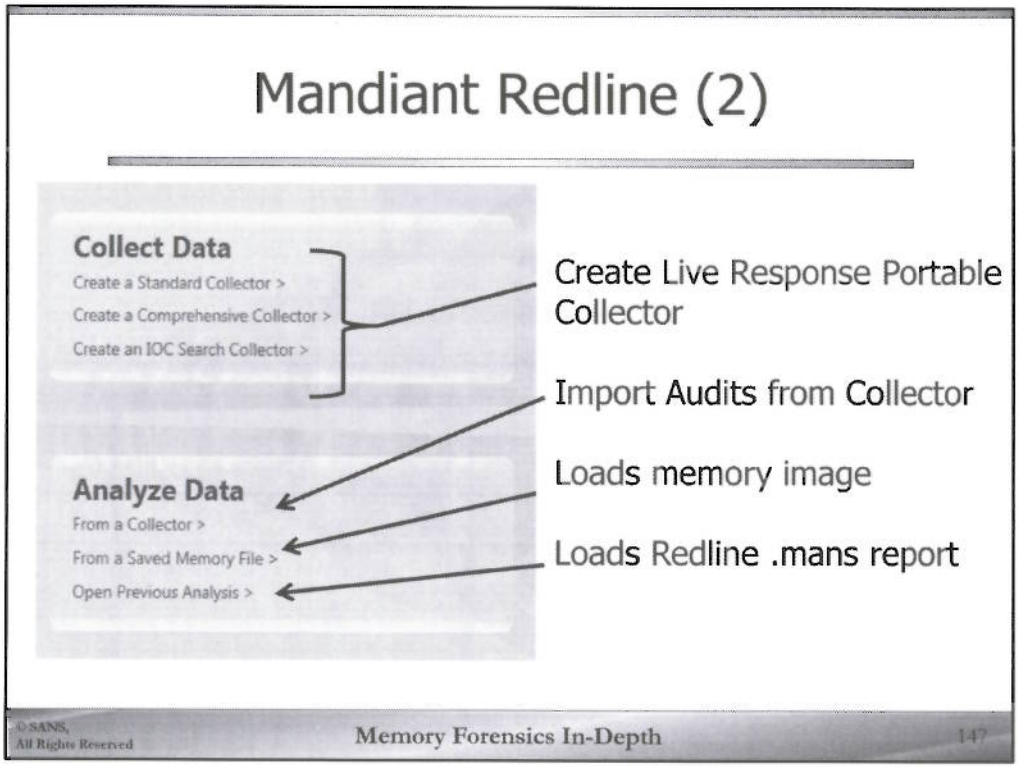
This page intentionally left blank.



Redline is one of our "go to" tools for memory analysis in FOR526. Much like the contrast between early version of EnCase and FTK, Redline does its preprocessing of audits and memory images upfront while Volatility, as we have just seen, works in a more granular, modular fashion. Each tool has different strengths, with Redline providing a more guided walk-through of memory analysis, great for those new to memory forensics. Additionally, its ability to generate a live audit collector and triage a system without requiring a full memory acquisition sets it aside from most other memory parsing tools.

