SANS

www.sans.org

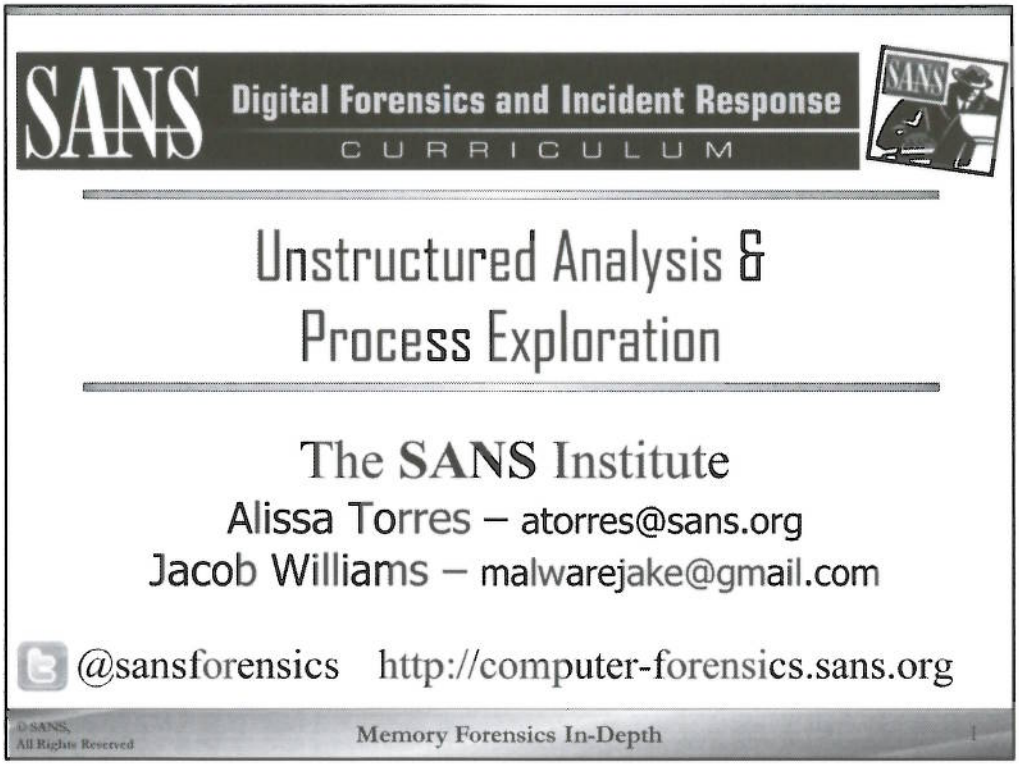
FORENSICS 526

MEMORY FORENSICS IN-DEPTH

526.2

Unstructured Analysis and Process Exploration

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

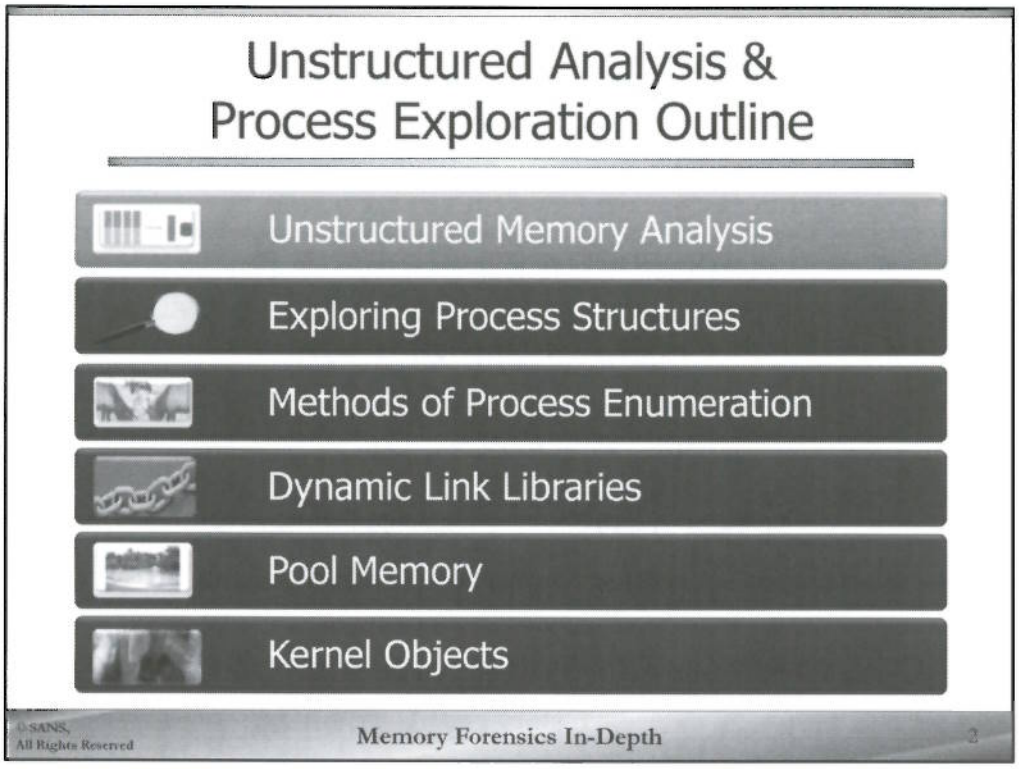


Unstructured Analysis & Process Exploration

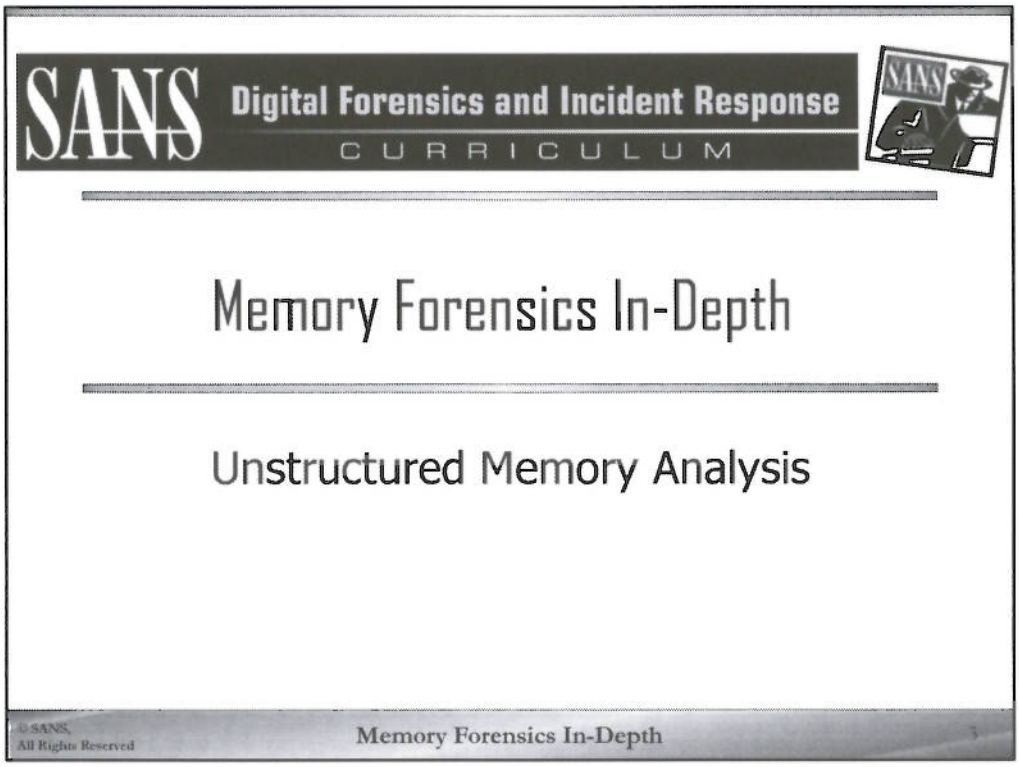
Alissa Torres – atorres@sans.org
Jacob Williams – malwarejake@gmail.com

Some original material contributed by Jesse Kornblum

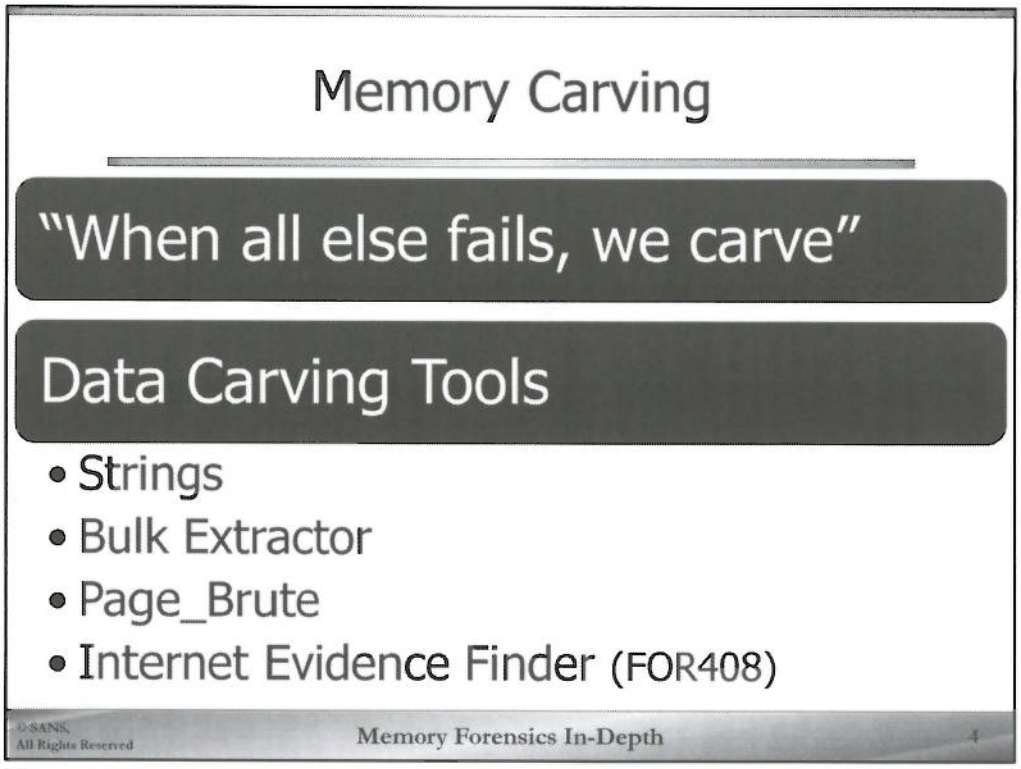
http://twitter.com/sibertor
http://twitter.com/malwarejake
http://twitter.com/sansforensics



This page intentionally left blank.



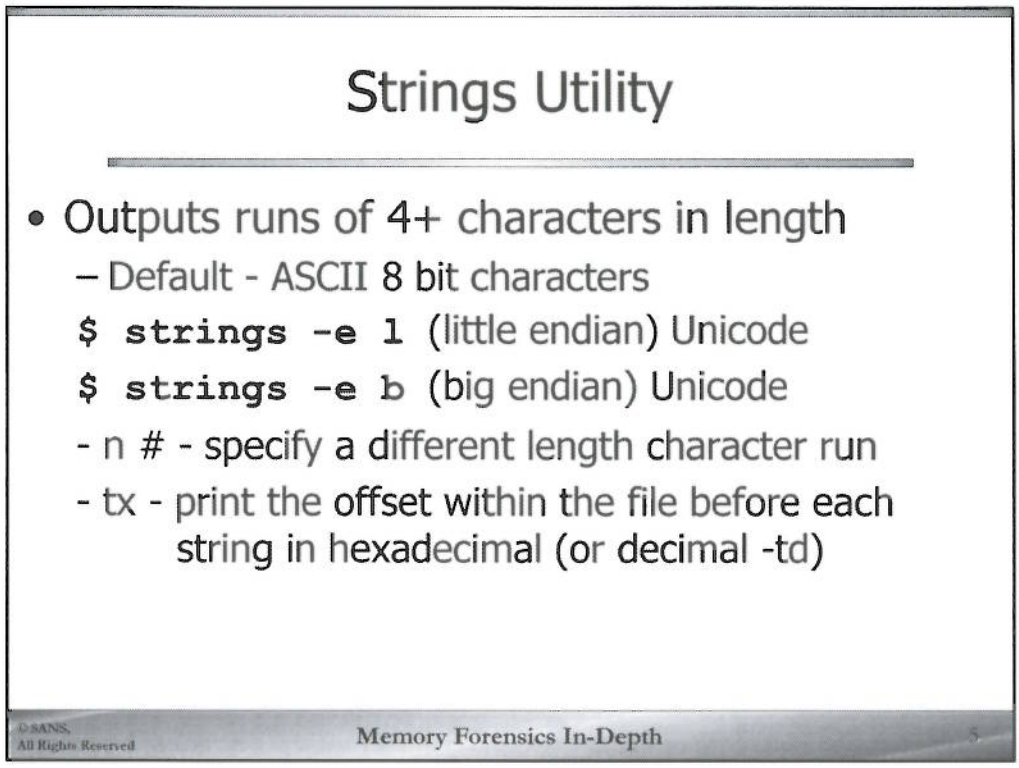
This page intentionally left blank.



There is an adage in computer forensics that when in doubt, we carve. That is, if you don't know where to start your examination, try carving out known file types of data. You can carve anything looking for data, of course. File systems, slack space, swap space, and especially memory images.

Most file carvers work on headers and footers, and you could search memory images for headers and footers. But such a search probably would not be fruitful. As we noted earlier the default page size for most architectures is 4KB. Few file types take up so little data, and so most files in memory will span multiple pages. Unlike file systems, memory pages tend to be heavily fragmented. Attempting to carve full files is not worth your time.

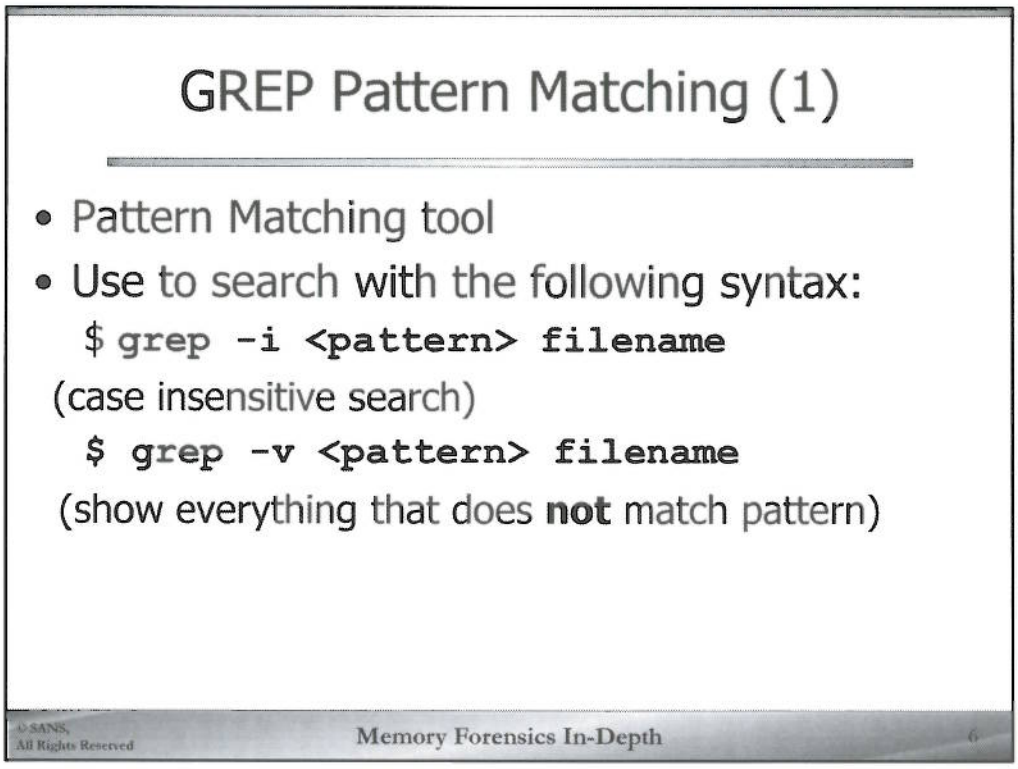
But when the data we looking for is smaller than 4KB, carving memory can be beneficial indeed. We're going to take a closer look at several different carving techniques in the next section to include page_brute, a special stream-based parsing tool that allows us to implement YARA rules. In addition, we will make use of several plugins in the Volatility framework that perform unstructured data carving.



The strings utility is a native tool that prints character sequences that are at least four characters long by default. We commonly use strings against extracted memory sections or portable executable files to determine functionality based on functions or unique strings. For the forensics examiner, having mastery over the strings and grep (on the next slide) tools is essentially. Although quite basic in use and function, strings output can be used to expand the scope of an investigation based on the mere presence of a mutant string in a system's memory image or page file.

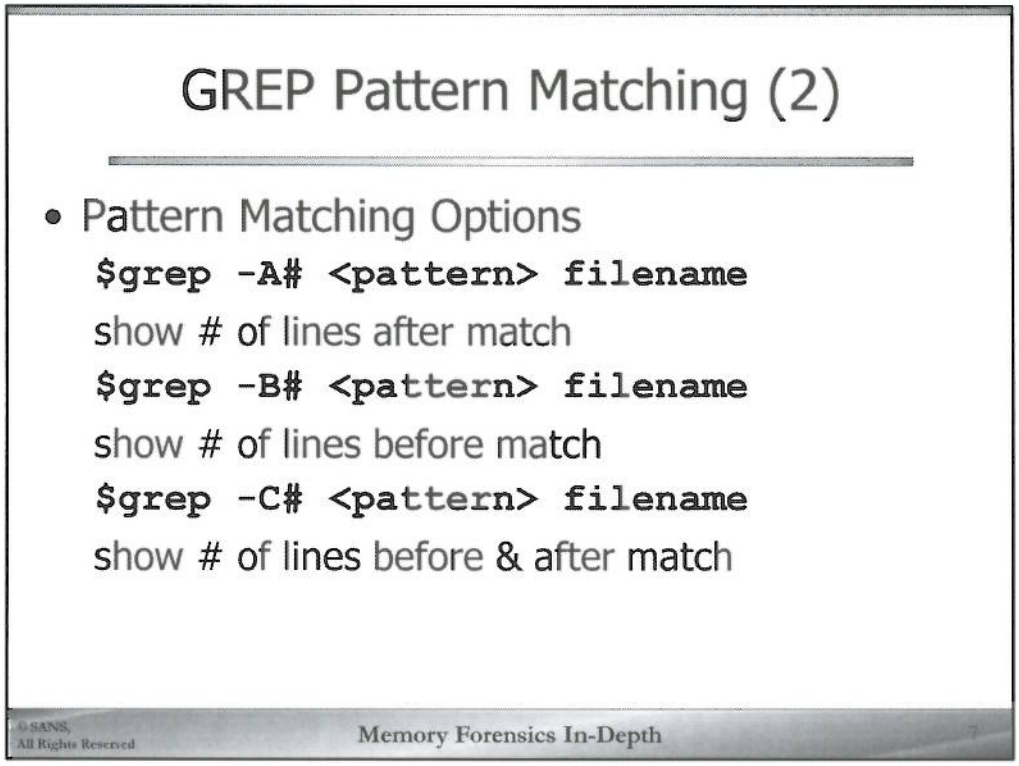
Strings, by default, searches for 8 bit ASCII characters. Windows memory images invariably contain Unicode AND ASCII strings, so having the capability of extracting both is imperative. In order to extract Unicode strings, use the -e option and specify whether you wish the tool to search for characters occurring in little endian or big endian format (-e l = little endian, -e b = big endian). The length of contiguous characters that strings prints can be modified as well with the -n <desired length> option.

Often, it is beneficial to include the offset of where the character string was found in the target file. By using the -t option, an examiner may include the offset in the strings output, with a h for hexadecimal or a d for decimal offset values.



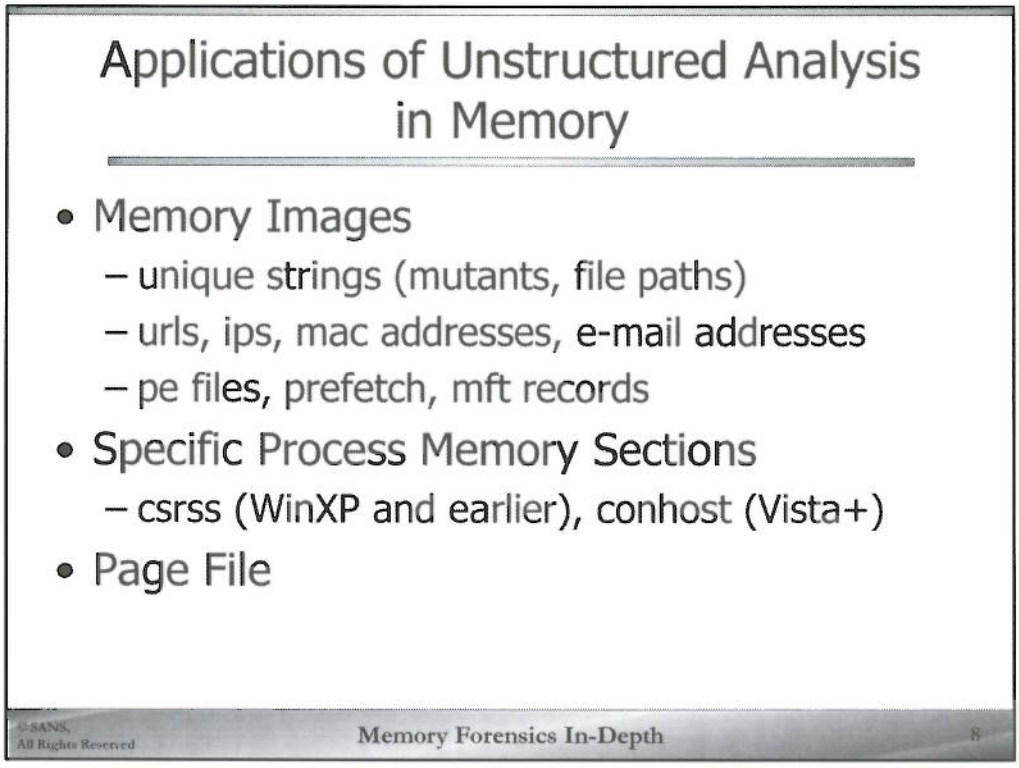
Once we have our strings output, the next step in unstructured analysis is to search based on pattern matching for strings of interest. To do this, we can use the native GREP **(Global Regular Expression Print)** tool on your SIFT Workstation. GREP scans a target file and identifies pattern matching, printing to standard out the line in which the match occurs. We will be using GREP in many different ways in FOR526 as we scan through process memory dumps attempting to identify mutant strings, subverted table entries and notepad text files.

Two options that are quite useful when using grep include the -i option for case-insensitive pattern matching and -v for doing a "reverse" search - "show me lines that do NOT contain this pattern". To obtain a count of the number of matches that occur, the -c option can be included in the grep command.



There are ways to manipulate GREP's output, having it print surrounding lines proximate to the pattern match. Using -A#, GREP will print # many lines after the pattern match. With -B#, GREP prints # lines before the pattern match. And finally, the most used option in this class is -C#, with GREP printing # lines before AND after the pattern match.

This is especially useful when parsing Volatility plugin output that dumps values from a kernel object. Viewing context surrounding a keyword like "Running" when attempting to identify the thread(s) that were active while the memory was being acquired will allow you to spot the TID and PID when looking at the enormous amount of output from the **threads** plugin, for example.

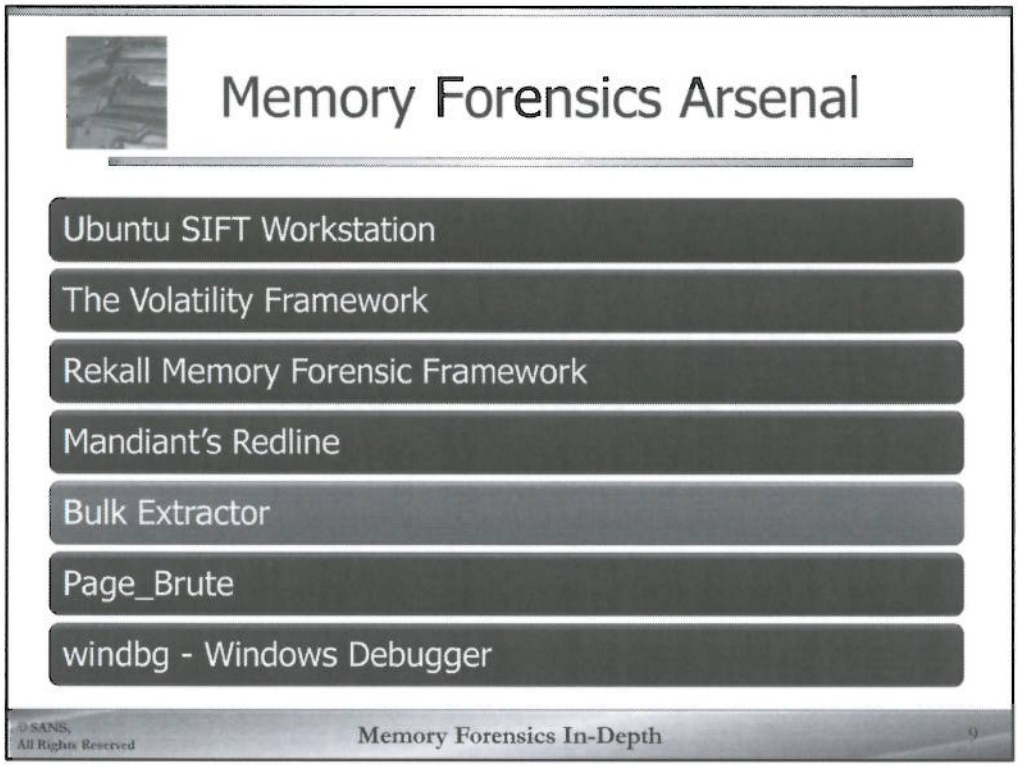


When does unstructured analysis apply in memory forensics?

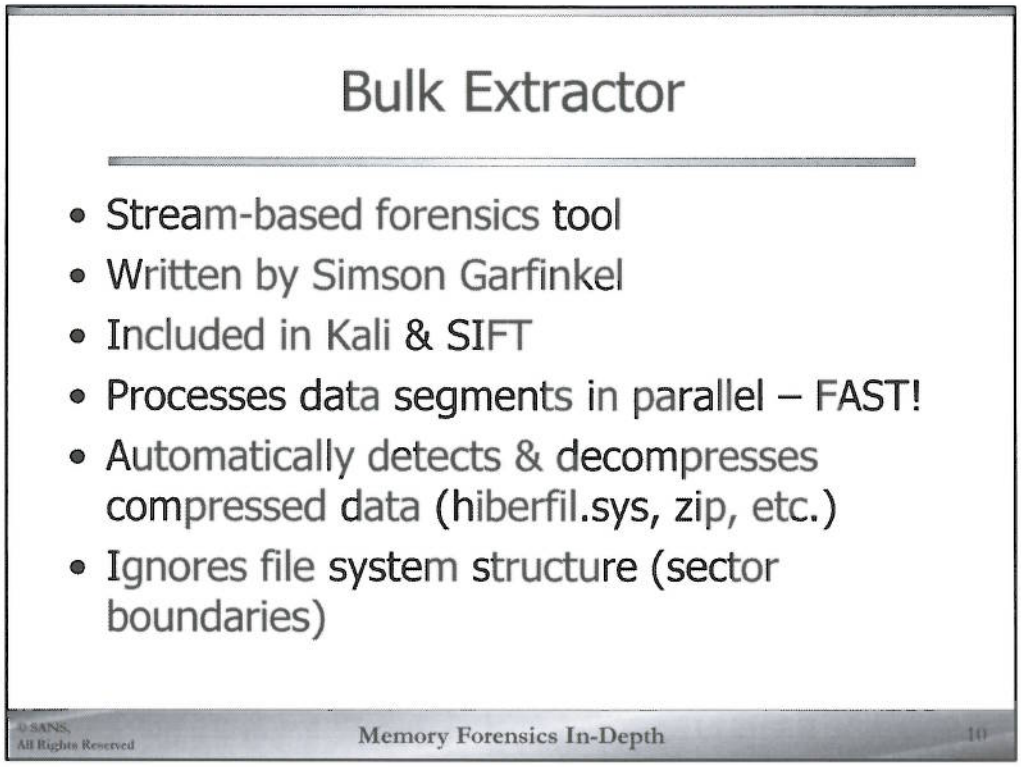
Investigators can use unstructured analysis quite effectively against all different data types, across entire memory images (hard drive images, too), the system page file, hibernation file or extracted process memory sections.

In fact, many incident response service providers dump the memory sections from specific Windows processes that handle terminal sessions in order to detect attacker commands in intrusion cases. In Windows XP and earlier, the csrss process is responsible for tracking terminal windows and if the memory sections of this process are dumped and strings is run against it, commands entered in cmd.exe processes, whether active or terminated, can be seen. For Vista and later, the conhost process assumed this function and should therefore be dumped and have strings run against it to search for terminal window commands. Conhost processes are spawned, one per cmd.exe, and are deallocated and overwritten with much greater speed than the csrss process, so not as much historical data is retained on Vista and later systems.

Another forensic use for unstructured analysis is the system page file. We will discuss the page file in upcoming slides, but notably it is a wealth of unstructured data that may pertain directly to your investigation.



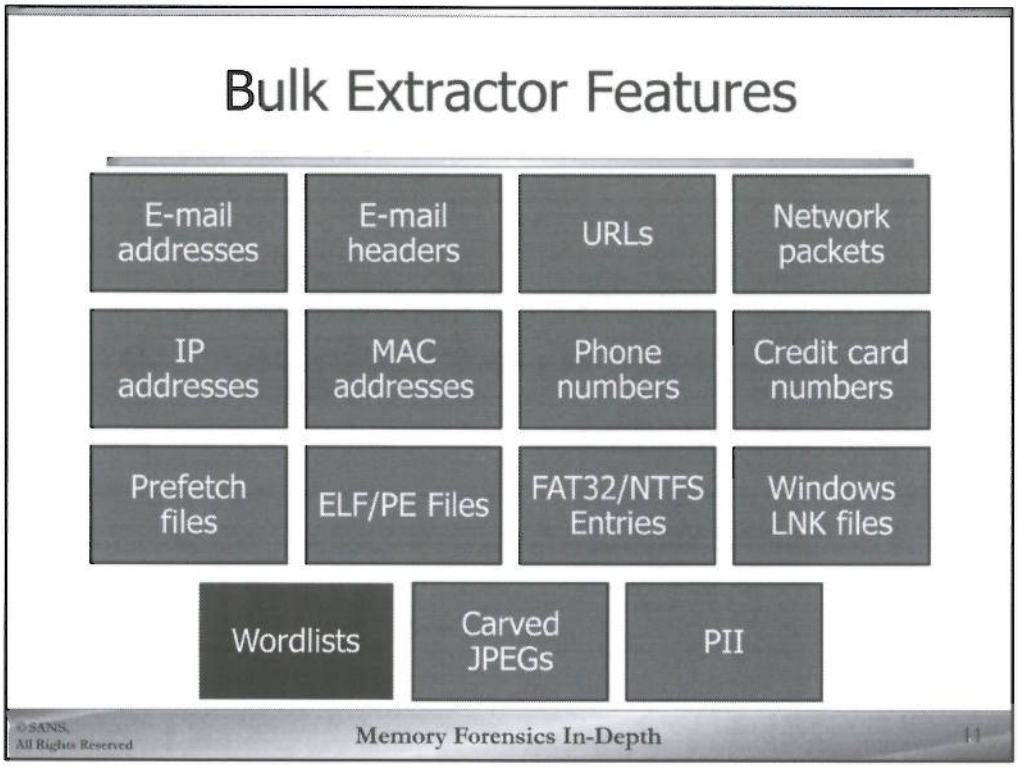
Redline is one of our "go to" tools for memory analysis in FOR526. Much like the contrast between early version of EnCase and FTK, Redline does its preprocessing of audits and memory images upfront while Volatility, as we have just seen, works in a more granular, modular fashion. Each tool has different 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.



One of the most impressive, efficient free open-source tools for unstructured data analysis is Bulk Extractor, written by Simson Garfinkel, an Associate Professor at the Naval Postgraduate School in Monterey, CA. Bulk Extractor can be run against any data, regardless of file system type, or even if there is no file system at all, as is the case with our physical memory images. It is multi-threaded so can make use of multiple cores at the same time. It parses data based on pattern matching, detecting (and extracting in some cases) network packets, e-mail addresses, credit cards and AES keys out into separate report files. Another unique feature is its ability to decompress files that it finds on the fly, adding back into its stream for analysis. So hibernation files, which use the Windows XPRESS format, are easily parsed as well as zip, gzip, rar and the compressed xml format seen in Office files as of Office 2007.[1]

Bulk Extractor was first publicly released in April 2010 and have been through several revisions since then, recently increasing its functionality with the inclusion of a portable executable carver, prefetch file parser and mft record parser.

[1] Bulk Extractor User Manual, dated July 8 2014,
http://digitalcorpora.org/downloads/bulk_extractor/BEUsersManual.pdf



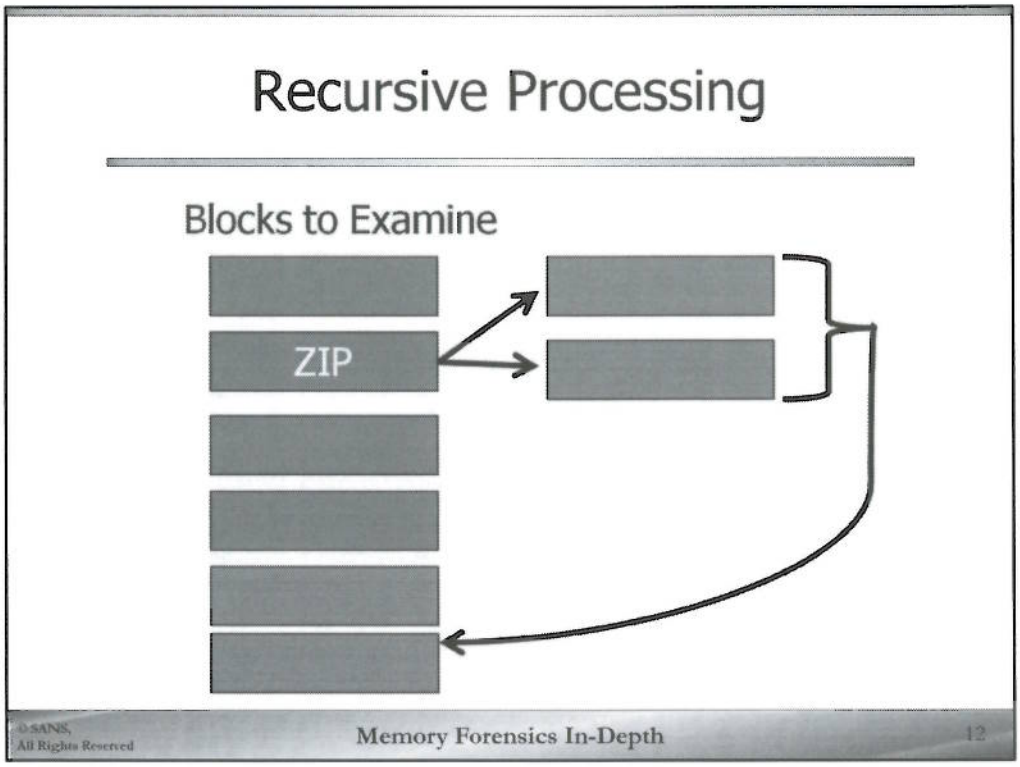
Bulk Extractor is a multi-threaded carving program that accepts a disk image, file or directory of files as data input. Rather than look for headers and footers, however, the program is designed to look for small shards of data. These shards have their own structure which can be validated. Credit card numbers, for example, have a distinct pattern of sixteen digits. Only certain values can be found in the first digit. Other values have a format which can be found via a regular expression. A US telephone number, for example, is three digits, a dash, three digits, another dash, and four digits.

The program uses a number of targeted scanners to parse whatever kind of data is present. The program makes no attempt to utilize filesystem structures for its parsing. That's one of its strengths. This means that the program works even if the filesystem has been damaged, corrupted, or partially deleted or in the case of a memory dump, does not exist at all.

The wordlist bulk extractor recovers, invoked using the "-e wordlist" option" is a fantastic resource for finding passwords. It's equivalent to running 'strings' on the memory image. Often times programs will not clear the buffers which contain a user's password. Their carelessness is our gain! For convenience, bulk extractor outputs a wordlist that can be searched using BE Viewer, the GUI report viewing tool included in the bulk extractor installation. In addition, a version of the wordlist can be directly used with password crackers. Other scanners which are not included by default but may be invoked include:

-e base16 - enable scanner base16
-e facebook - enable scanner facebook
-e outlook - enable scanner outlook
-e sceadan - enable scanner sceadan
-e xor - enable scanner xor

Bulk Extractor now includes a Windows PE scanner, "scan_winpe", that extracts portable executable files and a ELF executable scanner, "scan_elf", that extracts "Executable and Linkable Format" files found largely in Unix/Linux systems. With the latest version of Bulk Extractor, it is also possible to pull FAT32 and NTFS file and directory entries from the target data with the "scan_windir" scanner, carve for rar, zip, jpeg, GPS coordinates and Windows LNK files.

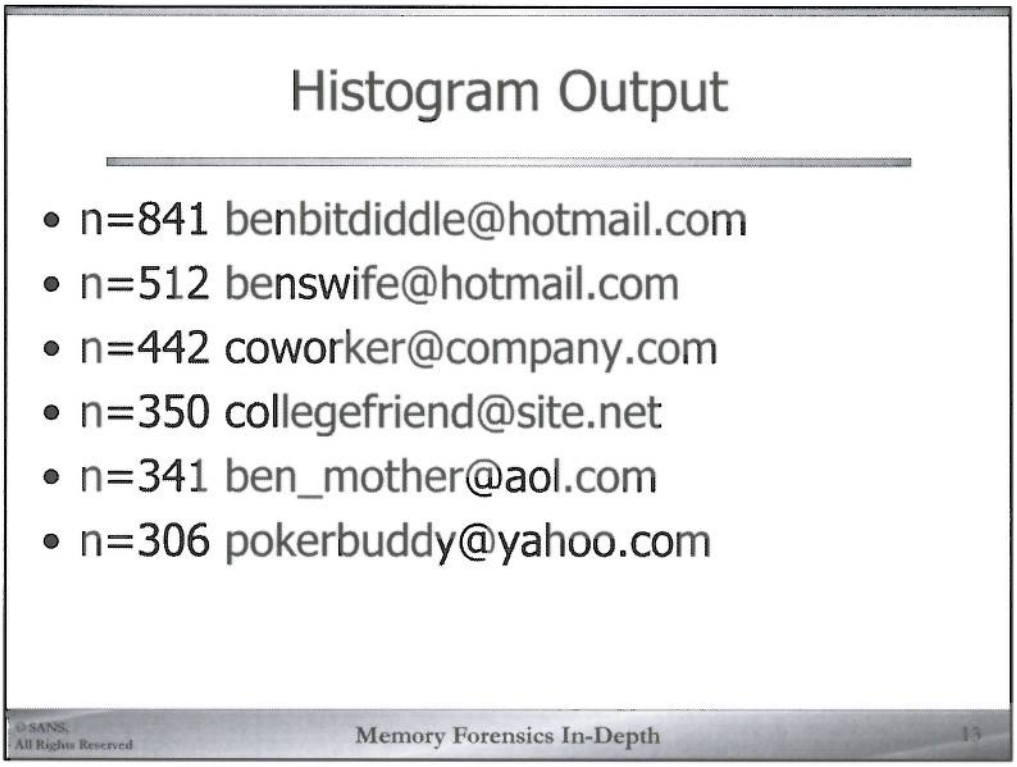


One of the most powerful features of bulk extractor is its ability to recursively process fragments.

The program can recognize fragments which are compressed. For example, parts of a Windows hibernation file or ZIP file. These files can't be carved directly. A compressed phone number, for example, may not be recognizable. A traditional carver, or regular expression matching system, would not detect such a compressed phone number. The examiner would miss that. When Bulk Extractor detects compressed data, it decompresses it and then recursively examines the now decompressed data. This program will keep recursing down, decompressing data as needed, down to five levels by default.

The picture above shows an example of what happens when bulk extractor encounters a zip file fragment, for instance. The fragment is decompressed, and the resulting data is added back into the chunks of data to be examined. The output from the data carved from this zip file will include a

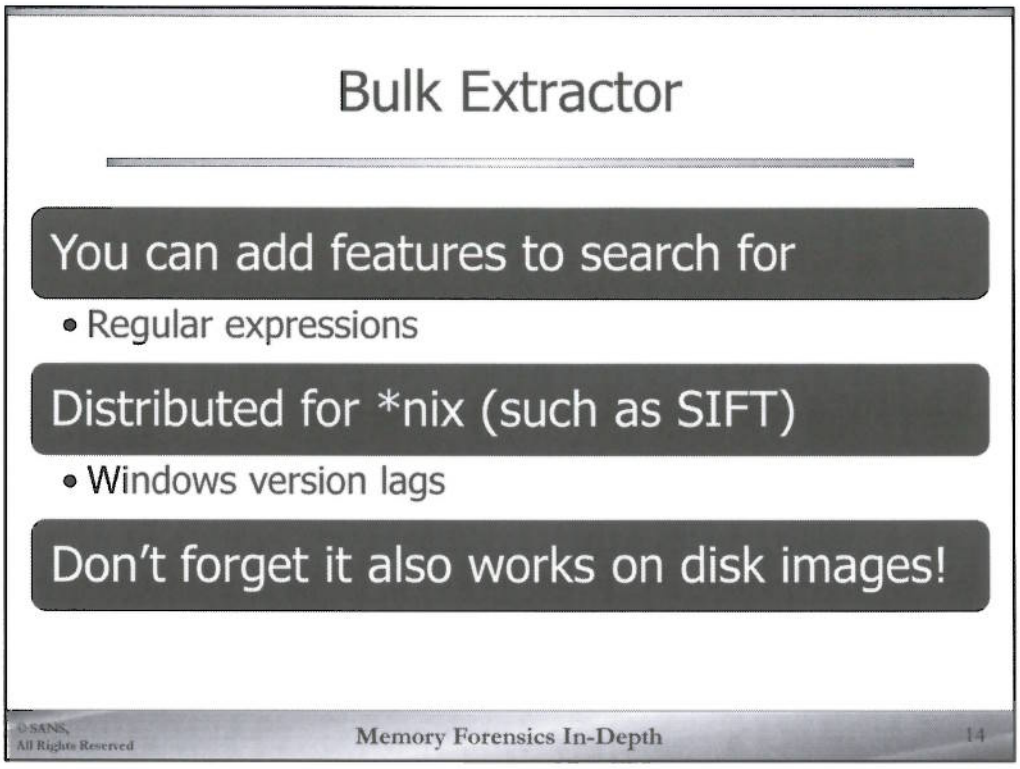
Note that although recursive processing works well on disk images, it does not tend to work as well on memory images where files in memory are much less likely to be contiguous across physical memory.



Along with searching for and recovering all of these features, bulk extractor also computes a histogram of the features it finds. A histogram is a count of how many times a feature occurs. The program then sorts the features, listing those which were found most often at the top. With e-mail addresses, for example, the most frequently encountered address is the address for the computer's owner. The same thing generally applies to memory images, but because there's less data in the input, the owner's e-mail address doesn't always stand out as much.

This picture shows a sample histogram of e-mail addresses. The number after the 'n' is the number of times the feature was found in the input file. The e-mail address benbitdiddle@hotmail.com was found 841 times, the most of any address. Because it was found the most, it is likely this address was the primary address used to send and receive mail on this computer.

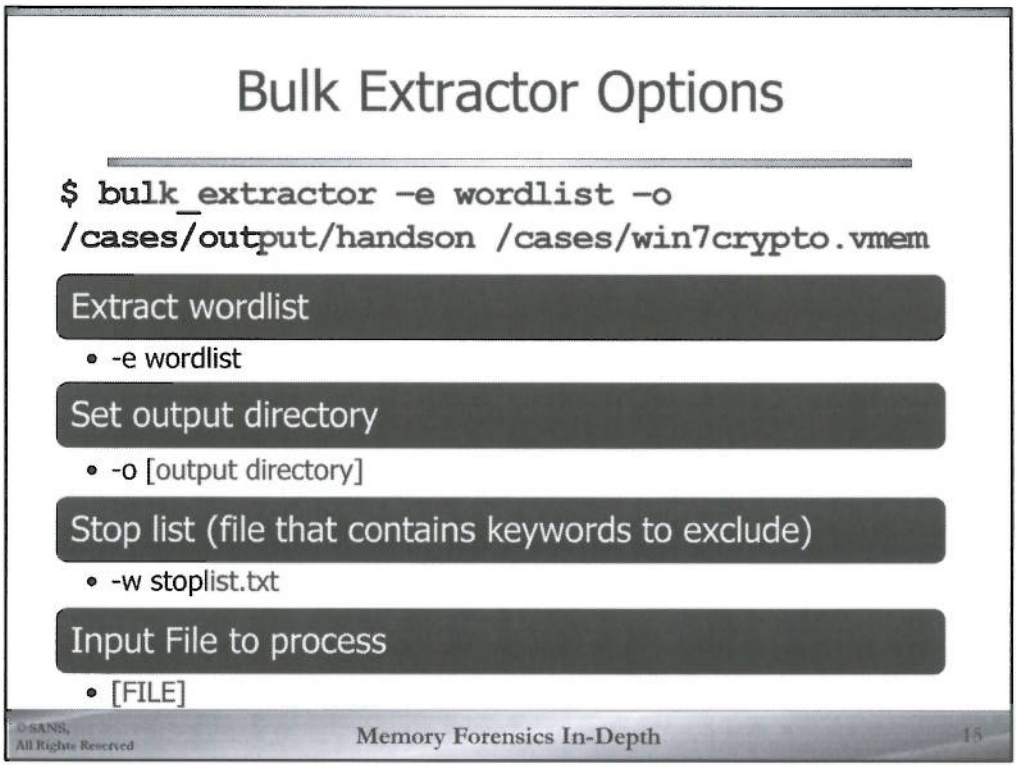
Remember, specificity counts in forensics. All we've done is count how many times this address was found. We can't say anything about who owns that address. It's possible that the user of this computer was using a stolen username and password. Maybe it's their secret e-mail address which none of their friends know about. When you write, describe *exactly* what you find. No more, no less. Unless you're going to be an expert witness or are specifically asked for your opinion, just report the facts.



Users can also add their own regular expressions to look for, but it's a complicated process.

The program is intended for *nix and works best on platforms like OS X and Linux. It is updated frequently with new features and bug fixes. If you become a regular bulk extractor user, you should check the web site for updates often, https://github.com/simsong/bulk_extractor. There is a pre-compiled version for Windows, but it lags behind the *nix version. That is, the Windows release is usually one or more versions behind the current source code.

As mentioned at the beginning of this section, bulk extractor was originally intended to work on disk images, not memory images. If you're running a case with disk images, you should consider using this powerful artifact carver!



The help screen we saw on the previous page details a LOT of command line options. In this course we will only be using one of them. Specifically we want the program to extract a wordlist for us. The option is enabled using:

Extracting wordlist:

```
-e wordlist
```

We will also need to set an output directory as before using the -o flag. Note: This directory must not currently exist. Bulk Extractor will create it.

Set output directory:

```
-o output
```

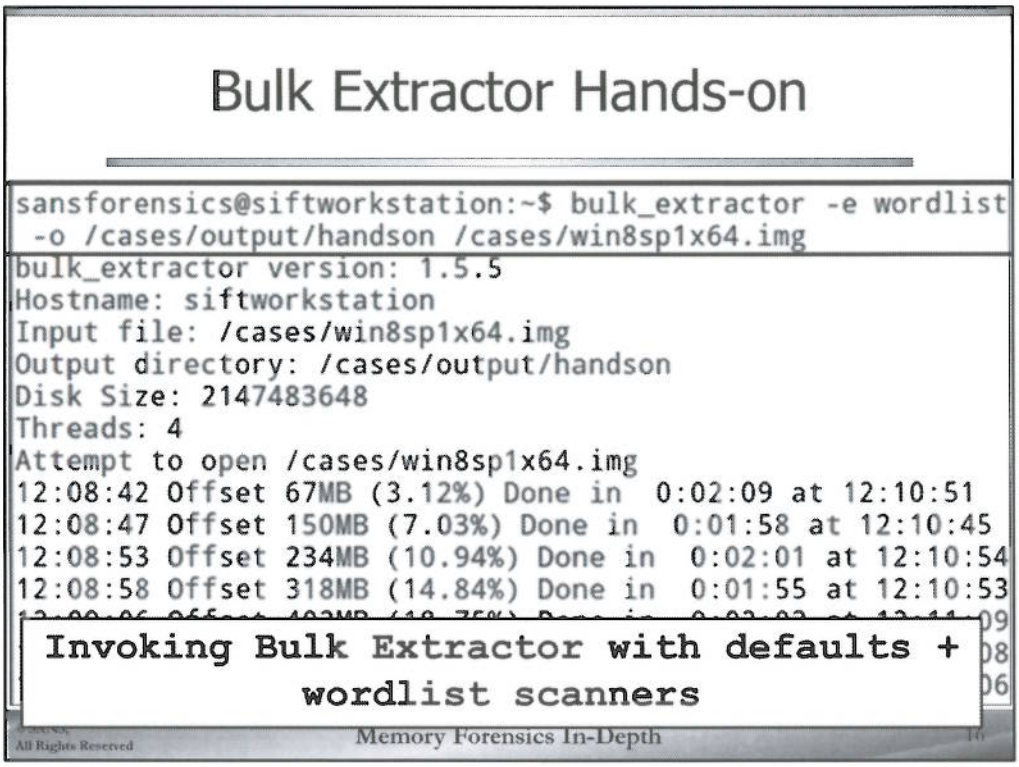
In order to reduce false positives (embedded e-mail addresses for example), a "stop list" can be included. Keywords (ssns, ccns, e-mail addresses, URLs, etc.) included in this file will be excluded from the Bulk Extractor output reports.

Include a stop list:

```
-w stoplist.txt
```

Assign number of analysis threads (processor cores) to use (default is one per core, up to 8):

```
-j #
```



We are now going to run Bulk Extractor on a memory image and look through the results together. This memory image came from a Windows XP system and was captured in 2005. Although it's a little old, it's still perfectly valid. The output will be written to the output/bulk_handson directory.

Here's the command line you want to type:

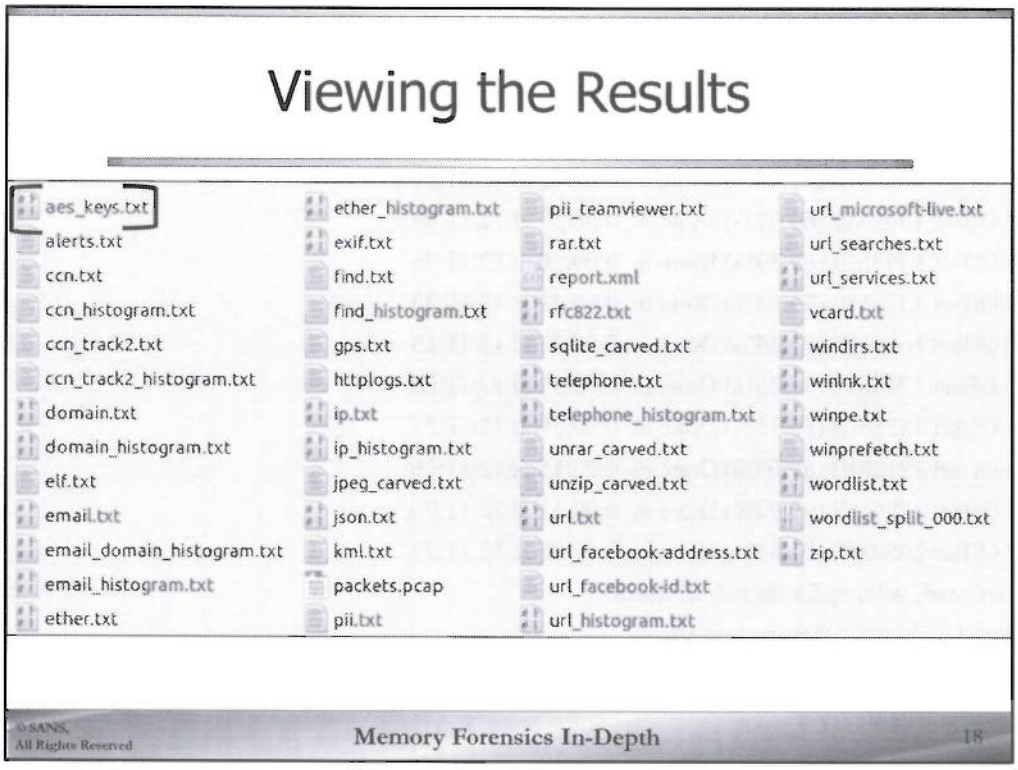
```
$ mkdir cd /cases/output
```

```
$ bulk_extractor -o /cases/output/handson -e wordlist
/cases/win8sp1x64.img
```

As you can see, Bulk Extractor is very verbose and make show errors as it attempts to carve things from memory. This is normal. We should expect that due to the fragmented nature of physical memory, some of the carving of portions of files will fail.

The program's run time depends on how many threads it is using and how much data per run you are parsing.
sansforensics@siftworkstation:~$ bulk_extractor -e wordlist -o /cases/output/handson /cases/win8sp1x64.img
bulk_extractor version: 1.5.5
Hostname: siftworkstation
Input file: /cases/win8sp1x64.img
Output directory: /cases/output/handson
Disk Size: 2147483648
Threads: 4
Attempt to open /cases/win8sp1x64.img

...
12:09:59 Offset 1073MB (50.00%) Done in 0:01:21 at 12:11:20
12:10:05 Offset 1157MB (53.91%) Done in 0:01:14 at 12:11:19
12:10:11 Offset 1241MB (57.81%) Done in 0:01:08 at 12:11:19
12:10:19 Offset 1325MB (61.72%) Done in 0:01:02 at 12:11:21
12:10:27 Offset 1409MB (65.62%) Done in 0:00:57 at 12:11:24
12:10:33 Offset 1493MB (69.53%) Done in 0:00:50 at 12:11:23
12:10:39 Offset 1577MB (73.44%) Done in 0:00:43 at 12:11:22
12:10:46 Offset 1660MB (77.34%) Done in 0:00:37 at 12:11:23
12:10:55 Offset 1744MB (81.25%) Done in 0:00:31 at 12:11:26
12:11:02 Offset 1828MB (85.16%) Done in 0:00:25 at 12:11:27
12:11:08 Offset 1912MB (89.06%) Done in 0:00:18 at 12:11:26
12:11:15 Offset 1996MB (92.97%) Done in 0:00:11 at 12:11:26
12:11:22 Offset 2080MB (96.88%) Done in 0:00:05 at 12:11:27
All data are read; waiting for threads to finish...
Time elapsed waiting for 4 threads to finish:
(timeout in 60 min.)
All Threads Finished!
Producer time spent waiting: 155.73 sec.
Average consumer time spent waiting: 0.237036 sec.
****
** bulk_extractor is probably CPU bound. **
** Run on a computer with more cores **
** to get better performance. **
****
MD5 of Disk Image: ba625dcc991126c6c395e8daab18e4f5
Phase 2. Shutting down scanners
Phase 3. Uniquifying and recombining wordlist
Phase 3. Creating Histograms
Elapsed time: 173.059 sec.
Total MB processed: 2147
Overall performance: 12.409 MBytes/sec (3.10225 MBytes/sec/thread)
Total email features found: 139



Bulk extractor has finished! Let's look through the data we found together.

First, we change the current directory to the output directory we specified in the bulk extractor command line:

```
$ cd /cases/output/handson
```

We then do a directory listing using the ls command.

```
$ ls -l
```

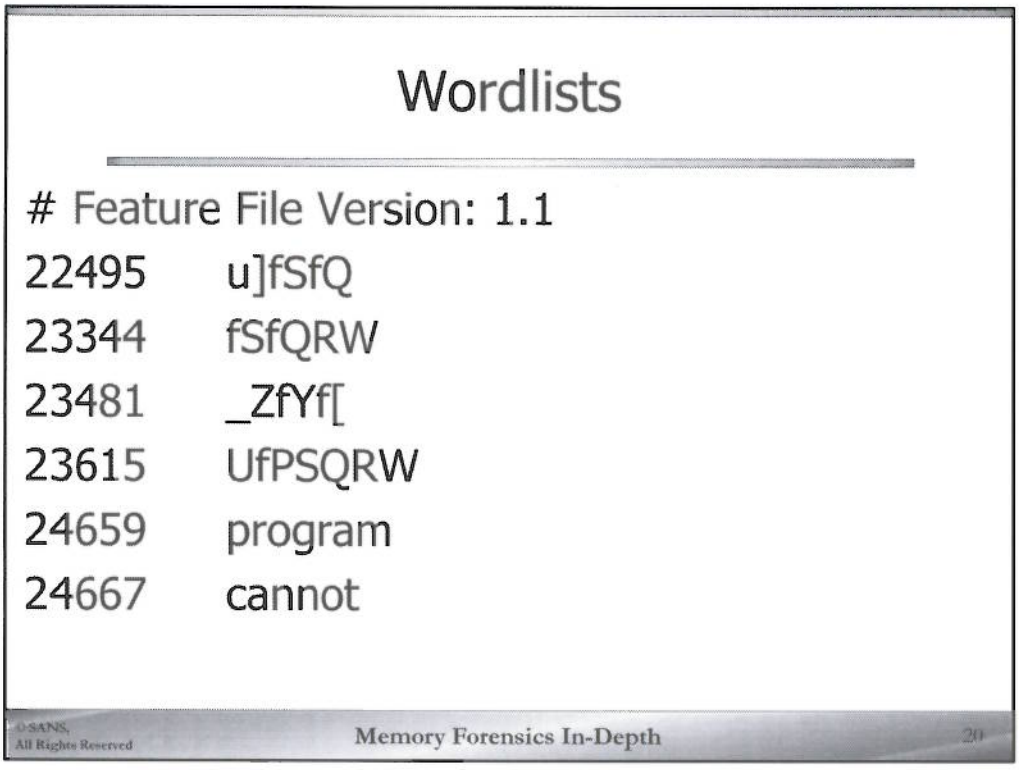
The -l flag displays more details about each file, in particular the file's size. There's no sense looking at files where the program didn't recover any data.

The first file we'll look at is the AES keys file. You can view this file using the cat command.

```
$ cat aes_keys.txt
```

Here is the carved AES data from our Win8SP1x64 image:

```
# BULK_EXTRACTOR-Version: 1.5.5 ($Rev: 10844 $)
# Feature-Recorder: aes_keys
# Filename: /cases/win8sp1x64.img
# Feature-File-Version: 1.1
85200332    5f c0 74 e9 75 67 6a aa 0a 6e 4a 84 95 33 55 5a          AES128
118067888   55 76 da d6 f5 fb 81 89 96 50 c2 09 28 33 3b 68          AES128
588100272   55 76 da d6 f5 fb 81 89 96 50 c2 09 28 33 3b 68          AES128
771858524   91 46 22 b2 ee 34 3c f6 00 13 3a 9f 4f 64 41 4c          AES128
1048867504  55 76 da d6 f5 fb 81 89 96 50 c2 09 28 33 3b 68          AES128
1831727196  91 46 22 b2 ee 34 3c f6 00 13 3a 9f 4f 64 41 4c          AES128
1833054300  91 46 22 b2 ee 34 3c f6 00 13 3a 9f 4f 64 41 4c          AES128
2115793568  3f 4a 50 1b b6 69 0a 76 0a 36 dd bf ac 0a dd cd          AES128
```



The first file we are going to examine is the file wordlist.txt. The results of this file are similar to running the program 'strings' on the input file. It finds every ASCII strings from 6 -14 characters in length in the input file.

```
$ less wordlist.txt
```

This starts a paging program which lets you scroll through the file. The spacebar advances to the next page, and the page-up, page-down keys do what you'd think they do. To get out of less, press the 'q' key. See the command line cheat for more details, such as searching.

The first column in this file is the offset where the string was found and the second is the string.

The wordlist file is usually quite large, and is not extracted by bulk extractor by default. You must explicitly tell bulk extractor to pull it out for you. As a convenience, bulk extractor creates a second version of the wordlist, called, wordlist_split_000.txt, which contains deduplicated, sorted and alphabetized extracted strings.

Because the wordlist file is so large, you probably don't want to review the whole thing by hand. Searching it with other tools is highly recommended. Grep, in particular, can be a huge help. Let's do an example together. Often, in memory, we can recover the strings of environment variables passed to programs. One variable in particular is USERNAME, which contains the login name of the current user. We're going to search for the string "USERNAME=" (without the quotes) in the wordlist file. The equals sign is a special character, and needs to be escaped on the command line. Thus, the command line will be:

```
$ grep USERNAME\= wordlist.txt
```

Running the output, we get something like:

```
434093808       USERNAME=Sarah
434645328       USERNAME=Sarah
448696112       USERNAME=Sarah
456152741       USERNAME=LOCAL
456154472       USERNAME=LOCAL
456730431       USERNAME=Sarah
456732376       USERNAME=Sarah
```

Some of these entries are probably for the user LOCAL SERVICE, but we definitely see the login name on this system was "Sarah".

The grep program is case sensitive by default. If you want to do case insensitive searching, use the -i flag. You can see this in action by searching for sarah with and without the -i flag.

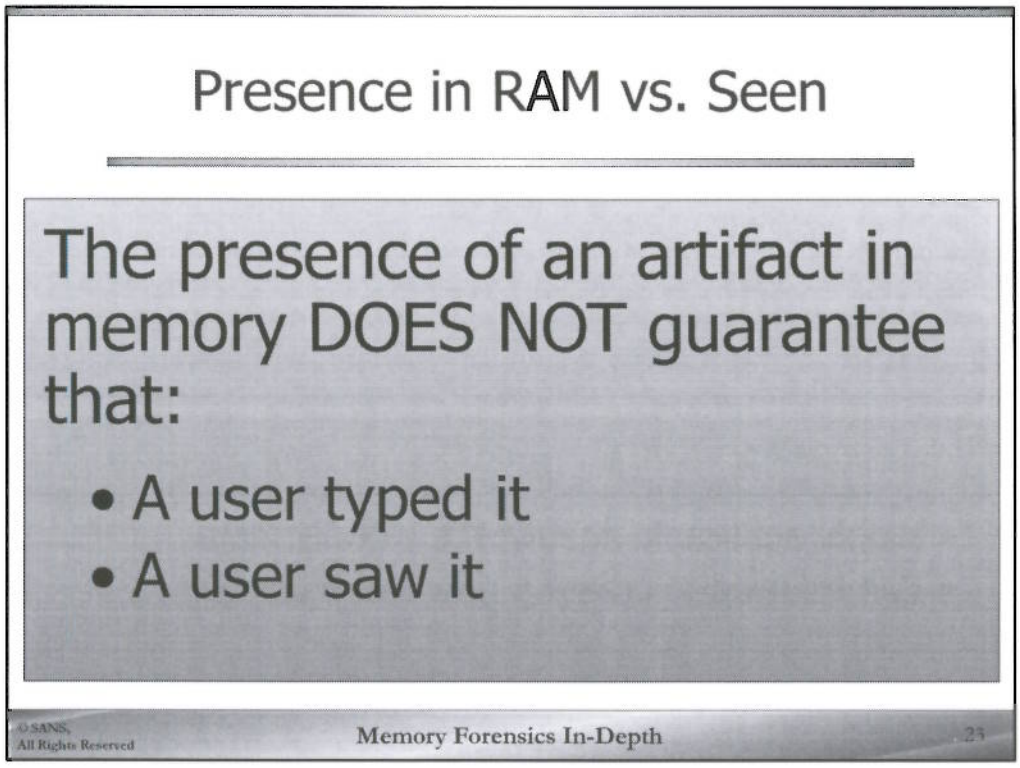
```
$ grep sarah wordlist.txt
$ grep -i sarah wordlist.txt
```

Let's now look at the e-mail addresses which bulk extractor was able to recover. First, let's look at the list of addresses. You can view them by typing:

```
$ less email.txt
```

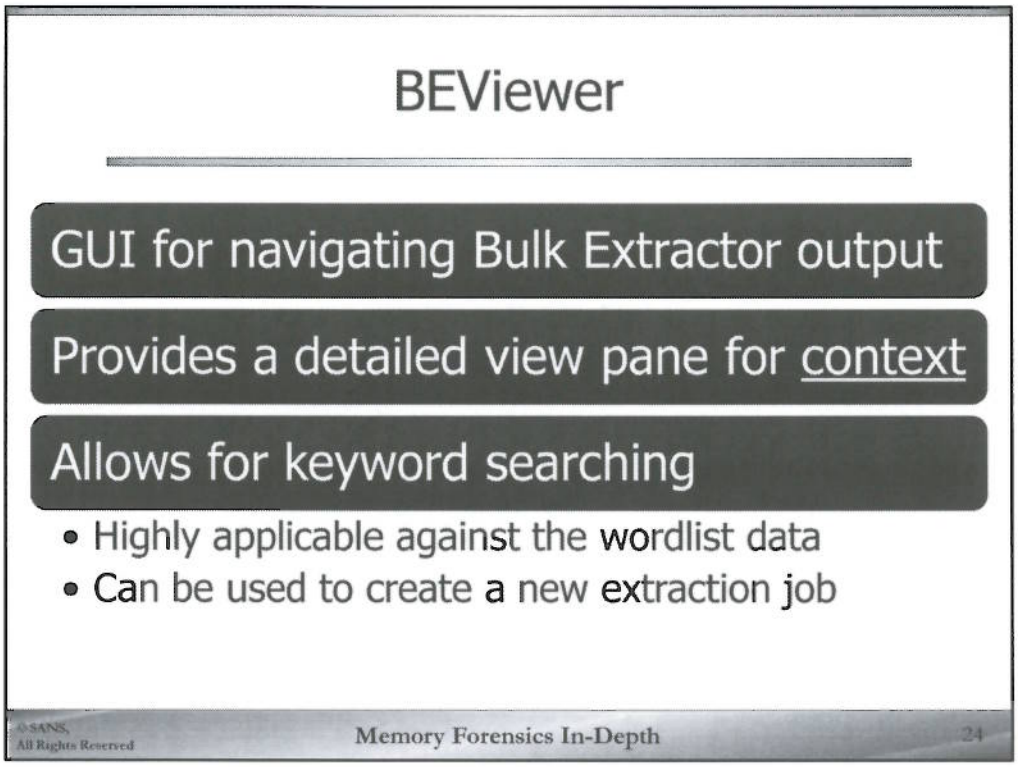
There are a lot of e-mail addresses in this file! Each line in the file contains three attributes. The first is the offset in the file where this feature was found. The second is the feature itself, and the third is the context in which the feature was found.

In the example on the slide, the program found the e-mail address "webmaster@espn.go.com" at offset 20297198. That address was in the context of some kind of tag for "Managing Editor".



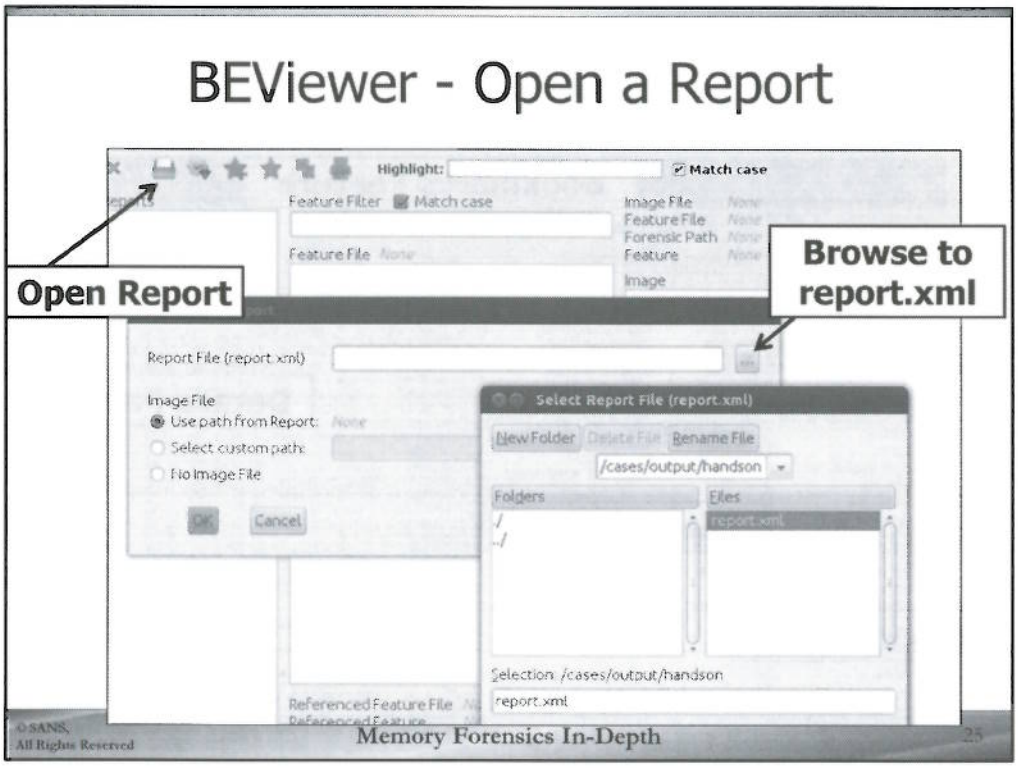
You will immediately notice that the most common e-mail addresses in the memory image were probably not ones which the user typed, or even saw. These addresses can be excluded through the use of a stop list. A comprehensive "stop list" is available for download from http://digitalcorpora.org/downloads/bulk_extractor/be15-stoplists.tar.gz as a reference of default/generic artifacts loaded by programs, drivers, or other operating system components. For example, the presence of the domain: trustcenter.de was most likely due to certificate signing for web traffic. The user never typed, and probably never saw this process occurring.

The same limitation applies to the other kinds of data we're about to examine. The URLs, including search string URLs, may not have been typed by the user. Proceed with caution!

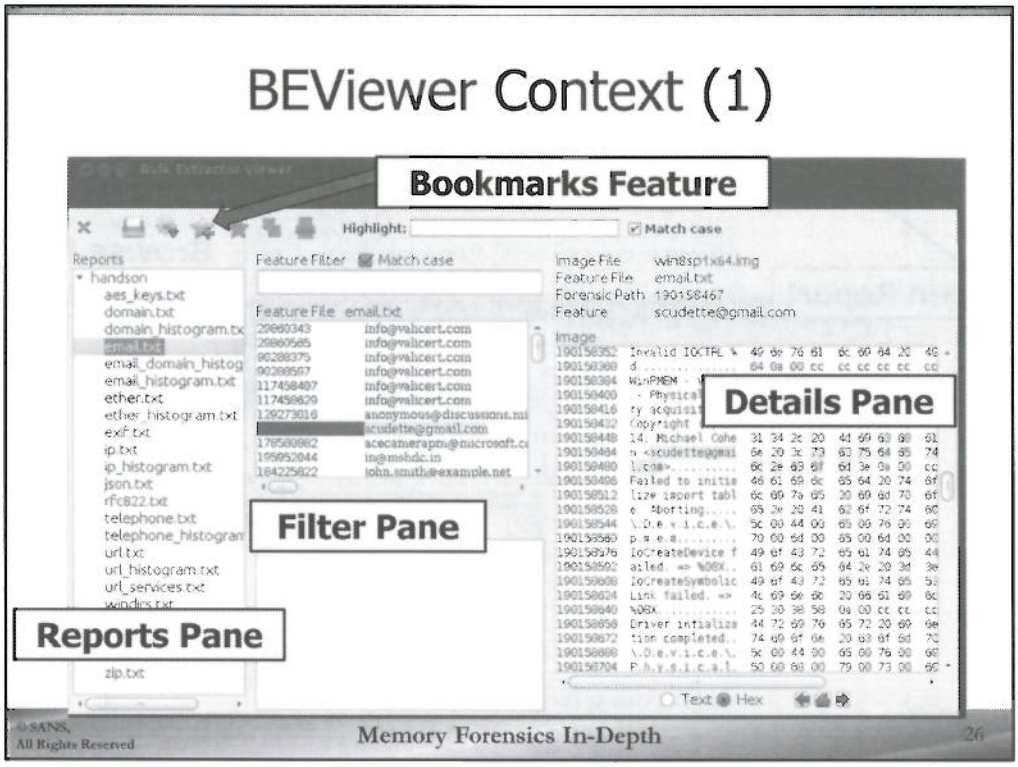


For those analysts who prefer to use a GUI, Bulk Extractor has a easy-to-use user interface tool called BEViewer. The BEViewer is included in the SIFT workstation and is accessed from the command line by typing **`BEViewer`**.

More information can be obtained on Bulk Extractor and BE Viewer from https://github.com/simsong/bulk_extractor . There is also an active Google Group for BEUsers: http://groups.google.com/group/bulk_extractor-users

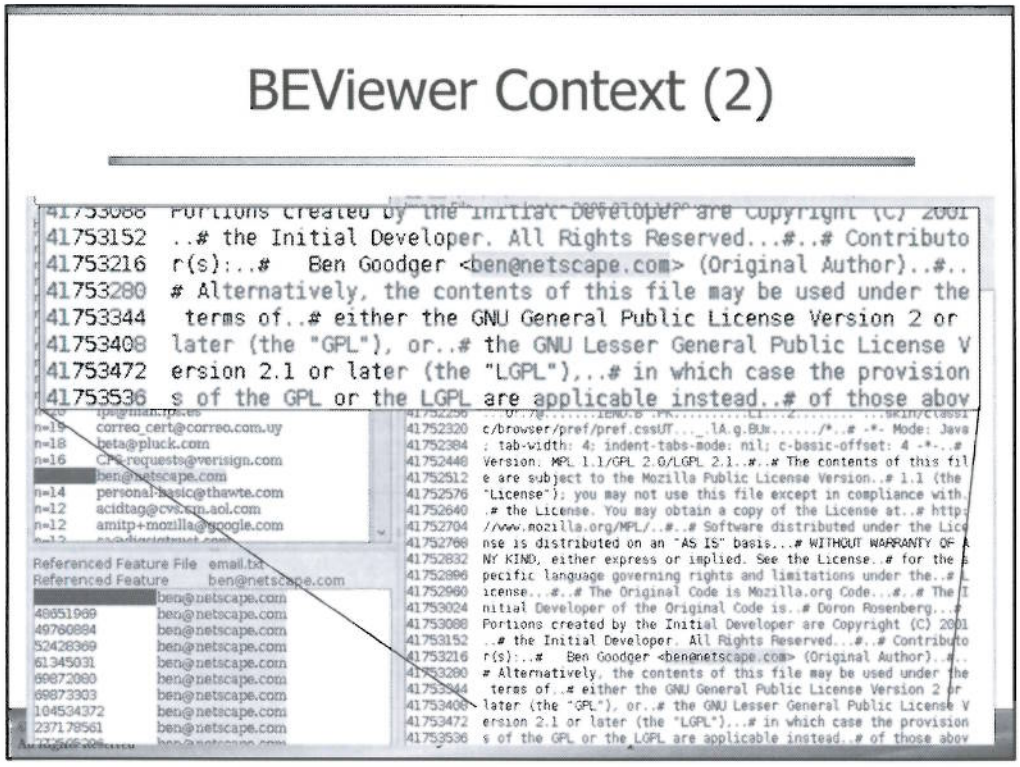


By opening the report.xml file found in the output directory of a bulk_extractor job, the BEViewer tool can be used to view and parse through the data and histogram reports of the target image file. Upon loading the report.xml, ensure that the path to the image file is correct or the tool will be unable to load the details pane with the data pointed to by the associated bookmarks.



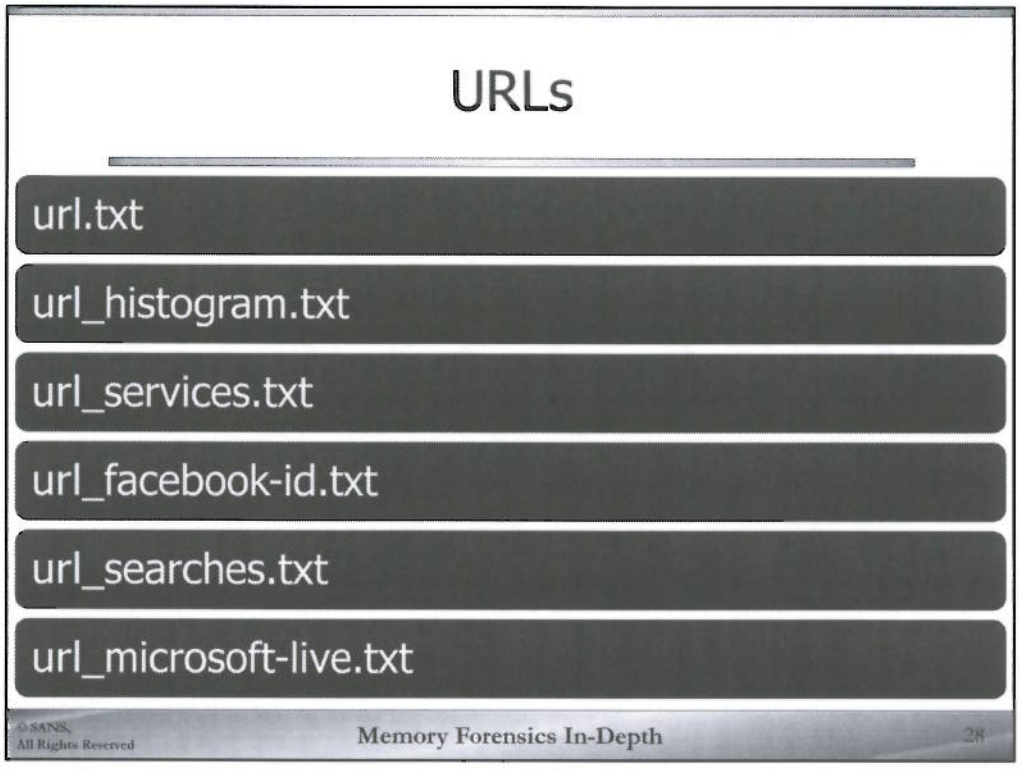
The BEViewer reports pane displays the results of a Bulk Extractor job that can include default, wordlist, and net scanners, in addition to the default output files. By using the viewer, an analyst can search through the output for specific keywords, viewing the data in hexadecimal or text view in the far right details pane. In addition, the Filter pane offers a "Match case" that provides case sensitive keyword search functionality.

The BEViewer tool has recently added a copy function for highlighted data in the active pane. Note that BEViewer also has a "bookmark" feature that enables a user to capture the entire contents of the details pane and save the data to a file. In order to create a bookmark file, click the star icon in the toolbar and then go to File>Export Bookmarks.



