SANS

www.sans.org

FORENSICS 526

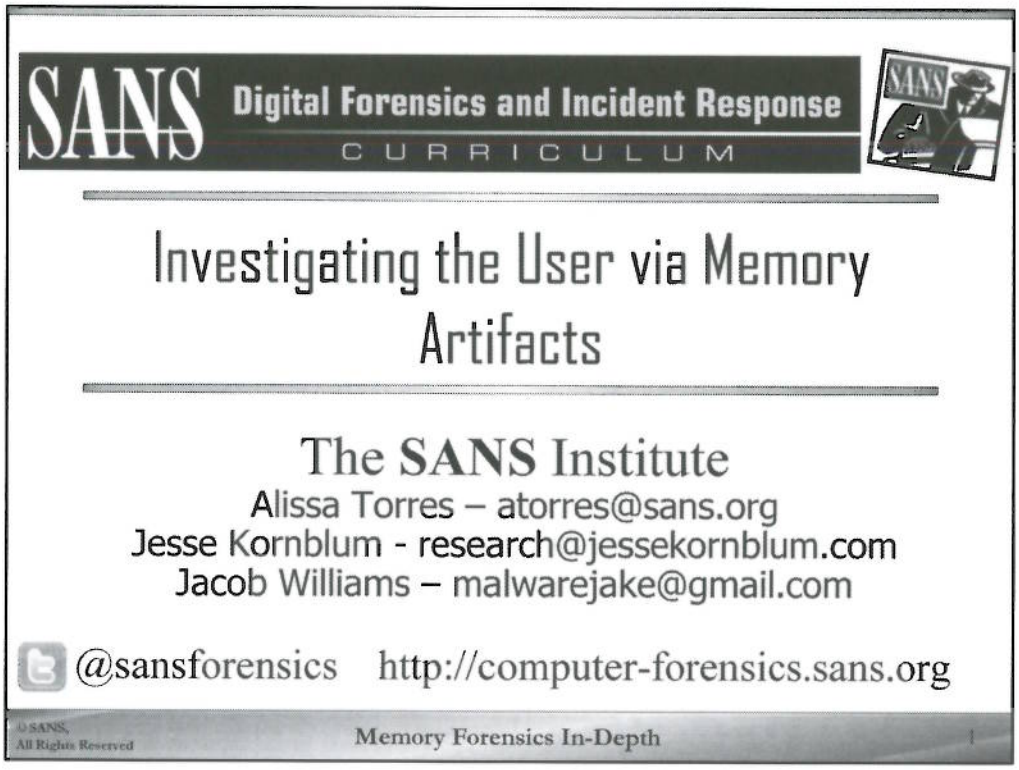
MEMORY FORENSICS IN-DEPTH

526.3

Investigating the User via Memory Artifacts

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

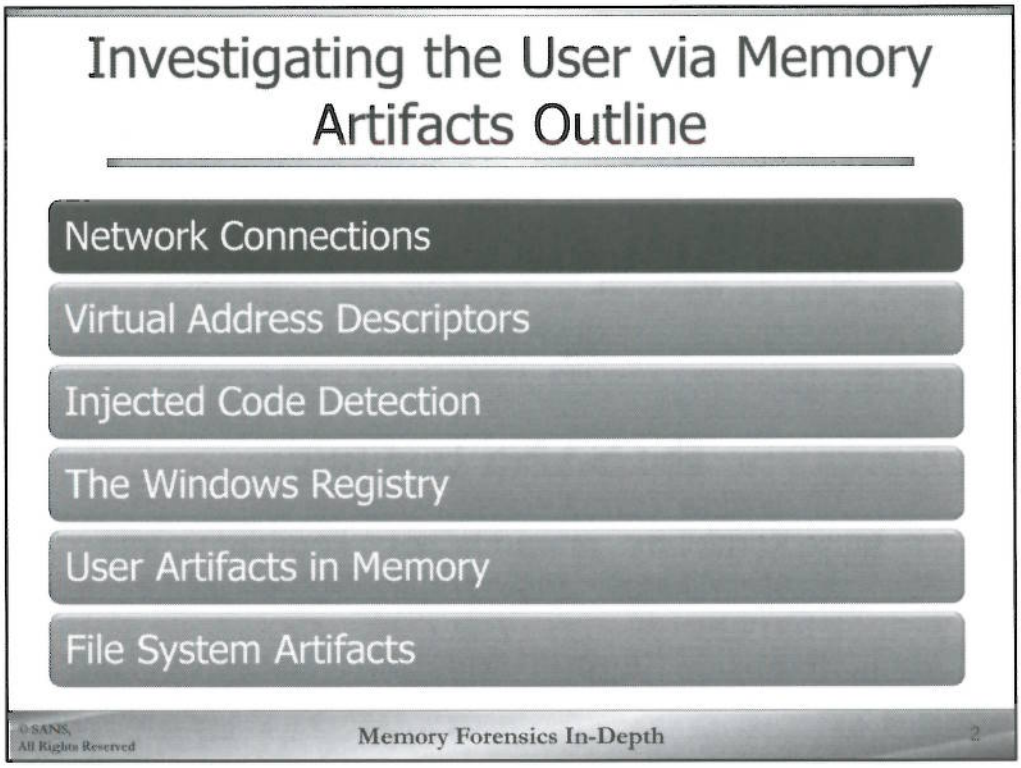
For526_3_A06_01



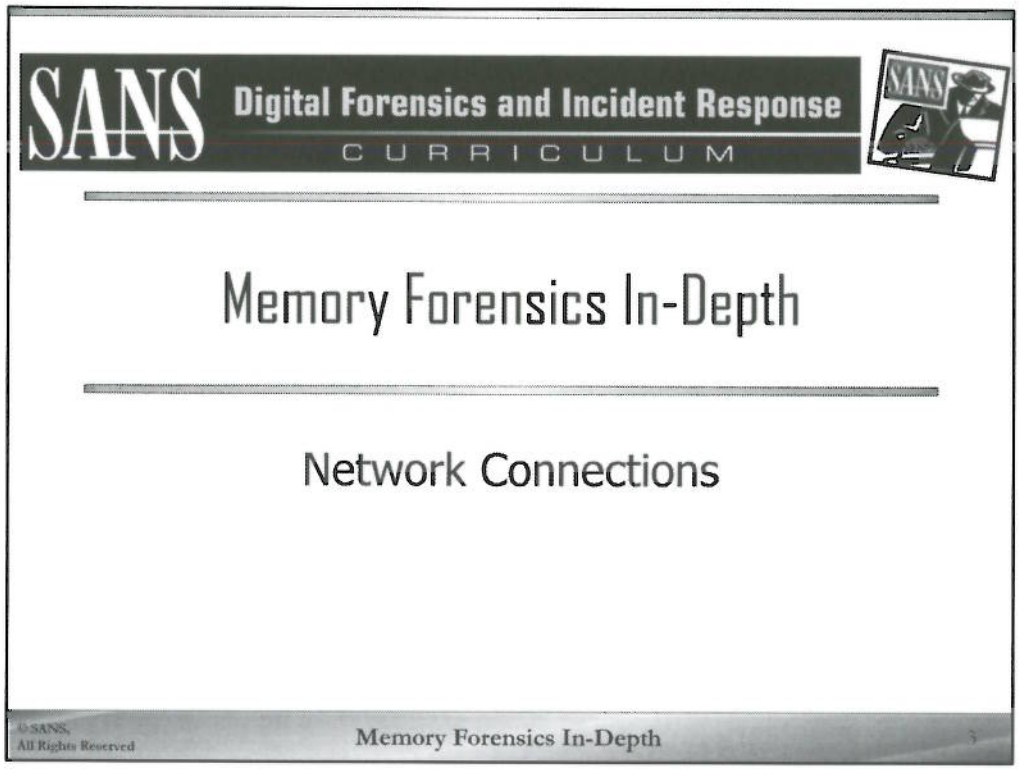
Welcome to Book 3 for FOR526: Investigating the User via Memory Artifacts.

Alissa Torres – atorres@sans.org
Jesse Kornblum – research@jessekornblum.com
Jacob Williams - malwarejake@gmail.com

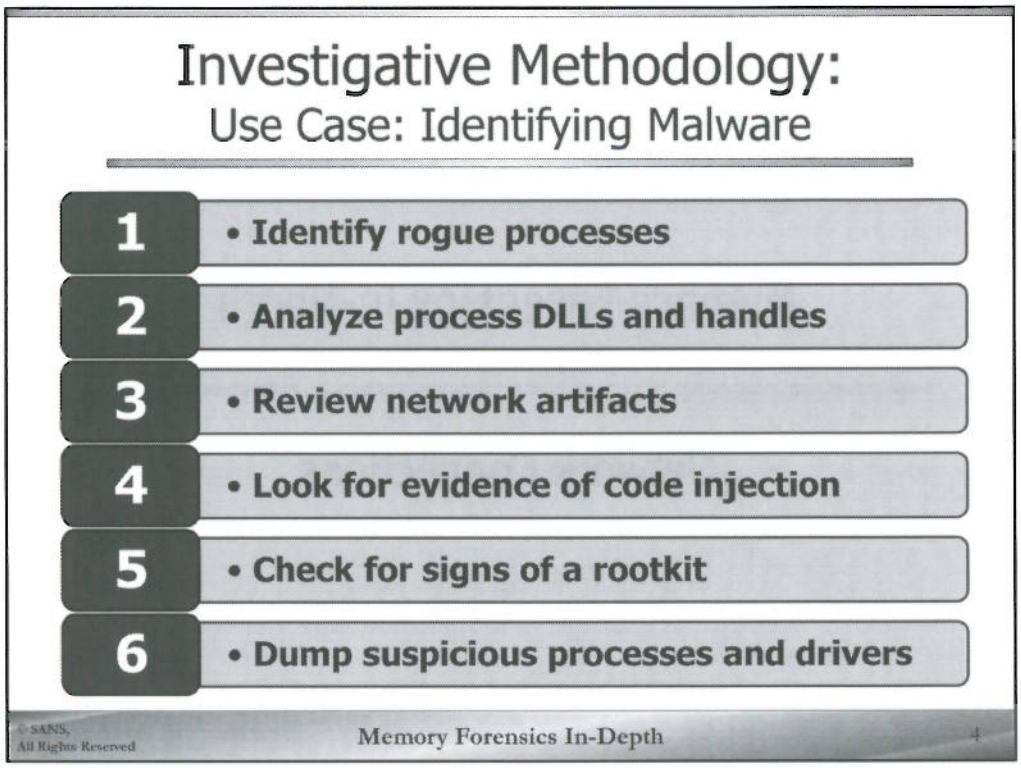
http://twitter.com/sibertor
http://twitter.com/jessekornblum
http://twitter.com/malwarejake
http://twitter.com/sansforensics



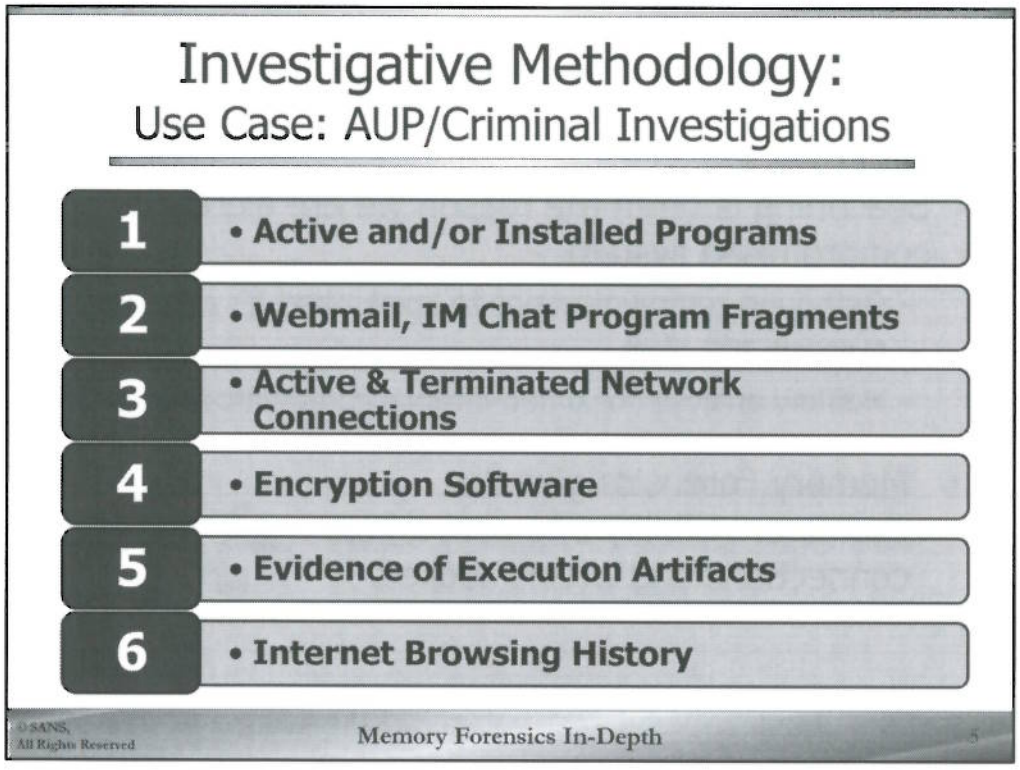
This page intentionally left blank.



This page intentionally left blank.