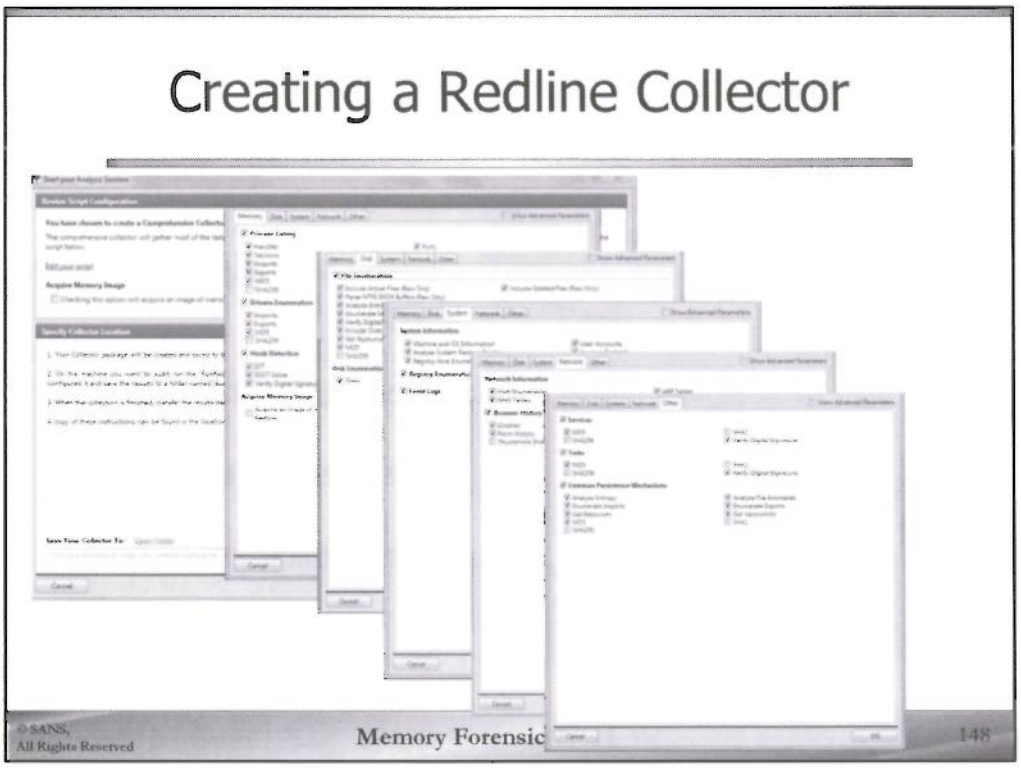


Redline is a live audit triage and analysis tool written and maintained by Mandiant. It is available for download for free from the Mandiant website: https://www.mandiant.com/resources/downloads/ Some of its advanced features include the ability to compare audit data to previously signature malware through the use of Indicators of Compromise. An examiner may also further enhance the tool's ability to pinpoint malicious processes and injected memory sections by utilizing the whitelisting feature. Mandiant provides a downloadable whitelist.txt file which can be imported into Redline, thereby reducing the data presented for analysis if the imported audits include a filesystem listing that includes MD5 hashes.



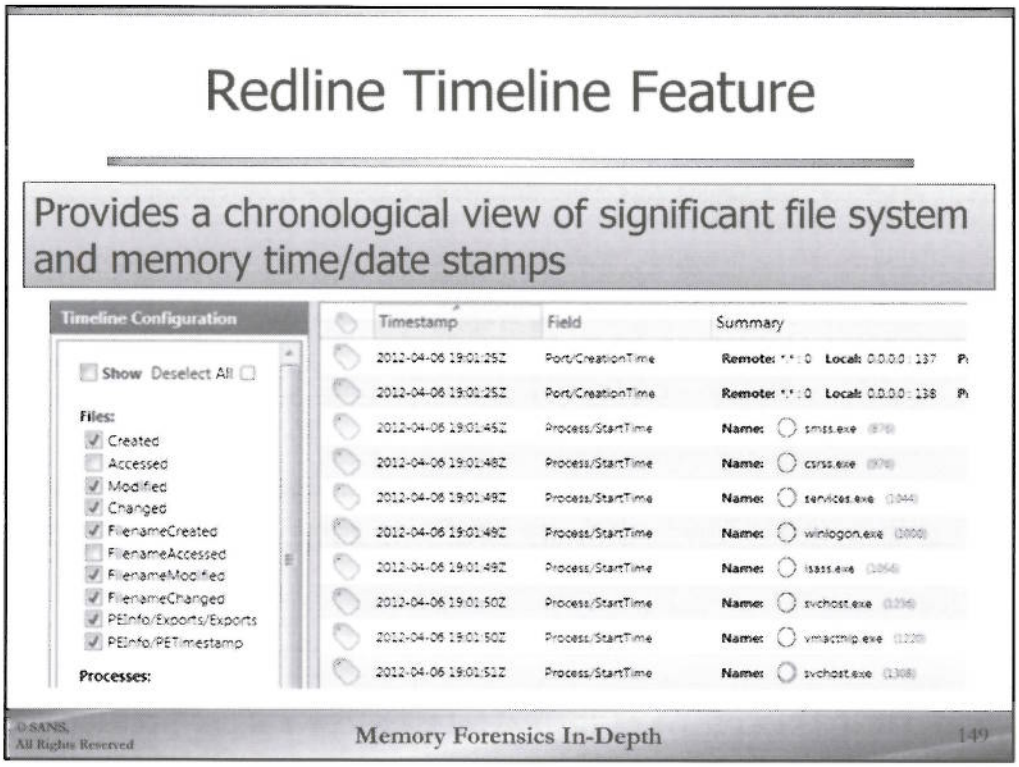
In its most recent version, Redline allows for the creation of a collector tool that can run on a target system, pulling back file system, memory artifacts, event logs, url history, prefetch and much more. There are three different collector categories, standard, comprehensive and IOC specific. The standard and comprehensive are very similar with the comprehensive offering more options for audit collection. An IOC specific collector can be customized to include all audits that pertain to an IOC library. Redline also allows the collectors to be set up to capture physical memory in addition to gathering audit files.

Again, the huge benefit to audit a system rather than dumping memory is efficiency. If an investigator knows an exact collection of data that will allow for the triaging of systems throughout the network in order to determine the proper scope of an intrusion, using a collector with limited audits is more time efficient than pulling back multiple large memory captures remotely across the network.



When creating a audit collector with Redline, there are three different options, standard, comprehensive or IOC specific. Shown above are the options checked by default for a comprehensive collector. Of course, the investigator may customize the collector for his/her specific needs, removing some of the time intensive audit such as registry and file enumeration. The Redline User Guide details the impact per option selected[1]. For example, including strings parsing in the audits will increase the size of the audits. An IOC-specific search collector retrieves data that matches only selected Indicators of Compromise, prompting during the collector creation for the location of the IOC library. IOC analysis can be performed after the import of audits collected from the standard and comprehensive collector as well.

[1] Redline v.1.11 User Guide. https://dl.mandiant.com/EE/library/Redline1.11_UserGuide.pdf



Redline offers a fantastic timeline view of data it analyzed from memory or file system audits. A customizable filter feature, shown in the left pane of this slide, gives the user the ability to only show events of interest as well as only events from a specific timeframe. These two features, "TimeWrinkle" and "TimeCrunch" allow an examiner to reduce data and hone in on specific windows of time when notable activity may have occurred and filter out the "noise". If you are importing data from an audit collection, the timeline will only populate those fields which the collector was configure to acquire from the target host. Including file system audits in the collector tool will obviously provide a more comprehensive Redline report and timeline, but will take hours longer to run on the system.