Using BEViewer, it is easy to see that one of the e-mail addresses that was seen over 35 times in the xp-laptop-2005-07-04-1430.vmem memory image is clearly not a reference to an e-mail address that was being used by a mail client, but contact information for a Mozilla contributor. Having visibility of the data surrounding a keyword hit in the detailed view pane allows for further analysis and data deduction of false positives. In addition, by using the ***stop list*** (or whitelist) feature of Bulk Extractor, these benign hits can be removed from feature file output. To create a stop list, compile all keywords that should be ignored into a text file and include the **–w <stop_list.txt>** option.

```
$ bulk_extractor -o /cases/output/handson -e wordlist -w stoplist.txt
/cases/win8sp1x64.img
```



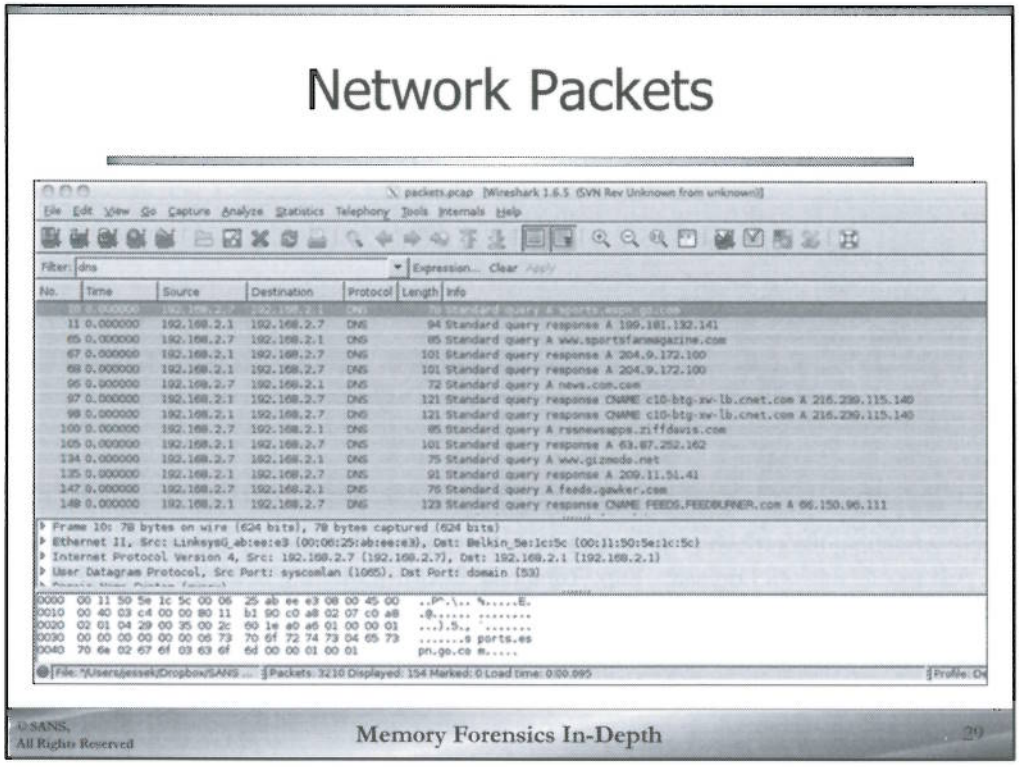
There are several pertinent files related to URLs. First, the url.txt contains a complete listing of every URL found in the input file. As with the other types of data, there are a LOT of entries in this file. Parsing it by hand can be difficult. You may wish to start with the histogram of URLs. Remember, with memory images these are the URLs which were found in the memory image. This may mean the URLs appeared in a web page, or even just in a program, but not necessarily that they were visited!

The last four URL files are special values which bulk extractor has highlighted for you. The url_services.txt is a histogram of the URLs, but instead of comparing the entire URL, only looks at the domain name involved in the search.

The file url_facebook-id.txt contains any Facebook id numbers which were found in URLs. They are listed in the form "id=[number]". To view the profile associated with that id number, go to https://www.facebook.com/[number]. This will redirect your browser to the person's username page.

The url_searches.txt file contains potential search terms found in URLs. This can be useful for examining what a user has searched for on the Internet. Again, the presence of a search term in this file doesn't mean the user actually searched for it! It could have been found in a link on a web page, but the user didn't click on it.

The Microsoft Live file refers to ids for the Microsoft Live service.



Network packets are the data being sent or received via the network. Those data are wrapped into TCP and IP packets before they can be put on the wire. Those structures live in memory. And just like the disk, those structures stay in memory until they are overwritten. This means we can capture them in a memory image!

The number of network packets in a memory image depends on how much memory was being used on the system and how much the network was being used. If the operating system needed to create and send lots of packets, the memory used to store those packets will be reused more often, and the history of network traffic won't go back very far. But if the system was lightly loaded, a memory image may capture several minutes of full network content.

The network scanner is now included in the default scanners of Bulk Extractor. It carves network packets and create the file "packets.pcap", a PCAP formatted file. These are generally not found in disk images, but a wealth of them are usually in a memory image.

There are several programs you can use to view or parse these data. The program Wireshark, included on your SIFT workstation, is a great graphical program for doing so. To run Wireshark, at the command prompt type:

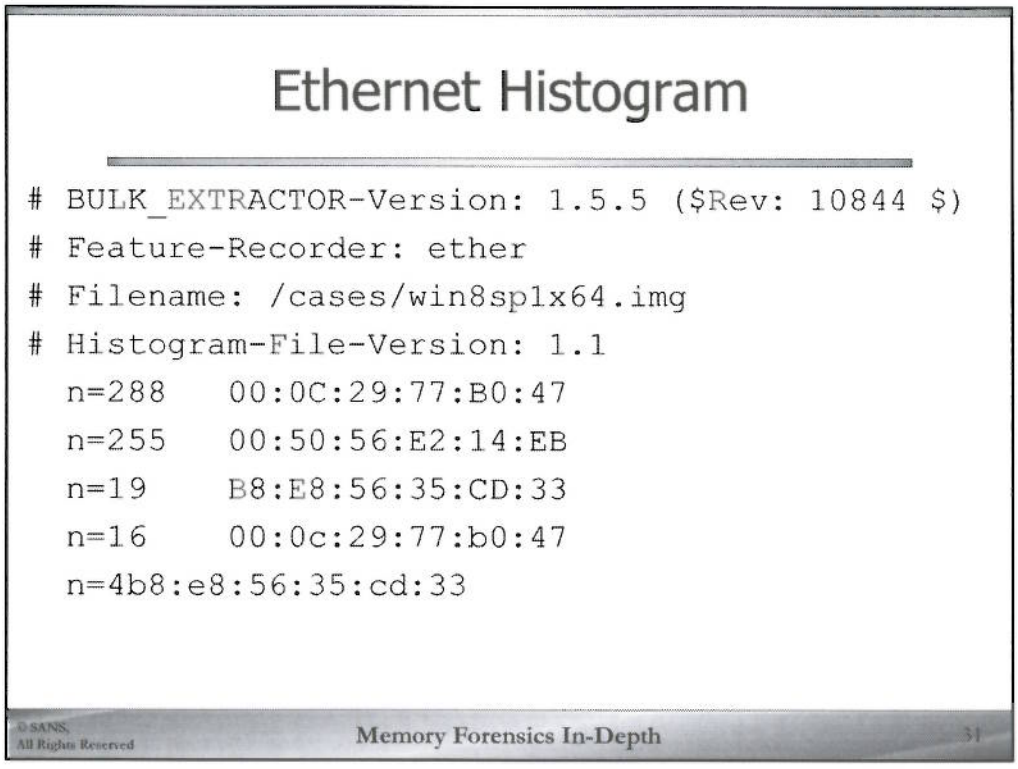
```
$ wireshark packets.pcap
```

The window you see now is a listing of all of the potential packets found. There may be invalid data in here, along with duplicate packets, and things will almost certainly be out of order. That's just what the program found in the memory image. Note that the "Time" field is zero for all packets. The timestamps are not preserved in the memory image. Normally this field corresponds to the start of the capture. But since all of these packets were captured together, there is no difference in the time.

You can scroll up and down the list of packets and examine the traffic. To highlight some pertinent data, try using the Wireshark filters. The filter toolbar is at the top of the screen. Some filters you may find helpful are http and dns. In the screenshot above, we are filtering on DNS packets. We can see the DNS requests made and some of the responses. What domains were requested? What were the responses?

```
Request: www.sportsfanmagazine.com
Response: 204.9.172.100
Request: news.com.com
Response: 216.239.115.140
Request: rssnewsapps.ziffdavis.com
Response: 63.87.252.162
Request: gizmodo.net
Response: 209.11.51.41
```

We are barely scratching the surface of Wireshark here. It is a powerful program and covered extensively in SANS 572, Advanced Network Forensics, http://computer-forensics.sans.org/training/course/advanced-network-forensics-analysis#

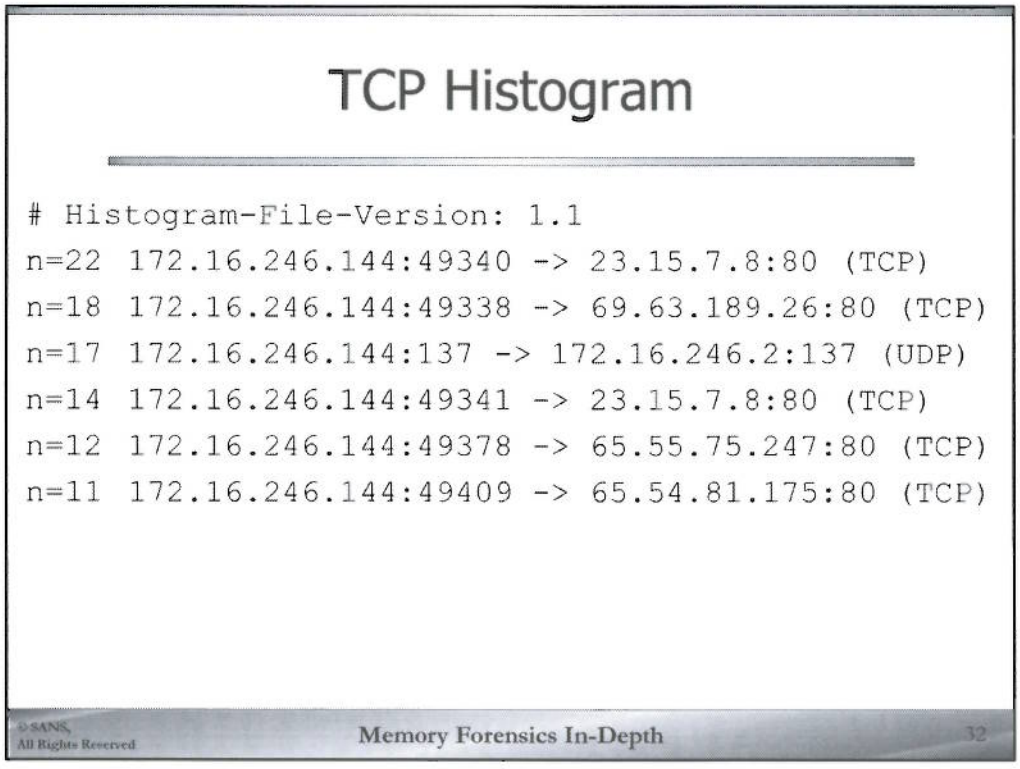


MAC addresses are a globally unique identifier for a network card. They are assigned by the manufacturer and cannot be changed. Each MAC encodes which company made the device and a serial number for that device.

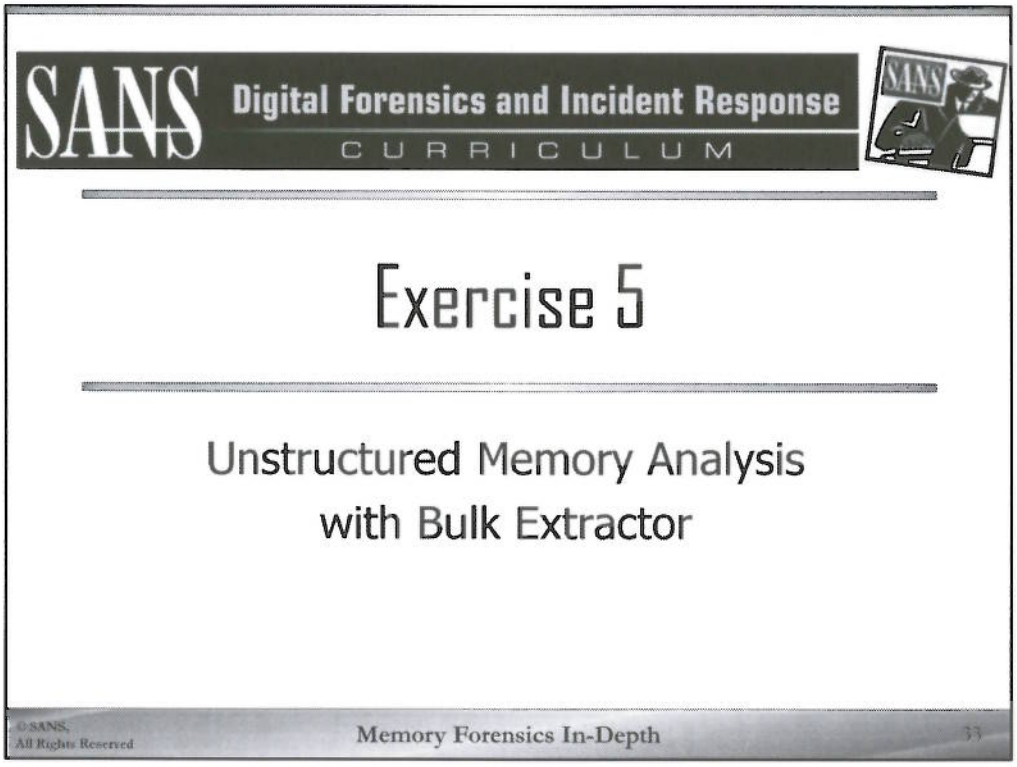
Being able to uniquely identify the source of network traffic can be critical for an investigation!

Looking at the file ether.txt, we can see the MAC addresses found in the memory image. There certainly are a lot of them! To make sense of them, it's often easier to look at the histogram of MAC addresses. Looking at the histogram, ether_histogram.txt, we can clearly see two MAC addresses were found quite often. The rest were found rarely. We can't tell which of the two most common belonged to whom, but almost certainly one of these belonged to the computer from which we got this memory image. The other almost certainly came from the router it was communicating with.

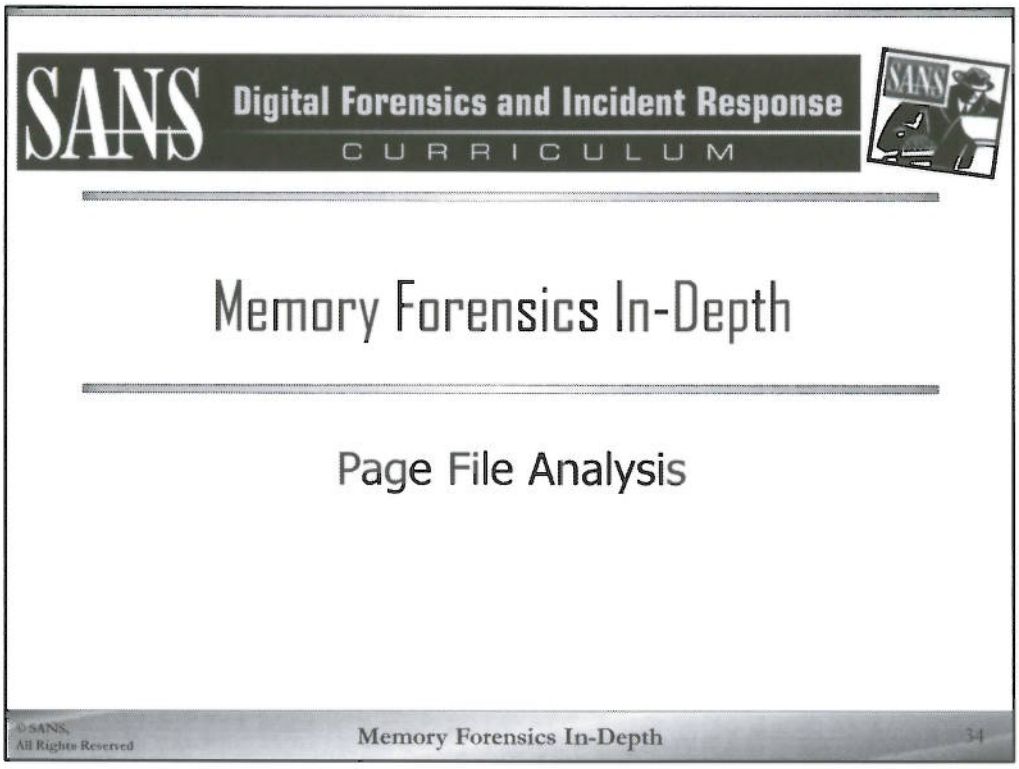
Ethernet addresses may be used to infer proximity to a wireless access point, which may be useful for placing the user at a particular location (e.g. a coffee shop that offers public wireless).



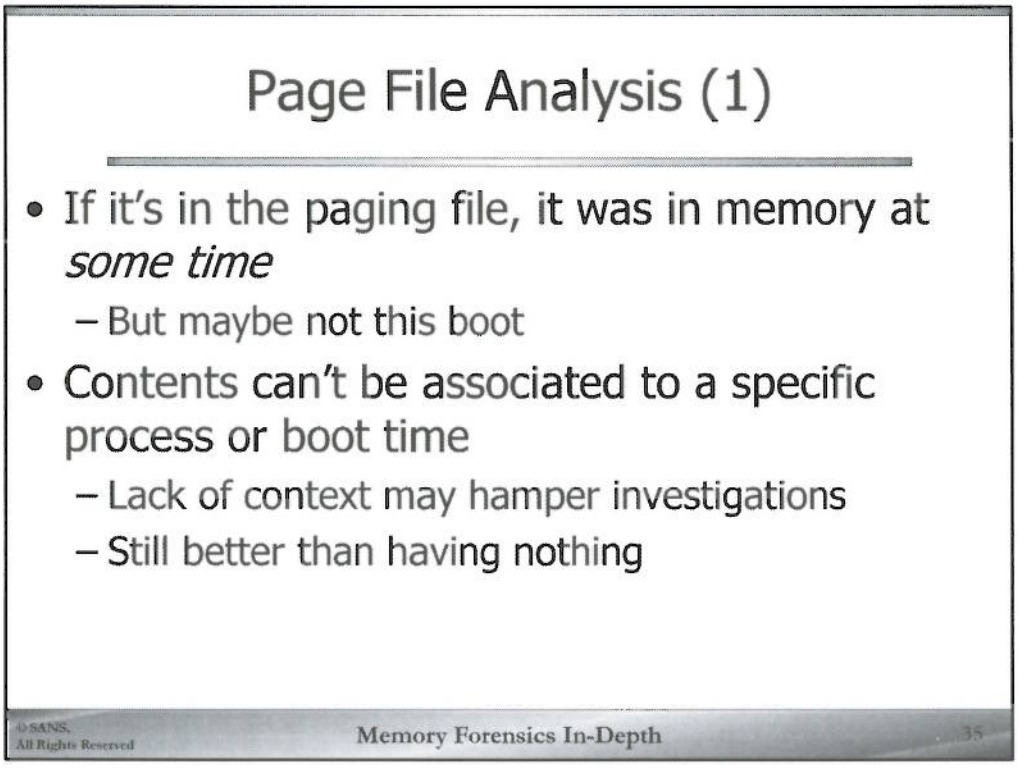
One of the options in Bulk Extractor's net scanner is the TCP histogram report. Disabled by default, it can be invoked using the "-S carve_net_memory=YES" command line parameters. The TCP histogram contains a summary, ordered by frequency, of all extracted UDP and TCP packets. In examining this data, we see that the "top talkers" are the target system, 172.16.246.144, and a remote system, 23.15.7.8. With this inclusion of the port numbers, we can ascertain that the target system was communicating to the remote host via http. False positives are possible with this data and it does not indicate a full list of connections.



This page intentionally left blank.

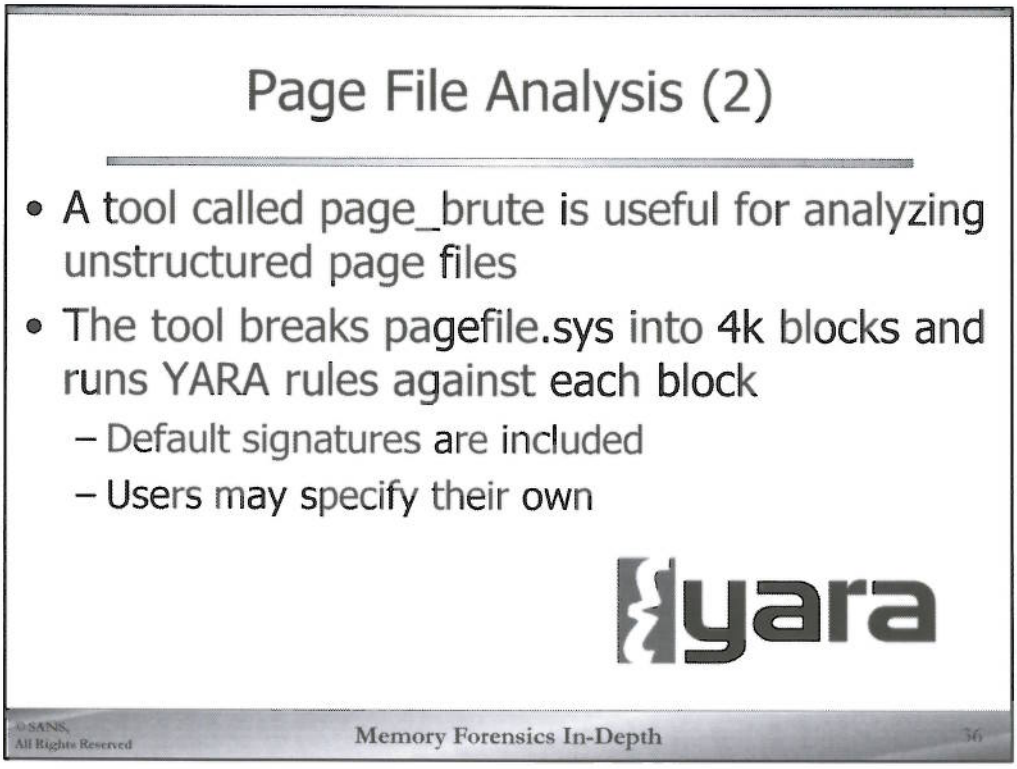


This page intentionally left blank.



In addition to analyzing memory dump files, we can also analyze the page file. Everything in the page file was in memory at one time or another and is not directly backed by another on-disk file. It is important to note that Windows does not track entries in the page file with the a specific process on disk (only in memory, at run time). This means that an interesting entry in the page file cannot be associated with a specific process of interest. This lack of context may hamper investigations. Additionally, because the default page size is 4kb, you may find interesting information spread across non-contiguous blocks.

However, note that sometimes just the existence of a specific artifact may be enough to aid in an investigation. For instance, some machines should never process regulated data (PCI, PII, HIPAA, etc.) and the mere existence of such data is indicative of a policy violation.



The page_brute tool breaks a larger file (such as a pagefile.sys) into 4KB chunks and runs a set of YARA rules against the file. While this tool is fairly simplistic in operation, it is sufficient to discover a number of artifacts in the page file due to it's default set of YARA signatures.

Users can write their own signatures, but the default signature set includes the following categories:

administrative_share_abuse

remote_system_syntax

http_request_header

http_response_header

webartifact_html

webartifact_javascript

cmdshell

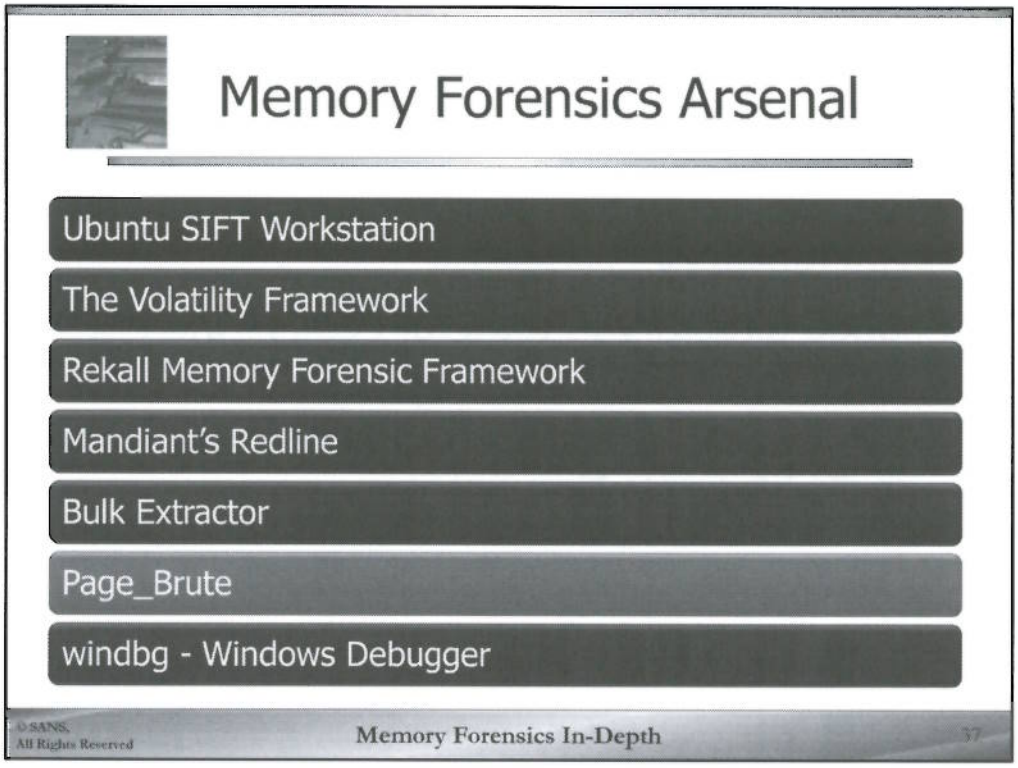
webartifact_gmail

social_security_syntax

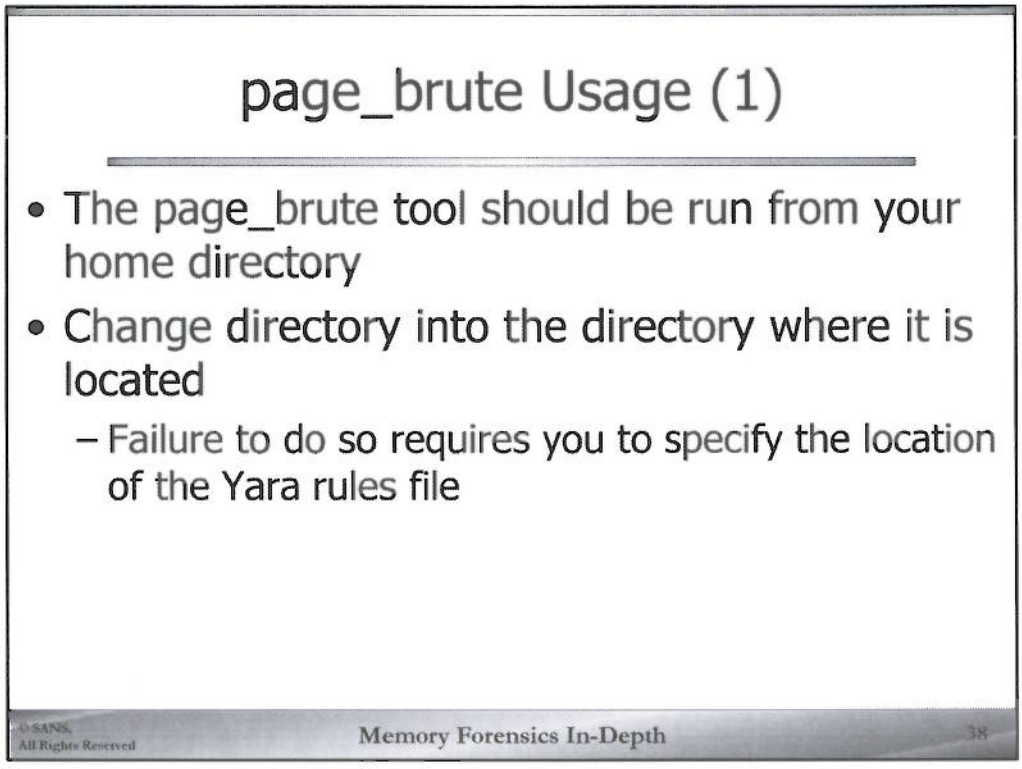
smtp_fragments

irc

ftp



Page_Brute is the first tool we will introduce in this course that supports the use of YARA rules, allowing the examiner to incorporate signatures based on malware or behavioral indicators easily into current investigations.

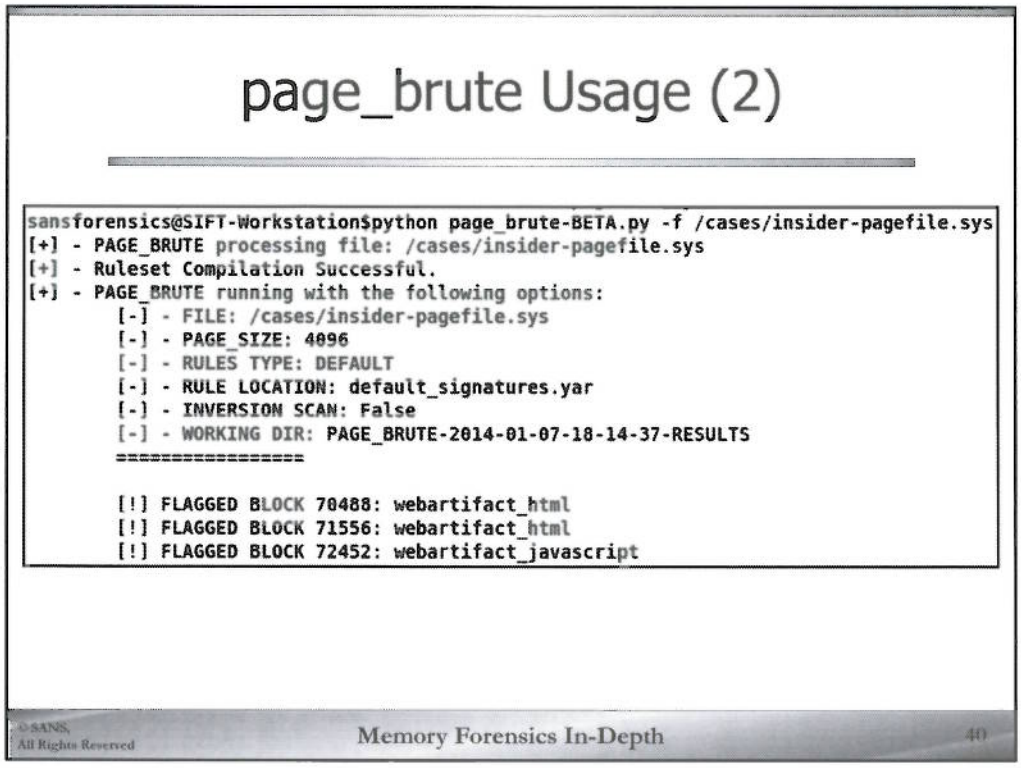


The page_brute tool by Michael Matonis - @matonis (https://github.com/matonis/page_brute) is a fairly simple python script that breaks a pagefile into 4k chunks and runs yara signatures against each chunk.

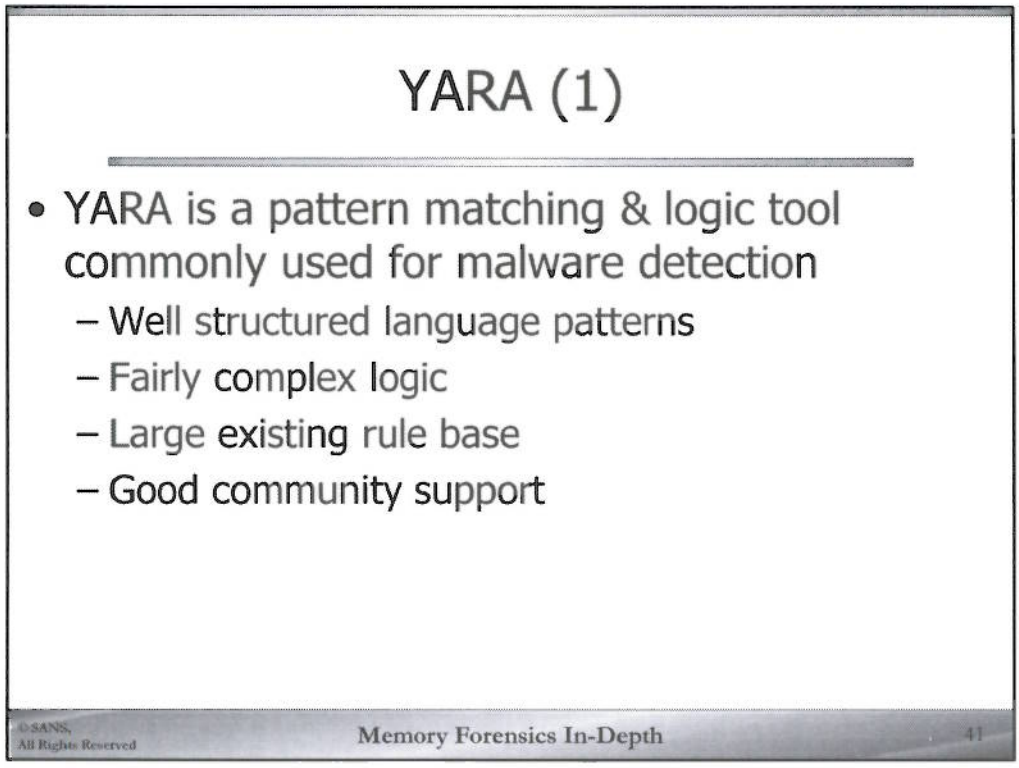
Because of the way the tool is written, you should use the copy locally in your home directory. You should change into the page_brute directory before executing the tool or you will have to specify the location of the yara rules.

```
root@SIFT-Workstation:~/page_brute# page_brute-BETA.py
usage: page_brute-BETA.py [-h] [-f FILE] [-p SIZE] [-o SCANNAME] [-i]
                          [-r RULEFILE]

Checks pages in pagefiles for YARA-based rule matches. Useful to identify forensic
artifacts within Windows-based page files and characterize blocks based on regular
expressions.

optional arguments:
  -h, --help            show this help message and exit
  -r RULEFILE, --rules RULEFILE
                        File/directory containing YARA signatures (must end  with
.yar)
-f FILE, --file FILE  Pagefile or any chunk/block-based binary file
 -p SIZE, --size SIZE  Size of chunk/block in bytes (Default 4096)
```

```
-o SCANNAME, --scanname SCANNAME
                              Descriptor of the scan session - used for output
directory
  -i, --invert          Given scan options, match all blocks that DO NOT match a
ruleset
```



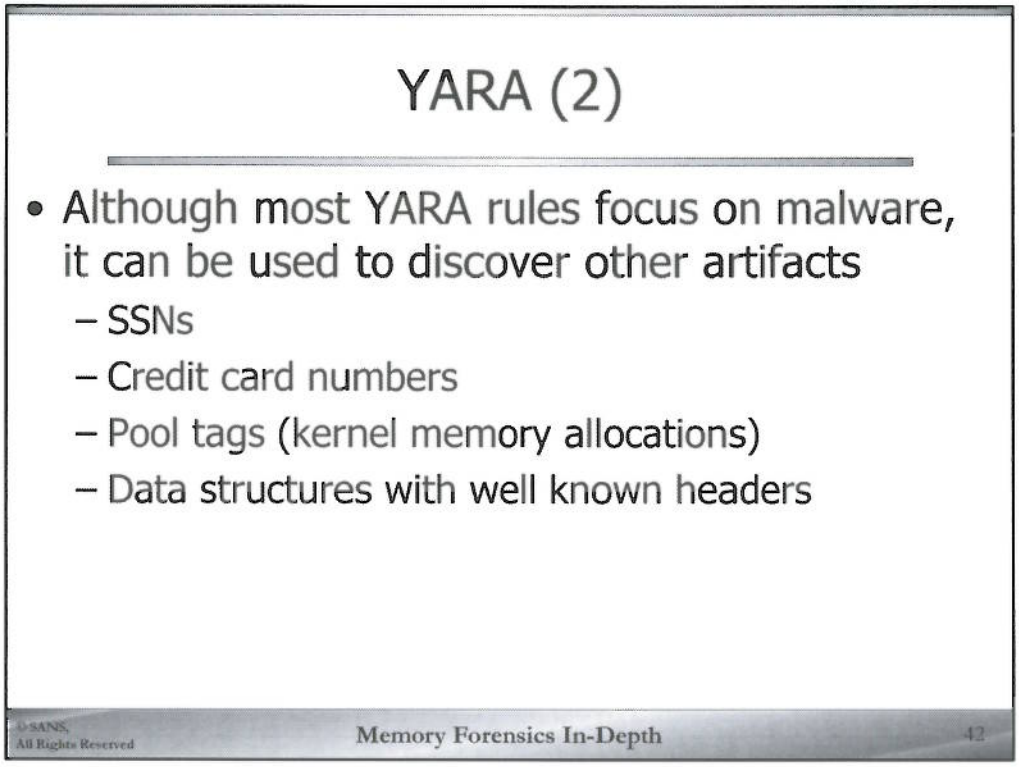
In our case, we'll just invoke page_brute with default signatures and an output directory of "insider_case". If the output directory isn't specified, page_brute will simply output in a directory formatted with the run time (e.g. PAGE_BRUTE-2013-12-13-20-52-11-RESULTS). Note that this directory name does not include the name of the file from which results were obtained. This can cause confusion if multiple runs of the software are made against different page files. This may also cause confusion if multiple runs are made against the same page file but different Yara signatures are used on each run.



From the YARA user's manual:

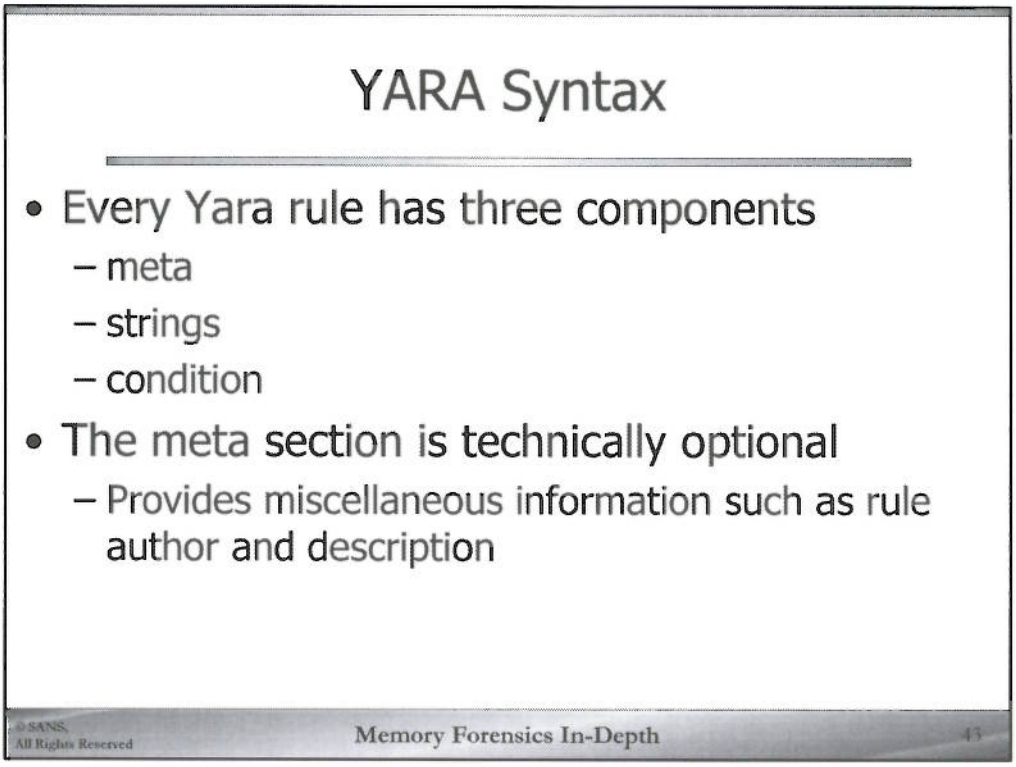
> YARA is a tool aimed at helping malware researchers to identify and classify malware families. With YARA you can create descriptions of malware families based on textual or binary information contained on samples of those families. These descriptions, named rules, consist of a set of strings and a Boolean expression which determines the rule logic . Rules can be applied to files or running processes in order to determine if it belongs to the described malware family.

For more details on how to build YARA signatures, see the yara website's most recent user manual at http://yara.readthedocs.org/en/v3.2.0/

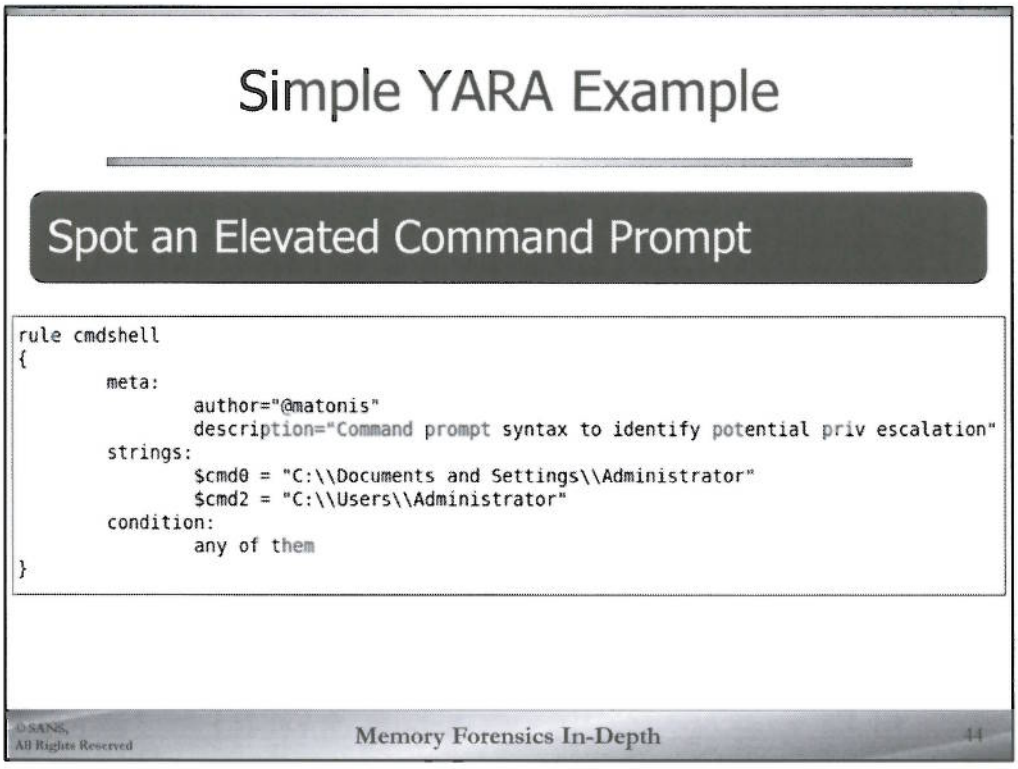


YARA has traditionally been used to discover previously unknown malware and to group malware into families. However, YARA can also be used for other data discovery. If you've ever wanted to search for text or binary patterns that couldn't be accomplished with a simple grep, YARA is for you! No programming required, so the old (and frustrating for some) advice of "you should write a script to do that" doesn't apply.

As long as the data you are searching for is defined, YARA can be used to find it. This includes SSN's, Credit Card Numbers, Pool Tags, and other data structures that have some predictable content to match on.



The most basic YARA rules contain only one or more strings to search for and conditions that specify how many of the strings must be matched. The rules range from basic to insanely complex. They support ASCII and unicode, byte patterns, regular expressions, and a number of complex conditional options. Additionally, conditions can specify that strings can be at related offsets (i.e. string1 is found 20 bytes from string2).



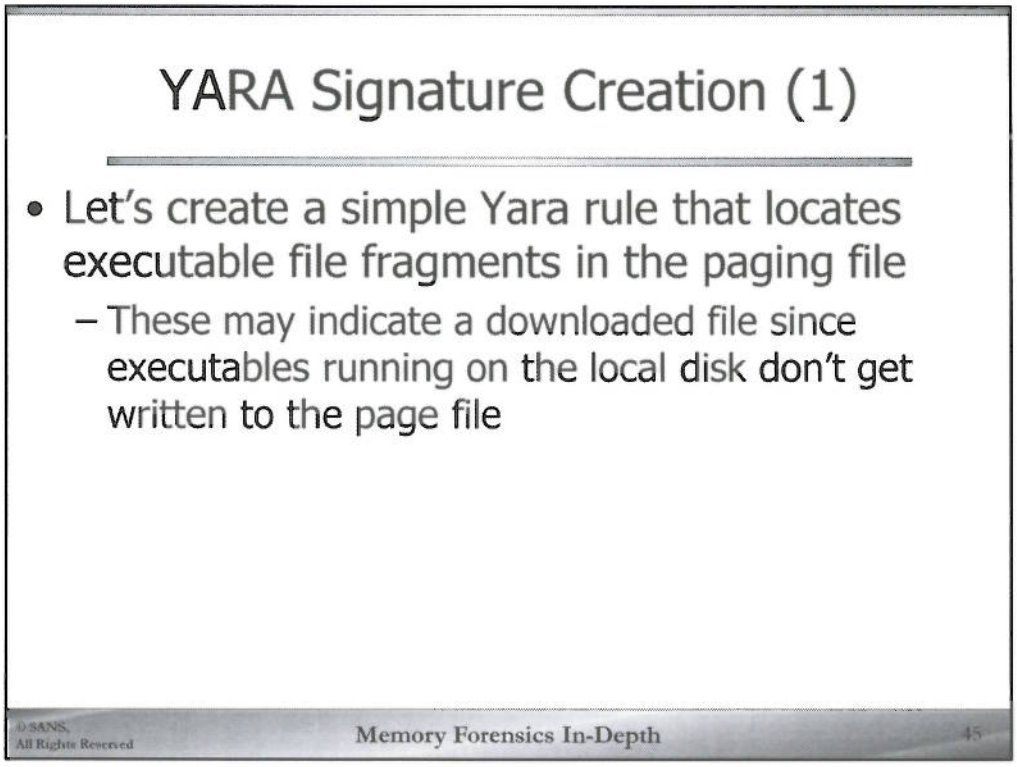
Let's examine this simple rule. This rule seeks to discover memory sections which may indicate that a process is using an elevated command shell. We won't debate whether this rule may have false positives (it does) or whether an elevated command prompt might be open that isn't caught by the rule (it might). This rule was taken directly from those included with the page_brute tool, which you'll use in the course.

The first section, meta, contains two directives which are fairly obvious in their usage. We won't examine them further here. As mentioned previously, the meta section is optional.

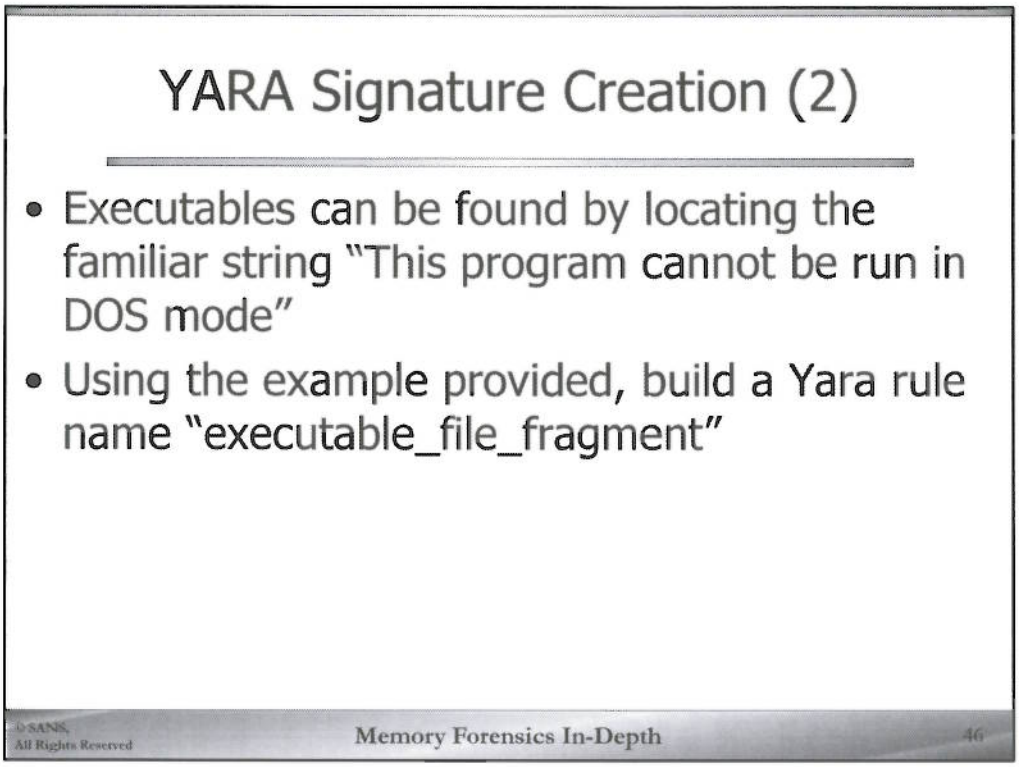
The next section, strings, lists one of more strings that may be matched by the rule. Although many complex options are supported, this rule doesn't use any of them. It simply matches literal strings.

The final section, condition, specifies the conditions under which the rule will be considered a match. In this case, it simply says that if any strings are matched, then the rule is considered a match. Again, more complex options are available (but we wanted to start out with a trivial rule).

If a rule is matched, (according to the strings and condition) then the name of the rule (cmdshell in this example) is displayed on the command line.

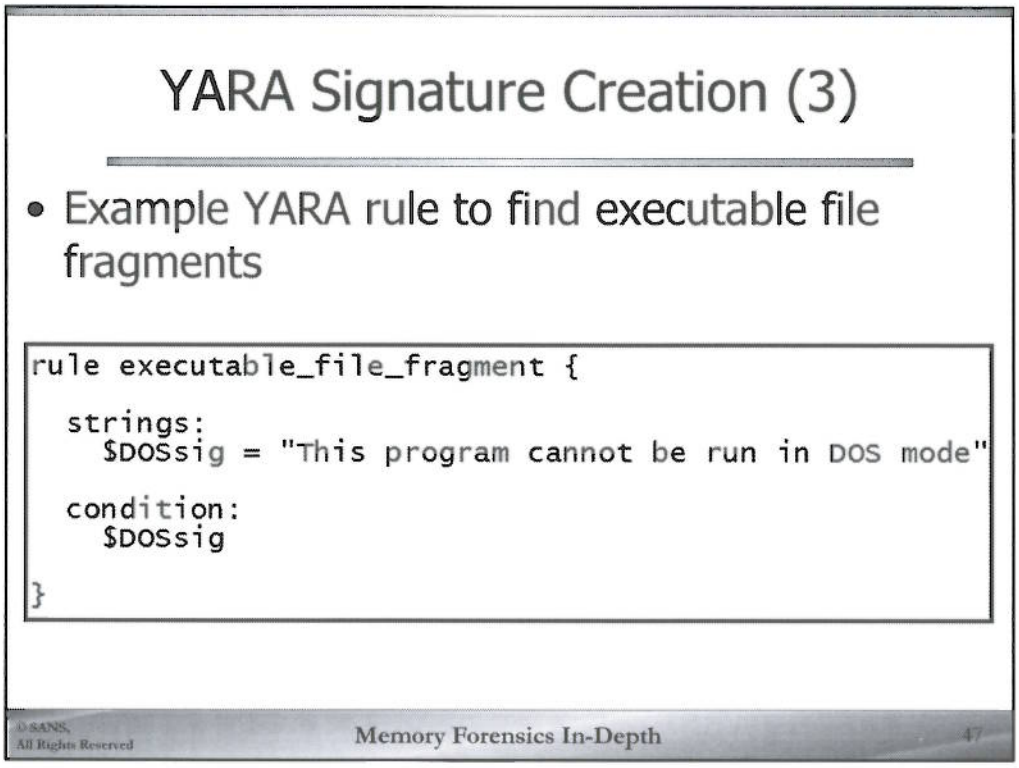


In this hands on exercise, you'll create a signature to find executable file fragments in the page file using the page_brute tool. Because of the way Windows memory management works, executable files on the local filesystem loaded in memory are not usually paged to disk. This means that any executable file fragments found in the page file may have been downloaded, extracted from a zip file, or run from a network share. Any of these cases may be interesting for the forensic examiner.



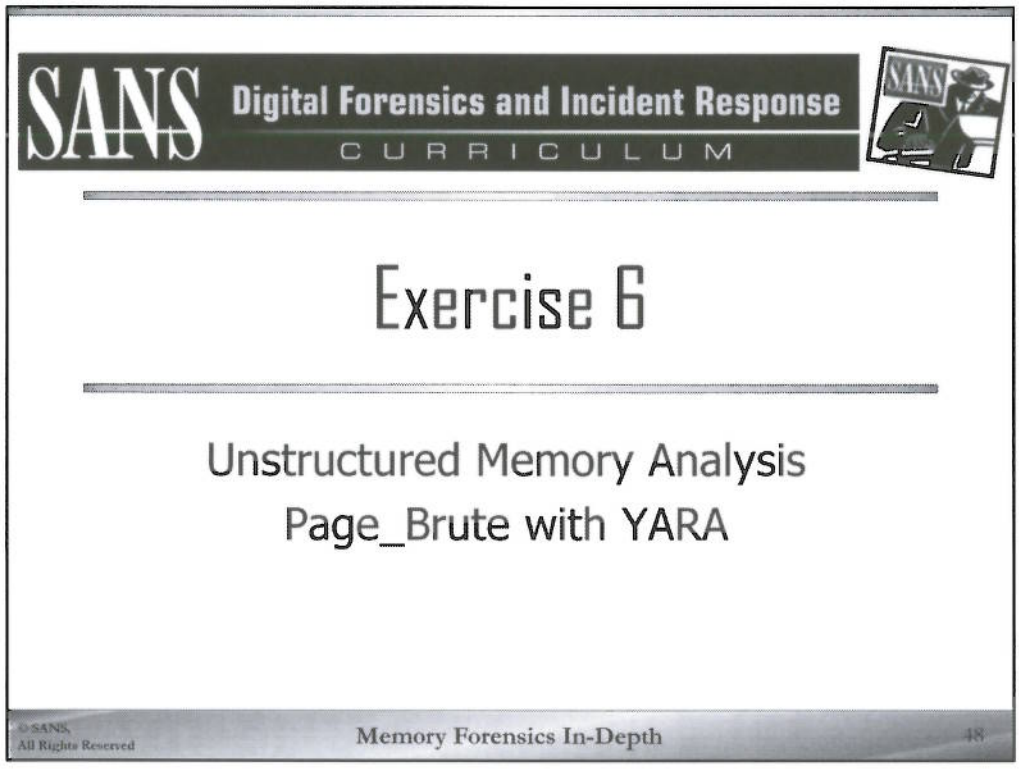
Anyone who has looked at a Windows executable knows that in the beginning of the header, there's an ASCII string that says "This program cannot be run in DOS mode". This string is part of the DOS stub, designed to be run when the Windows program is run under DOS. It's a backward compatibility thing… but it's also a potential signature.

Of course we could add lots of other conditions, but the goal here is to get you up and running quickly with easy YARA rules.

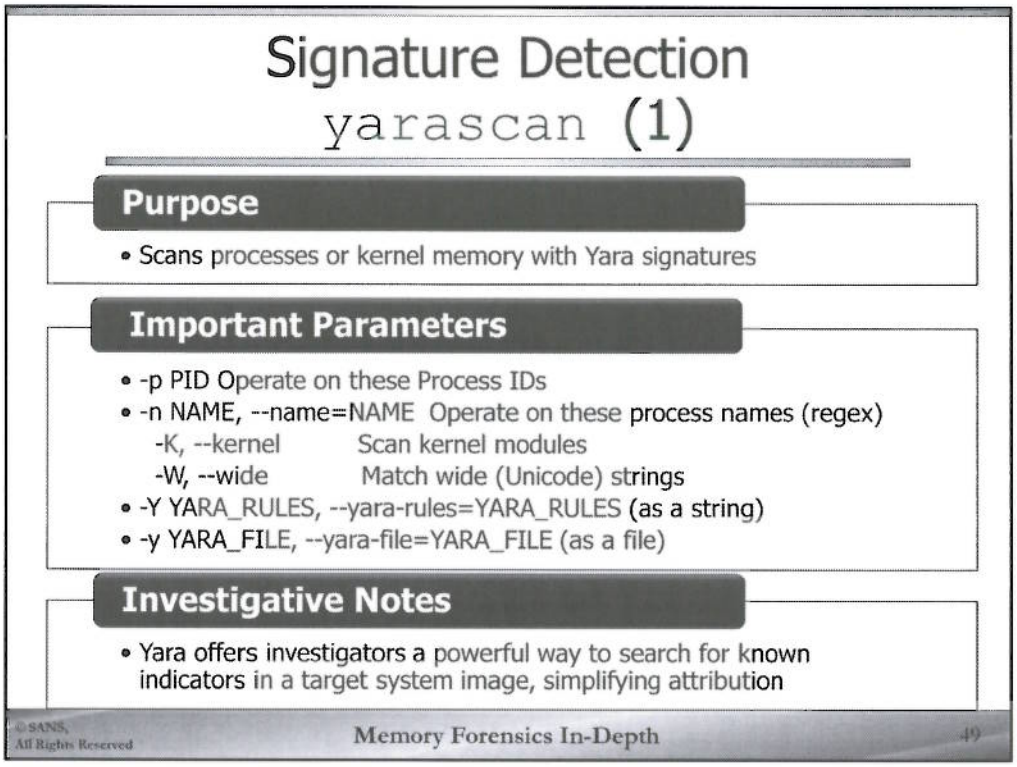


This slide shows the rule created to detect executable file fragments. This rule may have false positives, but again, this is only designed to be a trivial (though useful) example. This rule would be effective at scanning the page file, but would also be useful for scanning PDF or other document files for embedded executables.

```
rule executable_file_fragment {
  strings:
    $DOSsig = "This program cannot be run in DOS mode"
  condition:
    $DOSsig
}
```



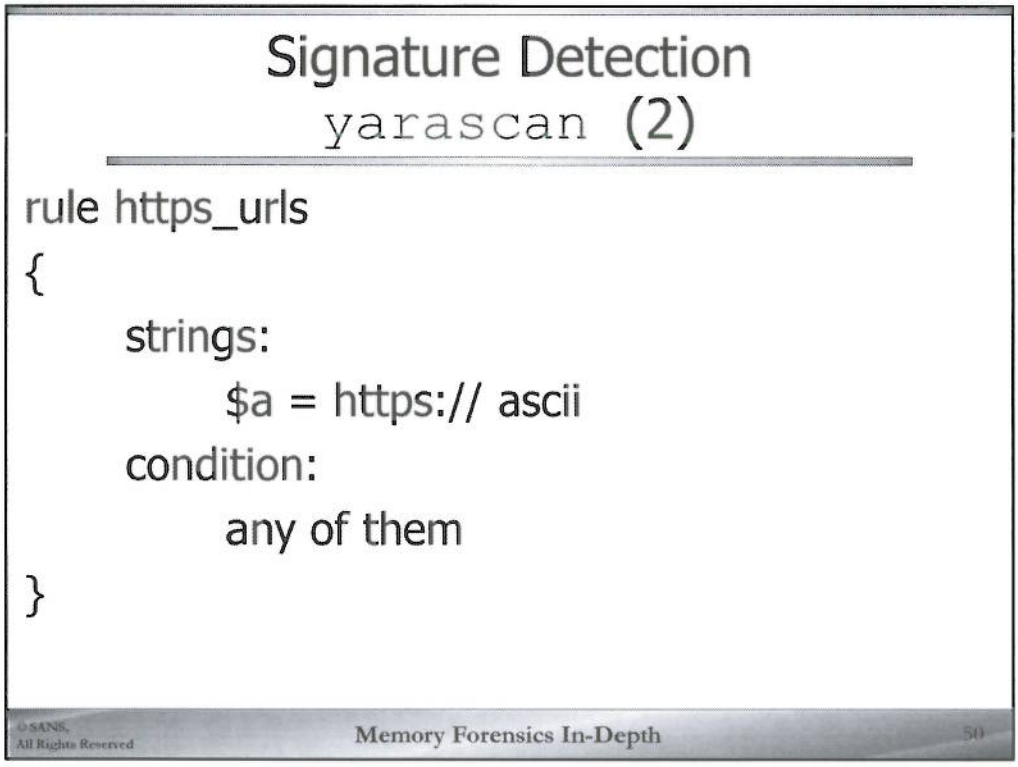
This page intentionally left blank.



The **yarascan** plugin scans memory for known signatures. This is extremely useful in performing attribution. Even if specifics of an attack group are not known (country of origin, motivation, etc.), attributing multiple attacks to a single group through similarities in attack tools can be hugely useful. The **yarascan** plugin simplifies this operation.

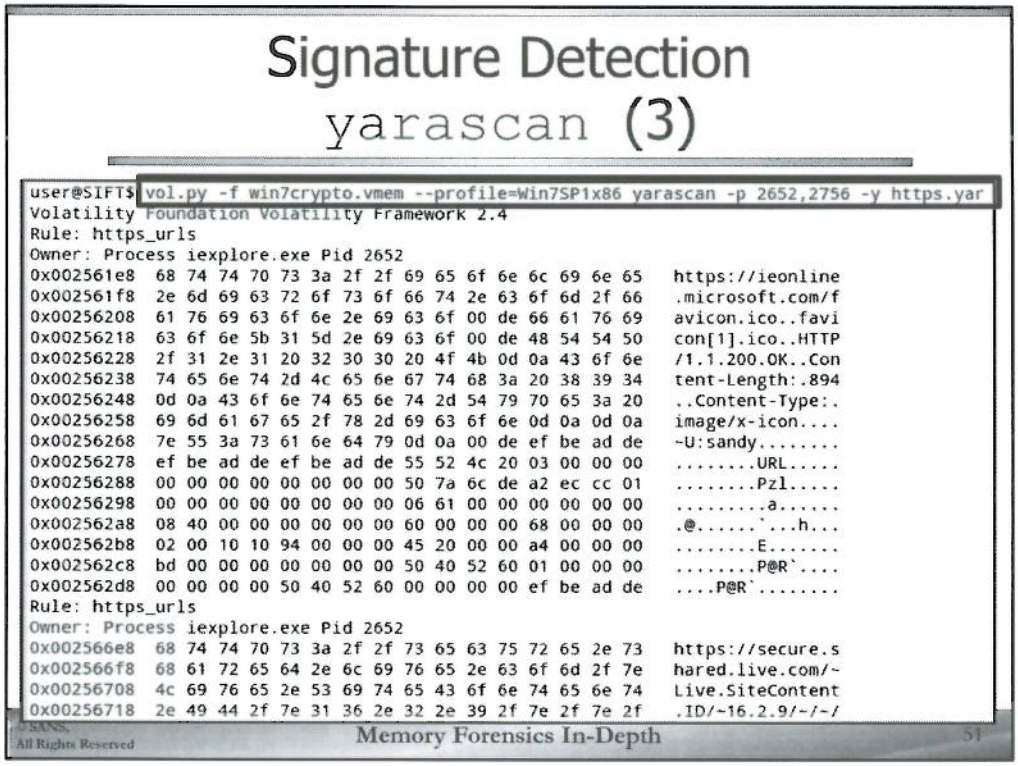
Insider investigations may benefit from searching through memory for keywords that would be of interest in a case. For instance, **yarascan** could be used with a list of engineering terms that should not be present on the janitorial staff's computer to detect possible insider threats.

Note that the **yarascan** plugin can be used to search all of physical memory or can search in a specific process's virtual memory by using the **memdump** plugin to first dump individual memory regions before scanning. The advantage of this method is that any matches found with **yarascan** have some context around them. However, the disadvantage is that only currently allocated memory will be searched. Searching a physical memory image will locate matches for both allocated and unallocated memory.



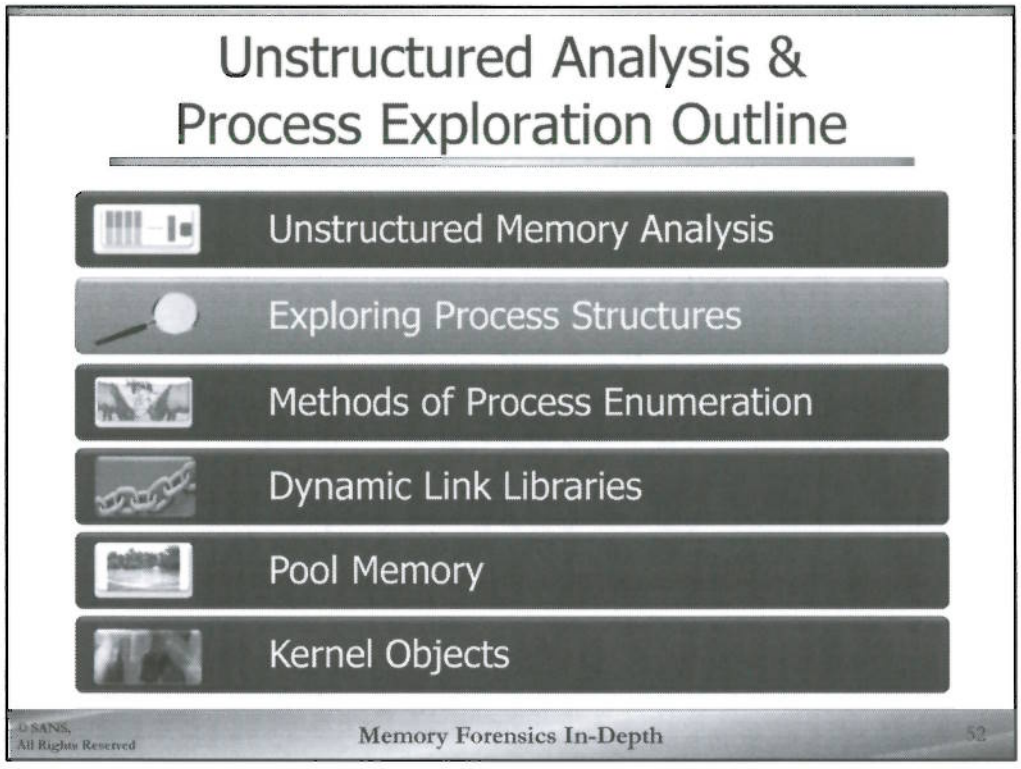
This slide shows a very simple yara rule used to find https URLs in a memory capture. One can imagine how it could be easily extended to locate URLs for particular web services in memory images as well.

```
rule https_urls
{
        strings:
                $a = "https://"  ascii
        condition:
                any of them
}
```

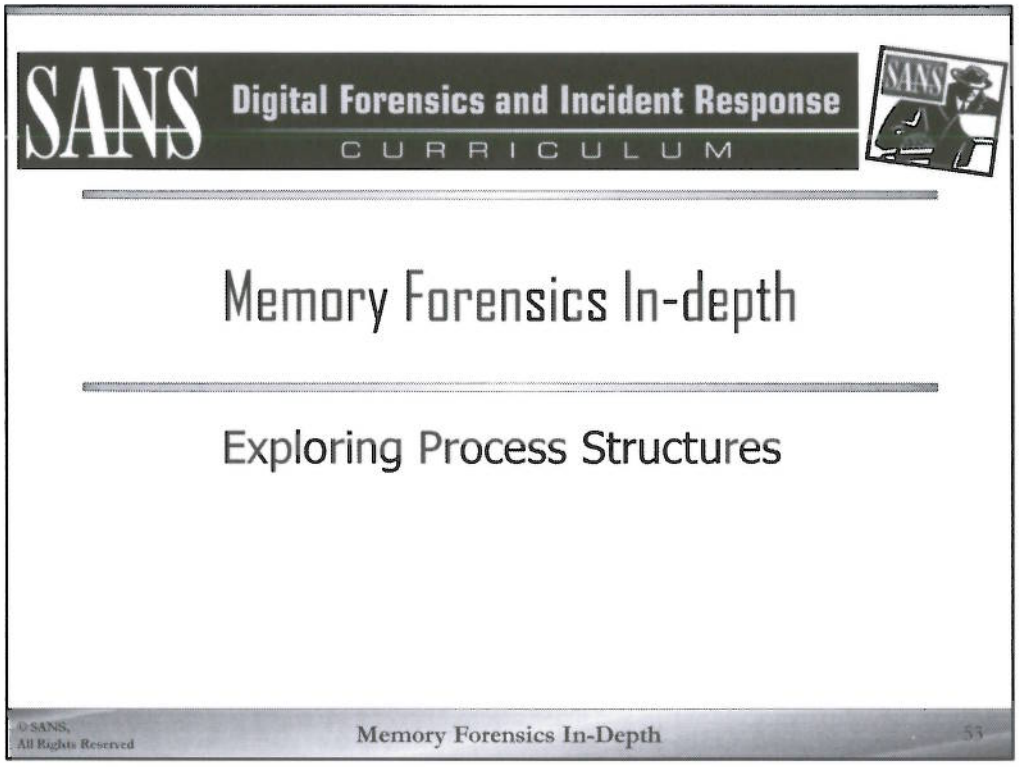


This slide shows example output from a run of the yarascan plugin.

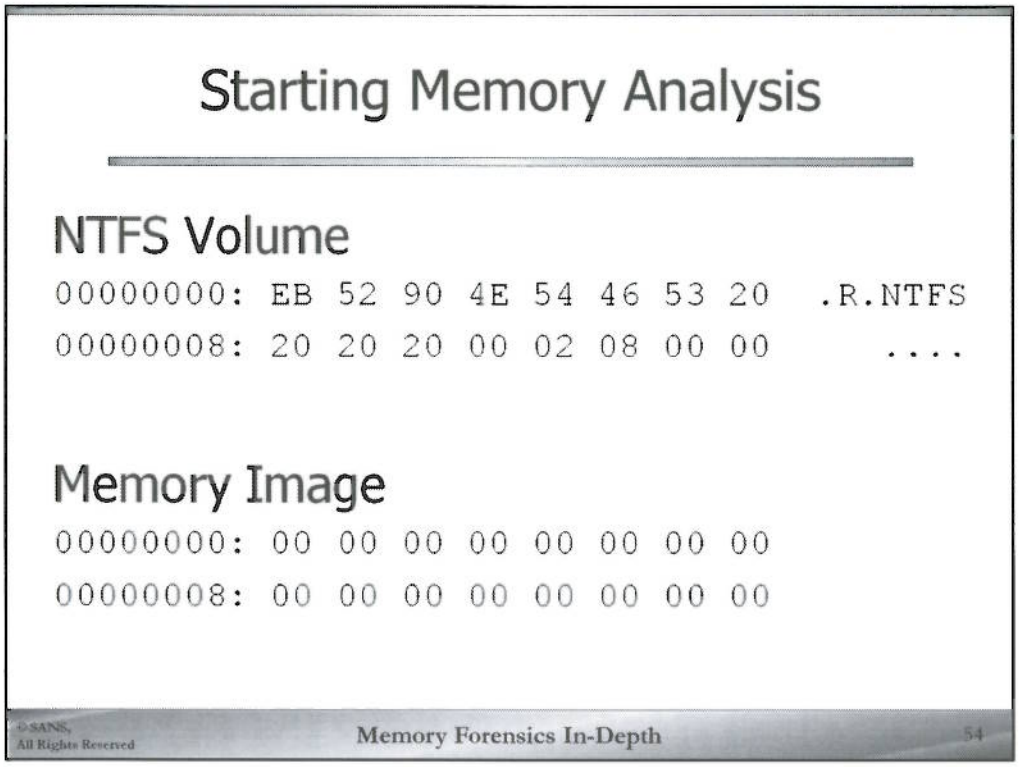
user@SIFT$ vol.py -f win7crypto.vmem --profile=Win7SP1x86 yarascan -p 2652,2756 -y https.yar
Volatility Foundation Volatility Framework 2.4
Rule: https_urls
Owner: Process iexplore.exe Pid 2652
0x002561e8 68 74 74 70 73 3a 2f 2f 69 65 6f 6e 6c 69 6e 65 https://ieonline
0x002561f8 2e 6d 69 63 72 6f 73 6f 66 74 2e 63 6f 6d 2f 66 .microsoft.com/f
… output truncated …



This page intentionally left blank.



This page intentionally left blank.

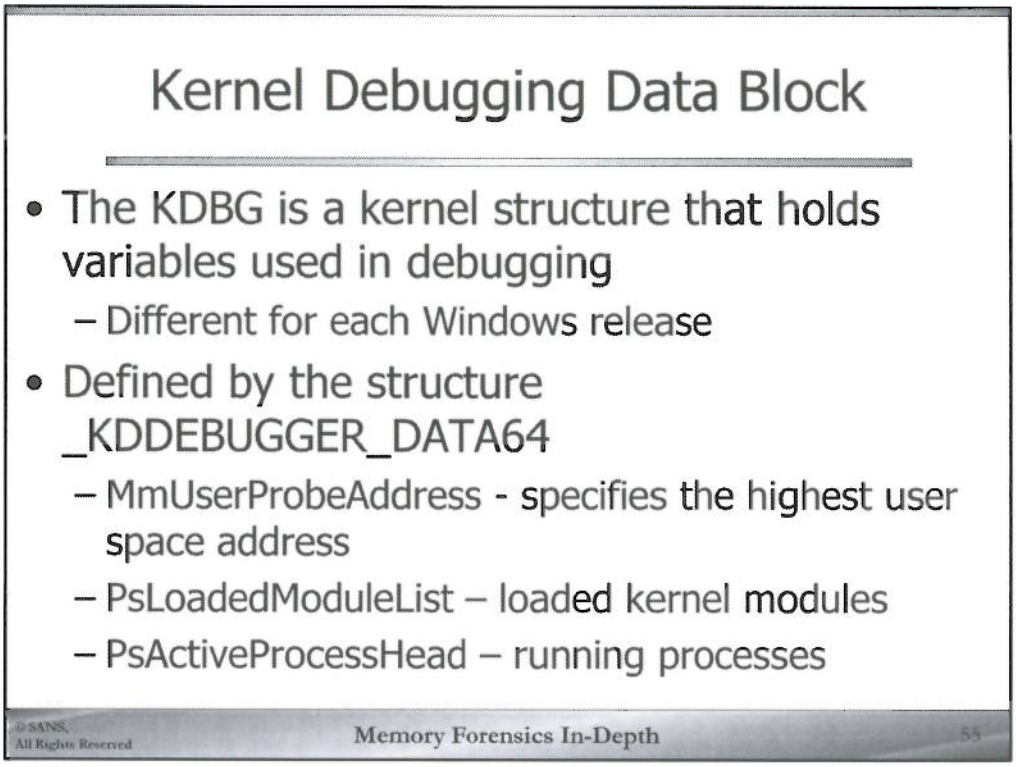


Although we can get data from a memory image by treating it as unstructured data, there is far more information to be had by parsing the operating system structures. Unfortunately memory images are not like disk images and cannot be parsed in the same manner. Every disk image has a defined structure which can be used for parsing it. There's a volume header, which points to a Master File Table or inode structure, or some other easy starting point for analysis. Memory images, however, have no such starting point. They were not intended to be parsed by any program, and thus do not follow any convention regarding their layout. The operating system and processor chose where to store their data when the system is booted. So long as they remain consistent during that boot, their choices of location can be arbitrary.

To get started with memory forensics, we need some kind of starting point. An anchor upon which we can base our analysis. For starters, we don't even know which operating system we're examining. Much of our later memory analysis will depend on our knowing what operating system we're examining. Kernel variables, which lead us to offsets in the memory image, contain valuable pieces of data. To know those offsets, which change between OS releases, service packs, and sometimes even patches, we need to know what OS we're looking at.

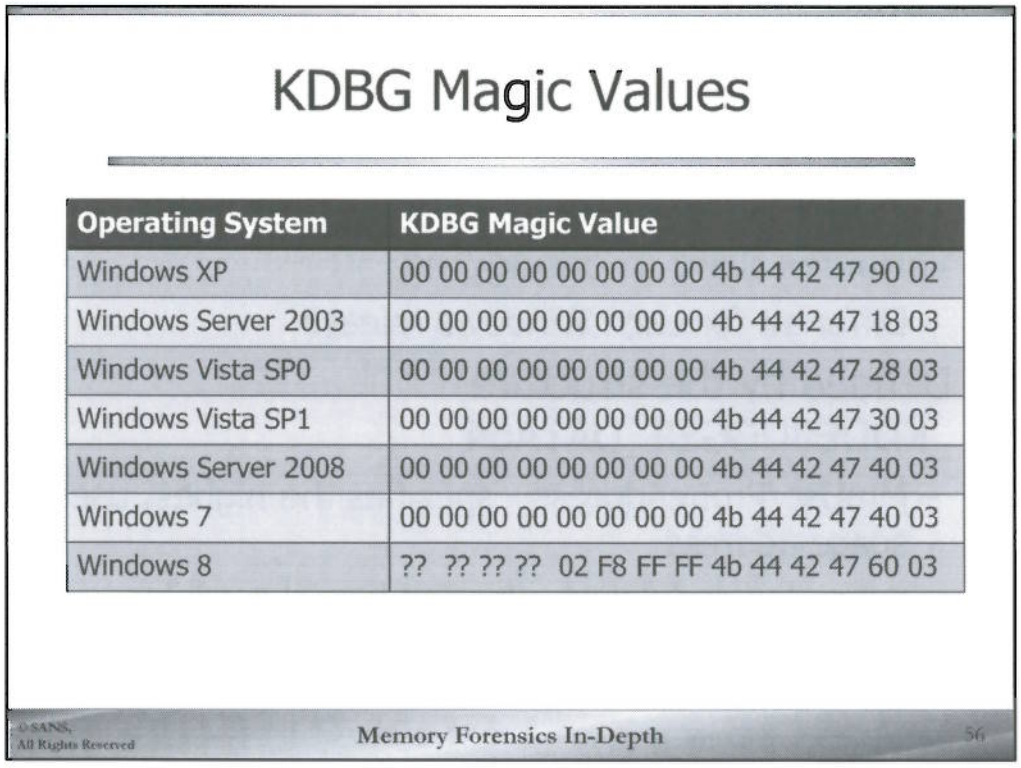
The structure and operation of Windows has changed from version to version. It's important to know if we're looking at a system running 32-bit Windows 2000 or 64-bit Windows Server 2008.

To find our starting point, we need to do a brute force search of the memory image. There are multiple ways to find our anchor, but in this course we're going to discuss only two of them. These methods are searching for individual processes and searching for the Kernel Debugging Data Block (KDBG). Both of these methods will yield information about the operating system and give us the data we need to start our analysis.

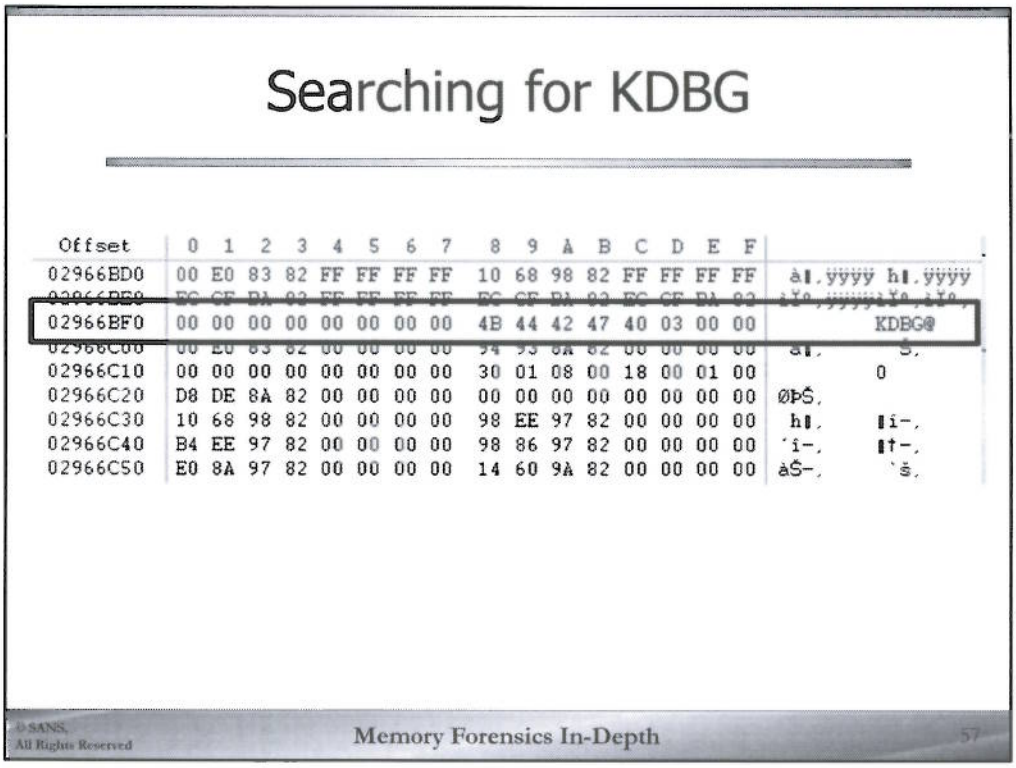


The Kernel Debugging Data Block or KDBG structure is stored in kernel memory and used for debugging purposes. If the system crashes, the data kept in the KDBG is used to create the crash dump file. But the KDBG is also tremendously useful for memory forensics. Not only does the structure have a distinct signature, but the signature is distinct for each operating system. Thus, by searching for the KDBG, not only can we determine which operating system we're examining based on the signature found, but we can get all kinds of useful information about the target operating system.