There are many instances where the 6-step Investigative Methodology will not be followed in order. One such occasion is when suspicious network activity leads you to the system. By enumerating network connections and their owning processes, examiners can get the "first hit" into the rest of the analysis, stemming from PID, time the socket was bound or process start time.



In "Cases Other Than Malware", investigating network connections and their owning processes can be especially applicable.

For example, in attempting to prove that a suspect's home system visited a website hosting **Low Orbit Ion Cannon (LOIC)**, an open source application that has been associated with past denial of service attacks, an investigator may find evidence of a past network connection to the hosting web server still residing in physical memory of that system. Despite having used the best privacy browser in the market today in order to keep all history off the file system, the suspect cannot prevent the TCP pool allocation from being created when an active network connection is made to the LOIC download server. If that structure still exists, even though not presently active, we can identify it with the network Volatility plugins. This use case is an excellent example of how memory forensics can be invaluable in a "COTM" investigation.

Network Connections

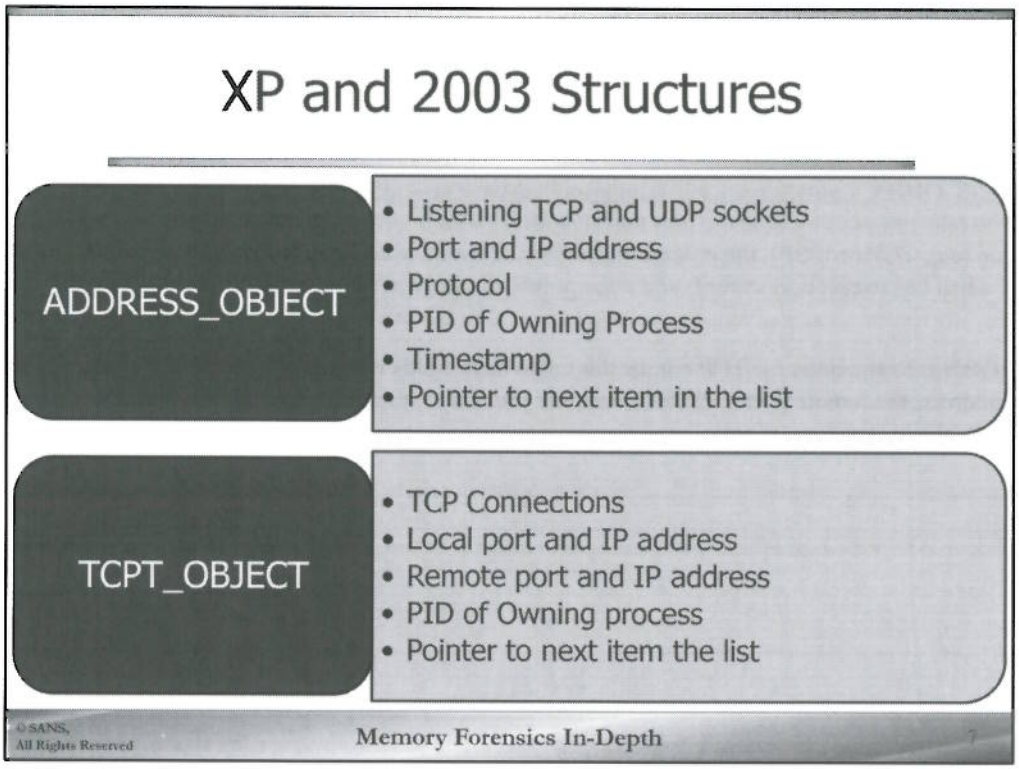
- Beaconing is often the reason we are alerted to a compromised system
  - Outbound communications to known bad IP, domain
  - Network IDS alert
  - Netflow analysis for timed-interval communications
- Memory Forensics give the examiner visibility into *active/past* connections and *bound* sockets

Earlier we found full-content network packets in our memory images. Obviously the full content of network communications is useful, but there is more data we can recover about the networking of the target computer. Along with seeing the active communications, looking at the networking structures can reveal previous connections and bindings to ports. That is, we can see which programs were listening on which ports for incoming data. Such data can be used to find malware listening for instructions, for example.

In this section we are going to look at the networking components of the Windows operating systems. There are two models for Windows networking. The first was used in Windows XP and 2003, the latter in Windows Vista, 2008, 7, and Windows 8. This second method, called the Next Generation TCP/IP stack, completely changed how networking information is stored in the kernel. In this section we will explain both methods, the data structures which are involved, and do some exercises to pull out this data.

In general, Windows uses different structures to keep track of listening sockets and network connections. Sockets are used for processes which are bound to incoming ports on the machine. Connections to the machine on those ports are routed to those processes. Bound sockets are used by programs like web servers, and by many legitimate system utilities. Many system processes communicate via sockets. Such sockets provide an abstraction layer between processes.

Please note that this is only an introduction to networking technologies. There are entire books written about how networking, and windows networking works. The most canonical is *TCP Illustrated, Volumes I-III*, by W. Richard Stevens. These are immensely technical books, and go into extreme depth about how the protocols work. But if you want to know everything, you want those books.



In Windows XP and Windows 2003, there were separate networking stacks for IPv4 and IPv6. IPv6 was seen as a separate networking protocol, and had to be installed and managed separately. IPv4 connections were handled by tcpip.sys, and IPv6 connections were handled by tcpip6.sys. When those operating systems were introduced, this generally was not a problem. IPv6 was not widely used. But as the new protocol version gained popularity, it created an issue which Microsoft addressed with the Next Generation IP Stack.

In XP and 2003, Windows used two structures, the ADDRESS_OBJECT and TCPT_OBJECT, to keep track of bound sockets and network connections, respectively. A bound socket listened for incoming connections. A socket was used to make outgoing connections.

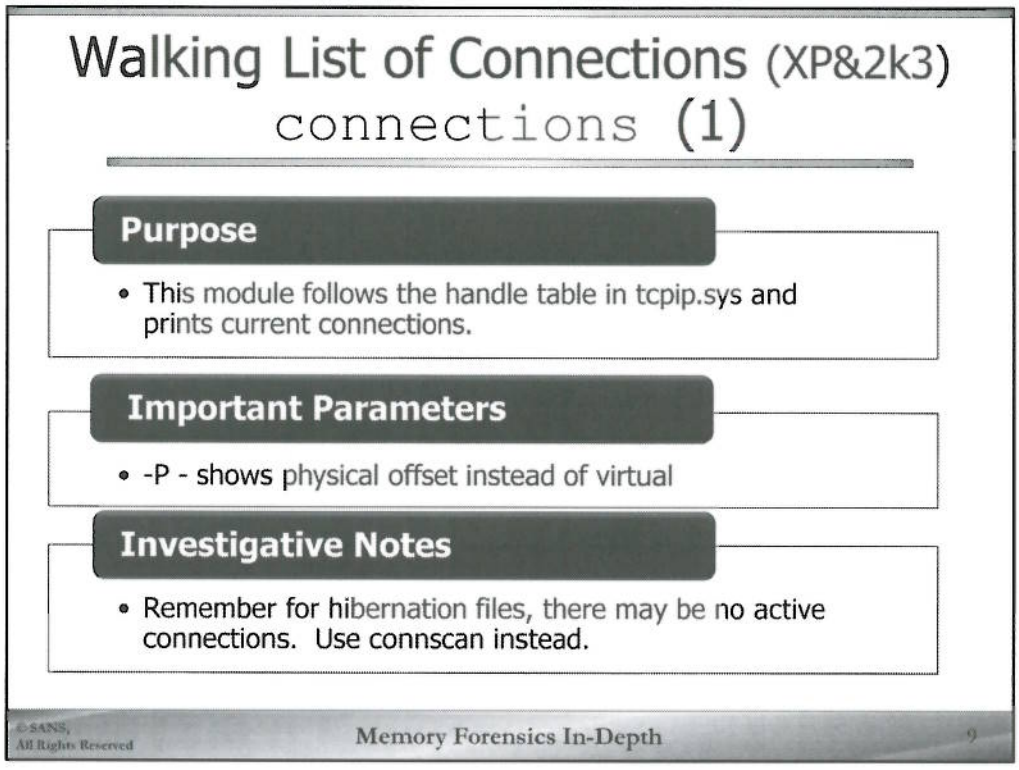
Address Objects were used for both TCP and UDP traffic, while TCPT Objects were for TCP connections only. Both of these structures were kept in a singly-linked list as part of the tcpip.sys driver. A roughly analogous structure was kept in tcpip6.sys. Having two copies of essentially the same code was both a maintenance headache for Microsoft and a source of potential security vulnerabilities. (Did the fixes from one code base get ported to the other?)

These are the first structures we've dealt with which are maintained by a driver instead of the kernel. But thanks to the mechanisms we've already covered, we don't have to introduce any new technologies. The tcpip.sys driver stored these objects in the non-paged pool. Although it's easy enough to scan the pool memory for these objects, we often see false positives in those scans. Hopefully you can weed such false positives out by the sheer absurdity of the values.

Walking the lists of these objects is a little tricky. Microsoft did not create a kernel variable which points to the start of the lists. Unlike say, processes, which has the variable PsActiveProcessHead, there is no such variable for socket and connection objects. The Volatility framework deals with this problem by finding the tcpip.sys driver and doing a brute force search through it to find what looks like the start of the list.

The ADDRESS_OBJECT structures for listening sockets have several fields, only some of which are relevant for forensics. The relevant ones include the port and IP address which have been bound, the protocol which this socket is listening on (e.g. TCP or UDP), the process id number of the owning process, the time when this structure was created (i.e. when the connection started) and a the pointer to the next address object structure, if any.

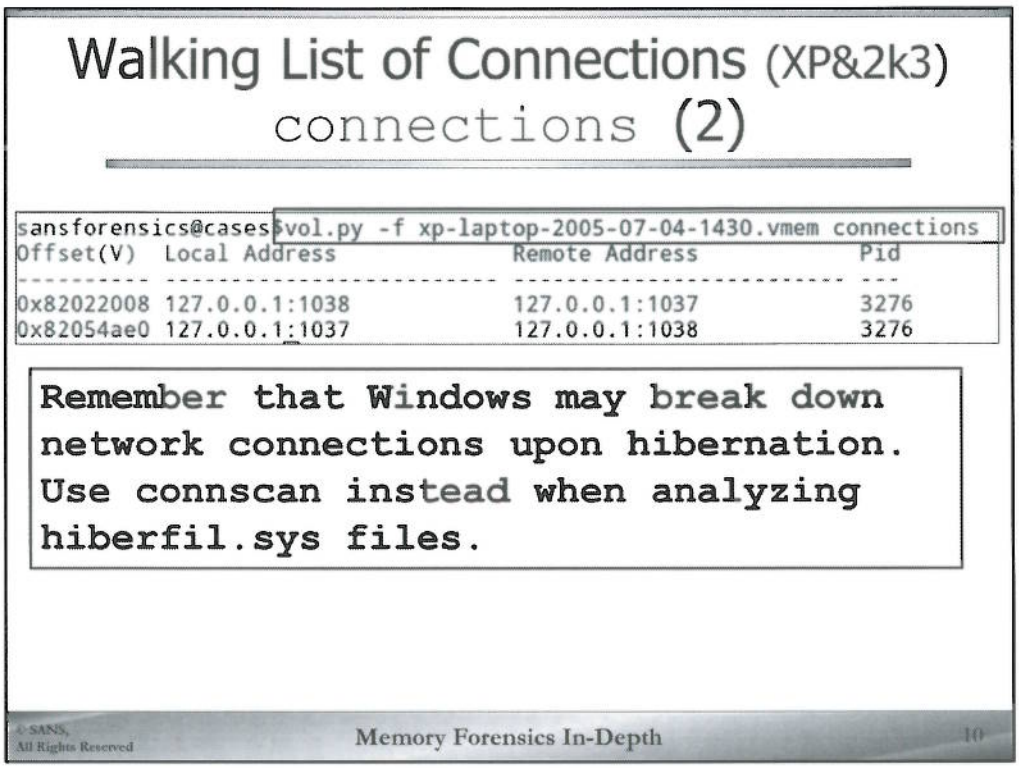
The TCPT_OBJECT structures for TCP connections also have fields relevant for forensics. These include the local port and IP address, the remote port and IP address, the process id number of the owning process, and a pointer to the next TCPT Object structure, if any.



Due to the lack of a list head, the connections plugin uses the tcpip.sys driver in order to gain access to the list of TCPT_OBJECTs structures. Once a TCPT_OBJECT is identified, the plugin walks the f-links through the list of structures in order enumerate active TCP connections. By default, the virtual offsets of the structures are displayed. By using the -P option, the connections plugin will display the physical offset instead.

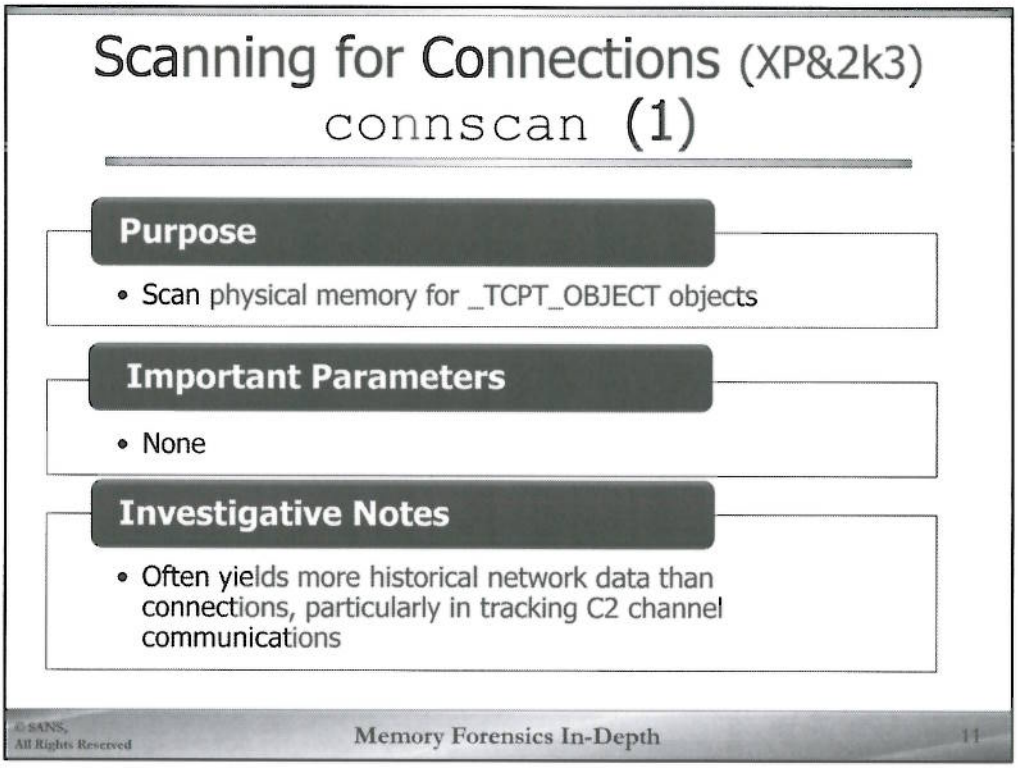
Below is the _TCPT_OBJECT structure:

```
dt("_TCPT_OBJECT")
'_TCPT_OBJECT' (32 bytes)
0x0  : Next                 ['pointer', ['_TCPT_OBJECT']]
0xc  : RemoteIpAddress      ['IpAddress']
0x10 : LocalIpAddress       ['IpAddress']
0x14 : RemotePort           ['unsigned be short']
0x16 : LocalPort            ['unsigned be short']
0x18 : Pid                  ['unsigned long']
```

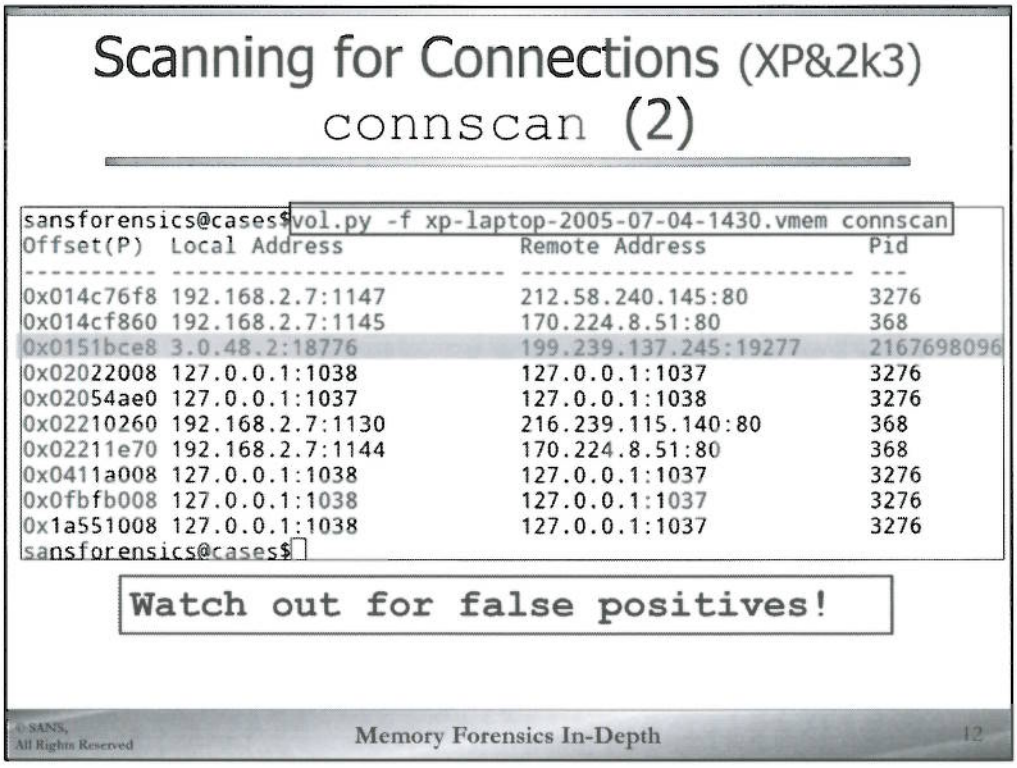


Using the connections plugin on the xp-laptop-2005-07-04-1430.vmem, we are able to identify two active tcp connections. This plugin's output shows the virtual offset for the TCPT_OBJECT, in addition to the local port and IP address, the remote port and IP address, the process id number of the owning process.

$vol.py -f xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86 connections

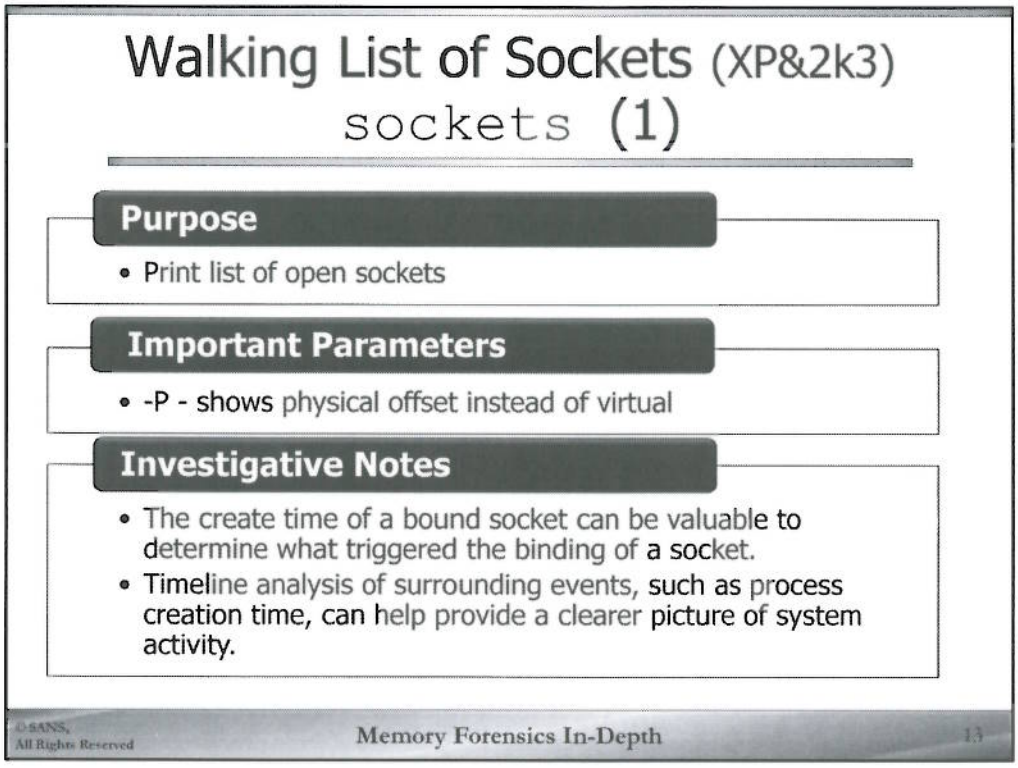


Often when an investigator is notified of a suspicious system on the network and initiates a response, the system no longer has an active connection to the C2 (command and control) server nor is it exhibiting the same network behavior that originally created the alert. The investigator requires a more historic view of network activity and often times, **connscan** provides this view into past, terminated tcp connections. The **connscan** plugin scans for TCPT_OBJECT structures and does not rely on following the f-links (forward links) from one structure to the next. Therefore, **connscan** output will be more verbose but may include duplicates or false positives.



The connscan plugin is notorious for false positives so some validation of the output is required. In the example above, the third line shows a local address of 3.0.48.2, which may be legitimate if this system had more than one network interface card. But the value that makes this entry an obvious false positive is the outrageously high PID specified in that entry. Though the PID is a D-Word value, having a max value of 2^{32}, we can see, based on the other owning PIDs of 3 and 4-digits in length, that the owning PID of "2167698096" is an anomaly.

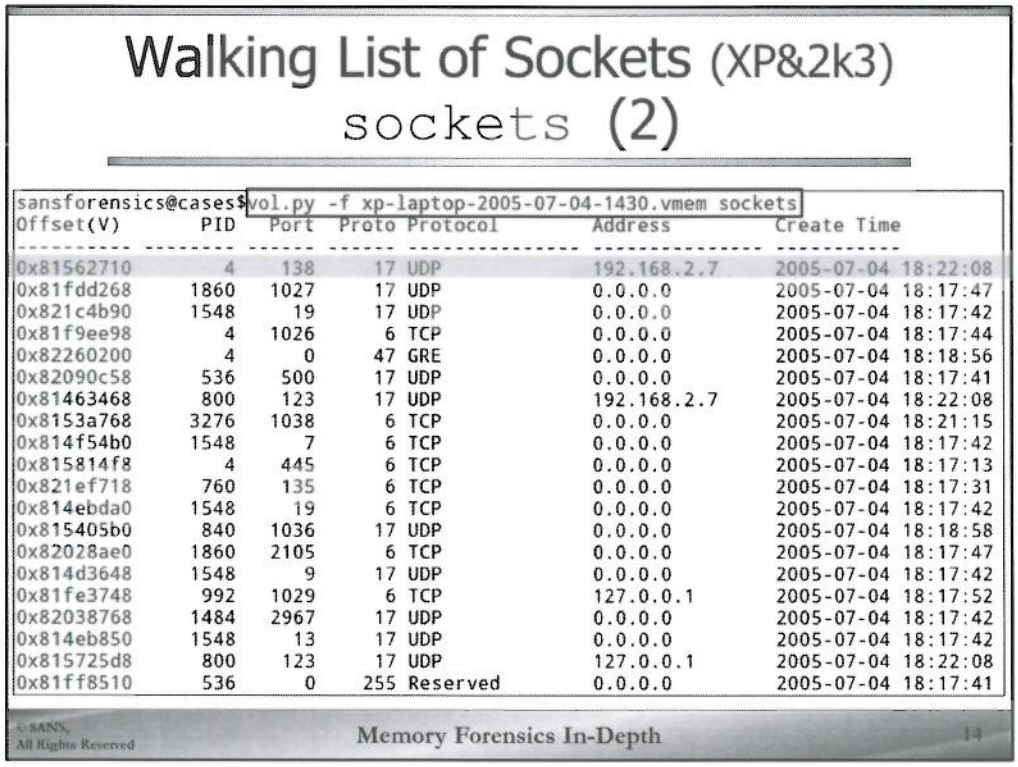
In attempting to investigate the process responsible for a past network connection based on the PID seen in **connscan** output, the examiner may be unable to find the _EPROCESS block for that PID. It is possible that the _TCPT_OBJECT enumerated by **connscan** references a process whose _EPROCESS block has been overwritten.



Much like the connections plugin, sockets uses the tcpip.sys driver in order to gain access to a "first hit" of ADDRESS_OBJECTs and walks the linked lists of active bound sockets by following the flinks for each of these structures. The output of this plugin includes the Create Time for the binding of the socket, either TCP or UDP, that can be used in timeline correlation to determine what else happened on the system around the same timeframe.