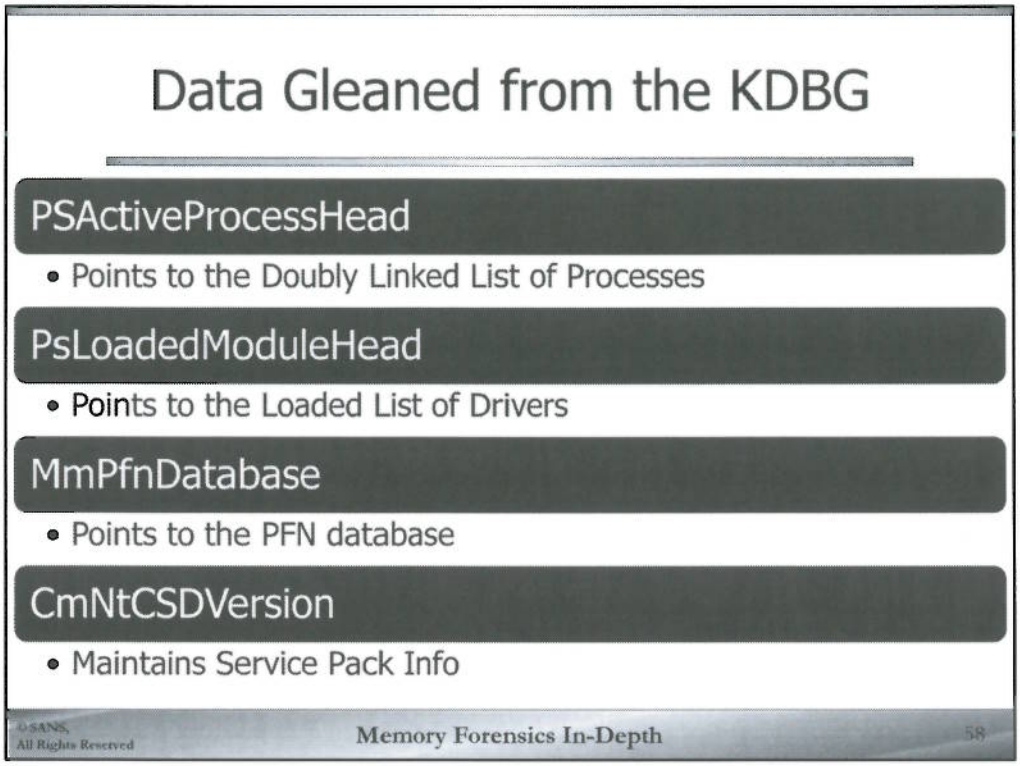
The contents of the KDBG include a large number of kernel variables, or symbols. These are values used by the operating system which detail where important structures are stored. These values are normally defined in the symbols for each operating system. But particular hotfixes or patches from Microsoft can alter those values. For that reason, it's always better to get these variables directly from the operating system. For example, the KDBG contains the address of the list of active processes running on the system, PsActiveProcessHead. Although we should be able to determine this address by looking at the process structures, why bother computing it when we can simply look it up?



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. The KDBG started out at 0x290 bytes 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 KDBG signature, without the eight bytes of zeros.



In this slide you can see the start of a KDBG structure at 0x2966bf0. There are the eight zero bytes, the ASCII bytes for "KDBG" and then the bytes 40 03. Using the table on the previous slide, we can tell this is a potential KDBG block from a Windows 7 or Windows Server 2008 system.



Here is just a small sample of the kernel variables which we can get from the KDBG. We get the following kernel variables, which are the virtual addresses of:

PsActiveProcessHead points to the list of active (running) processes on the system. This lets us see what processes the operating system knew about.

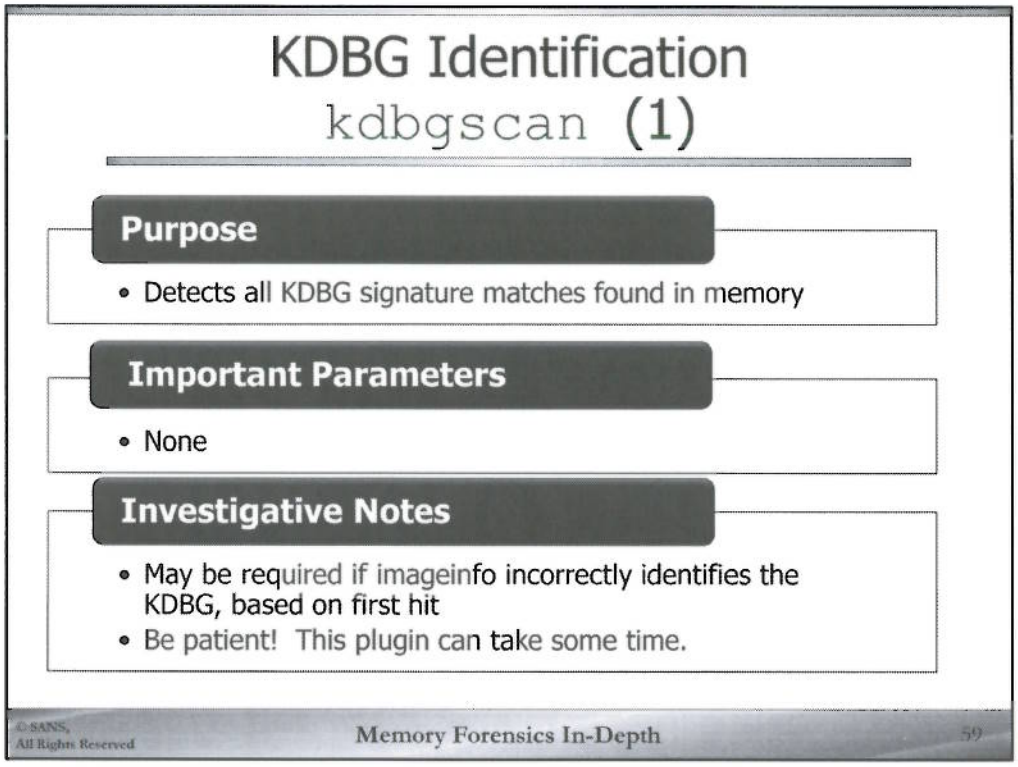
PsLoadedModuleHead points to the list of active (running) drivers on the system. Again, this lets us see what drivers the operating system knew about.

MmPfnDatabase is the Page Frame Number database. It's a record of how every frame of physical memory is being used.

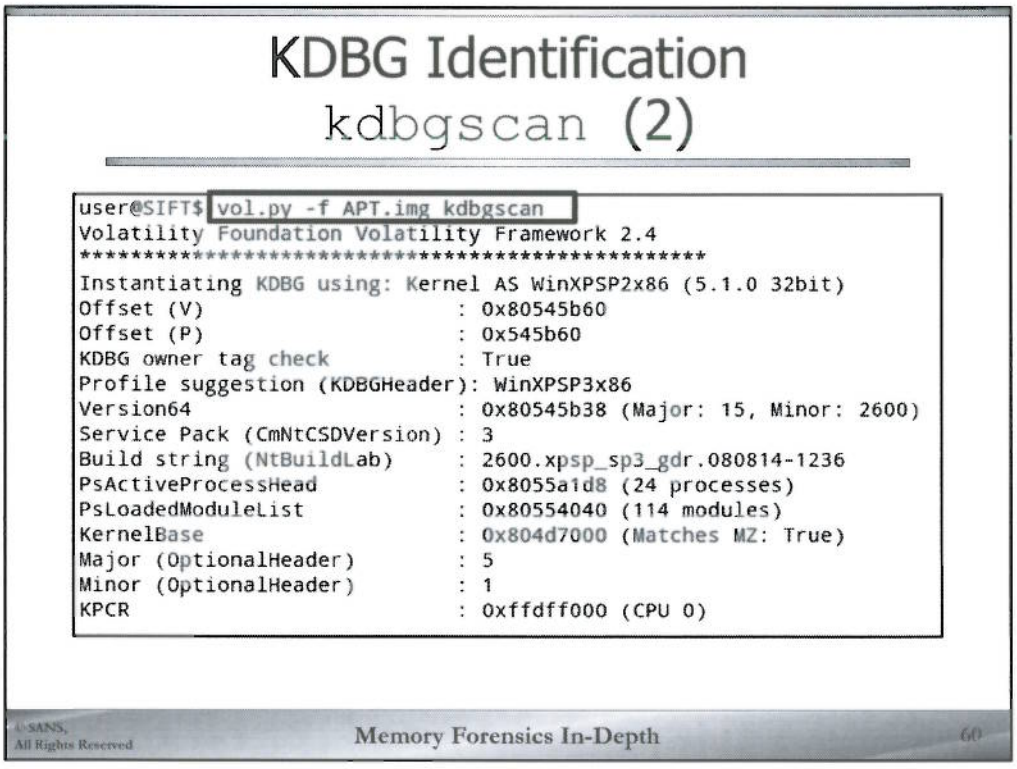
CmNtCSDVersion is a representation of the Service Pack number of the current operating system.

There are also pointers to the list of unloaded drivers, free frames of memory, offsets of values inside important data structures, and more.

Although the magic value is useful for finding KDBG structures, your forensics tool should do some sanity checking on the values it finds in the structure. For example, the virtual addresses listed above should all be in kernel space. That is, greater than 0x8000000.



The **kdbgscan** plugin may be very useful if there are multiple hex values that exist in a Windows memory image that match the KDBG signature. **Imageinfo** scans for a KDBG signature and stops once it locates its first match and parses the structure based on known offset. Imagine if the same pattern existed natively in the wild (8 null bytes followed by KDBG and two size bytes).



The first hands-on exercise we're going to do with Volatility is to do a brute force search for the kernel debugging data block (KDBG) structures we talked about in the last section. Here's the command line we're going to run:

Note that the output gives, on each line, after KDBG, whether this is a virtual (V) or physical (P) address, the address or offset, and the profile for which this is valid. In our case we see really only one offset, 0x54c060, which is either a Windows XP Service Pack 2 or 3 system. The Volatility framework found the magic value for the Windows XP KDBG header, 00 00 00 00 00 00 00 00 4b 44 42 47 90 02, at those addresses.

user@SIFT$ vol.py -f APT.img kdbgscan

Volatility Foundation Volatility Framework 2.4

****

Instantiating KDBG using: Kernel AS WinXPSP2x86 (5.1.0 32bit)

Offset (V) : 0x80545b60

Offset (P) : 0x545b60

KDBG owner tag check : True

Profile suggestion (KDBGHeader): WinXPSP3x86

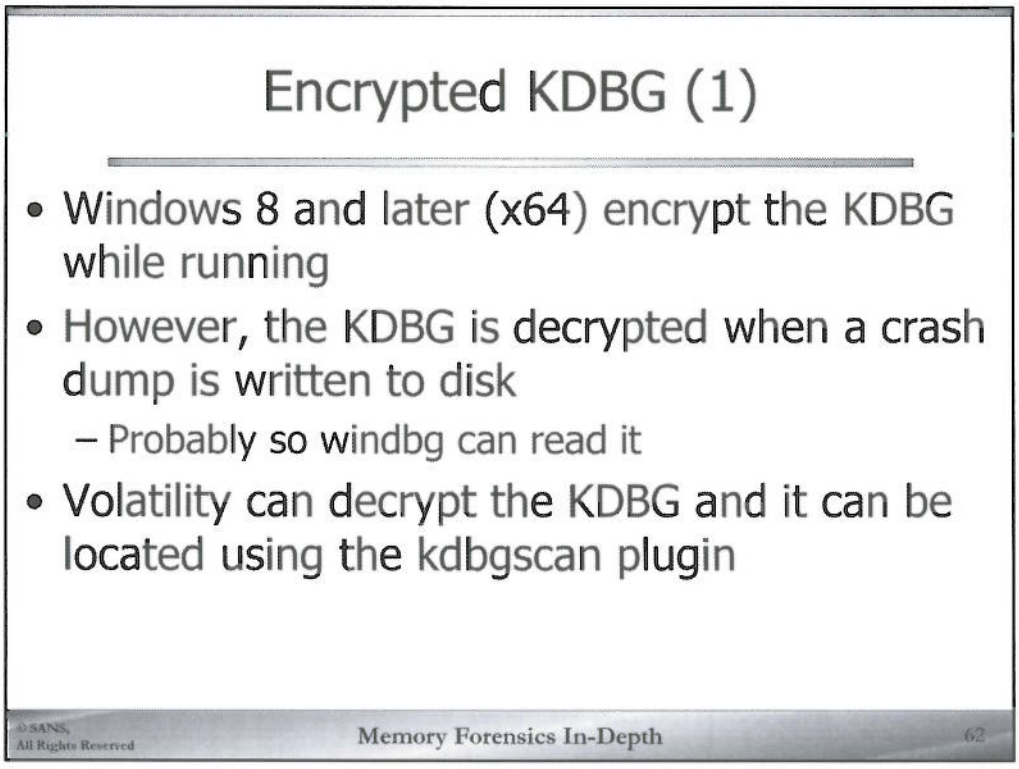
Version64 : 0x80545b38 (Major: 15, Minor: 2600)

Service Pack (CmNtCSDVersion) : 3

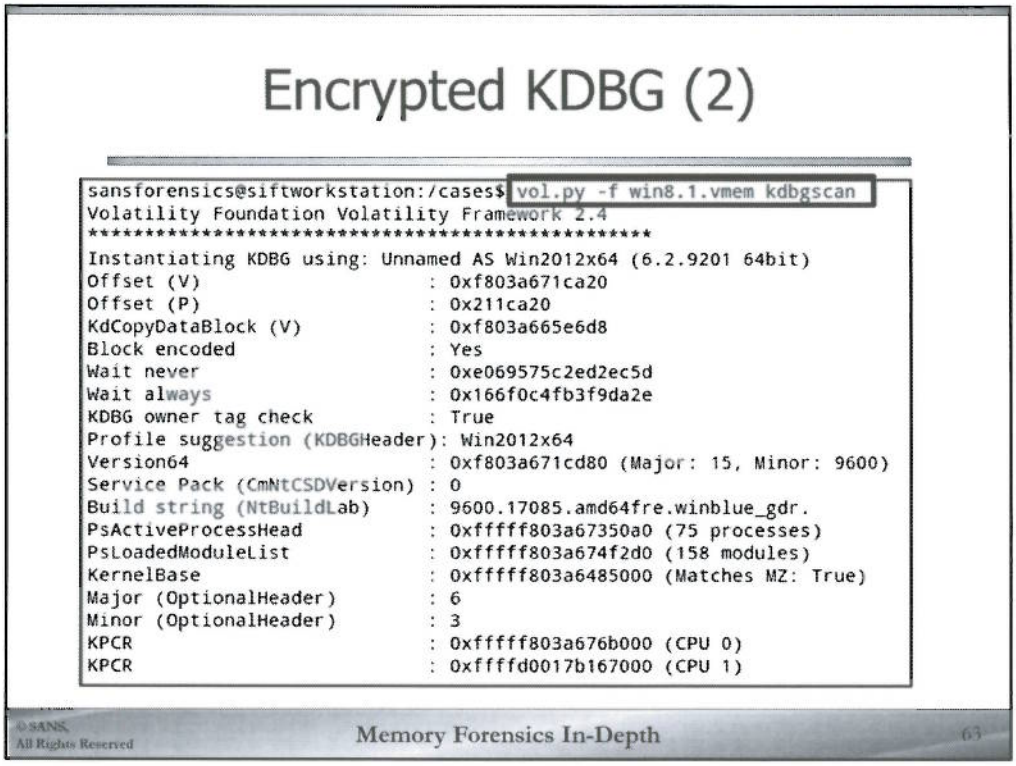
Build string (NtBuildLab) : 2600.xpsp_sp3_gdr.080814-1236

PsActiveProcessHead : 0x8055a1d8 (24 processes)

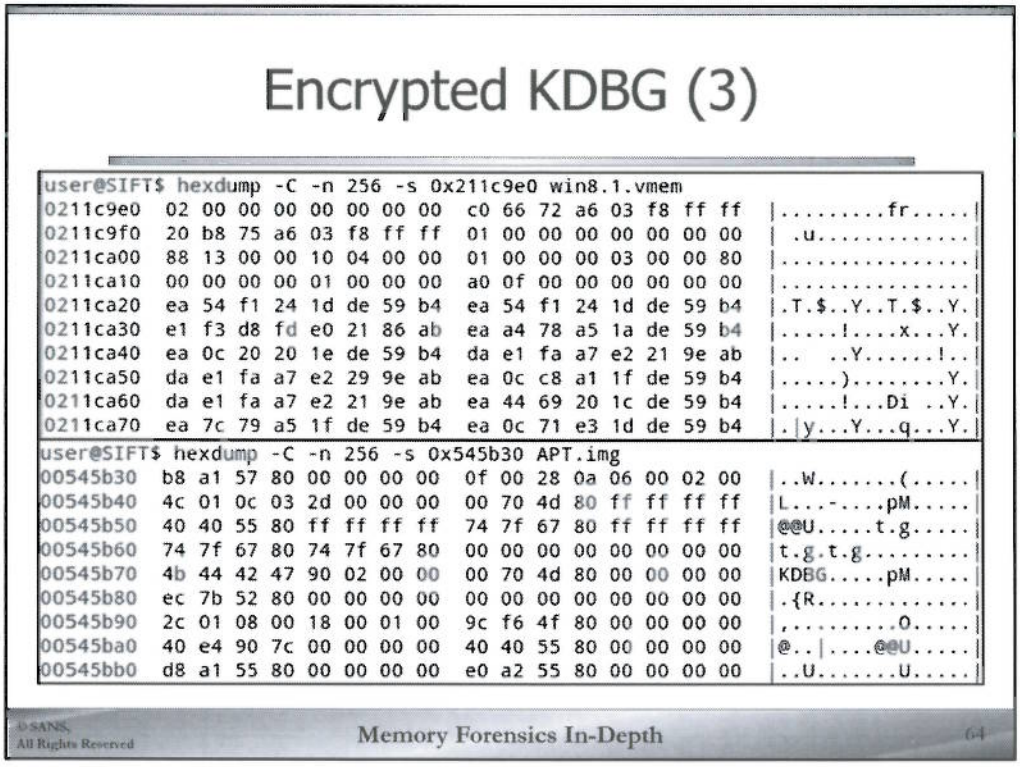
```
PsLoadedModuleList          : 0x80554040 (114 modules)
KernelBase                  : 0x804d7000 (Matches MZ: True)
Major (OptionalHeader)      : 5
Minor (OptionalHeader)      : 1
KPCR                        : 0xffdff000 (CPU 0)

**************************************************
Instantiating KDBG using: Kernel AS WinXPSP2x86 (5.1.0 32bit)
Offset (V)                  : 0x80545b60
Offset (P)                  : 0x545b60
KDBG owner tag check        : True
Profile suggestion (KDBGHeader): WinXPSP2x86
Version64                   : 0x80545b38 (Major: 15, Minor: 2600)
Service Pack (CmNtCSDVersion) : 3
Build string (NtBuildLab)   : 2600.xpsp_sp3_gdr.080814-1236
PsActiveProcessHead         : 0x8055a1d8 (24 processes)
PsLoadedModuleList          : 0x80554040 (114 modules)
KernelBase                  : 0x804d7000 (Matches MZ: True)
Major (OptionalHeader)      : 5
Minor (OptionalHeader)      : 1
KPCR                        : 0xffdff000 (CPU 0)
```



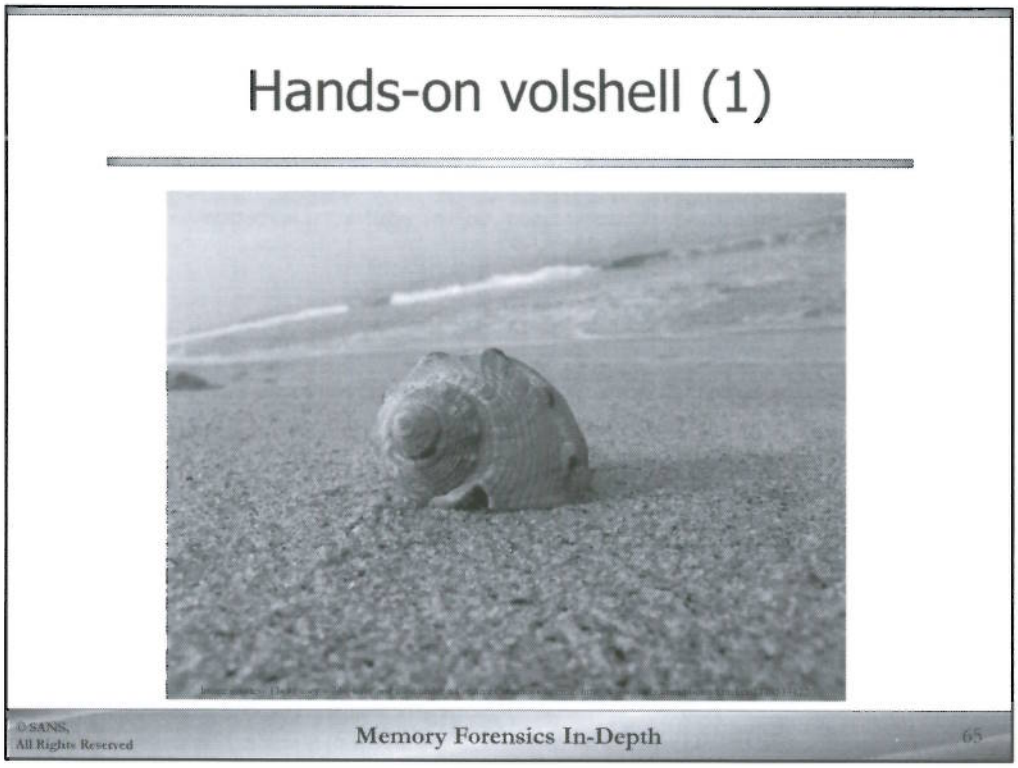
Windows 8 and later (x64) encrypt the KDBG while running. However, the KDBG is decrypted when a crash dump is written to disk. This is probably so windbg can read it. Volatility can decrypt the KDBG and it can be located using the kdbgscan plugin.



Note that volatility can located the KDBG, even though it is encrypted. This can be verified by using volshell as seen on the next slide.

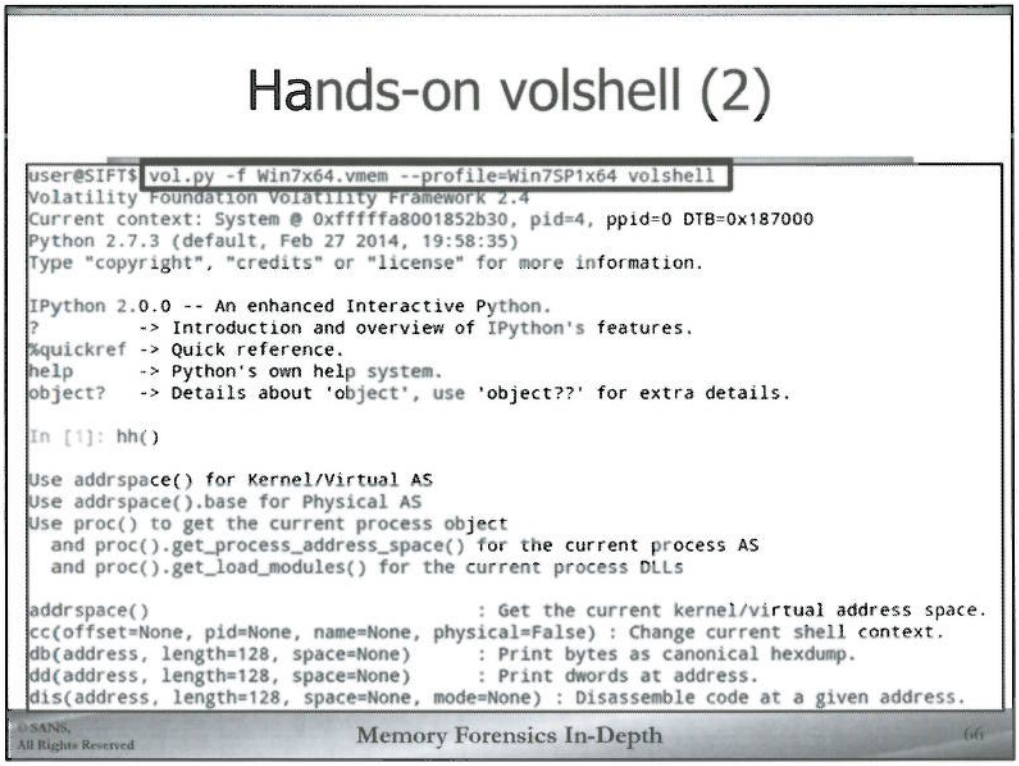


Notice that dumping the location of the KDBG specified in the x86 XP image shows the typical KDBG signature. However, the Windows 8.1 x64 image shows that the KDBG signature is encrypted.



We can see all of the values in the EPROCESS structures, and, in fact, any structure, using a special plugin in Volatility called volshell. This plugin opens a Volatility overlay on top of a standard Python shell. You can then do anything you can in regular Python, but with some extra functionality for memory forensics. We are now going to look at volshell, using the knowledge of Windows structures we talked about earlier. We can't use the Windows debugging tools on memory images, but we can use Volatility to explore them.

Volshell was intended to help researchers explore Windows memory structures. It is an interactive shell and somewhat mimics the commands of kd and WinDBG. Not all of the commands are implemented, but a few of them are. The result is a powerful tool for learning about your memory images.



Starting the volshell plugin is similar to starting other plugins. We specify a memory image and profile along with the volshell command.

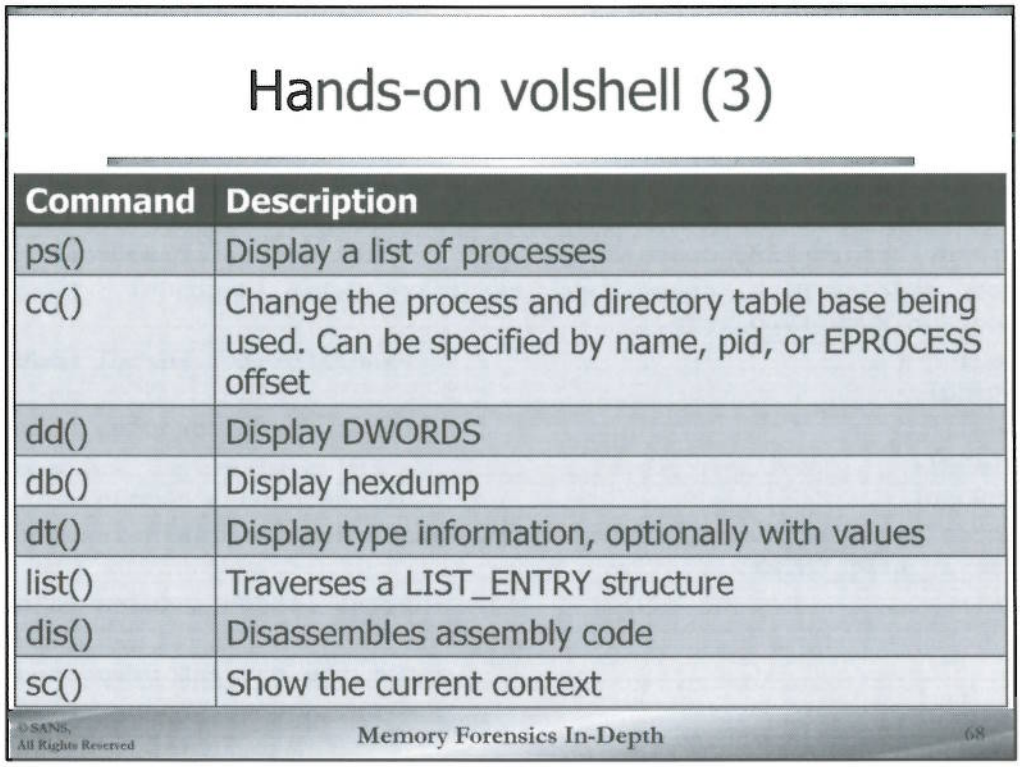
```
user@SIFT$ vol.py -f Win7x64.vmem --profile=Win7SP1x64 volshell
Volatility Foundation Volatility Framework 2.4
Current context: System @ 0xfffffa8001852b30, pid=4, ppid=0 DTB=0x187000
Python 2.7.3 (default, Feb 27 2014, 19:58:35)
Type "copyright", "credits" or "license" for more information.

IPython 2.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: hh()

Use addrspace() for Kernel/Virtual AS
Use addrspace().base for Physical AS
Use proc() to get the current process object
  and proc().get_process_address_space() for the current process AS
  and proc().get_load_modules() for the current process DLLs

addrspace()                              : Get the current kernel/virtual
address space.
```

```
cc(offset=None, pid=None, name=None, physical=False) : Change current
shell context.
db(address, length=128, space=None)      : Print bytes as canonical
hexdump.
dd(address, length=128, space=None)      : Print dwords at address.
dis(address, length=128, space=None, mode=None) : Disassemble code at a
given address.
dq(address, length=128, space=None)      : Print qwords at address.
dt(objct, address=None, space=None, recursive=False, depth=0) : Describe
an object or show type info.
getmods()                                : Generator for kernel modules
(scripting).
getprocs()                               : Generator of process objects
(scripting).
hh(cmd=None)                             : Get help on a command.
list_entry(head, objname, offset=-1, fieldname=None, forward=True) :
Traverse a _LIST_ENTRY.
modules()                                : Print loaded modules in a table
view.
proc()                                   : Get the current process object.
ps()                                     : Print active processes in a
table view.
sc()                                     : Show the current context.

For help on a specific command, type 'hh(<command>)'
```

There is a lot going on here! First, volshell has displayed some details about the current process which it is examining. We can examine any process using volshell, but this is the one it's started with. In this run it's started with the System process, pid 4, which has a parent pid (ppid) of zero. The directory table base for this process is 0x39000.

Volshell is also reminding us that the command hh() will display a help message. We'll get to that on the next page.

The three greater than signs at the bottom of the output are a Python prompt. We can now type any valid Python commands and the interpreter will execute them (e.g. 2+3, len([1,2,3]), [X ** 2 for X in [1,2,3]], etc.)

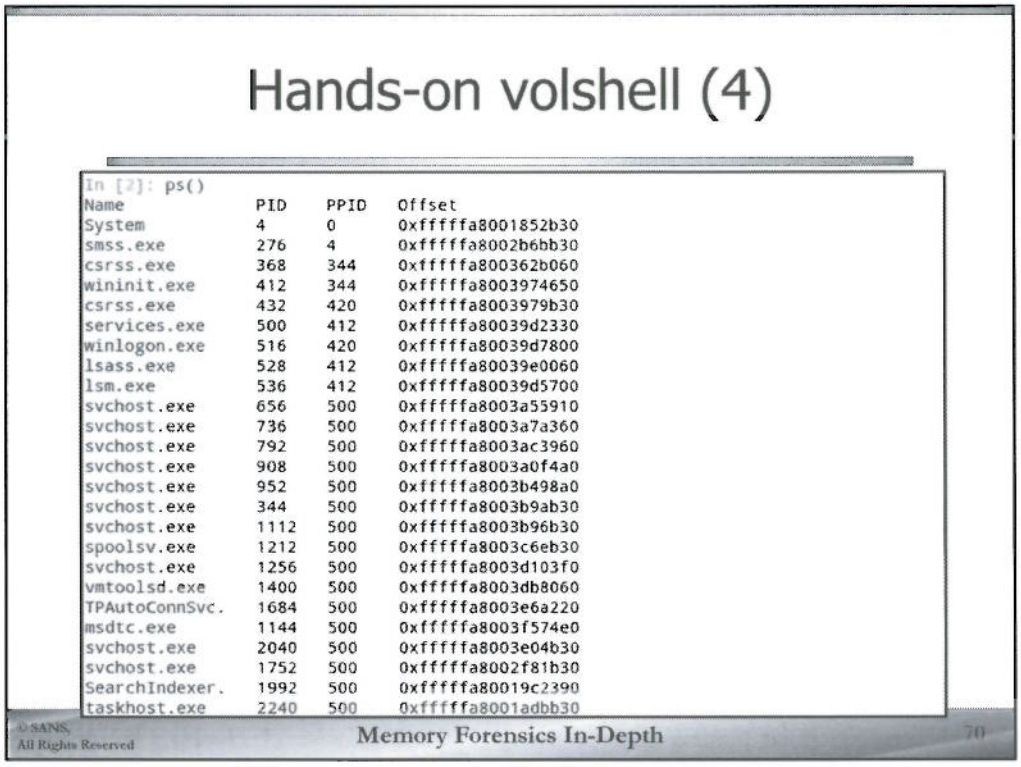


Let's start with the help command in volshell to see what we can do:

```
>>> hh()
Use self.addrspace for Kernel/Virtual AS
Use self.addrspace.base for Physical AS
Use self.proc to get the current _EPROCESS object
  and self.proc.get_process_address_space() for the current process AS
  and self.proc.get_load_modules() for the current process DLLs

cc(offset=None, pid=None, name=None)      : Change current shell context.
db(address, length=128, space=None)       : Print bytes as canonical
hexdump.
dd(address, length=128, space=None)       : Print dwords at address.
dis(address, length=128, space=None, mode=None) : Disassemble code at a
given address.
dq(address, length=128, space=None)       : Print qwords at address.
dt(objct, address=None, space=None)       : Describe an object or show type
info.
hh(cmd=None)                              : Get help on a command.
list_entry(head, objname, offset=-1, fieldname=None, forward=True) :
Traverse a _LIST_ENTRY.
modules()                                 : Print a module listing.
ps()                                      : Print a process listing.
sc()                                      : Show the current context.
```

```
For help on a specific command, type 'hh(<command>)'
```

Each command can take some required and optional arguments. The defaults for these arguments are shown in the help message. You can change the defaults by overriding them on the command line. This is easier to show than explain, and we'll start on the next page.

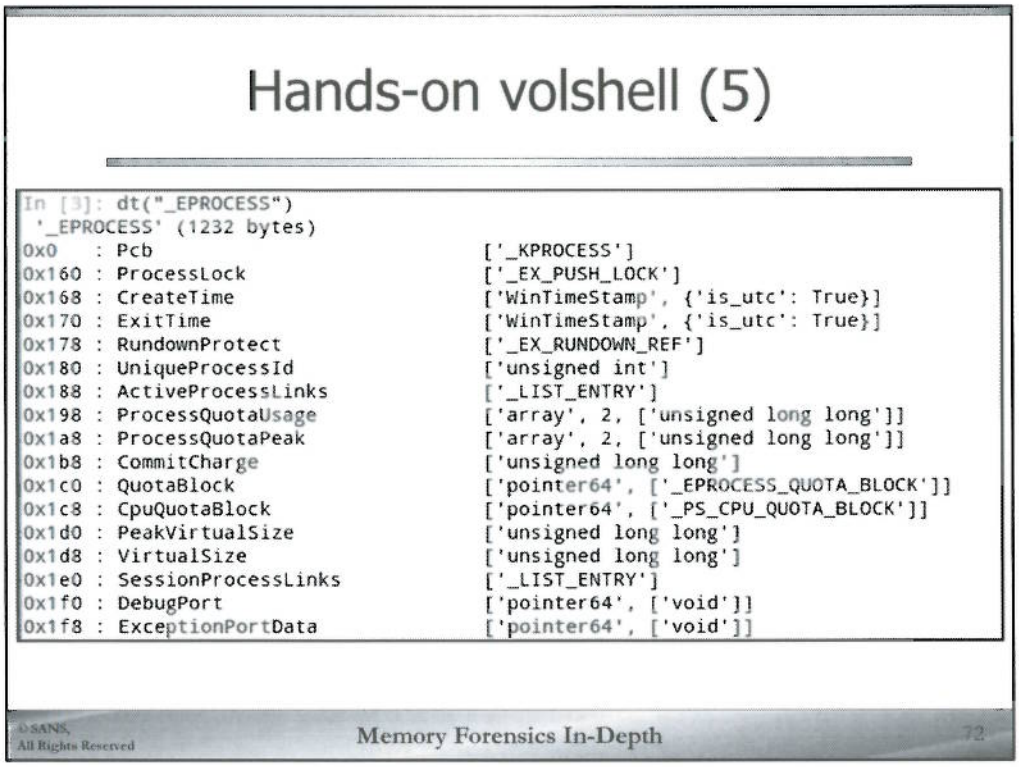


Let's start with the ps() command. This command takes no arguments, but displays a list of processes.

```
In [2]: ps()
Name                 PID    PPID   Offset
System               4      0      0xfffffa8001852b30
smss.exe             276    4      0xfffffa8002b6bb30
csrss.exe            368    344    0xfffffa800362b060
wininit.exe          412    344    0xfffffa8003974650
csrss.exe            432    420    0xfffffa8003979b30
services.exe         500    412    0xfffffa80039d2330
winlogon.exe         516    420    0xfffffa80039d7800
lsass.exe            528    412    0xfffffa80039e0060
lsm.exe              536    412    0xfffffa80039d5700
svchost.exe          656    500    0xfffffa8003a55910
svchost.exe          736    500    0xfffffa8003a7a360
svchost.exe          792    500    0xfffffa8003ac3960
svchost.exe          908    500    0xfffffa8003a0f4a0
svchost.exe          952    500    0xfffffa8003b498a0
svchost.exe          344    500    0xfffffa8003b9ab30
svchost.exe          1112   500    0xfffffa8003b96b30
```

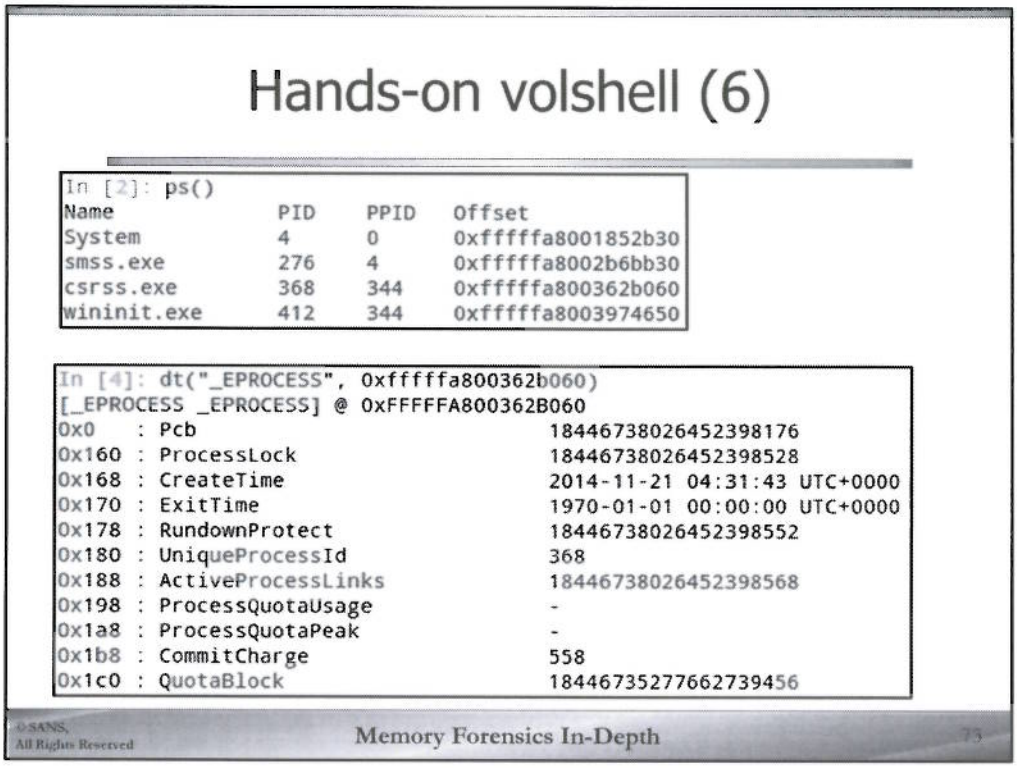
```
spoolsv.exe      1212   500    0xfffffa8003c6eb30
svchost.exe      1256   500    0xfffffa8003d103f0
vmtoolsd.exe     1400   500    0xfffffa8003db8060
TPAutoConnSvc.   1684   500    0xfffffa8003e6a220
msdtc.exe        1144   500    0xfffffa8003f574e0
svchost.exe      2040   500    0xfffffa8003e04b30
svchost.exe      1752   500    0xfffffa8002f81b30
SearchIndexer.   1992   500    0xfffffa80019c2390
taskhost.exe     2240   500    0xfffffa8001adbb30
TPAutoConnect.   2324   1684   0xfffffa8001a0cb30
dwm.exe          2332   908    0xfffffa8001a0d910
conhost.exe      2344   432    0xfffffa8001a64320
explorer.exe     2364   2308   0xfffffa8001a17920
vmtoolsd.exe     2508   2364   0xfffffa8001b91b30
cmd.exe          1408   2364   0xfffffa8001b146b0
conhost.exe      904    432    0xfffffa80039a0970
idaq.exe         1452   2364   0xfffffa8001de6b30
firefox.exe      3240   2880   0xfffffa8003fd1b30
chrome.exe       1040   1732   0xfffffa8001ea9670
chrome.exe       3640   1040   0xfffffa800397cb30
chrome.exe       1032   1040   0xfffffa8003c37060
notepad.exe      2496   2364   0xfffffa8001f04560
taskhost.exe     2784   500    0xfffffa800369b060
taskhost.exe     624    500    0xfffffa8004028920
audiodg.exe      2116   792    0xfffffa80036eb470
cmd.exe          1100   1400   0xfffffa8002c58060
conhost.exe      2608   368    0xfffffa8002a77b30
ipconfig.exe     3200   1100   0xfffffa80021b0760
```

Here we can see the name, pid, parent pid, and EPROCESS offset for every process on the system.



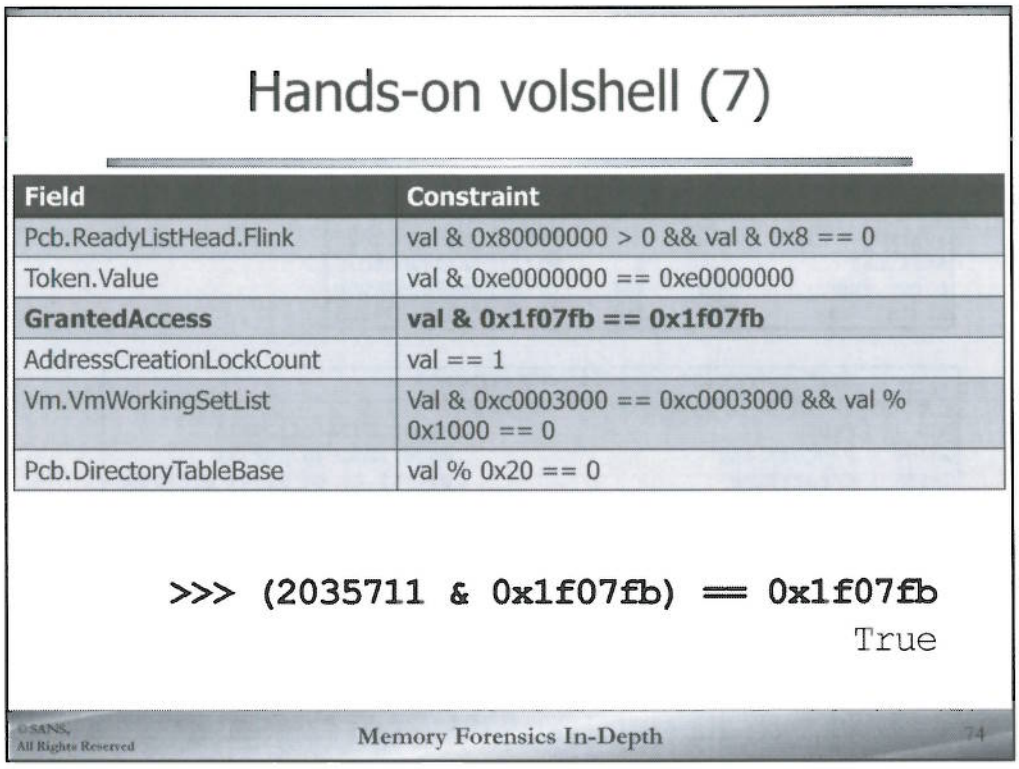
Next, let's explore a structure. The dt() command can be used to display a process structure much like it does in the Windows kernel debuggers, kd and WinDBG. Let's look at the EPROCESS structure. Because of a quirk in how Volatility is built, structure names are preceded by an underscore and are case sensitive! Thus, to see the EPROCESS structure, we call:

```
>>> dt("_EPROCESS")
'_EPROCESS' (608 bytes)
0x0    : Pcb                          ['_KPROCESS']
0x160  : ProcessLock                  ['_EX_PUSH_LOCK']
0x168  : CreateTime                   ['WinTimeStamp', {'is_utc': True}]
0x170  : ExitTime                     ['WinTimeStamp', {'is_utc': True}]
0x178  : RundownProtect               ['_EX_RUNDOWN_REF']
0x180  : UniqueProcessId              ['unsigned int']
0x188  : ActiveProcessLinks           ['_LIST_ENTRY']
… output truncated …
```



Finally we get to the real power of Volshell. We can populate this listing with the values from data on the system. We add an optional parameter to the dt command, an address to use as the base of the structure, and now get the structure populated by the data from that offset. Let's use the offset of the EPROCESS block from the dd.exe process. You can get this offset by using the ps() command.

```
In [3]: ps()
Name         PID   PPID  Offset
System       4     0     0xfffffa8001852b30
smss.exe     276   4     0xfffffa8002b6bb30
csrss.exe    368   344   0xfffffa800362b060
... output truncated ...

In [4]: dt("_EPROCESS", 0xfffffa800362b060)
[_EPROCESS _EPROCESS] @ 0xFFFFFA800362B060
0x0   : Pcb                          18446738026452398176
0x160 : ProcessLock                  18446738026452398528
0x168 : CreateTime                   2014-11-21 04:31:43 UTC+0000
0x170 : ExitTime                     1970-01-01 00:00:00 UTC+0000
... output truncated ...
```



Let's put this work with the volshell plugin into practice. Above you should see the constraints that moyix, Brendan Dolan Gavitt, found for Windows XP Service Pack 3 processes. In particular, let's focus on the constraint for the GrantedAccess field in the EPROCESS structure.

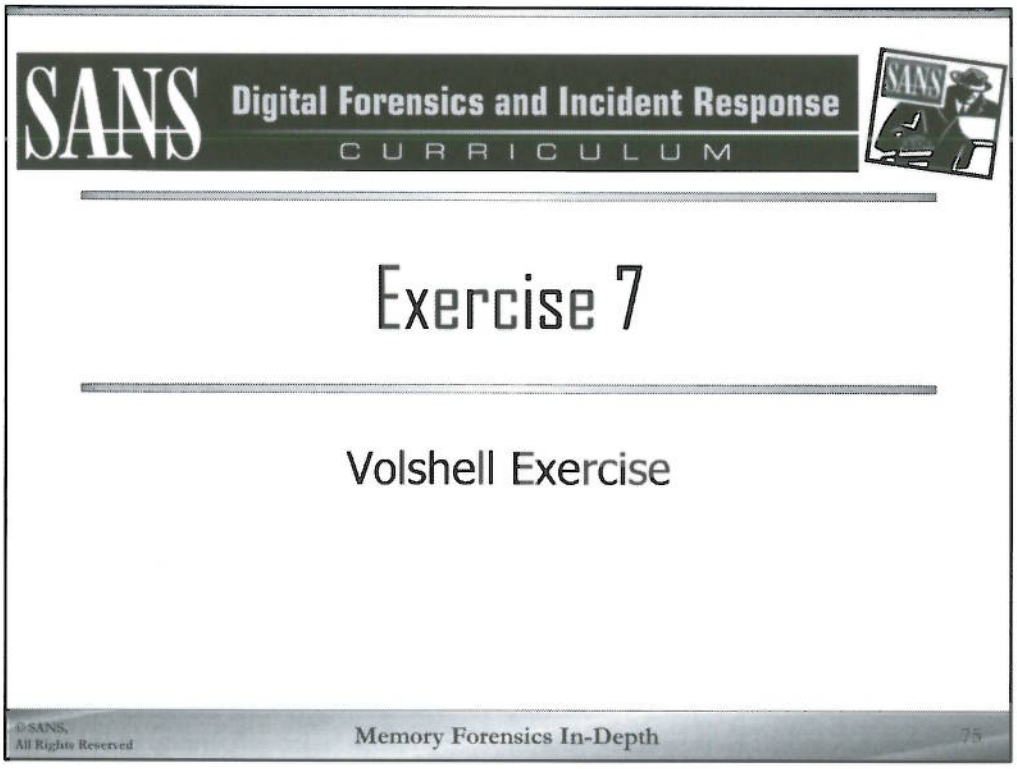
Looking back at our previous output, we can see that the GrantedAccess field for the dd.exe process was 2035711. That number is in decimal, while the value specified above, 0x1f07fb, is in hexadecimal. We can use the power of the Python interpreter to help us determine if these values are equal. We could convert one number to the other:

```
>>> hex(2035711)
'0x1f0fff'
```

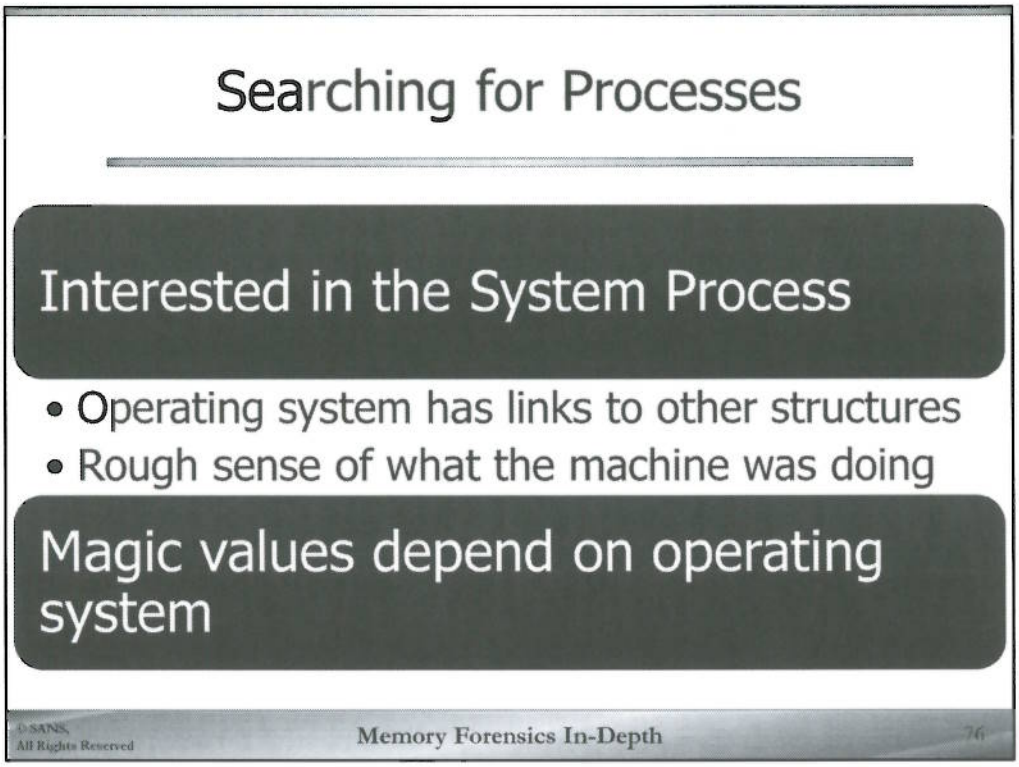
Or we could just ask the interpreter to do the comparison for us. Note that we're using a double equals sign. A single equals sign in Python means assignment.

```
>>> (2035711 & 0x1f07fb) == 0x1f07fb
True
```

The values are equal! This means that the block in question could be a valid EPROCESS block for a Windows XP Service Pack 3 system. (Yes, this memory image is from a Service Pack 2 system.) This is research in action!



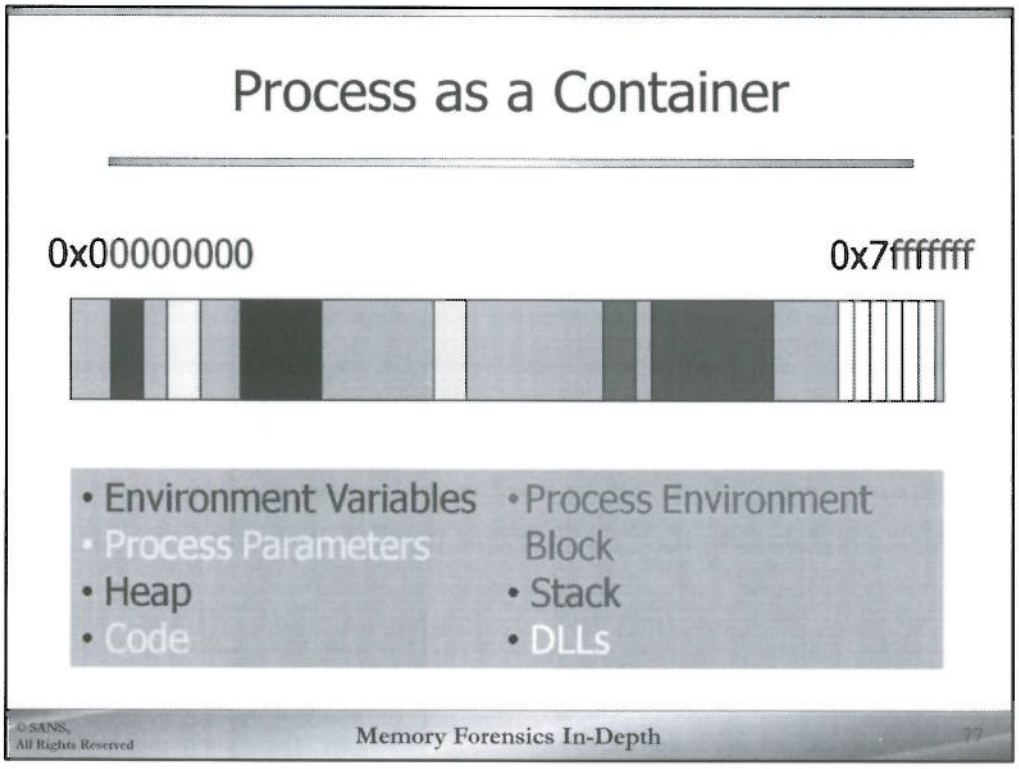
This page intentionally left blank.



Searching for processes allows an examiner to quickly get a sense of what was running on the computer system. Depending on the nature of the investigation, this could be the most important feature. Was the machine running the malware in question? Is there a full-disk encryption program which needs to be examined? A list of the running processes can start to answer those questions.

Because the process structures changed between each version of the operating system, searching for processes can help the examiner determine what operating system was running on the system. Every legitimate process begins with some magic values which can be used to recognize them. These magic values are generally the Pool Header, which will talk about in the next section, or the Dispatcher Header. The former is used by the operating system to efficiently reserve small chunks of memory. In this case, the process information we're looking at is the Executive Process, or EPROCESS structure. The Dispatcher Header is the first item in the EPROCESS structure*.

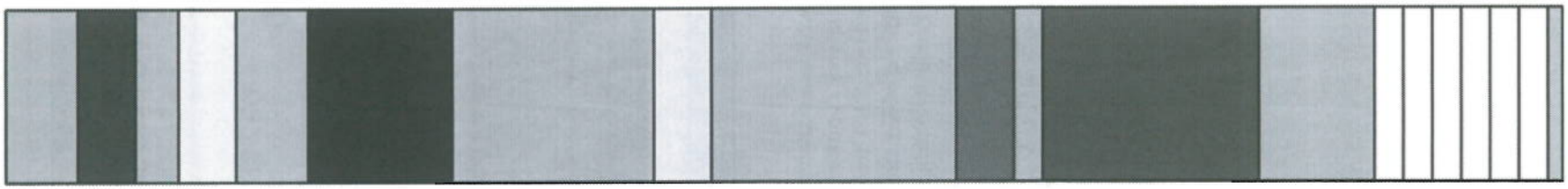
* If you want to be really technical, the first item in the EPROCESS structure is a KPROCESS, and the first item in the KPROCESS is the Dispatcher Header.



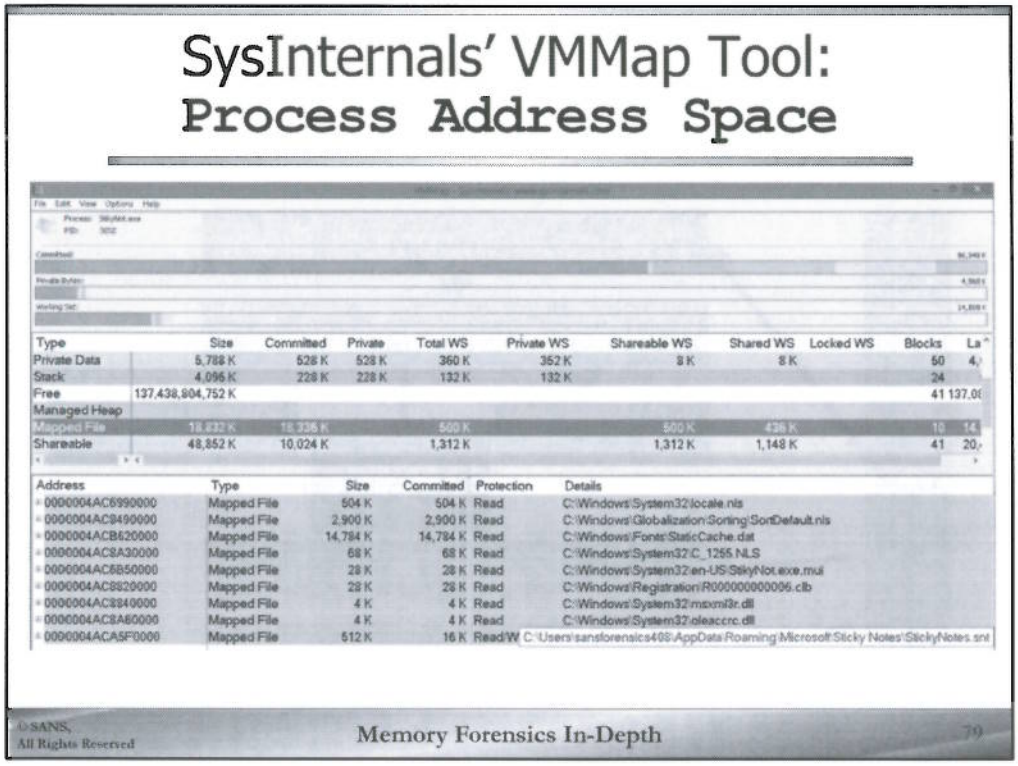
Previously we described a process as a container for many types of data. We're now going to do a more in-depth look at what a process contains. A process contains executable code, which you are most familiar with as a file on the disk like notepad.exe. But a process also contains a Process Environment Block (PEB), a set of Process Parameters, environment variables, one or more heaps, one or more stacks, and a set of DLLs. Heaps are blocks of allocated memory, which we will cover later. Stacks are related to the state of threads, which will also come later.

In this section we're going to focus on the PEB, process parameters, and environment variables. Using these pieces we can get a feel for what was running on the system and whether it was legitimate or not. We're doing this before looking at any of the code or DLLs which were used in this process. That will be in the next section. For now we're looking at, for lack of a better term, metadata. But don't let the name fool you. The details surrounding a process are a rich source of data!

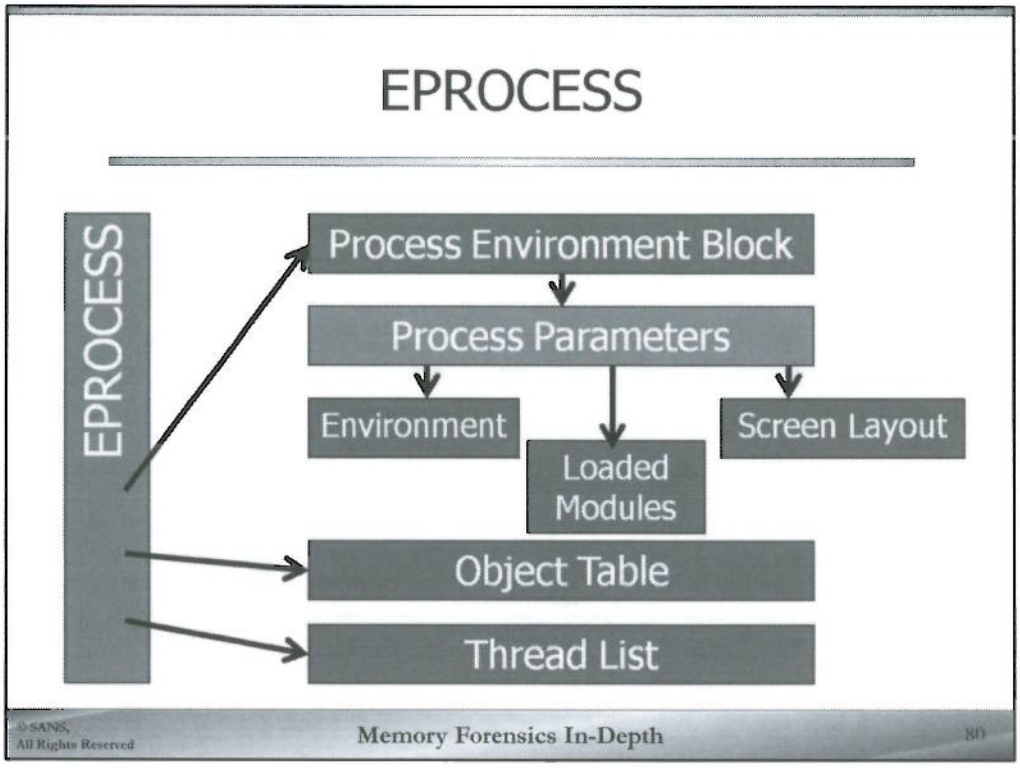
0x00000000 0x7fffffff



- Environment Variables
- Process Parameters
- Heap
- Code
- Process Environment Block
- Stack
- DLLs



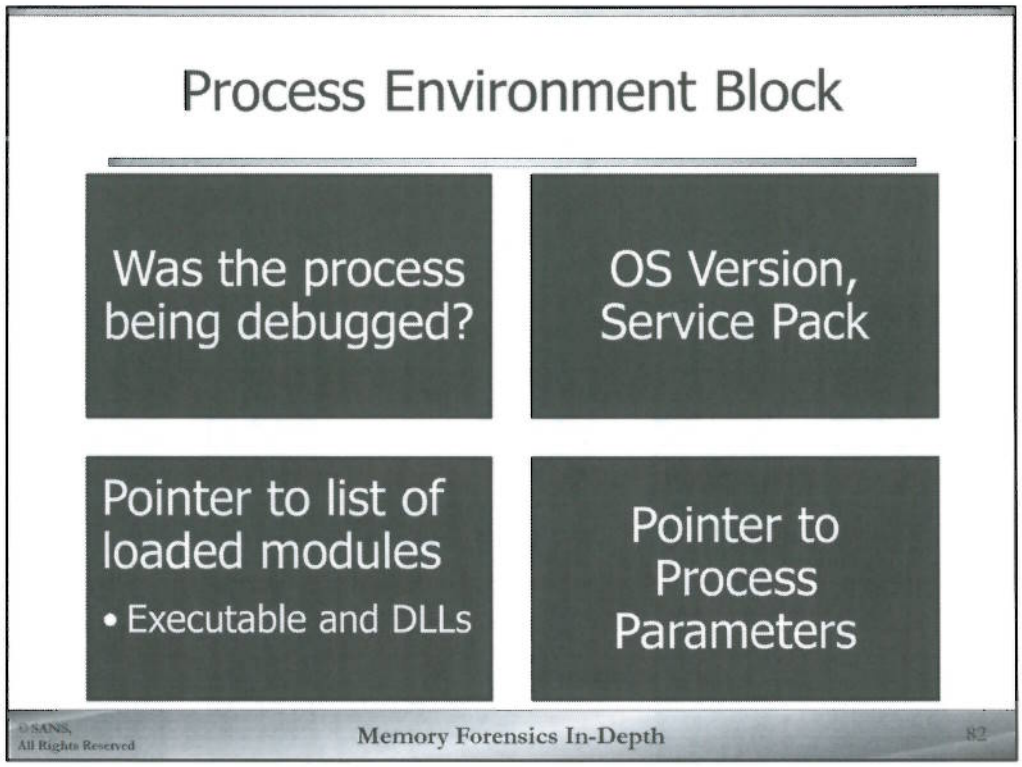
An effective tool for visualizing the virtual address layout of a process, VMMap, is included in your SIFT 8.1 Workstation as part of the SysInternals Suite. Shown in the screen capture above, we see the virtual address space of the StickyNotes process on a x64 system to include the memory range containing the .snt file itself as seen in the VMMap tool process view.



Here's another view of the EPROCESS structure. Although we've been working with these structures, we have yet to fully explore them. Until now we have been looking solely at data contained within the EPROCESS structure. These include things like the Directory Table Base, the Active Process Links, the pid and parent pid, and the Image File Name (the 16 character representation of the executable name). Now we're going to examine some of the structures which the EPROCESS points to, and the structures which they in turn point to.

The EPROCESS has pointers to many structures. We are going to start with the Process Environment Block (PEB), which in turn points to several other structures. Then we'll discuss the object table, thread list, VAD, Section object, and a few others later in the course.

PROCESS
Process Environment Block
Process Parameters
Environment
Screen Layout
Loaded Modules
Object Table
Thread List



The Process Environment Block (PEB), contains information about the process and the operating system as a whole. Let's examine some of the more interesting fields in the PEB. The first is the field "BeingDebugged". This one-bit value is normally set to zero unless the program is being monitored by a debugger. Debuggers are used not only to debug legitimate programs, but also by reverse engineers to closely monitor what a piece of software is doing. Many malicious programs will check if this bit has been set. If it has, the malware author may alter the behavior of the program. They could make their software not run at all, or more insidiously, make it behave differently.

In addition to debugging and reverse engineering, some malicious programs will set up a process and then connect a "debugger" to it. That is, another piece of software will monitor the first one. This is part of a technique designed to defeat reverse engineering. The important thing to note is that during normal operation, almost no legitimate process should be being debugged. That is, for a legitimate process, the BeingDebugged field should always be zero.

Next, there are a few fields related to the operating system and its service pack. These fields include OSMajorVersion and OSMinorVersion, which correspond to the host operating system. The major and minor version numbers are defined by Microsoft at http://msdn.microsoft.com/en-us/library/windows/desktop/ms724833(v=vs.85).aspx. Here's a quick summary of our supported OSes:

Windows XP: 5.1

Windows Server 2003: 5.2

Windows Vista: 6.0

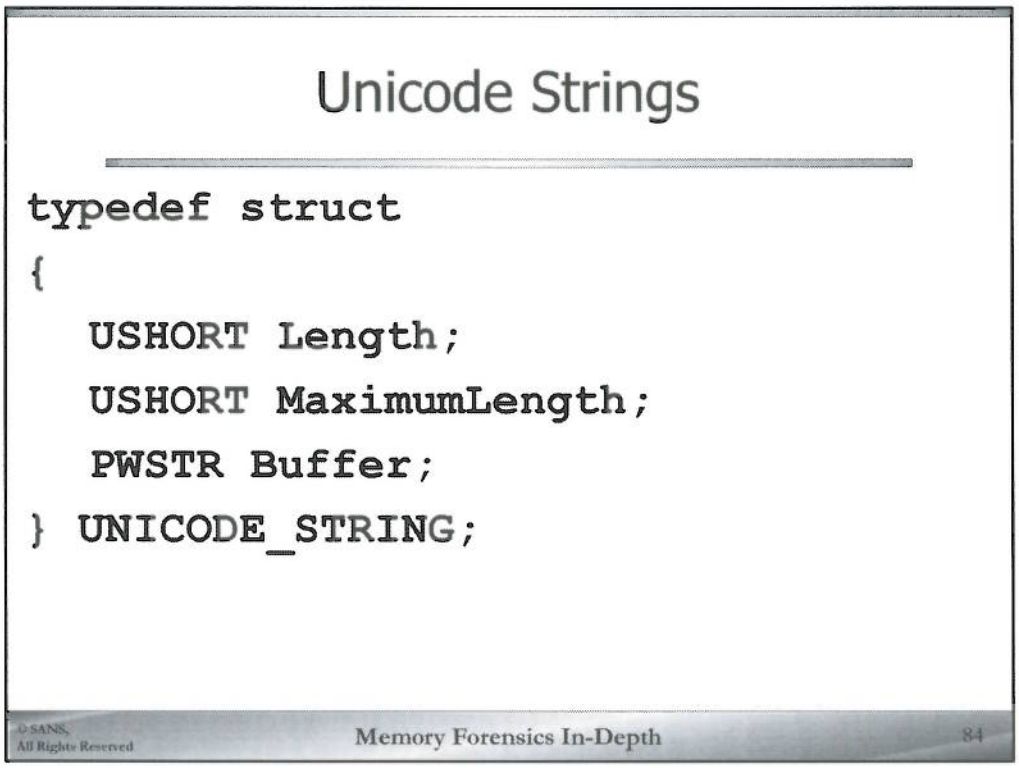
Windows Server 2008: 6.0

Windows 7, 2008R2: 6.1

There is also a field for the build number, OSBuildNumber, which gives a little more granularity into which release of the operating system was running. The Service Pack is indicated by two fields. First, there's a number, OSCSDVersion, which is the service pack number multiplied by 0x100. That is, for Service Pack 2, OSCSDVersion would be set to 0x200. This value is also represented as a string, called CSDVersion, with values such as "Service Pack 2". When Volatility's imageinfo plugin reports the Service Pack used in the operating system, it is counting the most common of the strings found in the CSDVersion strings in the PEBs.

Finally, the PEB has pointers to two other important structures. There is a pointer to the list of loaded modules for this process. Modules are blocks of executable code, including the original executable (e.g. Notepad.exe) and any DLLs used by the program (e.g. ws2_32.dll). The other pointer is to the process parameters.

The PEB is partially documented by Microsoft at http://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx. They only document a few fields and the rest they being "Reserved". That's their term for "undocumented". There is a little bit more documentation, oddly enough, on the Wikipedia page for the PEB, http://en.wikipedia.org/wiki/Process_Environment_Block.

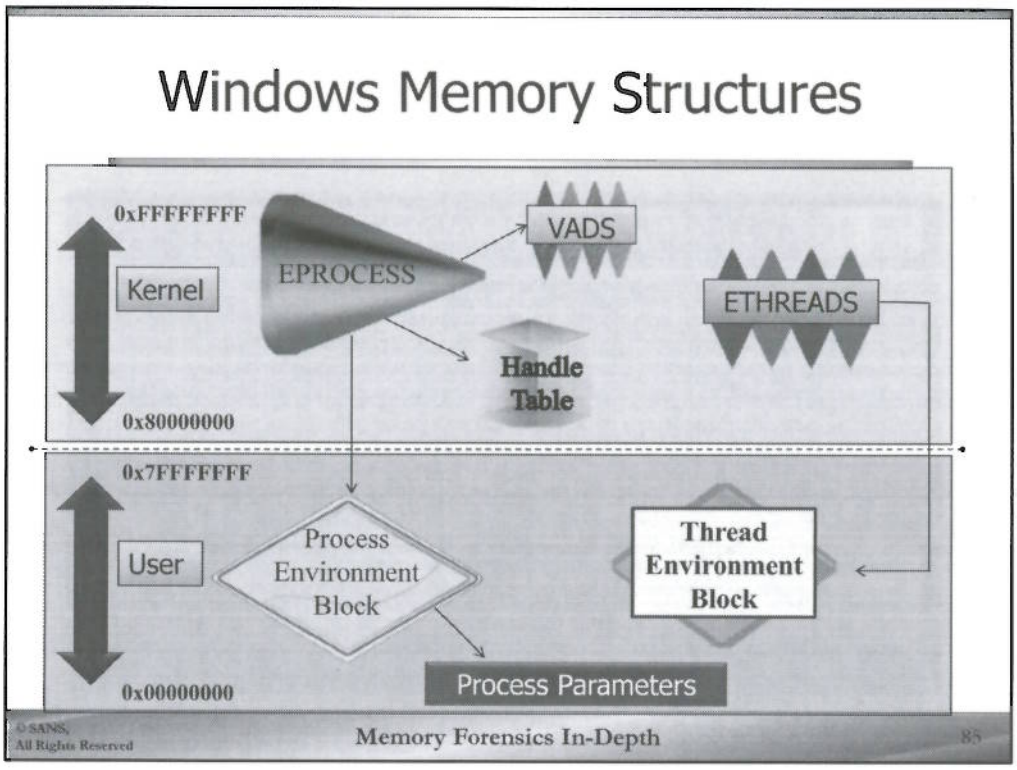


On the last page we mentioned the field CSDVersion as being a string. Although this is true, it is technically a structure called UNICODE_STRING, and, as you might guess, refers to a Unicode string. Thankfully you will not need to parse this structure by hand, but you should understand that this structure does not contain the string data directly. Instead it contains pointers to the actual string data, along with the string's length. When you are looking at a memory image, don't be surprised when you don't see the readable characters in the UNICODE_STRING structure.

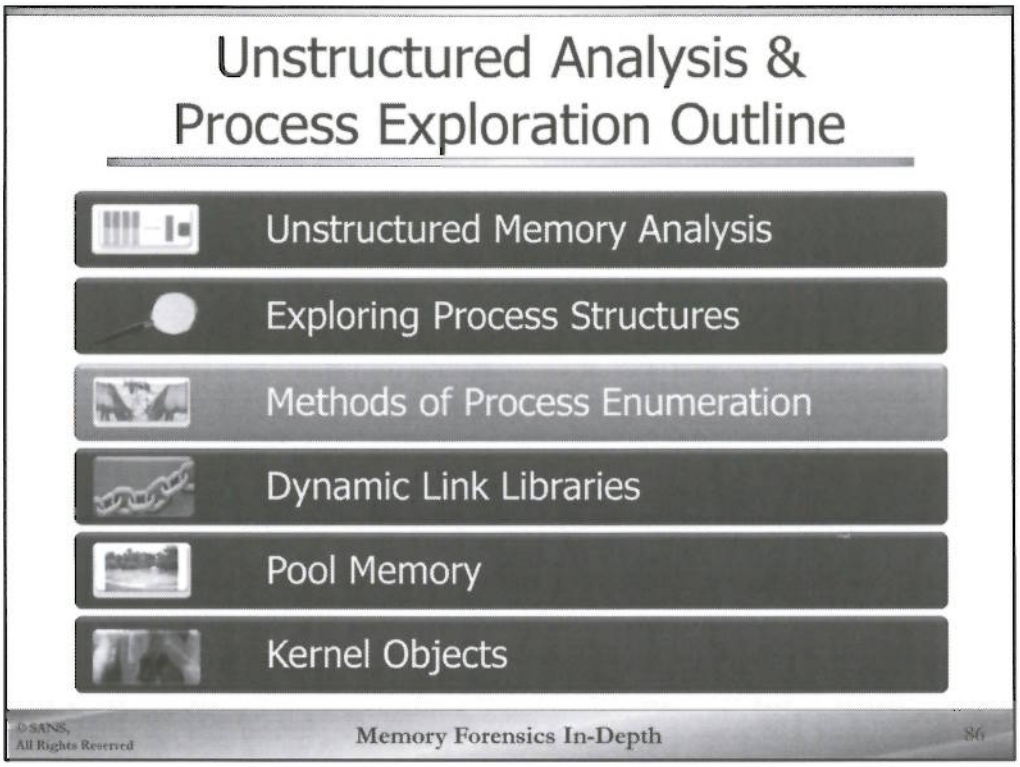
The structure has three parts: A length value, a maximum length value, and a pointer to the buffer where the data lives. (As a sanity check, your forensics tool should check that the actual length is shorter than the maximum length!) In the picture above, USHORT stands for 'unsigned short', which in Microsoft's view of the world is a 16-bit unsigned value. It can take on values between zero and 2^16-1 = 65535. The prefix 'P' in the line for Buffer denotes a pointer. In this case, a pointer to a WSTR, or a wide string.

An important gotcha for programmers working with UNICODE_STRING structures is that the buffer should be 'Length' bytes long. That buffer may contain NUL characters, or \0 characters. In standard C strings a NUL character signifies the end of the string. But these special structures can contain NULs which aren't the end of the string. This is especially important when reading the command lines for processes, for example. Several legitimate Windows system processes have command lines which contain such NUL characters.

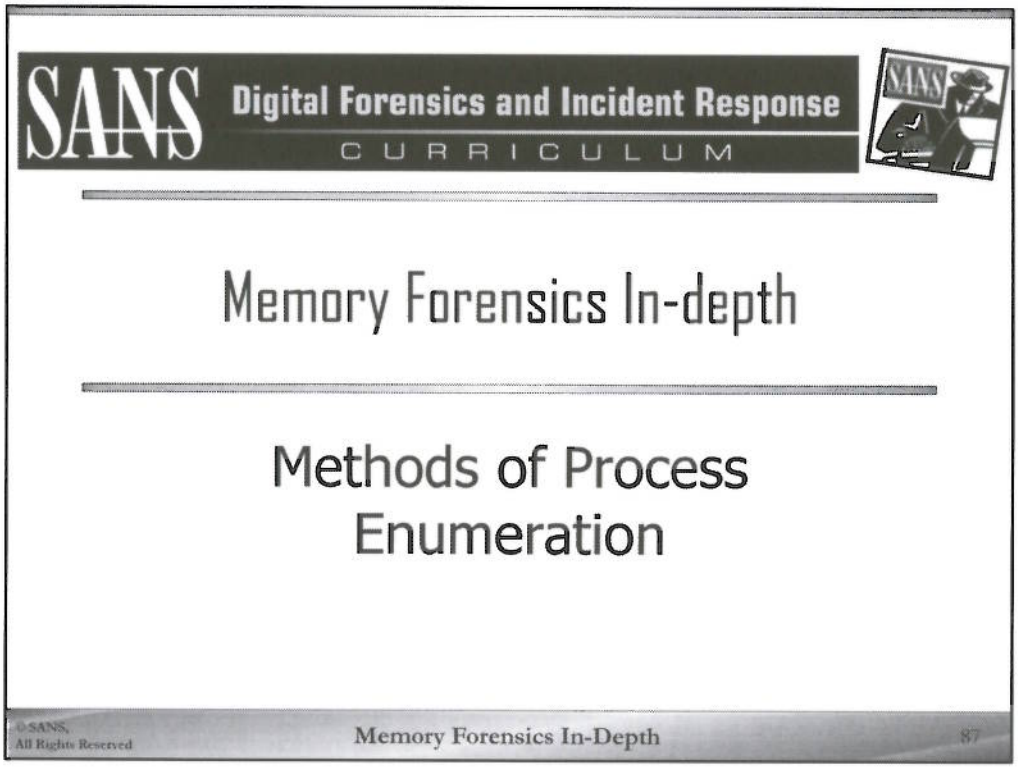
As a final note, modern Windows systems encode the buffer using UTF-16. Prior to Windows XP, the data was encoded with UCS-2.



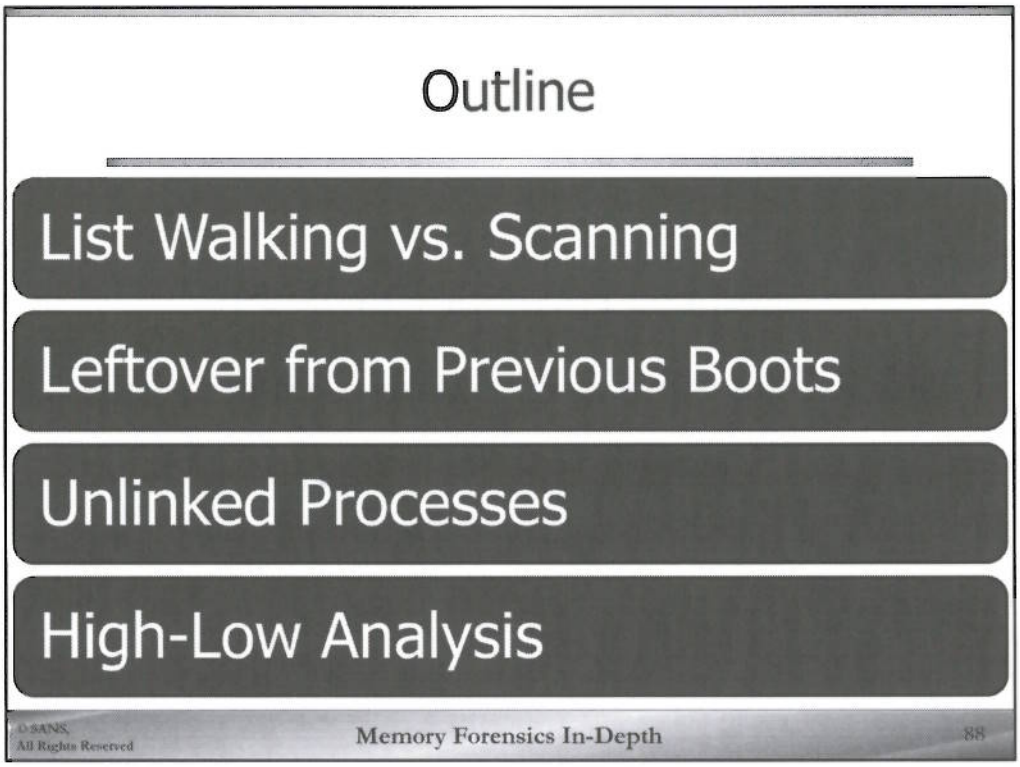
So far, we have discussed the significance of many of the parameters maintained within the Process Environment Block. This diagram above gives you a general overview of process structures and how they relate to one another. As you can see, the EPROCESS structure is in kernel memory and the PEB is in user space. There is a similar relationship between the ETHREAD and Thread Environment Block structures. We will be discussing the VAD structures, Virtual Address Descriptors, and the significance of process Handles later in this course.



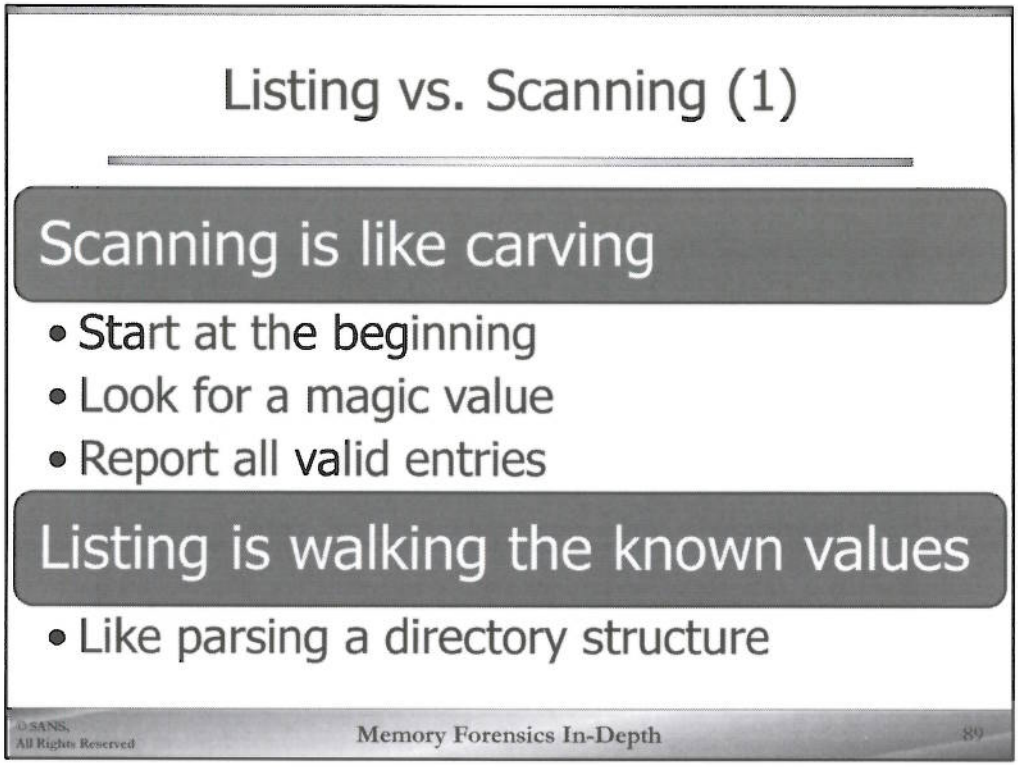
This page intentionally left blank.



This page intentionally left blank.



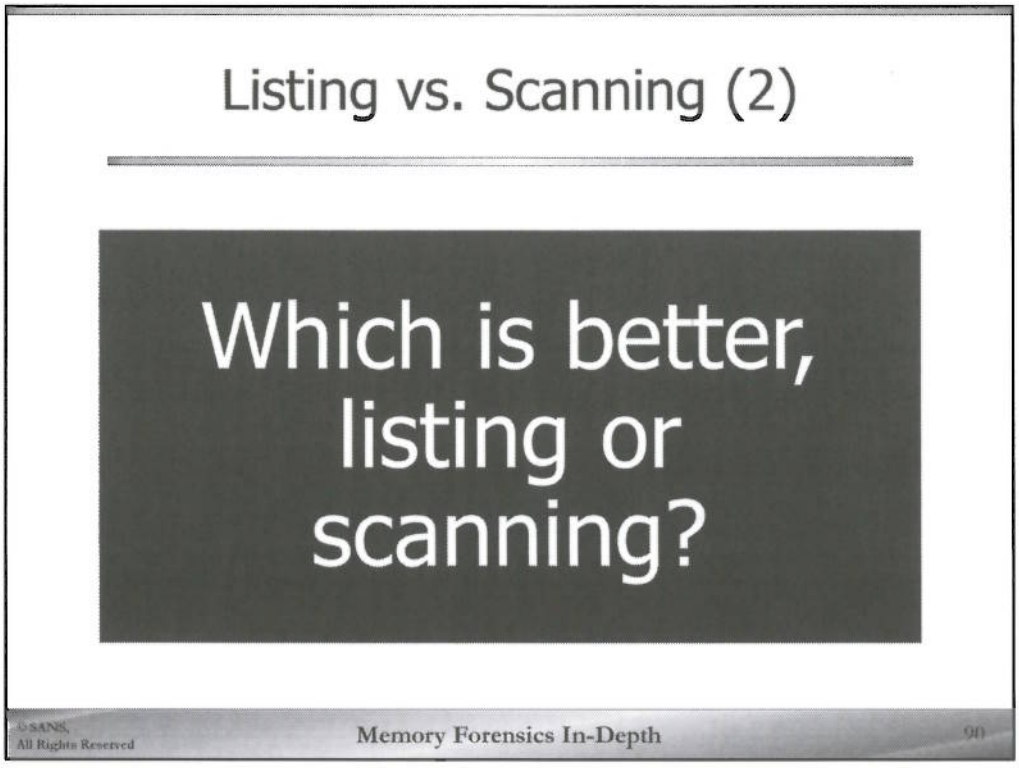
This page intentionally left blank.



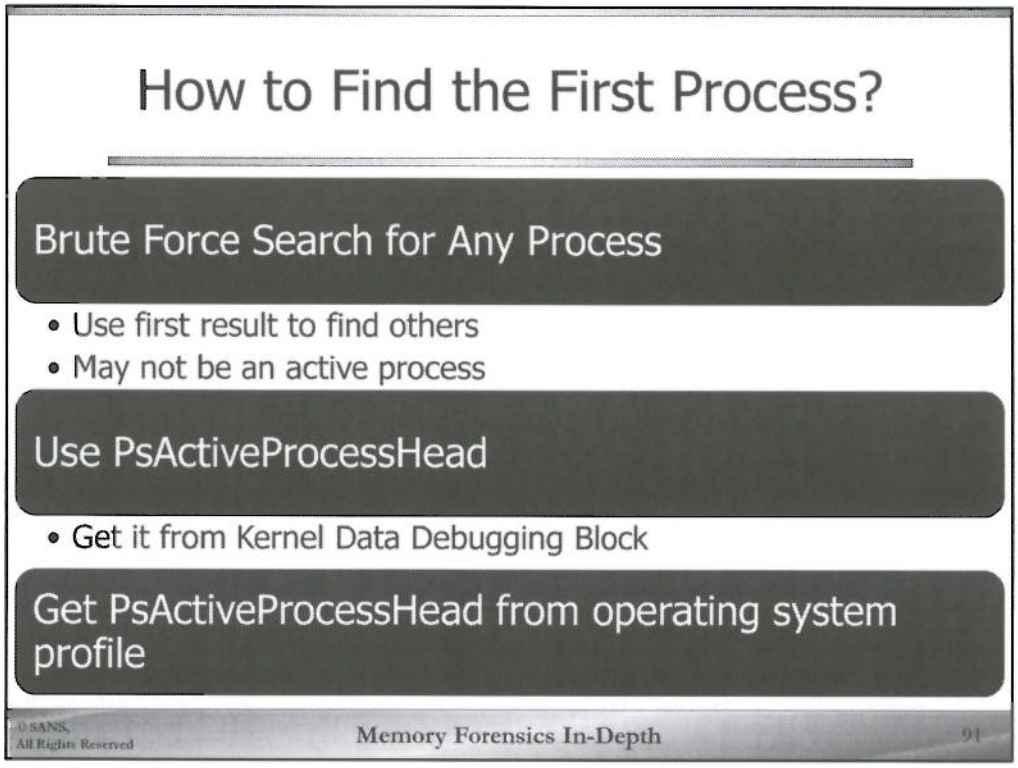
Until now we've been using brute force techniques to do memory analysis. We are now going to start looking at data structures maintained by the operating system. The technique we've been doing so far, scanning, starts at the beginning of the data and searches through to the end. Listing starts at a specified offset and parses the operating system structures. It's like walking the directories of an NTFS volume.

Listing is faster than scanning. When listing, ideally we only parse data which is valid and the operating system knew about. (We do have to be careful about finding invalid data, especially in memory images, which are not usually acquired cleanly like a disk image.) Of course, as with looking at file systems, listing misses any deleted or hidden data. Scanning is slower, but finds the data in question.

In this module we'll discuss how we get started with listing data used by the operating system, what we find there, and comparing the results to what we found with scanning.



Which is the better technique to use in your investigations? It completely depends on what you're trying to do. If you want to see things as the operating system did, you want to do listing. If you want a low-level view of what was in memory, regardless if the operating system knew about it or not, you want to do scanning.

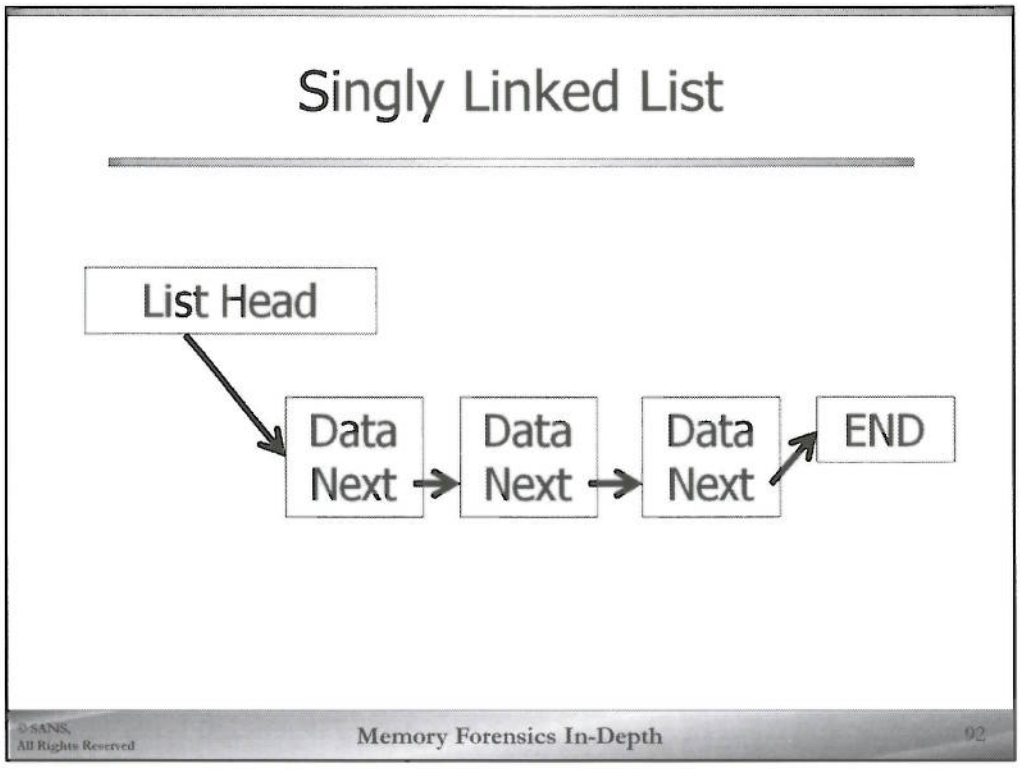


In order to walk any list of operating system structures, we have to start *somewhere*. Unfortunately we usually don't have a good place to start. There is no volume header like there is on a disk. Instead we have to find a first structure and then use it to find the others. There are several methods to do this.

We could use brute force searching, or scanning, to find a process--any process. That process could serve as the starting point for a walk of operating system processes. Unfortunately we don't know if the process we found was active. That is, it might be a leftover from a previous boot. Following the pointers from such a process would quickly lead us into potentially invalid data. As such, choosing a random process at the start of a process walk is not advised.

We could use the kernel variable PsActiveProcessHead. This variable, found in the Kernel Data Debugging Block, should point into the list of active processes. As this variable was maintained by the operating system, it should still be valid. Using the found value of PsActiveProcessHead is a good choice for doing the walk.

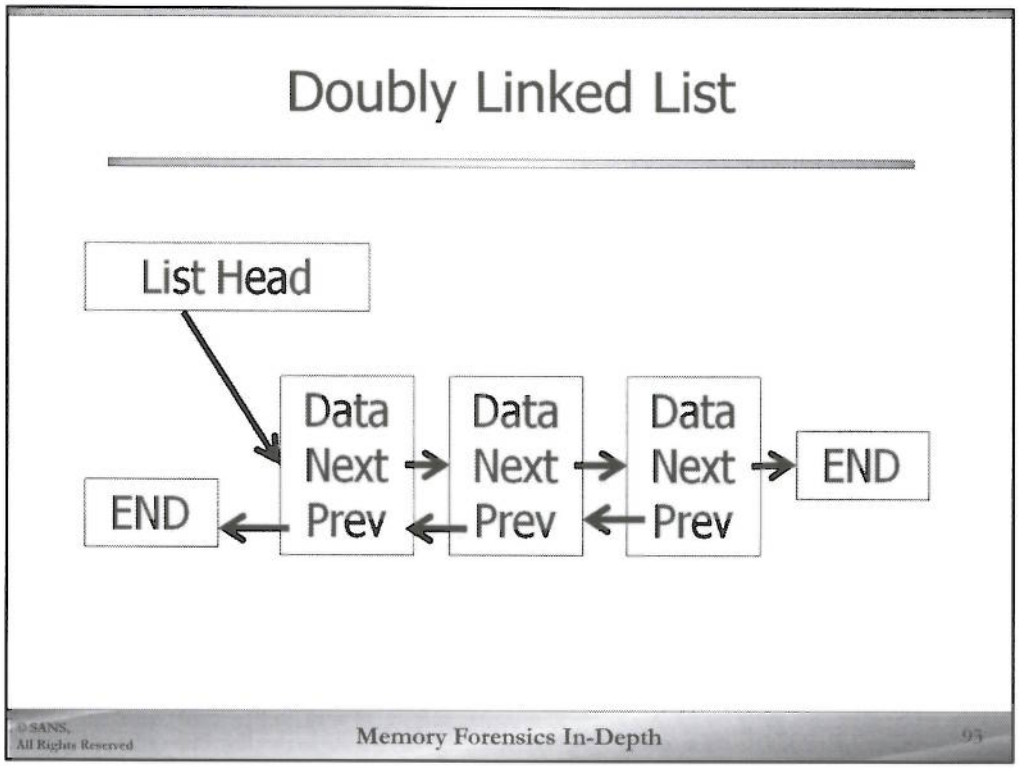
Without doing any kind of search, we could use the value for PsActiveProcessHead associated with an operating system and service pack profile. Although some kernel variables change as the result of hotfixes, this is a good fallback method.



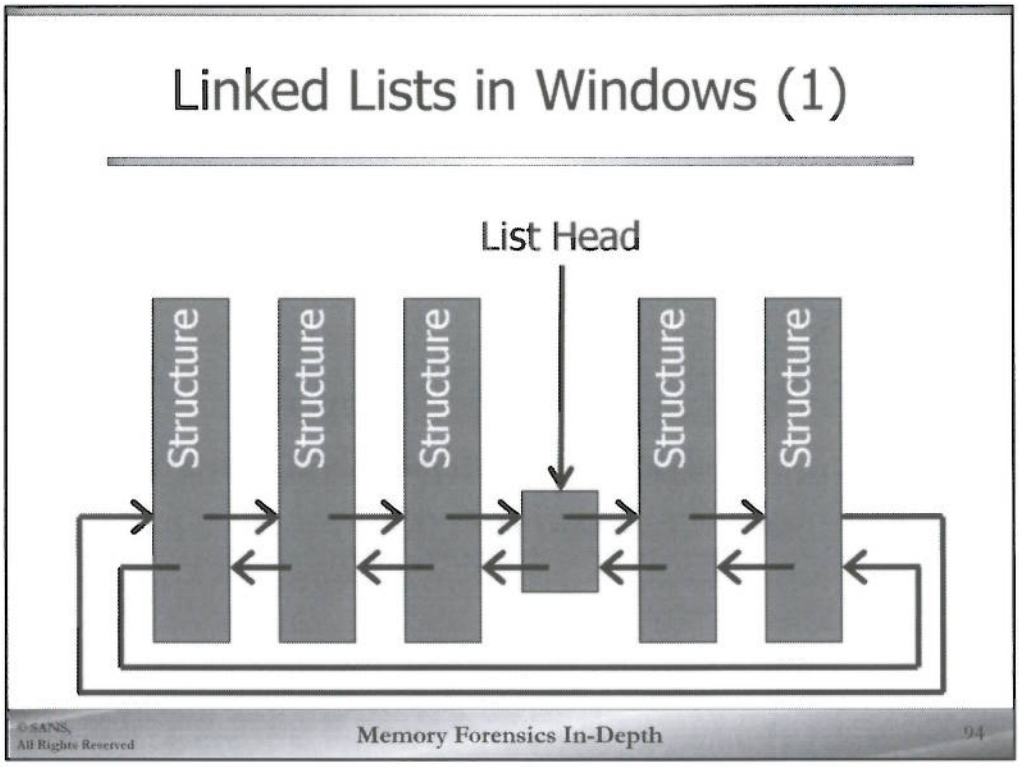
To talk about listing and list walking we need to talk about linked lists. A linked list is a data structure used to hold a variable number of items. The list starts with a pointer to the head of the list. Usually a pointer gives the address in memory where the first item of the list is stored, but it could also point to an offset in physical memory. Each item in the list contains some data, and then a pointer to the next item in the list. The last item in the list points to a special piece of data which indicates that there are no more items.

A program can enumerate all of the items in the list by starting at the head, examining each datum, and then proceeding to the next datum, and repeating until there are no more items left to process.

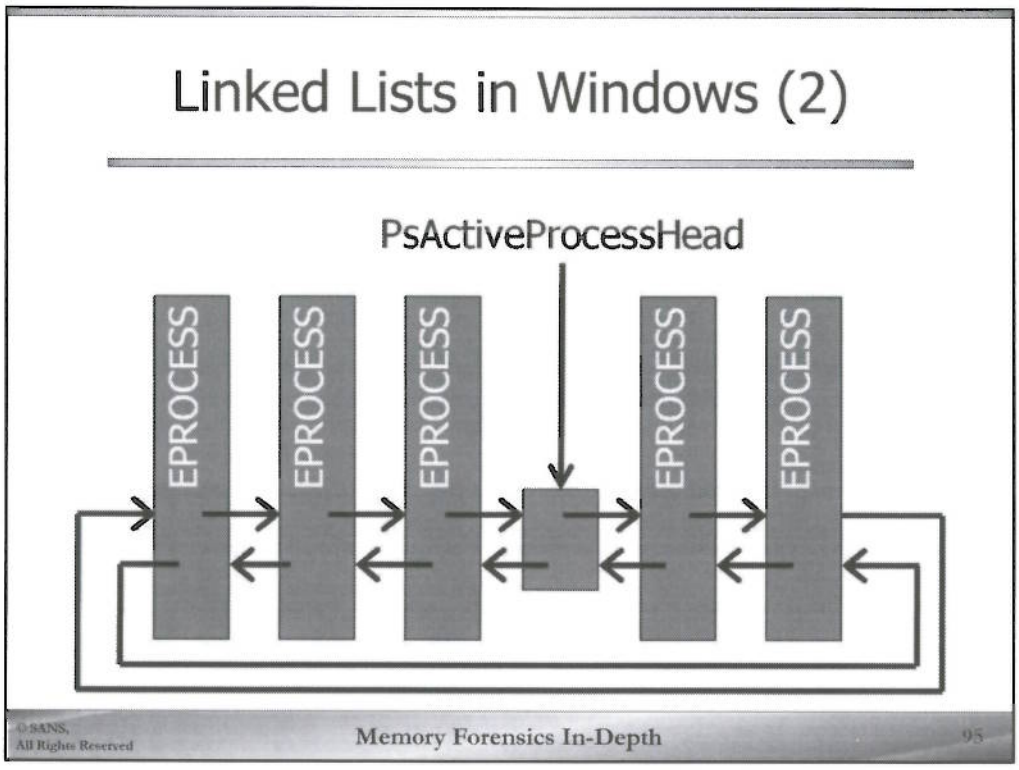
This picture shows a singly linked list. Each item in the list has some data and a pointer to the next item. It's called singly linked, because each item in the list has only one link. This is different from a doubly linked list, as shown on the next page.



In a doubly linked list, each item has a pointer to both the next item in the list, and also the previous item in the list. This way a program can traverse the list by starting at the list head, but can then go forward or backward along the list. Each end of the list has the special pointer to denote the end of the list. These special pointers are not actually required. We could make the list entirely circular by having the end of the list point back to the beginning, as you'll see on the next slide.



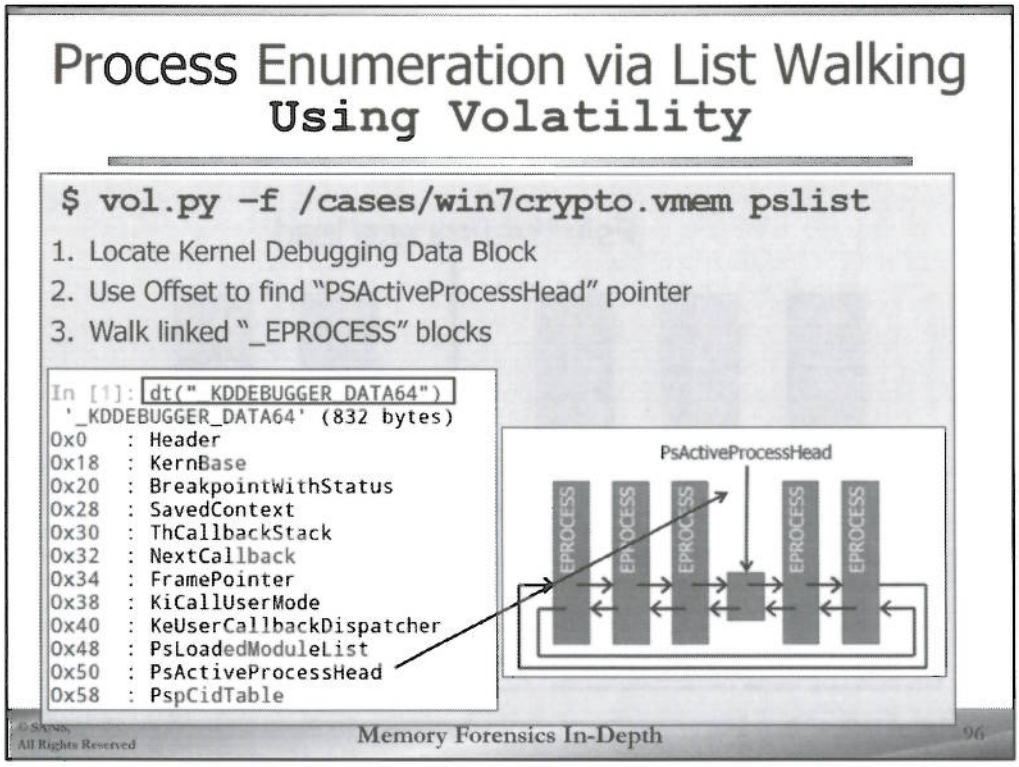
Microsoft likes to use not only doubly linked lists, but circular doubly linked lists. These are similar to doubly linked lists, but there is no 'end' value to denote the last item in the list. A program examining the list would just end up back where it started. The "start" of the list is a special value, the list head. Note that all of the pointers in the list exist in the middle of the structure, and point to the pointers in the middle of the next structure. The list head is not part of any element in the list. It's just a set of pointers to other values. A program walking the list should start at the list head, and keep reading values until it gets back to and reads the list head again.



Here's an example of a real doubly linked list in Windows. We've been talking about processes, so let's look at the Active Process Links. The list head for the list of processes is a kernel variable called PsActiveProcessHead. It's one of the values found in the KDBG. The forward and backward pointers of PsActiveProcessHead point into the EPROCESS structures, which in turn describe the processes on the system. Again, note that these pointers are in the middle of the EPROCESS structure. In fact, the offset of these pointers varies slightly between operating systems.

A good way to test the integrity of any step in the list walking procedure is to follow a link, and then check that the link from the new data structure going the other way points to where we just were. For example, let's say we were starting the list parsing at PsActiveProcessHead, above. We follow the forward link and go one process to the right. The backward link in that process should point at PsActiveProcessHead. If it doesn't, something has gone wrong.

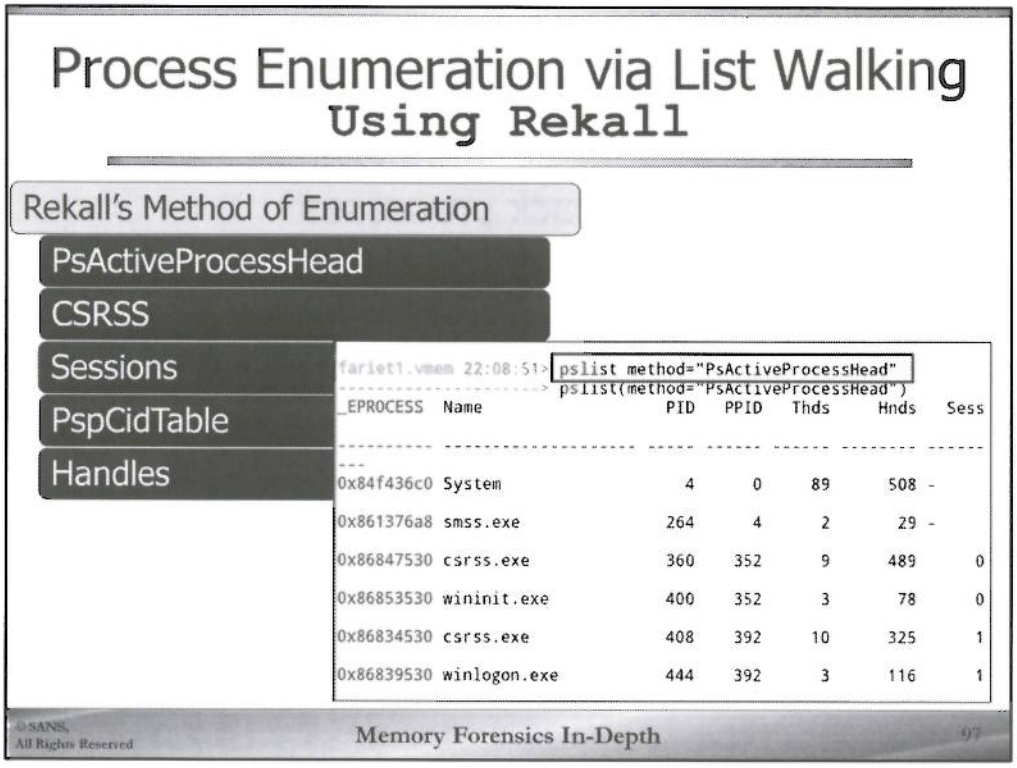
You may be thinking that rather than search for the KDBG, we could just find any random process on the system, using scanning, and then follow the pointers in that process to find the other processes on the system. This could work, but only if we were lucky enough to find a process which was in the list of active processes during our scan. If we start on a process which isn't in the list, such as a terminated process or a leftover from a previous boot, we might follow a pointer into invalid memory.



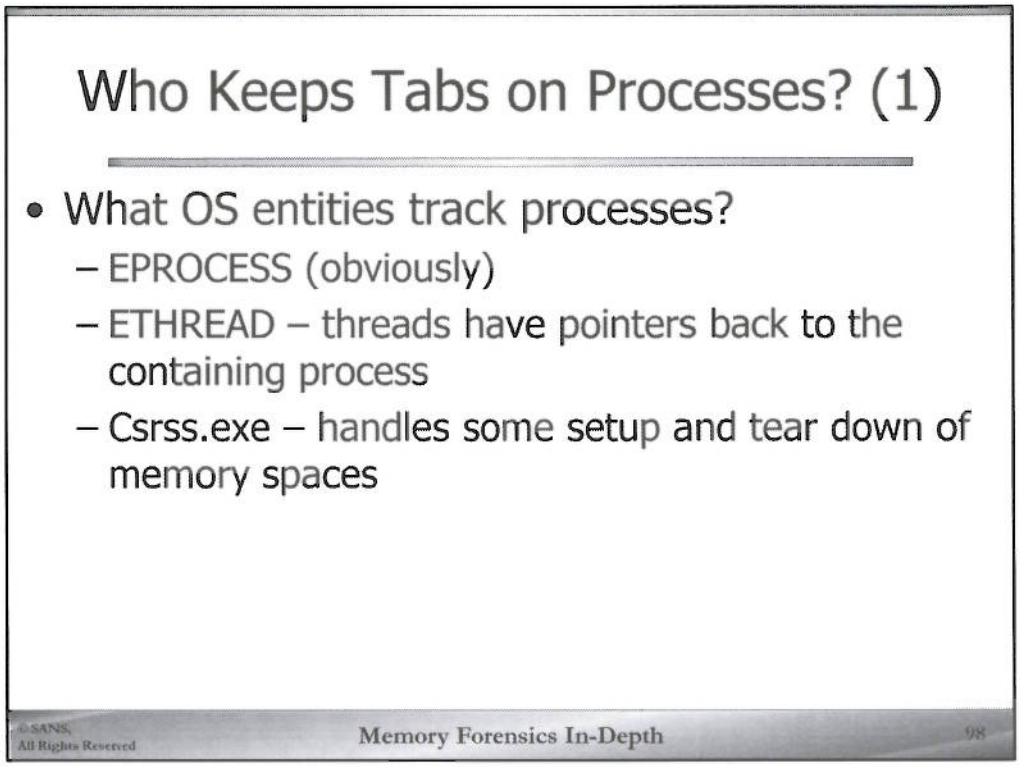
Shown above is the method of _EPROCESS block list-walking employed by the Volatility Framework. When an examiner runs **pslist**, Volatility first locates the KDBG, and based on the profile specified in the pslist command, goes to the offset specific to the Windows version that maintain the psActiveProcessHead, the pointer to the beginning of the doubly-linked list of _EPROCESS blocks. In the Windows 7 SP0x86 KDBG structure shown above, the PsActiveProcessHead is found at offset 0x50.

Inability to identify the KDBG is an obvious point of failure for many Volatility plugins that walk lists (to include pslist, modules, dlllist and connections). Without finding the KDBG, these plugins are unable to parse the linked kernel objects. Below is the output from a failed run of **pslist** after the system's KDBG structure signature was been modified to evade detection.

```
sansforensics@siftworkstation:/cases$ vol.py -f ds_futz.img --profile=WinXPSP2x86
pslist
Volatility Foundation Volatility Framework 2.4
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start
Exit
---------- -------------------- ------ ------ ------ -------- ------ ------ -------
----------------------- ------------------------------

sansforensics@siftworkstation:/cases$ vol.py -f ds_futz.img --profile=WinXPSP2x86
kdbgscan
Volatility Foundation Volatility Framework 2.4
sansforensics@siftworkstation:/cases$
```



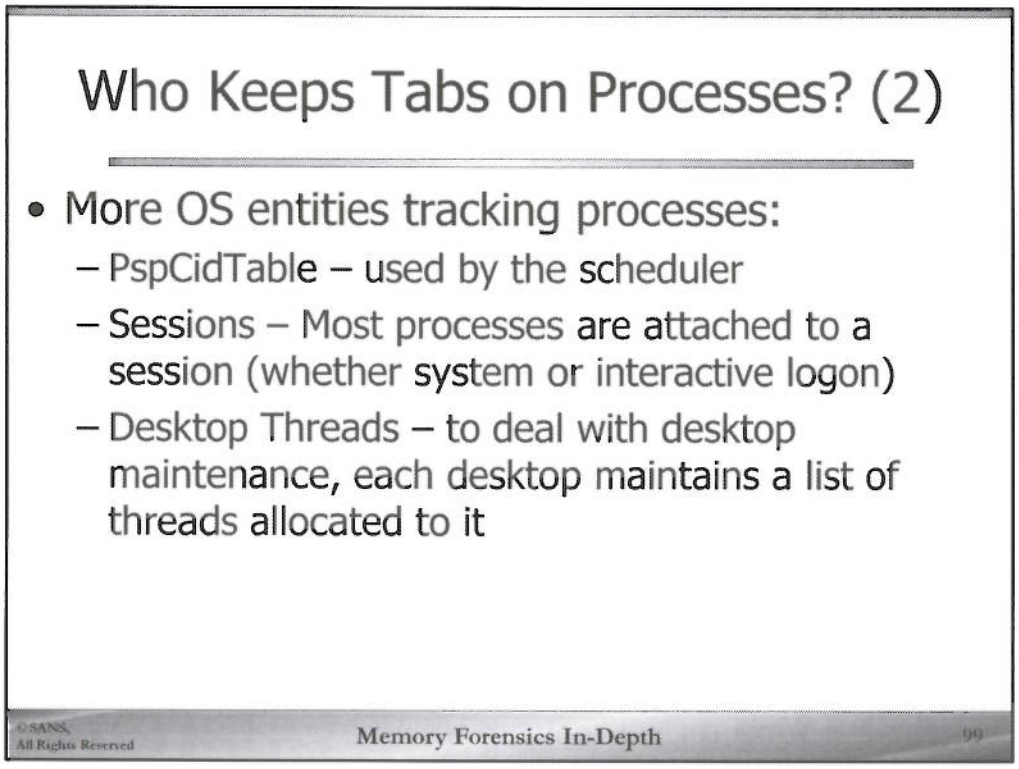
The pslist module in Rekall makes use of five methods of list walking to perform process enumeration: PsActiveProcessHead, CSRSS, Sessions, PspCidTable and Handles. These additional resources for enumerating processes will be covered in the upcoming section. An examiner can specify a single method for Rekall's pslist module to use by invoking the plugin with method="PsActiveProcessHead" for example. In a default run of Rekall's pslist, all of these enumeration methods are included - therefore the output will be more comprehensive than that of Volatility's pslist. In addition, Rekall's pslist output will not be in chronological order.



EPROCESS - The doubly linked list of EPROCESS structures is obviously a good place to start looking for processes. So good in fact, that it's the default that is used with built-in Windows commands such as tasklist or the task manager. Even if malware removes an entry from the doubly linked list, it may be discovered by scanning (psscan).

ETHREAD – Each ETHREAD has a pointer to a KTHREAD structure. This in turn points to a structure KAPC_STATE. This then turns to a KPROCESS structure for the process. Wow, that was a lot of pointers! But think about it, if we can scan for threads on the system, we may find processes that were unlinked from the EPROCESS list and for whatever reason were missed in the psscan.

Csrss.exe – All processes have handles. They maintain a table of handles to OS objects they are using. Usually these objects are things like files or registry keys. Csrss.exe is special however. It has in it's handle table a handle to all other processes on the system. This is because it is responsible for handling some setup and tear down of processes when they start and stop. For example, csrss.exe is responsible for notifications to the desktops that processes have attached or are detaching.



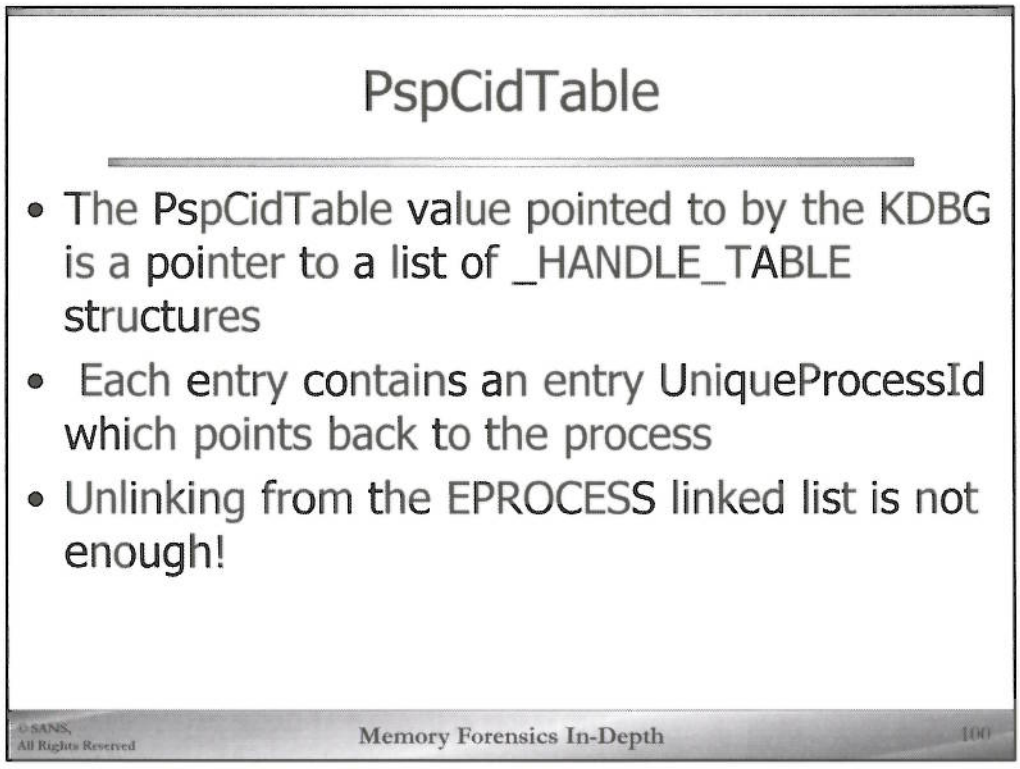
PscpCidTable – this structure is used by the scheduler. We discuss it in more detail on the next slide. There's a good reference to the PspCidTable here:

http://uninformed.org/index.cgi?v=3&a=7&p=6

Sessions – with the exception of System and smss.exe, processes will be associated with a session. Per the technet blog, "a session consists of all of the processes and other system objects that represent a single user's logon session." (http://blogs.technet.com/b/askperf/archive/2007/07/24/sessions-desktops-and-windows-stations.aspx). Each session has a LIST_ENTRY member called 'ProcessList' that points to the processes associated with the session. By finding one process in a session, (and locating it's _MM_SESSION_SPACE object), we can easily locate other processes belonging to the same session. No rootkit is safe*!

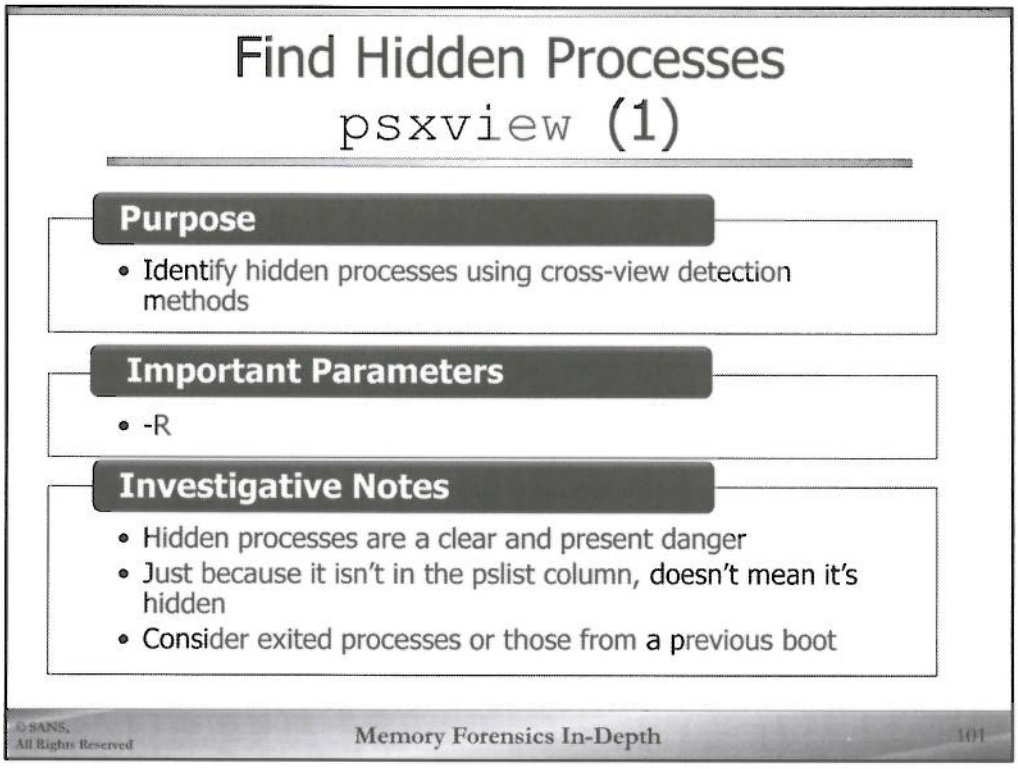
Desktop Threads – Desktops have Window Stations. These are transparent to the user, but they are there (even for invisible desktops). Each Window Station has a pointer to a list of threads that are associated with the station. Since we already know that threads point back to processes, we are able to locate otherwise hidden processes. You can hide, but you must run!

* Actually, rootkits might be safe, it just requires a LOT more work.



The PspCidTable is pointed to by the KDBG. It contains a pointer to a list of _HANDLE_TABLE structures (definition shown below). Note that one member of the _HANDLE_TABLE structure is UniqueProcessId, which holds the process ID of the process. Clearly, simply unlinking the process from the doubly linked list of EPROCESS structures is not enough. The process will not be hidden if we can enumerate the PspCidTable.

```
typedef struct _HANDLE_TABLE {
        PVOID p_hTable;
        PEPROCESS QuotaProcess;
        PVOID UniqueProcessId;
        EX_PUSH_LOCK HandleTableLock [4];
        LIST_ENTRY HandleTableList;
        EX_PUSH_LOCK HandleContentionEvent;
        PHANDLE_TRACE_DEBUG_INFO DebugInfo;
        DWORD ExtraInfoPages;
        DWORD FirstFree;
        DWORD LastFree;
        DWORD NextHandleNeedingPool;
        DWORD HandleCount;
        DWORD Flags;
}
```



The psxview plugin locates hidden processes by using cross view detection. What is cross view detection? It's just like what the police do when questioning a suspect: ask lots of questions and hope that the suspect gets caught up in a lie (if they are lying to begin with). Psxview tries to enumerate processes using multiple methods. It then outputs for each of these methods whether a given process was found.

The methods used for detection are:

Pslist

Psscan

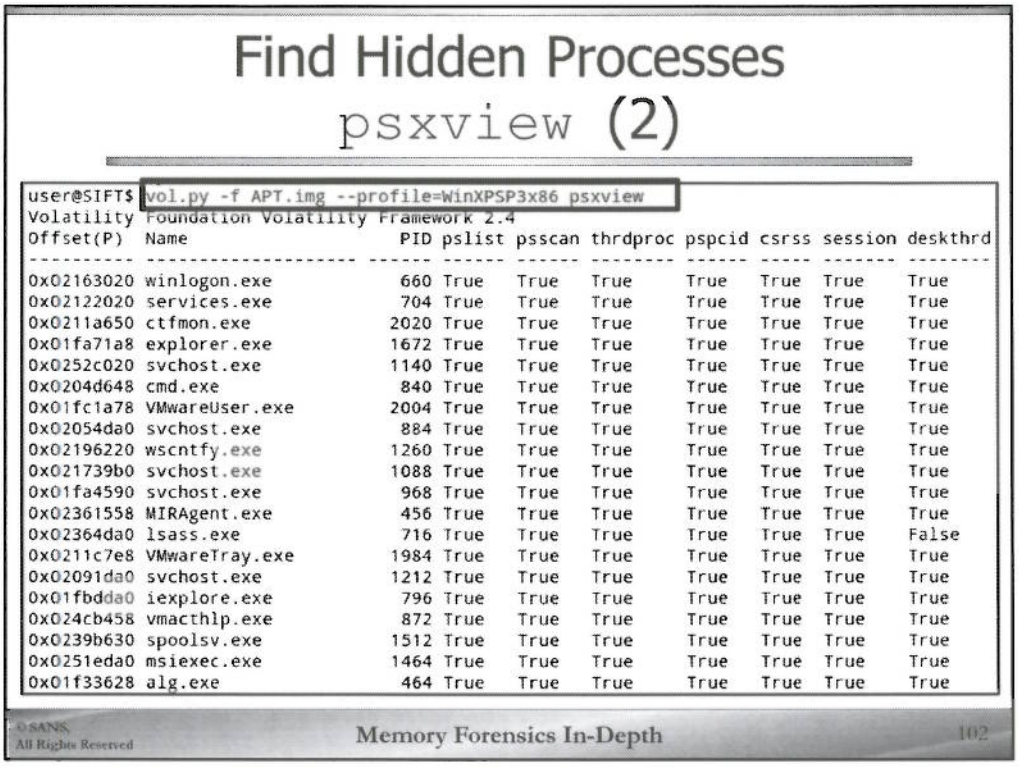
Thrdproc

Pspcid

Csrss

Session

Desktop thread



This slide shows the output of running the psxview plugin against a sample memory image.

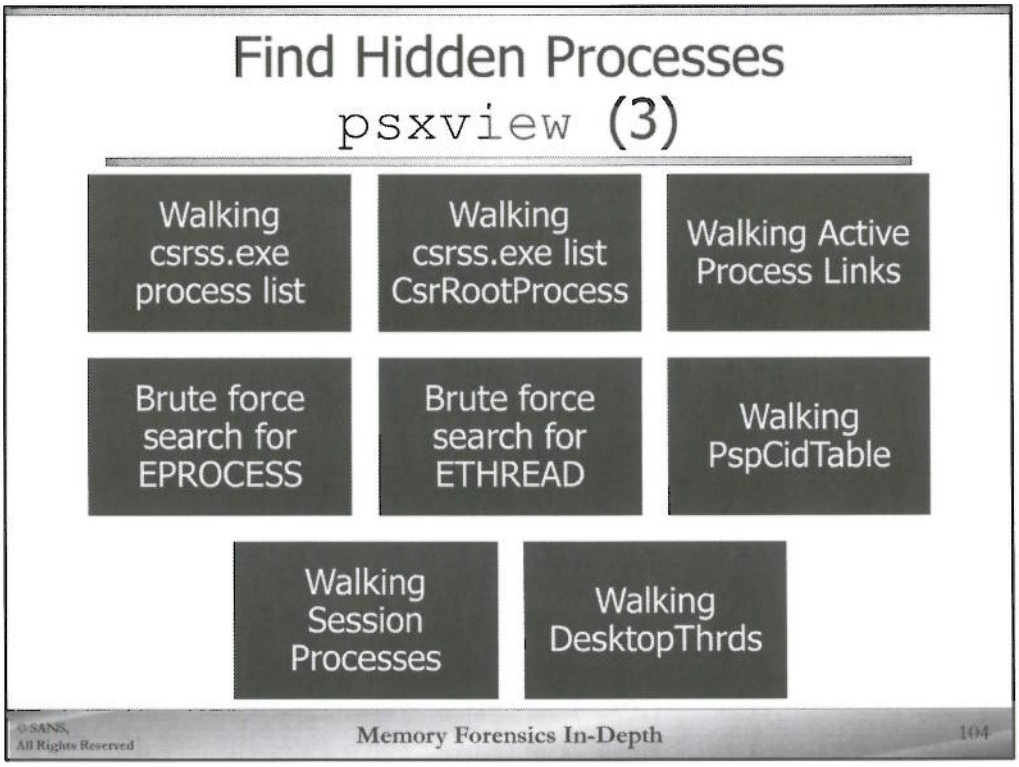
user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 psxview

Volatility Foundation Volatility Framework 2.4

Offset(P)	Name	PID	pslist	psscan	thrdproc	pspcid	csrss	session	deskthrd	ExitTime
0x02163020	winlogon.exe	660	True	True	True	True	True	True	True	
0x02122020	services.exe	704	True	True	True	True	True	True	True	
0x0211a650	ctfmon.exe	2020	True	True	True	True	True	True	True	
0x01fa71a8	explorer.exe	1672	True	True	True	True	True	True	True	
0x0252c020	svchost.exe	1140	True	True	True	True	True	True	True	
0x0204d648	cmd.exe	840	True	True	True	True	True	True	True	

… output truncated …

0x02091da0 svchost.exe	1212	True	True	True	True	True	True	True
0x01fbdda0 iexplore.exe	796	True	True	True	True	True	True	True
0x024cb458 vmacthlp.exe	872	True	True	True	True	True	True	True
0x0239b630 spoolsv.exe	1512	True	True	True	True	True	True	True
0x0251eda0 msiexec.exe	1464	True	True	True	True	True	True	True
0x01f33628 alg.exe	464	True	True	True	True	True	True	True
0x01fc2570 VMwareService.e	1032	True	True	True	True	True	True	True
0x0250aad8 smss.exe	564	True	True	True	True	False	False	False
0x025c8830 System	4	True	True	True	True	False	False	False
0x024ca2c0 csrss.exe	636	True	True	True	True	False	True	True
0x03178220 wscntfy.exe	1260	False	True	False	False	False	False	False
0x0c605020 svchost.exe	1140	False	True	False	False	False	False	False
0x0ad69da0 iexplore.exe	796	False	True	False	False	False	False	False
0x0edd0628 alg.exe	464	False	True	False	False	False	False	False
0x032b3da0 svchost.exe	884	False	True	False	False	False	False	False
0x0eed3628 alg.exe	464	False	True	False	False	False	False	False
0x10b54628 alg.exe	464	False	True	False	False	False	False	False
0x15934830 System	4	False	True	False	False	False	False	False
0x1b217da0 iexplore.exe	796	False	True	False	False	False	False	False
0x04097020 svchost.exe	1140	False	True	False	False	False	False	False
0x035c1590 svchost.exe	968	False	True	False	False	False	False	False
0x07b1ada0 iexplore.exe	796	False	True	False	False	False	False	False
0x0edd59b0 svchost.exe	1088	False	True	False	False	False	False	False
0x12f3dda0 svchost.exe	884	False	True	False	False	False	False	False



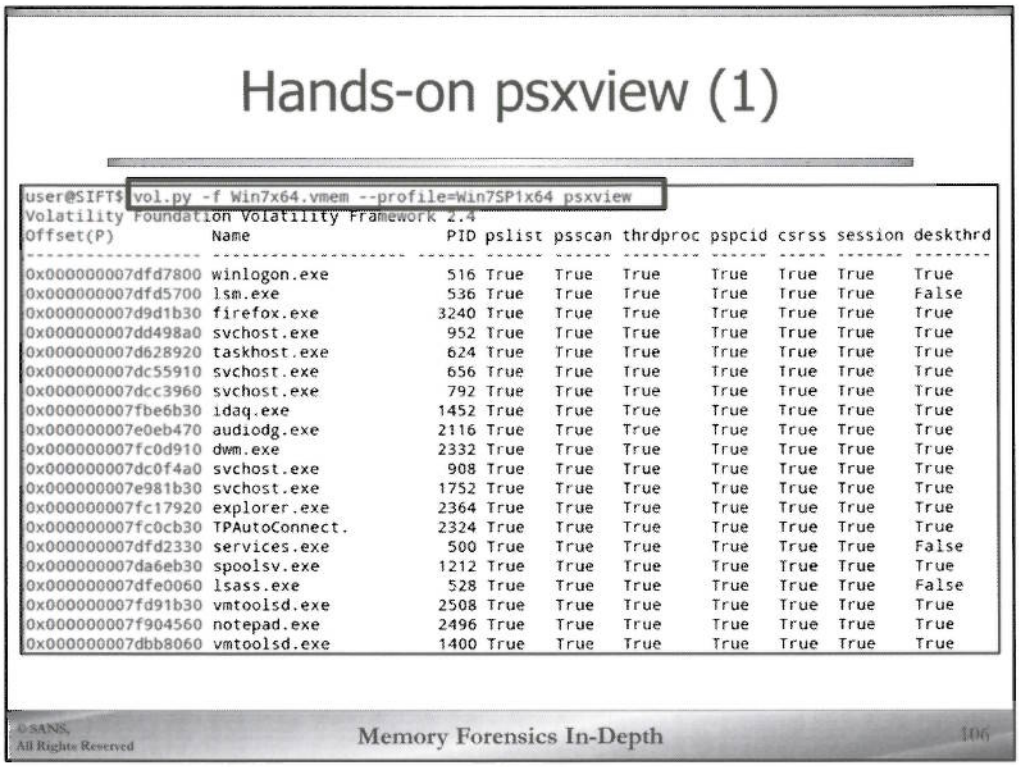
The psxview plugin displays information about each process on the system found using a variety of list walks and brute force searches. The information is designed to help you find hidden processes. The potential processes are found by:

- Walking the list of processes found in the Client/Server Runtime Subsystem (csrss.exe).
- Walking the list of handles in the csrss.exe list, CsrRootProcess, and following them back to their owning process. This is a completely undocumented structure inside of the csrss.exe. It's a great example of how, if you do enough research on Windows kernel structures, eventually all of the references you find on the web are in Chinese.
- Walking the list of processes found via the PsActiveProcessHead variable (i.e. the Active Process links).
- A brute force search for EPROCESS objects.
- A brute force search for ETHREAD objects and following them back to their owning process.
- Walking the PspCidTable. This is a table of handles which point back to processes. It's used by the scheduler.
- Session - enumerates Session processes
- DesktopThrd - enumerates processes from Desktop threads

For each of these resources, the plugin displays a 'True' if the process found via that method. If the process was not found via that method, the plugin displays a 'False'. There is a complete command reference at http://code.google.com/p/volatility/wiki/CommandReference#psxview. The reference notes, "On Windows Vista and Windows 7 the internal list of processes in csrss.exe is not available. It also may not be available in some XP images where certain pages are not memory resident."

Also note that the csrss process doesn't hold information about processes which started before it did. You'll see some zeros in the output for this--don't be alarmed!

The PspCidTable is a handle table for process and thread ids.

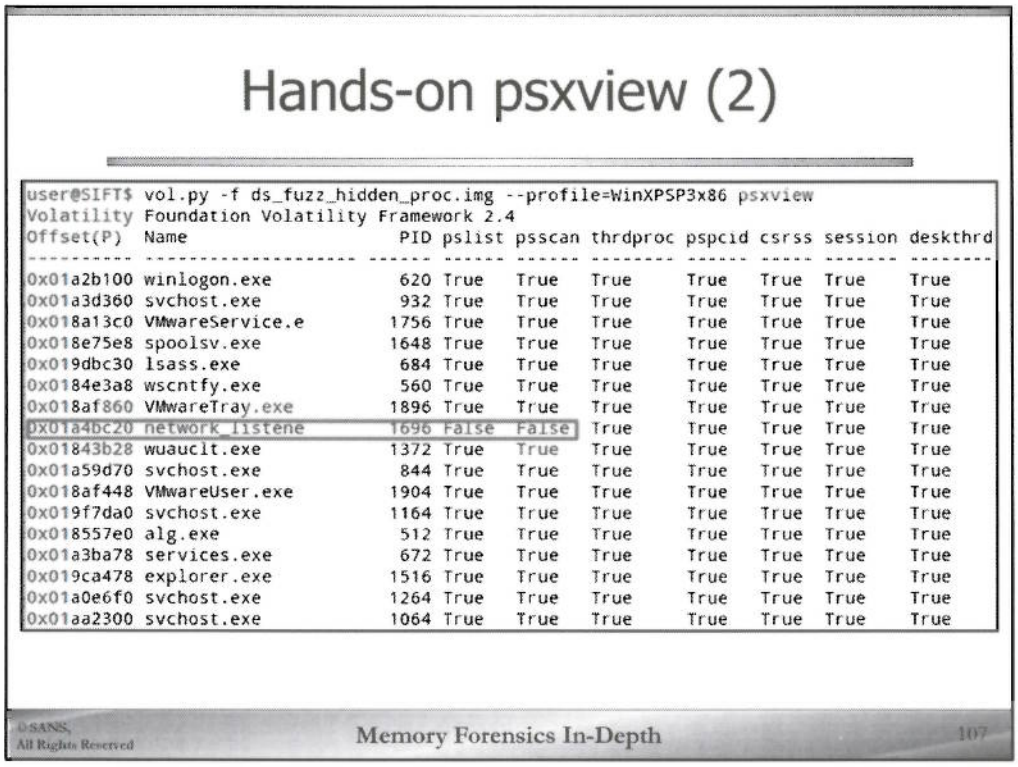


Let's try out psxview by using our old friend the xp-laptop memory image. Here's the syntax to execute the plugin:

user@SIFT$ vol.py -f Win7x64.vmem --profile=Win7SP1x64 psxview

You should get the following output:

```
Volatility Foundation Volatility Framework 2.4
Offset(P)          Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd ExitTime
------------------ -------------------- ------ ------ ------ -------- ------ ----- ------- -------- --------
0x000000007dfd7800 winlogon.exe            516 True   True   True     True   True  True    True
0x000000007dfd5700 lsm.exe                 536 True   True   True     True   True  True    False
0x000000007d9d1b30 firefox.exe            3240 True   True   True     True   True  True    True
… output truncated …
```



Now let's run the psxview plugin on a memory image which does have some hidden processes in it. Specifically, the ds_fuzz_hidden_proc.img memory image, which we mentioned when we discussed Direct Kernel Object Manipulation (DKOM). There is a hidden process in this memory image, called network_listene. This process has been munged so that it's not found via traditional brute force means.

We can run the psxview plugin on this memory image like this:

```
user@SIFT$ vol.py -f ds_fuzz_hidden_proc.img --profile=WinXPSP3x86 psxview
```

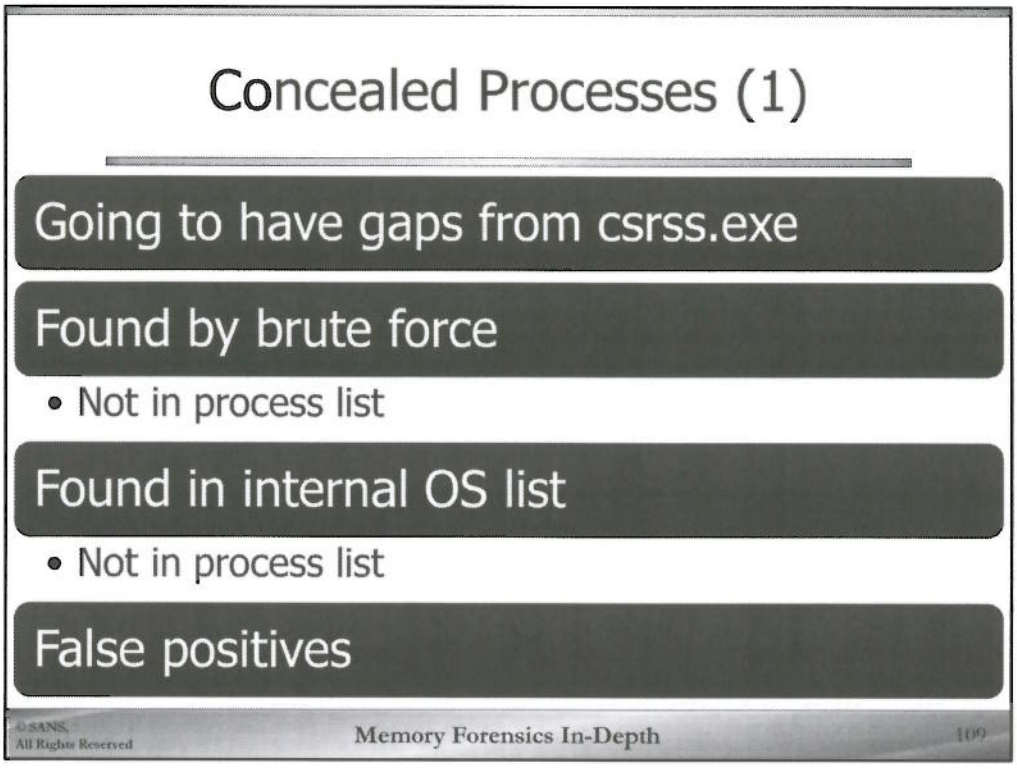
Which generates a lot of output:

```
Volatility Foundation Volatility Framework 2.4
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd ExitTime
---------- -------------------- ------ ------ ------ -------- ------ ----- ------- -------- --------
0x01a2b100 winlogon.exe            620 True   True   True     True   True  True    True
0x01a3d360 svchost.exe             932 True   True   True     True   True  True    True
0x018a13c0 VMwareService.e        1756 True   True   True     True   True  True    True
0x018e75e8 spoolsv.exe            1648 True   True   True     True   True  True    True
0x019dbc30 lsass.exe               684 True   True   True     True   True  True    True
0x0184e3a8 wscntfy.exe             560 True   True   True     True   True  True    True
0x018af860 VMwareTray.exe         1896 True   True   True     True   True  True    True
0x01a4bc20 network_listene        1696 False  False  True     True   True  True    True
0x01843b28 wuauclt.exe            1372 True   True   True     True   True  True    True
```

Some of these processes with zeros in their fields are perfectly legitimate, just terminated or leftovers from a previous boot.

We can run down each of the processes noted by pslist and psscan and resolve them as being either terminated, duplicates or leftovers. For example, in viewing the pslist output, one can clearly identify that PluckUpdater.exe (PID 368) is a zombie process that has not been reaped from the doubly-linked list of processes based on the presence of an Exit Time . And there are several duplicate EPROCESS structures identified in the psxview output, for example, mqtgsvc.exe (PID 712) which is an active process with three duplicate EPROCESS structures enumerated by psxview.

But one process does remain and cannot be ruled out due to zombie, duplicate or previous boot data reduction! The network_listene process, pid 1696, shows up clear as day in the psxview output. We wouldn't have seen it in either the pstree or psscan plugins. That process was protected via DKOM and thus made more difficult to see. But we found it!



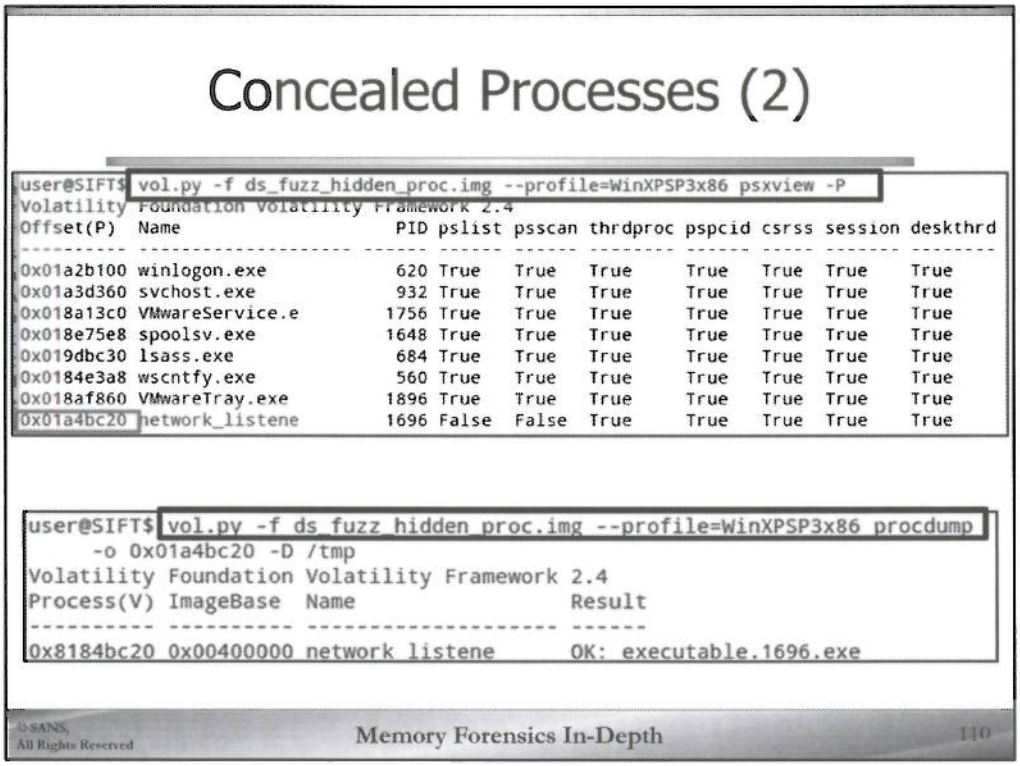
Looking at the output of the psxview plugin can be more of an art than a science.

First, remember that csrss.exe isn't going to have any entries from processes which started before it. You will see "holes" in its entries.

Second, look for any process which was found by a brute force method but was not in the list of known processes. They are good candidates for hidden processes.

Third, look for processes which are in one or more of the operating system's internal lists of processes.

In all of these cases there can certainly be false positives. In general, if you see a process which was only found by one detection method, has no name, and a wacky PID, it's most likely a false positive. That is, random data which tripped the tool's methods for detecting processes.

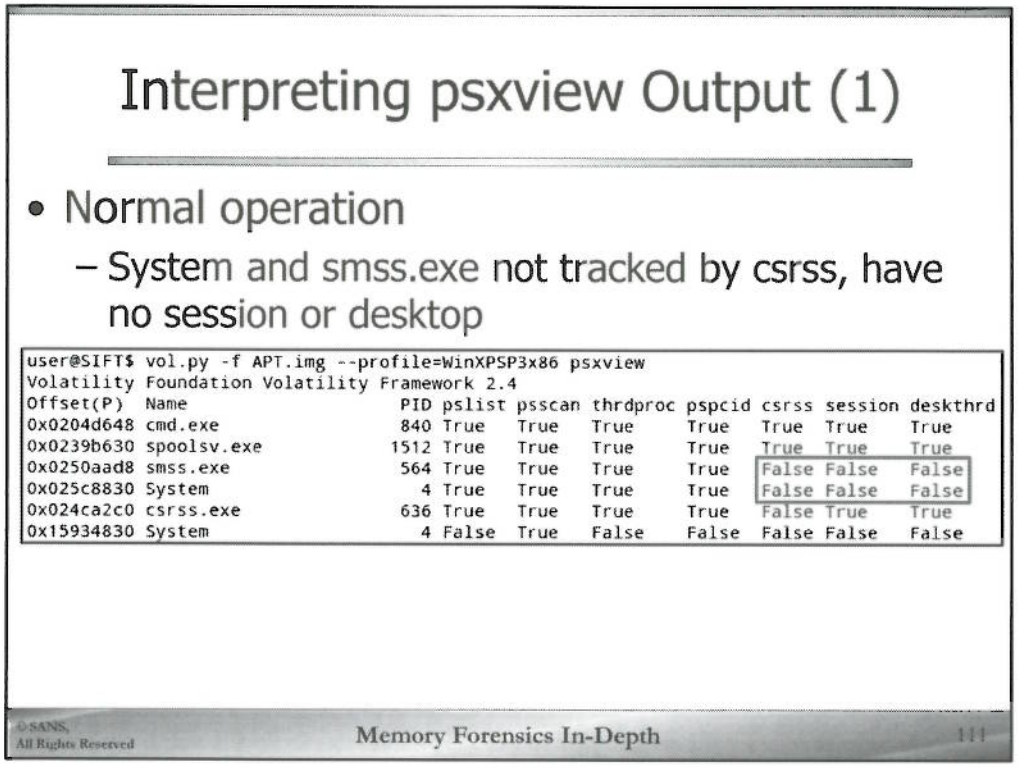


For any processes, when you are unsure, dump out a sample and examine. Don't assume!

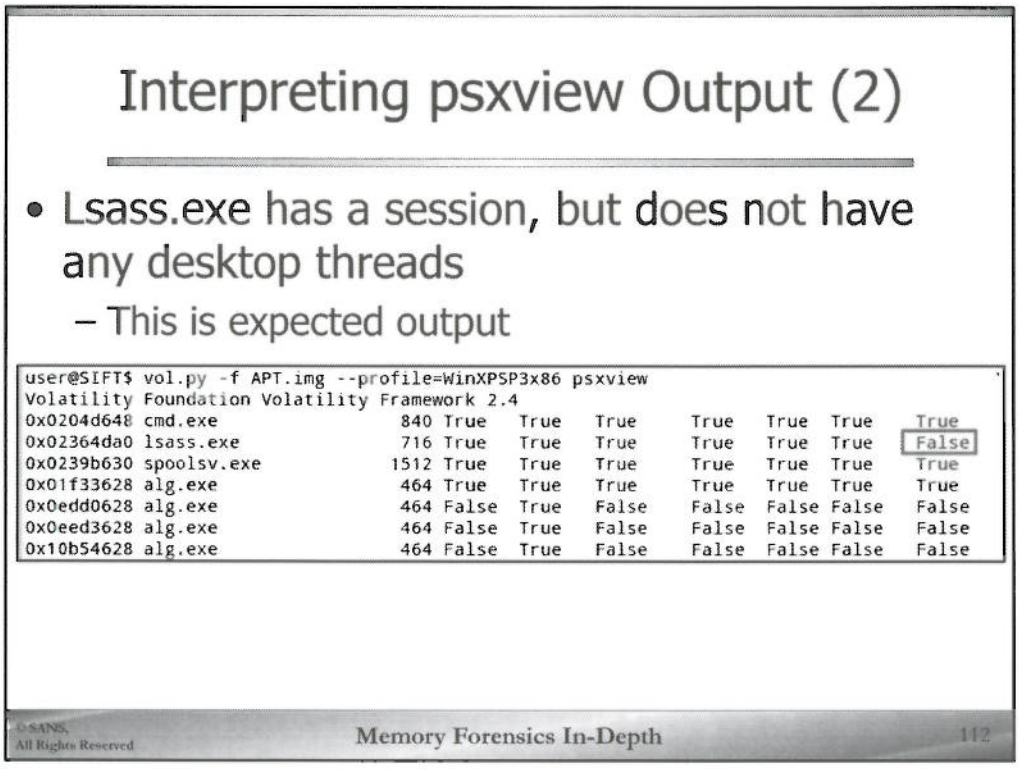
To dump out a sample of a hidden process, you have to use the physical offset of the process' EPROCESS block. You can't use the PID. The process dumping tools generally use the operating system's list of processes. But by definition a hidden process isn't in the list of processes! So we need that physical offset.

You can get the physical offset of the EPROCESS block, which can then be passed to procexedump or procmemdump, using the -p flag.

```
$ vol.py -f /cases/ds_fuzz_hidden_proc.img  --profile=WinXPSP3x86 psxview -P
Volatility Foundation Volatility Framework 2.4
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session
deskthrd ExitTime
---------- -------------------- ------ ------ ------ -------- ------ ----- ------- ------
-- --------
0x01a4bc20 network_listene        1696 False  False  True     True   True  True    True

user@SIFT$ vol.py -f ds_fuzz_hidden_proc.img --profile=WinXPSP3x86 procdump -o
0x01a4bc20 -D /tmp
Volatility Foundation Volatility Framework 2.4
Process(V) ImageBase  Name                 Result
---------- ---------- -------------------- ------
0x8184bc20 0x00400000 network_listene      OK: executable.1696.exe
```



The output shown about is completely normal. The System process starts first, followed by smss.exe, followed by csrss.exe. It stands to reason that smss.exe and System won't be tracked by a process that starts after they do.



Note that although lsass.exe is tracked by csrss.exe and has an associated session (the system session), it does not have any desktop threads. There is nothing funny going on here, this is completely expected.

Interpreting psxview Output (3)

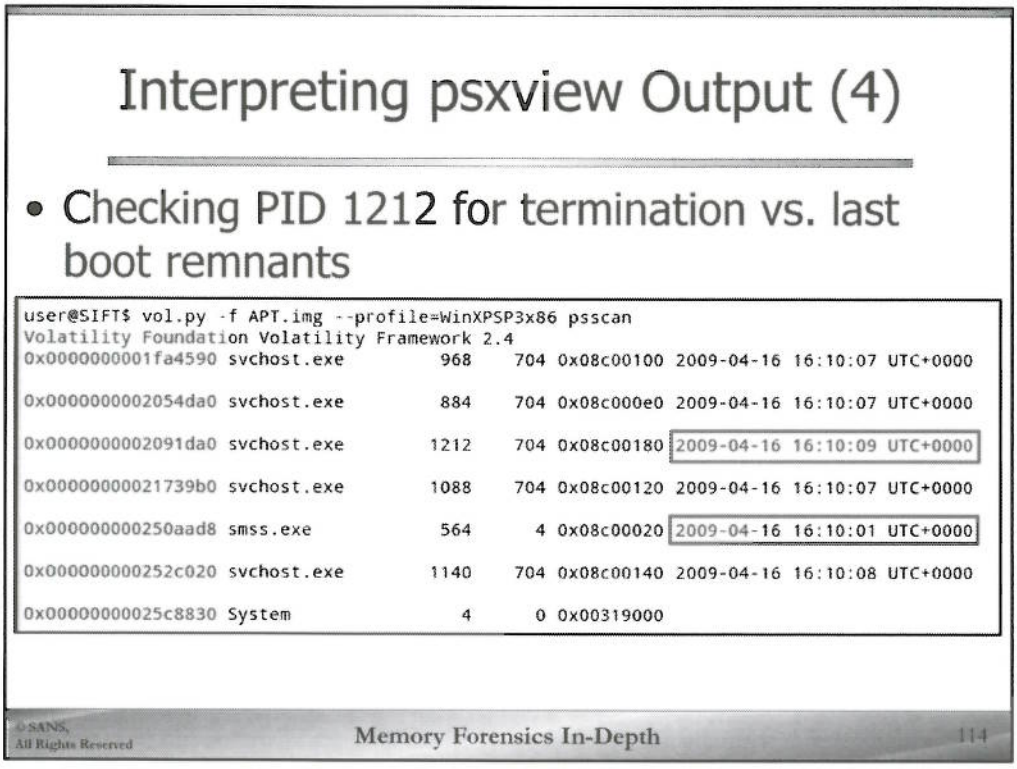
- Entries that are only present in psscan may be exited or from a previous boot
  - Run psscan to confirm

```
user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 psxview | egrep "False\s+True\s+False|pslist"
Volatility Foundation Volatility Framework 2.4
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd Exit
0x03178220 wscntfy.exe            1260 False  True   False    False  False False   False
0x0c605020 svchost.exe            1140 False  True   False    False  False False   False
0x0ad69da0 iexplore.exe            796 False  True   False    False  False False   False
0x0edd0628 alg.exe                 464 False  True   False    False  False False   False
0x032b3da0 svchost.exe             884 False  True   False    False  False False   False
0x0eed3628 alg.exe                 464 False  True   False    False  False False   False
0x10b54628 alg.exe                 464 False  True   False    False  False False   False
0x15934830 System                    4 False  True   False    False  False False   False
0x1b217da0 iexplore.exe            796 False  True   False    False  False False   False
```


Unfortunately, in the current version of volatility, there is no field in the psxview output that denotes whether a process has exited. Processes that appear only in the psscan output may either be exited processes or they may be present from a previous boot.

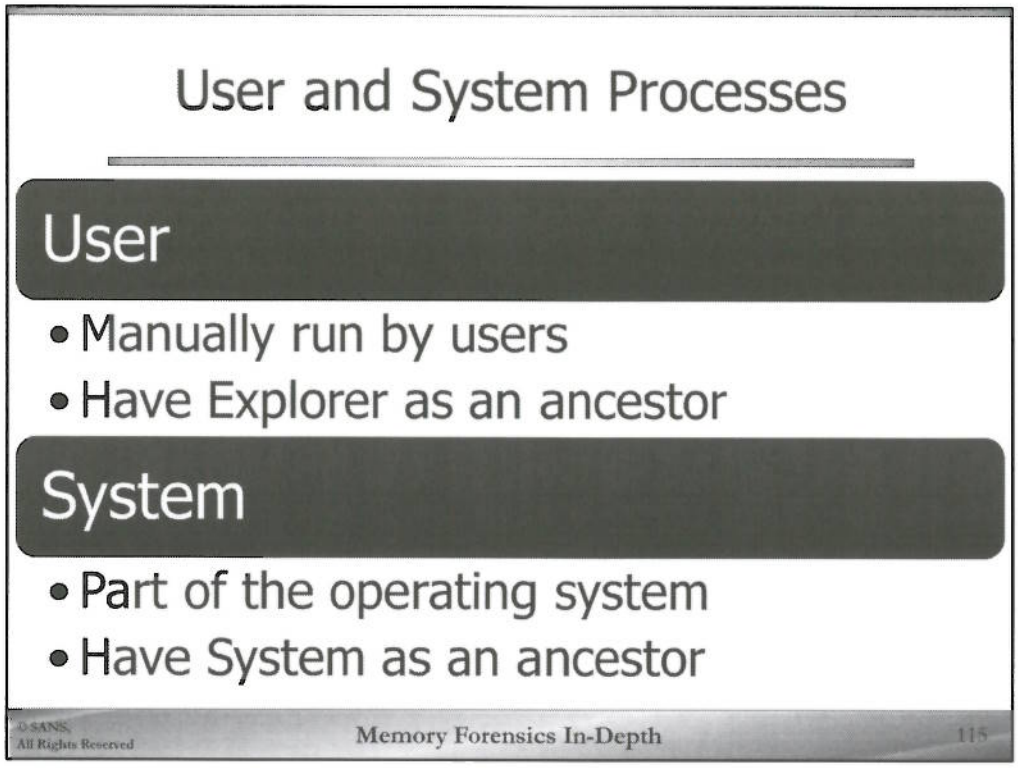
Note that to find processes that only exist in psscan, a good quick method is to pipe to egrep "False\s+True\s+False|pslist" – this gives both the column headers and the processes only in psscan.

Is the svchost.exe process with PID 1140 from a previous boot or has it exited on the current boot? Let's run psscan to find out.