Shown below is the _ADDRESS_OBJECT structure:

```
dt("_ADDRESS_OBJECT")
'_ADDRESS_OBJECT' (104 bytes)
0x0   : Next              ['pointer', ['_ADDRESS_OBJECT']]
0x2c  : LocalIpAddress    ['IpAddress']
0x30  : LocalPort         ['unsigned be short']
0x32  : Protocol          ['unsigned short']
0x148 : Pid               ['unsigned long']
0x158 : CreateTime        ['WinTimeStamp', {'is_utc': True}]
```



We can use **sockets** to enumerate ADDRESS_OBJECT structures which walks the list of listening sockets in tcpip.sys. Here's the command line to use:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
sockets
```

Which should generate output like this:

```
 Offset(V)   PID    Port    Proto         Address        Create Time
---------- ------ ------ ----------- --------------   -------------------
0x81562710      4    138      17 UDP     192.168.2.7      2005-07-04 18:22:08
0x81562710   1860   1027      17 UDP     0.0.0.0          2005-07-04 18:17:47
0x81fdd268   1548     19      17 UDP     0.0.0.0          2005-07-04 18:17:42
0x81f9ee98      4   1026       6 TCP     0.0.0.0          2005-07-04 18:17:44
0x81f9ee98      4      0      47 GRE     0.0.0.0          2005-07-04 18:18:56
0x82090c58    536    500      17 UDP     0.0.0.0          2005-07-04 18:17:41
0x81463468    800    123      17 UDP     192.168.2.7      2005-07-04 18:22:08
0x81463468   3276   1038       6 TCP     0.0.0.0          2005-07-04 18:21:15
```

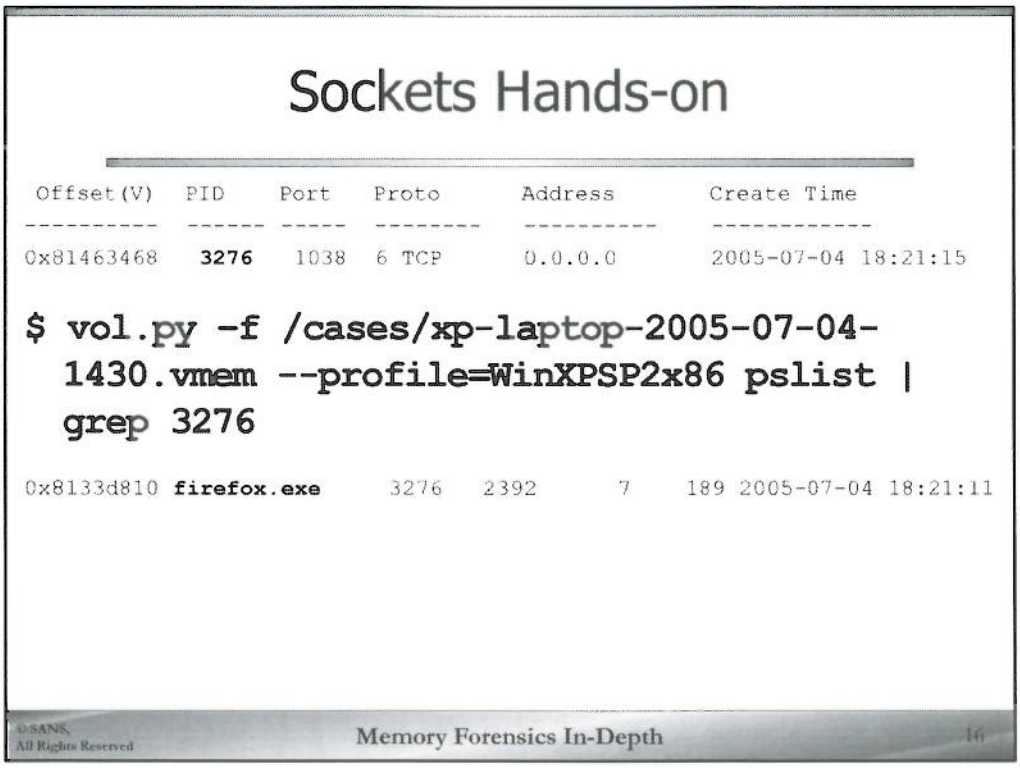
The first column is the virtual address of each ADDRESS_OBJECT structure. You can get the physical offsets in this column by using the -P flag.

The other fields are generally self-explanatory, except for the "Proto" field. This field gives the number of the protocol this socket was listening for. There's a complete reference for these fields at http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml. Volatility attempts to decode the

more common protocols for you, and displays them next to the protocol number. For example, in the first line, protocol number 17 corresponds to UDP traffic. In this case, PID 4, the System process, was listening on port 138 on the interface 192.168.2.7 for UDP traffic. The port was bound starting at 18:22:08 on 4 Jul 2005 UTC. (To see what process 4 was, you may need to re-run the pstree or pslist plugins.)

If the address field is 0.0.0.0, the socket was listening on all available interfaces.

It would certainly be possible for a piece of malware to unlink one of the socket entries from the list of sockets. Such a socket would not be displayed in the output of the sockets plugin. For that, we'll need to scan for socket objects with sockscan.



Let's take a closer look at one of those network connections. In particular, the eighth line in the output shows a listening process, pid 3276, for TCP traffic on port 1038. Which process was that? We can use the pslist plugin to quickly check.

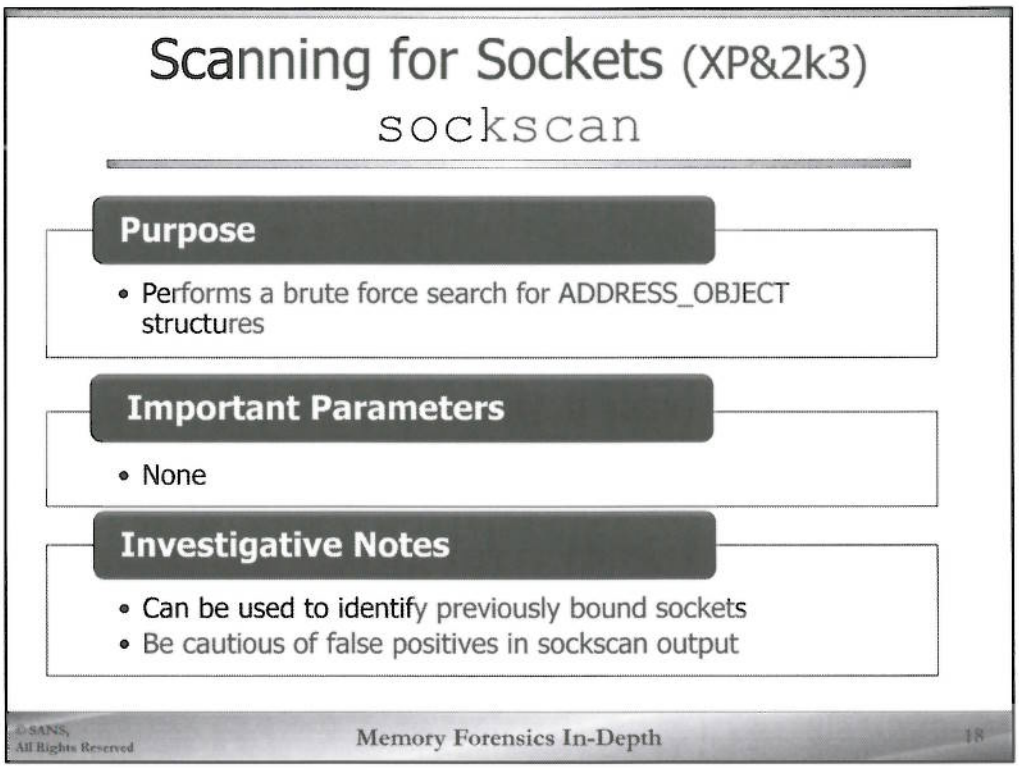
```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
pslist | grep 3276
0x8133d810 firefox.exe            3276   2392      7      189 2005-07-04
18:21:11
```

The Firefox browser had bound to a high port and was listening for incoming connections. Firefox does this to communicate with itself. Although you might think this is suspicious, it's a known issue. A little web searching can help you determine the suspicious bound ports from the legitimate ones (e.g. https://bugzilla.mozilla.org/show_bug.cgi?id=100154).

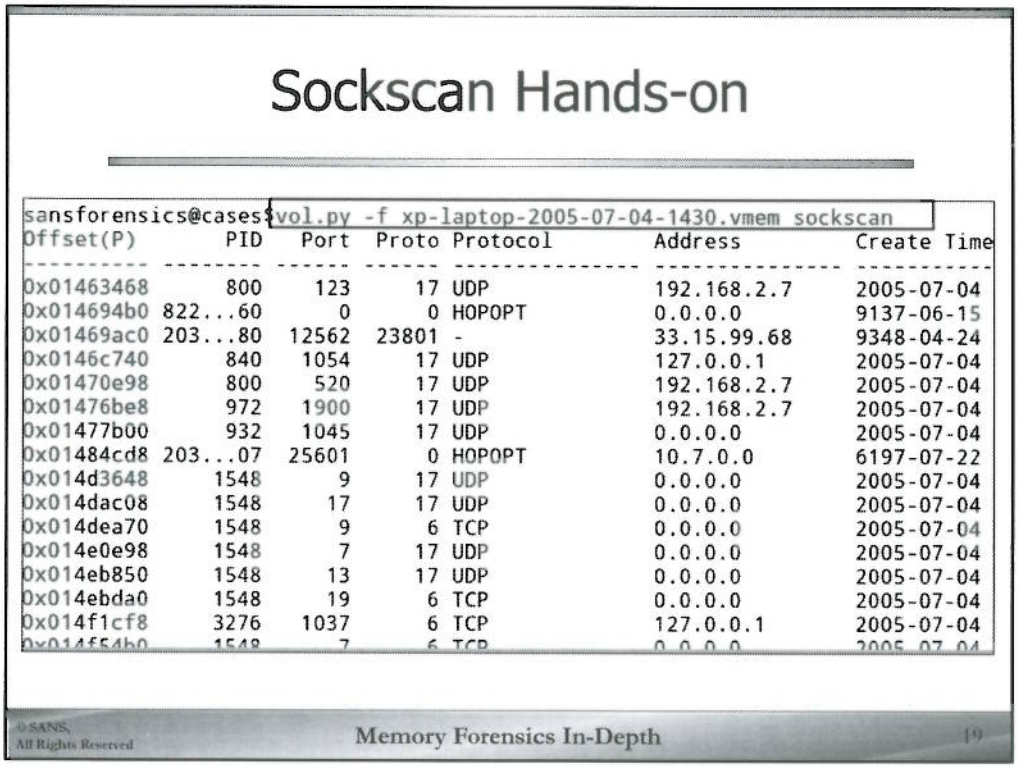
But what if we didn't find a process with id 3276? What would happen if we found a bound port but no process with the specified pid? That would happen if the process was hiding itself from the list of processes, but had not hidden the bound port.

Let's say a piece of malware ran and somehow hid itself from the list of active processes. Such a process would not show up in the output of the pslist plugin, for example. But unless that same malware was careful to hide all of its socket objects too, any port bindings would show up in the list of sockets. We'd see them in the output of the sockets plugin! The PID field of the 'sockets' output should point to a valid process in the 'pslist' output. If there is no corresponding process, then we've found a hidden process!

This would be an example of the rootkit paradox in action. Because the malware attempted to hide itself, it created an opportunity for some other part of the kernel to not look normal. There were 45 sockets listed in the output of the 'sockets' plugin. It would take a little while to go through every one of them and verify if it was legitimate or not. But if there's a socket which is held by a process which doesn't show up in the list of active processes, we know immediately that something is amiss. The malware's own efforts to hide itself have exposed it. Sockets are only the beginning when it comes to this kind of approach.



The **sockscan** plugin can be used to obtain an historic view of past bound sockets, in addition to ones actively bound. By scanning for ADDRESS_OBJECTs instead of walking the flinks of these structures, it provides a deeper view of system network activity. False positives are quite common in the output of sockscan, so investigator beware!



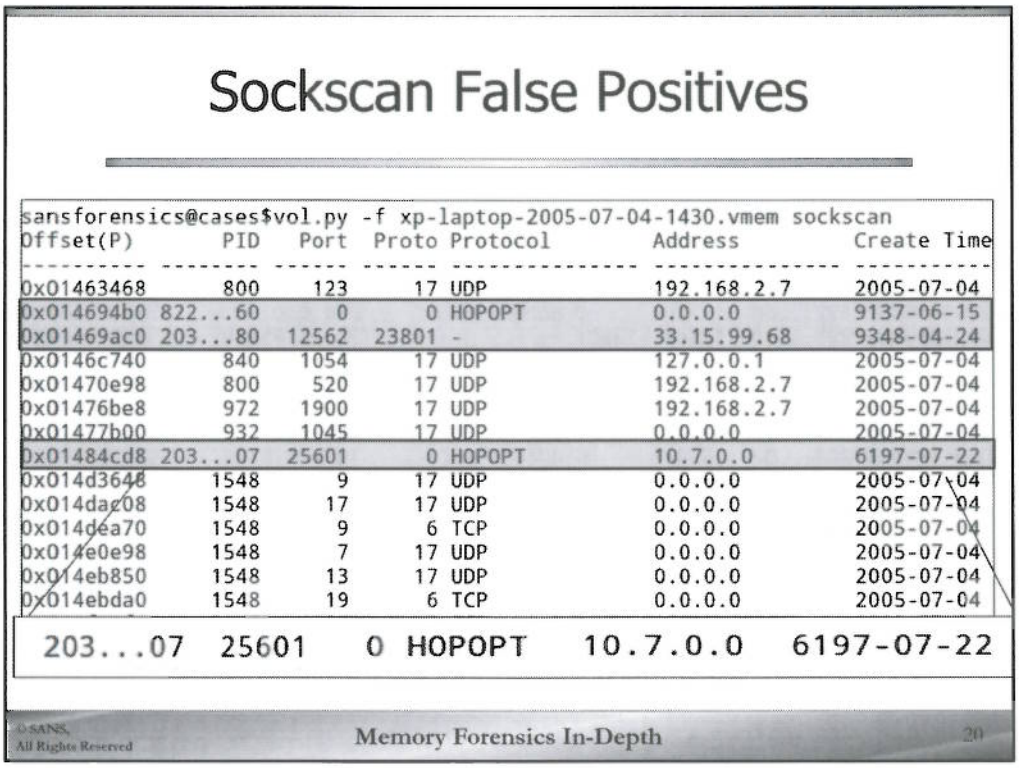
Along with walking the list of socket objects, we can also do a brute force search for them. Socket objects have the pool tag TCPA and can be found like any other pool tag. There are, however, far more false positives when searching for socket objects. Try it, you'll see. Run the sockscan plugin on the xp laptop image, like this:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
sockscan
```

You should see output which starts with:

```
Offset     PID    Port     Proto    Address      Create Time
---------- ------ ------   -------- ------------ ---------------------------
0x01463468   800   123    17 UDP         192.168.2.7      2005-07-04 18:22:08
0x014694b0 822211260     0      0 HOPOPT    0.0.0.0        9137-06-15 12:50:39
0x01469ac0 203500080 12562 23801 -    33.15.99.68      9348-04-24 22:55:49
0x0146c740   840  1054    17 UDP         127.0.0.1        2005-07-04 18:22:11
0x01470e98   800   520     17 UDP         192.168.2.7      2005-07-04 18:22:08
0x01476be8   972  1900    17 UDP         192.168.2.7      2005-07-04 18:22:08
```

The output fields should be exactly the same as the sockets plugin. The only difference is that we no longer have the option to get the virtual address of the ADDRESS_OBJECT structure. Because we're scanning we can only get the physical offset of the structure.