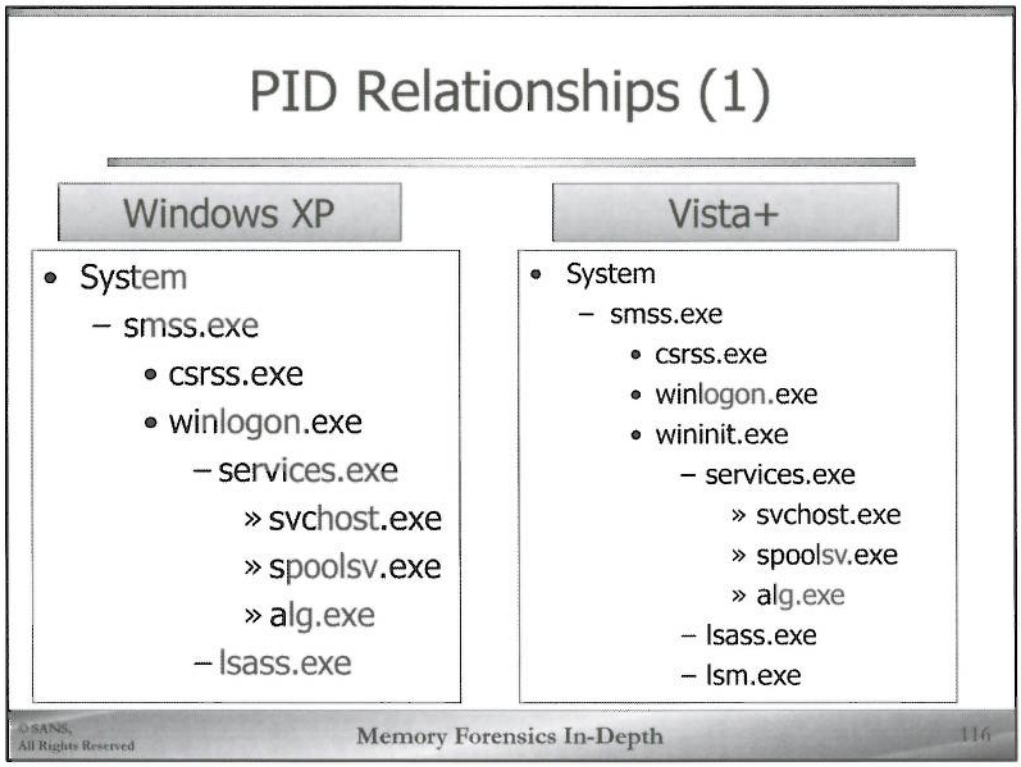


Examining the output of psscan and looking at smss.exe, we can see that the boot time for the machine was '2009-04-16 16:10:01 UTC+0000'. Because our target process, PID 1212, has a create time of '2009-04-16 16:10:09 UTC+0000' we can conclude that the process was created on the current boot and is not a remnant from a previous boot.

Also note that the System process (PID 4) has no time created. This is normal for the System process.



In this section we are going to talk about processes being either "user" or "system" processes. These are not precise definitions, but instead reflect a difference between operating system components and programs run by and for users. For example, Microsoft Word would be a user process. But svchost.exe would be a system process. Both of these processes run in userspace, the lower 2GB of the virtual memory model. But we want to draw a distinction between programs which a user would normally start and those which happen behind the scenes.



On an XP Windows system, you should see a set of processes as shown in left list above:

System
  smss.exe
    csrss.exe
    winlogon.exe
      services.exe
        svchost.exe
        spoolsv.exe
        alg.exe
    lsass.exe

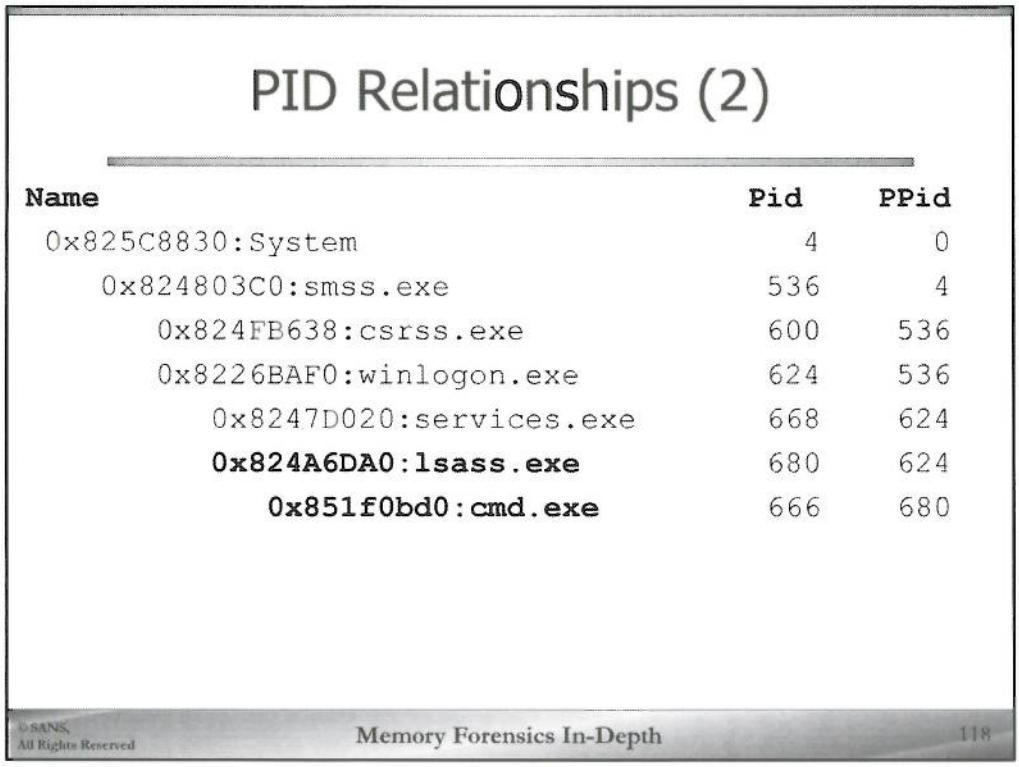
This is the tree of System processes, or operating system components specific to Windows XP. Note that they all have "System" as a common ancestor.

For Vista and beyond, the sequence of system processes is slightly difference (see right list), with wininit.exe launching services, lsass and lsm.

System
 smss.exe
   csrss.exe
   winlogon.exe
   wininit.exe
    services.exe
      svchost.exe
      spoolsv.exe
      alg.exe
    lsass.exe
    lsm.exe

These are the names of legitimate system processes. Now, finding those names does not mean these processes haven't been compromised, of course. But we're more interested in finding parent child relationships which break these structures.

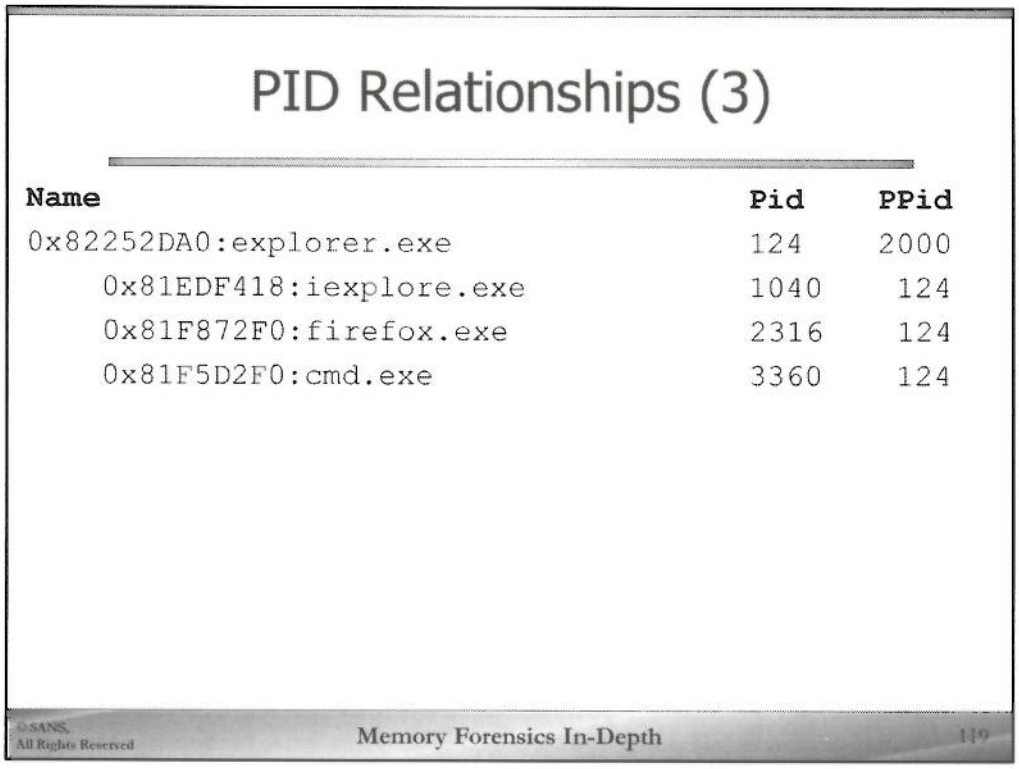
On any system there will always be other processes in here too. Services.exe, for example, will often be the parent of a wide variety of drivers, services, and other low-level programs. You have to do some searching and check some references to determine if the program name you're looking at is legitimate or not. The Windows Internals book, as well as MSDN online library, are excellent resources for this type of research. If you are uncertain of what a process is, for example, PluckSvr.exe, as it appears in the pslist output of the ? You've seen it several times now in the XP laptop image. As it turns out, it's part of an RSS program which worked in the background to gather RSS feeds. (That's also why you found RSS traffic in the XP laptop memory, too!)



The one thing you should never see in the tree of System processes is any user process. If you see a user program in the tree of system processes, it generally means the system has been compromised. Programs like svchost and lsass, for example, shouldn't be running browsers, command shells, or any other program normally run by an end user (like an intruder).

There are notable exceptions to this rule. For example, some VMware systems have a process in the System tree which spawns a command shell. That's perfectly normal.

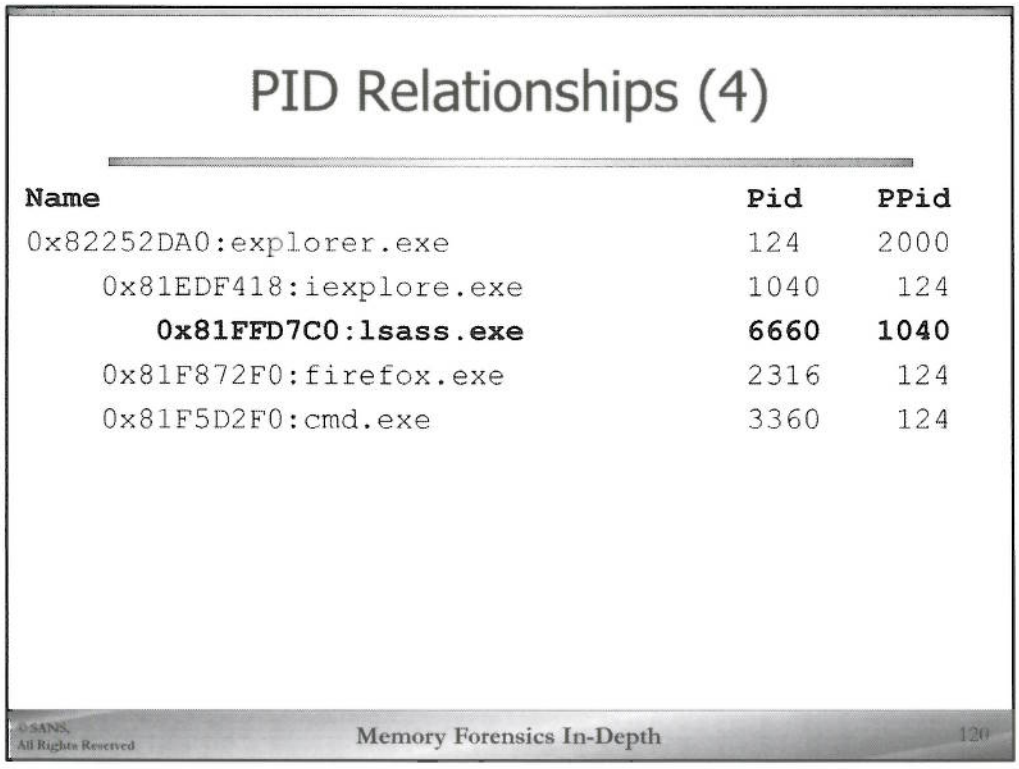
But for the most part, if you see a user process being spawned by a system process, that's a red flag that something is wrong.



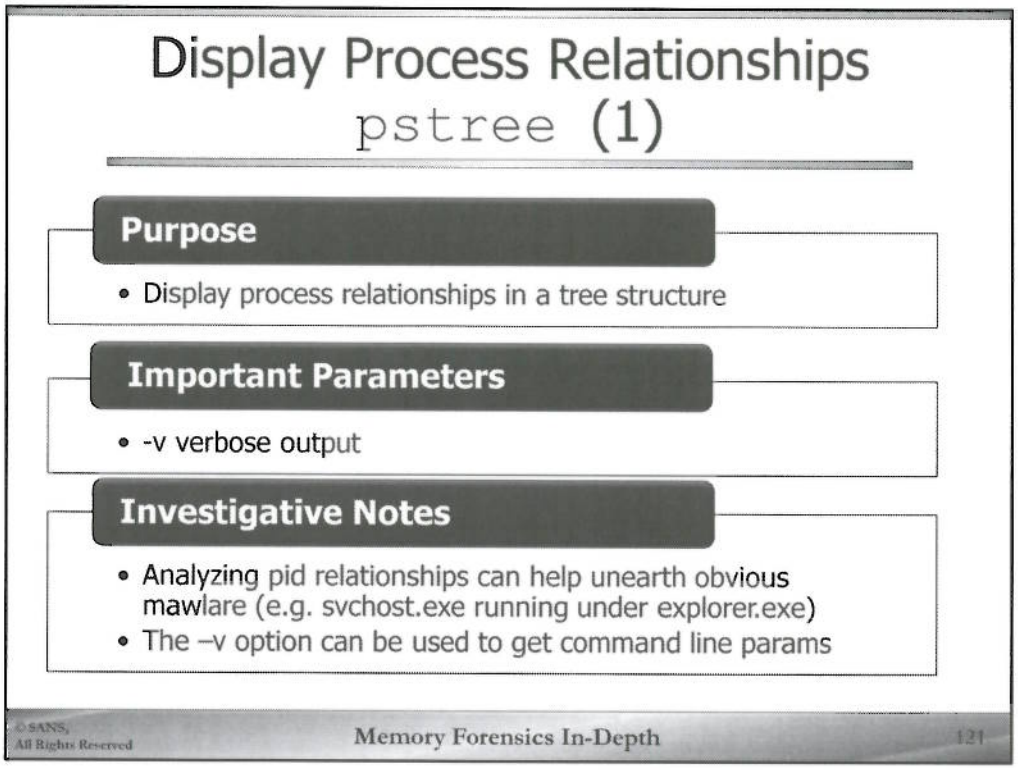
User processes should also follow their own standard of behavior. User processes should not spawn system processes.

We can see the user processes running on a system by looking for the children of the Explorer process. As shown above, here's Explorer running some legitimate programs.

When a user logs in, the winlogon.exe spawns a process userinit.exe, which is responsible for setting up the user's environment. If you capture a memory image immediately after a user logs in, you should be able to see userinit in the list of processes, but because it terminates right after it's finished, we normally don't see it. Because userinit no longer exists, explorer.exe, the main process started by userinit, appears to be an orphaned process.



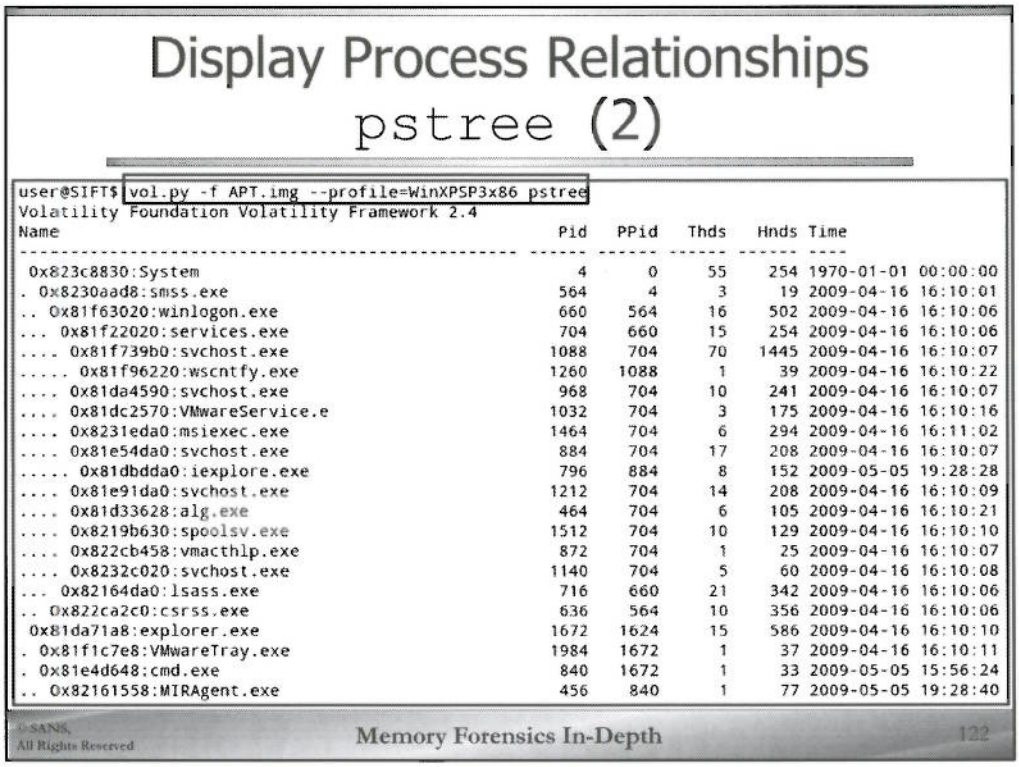
Among the children of explorer.exe you should see user processes—things users would run: browsers, editors, games, you name it. This is where they belong. It's possible to find legitimate orphan processes for a variety of reasons. Some programs have a launcher which sets up an environment and then terminates when the main program launches.



The pstree plugin is used to display process relationships graphically in a tree format. Sure you could do this yourself by analyzing parent/child relationships, but the tree output really helps with this. Note that there is no requirement for a process in Windows to have a parent process ID (unlike in Linux). Therefore, it is normal to see processes in the pstree output that are "orphaned" or have no parent.

The –v option adds three new fields to the output: audit, cmd, and path. Audit is the path to the binary starting with the actual "\Device" nomenclature. Cmd is the full command line. This includes arguments but may not include the path (depending on how the process was executed). Path shows the full path to the executable, starting from the logical drive letter, e.g. "C:".

The pstree plugin is implemented in volatility/plugins/pstree.py.



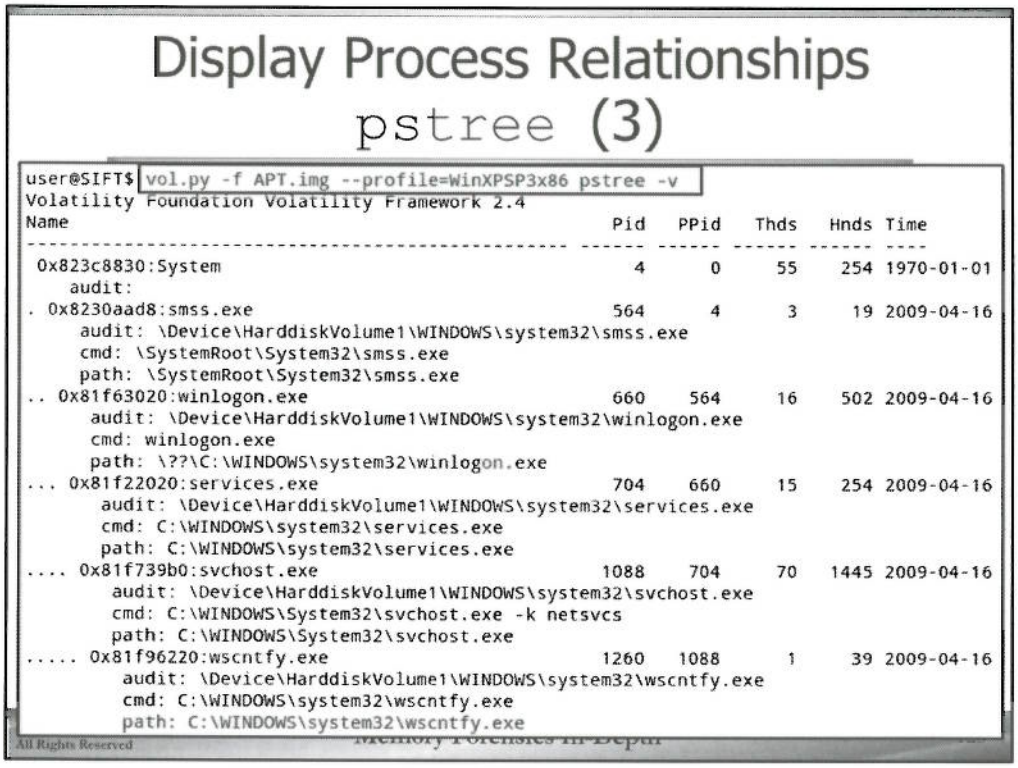
This slide shows an example of the pstree output. Note the periods on the left-hand side. These periods denote the indentation level. For instance, this output makes it obvious that the smss.exe process is the parent of the winlogon.exe process. One quick check you want to make is to ensure that all svchost.exe processes have services.exe as a parent process.

user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 pstree

Volatility Foundation Volatility Framework 2.4

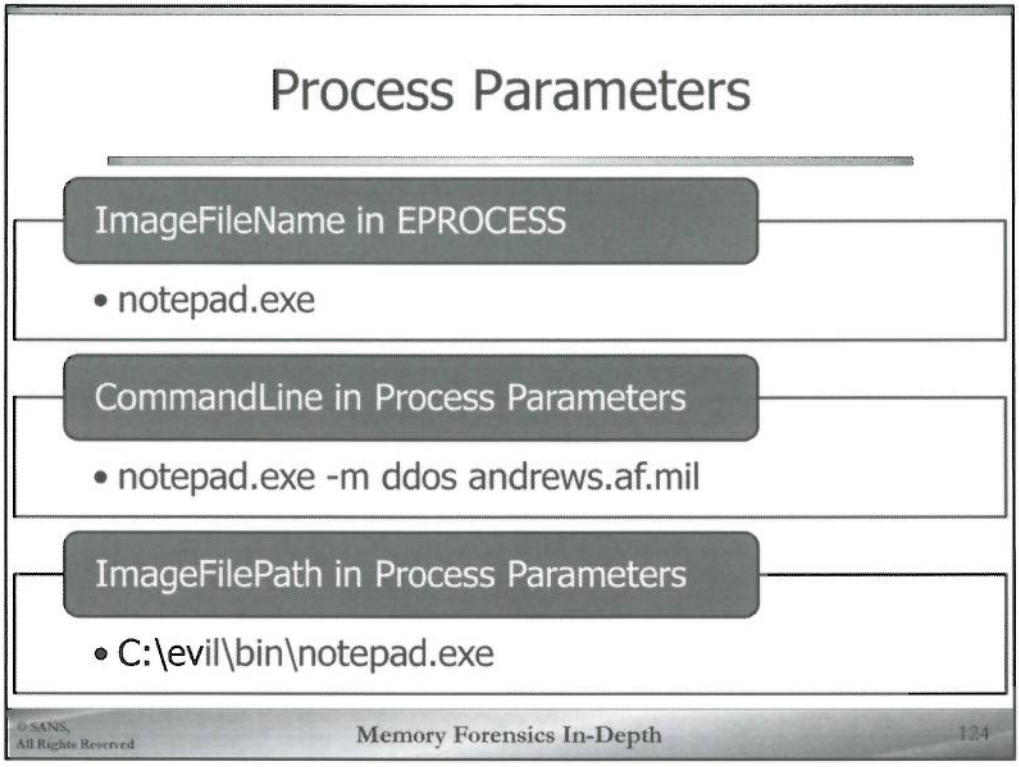
Name	Pid	PPid	Thds	Hnds	Time
0x823c8830:System	4	0	55	254	1970-01-01 00:00:00 UTC+0000
. 0x8230aad8:smss.exe	564	4	3	19	2009-04-16 16:10:01 UTC+0000
.. 0x81f63020:winlogon.exe	660	564	16	502	2009-04-16 16:10:06 UTC+0000
... 0x81f22020:services.exe	704	660	15	254	2009-04-16 16:10:06 UTC+0000
.... 0x81f739b0:svchost.exe	1088	704	70	1445	2009-04-16 16:10:07 UTC+0000
..... 0x81f96220:wscntfy.exe	1260	1088	1	39	2009-04-16 16:10:22 UTC+0000
.... 0x81da4590:svchost.exe	968	704	10	241	2009-04-16 16:10:07 UTC+0000
.... 0x81dc2570:VMwareService.e	1032	704	3	175	2009-04-16 16:10:16 UTC+0000

… output truncated ….



This slide shows the output of the pstree plugin using the –v option. Again, note the extra fields: audit, cmd, and path. Recall that these are defined as follows: Audit is the path to the binary starting with the actual "\Device" nomenclature. Cmd is the full command line. This includes arguments but may not include the path (depending on how the process was executed). Path shows the full path to the executable, starting from the logical drive letter, e.g. "C:".

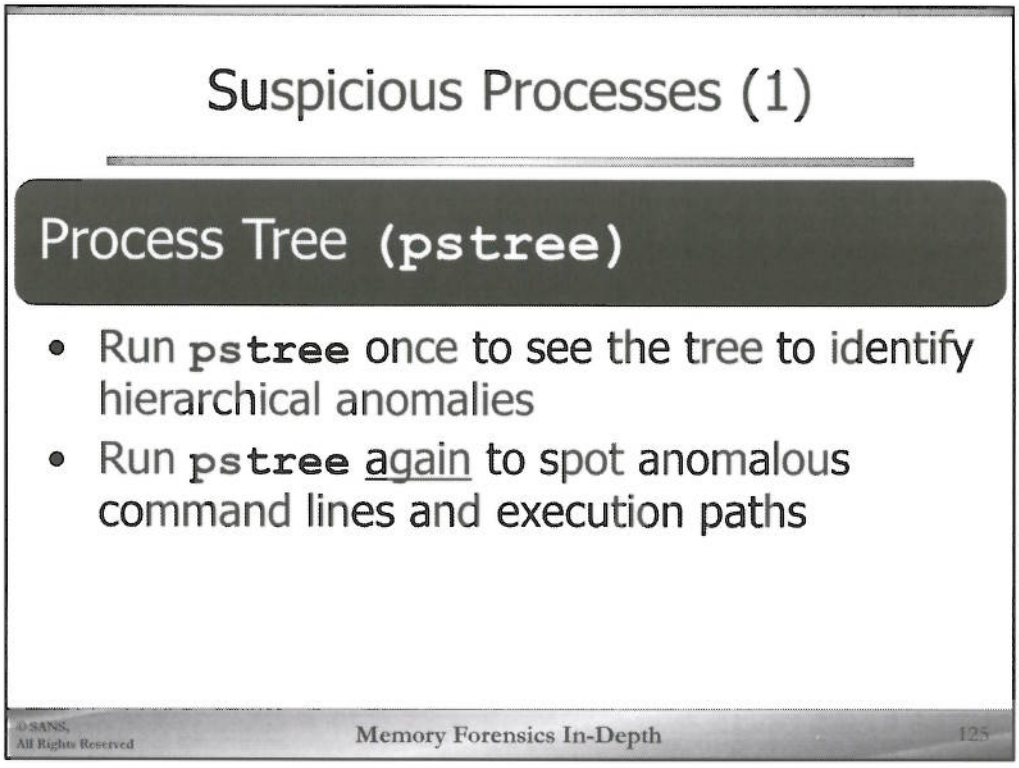
```
user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 pstree -v
Volatility Foundation Volatility Framework 2.4
Name                                  Pid   PPid   Thds   Hnds Time
-------------------------------------------------- ------ ------ ------ ------ ----
 0x823c8830:System                      4      0     55    254 1970-01-01 00:00:00 UTC+0000
   audit:
. 0x8230aad8:smss.exe                 564      4      3     19 2009-04-16 16:10:01 UTC+0000
   audit: \Device\HarddiskVolume1\WINDOWS\system32\smss.exe
   cmd: \SystemRoot\System32\smss.exe
   path: \SystemRoot\System32\smss.exe
.. 0x81f63020:winlogon.exe          660    564     16    502 2009-04-16 16:10:06 UTC+0000
    audit: \Device\HarddiskVolume1\WINDOWS\system32\winlogon.exe
    cmd: winlogon.exe
    path: \??\C:\WINDOWS\system32\winlogon.exe
... output truncated ...
```



We can follow a pointer from the PEB to the process parameters. The process parameters contain all kinds of interesting data. There are Unicode strings for the command line used to start the process, the image path name, the current working directory, and the current window title. The image path name is the full name of the executable for this process, which is not necessarily in the command line. For example, if the user typed "notepad" on the command line, the image path name would be something like "C:\Windows\Notepad.exe". The current window title may be blank for processes which don't display a window.

The process name we obtained from the EPROCESS block was not only limited to 16 characters, but provided very little information about the process in question. A process which sounds innocuous, like notepad.exe, is not necessarily trustworthy. Seeing the command line, which may include the path, but certainly includes the command line arguments, can shed more light on the situation. There are no hard and fast rules about what makes a command line suspicious. Sometimes searching the Internet for information about a process can help you gather more information. Sometimes, as with the example above, you just kind of have a sense that it might not be on the up and up. The ImageFilePath helps even further as it provides the exact location of the executable in question, even if it wasn't specified on the command line.

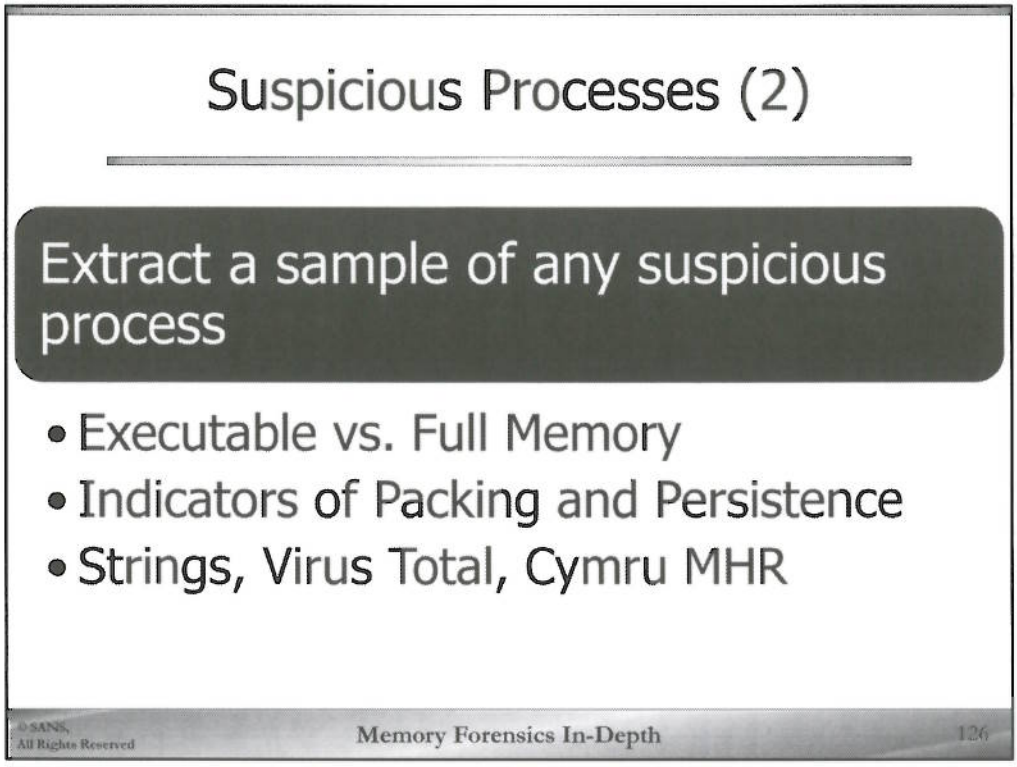
The process parameters structure is a type called "RTL_USER_PROCESS_PARAMETERS", which is documented on Microsoft's web site at http://msdn.microsoft.com/en-us/library/windows/desktop/aa813741(v=vs.85).aspx. Again, the vast majority of Microsoft's documentation is blank.



When looking for suspicious processes, we recommend starting with the process tree. With the Volatility framework you can see it using the pstree plugin. The process tree lets you see several details. First, we can see the PID relationships we talked about. Look for user processes being spawned by system processes. In particular, look for "cmd.exe" being spawned by svchost.exe or services.exe. It's a common technique of malware. Of course, not all of those are malware. Sometimes it's legitimate.

Running the pstree plugin a second time with the -v flag enabled lets you see the command lines and original executable location for each process. Examining these fields is more of an art than a science, but here are some tips. Obviously everything here will be superseded by what's normal for your enterprise.

- Sane Program Names - Programmers choose "logical" names for their programs. Logical is in quotes because what's logical to them may not make sense to you, of course. But in general they are human readable names. Short acronyms are common, but lookout for a truly random series of letter and numbers. That's a good indicator of malware.
- Ending in .exe - Legitimate programs have a valid extension. Malware often leaves a blank extension.
- More than one or two characters in the filename - This is a sure sign of malware. Legitimate programs have a name, not just an ID number.
- Spelling mistakes - Malware authors may not be native English speakers.
- Correct file locations - Which programs start in C:\Windows and which start from C:\Windows\System32. When in doubt, look it up on a known good system. But in general programs start from the Windows directory or from Program Files. Finding an executable starting from any other directory is a sign of trouble.
- Sane command line arguments



When you find a process which piques your interest for any reason, go ahead and extract a sample of it. With the Volatility Framework you can do this with the procexedump and procmemdump plugins. Use the latter especially if you suspect the program has been packed, encrypted, or obfuscated in any way. Remember that procmemdump dumps out the entire memory of the executable, which could include the unpacked code you are most interested in.

Once you have that sample, you can examine it however you'd like. A good basic starting point is to look at the strings in an executable. Don't forget about Unicode strings, too! On the SIFT you can search for regular strings like this:

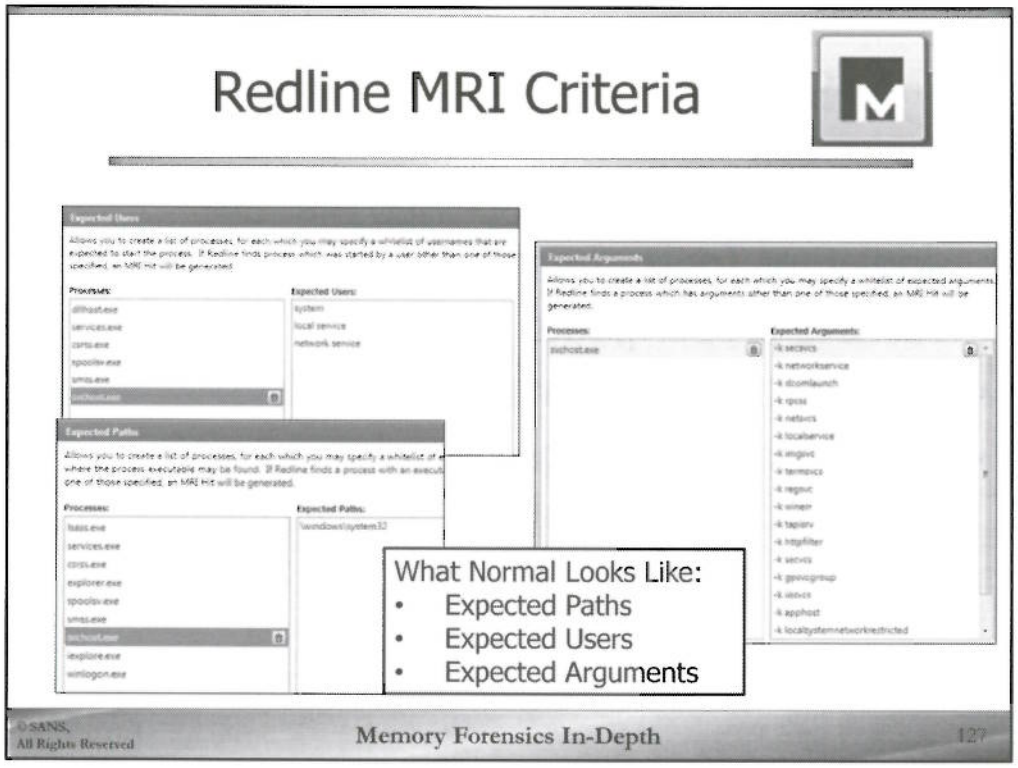
$ strings -a [FILE]

And for Unicode strings:

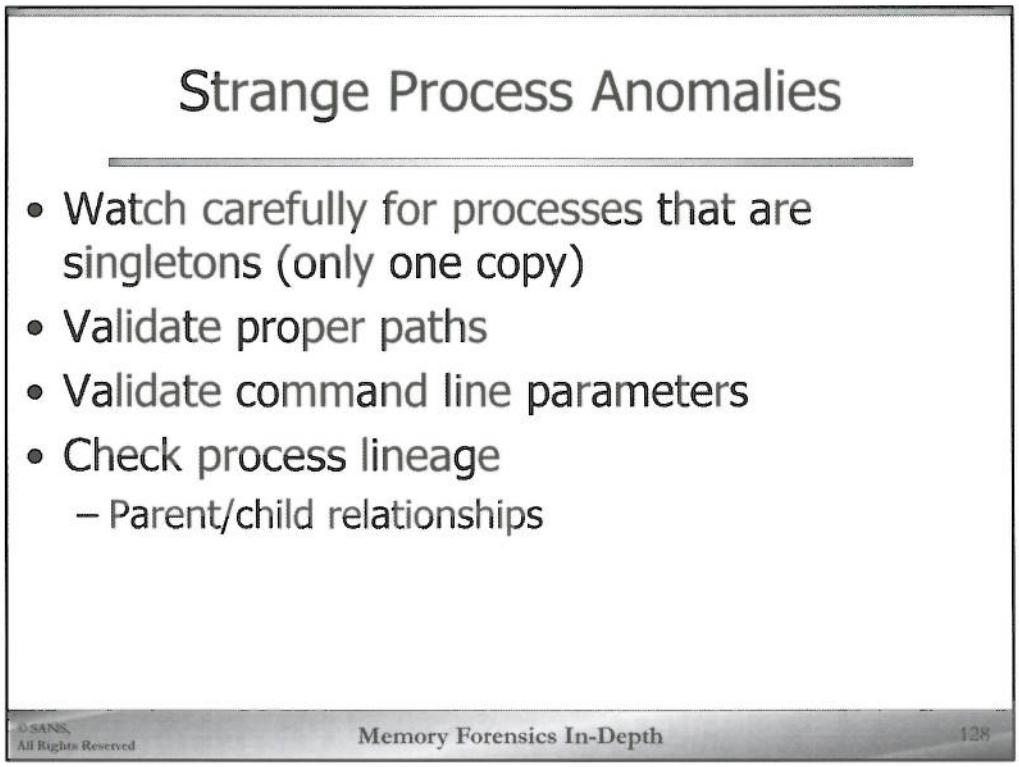
$ strings -a -e l [FILE]

Be on the lookout for any indicators of packing. These usually are found at the start of the file with the section names. If you see "UPX" anywhere, suspect a packed program! Most malware will contain a persistence mechanism of some kind--a registry key, startup item, etc. Installing that mechanism should be a part of the program. Be on the lookout for such mechanisms.

You can always submit the file, or even just its hash, to services like Virus Total or the Cymru Malware Hash Registry (MHR), http://www.team-cymru.org/Services/MHR/.



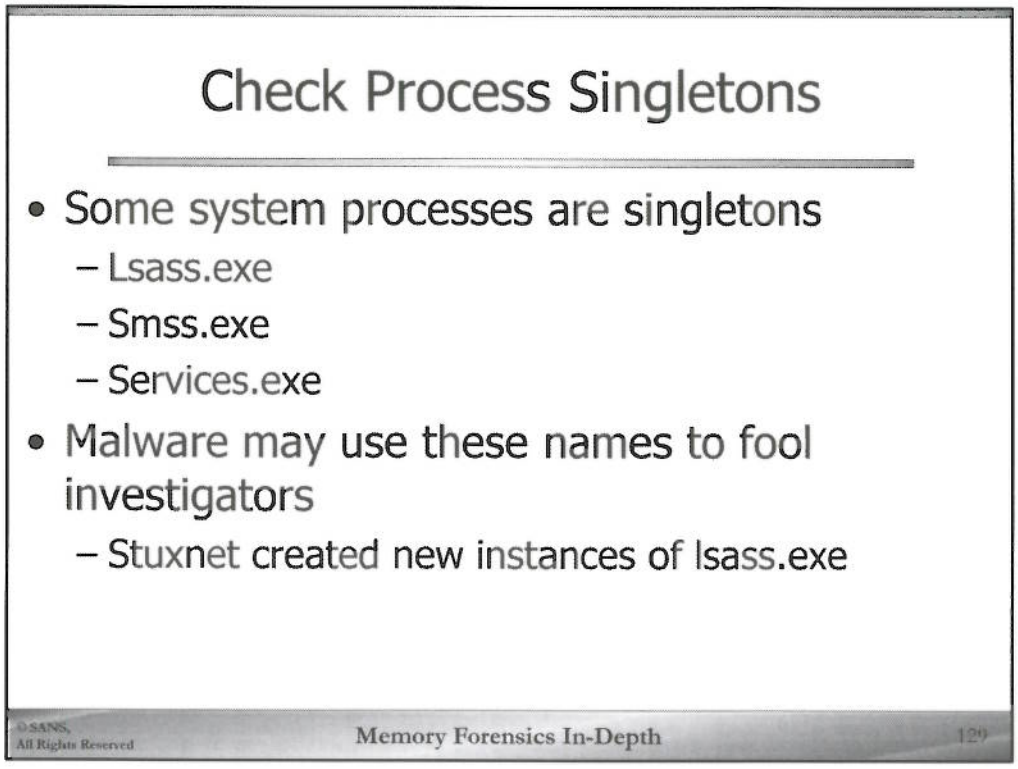
Mandiant's Redline Memory and Audit Analysis tool makes use of its own Malware Risk Index that it relies on to evaluate processes for indications of malware. It is based on, among other things, specific process criteria that include Process Owners and Process Commandline and Arguments. It relies on a database of conditions that it expects to exist for well known Windows processes, which coincidentally are common targets for malware attempting to "blend in" with normal processes. By evaluating whether svchost.exe is running from the expected directory, Redline is able to assign a higher risk index score to svchost processes that deviate from normal (which would be \Windows\system32. Investigators can make use of this default database to become familiar with process evaluation, and may also add to the database with addition processes and their expected details.



There are a number of ways to get quick wins and find malicious processes. The four easiest ways are to:

- Watch carefully for processes that are singletons (only one copy)
- Validate proper paths
- Validate command line parameters
- Check process lineage

We'll discuss each of these in an upcoming slide.



There are some processes that are singletons, meaning there should never be more than one copy of the process executing at any given time. Some good examples of singleton processes are lsass.exe, smss.exe, and services.exe. Stuxnet is one example of malware that created additional copies of singletons. It created additional copies of the lsass.exe process. Investigators who know which processes should be singletons immediately knew something was wrong on infected systems.

A partial list of processes that should be singletons are:

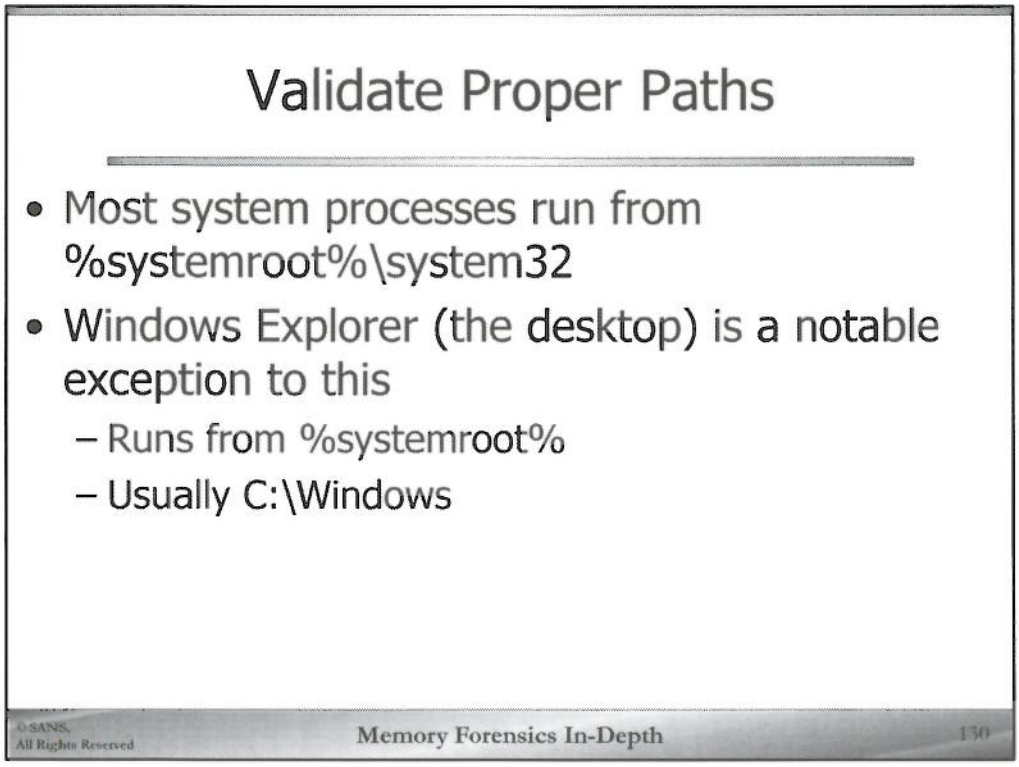
Lsass.exe

Smss.exe

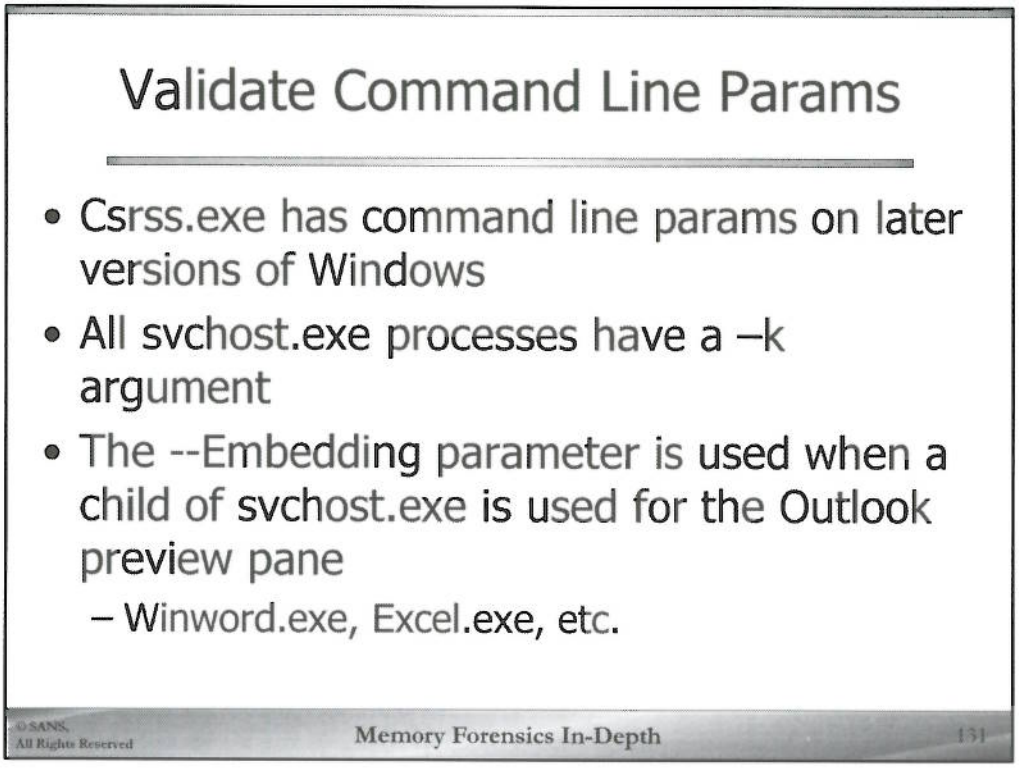
Services.exe

System

Interestingly, csrss.exe (on later versions of Windows) is not a singleton. Additional copies are created for new user sessions.



Ensure that processes are running from the expected paths. Attackers want to blend in, and how better to do that than to run malware using the name of a legitimate process. To detect this, you should validate that the process is running from the expected path. Most often, this path will be %systemroot%\system32 for most system processes. One notable exception to this rule is Explorer.exe (the user's desktop). This process runs from %systemroot%. On multiuser systems, expect to see one explorer.exe process per user logon. Multiuser logons to a single machine are more common with fast user switching and RDP servers.



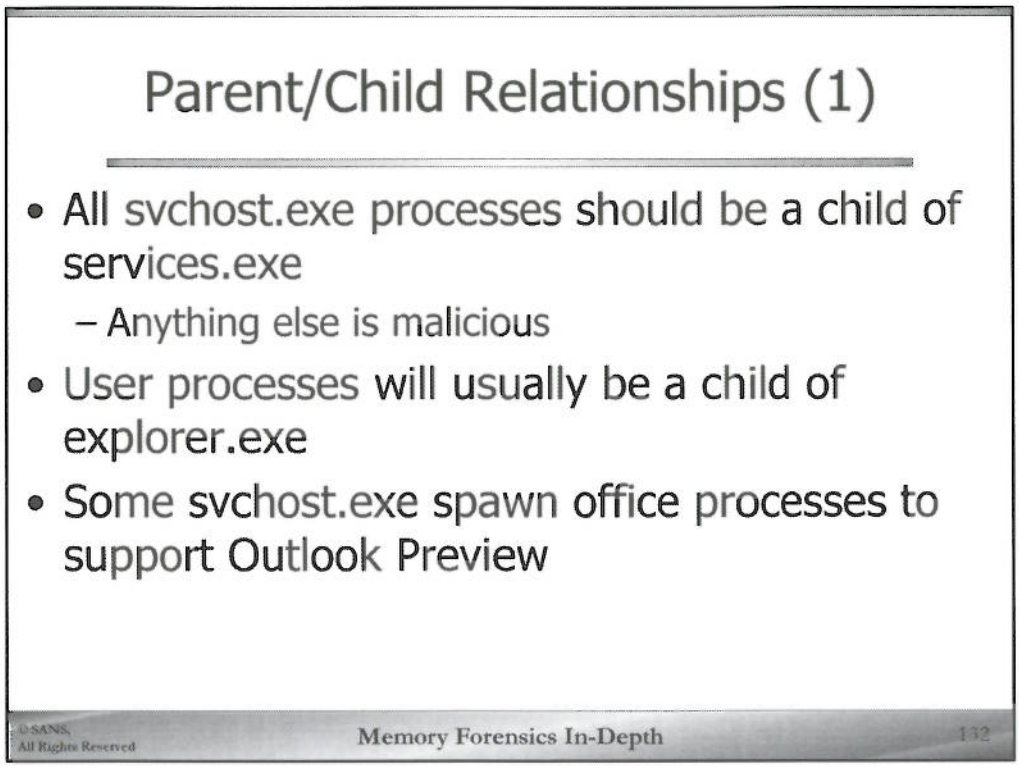
You should examine command line parameters and validate that they make sense as well. Many system processes don't have command line parameters. However, others should always have command line parameters. Closely examine command line parameters of processes. Do any processes have arguments that make it look like a netcat process in disguise? For example "svchost.exe –l –p 4444".

Speaking of svchost.exe, all svchost.exe processes should have a –k parameter. This parameter tells services.exe which service group (and service DLLs) should run under the svchost process.

The csrss.exe process on later versions of Windows will also have command line parameters. For example:

%SystemRoot%\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,20480,768 Windows=On SubSystemType=Windows ServerDll=basesrv,1 ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=winsrv:ConServerDllInitialization,2 ServerDll=sxssrv,4 ProfileControl=Off MaxRequestThreads=16

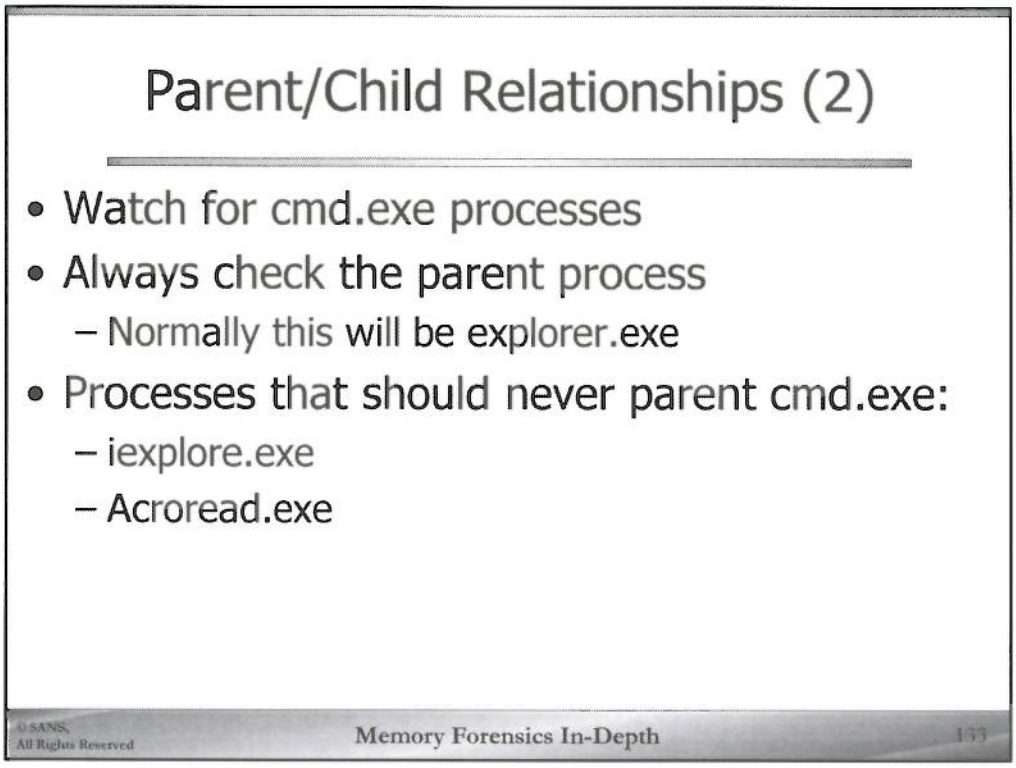
Stepping through each of these parameters is beyond the scope of this course, however be aware that finding a csrss.exe process with no command line arguments on Windows 7 or later is a giant red flag.



Check your process parent/child PID relationships carefully. These are useful in identifying malware. The most common name for malware is svchost.exe. One thing that most attackers screw up is that they forget to re-parent their fake svchost.exe process to run as a child of services.exe. Every single legitimate instance of svchost.exe that has ever been created has the parent svchost.exe. If you EVER find one that does not have a parent of services.exe, you have malware running on the system. This is true even if the process is running from the proper path (%systemroot%\system32) and has an expected command line parameter (e.g. -k netsvcs).

Most desktop processes should be children of explorer.exe.

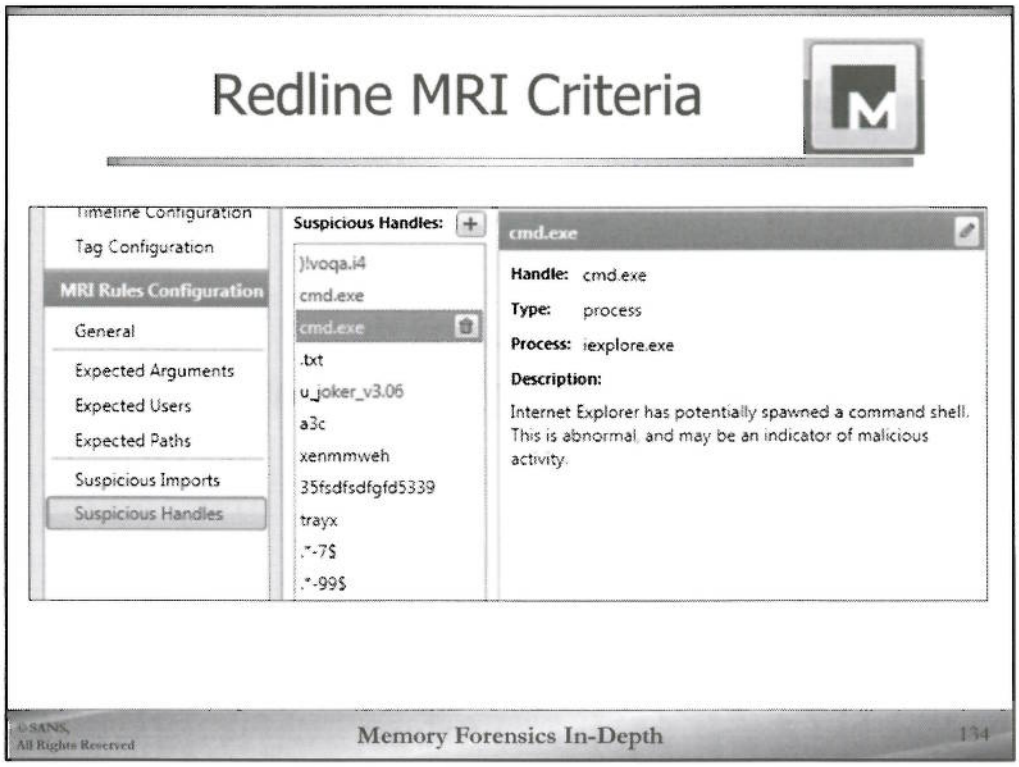
If Outlook is installed, it is not uncommon to see desktop processes running as children of explorer.exe.



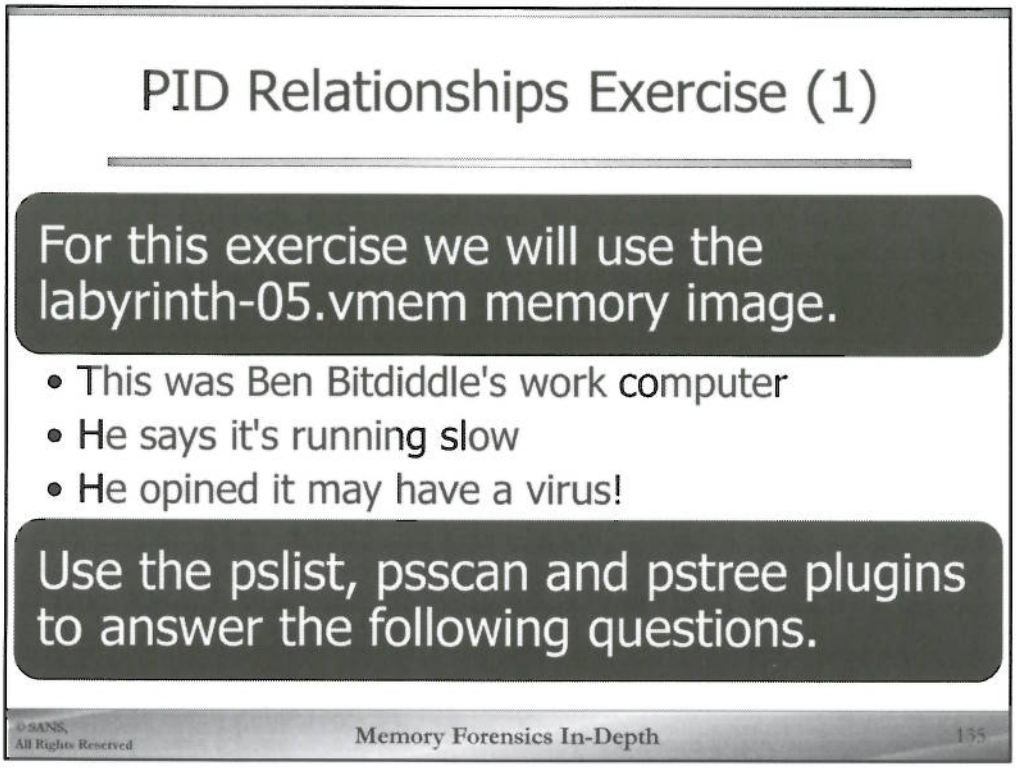
You should always check for cmd.exe processes. When one is found, ensure that the parent process makes sense. Usually the parent process will be explorer.exe. However, just as important are processes that don't make sense as parents of cmd.exe. Is there ever a reason where winword.exe, acroread.exe, or iexplore.exe should spawn a cmd.exe? It's doubtful that this will be legitimate. Most often, this is a sign of an exploited user service.

Once you find cmd.exe processes, use the cmdscan and consoles plugins (discussed later) to gain greater detail. You may also find out what the cmd.exe is doing based on any child processes it has.

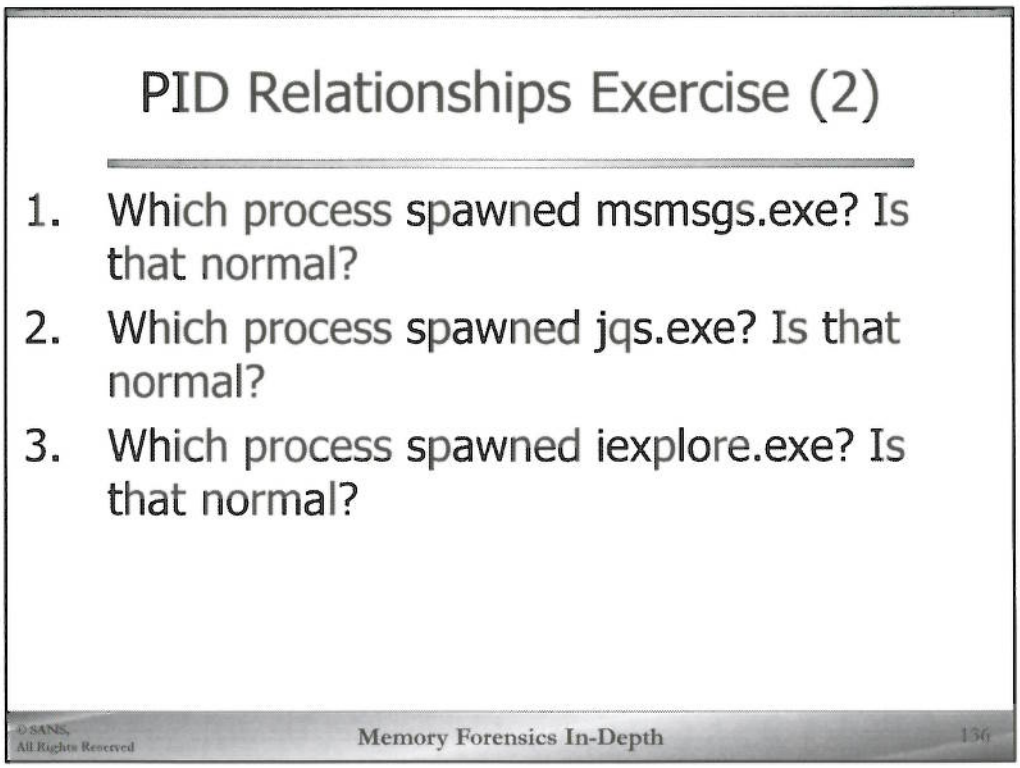
Note that the Outlook preview pane creates child processes under svchost.exe for viewing word documents, excel documents, and more. The svchost.exe will spawn the child process with the --Embedding command line option.



Remember Redline? Redline has a signature that checks for handles to cmd.exe from insider of iexplorer.exe. While the method of enumerating the handle table is not the same as checking parent/child PID relationships, the result is the same. When a process creates a child process, it receives a handle to the child process. This handle is what Redline is looking for to enumerate parent/child relationships.

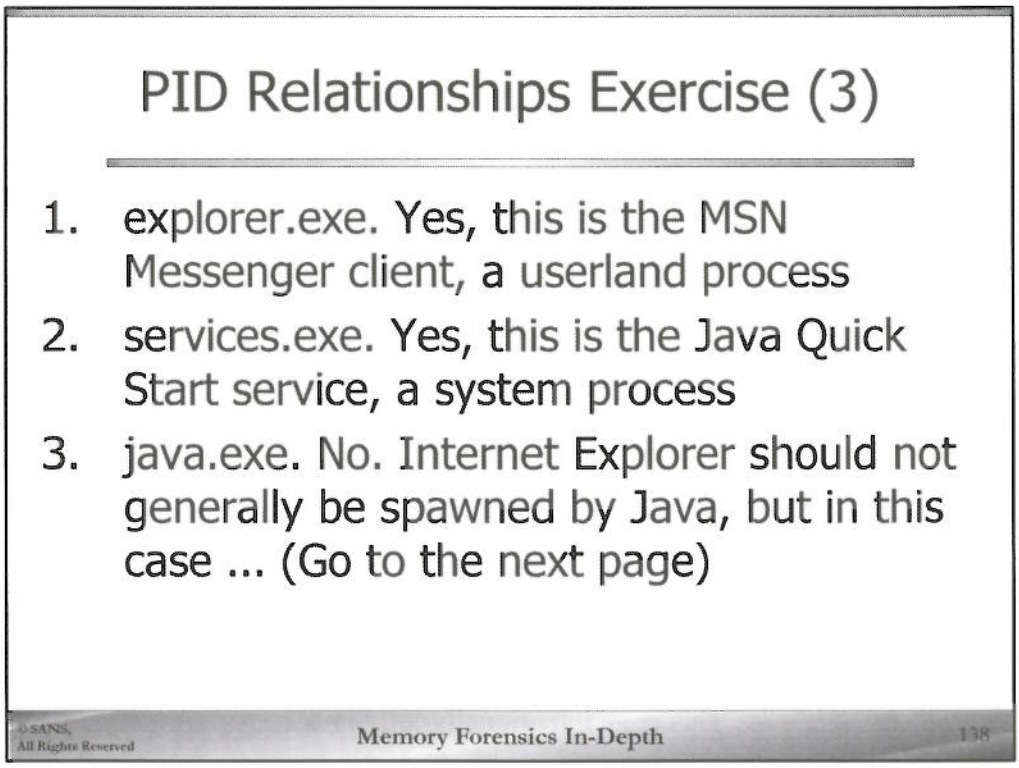


This page intentionally left blank.



This page intentionally left blank.

This page intentionally left blank.



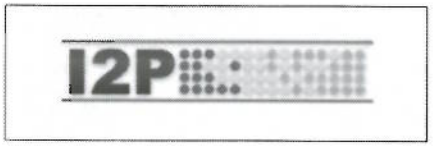
1. Ben appears to have been running MSN messenger. Whether that's an approved activity on his work computer is a policy question, not a technological one.

2. The Java Quick Start service is a legitimate system process.

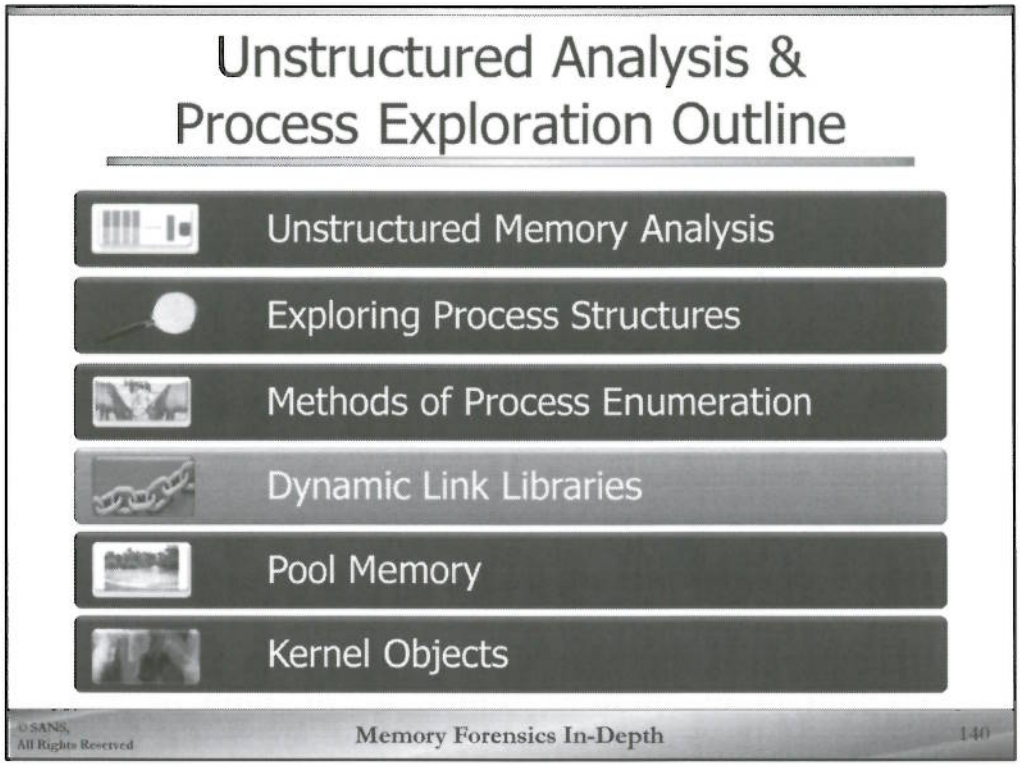
3. But a browser being spawned by Java is not generally a good sign. But in this case, it's actually legitimate. Go on to the next page.

PID Relationships Exercise (4)

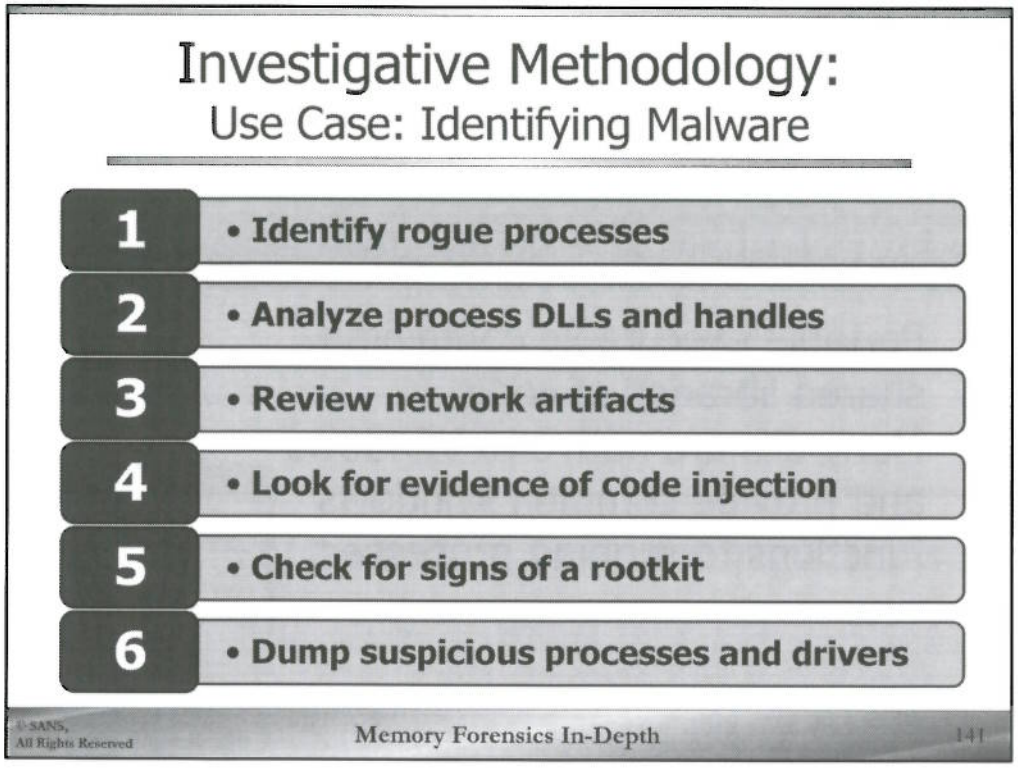
Name	Pid	PPid
0x81FF1020:I2Psvc.exe	3316	668
0x820D4290:java.exe	3244	3316
0x81F68020:iexplore.exe	1736	3244



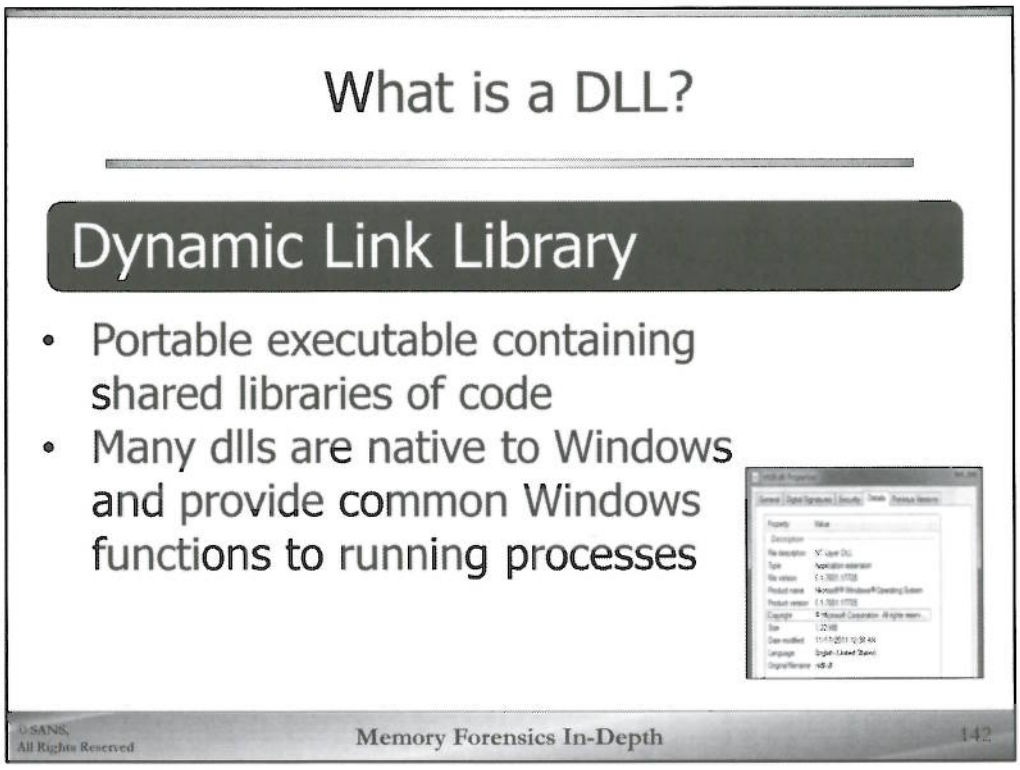
As with most things, the answer to question number three is a little more complicated than it would first appear. The program in question I2Psvc, is part of a legitimate program called i2p, http://www.i2p2.de/. This is a proxy for outbound and inbound traffic through an anonymization service. It's current implementation is done in Java. In this case, the user deliberately installed this program. It's legitimate. But looking at the larger picture, this is an installed program which is gathering and processing all outbound network traffic. That's certainly something a malicious program would try to do! Although we didn't find malware in this case, we found a program engaging in a potentially malicious manner. That's still a victory.



This page intentionally left blank.



In reviewing our progress through the 6 step Investigative Methodology for Identifying Malware through memory analysis, we have sufficient covered Step 1. Identifying Rogue Processes. There are numerous ways to enumerate processes as we have seen using both Volatility and Rekall. This next section, focused on Step 2, will dive deeper into the handles and dynamic link libraries (DLLs) of these suspicious processes, furthering our insight into the functionality and capabilities of the malicious code.

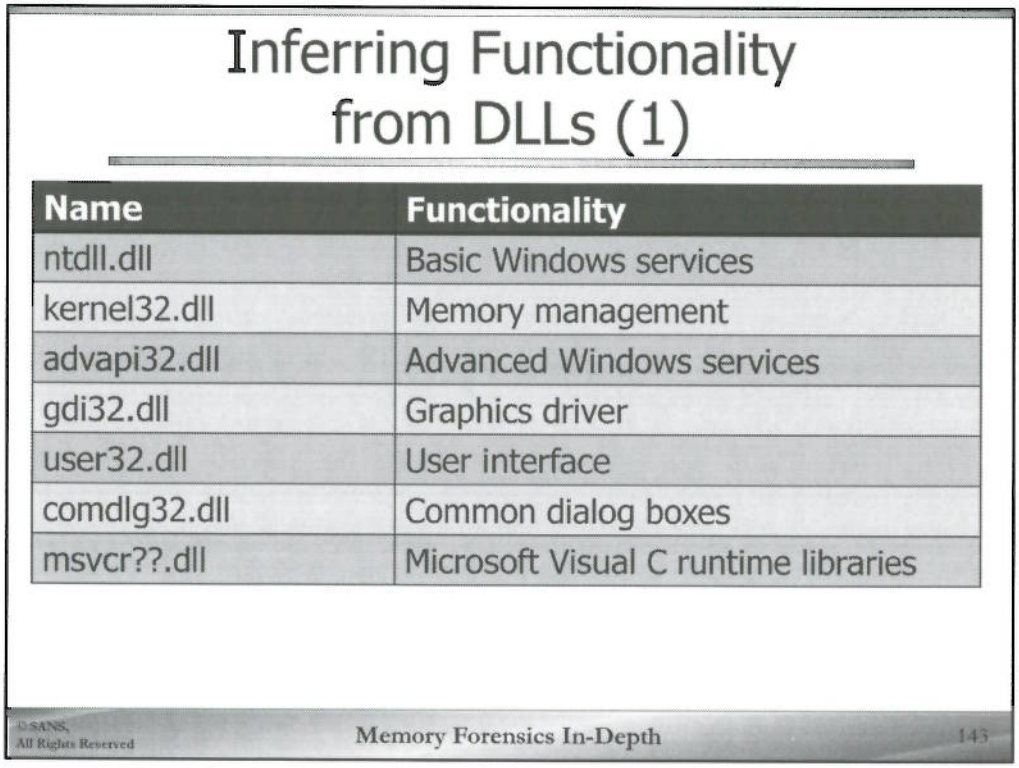


Dynamic Link Libraries, or DLLs, are shared libraries of code. They contain functionality which can be used by multiple programs. Some of them are provided by Microsoft for the functionality of the operating system, but many come from third-party developers. Separating functionality from a program is a long standing technique in computer science. It allows application programmers to rely on third party libraries of established, tested, and proven code, especially for difficult operations like cryptography, graphical interfaces, or networking.

Like traditional Windows executables, Windows DLLs are PE files. (The PE file format, or Portable Executable, is a standard Windows executable file format.) DLLs have the same format as traditional PE files, but cannot be executed directly. They have to be called from another program.

Every legitimate process loads some DLLs into memory when it runs. For example, ntdll.dll, shown above, provides some basic functionality necessary for the process to start. Not only are DLLs loaded when a process starts, but they can also be dynamically loaded as needed while the process is running. Let's say a process does some work with cryptography. Rather than loading all of the crypto code into memory when the process starts, the operating system may indicate that the code is available in memory, but not actually load anything until the code is actually called.

Going one step further, a program can manually load DLLs during execution. This technique, although perfectly valid, is more often used by malicious programs. They hide which DLLs they import by not including them in the original list of DLLs to load when the process starts. After the process is already running, they manually call the functions necessary to load and execute the code in o



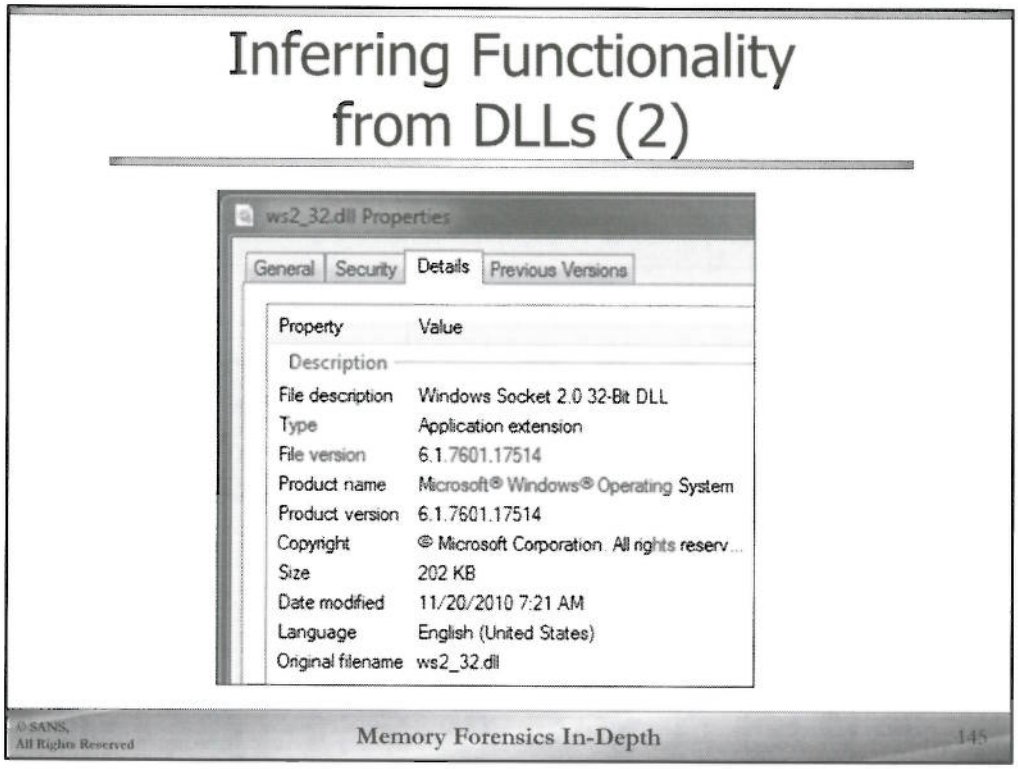
Windows system DLLs are divided by functionality. There are DLLs related to networking functions, cryptography functions, application frameworks, and so on. By looking at the list of DLLs which are used by a process, we can get a (very) rough sense of what that program does. Admittedly, just because a program loads a DLL doesn't mean it actually *uses* it. "Furthermore, we can't tell what functions in a DLL a program uses just by seeing that it's loaded. The specific list of imported functions can be determined statically during reverse engineering by examining the program's import table.

Here are just a few common DLLs used by many programs on Windows. There are many more. You'll see these often:

- Windows kernel basic services. These are low-level services necessary for processes to start and talk to the operating system.
- The Windows APIs for things like memory management, input/output, etc., are found in kernel32.dll.
- Advanced kernel services, such as accessing the registry, are in advapi32.dll.
- Gdi32.dll provides control of the graphics interface.
- Functions for creating and manipulating the user interface, such as scroll bars, buttons, mouse and keyboard input, are in user32.dll.

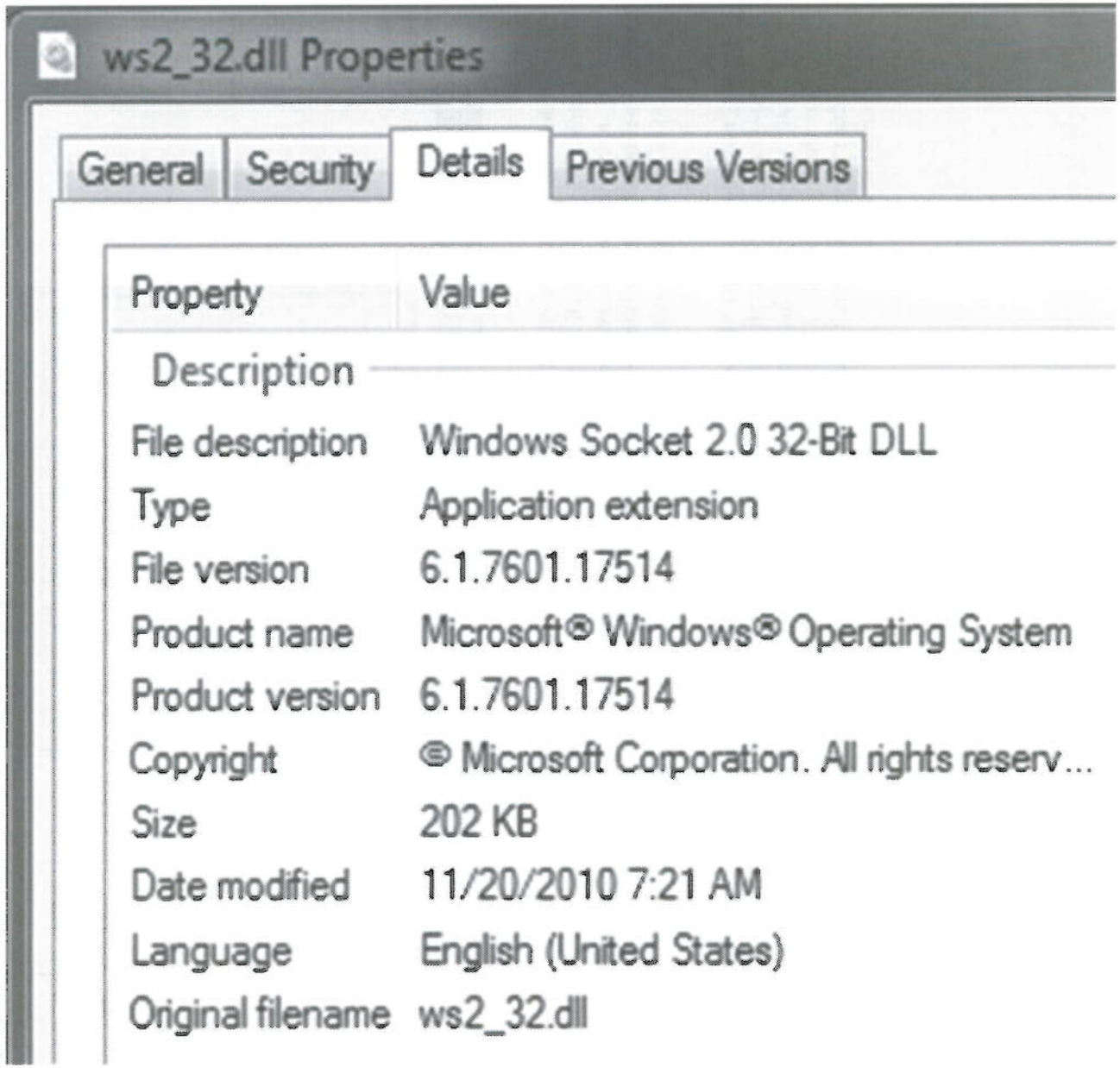
- Dialog boxes for opening and saving files, etc., are in comdlg32.dll.

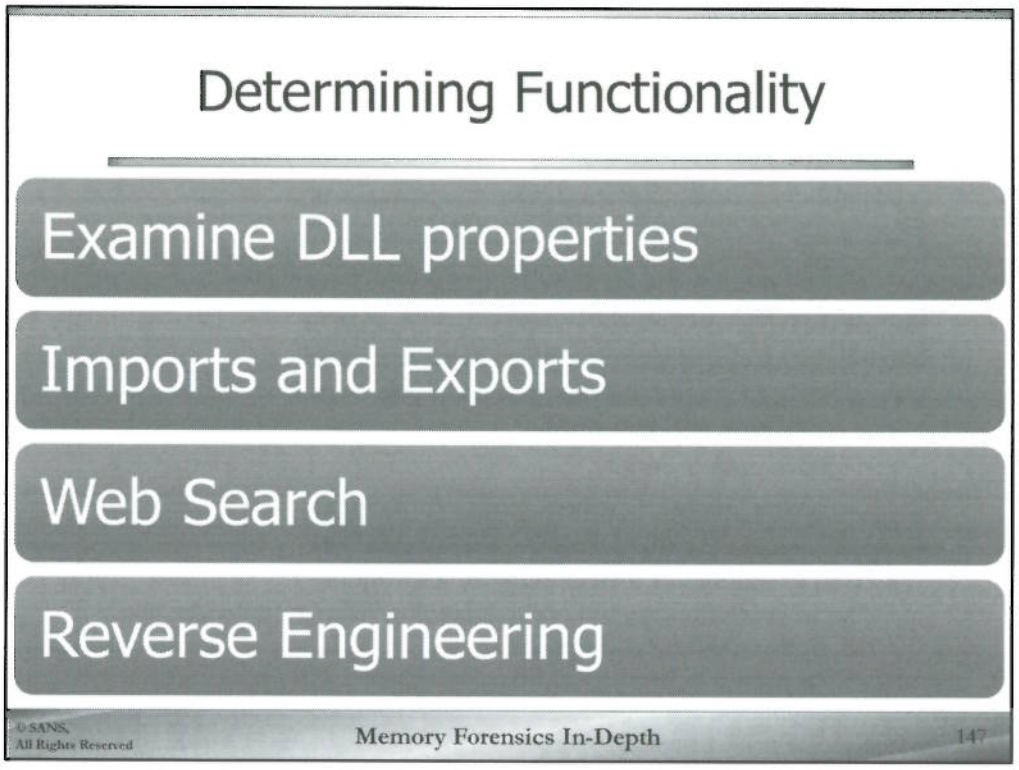
- The Microsoft Visual C Runtime Libraries – These DLLs are distributed by Microsoft as part of their Visual C and C++ development tools. Software authors either need to link their programs statically or include these DLLs to ensure their programs run correctly. The DLLs have filenames of the format msvcr[version].dll. For example, you will see msvcr70.dll, msvcr71.dll, msvcr80.dll, msvcr90.dll, and the current version, msvcr100.dll. There are more details on this DLL at http://msdn.microsoft.com/en-us/library/abx4dbyh(v=vs.100).aspx.



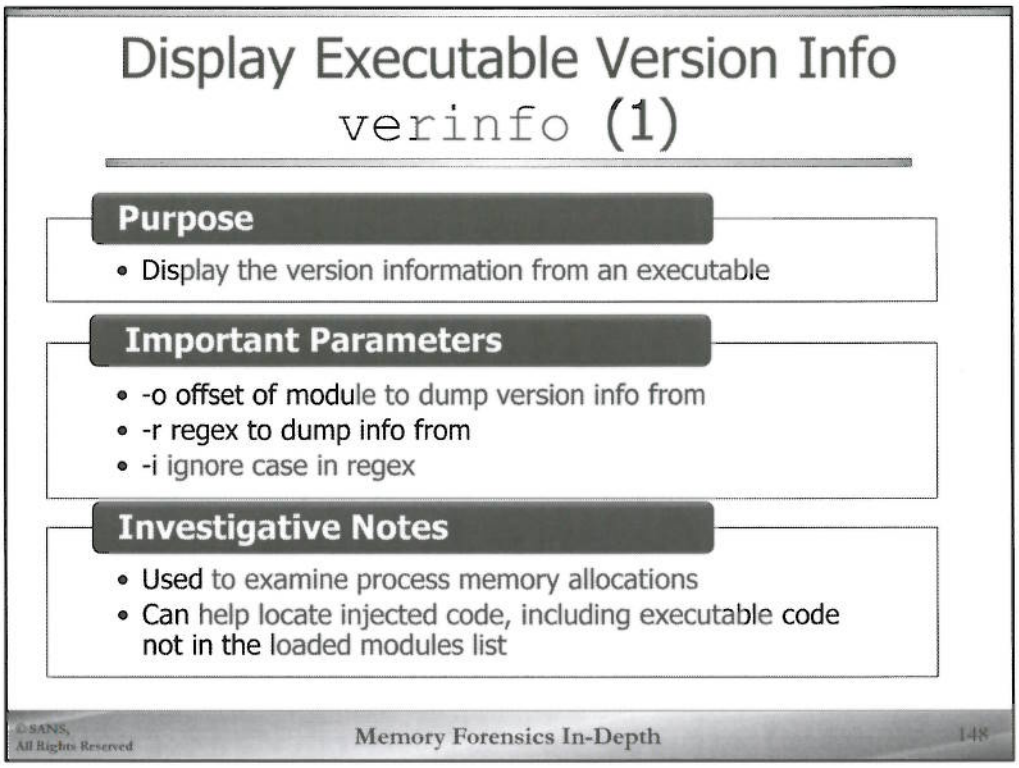
Though these are all widely used and legitimate DLLs, some of them may be of interest in your investigations. For example, ws2_32.dll is a DLL from Microsoft used for low-level networking functionality. These functions, also called raw sockets, are used by legitimate programs to access the network. Many functions use higher-level functionality by Windows. In general most programmers don't want to deal with the full complexity of network programming.

But many malicious programs *do* want to use raw sockets. They don't want to garner the attention of the operating system, and thus do all of the work themselves. (That, or they *can't* invoke the operating system's functionality.) So many malicious programs will use ws2_32.dll. Finding that DLL is not, in and of itself, suspicious. But if you see ws2_32.dll being used by a program called "Notepad.exe", for example, there may be something strange going on. The legitimate Notepad does not use the network, and does not import any functions from ws2_32.dll.


ws2_32.dll Properties
General
Security
Details
Previous Versions
Property
Value
Description
File description
Windows Socket 2.0 32-Bit DLL
Type
Application extension
File version
6.1.7601.17514
Product name
Microsoft® Windows® Operating System
Product version
6.1.7601.17514
Copyright
© Microsoft Corporation. All rights reserv...
Size
202 KB
Date modified
11/20/2010 7:21 AM
Language
English (United States)
Original filename
ws2_32.dll



The filename of a DLL is only the first step in figuring out what it does. To get more information on any particular DLL, you are limited only by your imagination (and resources). You can view the properties of a DLL in Windows by right-clicking on it and choosing "Properties". Running a web search on the DLL name can give you information about other files encountered with that name. Note that such information may not apply to the DLL on the system you're examining! A common trick for malware authors is to name their programs identical to legitimate system files! You can use dedicated PE analysis tools to look at the properties of a DLL. What functions does import? What functions does it export? These names could give you clues. Finally, if resources allow, you can do a full reverse engineering of the DLL. This is a time-consuming process, and should only be a last resort.

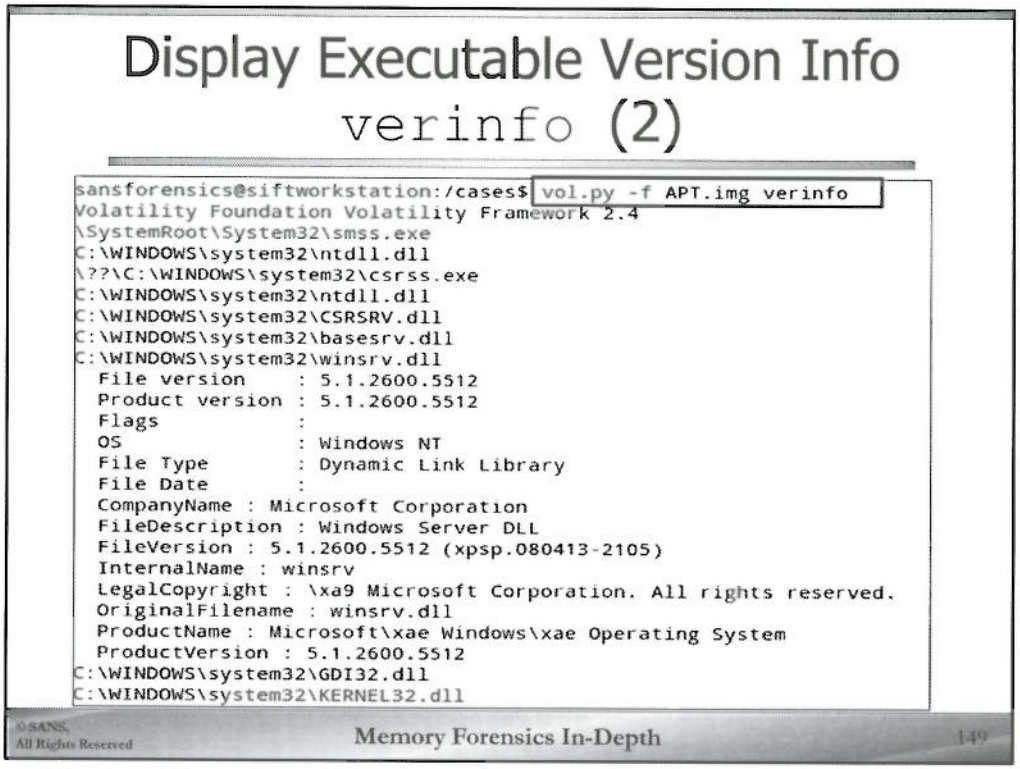


One Volatility plugin that may provide additional information on a suspicious dll is **verinfo**. The verinfo plugin is used to display the version information from an executable found in memory. This data may be useful for attribution of malware to a specific adversary. Additionally, it may be used to determine the lineage of a particular file (higher version numbers were probably developed later). Obviously, malware authors may try to forge version information, so that is something to be mindful of.

Why would an attacker put version information in a piece of malware? Simple: the compiler does it for them. Visual Studio embed version information in the binary. Some malware authors simply don't know that it's there. Others know it's there, but forget to strip it out before deploying the malware. Either way, it's a treasure trove of information. Currently the plugin only works on user space DLL's and EXE's but may later be expanded to cover kernel modules.

Note: because the verinfo plugin is located in the contrib directory, you must specify the plugins directory on the command line to access it.

The code for this plugin is located in the file contrib/plugins/verinfo.py.



Example output follows:

sansforensics@siftworkstation:/cases$ vol.py -f APT.img verinfo

Volatility Foundation Volatility Framework 2.4

\SystemRoot\System32\smss.exe

C:\WINDOWS\system32\ntdll.dll

\??\C:\WINDOWS\system32\csrss.exe

C:\WINDOWS\system32\ntdll.dll

C:\WINDOWS\system32\CSRSRV.dll

C:\WINDOWS\system32\basesrv.dll

C:\WINDOWS\system32\winsrv.dll

File version : 5.1.2600.5512

Product version : 5.1.2600.5512

Flags :

OS : Windows NT

File Type : Dynamic Link Library

File Date :

CompanyName : Microsoft Corporation

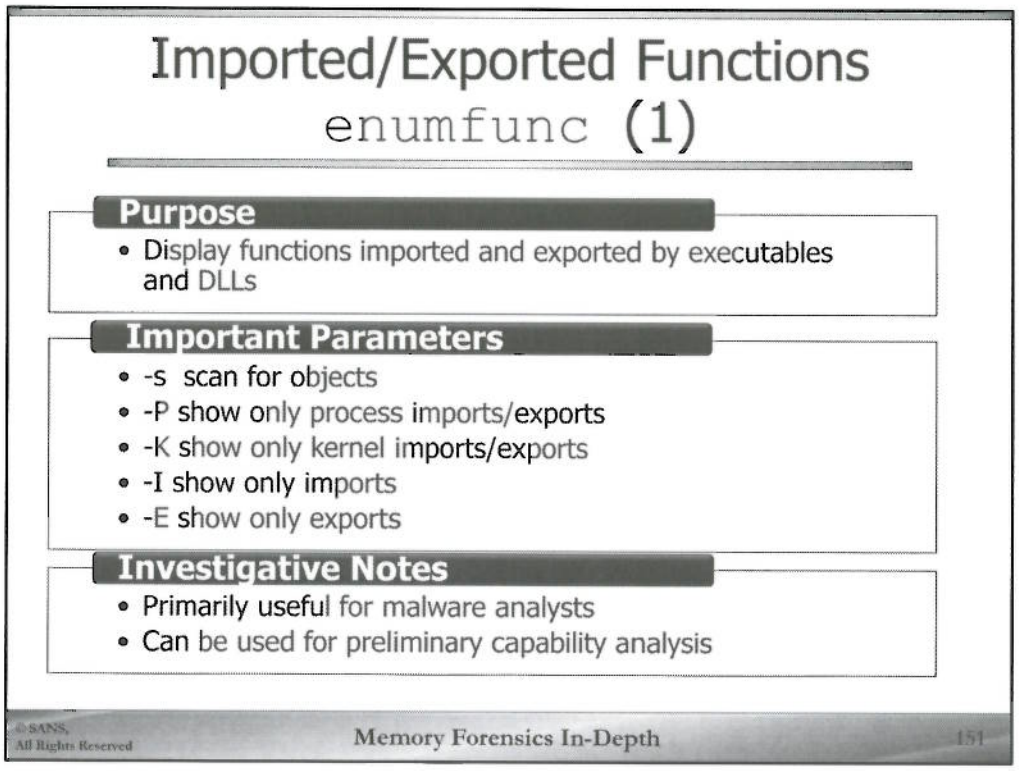
FileDescription : Windows Server DLL

FileVersion : 5.1.2600.5512 (xpsp.080413-2105)

InternalName : winsrv

LegalCopyright : \xa9 Microsoft Corporation. All rights reserved.
OriginalFilename : winsrv.dll
ProductName : Microsoft\xae Windows\xae Operating System
ProductVersion : 5.1.2600.5512
C:\WINDOWS\system32\GDI32.dll
C:\WINDOWS\system32\KERNEL32.dll
C:\WINDOWS\system32\USER32.dll
File version : 5.1.2600.5512
Product version : 5.1.2600.5512
Flags :
OS : Windows NT
File Type : Dynamic Link Library
File Date :
CompanyName : Microsoft Corporation
FileDescription : Windows XP USER API Client DLL
FileVersion : 5.1.2600.5512 (xpsp.080413-2105)
InternalName : user32
LegalCopyright : \xa9 Microsoft Corporation. All rights reserved.
OriginalFilename : user32
ProductName : Microsoft\xae Windows\xae Operating System
ProductVersion : 5.1.2600.5512
…. Output truncated …

File Date :
CompanyName : Microsoft Corporation
FileDescription : Generic Host Process for Win32 Services
FileVersion : 5.1.2600.5512 (xpsp.080413-2111)
InternalName : svchost.exe
LegalCopyright : \xa9 Microsoft Corporation. All rights reserved.
OriginalFilename : svchost.exe
ProductName : Microsoft\xae Windows\xae Operating System
ProductVersion : 5.1.2600.5512

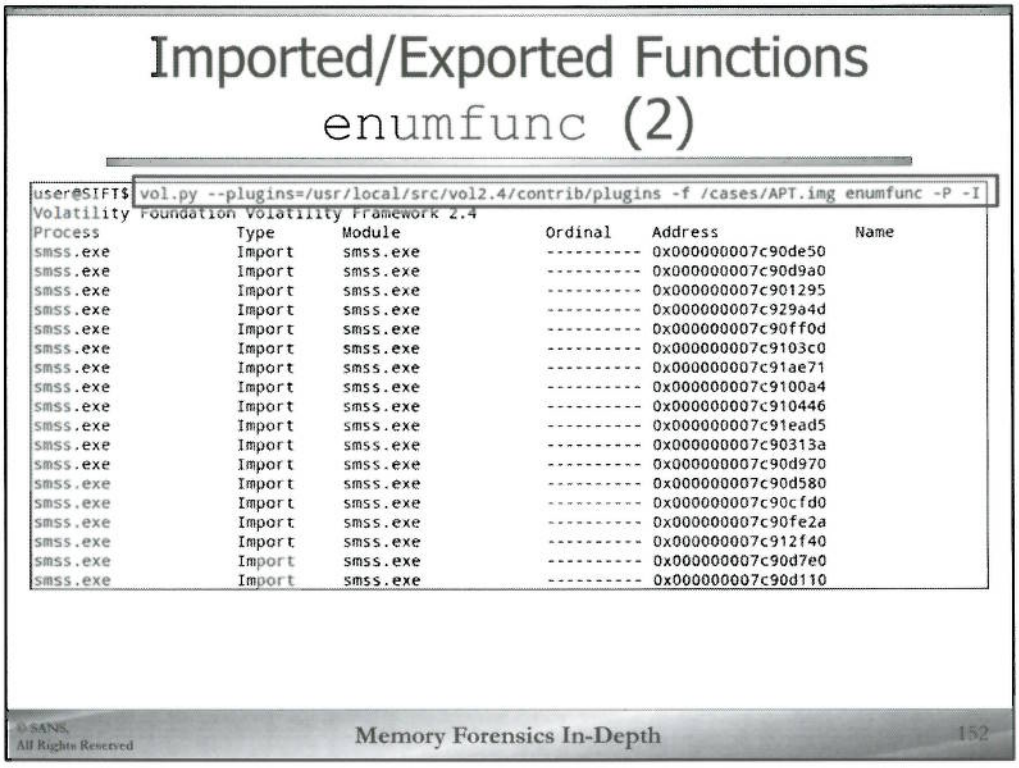


Volatility can give us deeper visibility into a dll's functionality as well, with the **enumfunc** plugin. The enumfunc plugin is used to display the imported and exported functions present in a memory dump. It enumerates across all processes and kernel modules loaded in the memory dump.

Note that this plugin can be used to find stealthy injected code. For instance, we all know that the process notepad.exe (built into windows) doesn't import any networking capability. However, if we see exports from networking DLLs in notepad.exe's process space, this is clear sign that something was injected into notepad's address space. Using the impscan plugin (covered separately) you may be able to discover which memory addresses the injected code is located at.

NOTE: enumfunc will not enumerate imported or exported functions from hidden processes and modules.

The code for this plugin is also located in the file contrib/plugins/enumfunc.py

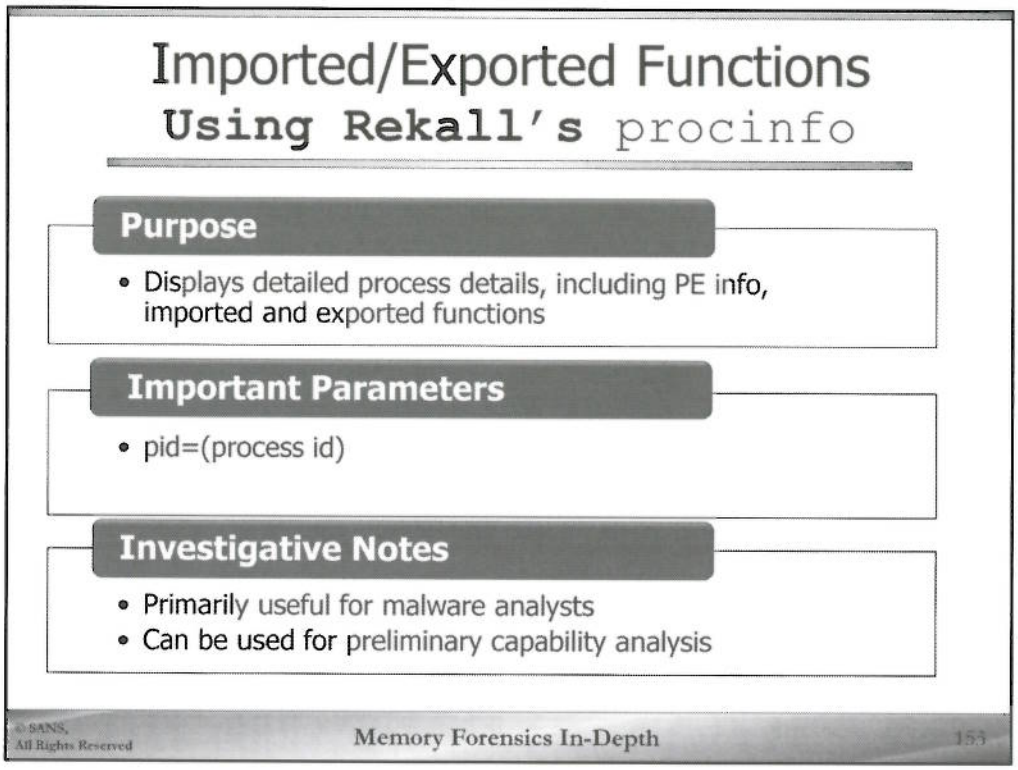


NOTE: because enumfunc is a contributed plugin, you must specify the --plugin directory for it to execute correctly. It may still emit some errors due to other plugins in non-standard locations. These errors can be safely ignored.

Note that it is not currently possible to specify a process ID to obtain imports and exports from. However, liberal use of the grep command can trim the output to a particular process name of interest.

user@SIFT$ vol.py --plugins=/usr/local/src/vol2.4/contrib/plugins -f /cases/APT.img enumfunc -P -I | head -40

```
Volatility Foundation Volatility Framework 2.4
Process          Type     Module          Ordinal    Address            Name
smss.exe         Import   smss.exe          ---------- 0x000000007c90de50
smss.exe         Import   smss.exe          ---------- 0x000000007c90d9a0
smss.exe         Import   smss.exe          ---------- 0x000000007c901295
smss.exe         Import   smss.exe          ---------- 0x000000007c929a4d
smss.exe         Import   smss.exe          ---------- 0x000000007c90ff0d
smss.exe         Import   smss.exe          ---------- 0x000000007c9103c0
smss.exe         Import   smss.exe          ---------- 0x000000007c91ae71
smss.exe         Import   smss.exe          ---------- 0x000000007c9100a4
smss.exe         Import   smss.exe          ---------- 0x000000007c910446
smss.exe         Import   smss.exe          ---------- 0x000000007c91ead5
smss.exe         Import   smss.exe          ---------- 0x000000007c90313a
smss.exe         Import   smss.exe          ---------- 0x000000007c90d970
smss.exe         Import   smss.exe          ---------- 0x000000007c90d580
smss.exe         Import   smss.exe          ---------- 0x000000007c90cfd0
smss.exe         Import   smss.exe          ---------- 0x000000007c90fe2a
smss.exe         Import   smss.exe          ---------- 0x000000007c912f40
... output truncated ...
```




```
[1] stuxnet.vmem 16:24:39> procinfo pid=1928
------------------------> procinfo(pid=1928)
**********************************************
Pid: 1928 lsass.exe

Process Environment
  ALLUSERSPROFILE=C:\Documents and Settings\All Users
  CommonProgramFiles=C:\Program Files\Common Files
  COMPUTERNAME=JAN-DF663B3DBF1
  ComSpec=C:\WINDOWS\system32\cmd.exe
  FP_NO_HOST_CHECK=NO
  NUMBER_OF_PROCESSORS=1
  OS=Windows_NT

Path=C:\Perl\site\bin;C:\Perl\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\
Python26;C:\Program Files\TortoiseSVN\bin
  PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
  PROCESSOR_ARCHITECTURE=x86
  PROCESSOR_IDENTIFIER=x86 Family 6 Model 37 Stepping 5, GenuineIntel
  PROCESSOR_LEVEL=6
  PROCESSOR_REVISION=2505
  ProgramFiles=C:\Program Files
```

```
SystemDrive=C:
  SystemRoot=C:\WINDOWS
  TEMP=C:\WINDOWS\TEMP
  TMP=C:\WINDOWS\TEMP
  USERPROFILE=C:\Documents and Settings\LocalService
  VS100COMNTOOLS=C:\Program Files\Microsoft Visual Studio 10.0\Common7\Tools\
  windir=C:\WINDOWS

PE Infomation
        Attribute                          Value
------------------------------ ----------------------------------------------------------------
Machine                  IMAGE_FILE_MACHINE_I386
TimeDateStamp              2010-01-13 10:00:53+0000
Characteristics          IMAGE_FILE_32BIT_MACHINE, IMAGE_FILE_EXECUTABLE_IMAGE
GUID/Age                  -
PDB                     -
MajorOperatingSystemVersion   5
MinorOperatingSystemVersion   0
MajorImageVersion          0
MinorImageVersion          0
MajorSubsystemVersion        5
MinorSubsystemVersion        0

Sections (Relative to 0x01000000):
Perm  Name     VMA        Size
---- -------- ---------- ----------
xrw  .verif   0x00001000 0x00000000
xrw  .text    0x00002000 0x00001c00
-rw  .bin     0x00004000 0x00000200
-r-  .reloc   0x00005000 0x00000200

Data Directories:
----------------------------------------  VMA       Size
                                  ---------- ----------
IMAGE_DIRECTORY_ENTRY_EXPORT          0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_IMPORT          0x010038fc 0x00000050
IMAGE_DIRECTORY_ENTRY_RESOURCE        0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_EXCEPTION       0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_SECURITY        0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_BASERELOC       0x01005000 0x000000f4
IMAGE_DIRECTORY_ENTRY_DEBUG           0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_COPYRIGHT       0x00000000 0x00000000
```

IMAGE_DIRECTORY_ENTRY_GLOBALPTR	0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_TLS	0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_IAT	0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	0x00000000 0x00000000
IMAGE_DIRECTORY_ENTRY_RESERVED	0x00000000 0x00000000

Import Directory (Original):

Name	Mapped Function	Ord
KERNEL32.dll!OpenProcess		819
KERNEL32.dll!ExitProcess		260
KERNEL32.dll!WaitForSingleObject		1124
KERNEL32.dll!SetUnhandledExceptionFilter		1045
KERNEL32.dll!SetErrorMode		978
KERNEL32.dll!CloseHandle		67
KERNEL32.dll!GetCurrentProcess		425
KERNEL32.dll!CreateThread		163
KERNEL32.dll!TerminateProcess		1069
KERNEL32.dll!VirtualProtect		1114
KERNEL32.dll!GetModuleHandleW		505
KERNEL32.dll!GetCurrentThreadId		429
KERNEL32.dll!GetTickCount		614
KERNEL32.dll!lstrcpyW		1200
KERNEL32.dll!lstrlenW		1206
KERNEL32.dll!GetProcAddress		544
ADVAPI32.dll!LookupPrivilegeValueW		401
ADVAPI32.dll!AdjustTokenPrivileges		30
ADVAPI32.dll!OpenProcessToken		497
USER32.dll!wsprintfW		776

Export Directory:

Entry	Stat	Ord	Name

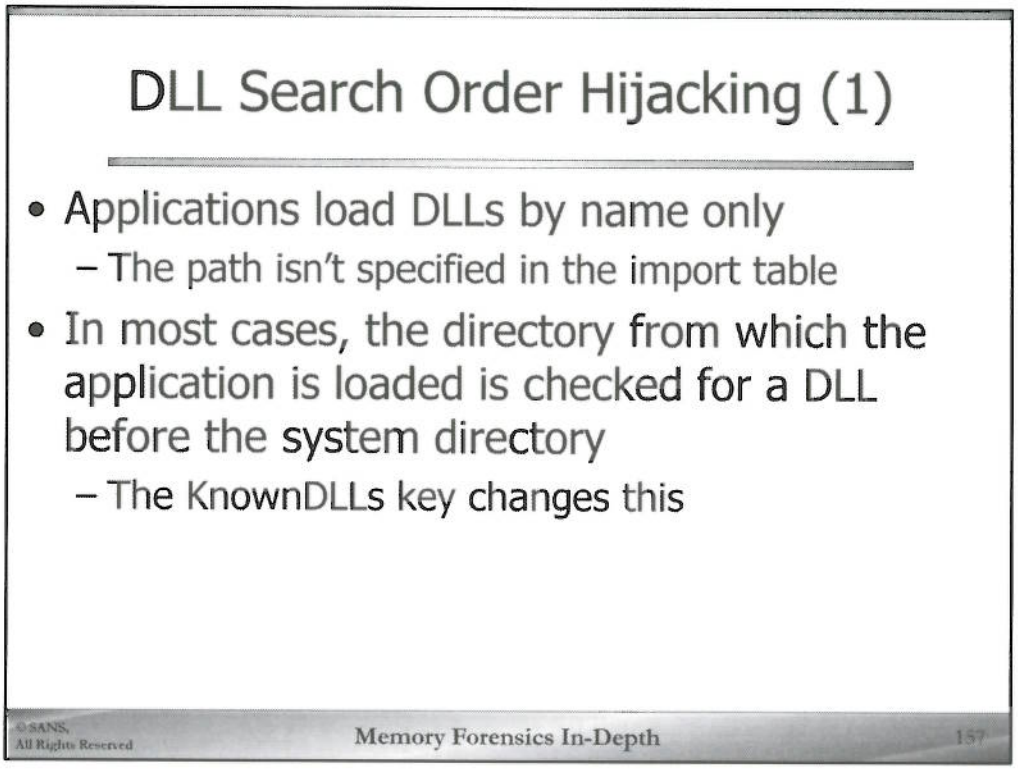
Version Information:

key	value

Imported/Exported Functions
Using Rekall's procinfo

```
Import Directory (Original):
                        Name                 Mapped Function                  Ord
-------------------------------------------------------------------------- -----
KERNEL32.dll!OpenProcess                                                    819
KERNEL32.dll!ExitProcess                                                    260
KERNEL32.dll!WaitForSingleObject                                            1124
KERNEL32.dll!SetUnhandledExceptionFilter                                    1045
KERNEL32.dll!SetErrorMode                                                   978
KERNEL32.dll!CloseHandle                                                    67
KERNEL32.dll!GetCurrentProcess                                              425
KERNEL32.dll!CreateThread                                                   163
KERNEL32.dll!TerminateProcess                                               1069
KERNEL32.dll!VirtualProtect                                                 1114
KERNEL32.dll!GetModuleHandleW                                               505
KERNEL32.dll!GetCurrentThreadId                                             429
KERNEL32.dll!GetTickCount                                                   614
KERNEL32.dll!lstrcpyW                                                       1200
KERNEL32.dll!lstrlenW                                                       1206
KERNEL32.dll!GetProcAddress                                                 544
ADVAPI32.dll!LookupPrivilegeValueW                                          401
```


This page intentionally left blank.



Malicious programs have used a technique called DLL Search Order Hijacking to gain execution on a system. In this technique, the malicious software does not attempt to compromise a program on the system directly. Rather, the malware places a DLL in one of the directories Windows searches for requested DLLs. Windows maintains a list of knowndlls, in a registry key of the same name (HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs), that it populates with some native Windows dlls that will always launch from a set location (System32)[1] for easier loading. When a DLL is needed and it is NOT in the knowndlls list, Windows searches, in order, the directory where the application is stored, the system directory, the windows directory, the current directory, and then directories listed in the PATH environment variable[2]. If the malware can place a DLL into one of those directories which is searched before the location of the legitimate DLL, the program will load the malware author's DLL instead of the legitimate one.

DLL search order:

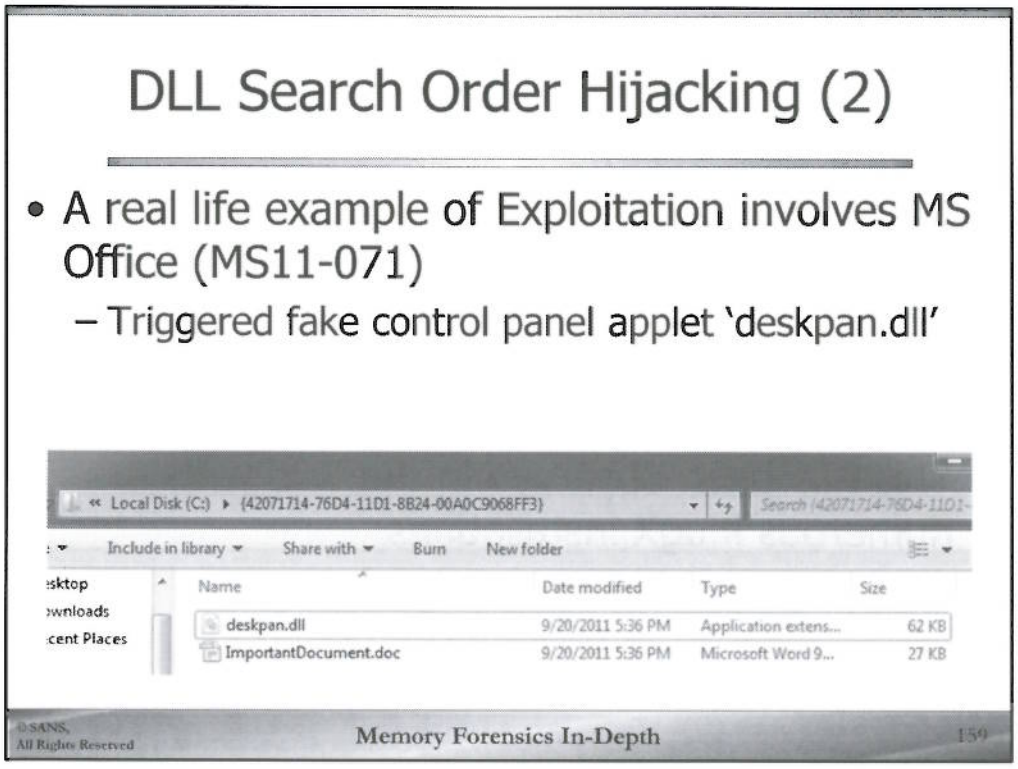
1. The directory from which the application loaded.
2. The system directory. Use the GetSystemDirectory function to get the path of this directory.
3. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
4. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key. The App Paths key is not used when computing the DLL search path.

One of the easiest techniques for identifying search order hijacking is to look at DLLs which were loaded out of any directory *other* than C:\Windows\System32. There should only be a few per process, and those should all be out of the directory where the application was stored.

For more details on DLL search order hijacking, see http://isc.sans.edu/diary.html?storyid=9445.

[1] Harbour, Nick. Malware Persistence without the Windows Registry. https://www.mandiant.com/blog/malware-persistence-windows-registry/

[2] Dynamic-Link Library Search Order. http://msdn.microsoft.com/en-us/library/ms682586%28VS.85%29.aspx



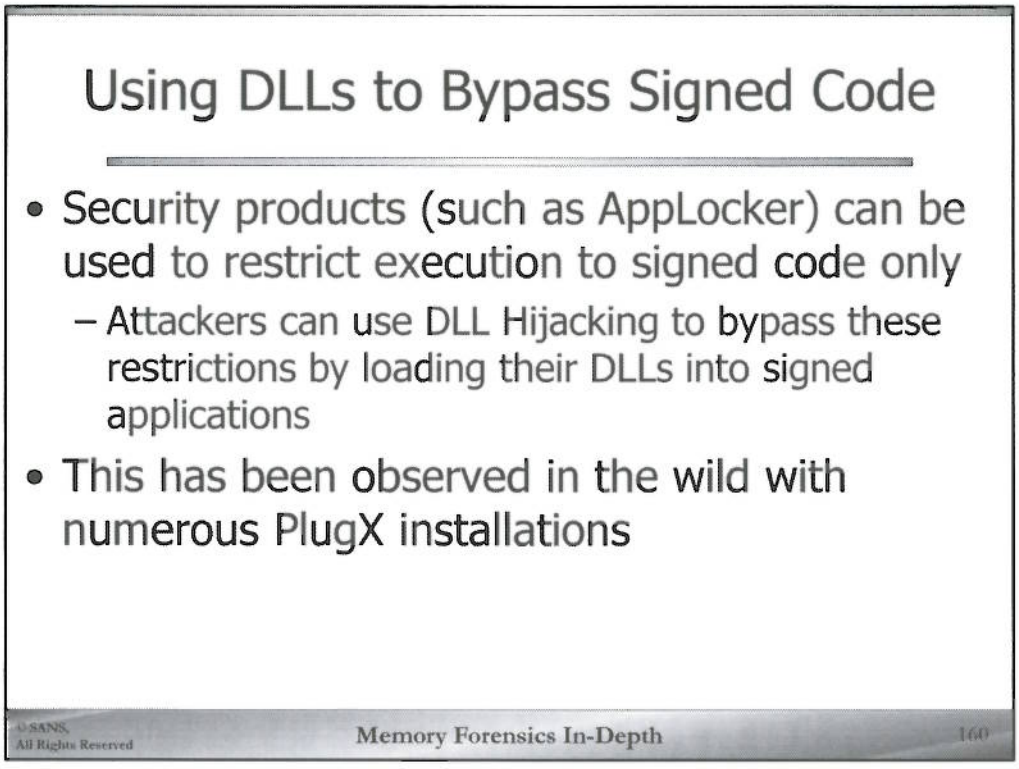
The MS11-071 vulnerability involves the use of a specially crafted deskpan.dll that loads because of its position in the same directory as a specially crafted word document. Because the deskpan.dll file is in the same folder as the winword.exe process, the attacker's deskpan.dll is loaded when the document is opened. This attack required that the current directory of the process be the directory where deskpan.dll is located. Also note the GUID directory name. This GUID essentially tricks Windows into opening the deskpan.dll control panel applet (which is not loaded by default).

Although exploitation of this vulnerability may seem complex, note that many DLL Hijacking vulnerabilities are exploited using trivial techniques. This example was used to emphasize that even Microsoft, who should know better, still gets it wrong. When Microsoft falls under fire, what chance do others have?

This attack works because the present working directory is checked for DLLs (deskpan.dll) before system directories are checked. Due to the implementation of **SafeDllSearchMode** this attack would not work on a default configured Windows system, though many systems do not use default configurations.

See also:

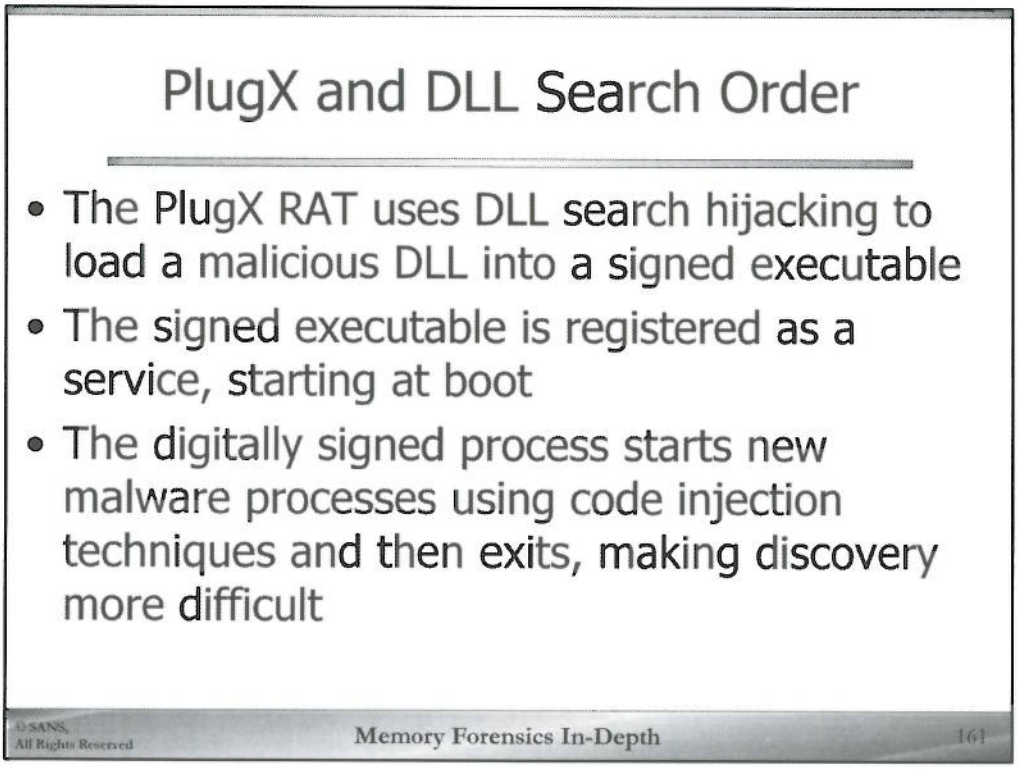
http://technet.microsoft.com/en-us/security/bulletin/MS11-071



Not only can DLLs be used to hijack programs, but they can also be used to bypass restrictions on signed code. A signed program is allowed to load and execute unsigned DLLs. Malware authors have used this trick in the real world to bypass security protections. For example, a malware author could find a signed application which loads and executes a DLL. The malware author writes their own DLL, gives it the same name as the legitimate DLL, and even writes functions with the same names as are loaded by the legitimate DLL. These functions, however, do the bidding of the malware author. The malware author then distributes the legitimate program and their own DLL.

When this program is installed and executed on a victim system, the operating system validates the signed executable and allows it to run. This executable then loads and executes the unsigned (and malicious) code in the DLL. For example, DriverVendor Inc writes and signs a program called "DisplayDriver.exe". They sign this program with their encryption key. Run run, this program looks for and executes DriverHelper.dll. Specifically, there is a function "SetupDriver", which is called. A malware author could write their own function, SetupDriver, saved in a DLL called DriverHelper.dll, and distribute it with the legitimate DisplayDriver.exe. This program passed signature verification, but when run, executes the evil SetupDriver function!

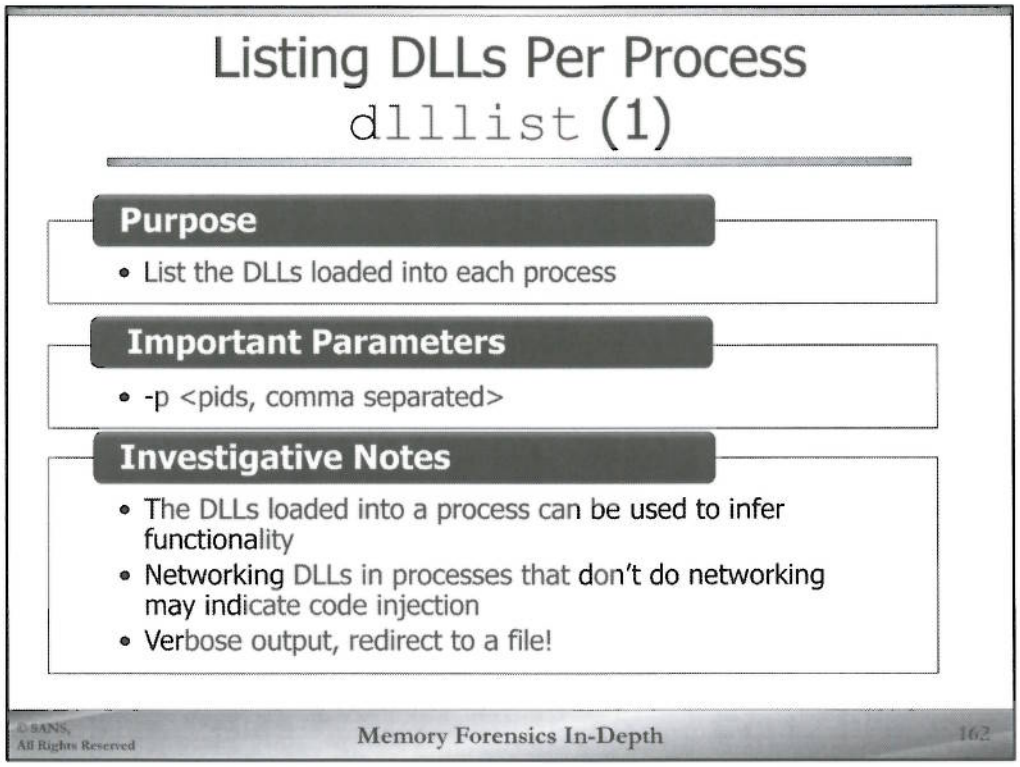
Note that although AppLocker has DLL Rules, they are considered Advanced properties. A warning screen appears when the admin attempts to enable DLL Rules noting that "DLL rules can affect system performance." DLL rules do indeed impact system performance. For this reason, they are rarely seen enabled in production.



The PlugX RAT uses DLL search hijacking to load a malicious DLL into a signed executable. The signed executable is registered as a service, starting at boot. The digitally signed process starts new malware processes using code injection techniques and then exits, making discovery more difficult.

This type of search order hijacking relies on the directory of the application being checked before the system directories. This type of search order hijacking is not impacted by **SafeDllSearchMode.**

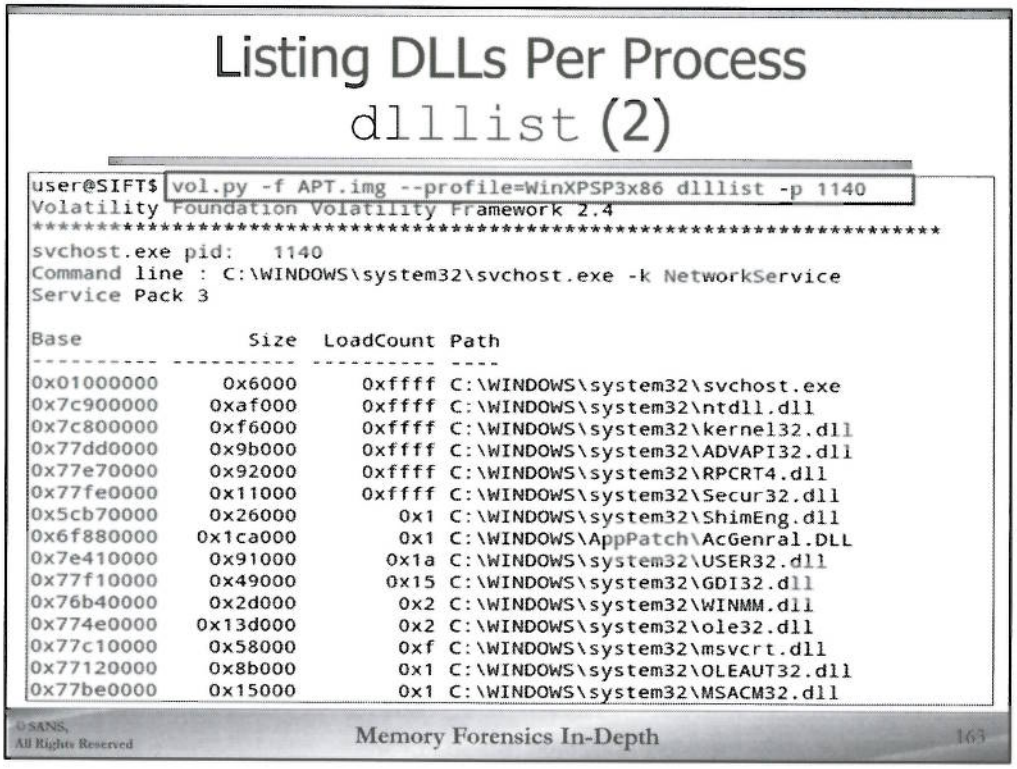
AlienVault has an outstanding writeup of the PlugX malware using this technique at https://www.alienvault.com/open-threat-exchange/blog/tracking-down-the-author-of-the-plugx-rat



The dlllist plugin lists the DLLs loaded into each process. The dlllist plugin walks the list of processes and for each process enumerates its loaded modules list. Each entry in this list represents a loaded DLL.

Examining the output of dlllist can help an investigator get a handle on the purpose of an unknown process. For example, processes that don't do any networking shouldn't have networking DLLs loaded (think calc.exe). Some processes like web browsers may implement plugin functionality using DLLs. Pay special attention to the DLLs loaded into iexplore.exe.

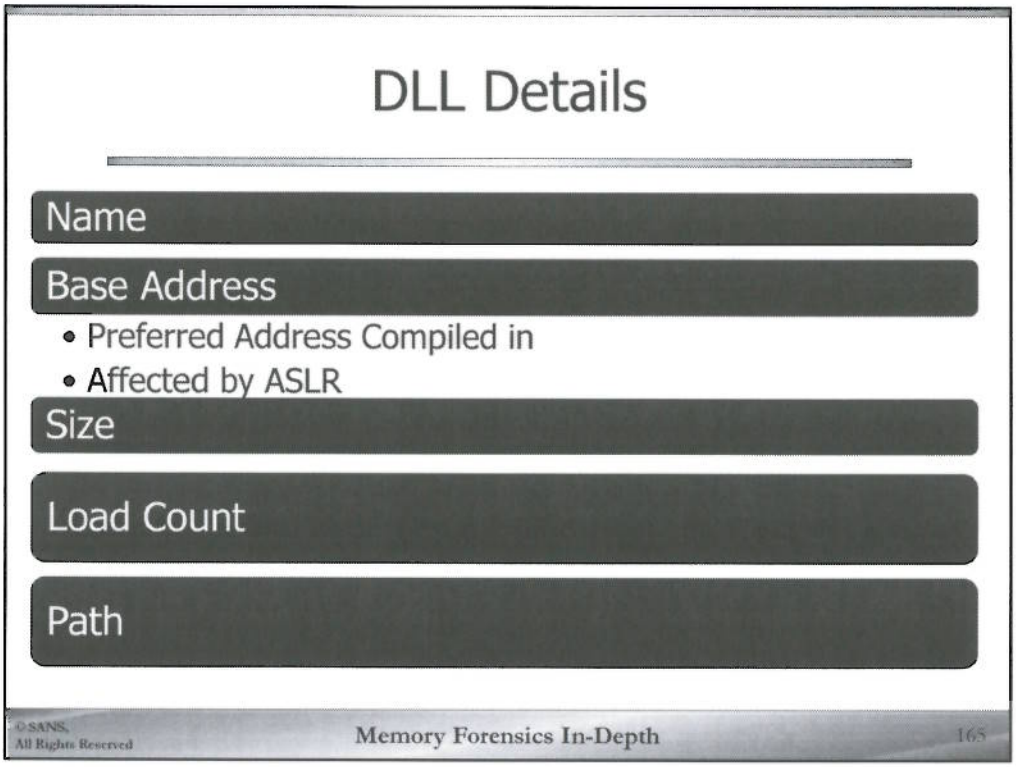
Also note that some security products load DLLs into all (or selected) processes. These DLLs are often the target of malicious software. Some security products combat this by randomizing the name of the DLLs loaded into processes. Usually randomly named DLLs are malicious, but these are legit. Later, you'll learn how to dump these DLLs and inspect their contents.



Displayed below is the output of the dlllist plugin. To save space, the plugin was only run against the selected process ID (-p 1140).

```
user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 dlllist -p 1140
Volatility Foundation Volatility Framework 2.4
************************************************************************
svchost.exe pid:   1140
Command line : C:\WINDOWS\system32\svchost.exe -k NetworkService
Service Pack 3

Base             Size  LoadCount Path
---------- ---------- ---------- ----
0x01000000     0x6000     0xffff C:\WINDOWS\system32\svchost.exe
0x7c900000    0xaf000     0xffff C:\WINDOWS\system32\ntdll.dll
0x7c800000    0xf6000     0xffff C:\WINDOWS\system32\kernel32.dll
0x77dd0000    0x9b000     0xffff C:\WINDOWS\system32\ADVAPI32.dll
0x77e70000    0x92000     0xffff C:\WINDOWS\system32\RPCRT4.dll
0x77fe0000    0x11000     0xffff C:\WINDOWS\system32\Secur32.dll
0x5cb70000    0x26000        0x1 C:\WINDOWS\system32\ShimEng.dll
0x6f880000   0x1ca000        0x1 C:\WINDOWS\AppPatch\AcGenral.DLL
0x7e410000    0x91000       0x1a C:\WINDOWS\system32\USER32.dll
0x77f10000    0x49000       0x15 C:\WINDOWS\system32\GDI32.dll
0x76b40000    0x2d000        0x2 C:\WINDOWS\system32\WINMM.dll
0x774e0000   0x13d000        0x2 C:\WINDOWS\system32\ole32.dll
0x77c10000    0x58000        0xf C:\WINDOWS\system32\msvcrt.dll
0x77120000    0x8b000        0x1 C:\WINDOWS\system32\OLEAUT32.dll
0x77be0000    0x15000        0x1 C:\WINDOWS\system32\MSACM32.dll
```

0x77f10000 0x49000 0x15 C:\WINDOWS\system32\GDI32.dll
0x76b40000 0x2d000 0x2 C:\WINDOWS\system32\WINMM.dll
0x774e0000 0x13d000 0x2 C:\WINDOWS\system32\ole32.dll
0x77c10000 0x58000 0xf C:\WINDOWS\system32\msvcrt.dll
0x77120000 0x8b000 0x1 C:\WINDOWS\system32\OLEAUT32.dll
0x77be0000 0x15000 0x1 C:\WINDOWS\system32\MSACM32.dll
0x77c00000 0x8000 0x1 C:\WINDOWS\system32\VERSION.dll
0x7c9c0000 0x817000 0x1 C:\WINDOWS\system32\SHELL32.dll
0x77f60000 0x76000 0x3 C:\WINDOWS\system32\SHLWAPI.dll
0x769c0000 0xb4000 0x1 C:\WINDOWS\system32\USERENV.dll
0x5ad70000 0x38000 0x1 C:\WINDOWS\system32\UxTheme.dll
0x76390000 0x1d000 0x2 C:\WINDOWS\system32\IMM32.DLL
0x773d0000 0x103000 0x1 C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
0x5d090000 0x9a000 0x1 C:\WINDOWS\system32\comctl32.dll
0x76770000 0xd000 0x1 c:\windows\system32\dnsrslvr.dll
0x76f20000 0x27000 0x1 c:\windows\system32\DNSAPI.dll
0x71ab0000 0x17000 0x3 c:\windows\system32\WS2_32.dll
0x71aa0000 0x8000 0x2 c:\windows\system32\WS2HELP.dll
0x76d60000 0x19000 0x2 c:\windows\system32\iphlpapi.dll



For any DLL we can recover its name and the path on disk where the DLL was loaded from. As mentioned earlier, it's a great lead for determining what the DLL does.

We can also get the base virtual address for the DLL. This is the virtual address where the DLL would prefer to be loaded into memory. Although DLLs are relocatable, they can loaded to any base address, the relocation process takes time. For performance reasons, when DLLs are compiled a base address is chosen and the code is set to run from that address. If there's another module already loaded at that address, however, the DLL is relocated, moved, to another virtual address. When this happens a number of pointers must be changed in the DLL. That's the relocation process.

In addition, the dlllist plugin provides the size of the DLL in memory in bytes and the "load count" of the dll within the process. The Load Count refers to the number of references made to the specific dll within the process, with each time the dll is loaded, the count increments by one. Likewise, when a dll is freed, the load count decrements by one.

******"View Source

Lists dll modules loaded into a process by following the doubly linked list of LDR_DATA_TABLE_ENTRY stored in in _EPROCESS.Peb.Ldr.InLoadOrderModuleList. DLLs are automatically added to this list when a process calls LoadLibrary (or some derivative such as LdrLoadDll) and they aren't removed until FreeLibrary is called and the reference count reaches zero.

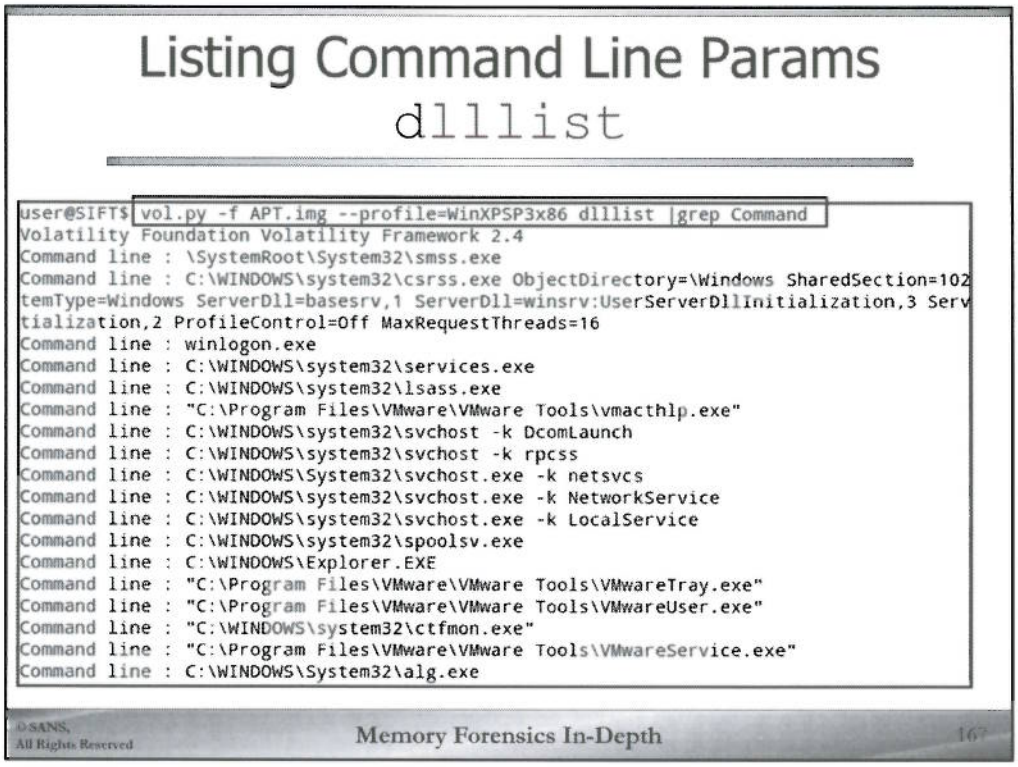
All the usual process selectors are supported.

Note:

Wow64 processes (i.e. 32-bit binaries running on 64-bit windows) load dlls through a different mechanism.

Since the InLoadOrderModuleList is maintained in the process address space, it is simple to manipulate from Ring 3 (without kernel access). This means that this plugin may not show all the linked in DLLs.

A better plugin to use is the ldrmodules plugin, which uses the VAD to enumerate dlls. The VAD is maintained in kernel memory and therefore can only be accessed through Ring 0 access."*****



The dlllist plugin also lists the complete command line of a process. This can be used to find out what that cmd.exe was really doing or whether that svchost.exe is really legit. The command line usually contains the full path to the executable, but this isn't *always* the case.

To get command line arguments for processes, just implement some grep-fu as shown below:

user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 dlllist |grep Command

Volatility Foundation Volatility Framework 2.4

Command line : \SystemRoot\System32\smss.exe

Command line : C:\WINDOWS\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,3072,512 Windows=On SubSystemType=Windows ServerDll=basesrv,1 ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=winsrv:ConServerDllInitialization,2 ProfileControl=Off MaxRequestThreads=16

Command line : winlogon.exe

Command line : C:\WINDOWS\system32\services.exe

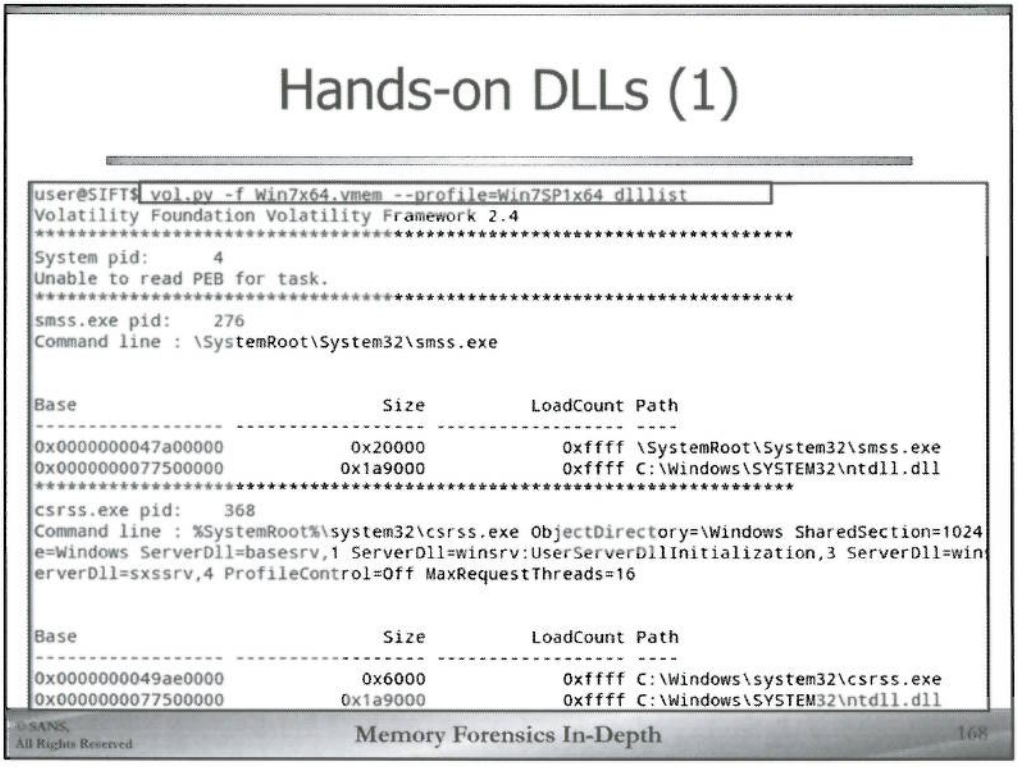
Command line : C:\WINDOWS\system32\lsass.exe

Command line : "C:\Program Files\VMware\VMware Tools\vmacthlp.exe"

Command line : C:\WINDOWS\system32\svchost -k DcomLaunch

Command line : C:\WINDOWS\system32\svchost -k rpcss

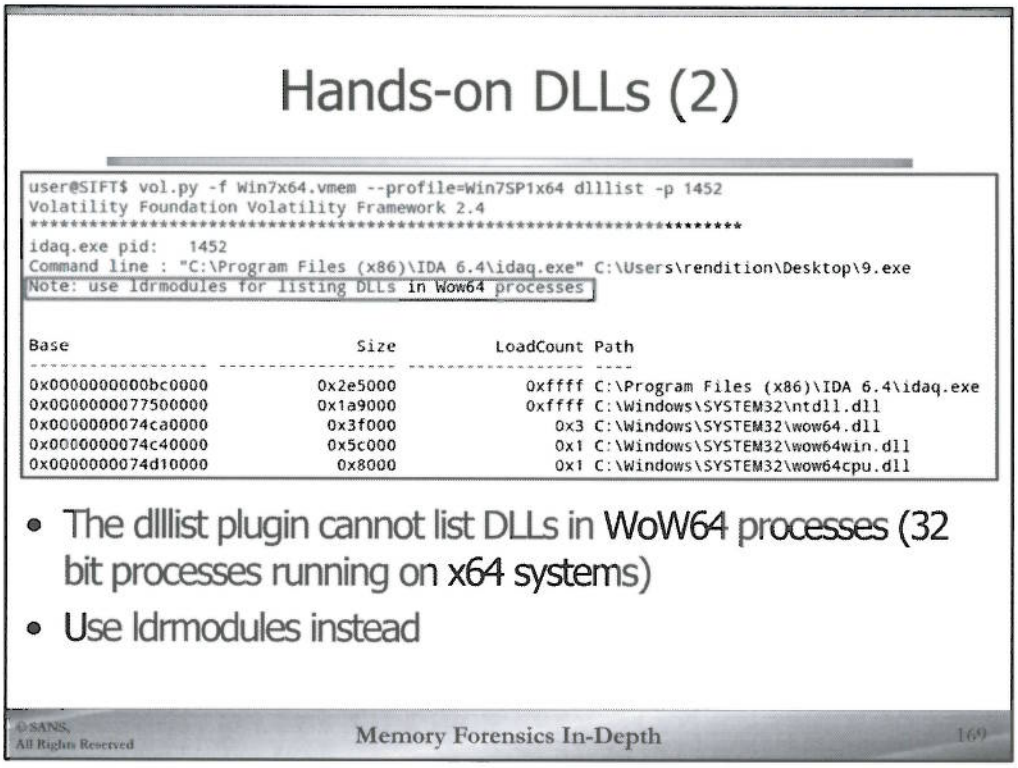
… output truncated …



We're going to take a look at the DLLs loaded in each process in a memory image. Let's start with analyzing a Windows 7 SP1 x64 memory image with the dlllist plugin in Volatility. This plugin parses a list, referenced by the process parameters, called InLoadOrderModuleList. This list contains a list of all of the modules loaded into a process' address space, their size, and virtual address where they were loaded. This list is a series of structures called LDR_DATA_TABLE, which are also used to hold information on drivers. Note that the original executable is considered a module loaded into the process too!

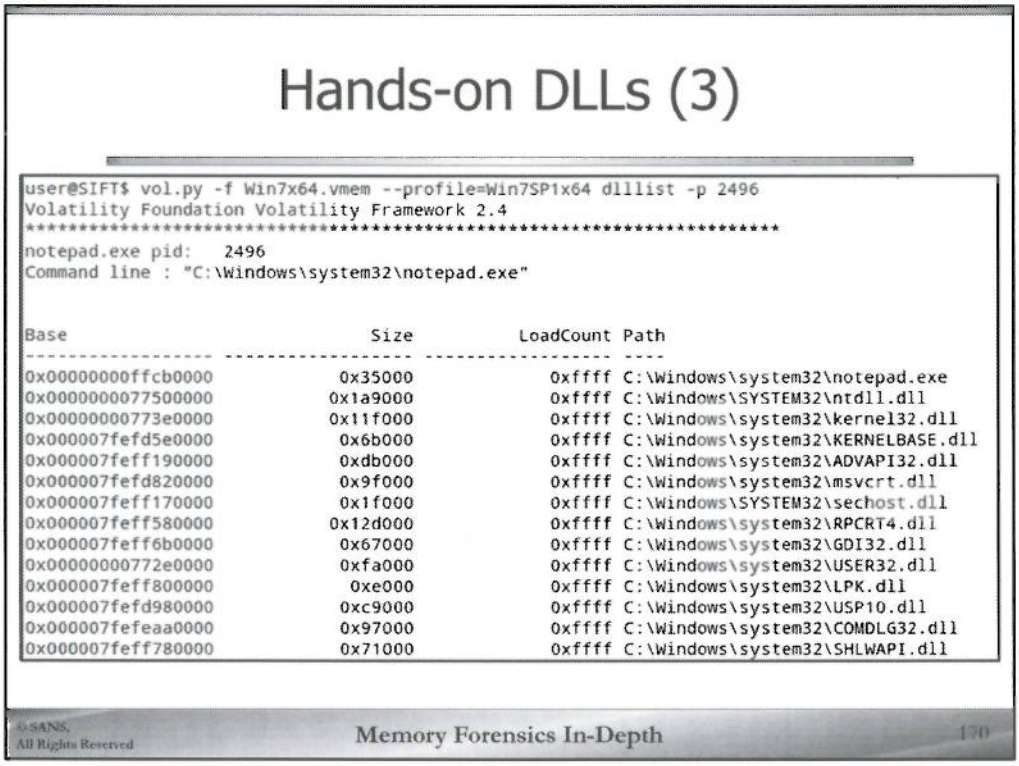
Here's the command line to run the dlllist plugin:

```
user@SIFT$ vol.py -f Win7x64.vmem --profile=Win7SP1x64 dlllist
Volatility Foundation Volatility Framework 2.4
************************************************************************
System pid:      4
Unable to read PEB for task.
************************************************************************
smss.exe pid:    276
Command line : \SystemRoot\System32\smss.exe


Base                             Size          LoadCount Path
------------------ ------------------ ------------------ ----
0x0000000047a00000            0x20000             0xffff \SystemRoot\System32\smss.exe
0x0000000077500000            0x1a9000            0xffff C:\Windows\SYSTEM32\ntdll.dll
************************************************************************
… output truncated …
```



Let's look at the end of the output, at the DLLs for the idaq.exe process. You can add a command line option, --pid (or -p), to the Volatility command line to restrict it to a single process. Here we get the DLLs for the idaq.exe, pid 1452.

```
user@SIFT$ vol.py -f Win7x64.vmem --profile=Win7SP1x64 dlllist -p 1452
Volatility Foundation Volatility Framework 2.4
**************************************************************************
idaq.exe pid:   1452
Command line : "C:\Program Files (x86)\IDA 6.4\idaq.exe"
C:\Users\rendition\Desktop\9.exe
Note: use ldrmodules for listing DLLs in Wow64 processes


Base                             Size          LoadCount Path
------------------ ------------------ ------------------ ----
0x0000000000bc0000           0x2e5000              0xffff C:\Program Files (x86)\IDA
6.4\idaq.exe
0x0000000077500000           0x1a9000              0xffff C:\Windows\SYSTEM32\ntdll.dll
0x0000000074ca0000            0x3f000                 0x3 C:\Windows\SYSTEM32\wow64.dll
0x0000000074c40000            0x5c000                 0x1
C:\Windows\SYSTEM32\wow64win.dll
0x0000000074d10000             0x8000                 0x1
C:\Windows\SYSTEM32\wow64cpu.dll
```



user@SIFT$ vol.py -f Win7x64.vmem --profile=Win7SP1x64 dlllist -p 2496

Volatility Foundation Volatility Framework 2.4

****

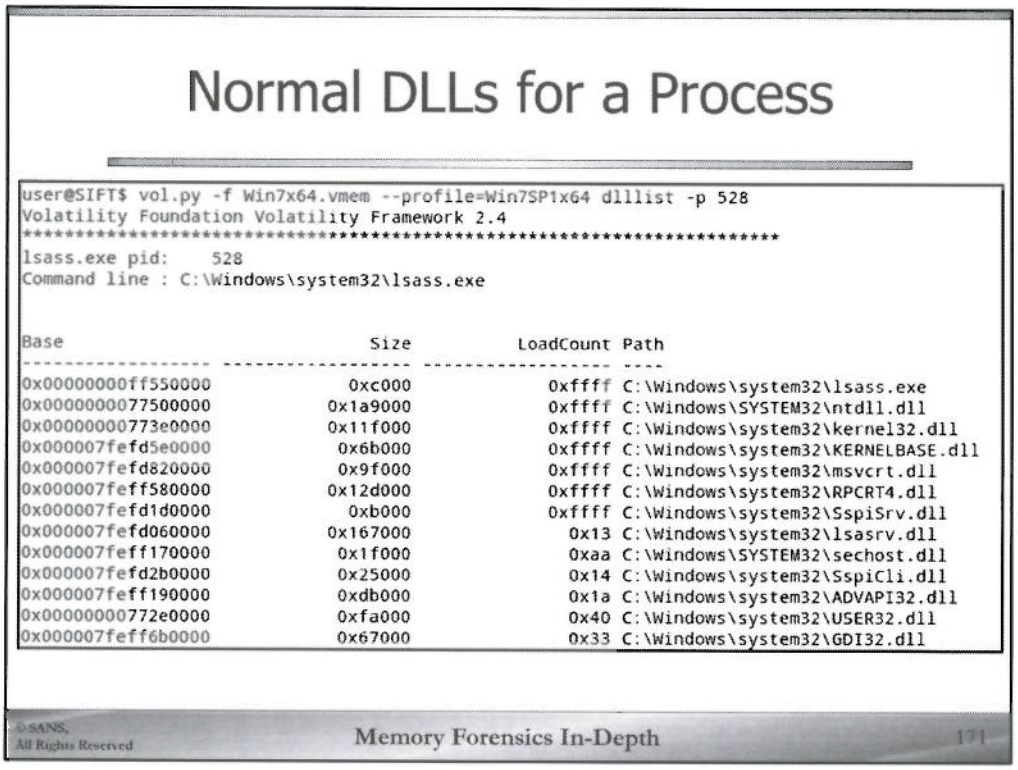
notepad.exe pid: 2496

Command line : "C:\Windows\system32\notepad.exe"

Base	Size	LoadCount	Path
0x00000000ffcb0000	0x35000	0xffff	C:\Windows\system32\notepad.exe
0x0000000077500000	0x1a9000	0xffff	C:\Windows\SYSTEM32\ntdll.dll
0x00000000773e0000	0x11f000	0xffff	C:\Windows\system32\kernel32.dll
0x000007fefd5e0000	0x6b000	0xffff	C:\Windows\system32\KERNELBASE.dll
0x000007feff190000	0xdb000	0xffff	C:\Windows\system32\ADVAPI32.dll
0x000007fefd820000	0x9f000	0xffff	C:\Windows\system32\msvcrt.dll

... output truncated ...

You can see ntdll and kernel32 at the top of the list. These two DLLs came from the Windows system directory, which is where the legitimate copies live. We then see the program loaded two DLLs which were in the same directory as the executable, getopt.dll and msvcr70.dll. The latter is the Microsoft Visual C runtime library. The former is probably some version of the *nix function getopt, used for parsing command line arguments. After these we see a number of Windows system DLLs, which this program needed to get its work done.

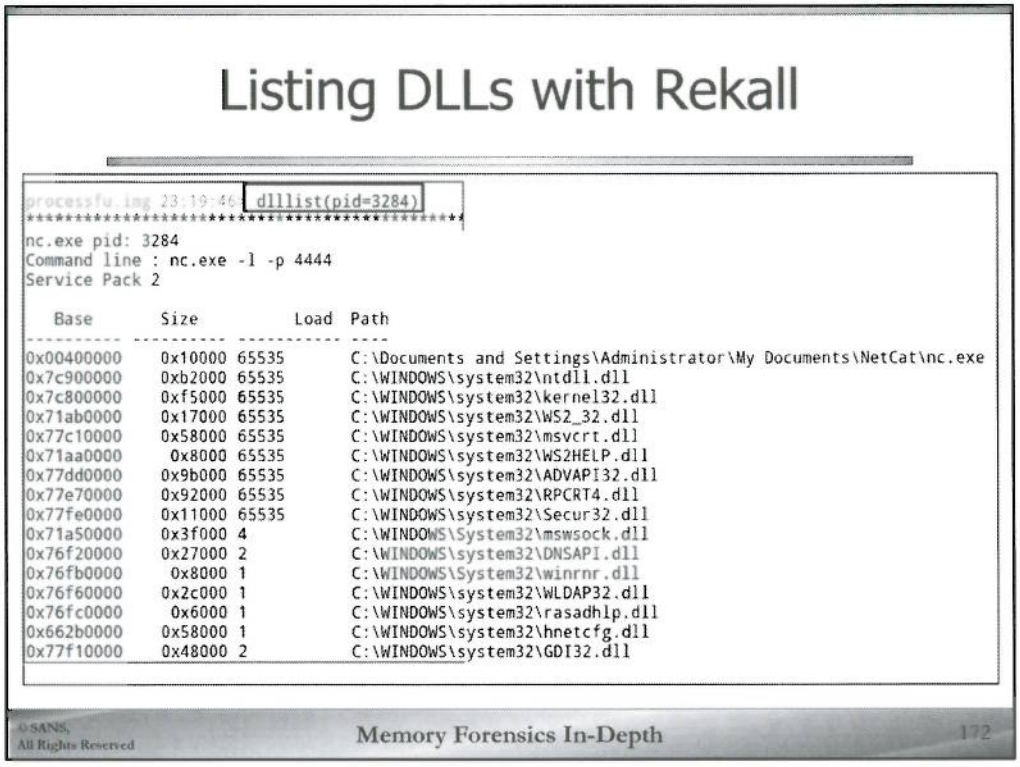


Every legitimate system process has a number of DLLs which it normally uses during its operation. When malware attempts to masquerade as a legitimate system process, one of the ways it gives itself away is that such programs load a different set of DLLs. Usually malware loads far fewer DLLs than the legitimate processes. While Windows system utilities are attempting to do everything the right way and keep the system running, malware is designed to have a small footprint and only do a few things.