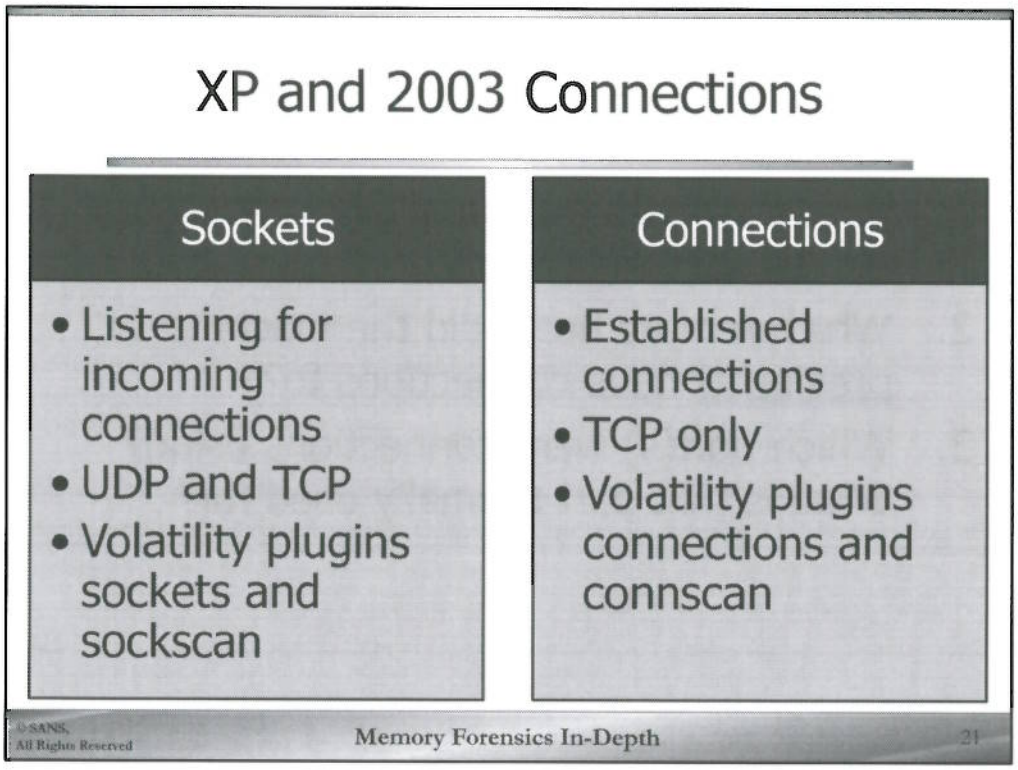
Let's look at the output a little more closely. That first line looks fine. In fact, if you look back at the output of the sockets plugin you'll see the same socket at virtual address 0x81463468. Notice how the virtual address has the same last three digits at the physical offset 0x1463468. That's a good sign.

But the second and third lines of the sockscan output just don't make sense. Those PID values are very odd. And according to this, the sockets were created in the year 9137 and 9348, respectively.

```
 Offset      PID     Port    Proto            Address        Create Time
---------- ------ ------ ---------------- -------------- --------------------
------
0x01463468    800    123      17 UDP       192.168.2.7         2005-07-04
18:22:08
0x014694b0 822211260  0       0 HOPOPT     0.0.0.0             9137-06-15
12:50:39
0x01469ac0 203500080 12562    23801 -      33.15.99.68         9348-04-24
22:55:49
0x0146c740    840    1054     17 UDP       127.0.0.1           2005-07-04
18:22:11
0x01470e98    800     520     17 UDP       192.168.2.7         2005-07-04
18:22:08
0x01476be8    972    1900     17 UDP       192.168.2.7         2005-07-04
18:22:08
```

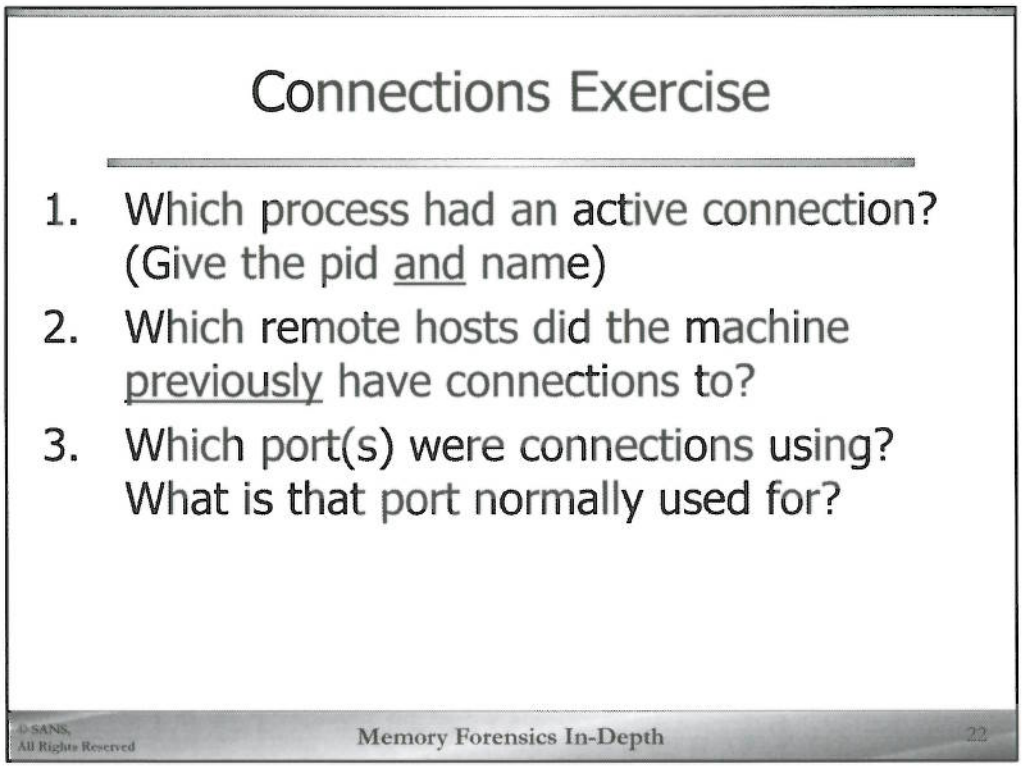
Because sockscan is a brute-force scanner, it often comes up with invalid data. Usually this is due to leftovers from pool memory which have been partially overwritten. But just be on the lookout for wacky values!

.



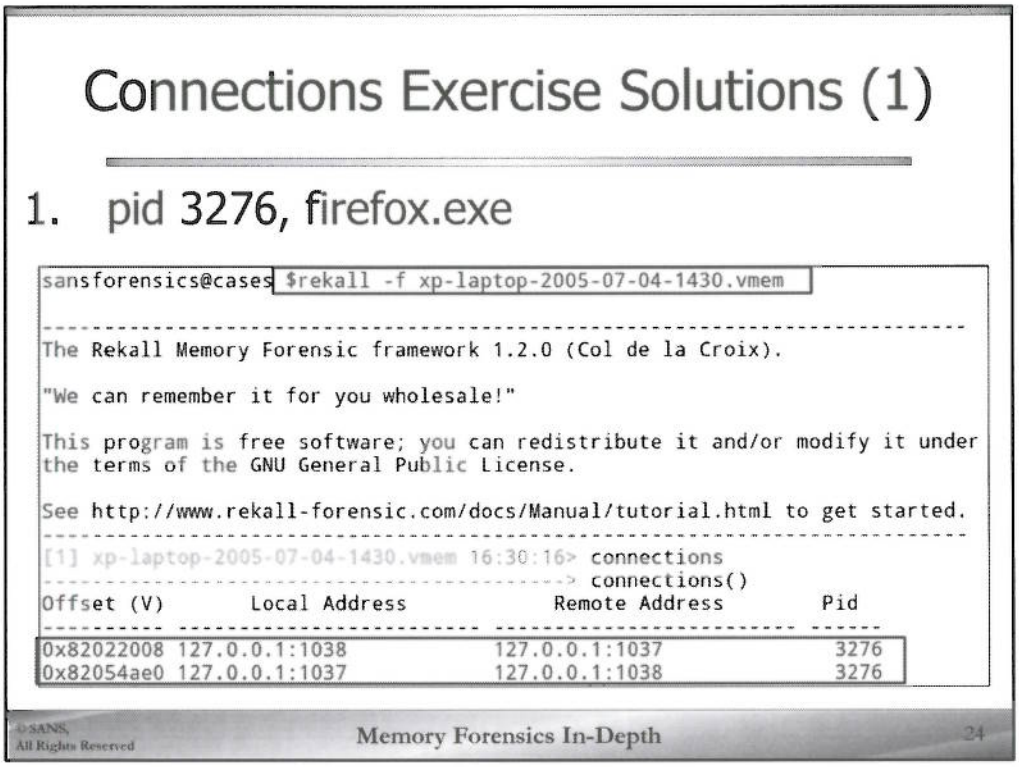
We're now going to switch gears and look at the TCPT_OBJECT structures in memory images. These structures recorded the TCP connections on Windows XP and Windows 2003 systems. The process of list walking and scanning for these structures is very similar to how we list walked and scanned for the ADDRESS_OBJECT structures. We walk the list by searching for what looks like a list head inside of the tcpip.sys module. We scan for these structures by searching for pool tags with the tag TCPT.

Two differences in the output of the connections and sockets Volatility plugins is that you won't see a field for protocol or a creation time/date in the output of the connection walking/scanning plugins. Remember that all of these connections are using TCP.



We're going to skip doing a walk-through here and jump straight into an exercise. Remember, what we're doing with connections is exactly the same with what we did for sockets in the previous hands on. Use the xp-laptop memory image and the connections and connscan plugins to answer the following questions.

This page intentionally left blank.



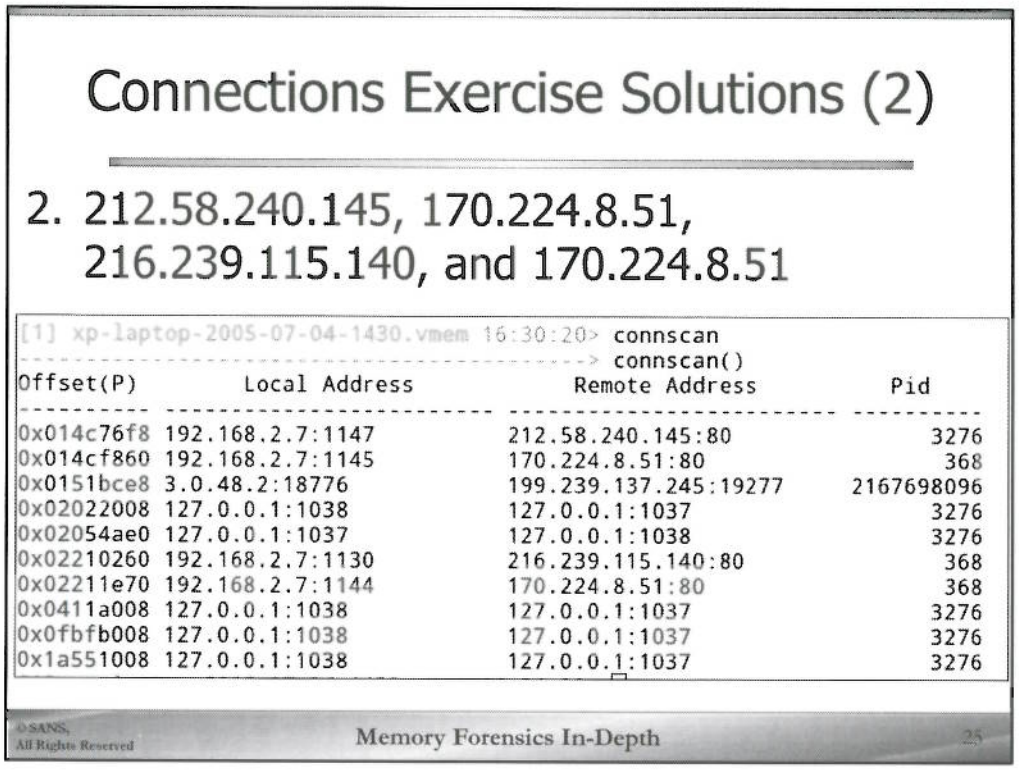
1. Which process had an active connection? (Give the pid and name.)