For example, look at the output of our hands on command above. If you want, you can re-run it with an extra flag, --pid, to limit the output to a single process. In this example we've limited the output to pid 536, or lsass.exe. You can see there are a lot of DLLs loaded by this process. With a little command line kung-fu we can have the computer count for us. We run the command limiting the results to lsass.exe, then select only the lines with "0x" in them, or the lines which are DLLs, and then count them:

```
user@SIFT$ vol.py -f Win7x64.vmem --profile=Win7SP1x64 dlllist -p 528
|grep 0x|wc -l
Volatility Foundation Volatility Framework 2.4
63
```

This version of lsass.exe loads 63 DLLs. The exact number loaded by this program can certainly vary. It would not be surprising if lsass only loaded 50 or as many as 70 DLLs, for example. But if you encounter a program called "lsass", which only loads four DLLs, give that a second look!

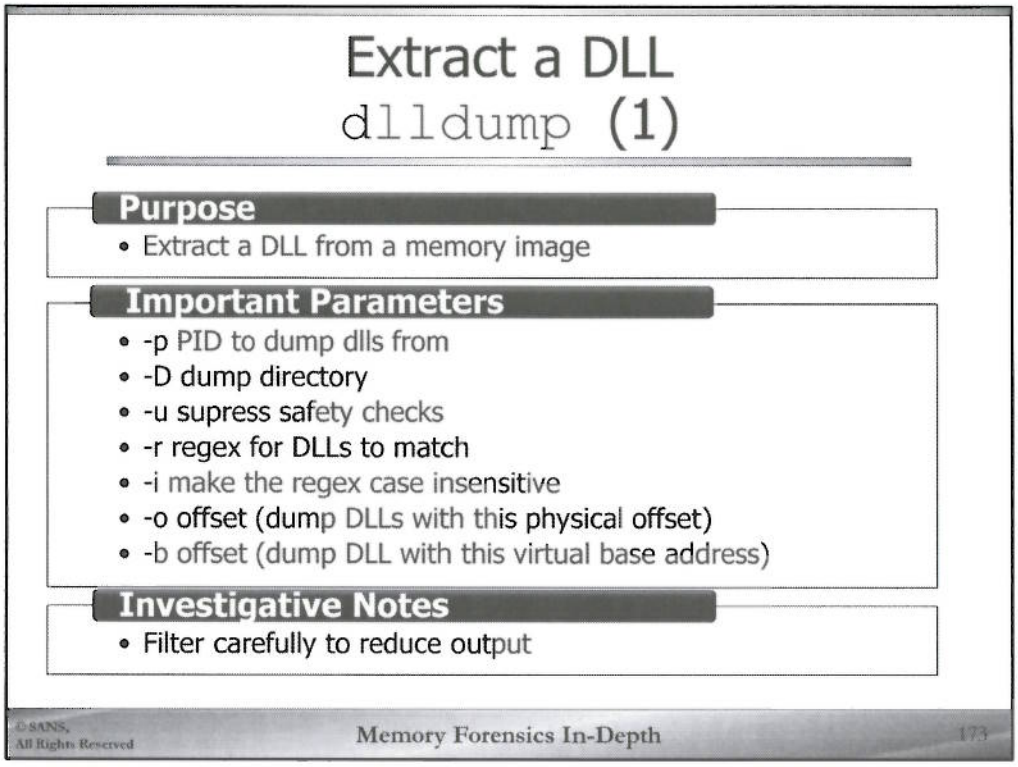


Running a dlllist plugin in Rekall can be done in the interactive shell, as shown above using dlllist and specifying which process identifier to filter on, or in a separate command as seen below. Since walking dll lists requires first enumerating processes, Rekall includes the five different options of listing processes seen in the pslist plugin:

--method {Handles,CSRSS,PsActiveProcessHead,PspCidTable,Sessions} [{Handles,CSRSS,PsActiveProcessHead,PspCidTable,Sessions} ...]

Method to list processes (Default uses all methods)

```
sansforensics@siftworkstation:/cases/exercise2$ rekall -f processfu.img dlllist --pid 4092
**************************************************
winpmem_1.4.exe pid: 4092
Command line : winpmem_1.4.exe processfu2.img
Service Pack 2

  Base      Size       Load Reason/Count      Path
---------- ---------- ------------------------------ ----
0x00400000    0x25000 65535                  C:\Documents and Settings\Administrator\Desktop\winpmem-1.4\winpmem_1.4.exe
0x7c900000    0xb2000 65535                  C:\WINDOWS\system32\ntdll.dll
0x7c800000    0xf5000 65535                  C:\WINDOWS\system32\kernel32.dll
0x77dd0000    0x9b000 65535                  C:\WINDOWS\system32\ADVAPI32.dll
0x77e70000    0x92000 65535                  C:\WINDOWS\system32\RPCRT4.dll
0x77fe0000    0x11000 65535                  C:\WINDOWS\system32\Secur32.dll
```

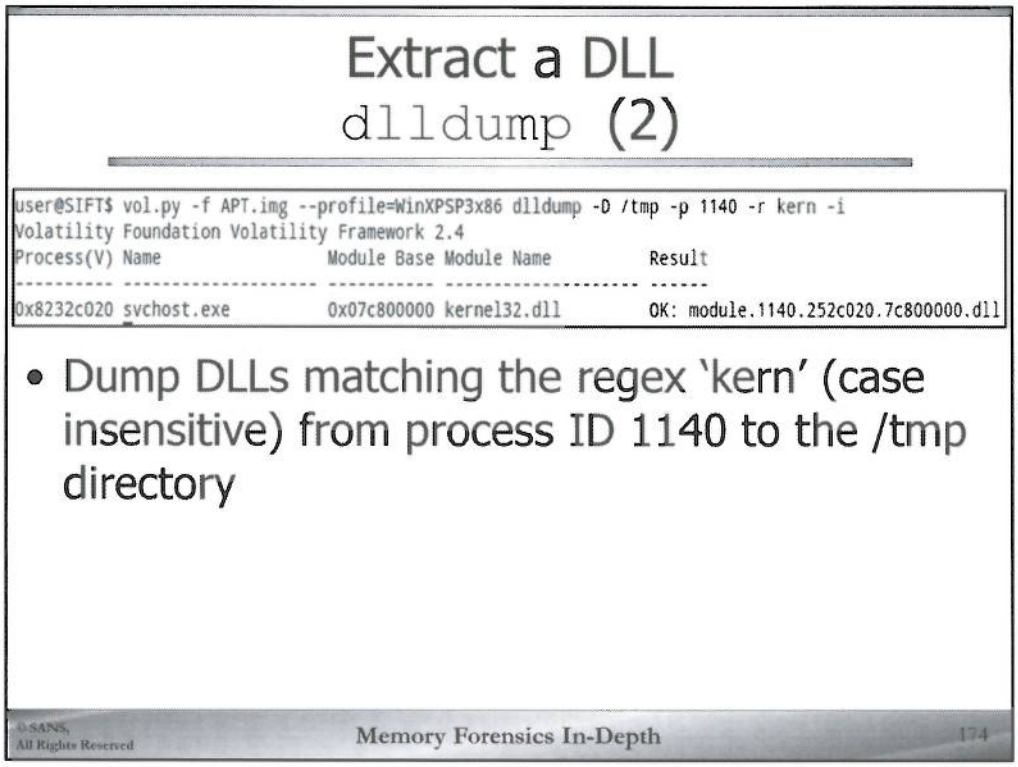


The dlldump plugin is used to dump DLLs from a given process (or all processes if the –p option is not specified). The extracted DLLs can be scanned with online virus scanners or other file scanners. Additionally, strings can be run against the extracted DLLs to provide context to the potential purpose of the DLLs.

Note that in many cases the base address of the DLL is paged out (not present in the memory dump). When this happens, the PE header is not present. Because the PE header is not present, the layout of the file on disk cannot be determined. Further, without a PE header, file checkers that depend on the header would not be able to parse the file. For this reason, if the base of the DLL is paged out of main memory, dlldump will not output the file. You'll need to use the volshell method we discussed earlier if you REALLY need the portions of the file still paged into main memory.

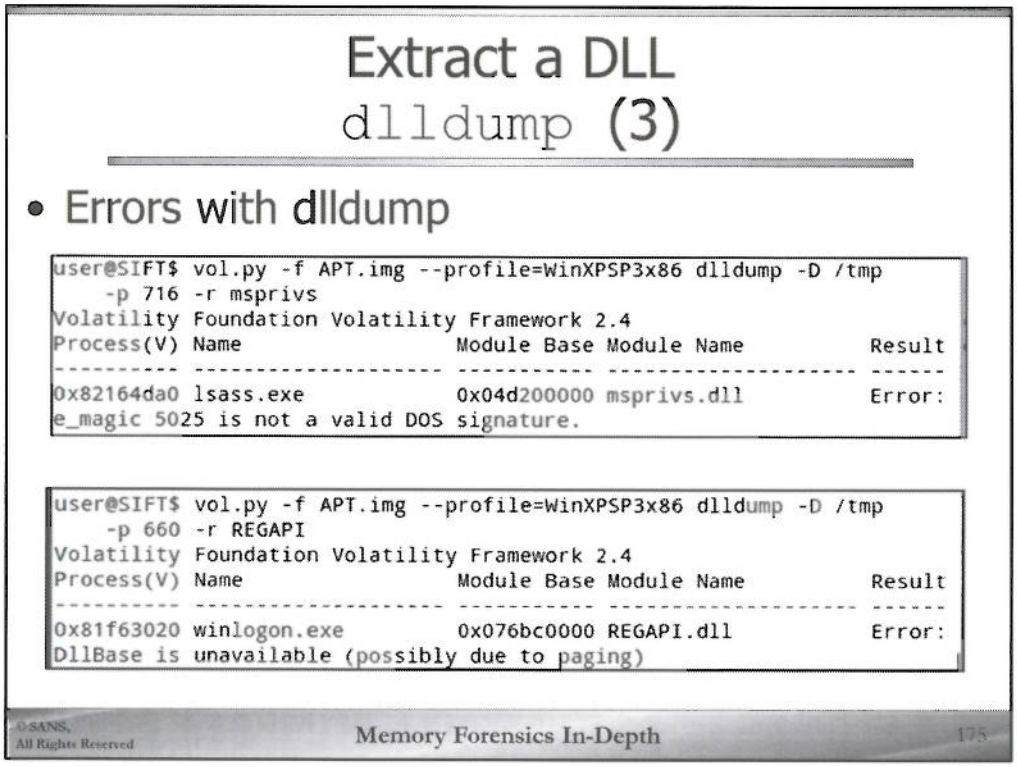
Even if the DLL Base is paged in, dlldump may still have a problem dumping the DLL. The plugin checks to ensure that the DLL has a valid signature at the beginning of the file. If the signature is invalid, dlldump will not extract the DLL. Use the volshell manual dumping technique discussed earlier to dump the DLL.

The dlldump plugin is implemented in volatility/plugins/dlldump.py.



This command dump DLLs matching the regex 'kern' (-r kern), case insensitive (-i), from process ID 1140 (-p 1140) to the /tmp directory (-D dump).

```
user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 dlldump -D /tmp -p 1140 -r kern -i
Volatility Foundation Volatility Framework 2.4
Process(V) Name              Module Base Module Name          Result
---------- ------------------- ----------- -------------------- ------
0x8232c020 svchost.exe         0x07c800000 kernel32.dll         OK: module.1140.252c020.7c800000.dll
```



This slide shows two potential errors that you may experience using dlldump. The first error shown occurs when there is a bad file header located at the base address for the DLL. The second error occurs when the DLL base address is paged out.

Error when the DLL does not have the appropriately formatted file header:

user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 dlldump -D /tmp -p 716 -r msprivs

Volatility Foundation Volatility Framework 2.4

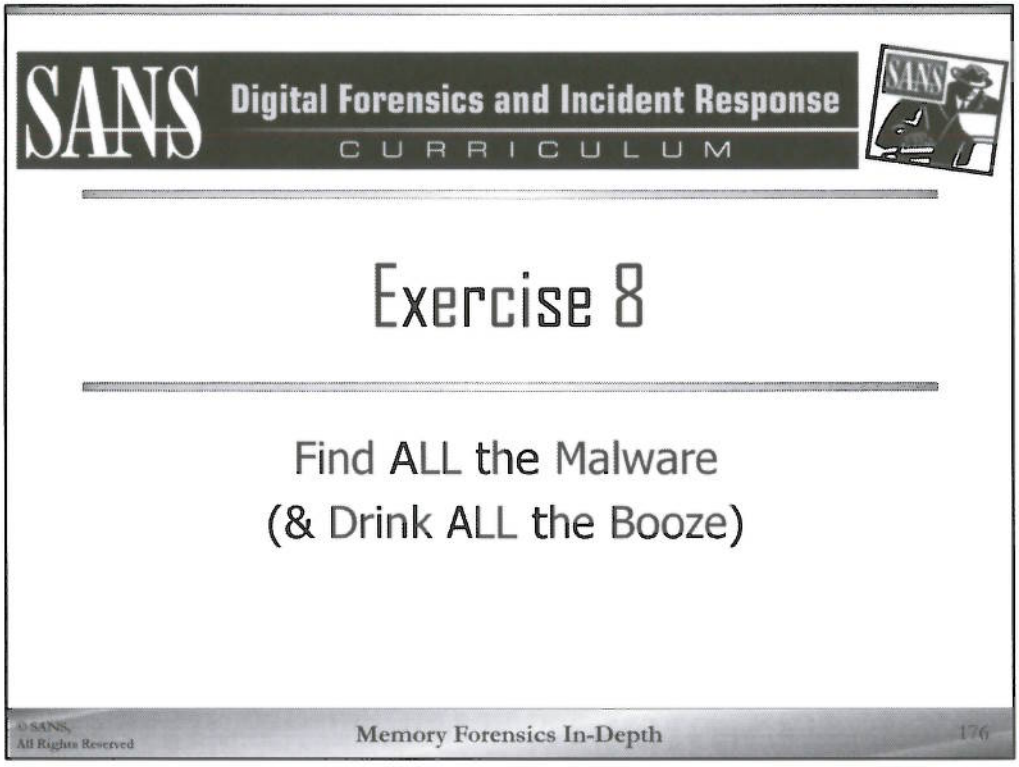
Process(V) Name	Module Base	Module Name	Result
0x82164da0 lsass.exe	0x04d200000	msprivs.dll	Error: e_magic 5025 is not a valid DOS signature.

Error when the DLL Base address is paged out of physical memory:

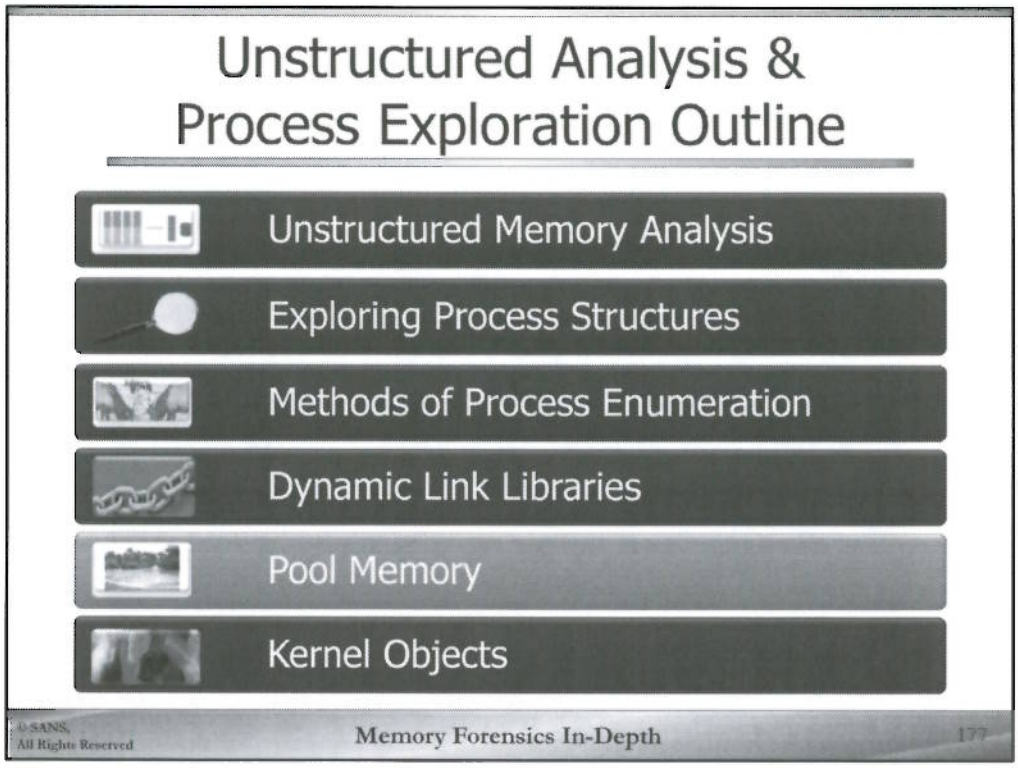
user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 dlldump -D /tmp -p 660 -r REGAPI

Volatility Foundation Volatility Framework 2.4

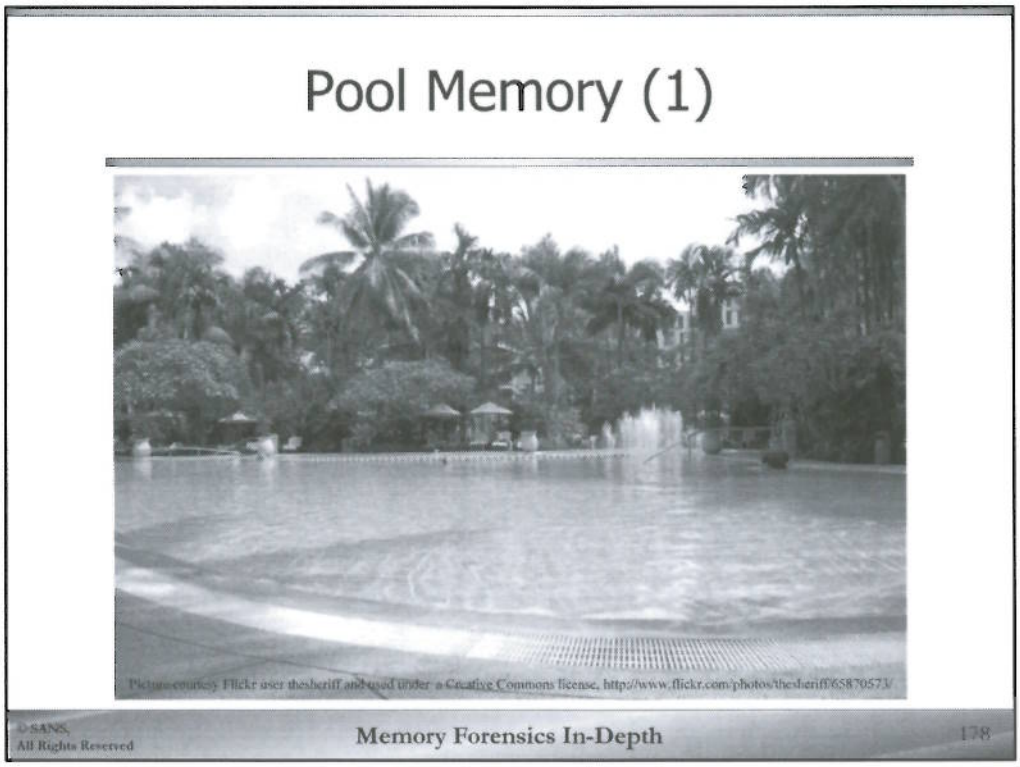
Process(V) Name	Module Base	Module Name	Result
0x81f63020 winlogon.exe	0x076bc0000	REGAPI.dll	Error: DllBase is unavailable (possibly due to paging)



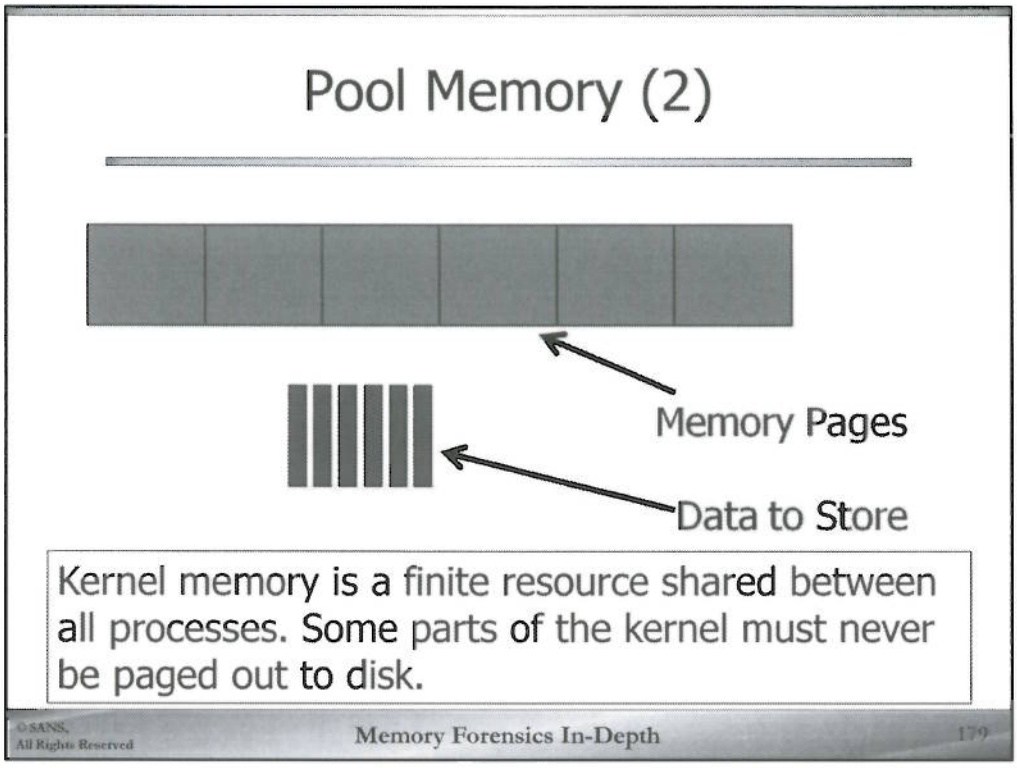
This page intentionally left blank.



This page intentionally left blank.

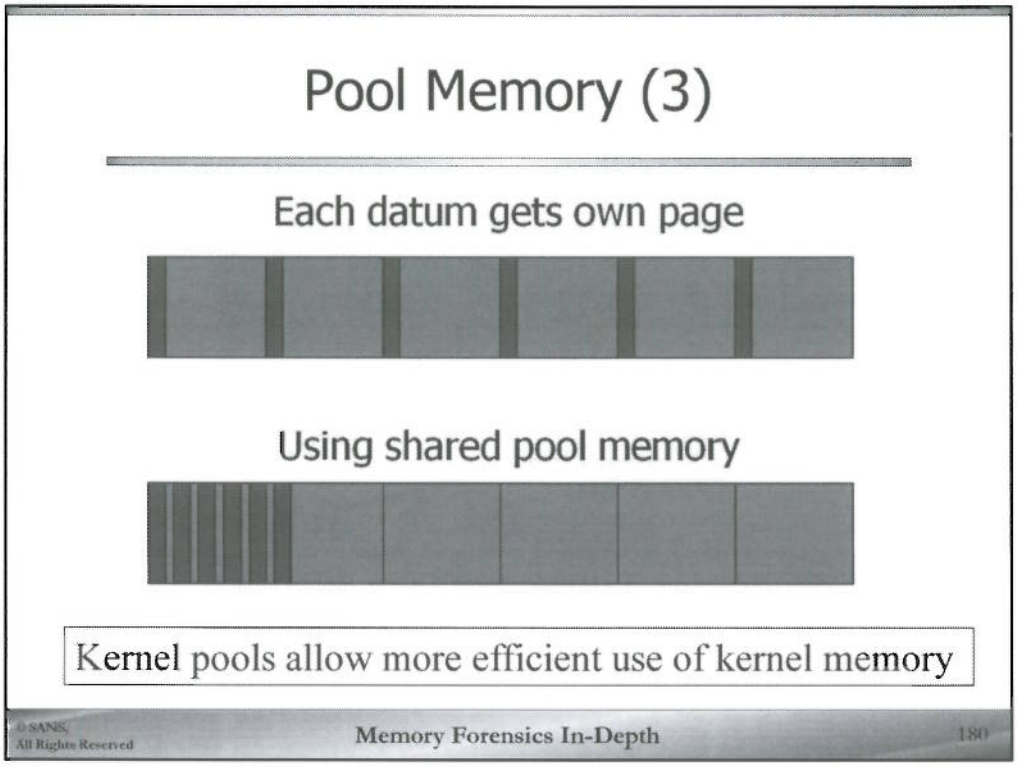


The first subject we'll tackle is pool memory. The operating system needs to allocate memory to hold data structures. There are lots and lots of these little data structures which need to be kept in memory. They are created and destroyed frequently, and to keep the system running responsively, they may need to be kept in physical memory (i.e., not paged out). These structures hold information on processes, threads, file handles, network connections, and so on. For efficiency, all of these structures are "pooled" together.

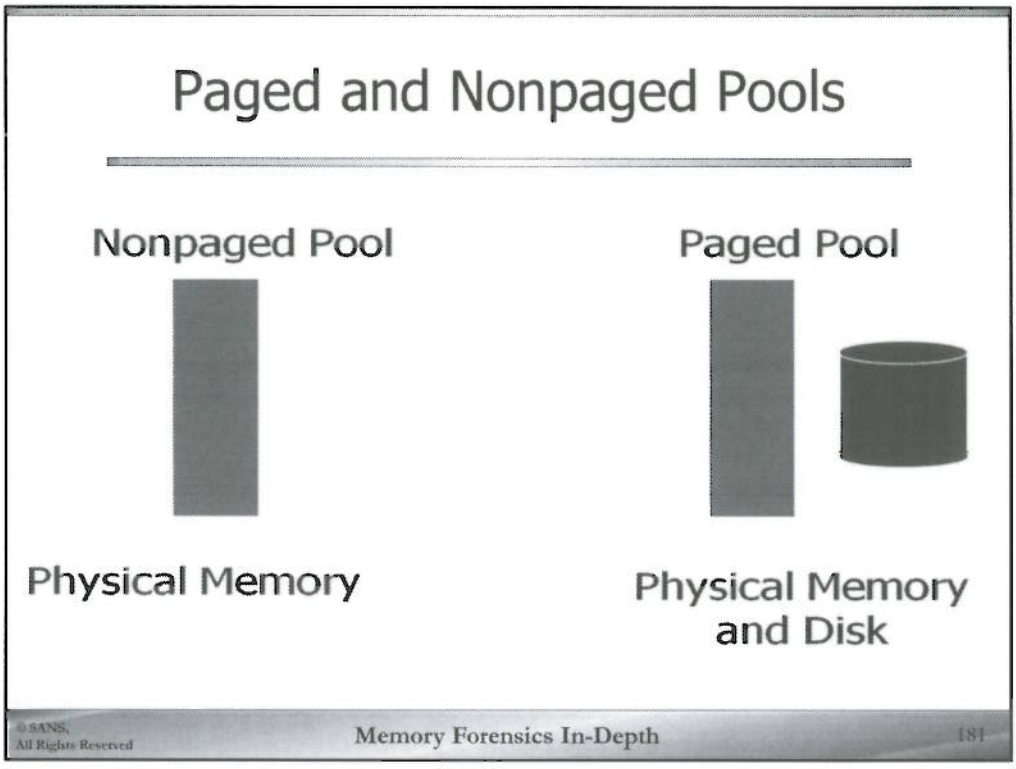


Both virtual and physical memory are allocated as whole pages. Allocating a whole page for each of these little structures would be wildly inefficient. Some of these structures are as small as 16 bytes. The standard memory page is 4KB. Allocating a 4KB page to hold just 16 bytes would be ridiculous. Along with the wasted space, there would be the overhead of constantly allocating and deallocating memory, which takes both time, and memory.

This picture shows a set of memory pages and some chunks of data the kernel would need to store. The data is much smaller than a page of memory.



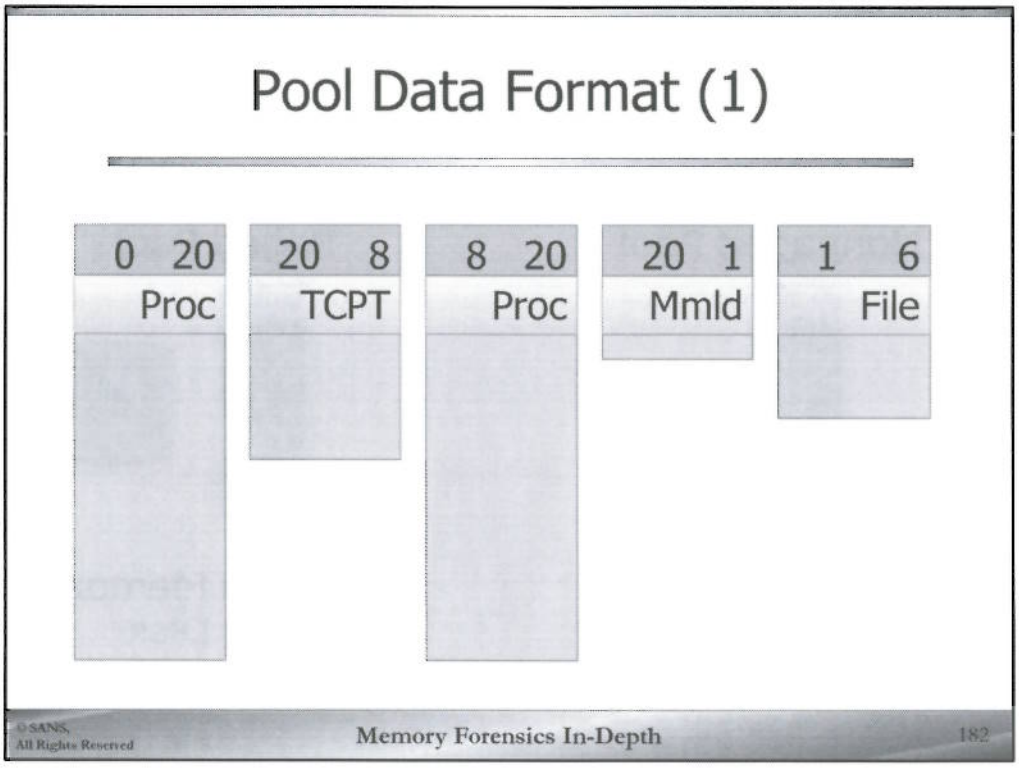
The top figure shows what would happen if the operating system allocated a whole page for each of the small structures we saw on the previous page. To avoid this kind of waste and the overhead of constantly dealing with memory allocations, the Windows memory stores data in pools. These pools are regions of virtual memory which are allocated by the kernel at boot. The pools start at a fixed size, but can grow if necessary. The bottom figure shows the same data structures as the top figure, but with them stored together in a shared pool. There is only a small amount of "wasted" space in there, which is the overhead necessary for administering the pool memory.



The Windows kernel has two kinds of pool memory, paged and non-paged. Data stored in the paged pool can be paged out to the disk if necessary. Data in the non-paged pool never gets paged out to the disk.

Some kernel structures can be paged out without a problem, but some cannot. For example, the operating system can't page anything in when dealing with a hardware interrupt (e.g., receiving a network packet, a keystroke has been typed, etc.) The structures which need to be available at all times are thus stored in the non-paged pool. This includes the basic structures for processes, threads, file objects, and mutexes. The secondary structures for these processes can be paged out.

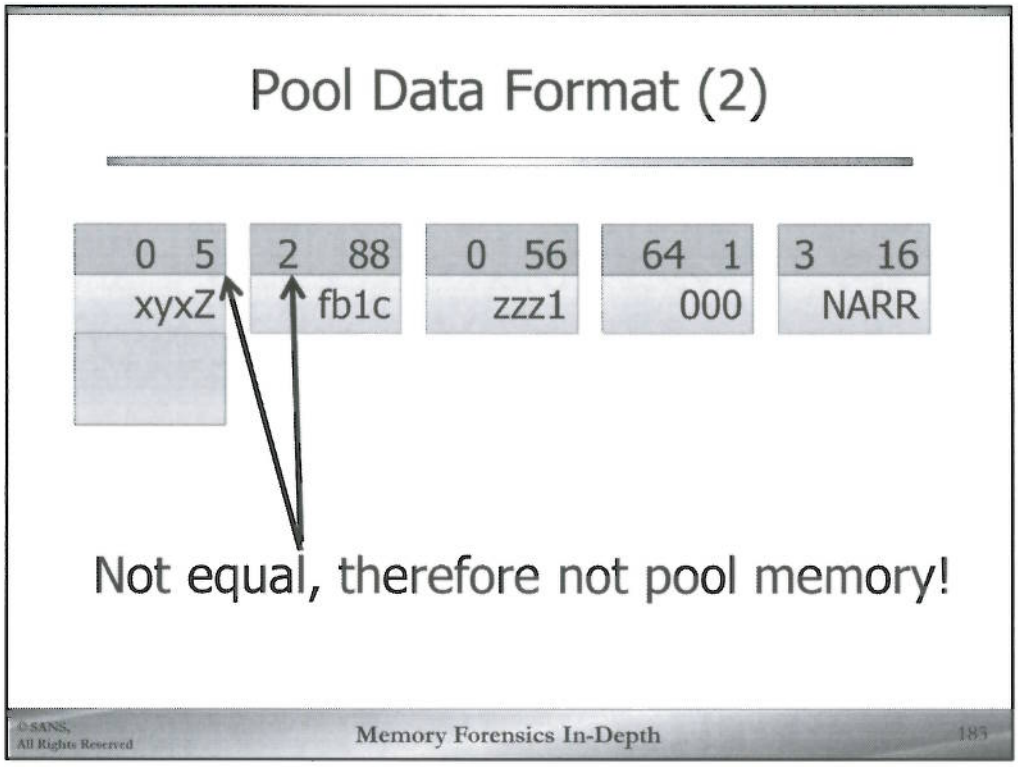
```
processfu.img 19:20:47> plugins.pools
----------------------> plugins.pools()
      Type        Index   Size      Start      End      Comment
-------------------- ----- ---------- ---------- ---------- -------
NonPagedPool      0         2949120 0x813ed000 0x893ed000
PagedPool         0         0 0xe1000000 0xf1000000
PagedPool         1         3571712 0xe1000000 0xf1000000
PagedPool         2         3485696 0xe1000000 0xf1000000
PagedPoolSession 0          0 0xbb800000 0xbbbfffff Session ID 0
```



The pool data format allows the operating system to store data of various sizes. Each item in the pool has a header. The header contains a tag which identifies the type of data, the size of the previous block of pool memory, and the size of the current block of pool memory.

In the example above, we see a representation of five entries in pool memory. Each of them has a tag which identifies the type of data being stored. The number of the left side of each header is the size of the previous entry, and the number of the right side of each header is the size of the current entry. For the first entry in the pool, the previous size is zero.

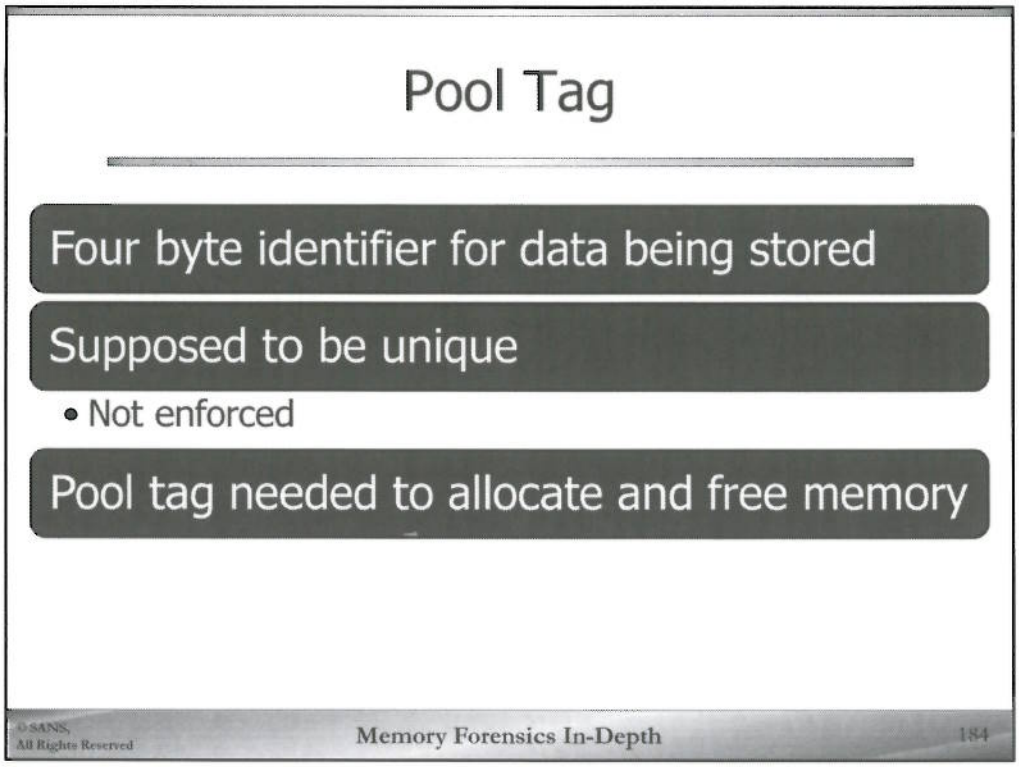
Looking at this picture, we can see that it represents a page of pool memory. In each pool header, the size of the previous entry matches the previous pool header's size of the current entry.



We can use these size values to help us identify frames which are part of pool memory. For any frame we examine, we can assume it was part of a pool unless proven otherwise. To test that assertion, we begin at the start of the frame and get the value for the next block of pool memory. We skip that many bytes ahead, and assume we've landed on another pool header. But if the value in that header for previous size is not the same as the value we just saw for the next size, then the frame cannot possibly be part of pool memory.

In the picture above, we see some potential pool headers. The 'next' value of the first entry is 16. But looking ahead that many bytes, we find the 'previous' value of the next potential header is two. These values should be the same, but they're not, which means that this frame cannot possibly be part of pool memory.

Doing this kind of validation on every frame in the memory image would allow us to find more frames of pool memory than a direct walk. That is, we could find the start of the pools in the memory image and follow them from beginning to end. But by examining each frame manually, we could find frames which were once part of the pool but have been set aside, or pages which were hidden from the operating system. We'll discuss this more in the next section, searching vs. scanning.

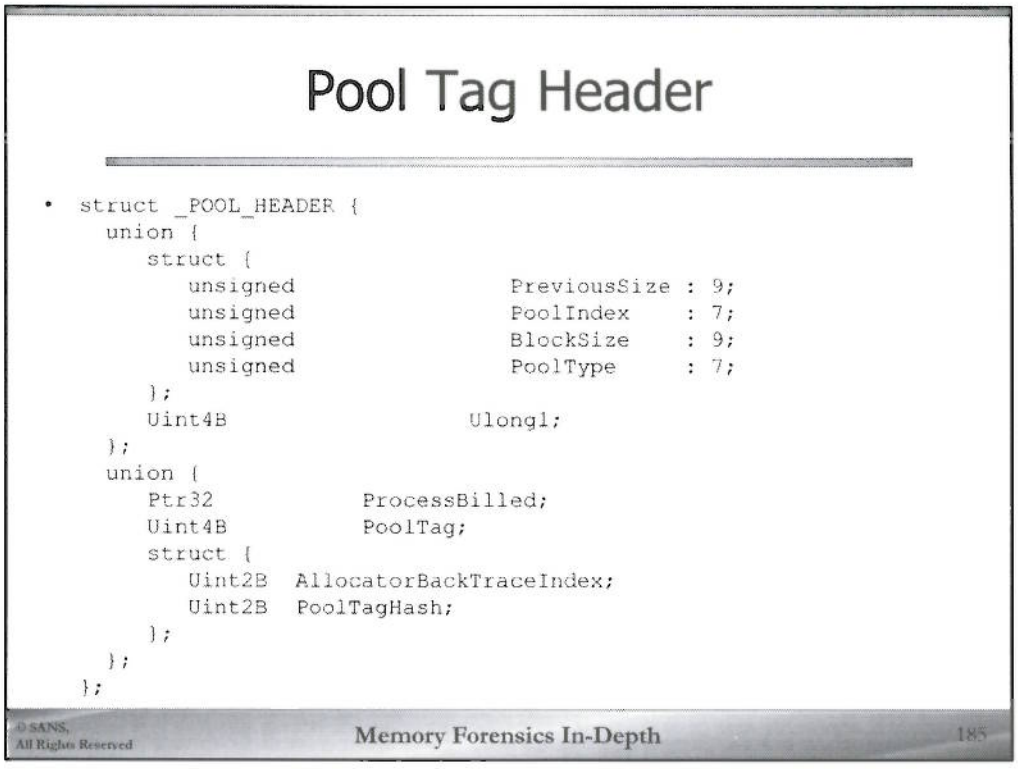


Each pool header has a four byte tag which is used to identify the type of data being stored in the entry. They are supposed to be unique. That is, if the block of pool memory has been tagged for process information, it should contain process information. But this restriction is not enforced. An operating system component, or a driver, can allocate a block of pool memory and request any tag it wants.

Calling functions can request any pool tag they want, or none at all. This goes for operating system components and third party code. If the calling function does not request a pool tag, Windows instead uses the space for a pointer. In particular, Windows puts a pointer to the EPROCESS block of the process which owns that chunk of pool memory.

If there is a pool tag associated with a block of pool memory, the calling function must submit that pool tag to both the function to allocate the memory and the function to release that memory. This is done as a safeguard, to help driver developers avoid accidently releasing memory which belongs to somebody else. Because a bug in a driver can bring down the whole system, it is vitally important to make sure drivers are free of bugs. Memory allocation bugs can be difficult to track down, so any help to those developers is quite welcome.

As a side note, once a driver is loaded, there are no restrictions on what memory it can free. If a third-party driver was programmed to do so, it could free the memory being used by the operating system. This would, of course, immediately crash the system, but it's an important reminder that at the operating system level, there are no safeguards against that kind of attack.



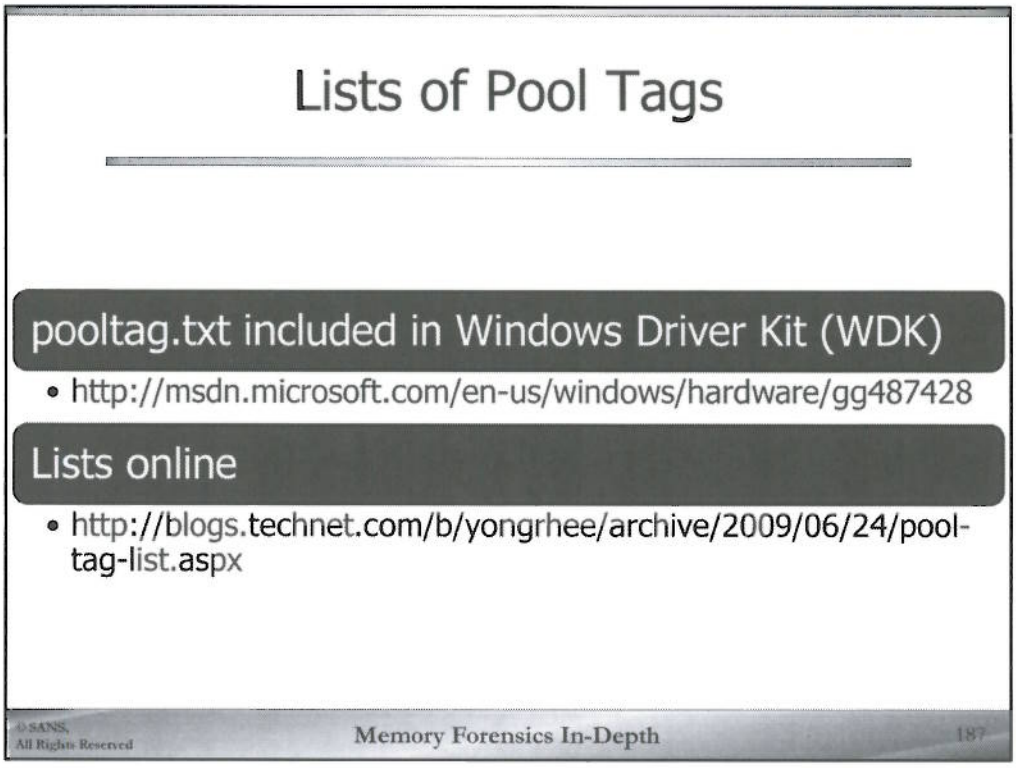
Here is the structure of a pool tag header. This is the definition of a structure in the C programming language. (The kernel is written in C.)

```
struct _POOL_HEADER {
  union {
     struct {
        unsigned                PreviousSize : 9;
        unsigned                PoolIndex    : 7;
        unsigned                BlockSize    : 9;
        unsigned                PoolType     : 7;
     };
     Uint4B                  Ulong1;
  };
  union {
     Ptr32           ProcessBilled;
     Uint4B          PoolTag;
     struct {
        Uint2B  AllocatorBackTraceIndex;
        Uint2B  PoolTagHash;
     };
  };
};
```

The structure starts with a union, or the clumping together of two different structures. The first two bytes of the structure can either be viewed as a Uint4B--a single 32-bit number--or a series of bitfields. The first nine bits correspond to the size of the previous entry, the next seven to a pool index, the next nine bits for the size of the current entry, and the last seven bits for the type of pool (e.g. paged, non-paged, etc.).

Both the previous and next size are given in units of pool memory allocation. On 32-bit operating systems, this is eight bytes. So if the pool size was 0x50 (i.e. 0b0 0101 0000), the actual size of the pool allocation would be 0b10 1000 0000. (To multiply by eight, shift the bits three places to the left. Each shift left represents a multiple by two.) The result is 0x280, which is the size of the pool memory in bytes.

The next field in the pool tag header is another union. This one contains either the virtual address of the EPROCESS for the process which requested the pool memory, or a pool tag. The process billed is used if no pool tag was requested.

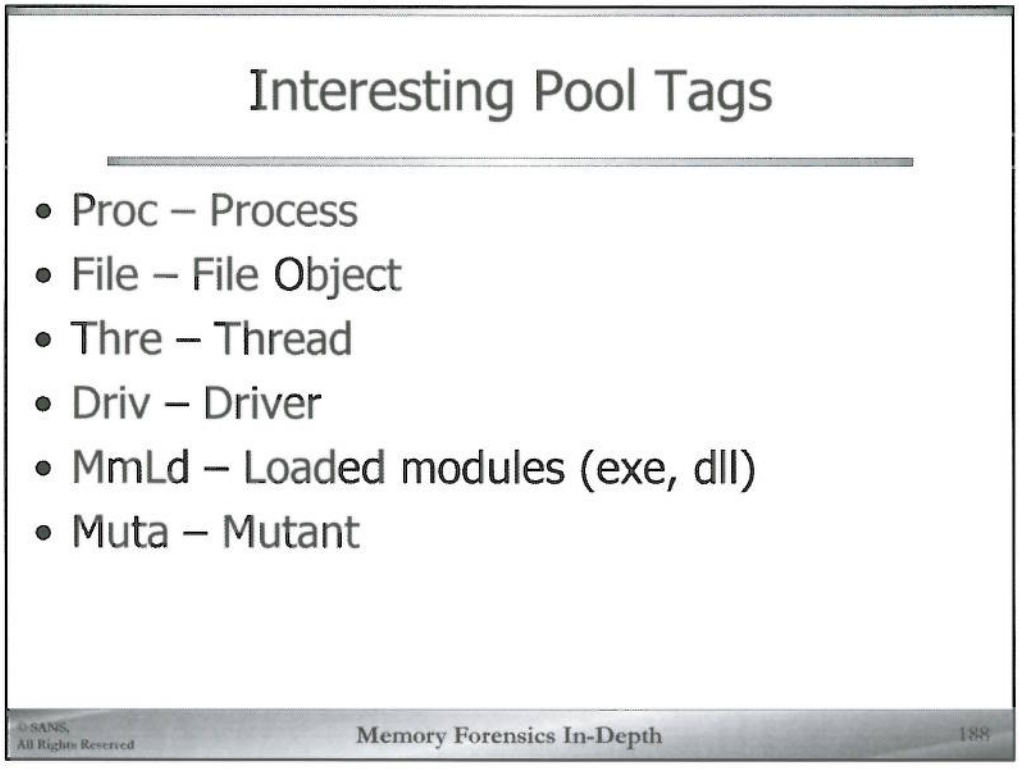


Driver developers and forensic examiners need to know which operating system component is responsible for any given pool tag they find in memory. Microsoft maintains a Windows Driver Kit (WDK) to help programmers write drivers. It contains many tools, debuggers, and lots of documentation. (The WDK used to be called the Windows Driver Development Kit and sometimes is referred to by the old acronym, Windows DDK.) The WDK includes a file, pooltag.txt, which lists the known pool tags used by Windows operating system components. The file is stored in the \triage directory of the installed WDK.

The WDK is *huge*. Microsoft recommends you download the installer, burn it to a DVD, and then install it. If you'd rather not go through that hassle, there are some versions of the pooltag.txt online. There's a version from 2009 at

http://blogs.technet.com/b/yongrhee/archive/2009/06/24/pool-tag-list.aspx.

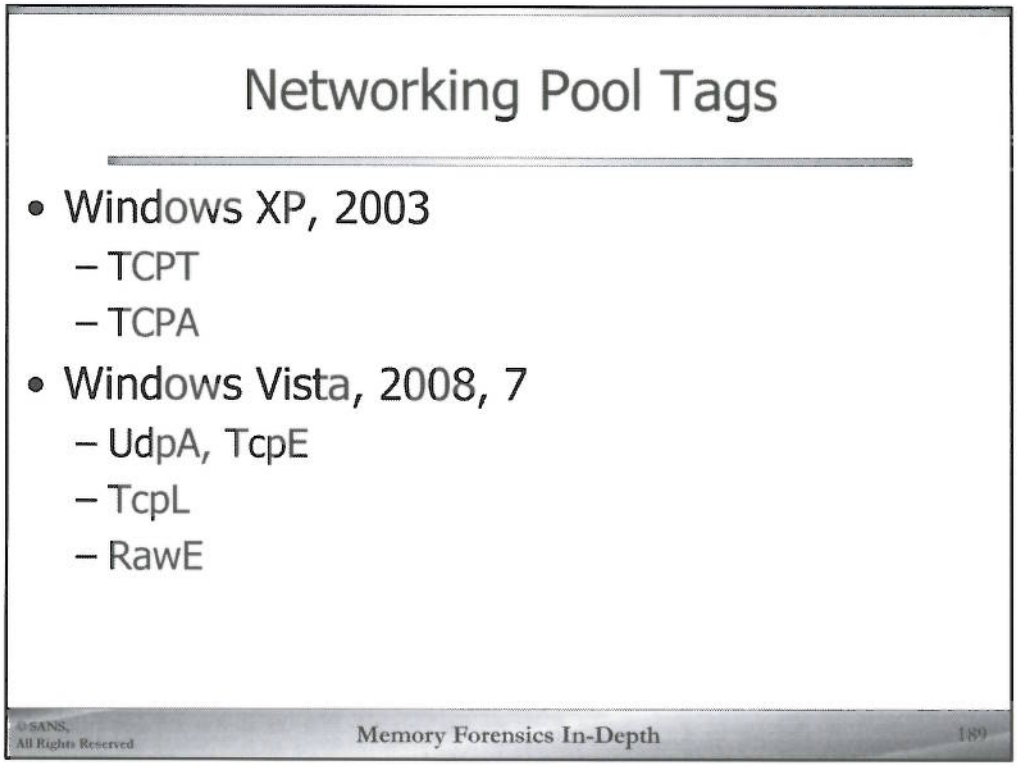
Why didn't we include a copy of the pooltag.txt file in these materials? Although the WDK doesn't cost anything, it is still copyrighted by Microsoft. Publishing their copyrighted material, to include a copy of pooltag.txt, could make Microsoft (and their lawyers) very angry. Don't do it.



Although we're not going to test fate (or our legal team) by listing all of the available pool tags, we will walk you through some of the tags which are most interesting for forensics.

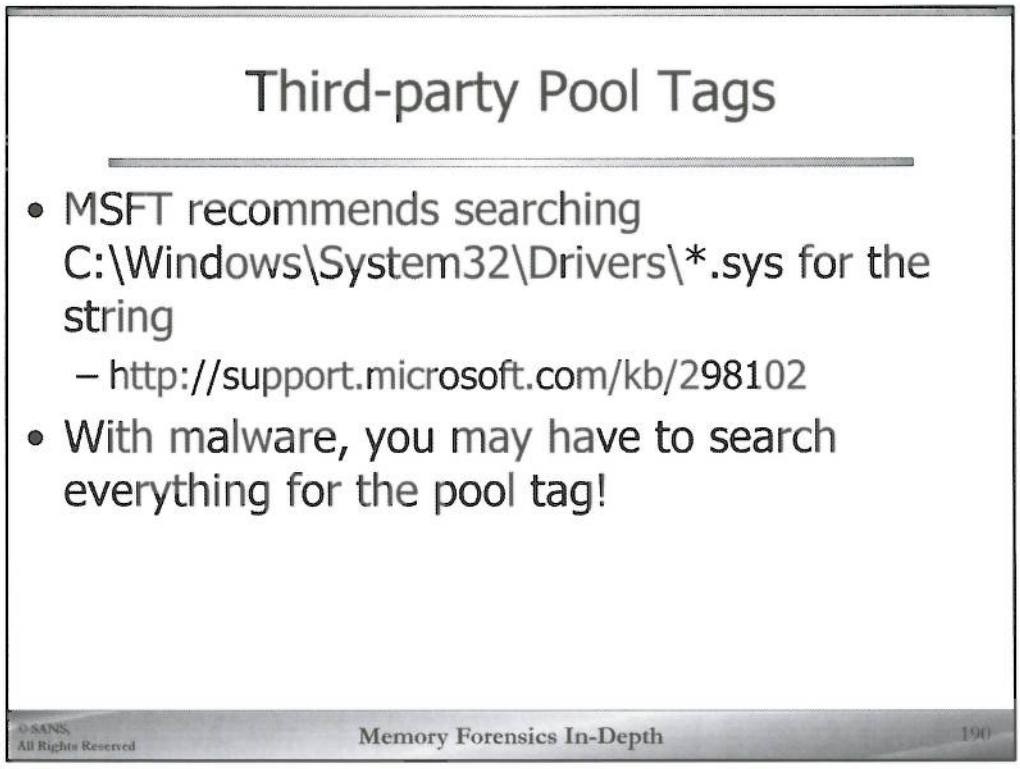
We've already mentioned Proc, for processes, but there is also "File" for file objects, "Thre" for threads, "Driv" for drivers, and "MmLD" for loaded modules. Modules are blocks of executable code, like executables and drivers. The MmLd is short for the structure which holds information about modules, which is LDR_DATA_TABLE_ENTRY. The name MmLd follows a Windows convention of naming things for the Kernel component which uses them, which in this case is the memory manager, Mm.

The last entry on this slide, mutant, is how the Windows kernel refers to mutexes. Mutuxes are a way for legitimate programs to share a resource on the system. For example, let's say that only one process can use some shared resource at a time. Each process can check if the mutex for that resource is available. If so, the process grabs it. If not, the process waits until it becomes available. Malicious programs also use mutexes, often to ensure that they don't install themselves more than once on a machine. For example, the Poison Ivy remote administration tool creates a mutex, ")!VoqA.I4 ". When Poison Ivy executes, it checks for the presence of that mutex. If it finds it, it does not install itself. (Reference: http://www.microsoft.com/security/portal/threat/Encyclopedia/Entry.aspx?Name=Backdoor%3AWin32%2FPoisonivy.gen!A)



Here are some pool tags related to network activity. TCPT and TCPA were used in Windows XP for connection objects and socket objects, respectively. The former were used for active network connections and the latter denotes processes which were bound to particular ports. Please note that although all of the pool tags are case sensitive, it's particularly important with networking pool tags. There were also pool tags Tcpt, TcPt, and TcpA.

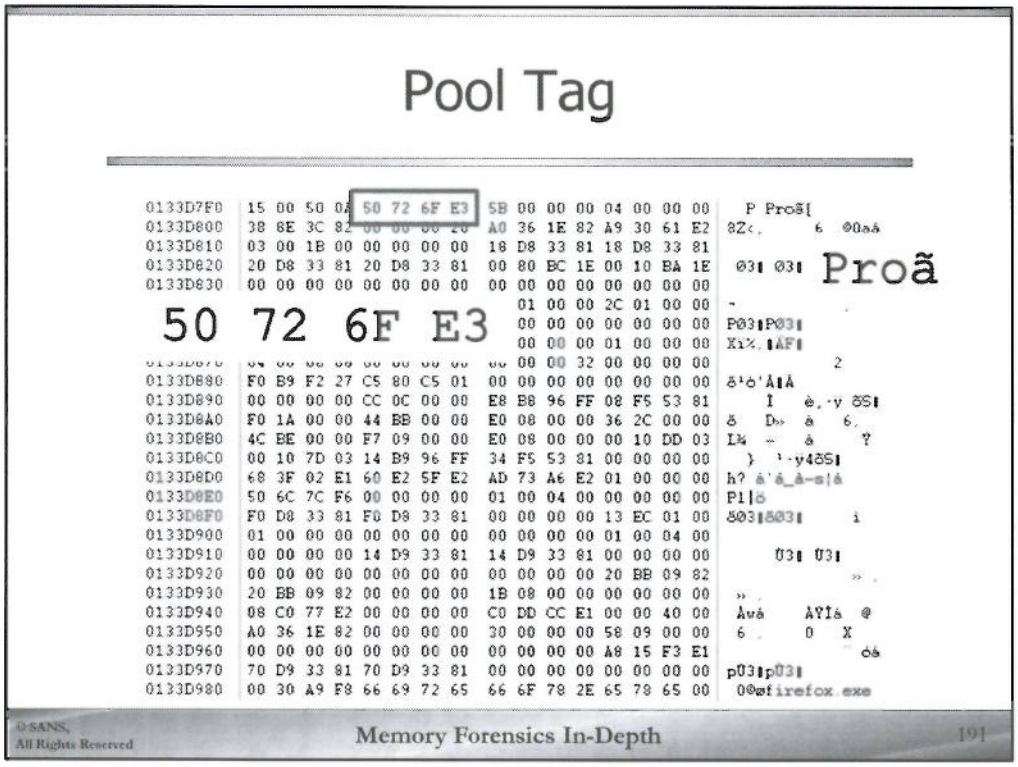
Microsoft changed how networking worked with Windows Vista. The pool tags associated with networking functions changed too. UdpA and TcpE refer to UDP endpoints and TCP endpoints, respectively. TcpL refers to things which are listening for TCP connections, and RawE refers to raw socket endpoints—network connections not done through the kernel's interface.



During your analysis you will find pool tags from third-party drivers in memory images. Some of these will be legitimate data from hardware vendor drivers, anti-virus engines, and so on. But you will probably also encounter pool tags created by malicious software. How can you determine which driver created which pool tag?

Microsoft recommends searching the files in \Windows\System32\Drivers for the pool tag in question. That's a good start for legitimate drivers. That's where they should be installed, so, yes, by all means start there. But in the case of malware, you may have to do some digging to find the driver file!

There are more details on how to do this kind of searching at http://support.microsoft.com/kb/298102.



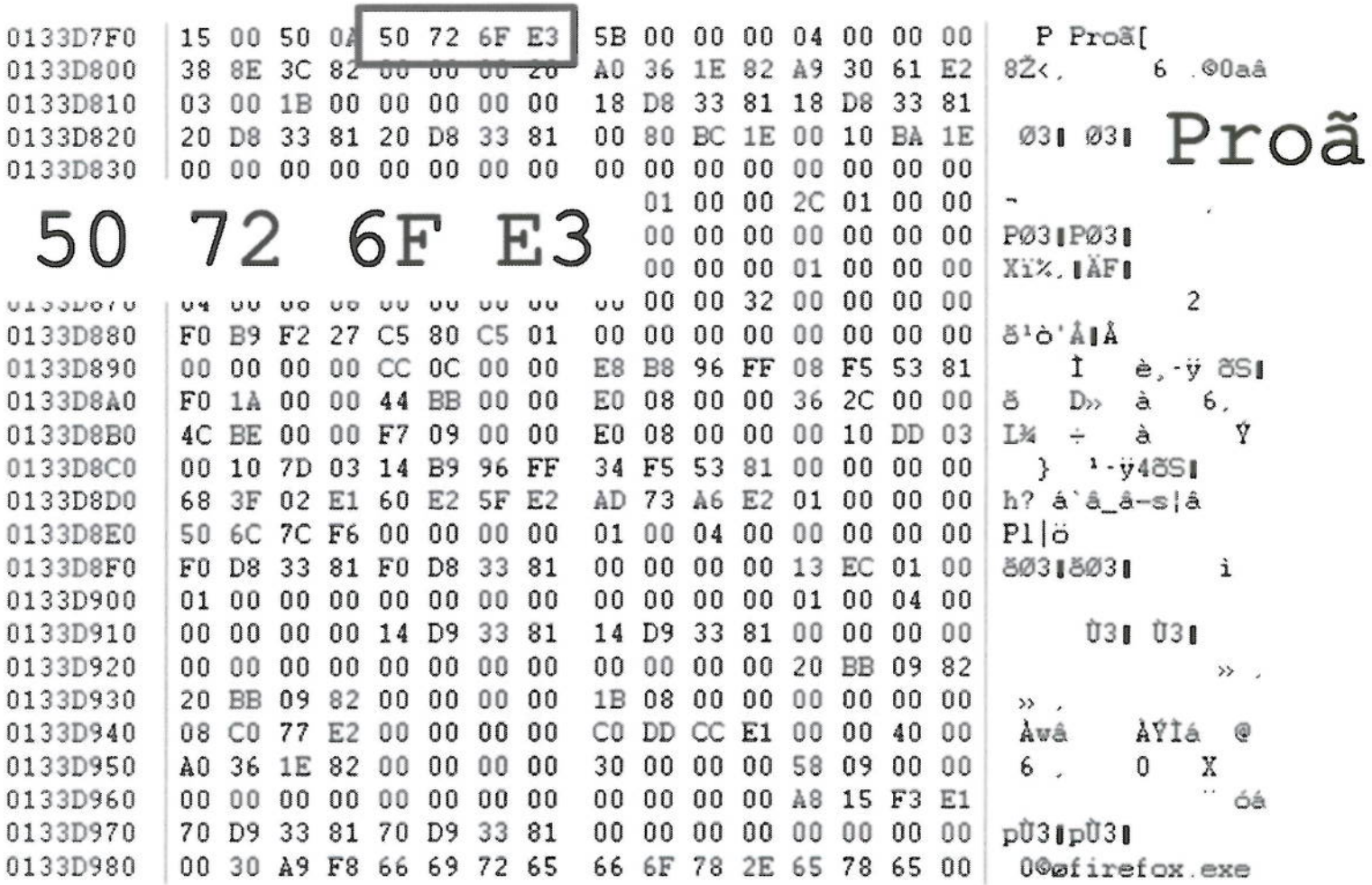
We've left out a small detail which has a big impact on searching for pool tags. This picture is the same one we used in the last section, showing the pool tag for the Firefox process in the xp-laptop memory image. The pool tag, or magic value, as we called it then, is at offset 0x133d7f0, where we see "Proã".

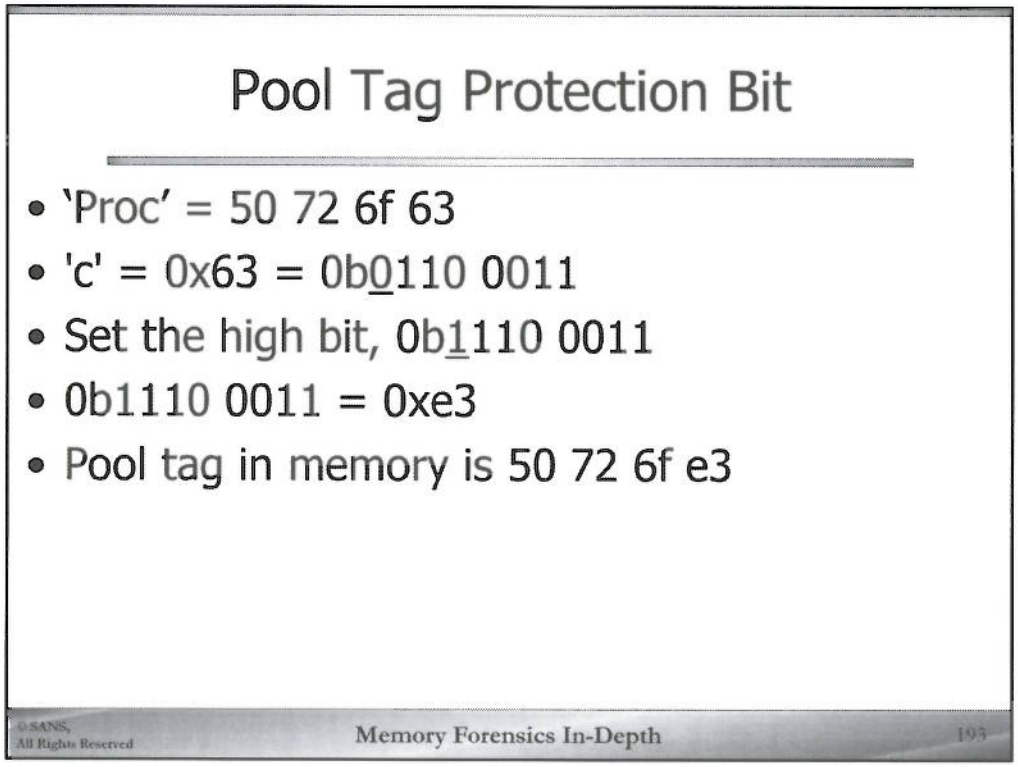
Wait a second! A few pages ago we said the pool tag for processes was "Proc". What is "Proã" doing in there?

Pool memory is allocated with the function ExAllocatePoolWithTag. That function takes several arguments, including one to indicate which kind of pool memory to use. The calling function can request paged pool, non-paged pool, and several other options. One of those options is flag, PROTECTED_POOL. If this flag is requested, Windows will set the high bit of the last character in the requested pool tag. Generally that bit is going to be zero by default. The pool tags are supposed to be readable ASCII characters, and readable ASCII characters are below 128, making the high bit a zero.

This bit, called the protection bit, does not actually offer any protection. Any driver can still free the memory stored in a block of pool memory with a protected pool tag. The only restriction is that the function to free pool memory, ExFreePoolWithTag, requires the calling function to submit the pool tag found with that block of pool memory. Because the function can just copy the tag in the block, it's not a problem at all.

References:
http://msmvps.com/blogs/windrvr/archive/2007/06/15/tag-you-re-it.aspx

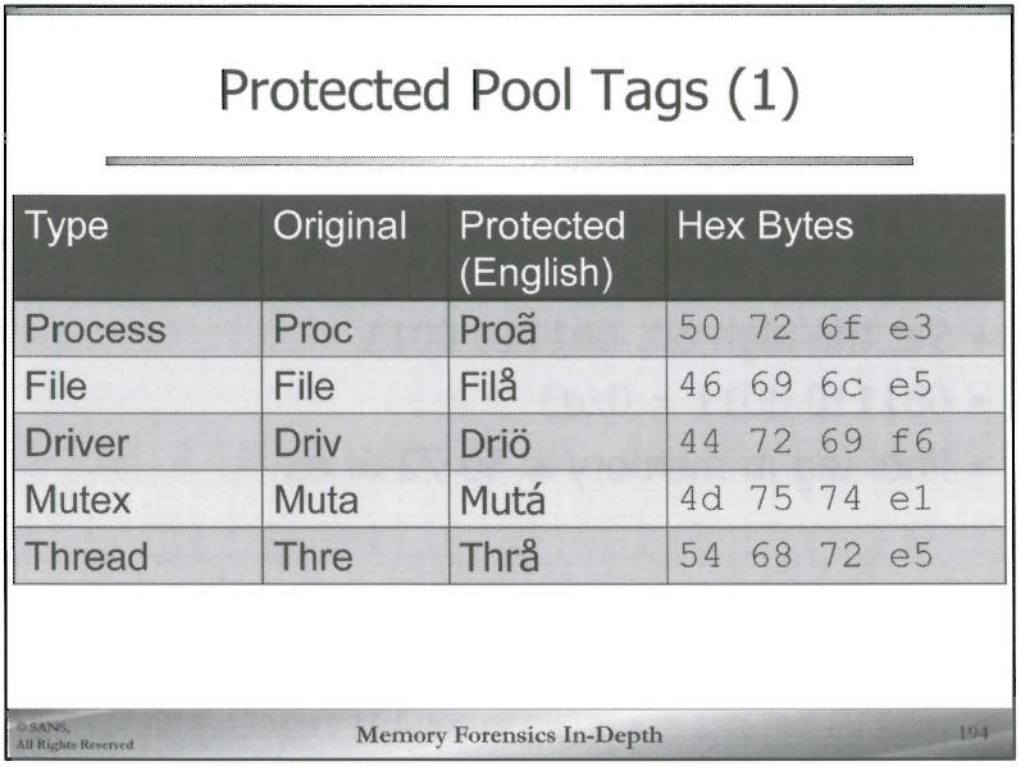

50 72 6F E3
Proã



Here's the math on how the PROTECTED_POOL flag turns "Proc" into "Proã".

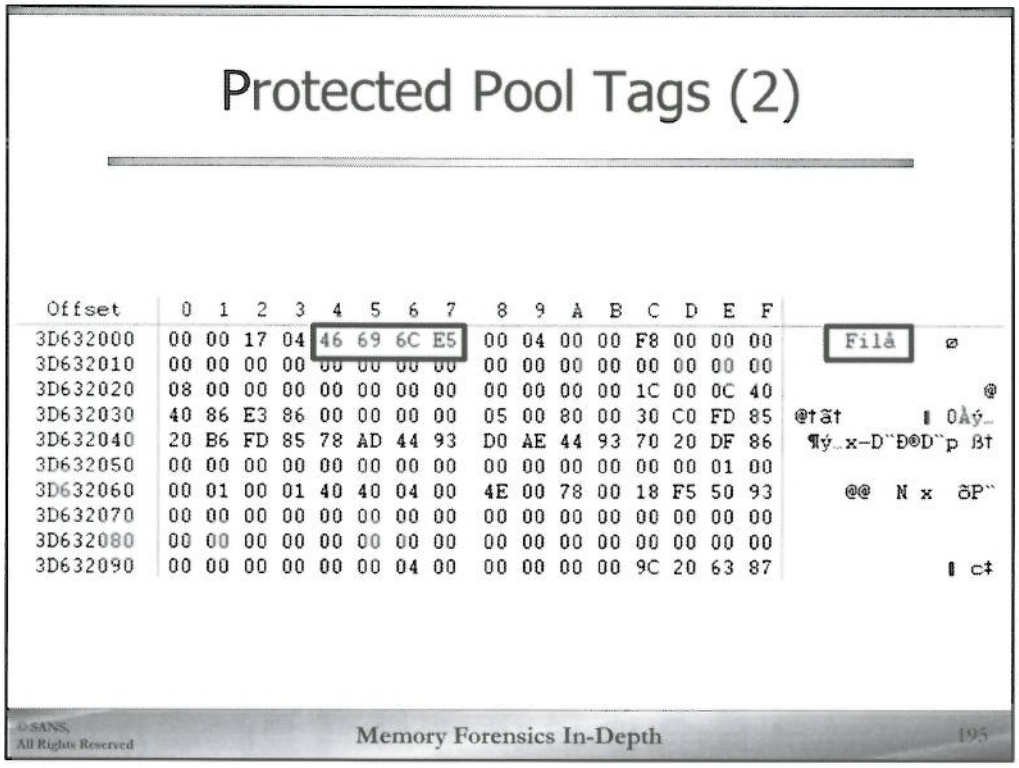
The ASCII string "Proc" is made up of the hex bytes 50 72 6f and 63. We're going to focus on that last byte, 0x63. When written in binary, we can plainly see that the high bit is zero. The high bit is always zero for printable ASCII characters. When the ExAllocatePoolWithTag function is called with the PROTECTED_POOL flag set, the function sets the high bit of the last character in the pool tag to one. We can see that changes the byte 0x63 to 0xe3.

Thus the pool tag returned by this function, and thus what we find in memory images, is "Proã".



There are several protected pool tags used by Windows. Here is a partial list of the ones which are protected and what they look like, When doing our searches, we have to search for the protected versions of these pool tags. Windows always uses the protected pool tags.

It is important to note that the string 50 72 6f e3 appears the way it does as text because we are viewing the memory image on a system using English. With other languages and code pages the character e3 appears as a different letter. When you are searching for pool tags, you should always search for the hex values!

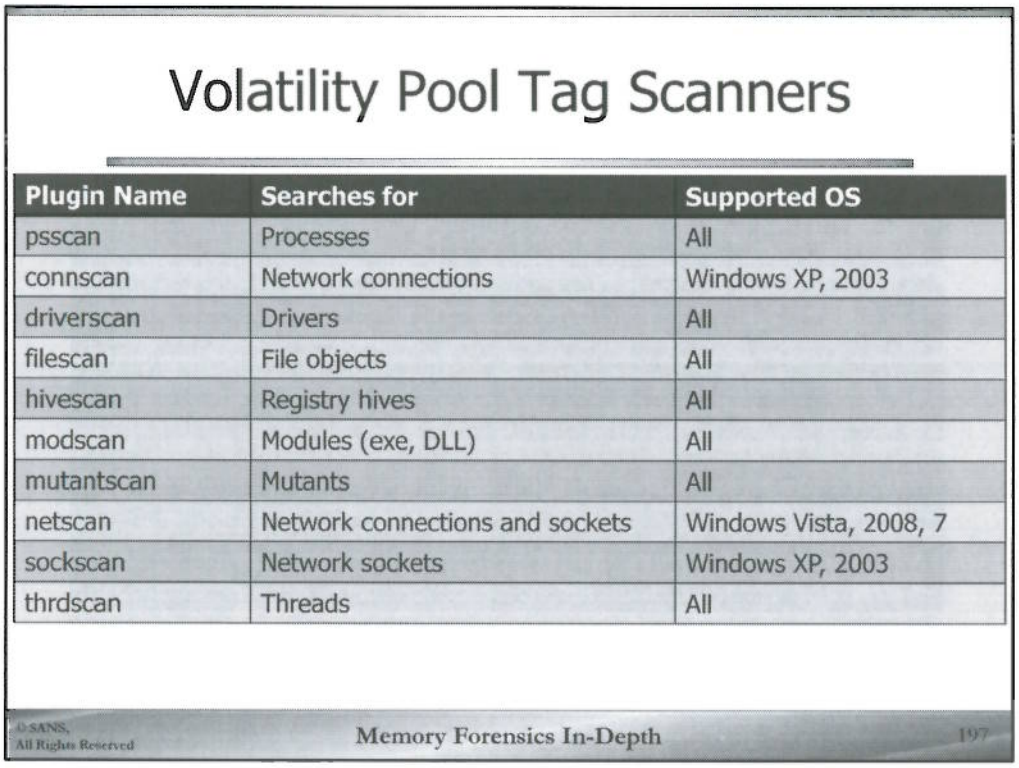


But don't take our word for it!

The screenshot above came from the win7crypto.vmem image at offset 0x3d632000. You can see the File object pool tag at offset 0x3d632004.

You can see this for yourself by firing up the Bless hex editor on your SIFT workstation. It's under the Applications menu, under the Programming section. Choose "Bless Hex Editor". From the File menu in that program, choose "Open", and then use the dialog box to open /cases/win7crypto.vmem. From the Search menu, choose "Go to offset", and enter 0x3d632000. Note that Bless does not show you the 'å' character, but you can see the byte sequence 49 69 6c e5. (Unfortunately GHex, the other hex editor on SIFT doesn't show the character to you either, nor does xxd, the command line hexdumper.)

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
3D632000	00	00	17	04	46	69	6C	E5	00	04	00	00	F8	00	00	00	Filå ø
3D632010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
3D632020	08	00	00	00	00	00	00	00	00	00	00	00	1C	00	0C	40	@
3D632030	40	86	E3	86	00	00	00	00	05	00	80	00	30	C0	FD	85	@†ã† ▌ 0Åý…
3D632040	20	B6	FD	85	78	AD	44	93	D0	AE	44	93	70	20	DF	86	¶ý…x-D"Ð®D"p ß†
3D632050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	
3D632060	00	01	00	01	40	40	04	00	4E	00	78	00	18	F5	50	93	@@ N x õP"
3D632070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
3D632080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
3D632090	00	00	00	00	00	00	04	00	00	00	00	00	9C	20	63	87	▌ c‡



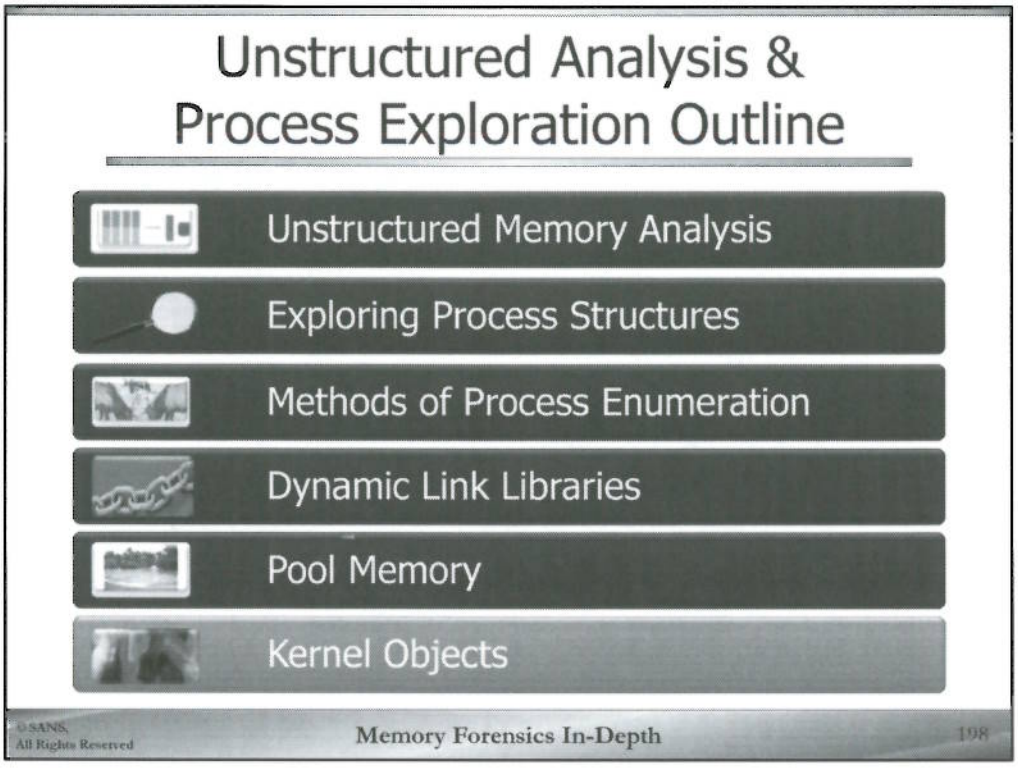
The Volatility framework contains several plugins for scanning for pool objects. These are all brute force scanners, and, as described earlier scan the entire memory image for potential frames of pool memory.

There are some definite 'gotchas' to using these plugins. First, some of the names don't correspond to what they're searching for. Searching for threads, for example, could be 'threadscan', but it's not. The letters 'thrd' also don't correspond to the pool tag for threads, which is 'Thre' before being protected to be 'Thrâ'.

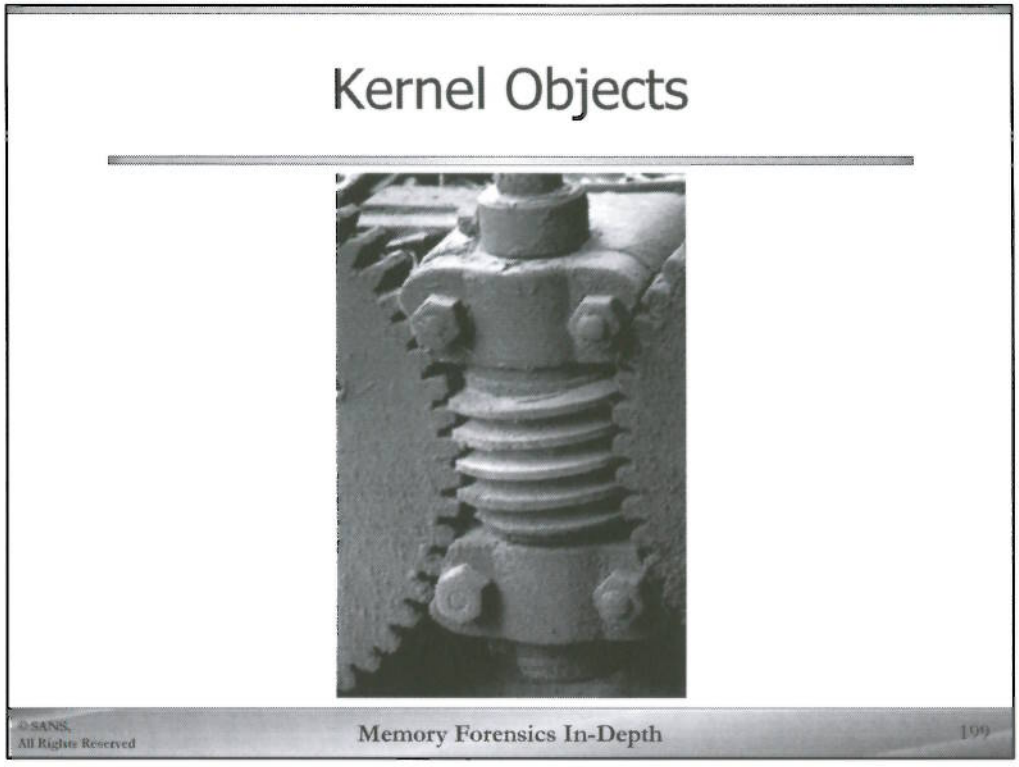
The next gotcha is that most of these plugins require the user to specify not only an input filename, but also a 'profile', detailing the operating system, service pack, and architecture used in the target system. You can get those details from the imageinfo plugin.

Assuming the user does specify the correct profile, there are times when the plugins just ignore it. The connscan plugin, for example, only supports Windows XP and Windows 2003 memory images. But the original (and as of press time, current) Volatility framework does not complain if you attempt to run connscan on a Windows 7 memory image. Running connscan, netscan, or sockscan on an unsupported system can lead to real problems, soaking up all of the available memory and not terminating. To prevent this the Volatility developers have added a check to ensure the user specified profile is supported by the plugin. If not, the plugin terminated with an error message. You can override this check if necessary, but Volatility does not, by default, prevent the user from doing something blindly stupid.

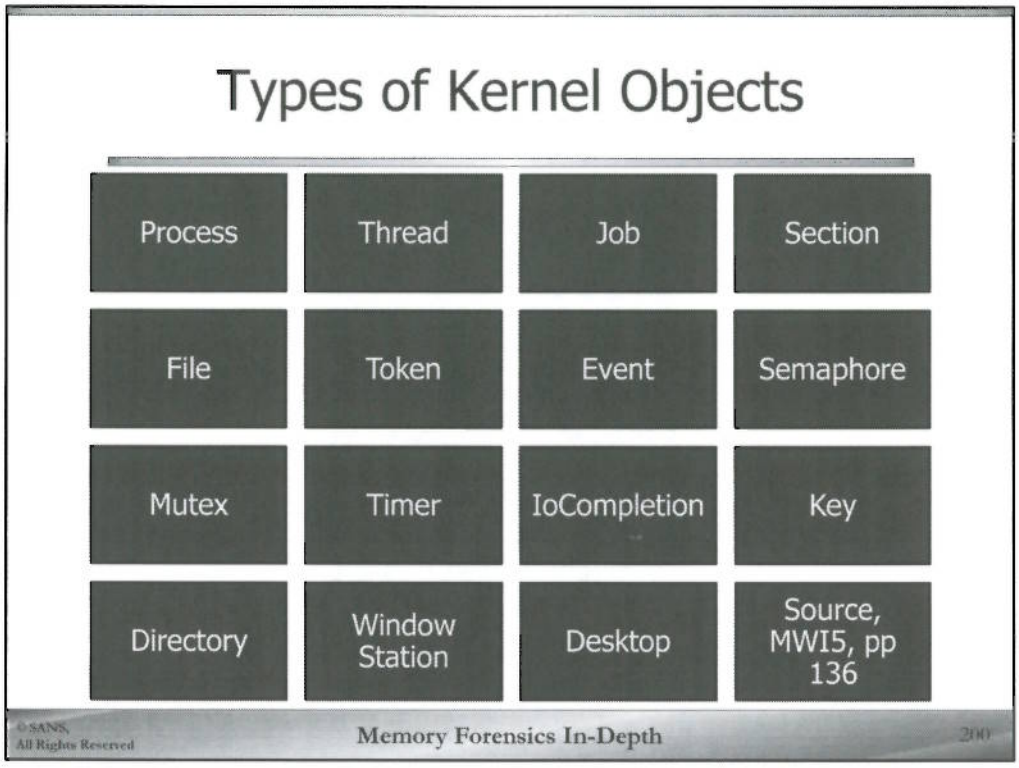
Finally, from the perspective of a programmer attempting to maintain, modify, or extend these plugins, their organization is a nightmare. Some of them are relatively self-contained. The sockscan plugin is in sockscan.py. But the process scanner, psscan, is in the file for scanning for file objects, filescan.py. You might think that searching for threads would be in there too. Nope, searching for threads is in the file for searching for modules, of course, modscan.py.



This page intentionally left blank.



Kernel objects are an operating system construct intended to provide a single interface for allocating, using, tracking, and disposing of system resources. They can hold many different kinds of data, some of which we have already covered, but some of which are brand new to us. In this section we'll explore the different kinds of data which are stored in kernel objects, what the operating system uses them for, how we can use them, and then the tools with which we can analyze those objects.



There are in fact 37 types of Windows kernel objects. Many of them are not accessible outside of the kernel components which define them, however. Here are the types of objects which you can access via the Windows API. We've already spent plenty of time talking about processes, but the rest are new.

Process – A container for executable code, DLLs, threads, etc.

Thread – An executable context inside of a process.

Job – Multiple processes grouped together for management purposes.

Section – A region of shared memory, also known as a file mapping object.

File – An opened file from the disk.

Token – A security profile for a process or thread. This includes the rights the process or thread has and limits on what it can do.

Event – Used for synchronization.

Semaphore – A counter used to control how many threads can access a resource simultaneously. These are sometimes implemented in hardware too, but here is a software construct for them.

Mutex – A counter used to ensure only one thread is using a resource at a time. A single mutex is called a Mutant.

Timer – Notifies a thread after a specified period of time.

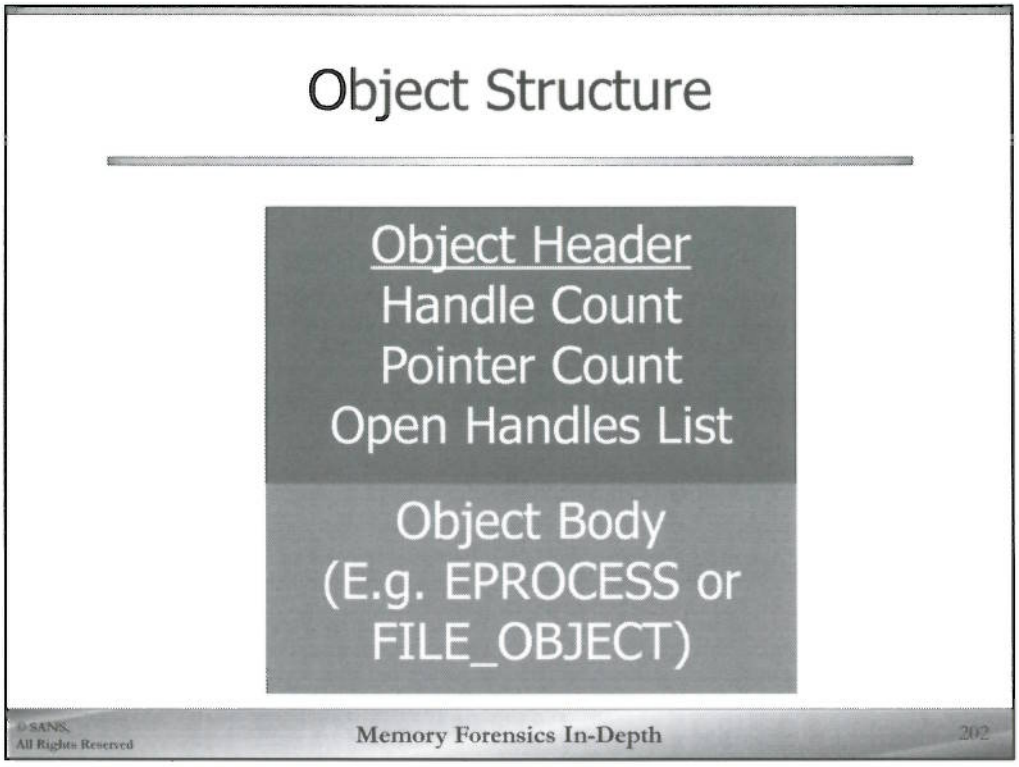
IoCompletion – Used to let threads know when I/O operations have completed. For example, when data has been read from the disk and is now available in memory.

Key – Refers to data in the registry.

Directory – Virtual directory to hold other objects.

Window Station – Contains a clipboard and desktop objects.

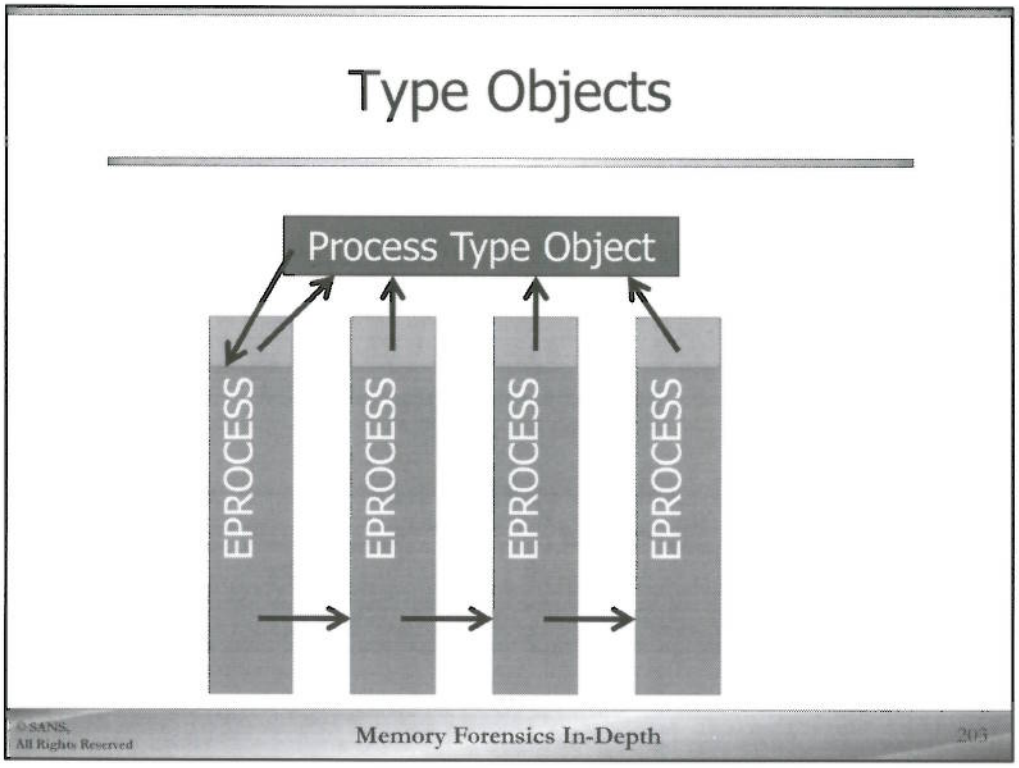
Desktop – Contains windows, menu, etc., to be displayed.



Every object has a header, which holds information common to all objects, and a body, which is specific to that type of object. The header contains things like the object's name and which directory it is stored in. (Objects don't have to have a name, and many don't.) The header also has a security descriptor, describing what can be done with the object and by whom.

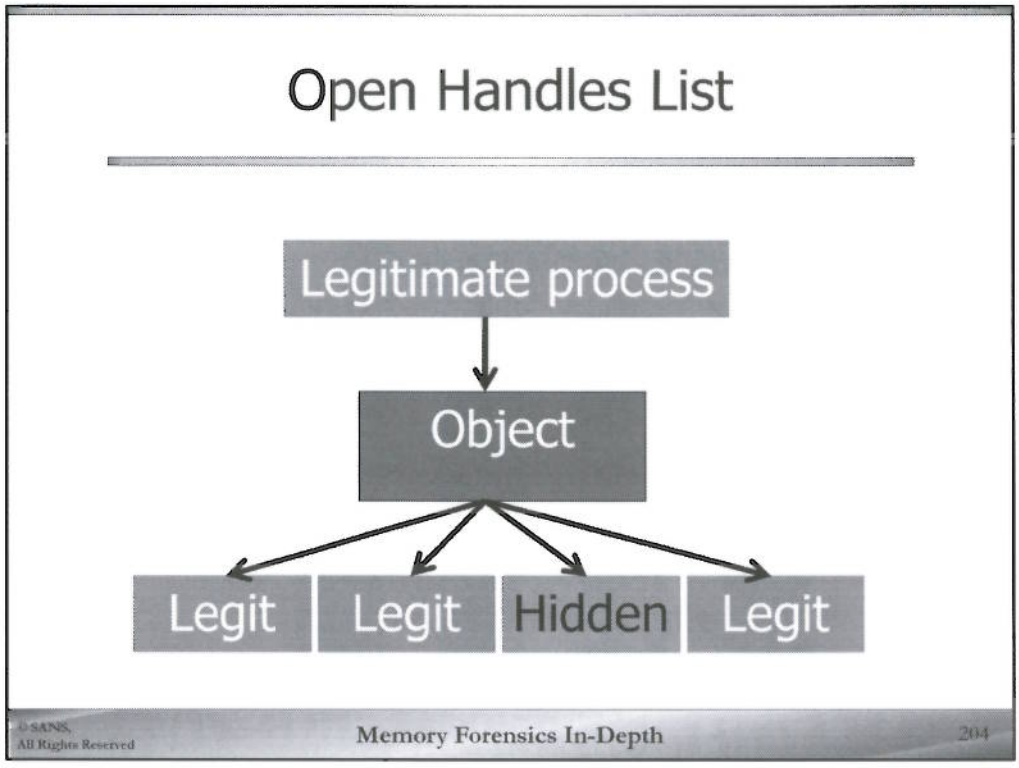
Other useful fields in the object header are the handle count, which holds the number of how many open handles there are to this object. The handle count is used to ensure that an object is not destroyed while there are open handles to it. Handles are used by user-mode processes. For example, when a program like Notepad opens a file on the disk, that program is creating a handle to that file. If another process opens that file, Windows can reuse the existing handle for the second process. In such an example, the handle count of that file would now be two. If Notepad closes its file handle, the handle count drops to one. Windows still keeps the file object, however, until the second process closes its handle too. At that point the handle count drops to zero and it's safe to destroy the object.

There's also a pointer count in the object header. Kernel mode components don't use handles, but instead reference kernel objects directly with pointers. Windows still keeps track of these pointers, however, so that it doesn't get rid of an object which is still in use. Handles serve as an abstraction barrier so that usermode programs don't mess around with kernel data structures. Kernel mode code has no such restrictions.

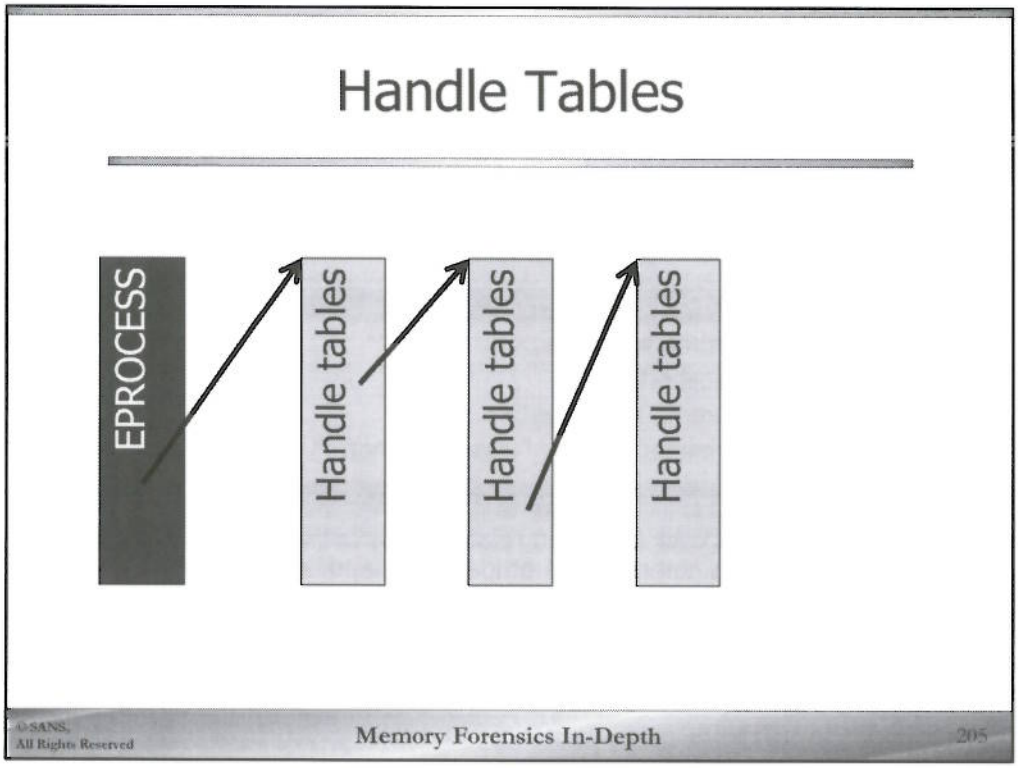


Every object of the same time points to a single object, a type object, which describes the objects of that type. For example all Process objects point to a single type object for processes. The type object contains a pointer to the list of kernel objects of that type. Continuing our example, the Process Type Object contains a pointer to the list of active processes. There are some standard usage numbers, such as the number of objects of that type which currently exist, the maximum number of this type of objects which have existed, how many handles there are to all of those objects, and so on.

But more importantly there is a pointer to another structure, an Object Type Initializer, which contains pointers to functions for handling how create, destroy, and engage in some low-level functionality with these types.



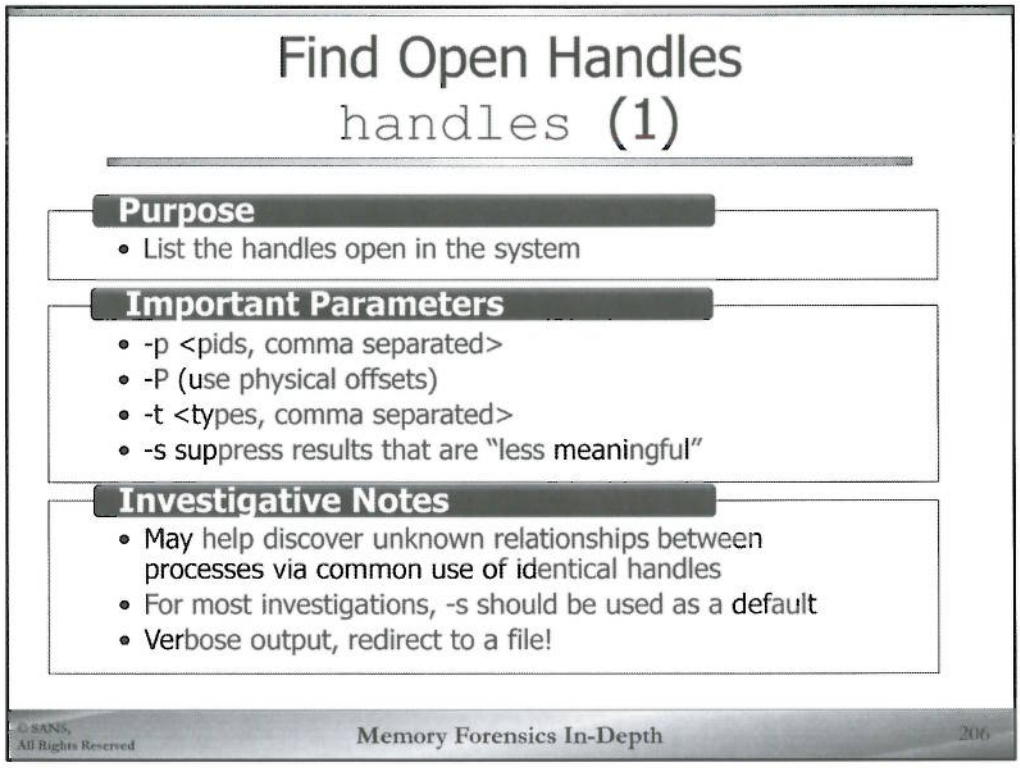
The open handles list is an optional field in the header which contains a list of the processes which have open handles to this object. It can be a method for finding hidden processes. Let's say that a process has been unlinked from the list of active processes. That process still has plenty of open handles to objects. We can find these objects, either through a brute force search or by following the handle tables from legitimate processes to them. (More on handle tables in a few pages.) If we follow the list of open handles from each object, we should end up a process we found during the walk of active processes. If we find a process which wasn't in that list, we have definitely found a hidden process! Terminated process or leftovers from a previous boot should not be pointed to by any active structure. If we find a pointer to a previously unknown process in an active structure, it means that something is definitely amiss.



Each process has a table of the handles which it holds. But, of course, it's not that simple. To maximize the number of handles a process can hold, the handle tables are built as a three-stage lookup process. Oh, you thought you got away from those messy three stage lookup processes when we moved on from virtual to physical address translation? Actually, if a design works in one area, operating system programmers are likely to use it again. In a world where even small errors can crash the system, it pays to reuse working code.

A pointer to the handle table is stored in the EPROCESS for each process. That table has entries, which can point to the top of another table of pointers, and so on down to the third level. The original handle number is split up into segments which are used as the indices into these tables. These tables are only created as needed. If a process is using fewer handles, there will be fewer handle tables.

All told this scheme allows each process to hold about 2^24, or 16 million handles.



The handles plugin enumerates the handles open on the system. The output of this plugin is VERY verbose. You should probably redirect it to a file unless you are filtering on a single PID (and possibly even then). This plugin walks the handle tables with each process, it does not scan for open processes.

The –t parameter allows you to filter on one or more types of objects. This is useful for instance if you only want to see open files or open registry keys but want to ignore everything else.

Valid types:

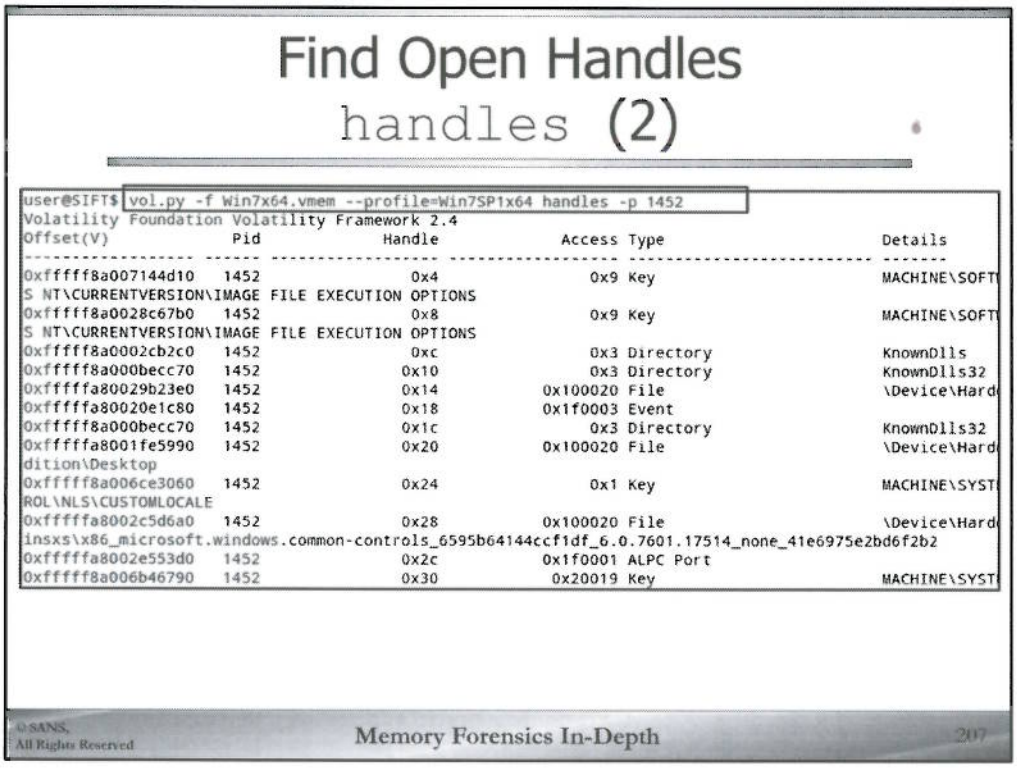
File – show file objects

Key – show open registry keys

Process – show open process handles. When a parent process launches a child process, it usually receives a handle to the child. Also, injecting processes often have an open file handle to victim processes.

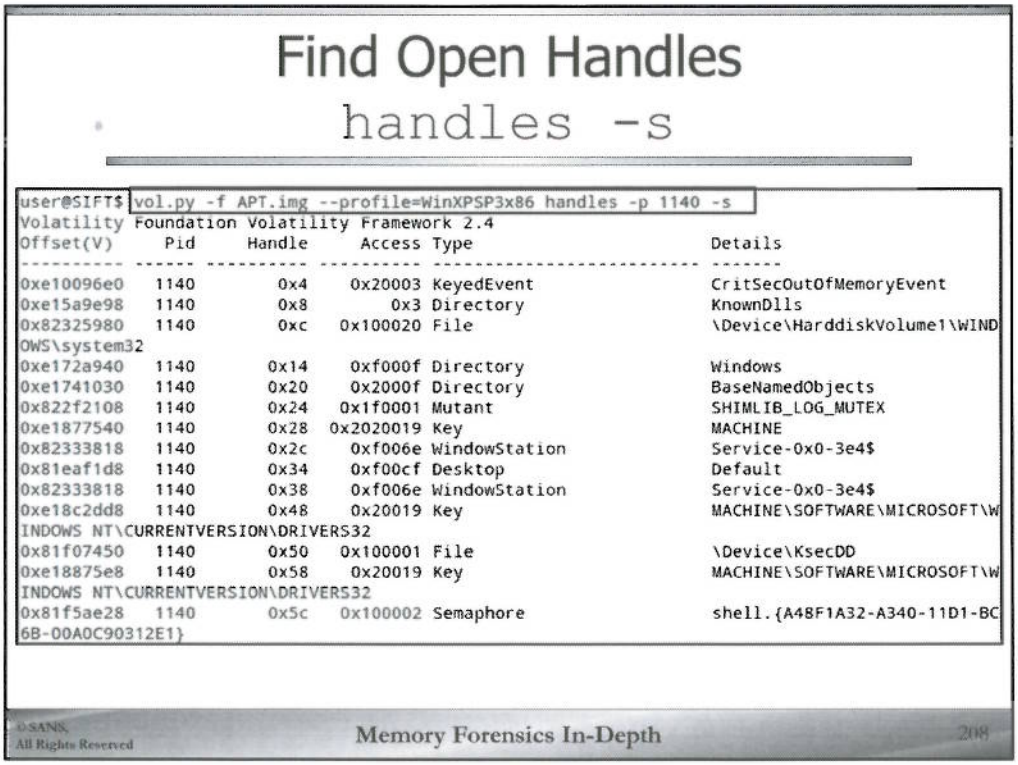
Thread – Processes have handles to their threads. This is a good way to get an idea of how many threads a given process has. In general, multiuser network servers tend to have a LOT of threads.

The –s option isn't magic. It doesn't do any real filtering of results. It just filters out unnamed handles. These are useful in some cases, but require manual follow up. If you won't be following up with your results manually in volshell or a debugger, use the –s option.



Thankfully Volatility contains a plugin to walk those handle tables for you. The plugin is called 'handles' and will display all of the open handles for each process. This can generate a LOT of output. You may find it useful to limit the output to a single process or set of processes using the --pid flag. You can specify a single pid, or a comma separated list. Here's an example using our xp-laptop memory image. If you run the command on the image, you'll get more than 9,000 lines output. But if you restrict the output to just pid 3276, the firefox.exe command, you'll see the following 190 lines:

```
user@SIFT$ vol.py -f Win7x64.vmem --profile=Win7SP1x64 handles -p 1452
Volatility Foundation Volatility Framework 2.4
Offset(V)              Pid             Handle     Access Type
Details
------------------ ------ ------------------ ------------------ ---------------------------
-------
0xfffff8a007144d10   1452                0x4                0x9 Key
MACHINE\SOFTWARE\MICROSOFT\WINDOWS NT\CURRENTVERSION\IMAGE FILE EXECUTION OPTIONS
0xfffff8a0028c67b0   1452                0x8                0x9 Key
MACHINE\SOFTWARE\MICROSOFT\WINDOWS NT\CURRENTVERSION\IMAGE FILE EXECUTION OPTIONS
0xfffff8a0002cb2c0   1452                0xc                0x3 Directory
KnownDlls
0xfffff8a000becc70   1452               0x10                0x3 Directory
KnownDlls32
0xfffffa80029b23e0   1452               0x14                0x100020 File
\Device\HarddiskVolume1\Windows
0xfffffa80020e1c80   1452               0x18                0x1f0003 Event

... output truncated ...
```



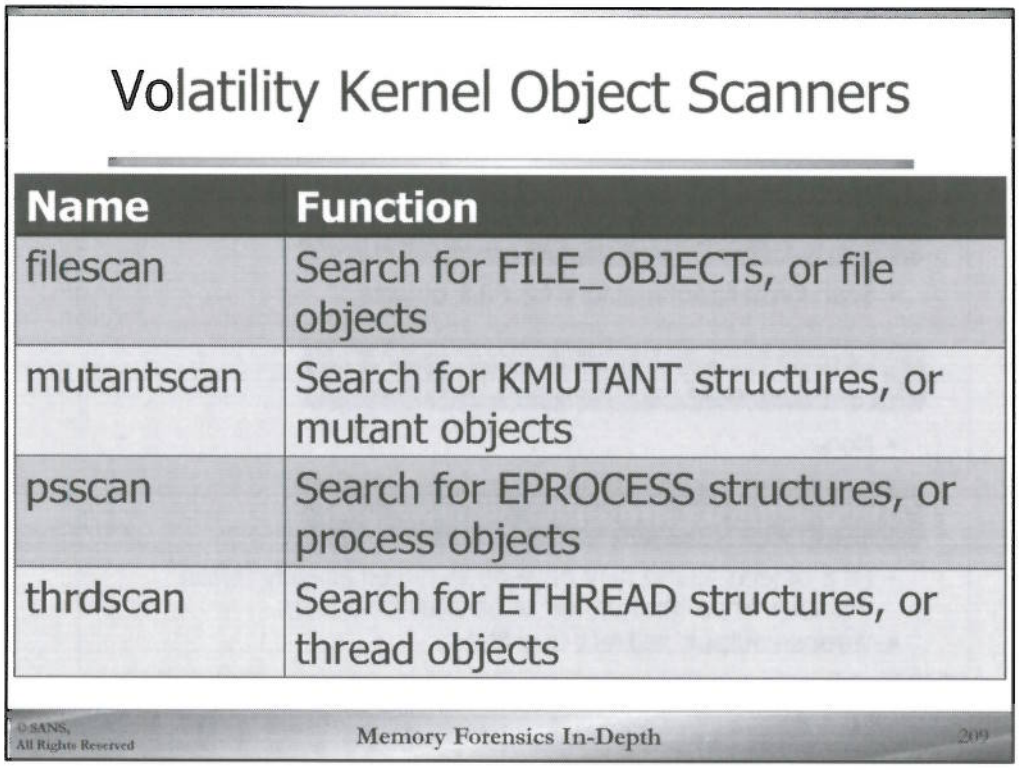
This slide shows the output of the handles plugin. Note that the –s option was specified to filter extraneous results.

Also note that the results have been filtered by process ID. By default, the plugin lists handles for all processes, but here we only want to examine the svchost.exe process with PID 1140 (what *is* that svchost.exe process doing anyway???).

user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 handles -p 1140 -s

Offset(V)	Pid	Handle	Access	Type	Details
0xe10096e0	1140	0x4	0x20003	KeyedEvent	CritSecOutOfMemoryEvent
0xe15a9e98	1140	0x8	0x3	Directory	KnownDlls
0x82325980	1140	0xc	0x100020	File	\Device\HarddiskVolume1\WINDOWS\system32
0xe172a940	1140	0x14	0xf000f	Directory	Windows
0xe1741030	1140	0x20	0x2000f	Directory	BaseNamedObjects
0x822f2108	1140	0x24	0x1f0001	Mutant	SHIMLIB_LOG_MUTEX
0xe1877540	1140	0x28	0x2020019	Key	MACHINE
0x82333818	1140	0x2c	0xf006e	WindowStation	Service-0x0-3e4$
0x81eaf1d8	1140	0x34	0xf00cf	Desktop	Default
0x82333818	1140	0x38	0xf006e	WindowStation	Service-0x0-3e4$
0xe18c2dd8	1140	0x48	0x20019	Key	MACHINE\SOFTWARE\MICROSOFT\WINDOWS NT\CURRENTVERSION\DRIVERS32
0x81f07450	1140	0x50	0x100001	File	\Device\KsecDD
0xe18875e8	1140	0x58	0x20019	Key	MACHINE\SOFTWARE\MICROSOFT\WINDOWS NT\CURRENTVERSION\DRIVERS32
0x81f5ae28	1140	0x5c	0x100002	Semaphore	shell.{A48F1A32-A340-11D1-BC6B-00A0C90312E1}

... output truncated ...



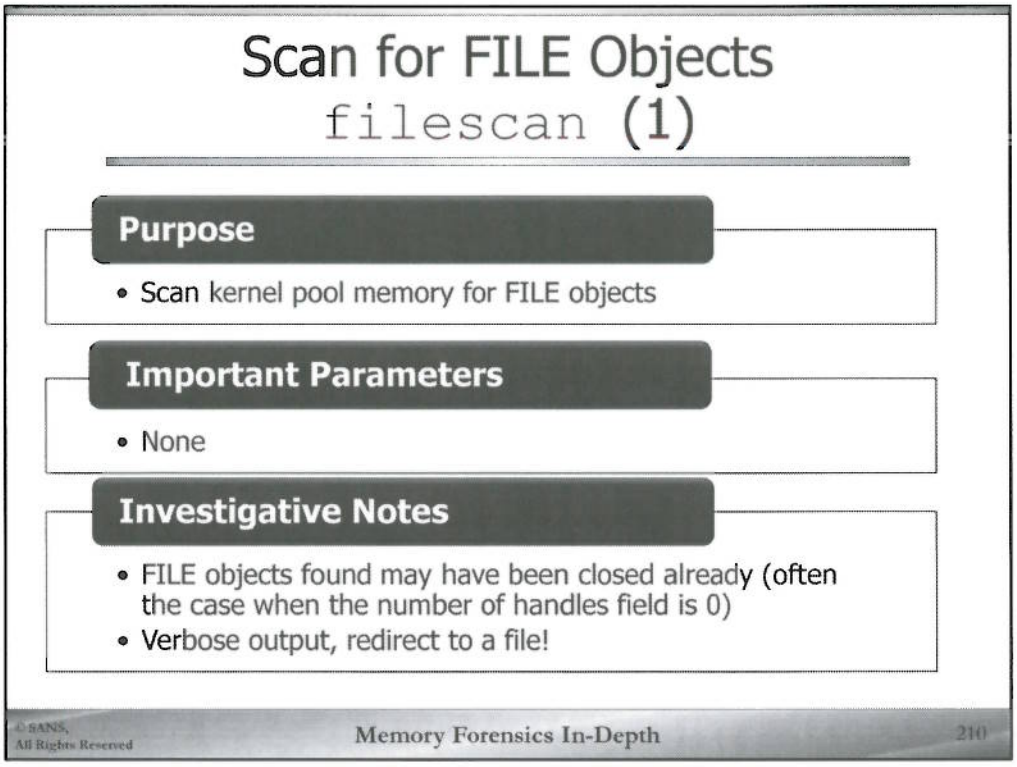
Along with the handles plugin for list walking, Volatility comes with four scanners for kernel objects. All of these scanners are pool scanners, searching pool memory for pool tags and then attempting to do validation on the data they find. As usual, any object which shows up in the brute force scanners but not the list walker is worthy of further investigation. Maybe it's just data which isn't being used anymore but hasn't been overwritten yet. Maybe it's a leftover from a previous boot. But maybe it's an object which has been unlinked from the list.

How can we tell? An object which has been unlinked should have a non-zero count for the number of handles or pointers to it. *Somebody* should be using it! Now, of course, it's possible that the malware authors have also co-opted the mechanisms for loading and maintaining kernel objects. But hopefully those counts are non-zero. Even better, in those objects the open handles list may point us back to a hidden process!

The first plugin is filescan, used for scanning for FILE_OBJECT structures. These correspond to File objects in the kernel, or open files on the disk. Obviously these are handy for seeing which files a process was using.

The next is mutant scan, which searched for KMUTANT objects, or kernel Mutant objects.

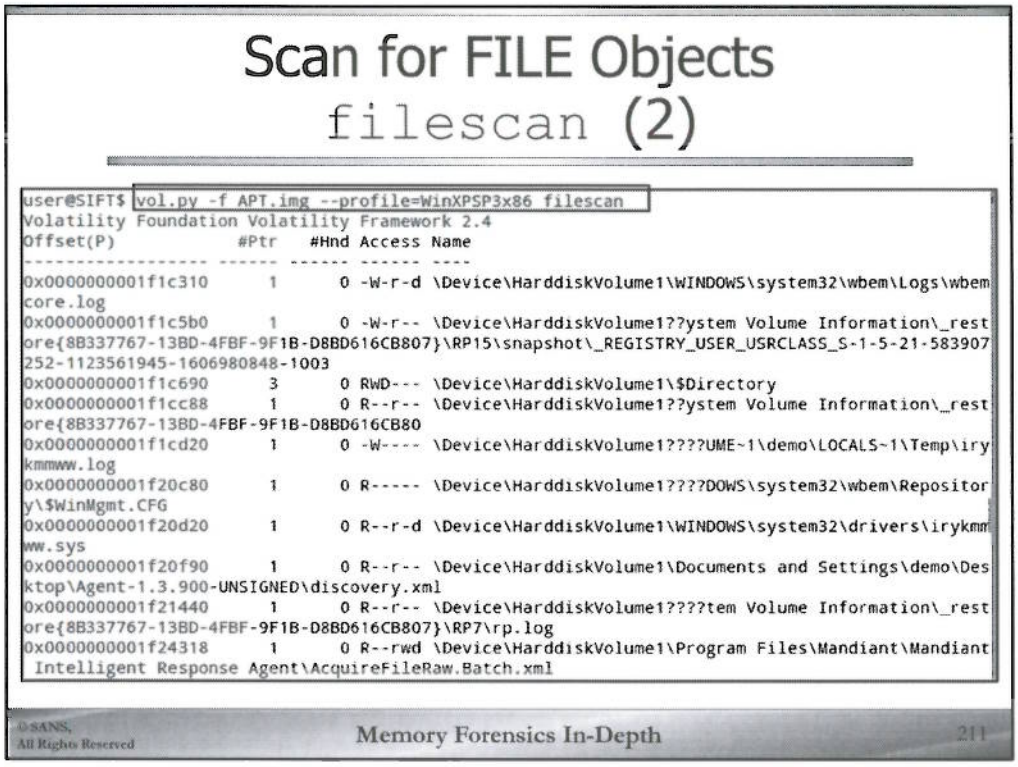
We've already discussed psscan and thrdscan. The former searches for EPROCESS structures, which in turn are kernel Process objects. The latter searches for ETHREAD structures, which in turn are kernel Thread objects.



The filescan plugin scans kernel pool memory for FILE_OBJECT structures. These structures are allocated with the pool tag "Fil\xe5". When these are found in memory, the scanner reports on these items. Note that these may be files that were previously opened but have since been closed. For this reason, you should expect to obtain more output with this plugin than with the handles plugin filtered for only file objects.

Investigators may want to use the filescan plugin to locate evidence that files were opened at some time on the system. Although previously opened files cannot be tied to the file that opened them, sometimes simply knowing that a file was opened on the system by some process is enough to jumpstart an investigation. Of course, investigators should consider that the file may have been opened by security software (for antivirus scanning for instance).

The filescan plugin is implemented in volatility/plugins/filescan.py.



The text below shows an example of the filescan plugin run against a sample memory image. The access mask field data comes from the OBJECT_HEADER object, not the FILE_OBJECT itself.

user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 filescan

Volatility Foundation Volatility Framework 2.4

Offset(P) #Ptr #Hnd Access Name

---------- ------ ------ ------ ----

0x0000000001f1c310 1 0 -W-r-d \Device\HarddiskVolume1\WINDOWS\system32\wbem\Logs\wbemcore.log

0x0000000001f1c5b0 1 0 -W-r-- \Device\HarddiskVolume1??ystem Volume Information_restore{8B337767-13BD-4FBF-9F1B-D8BD616CB807}\RP15\snapshot_REGISTRY_USER_USRCLASS_S-1-5-21-583907252-1123561945-1606980848-1003

0x0000000001f1c690 3 0 RWD--- \Device\HarddiskVolume1\$Directory

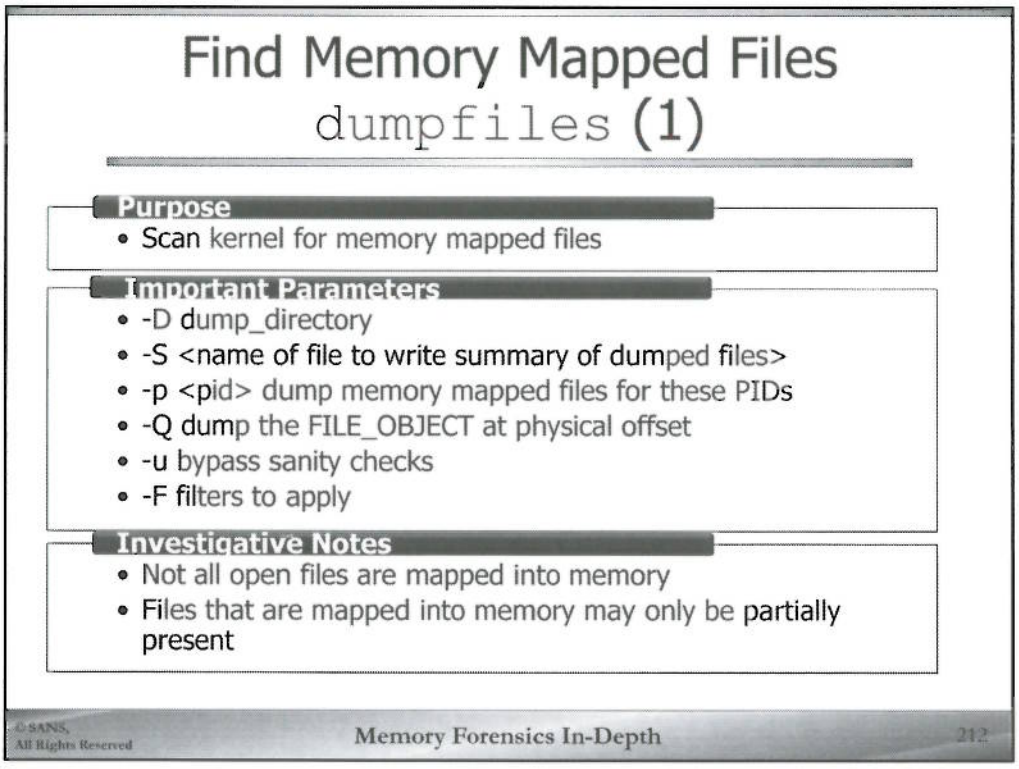
0x0000000001f1cc88 1 0 R--r-- \Device\HarddiskVolume1??ystem Volume Information_restore{8B337767-13BD-4FBF-9F1B-D8BD616CB80

0x0000000001f1cd20 1 0 -W----
\Device\HarddiskVolume1????UME~1\demo\LOCALS~1\Temp\irykmmww.log

0x0000000001f20c80 1 0 R-----
\Device\HarddiskVolume1????DOWS\system32\wbem\Repository\$WinMgmt.CFG

0x0000000001f20d20 1 0 R--r-d \Device\HarddiskVolume1\WINDOWS\system32\drivers\irykmmww.sys

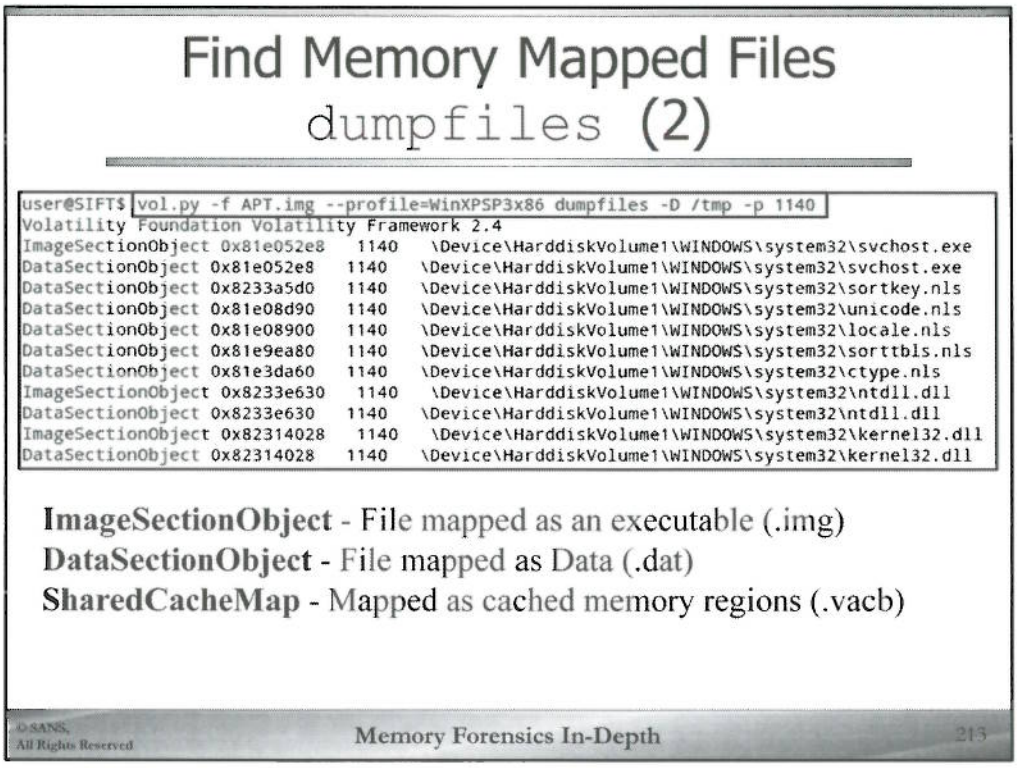
… output truncated …



The dumpfiles plugin identifies memory mapped files for each process. The plugin then writes the contents of the files mapped into memory out to the dump directory. Note that not all files are opened as memory mapped files. Even for those that are, the entire contents of the file may not be present in memory. Sometimes, only sections of a file are memory mapped. Other times, the whole file is memory mapped, but portions of the file are paged out.

Memory mapped files can be a potential gold mine when profiling an attacker, particularly if you only have a memory image. Later we'll talk about how to extract data from a hibernation file. Hibernation files (particularly those in volume shadow copies) may contain file data from files no longer present on disk.

The dumpfiles plugin is implemented in volatility/plugins/dumpfiles.py.



This slide shows some output from the dumpfiles plugin. In this use, we are writing data out to the previously created directory named "dump." Memory mapped files will only be dumped for our suspect svchost.exe process, PID 1140.

user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 dumpfiles -D /tmp -p 1140

Volatility Foundation Volatility Framework 2.4

ImageSectionObject 0x81e052e8 1140 \Device\HarddiskVolume1\WINDOWS\system32\svchost.exe

DataSectionObject 0x81e052e8 1140 \Device\HarddiskVolume1\WINDOWS\system32\svchost.exe

DataSectionObject 0x8233a5d0 1140 \Device\HarddiskVolume1\WINDOWS\system32\sortkey.nls

DataSectionObject 0x81e08d90 1140 \Device\HarddiskVolume1\WINDOWS\system32\unicode.nls

DataSectionObject 0x81e08900 1140 \Device\HarddiskVolume1\WINDOWS\system32\locale.nls

DataSectionObject 0x81e9ea80 1140 \Device\HarddiskVolume1\WINDOWS\system32\sorttbls.nls

DataSectionObject 0x81e3da60 1140 \Device\HarddiskVolume1\WINDOWS\system32\ctype.nls

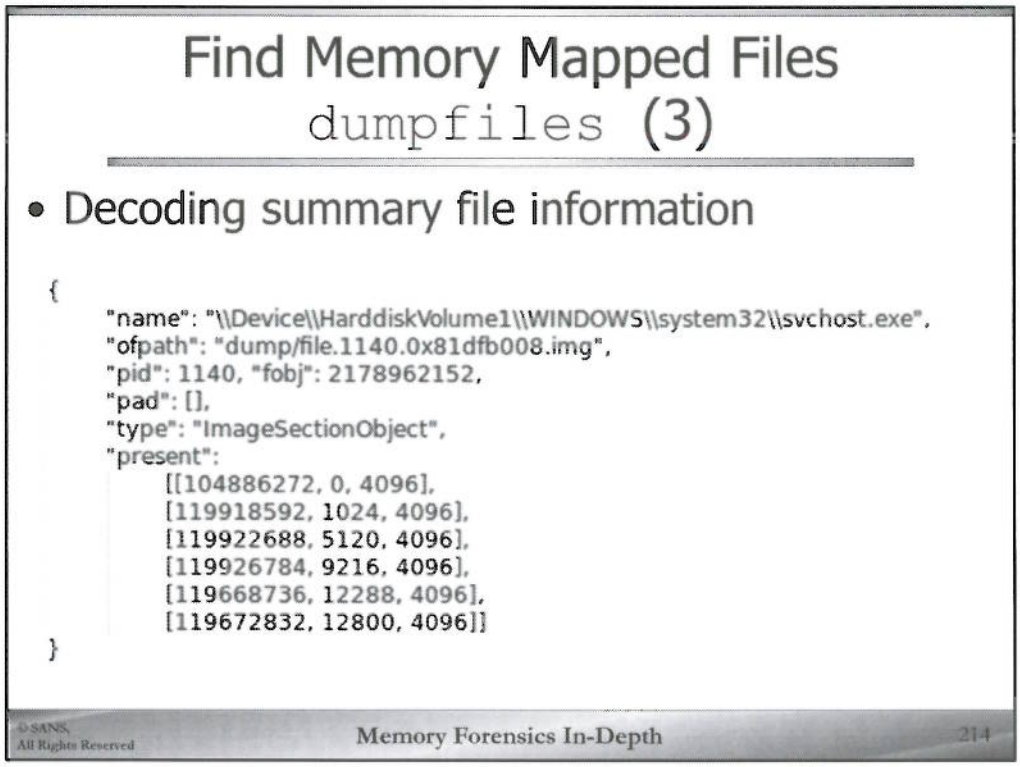
ImageSectionObject 0x8233e630 1140 \Device\HarddiskVolume1\WINDOWS\system32\ntdll.dll

DataSectionObject 0x8233e630 1140 \Device\HarddiskVolume1\WINDOWS\system32\ntdll.dll

ImageSectionObject 0x82314028 1140 \Device\HarddiskVolume1\WINDOWS\system32\kernel32.dll

… output truncated …

Extracted data is categorized by the type of object from which it was sourced: an ImageSectionObject (file mapped as an executable),that ends in an .img extension or a DataSectionObject (file mapped as data), ending with .dat or SharedCacheMap (used to track the state of cached regions) with a .vacb extension.

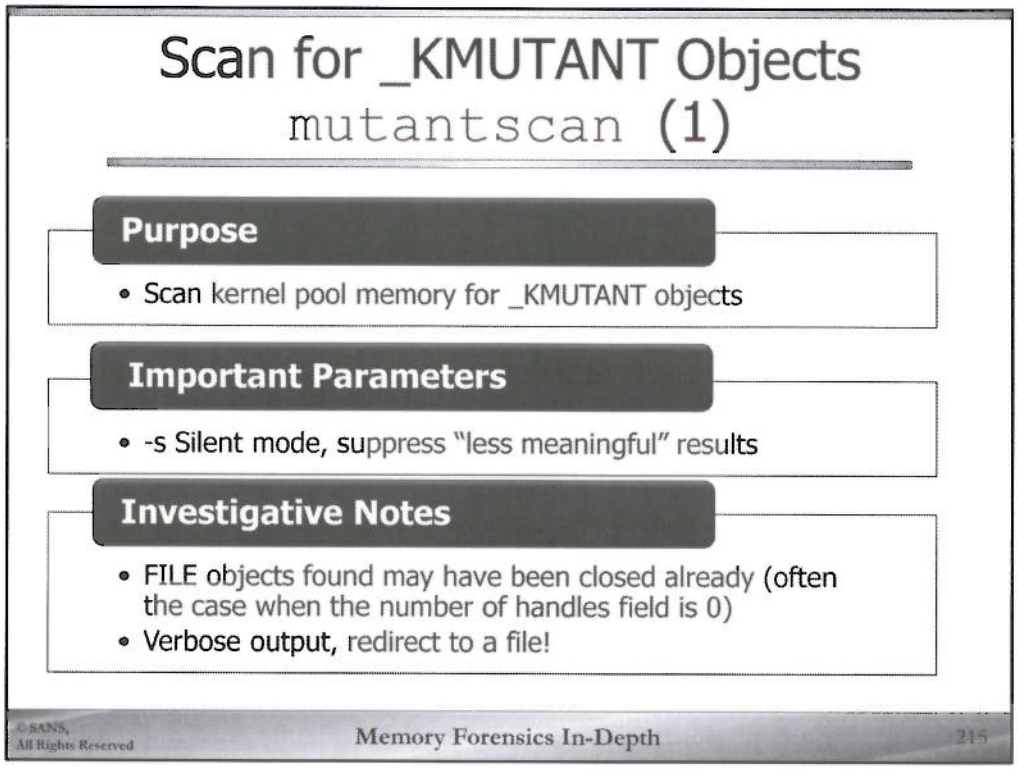


Below, we have a portion of the excerpt from the summary file. It is recommended that you always write a summary file so that you can reconstruct what holes (zero filled) may exist in files that have been dumped via the memory map. Further, these show an easy mapping between filenames in memory and the output file name from the plugin.

In this case the whole file is in memory. However, if it was not, the pad element would be filled in. Looking at this example is very illustrative. In the "present" field, there are a number of 3-tuples. For each 3-tuple, the key to decoding it as follows:
[physical_offset_in_mem_dump, offset_in_file_originally_on_disk, length_in_memory]

Note that the first and second entries here overlap. The first memory mapped segment in memory is bytes 0-4095 in the file. The second memory mapped segment represents bytes 1024-5119 in the file. This is means that 3k worth of the file is mapped into memory in two different locations. This is an artifact of interaction between the memory and IO managers, but is interested. Better to have multiple copies of portions of the file in memory than none at all!


```
{
        "name": "\\Device\\HarddiskVolume1\\WINDOWS\\system32\\svchost.exe",
        "ofpath": "dump/file.1140.0x81dfb008.img",
        "pid": 1140, "fobj": 2178962152,
        "pad": [],
        "type": "ImageSectionObject",
        "present":
                [[104886272, 0, 4096],
                [119918592, 1024, 4096],
                [119922688, 5120, 4096],
                [119926784, 9216, 4096],
                [119668736, 12288, 4096],
                [119672832, 12800, 4096]]
}
```

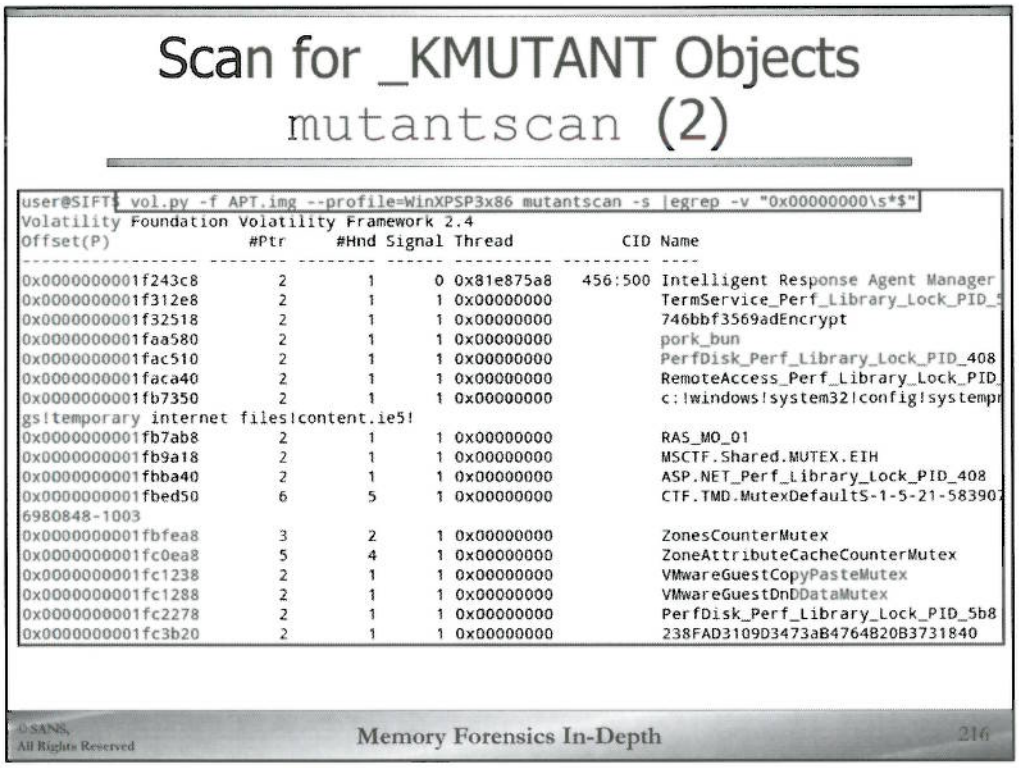


The mutantscan plugin scans kernel pool memory for _KMUTANT structures. These structures are allocated with the pool tag "Mut\xe1". When these are found in memory, the scanner reports on these items. Note that mutants are only present in memory while the process that opens the mutex is running (or until it is closed when no longer needed).

Malware often use Mutexes to prevent multiple reinfection. This is a critical mistake made by the authors of the first version of Conficker. It exploited a zero day vulnerability (MS 08-067) and dropped it's payload. The problem was that the malware couldn't detect that it was already running on the machine it just exploited. Machines exploited each other repeatedly and installed multiple instances of Conficker on each system. A mutex would have solved this problem easily.

In SILENT mode, "less meaningful" simply means an unnamed mutex. Since we are usually looking for mutexes with names that are well known (for malware), we don't really care about those unnamed mutexes.

The mutantscan plugin is implemented in volatility/plugins/filescan.py.



This slide shows some output of the mutantscan plugin. Note that in Volatility 2.4, the –s switch is broken in mutantscan. However, the same effect can be achieved with the following command:

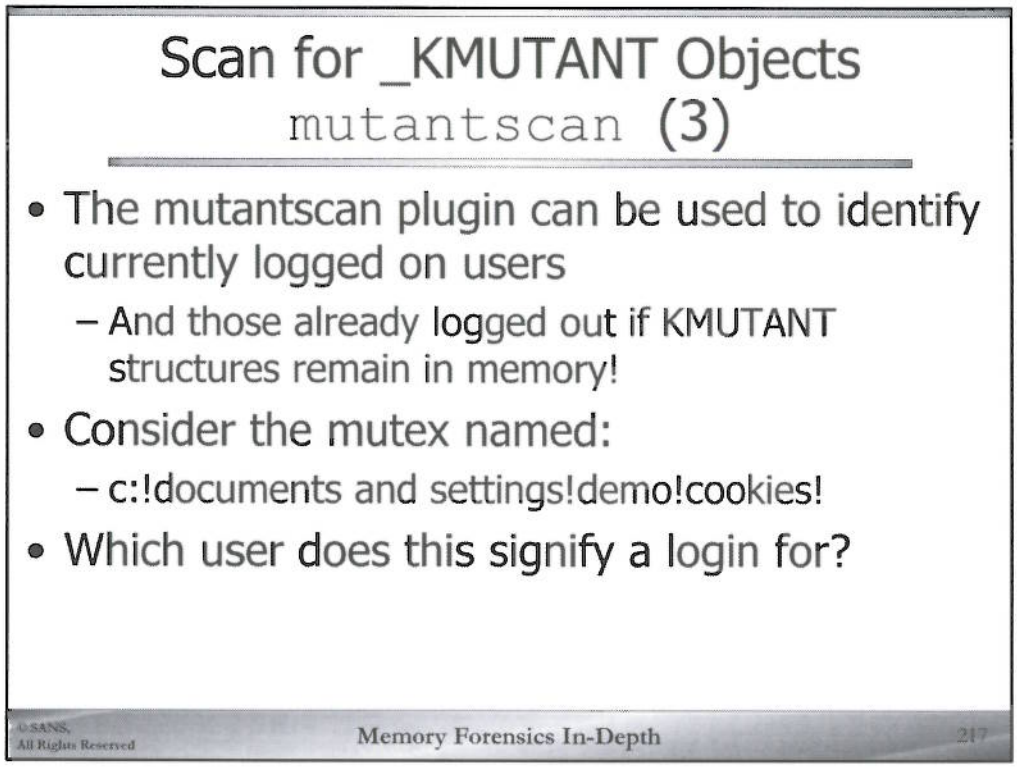
vol.py -f APT.img --profile=WinXPSP3x86 mutantscan -s |egrep -v "0x00000000\s*$"

user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 mutantscan -s |egrep -v "0x00000000\s*$"

Volatility Foundation Volatility Framework 2.4

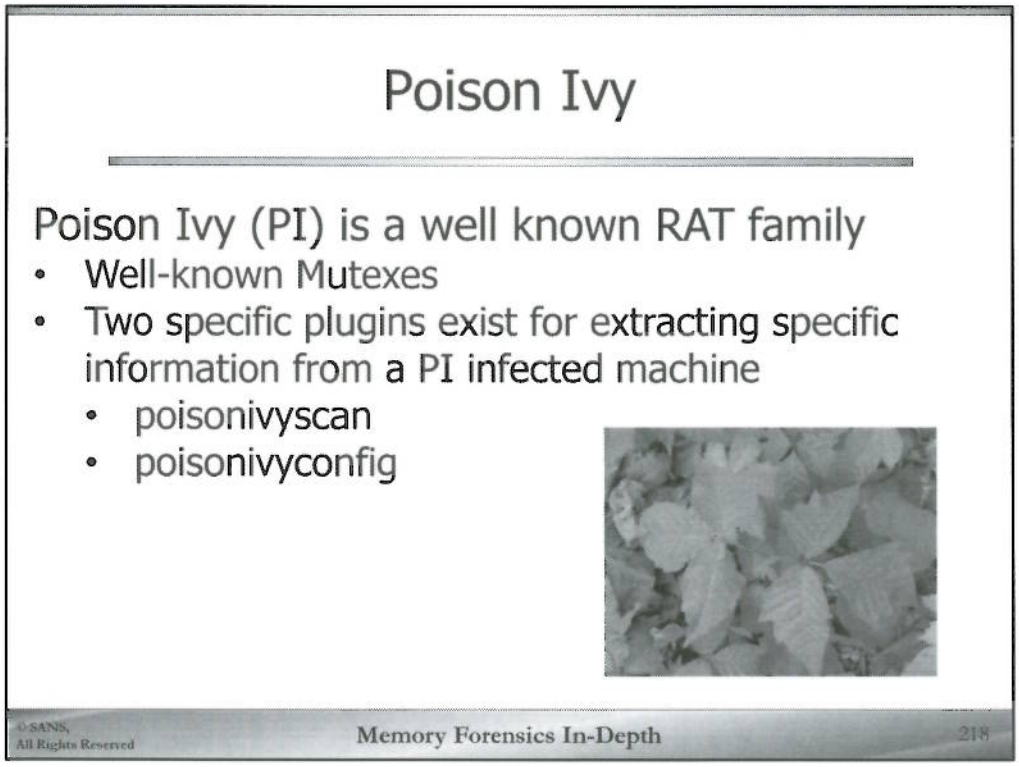
Offset(P)	#Ptr	#Hnd	Signal	Thread	CID	Name
0x0000000001f243c8	2	1	0	0x81e875a8	456:500	Intelligent Response Agent Manager
0x0000000001f312e8	2	1	1	0x00000000		TermService_Perf_Library_Lock_PID_5b8
0x0000000001f32518	2	1	1	0x00000000		746bbf3569adEncrypt
0x0000000001faa580	2	1	1	0x00000000		pork_bun
0x0000000001fac510	2	1	1	0x00000000		PerfDisk_Perf_Library_Lock_PID_408
0x0000000001faca40	2	1	1	0x00000000		RemoteAccess_Perf_Library_Lock_PID_5b8
0x0000000001fb7350	2	1	1	0x00000000		c:!windows!system32!config!systemprofile!local settings!temporary internet files!content.ie5!
0x0000000001fb7ab8	2	1	1	0x00000000		RAS_MO_01
0x0000000001fb9a18	2	1	1	0x00000000		MSCTF.Shared.MUTEX.EIH
0x0000000001fbba40	2	1	1	0x00000000		ASP.NET_Perf_Library_Lock_PID_408

… output truncated …



Note that we can use the mutantscan plugin to identify currently logged in users just by looking at the mutexes. There are a number of cases as well where previously logged in users have been discovered by examining _KMUTANT structures that remain in memory even after the user logged out. Consider this output from the mutantscan plugin where a grep command was run for documents (as in 'Documents and Settings'). Based on this output, we can infer that the user 'demo' has logged in at some point since the machine was powered on (but it may be from a previous boot).

```
# vol.py -f APT.img --profile=WinXPSP3x86 mutantscan -s |grep -i documents
Volatility Foundation Volatility Framework 2.3.1
0x01fca030   3   2    1 0x00000000        c:!documents and settings!demo!cookies!
0x02061a00   2   1    1 0x00000000        c:!documents and settings!localservice!local
settings!history!history.ie5!
0x0210e448   3   2    1 0x00000000        c:!documents and settings!demo!local settings!history!history.ie5!
0x02121a48   2   1    1 0x00000000        c:!documents and settings!localservice!local settings!temporary
internet files!content.ie5!
0x02395450   3   2    1 0x00000000        c:!documents and settings!demo!local settings!temporary internet
files!content.ie5!
0x024faa70   2   1    1 0x00000000        c:!documents and settings!localservice!cookies!
```



Along with searching for open files, we can now start searching for evidence of truly malicious software. Poison Ivy is a "Remote Administration Tool" (RAT), which is another way of saying a program designed to open backdoors in a computer so that other people can connect to it and do things. PI can manipulate files, open a remote shell (i.e., let a remote user enter commands on the system), and so on. It's freely available on the Internet and even has a support system online.

How can you tell if your system is infected with Poison Ivy? In order to prevent multiple installations on the same system, Poison Ivy creates a mutex when it runs. When the program is run, it checks for the presence of this mutex. If PI finds the mutex already exists, it immediately shuts down. There's no point in running two instances of the same program on one system. To avoid problems, the program uses a mutex to ensure that only copy is running.

This is a common mechanism amongst malware. Malware authors don't want to be detected, and so don't want to bog down a system with multiple copies of the same program. Not every malicious program uses a static name for their mutex, however. Some malware uses variable names to avoid detection. Stuxnet, for example, would create:

<quote>

a randomly-named mutex such as "FJKIKK" or "FJGIJK". The trojan also opens or creates one or more of the following mutexes:

@ssd<random hex number>

Global\Spooler_Perf_Library_Lock_PID_01F

Global\{4A9A9FA4-5292-4607-B3CB-EE6A87A008A3}

Global\{5EC171BB-F130-4a19-B782-B6E655E091B2}

Global\{85522152-83BF-41f9-B17D-324B4DFC7CC3}

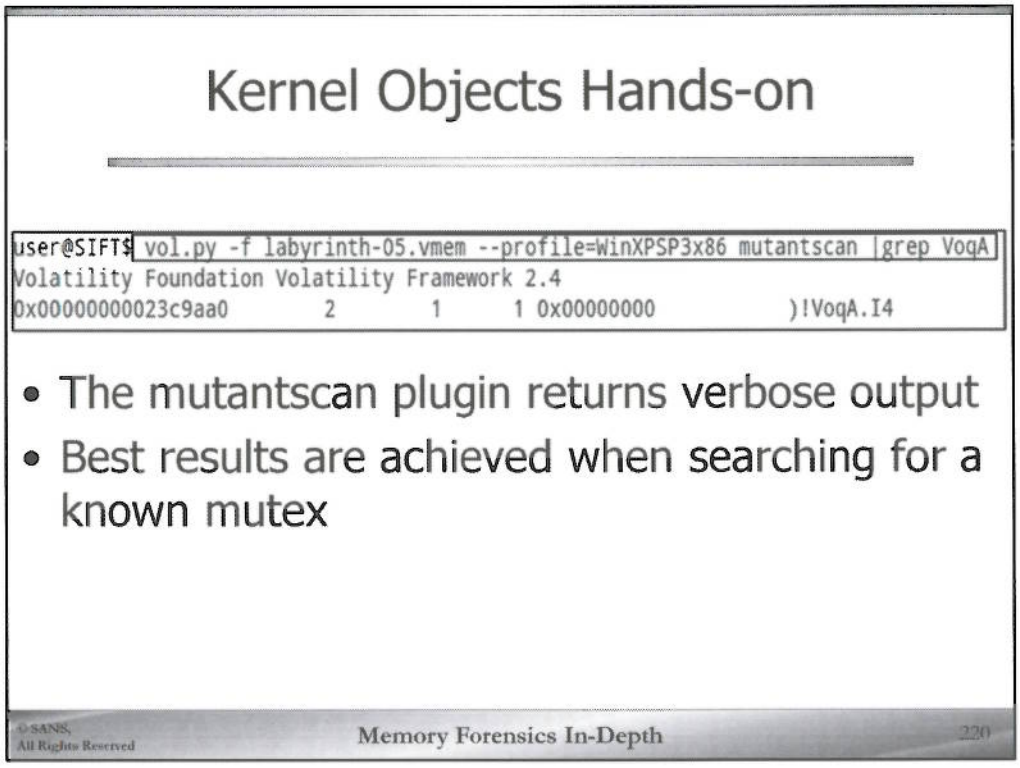
Global\{B2FAC8DC-557D-43ec-85D6-066B4FBC05AC}

Global\{CAA6BD26-6C7B-4af0-95E2-53DE46FDDF26}

Global\{E41362C3-F75C-4ec2-AF49-3CB6BCA591CA}

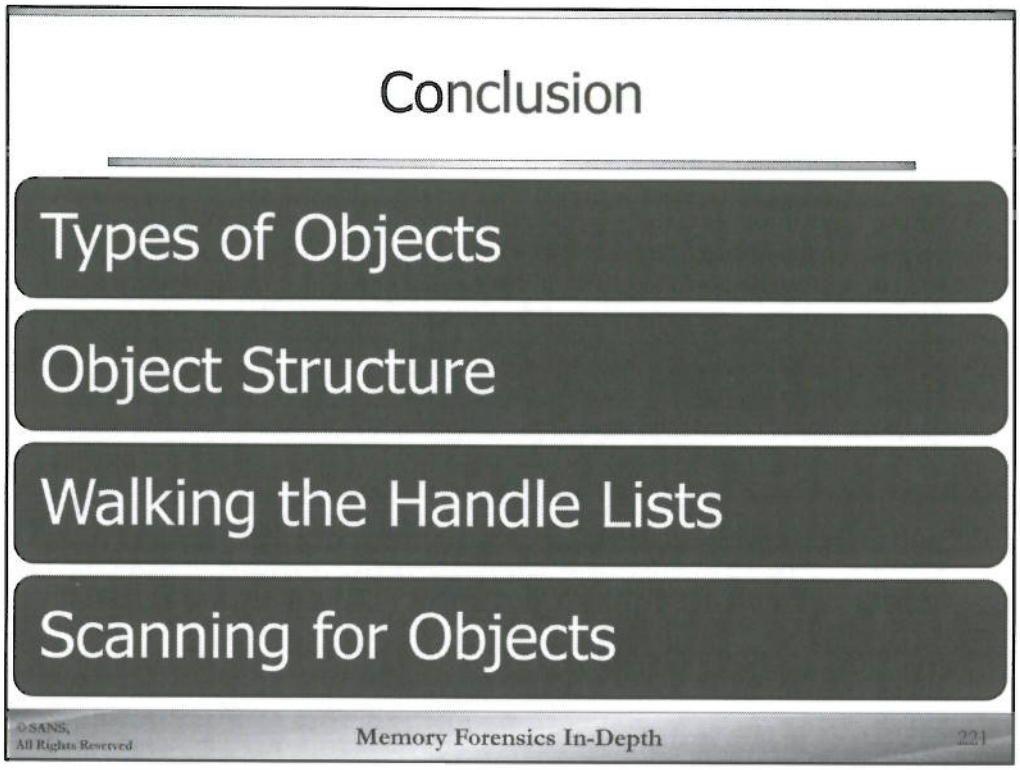
<end quote>

Source: Microsoft Malware Protection Center,
http://www.microsoft.com/security/portal/threat/Encyclopedia/Entry.aspx?Name=Worm%3AWin32%2FStuxnet.A

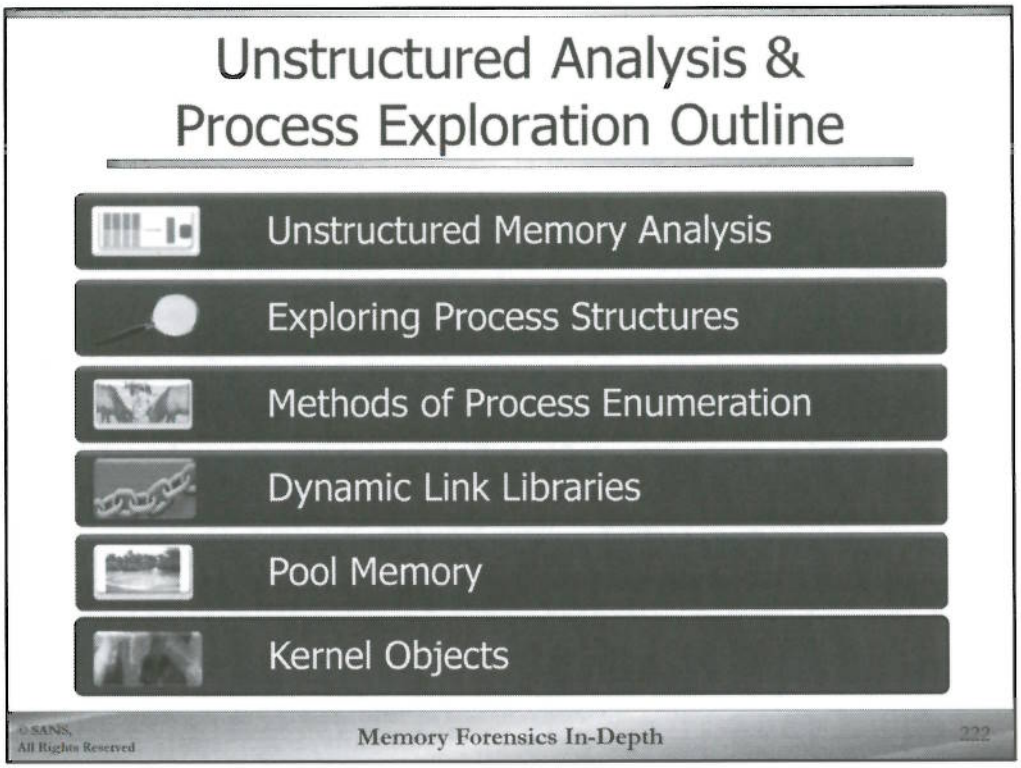


Here is another way to find the Poison Ivy mutant. Rather than modify Volatility, we can use the standard output of the mutantscan plugin, searching for the mutant directly. Using the labrynith-05.vmem memory image, you should see the following:

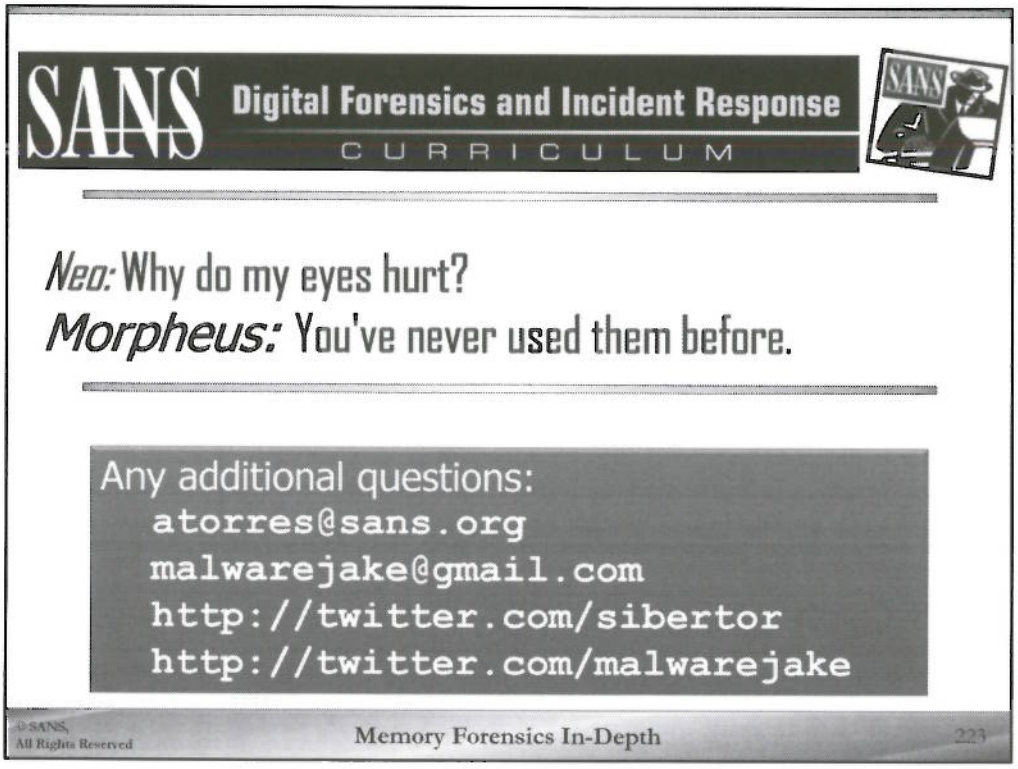
```
$ vol.py -f /cases/labyrinth-05.vmem --profile=WinXPSP3x86 mutantscan |
grep VoqA
0x023c9aa0 0x825c55e0    2    1      1 0x00000000      ')!VoqA.I4'
```



We've covered a lot of ground in this section! We started with introducing kernel objects. Although we've seen some of them before, we discussed what makes a kernel object and how they are managed. User processes can hold handles to these objects, while kernel structures keep pointers. Both are tallied in the objects, which can also help us find which processes are holding the handles to the objects. It's yet another way to find hidden processes! After experimenting with Volatility's built in plugins for walking the list of handles and searching for objects, we modified them to search for objects of particular interest. Volatility isn't just a static forensics tool. You can and should modify it to suit your needs!



This page intentionally left blank.



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