You can see the active connections by walking the list using the 'connections' plugin. Here's the command line and output:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
connections
 Offset(V)  Local Address             Remote Address            Pid
---------- ------------------------ ------------------------ ------
0x82022008 127.0.0.1:1038           127.0.0.1:1037             3276
0x82022008 127.0.0.1:1037           127.0.0.1:1038             3276
```

These are the active connections. Looks like pid 3276 was talking to itself. What was pid 3276? We can find out by running the pslist command and either searching through the output manually or using the grep command:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
pslist | grep 3276
0x8133d810 firefox.exe     3276   2392      7      189 2005-07-04 18:21:11
```

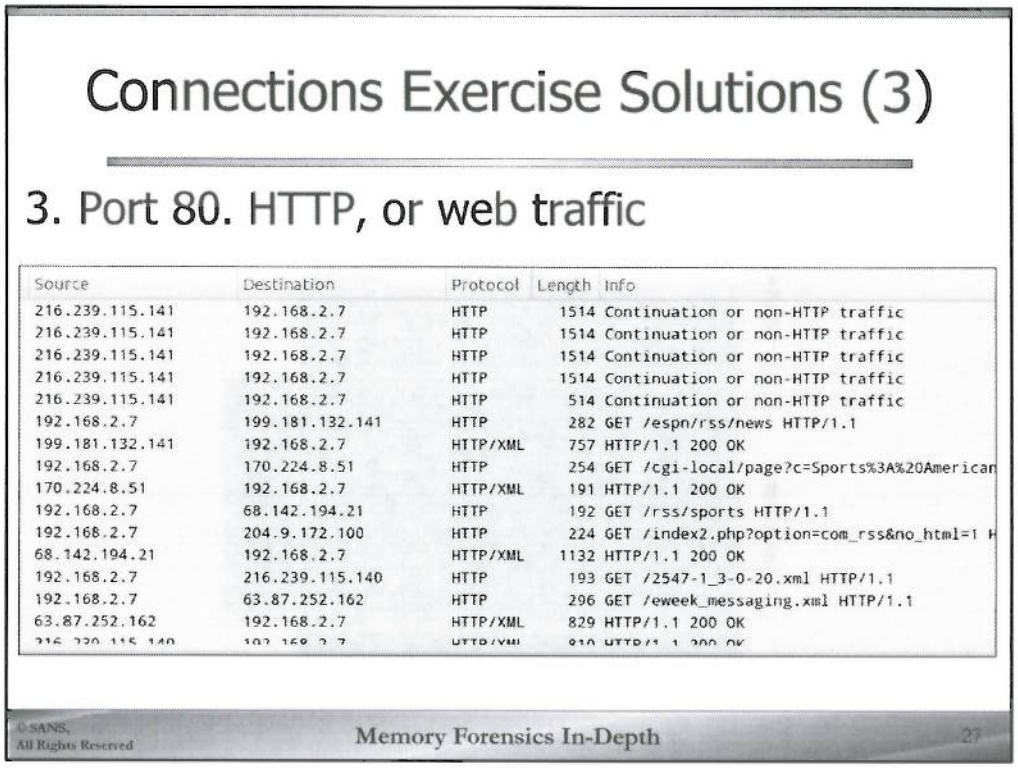


2. Which remote hosts did the machine previously have connections to?

We can see all of the connection objects by scanning for them with the connscan plugin. Here is the command line and output:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
connscan
 Offset        Local Address                Remote Address
Pid
----------       -------------------------       ------------------------
-           ------
0x014c76f8 192.168.2.7:1147          212.58.240.145:80          3276
0x014cf860 192.168.2.7:1145          170.224.8.51:80              368
0x0151bce8 3.0.48.2:18776            199.239.137.245:19277   2167698096
0x02022008 127.0.0.1:1038            127.0.0.1:1037                3276
0x02054ae0 127.0.0.1:1037            127.0.0.1:1038                3276
0x02210260 192.168.2.7:1130          216.239.115.140:80       368
0x02211e70 192.168.2.7:1144          170.224.8.51:80             368
0x0411a008 127.0.0.1:1038            127.0.0.1:1037                3276
0x0fbfb008 127.0.0.1:1038              127.0.0.1:1037                  3276
0x1a551008 127.0.0.1:1038            127.0.0.1:1037                3276
```

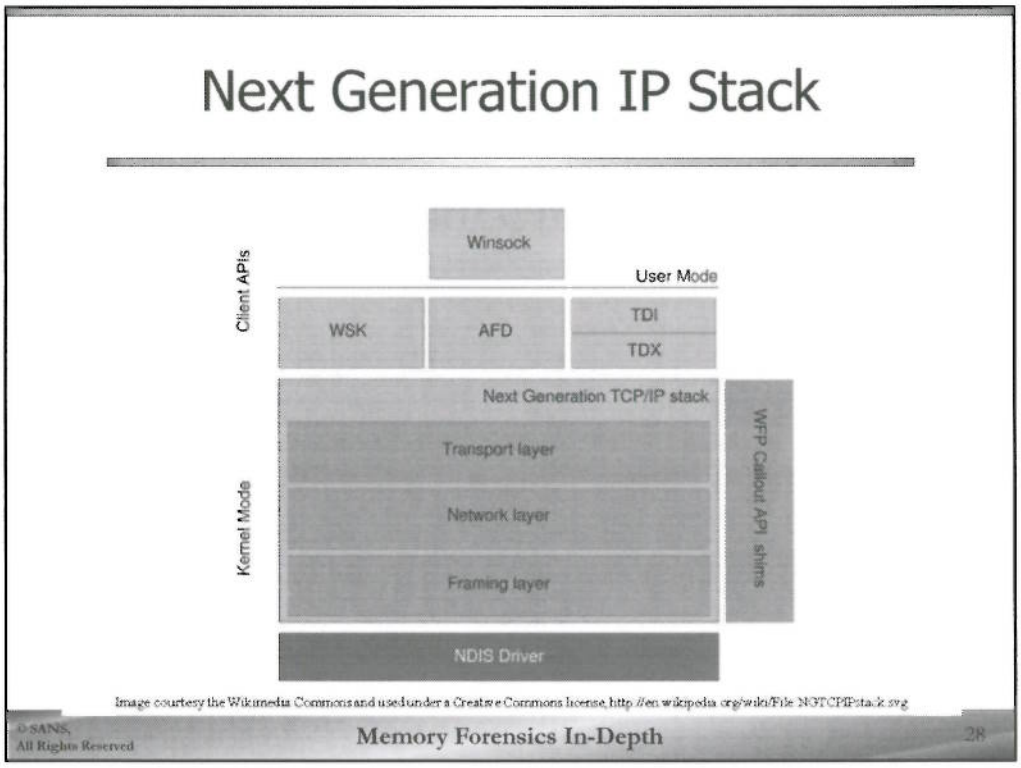
We can skip the connections which are solely from and to 127.0.0.1, that's the localhost. We can also throw out the connection from the host 3.0.48.2. That pid is just nonsensical, and IP addresses don't make much sense either. Chances are this was just something which got past the sanity checks. The addresses which are left are 212.58.240.145, 170.224.8.51, 216.239.115.140, and 170.224.8.51.



3. Which port were connections using? What is that port normally used for?

The port numbers follow the IP addresses for both the local and remote addresses. The local ports are all high ports, which is normal. The remote ports are all port 80. There are many possible ways to look up port values, such as the /etc/services file on your SIFT workstation. The canonical reference is at http://www.ietf.org/assignments/port-numbers.

Shown in the screenshot above is the portion of the extracted packets.pcap file that was parsed from the xp-laptop memory image by Bulk Extractor. The network connections seen in the connscan output are validated as well as other connections to remote servers via HTTP.



Microsoft introduced their "Next Generation TCP/IP Stack" in Windows Vista, and has used it in all of their operating systems since. The networking components are built out of several layers which work with one stacked on top of the other. As a result, the networking components are often referred to as the networking stack. The rewrite was intended to improve performance, the user experience, add better support for IPv6, and continue Microsoft's domination of the operating systems market. (They never mention that in the promotional literature, but it's true.)

The new IP stack changed the networking architecture and introduced several new features. The new architecture supports three APIs for accessing network functions, Winsock (which in turn calls the Ancillary Function Driver, of AFD, to get work done), a Winsock Kernel mode (WSK), and a Transport Driver Interface (TDI). These all sit on top of functionality in the new, unified tcpip.sys, which handles the transport, network, and framing layers of the OSI model. Each of these layers support the Windows Filtering Platform (WFP) for packet inspection and firewalling. We could do an entire class on the WFP, but for now, your best bet is to consult the Microsoft documentation at http://msdn.microsoft.com/en-us/windows/hardware/gg463267.aspx.

The new architecture is modular, and allows for other components to be inserted at places in the stack. It also allows, and it's scary to admit it took this long, the user to reconfigure the networking system without needing to restart the computer.

If you'd like to read more about the new networking stack, see Microsoft's documentation at http://technet.microsoft.com/en-us/network/bb545475.

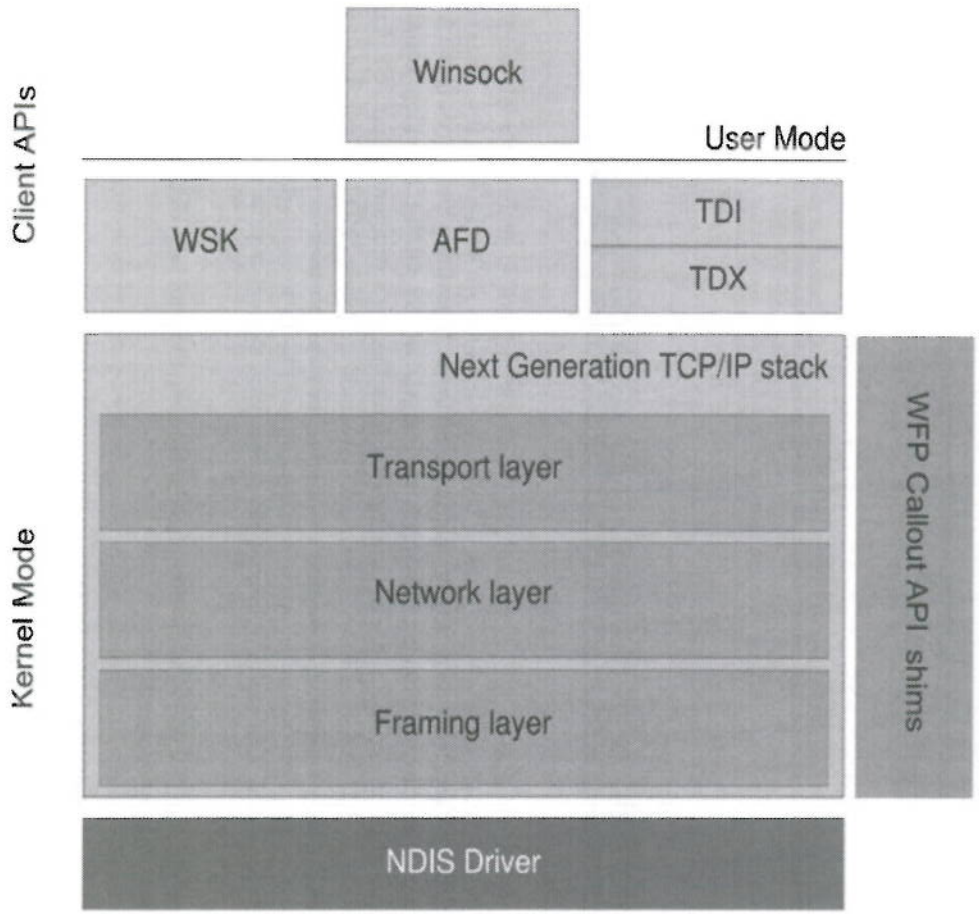
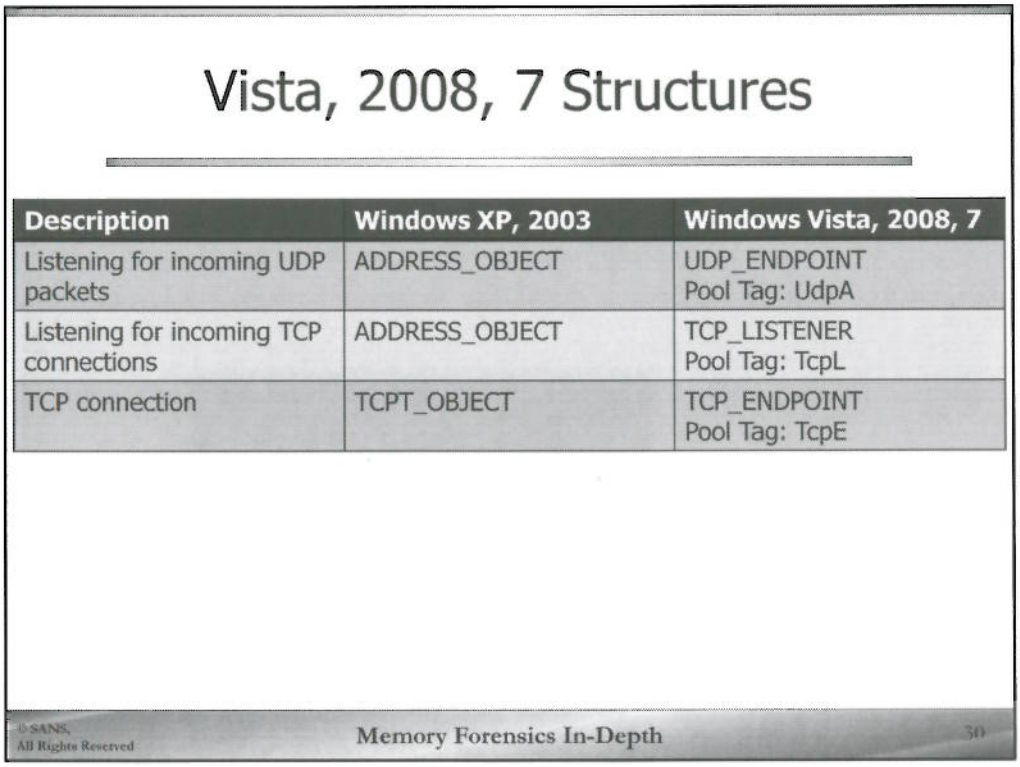


Image courtesy the Wikimedia Commons and used under a Creative Commons license, http://en.wikipedia.org/wiki/File:NGTCPIPstack.svg

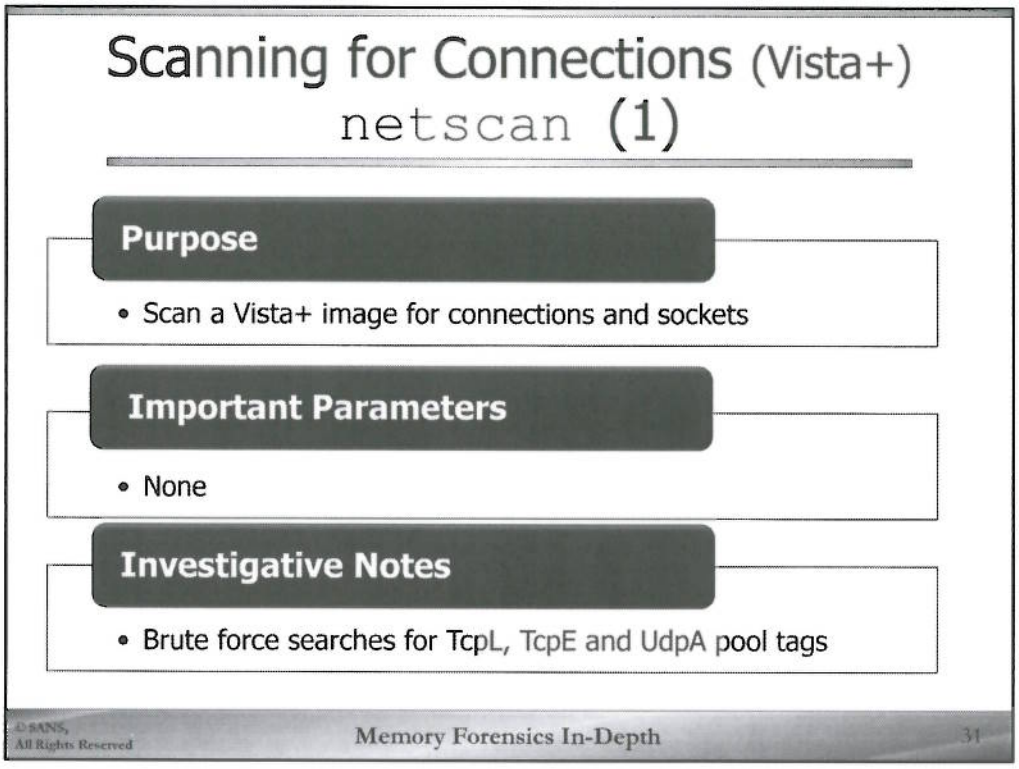


The changes in the Next Generation IP stack changed the structures forensic analysts are interested in. From the forensics perspective, the new network stack changed the structures we are looking for. The symbols in tcpip.sys are still undocumented—there was no improvement there. If anything, the names in the new tcpip.sys are more confusing than they were before. Previously we had ADDRESS_OBJECTS which recorded information about listening for TCP and UDP connections. Now there is a structure called TCP_LISTENER, which records listening TCP sockets. But the structure for recording listening UDP sockets is called UDP_ENDPOINT. This is confusing, because it's similar to the new structure for recording TCP connection information. (Remember, UDP has no connection information!) The TCP connection structures used to be TCPT_OBJECT, but in the Next Generation stack these structures are called TCP_ENDPOINTs.

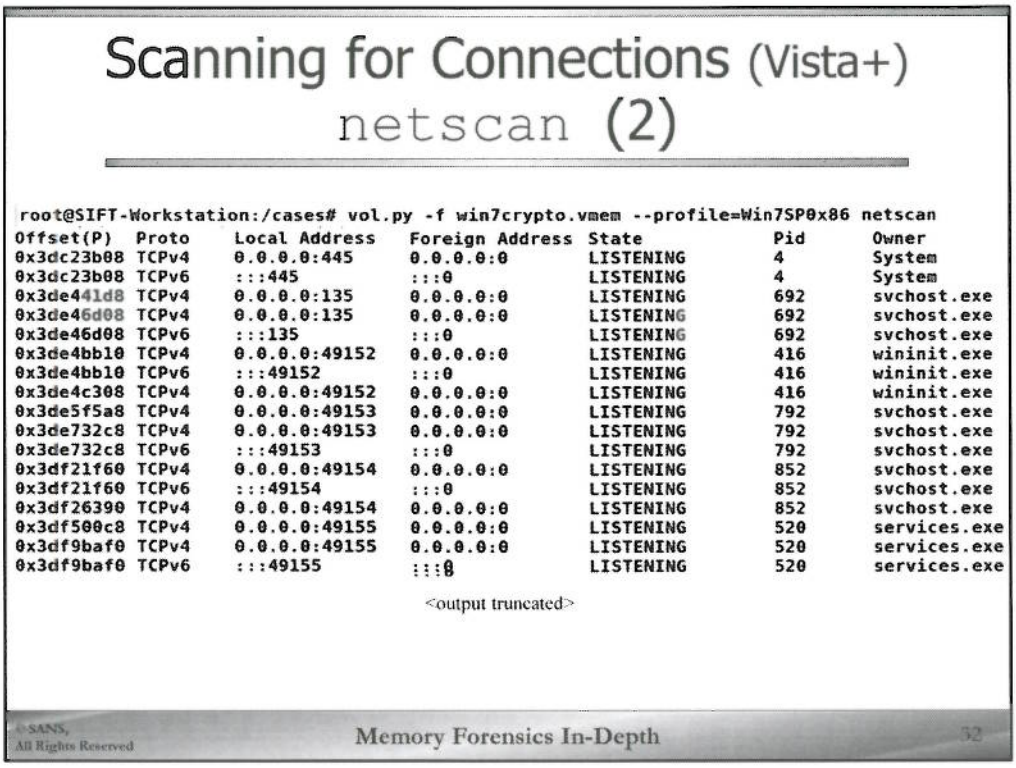
These three structures are still stored in the non-paged pool using the pool tags UdpA, TcpL, and TcpE, respectively. Capitalization matters!

Below is the structure for the TCP_ENDPOINT:

```
In [2]: dt("_TCP_ENDPOINT")
'_TCP_ENDPOINT' (None bytes)
0x0  : CreateTime          ['WinTimeStamp', {'is_utc': True, 'value': 0}]
0xc  : InetAF              ['pointer', ['_INETAF']]
0x10 : AddrInfo            ['pointer', ['_ADDRINFO']]
0x14 : ListEntry           ['_LIST_ENTRY']
0x34 : State               ['Enumeration', {'target': 'long', 'choices': {0: 'CLOSED', 1: 'LISTENING', 2: 'SYN_SENT', 3:
'SYN_RCVD',                    4: 'ESTABLISHED', 5: 'FIN_WAIT1', 6: 'FIN_WAIT2', 7: 'CLOSE_WAIT', 8:
'CLOSING', 9: 'LAST_ACK', 12:                  'TIME_WAIT', 13: 'DELETE_TCB'}}]
0x38 : LocalPort           ['unsigned be short']
0x3a : RemotePort          ['unsigned be short']
0x174 : Owner              ['pointer', ['_EPROCESS']]
```



There is a single plugin in Volatility for parsing all of the network structures. The plugin 'netscan' works on Windows Vista, 2008, and 7 memory images, during a brute force scan for the TcpL, TcpE, and UdpA pool tags. The output of this plugin is designed to imitate the output of netstat. (It's also similar to TCPview, http://technet.microsoft.com/en-us/sysinternals/bb897437, which is a handy system utility for viewing Windows network connections!)



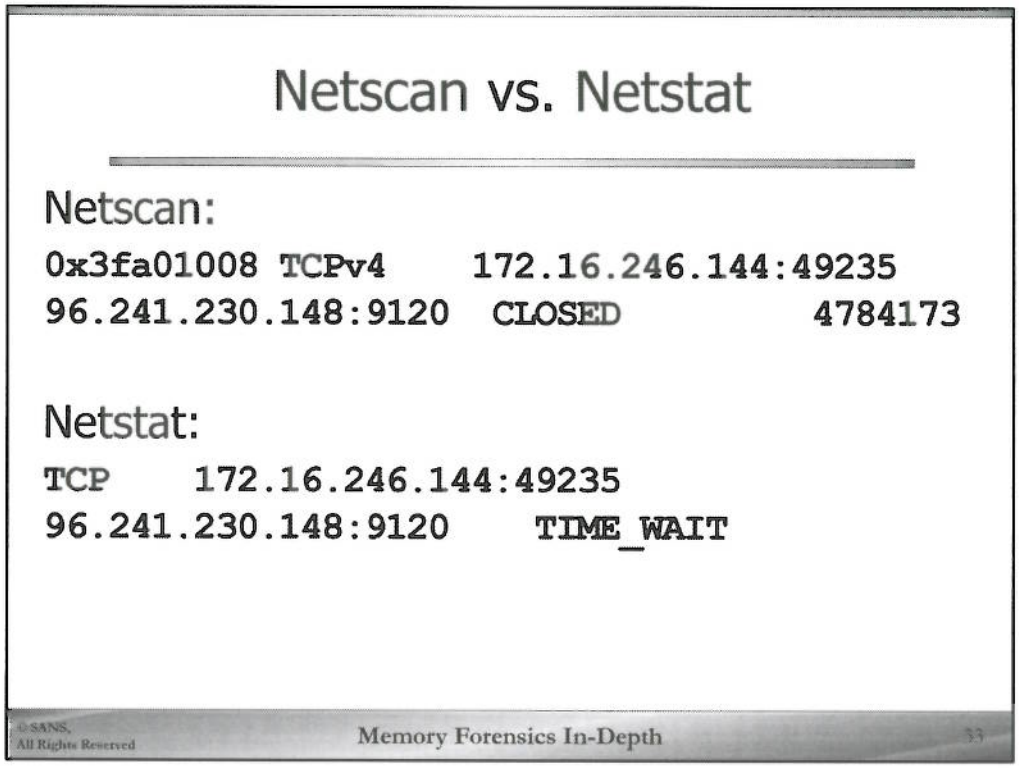
We're going to do a hands on exercise now using that plugin on the win7crypto memory image. Here's the command line we'll use:

```
$ vol.py -f /cases/win7crypto.vmem --profile=Win7SP0x86 netscan
```

There's a lot going on here! First, we get the physical offset of the structure being parsed. This is a brute force scanner, so we only get the physical offset—no virtual addresses. Next is the protocol, which can be either TCPv4, TCPv6, UDPv4, or UDPv6. This field is a combination of two different protocols layers. The first layer is either UDP or TCP. The second is IPv4 or IPv6. This field concatenates the two values. The next two fields in the output are local address and port number, and the remote address and port number. The state of connection is given, as you might see it in netstat*. Next we get the process id and process name are given for the owning process. Finally, for UDP listeners we get time they were created.

Looking at the output for netscan, for this memory image, we can see there was some web surfing going on from Internet Explorer. Notice all of the closed connections from iexplore.exe to remote hosts on port 80. You can do a reverse DNS lookup on those IP addresses to see which hosts the computer connected to.

* The netscan plugin was written by reversing engineering how netstat worked. When in doubt, do it the way Microsoft did. See http://mnin.blogspot.com/2011/03/volatilitys-new-netscan-module.html for the full write-up.



The netscan plugin is designed to imitate the interface of the Windows netstat program, but actually provides slightly different information. Here's a comparison of what we get from the netscan plugin versus the netstat command. We've edited the data down to just some highlights. First netstat:

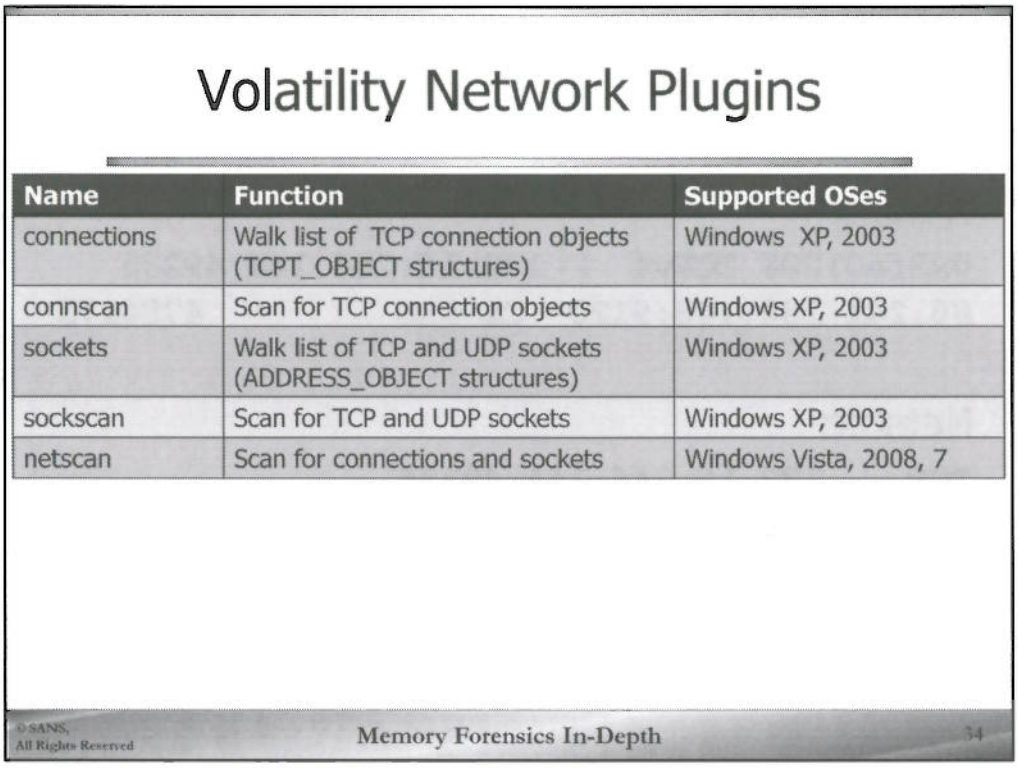
```
$ python vol.py -f ~/Documents/Virtual\
Machines.localized/win7sans.vmwarevm/win7sans-a5734d2a.vmem --profile=Win7SP1x86
netscan

Offset     Proto    Local Address      Foreign Address     State  Pid     Owner    Created
0x3fa01008 TCPv4 172.16.246.144:49235 96.241.230.148:9120 CLOSED 4784173
```

The 4784173 number is the pid. The remaining fields are blank.

The same data as displayed by netstat:

```
C:\> netstat -anob

 Proto  Local Address          Foreign Address        State
 TCP    172.16.246.144:49235   96.241.230.148:9120    TIME_WAIT
```

In this example the user had executed the NSRLlookup program, which uses the default port of 9120. Both netstat and netscan show the socket in a closed state, although netscan is more specific about the nature of it. Also note that netstat attempts to display a process id number, even though we can tell it's obviously invalid. The nsrllookup program had already exited.

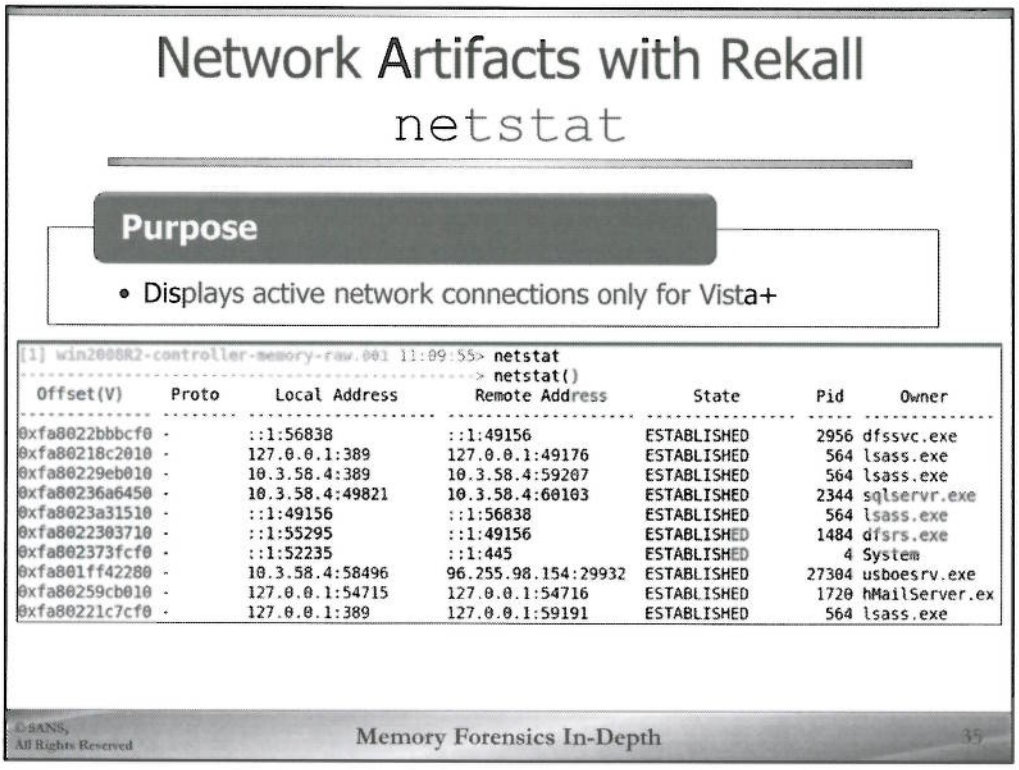


Here's a summary of the networking plugins in Volatility we have covered. For more details on any of them, you can see http://code.google.com/p/volatility/wiki/FeaturesByPlugin.

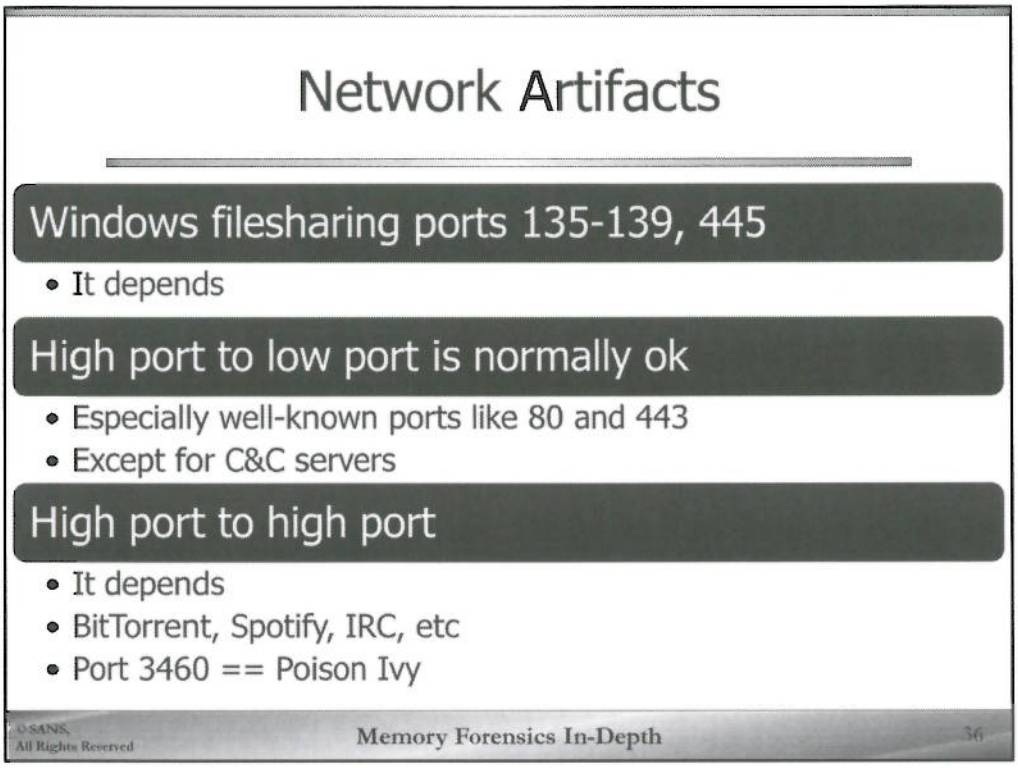
Be cognizant of attempting to run plugins specific to other OS versions than that which you are attempting to parse. The error messages can be less than intuitive and at times, there may be no output to the screen, making an examiner think that there were, in fact, no network connections, when really he just choose the wrong plugin for the version of Windows needed.

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
netscan
```

If you do this on some versions of Volatility, you run the risk of making the framework very angry. The list walking plugins, especially, can get horribly confused, attempting to parse lists which have millions of entries. Although you won't damage your computer, the runaway process can soak up all of the free memory on your system.



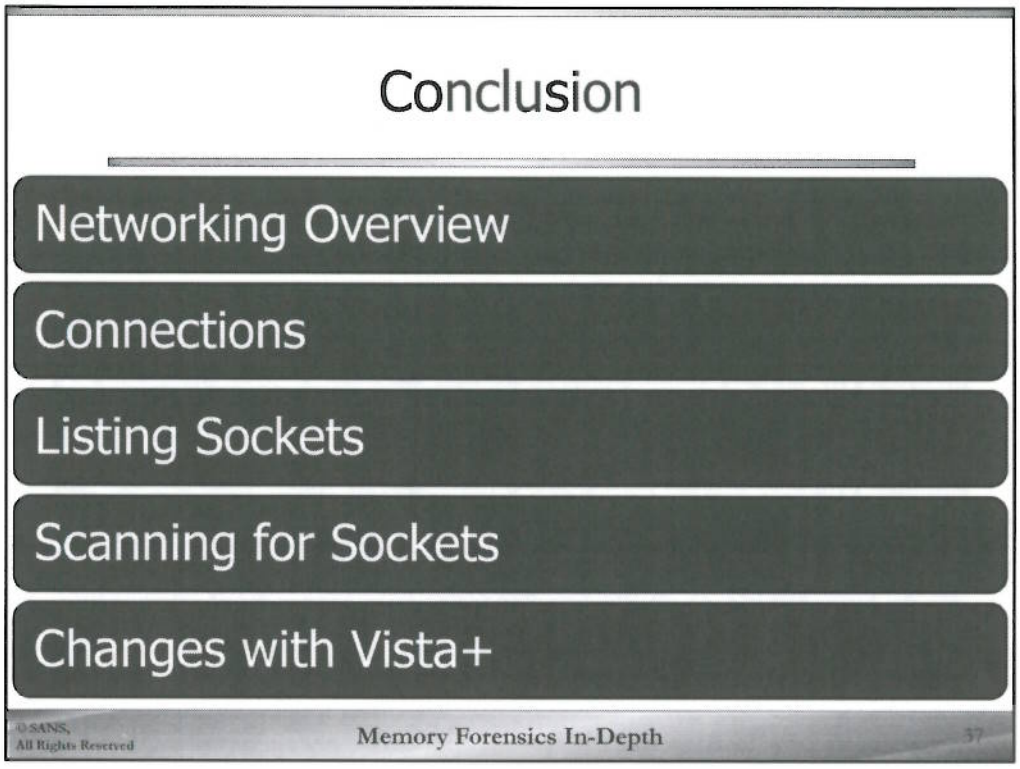
Although there is only one Vista+ connections/sockets plugin included in Volatility's framework (netscan), Rekall includes additional capabilities. The plugin **`netstat`** allows the examiner to enumerate active network connections, removing listening ports from the netscan output.



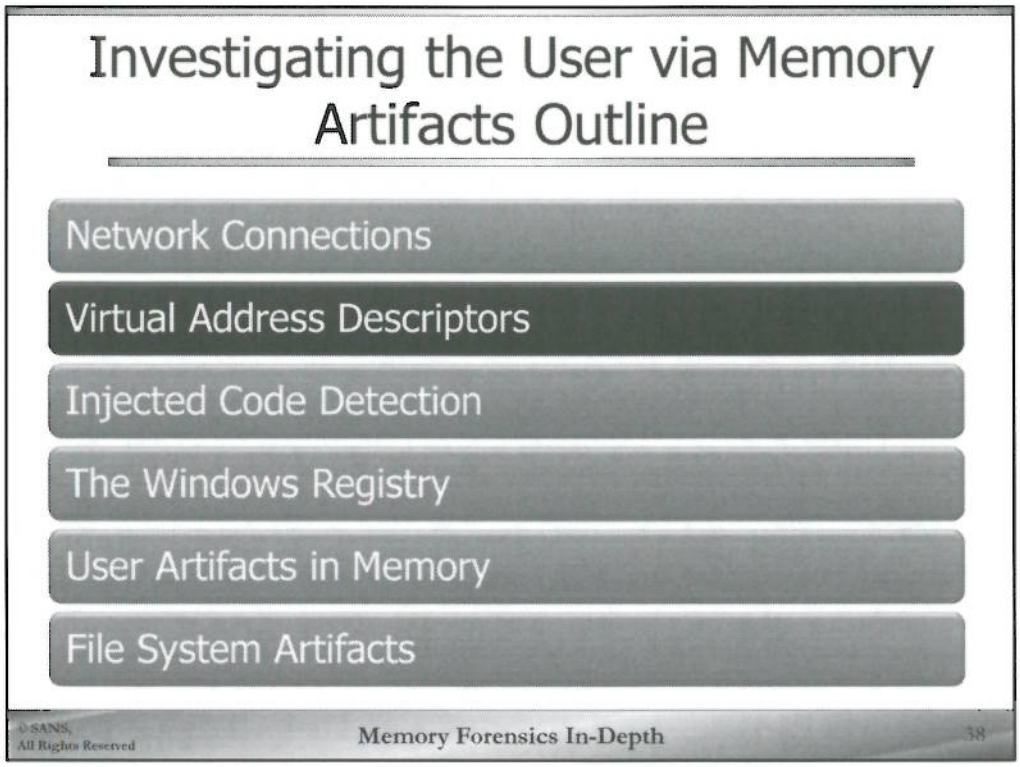
One of the most common things you'll see in your cases are sockets listening on the Windows filesharing ports, ports 135-139, 445. Although Windows filesharing can be a part of an enterprise, sometimes it's not. When you find it, be sure to ask if it's supposed to be there.

Legitimate traffic on a system usually looks like connections from high ports on the local system to low ports--under 1024--on a remote host. Admittedly, malware authors know this and will attempt to hide their traffic in ordinary web traffic. Are the sites being visited part of the user's normal activity? Are there connections to update.microsoft.com? some-random-university.edu?

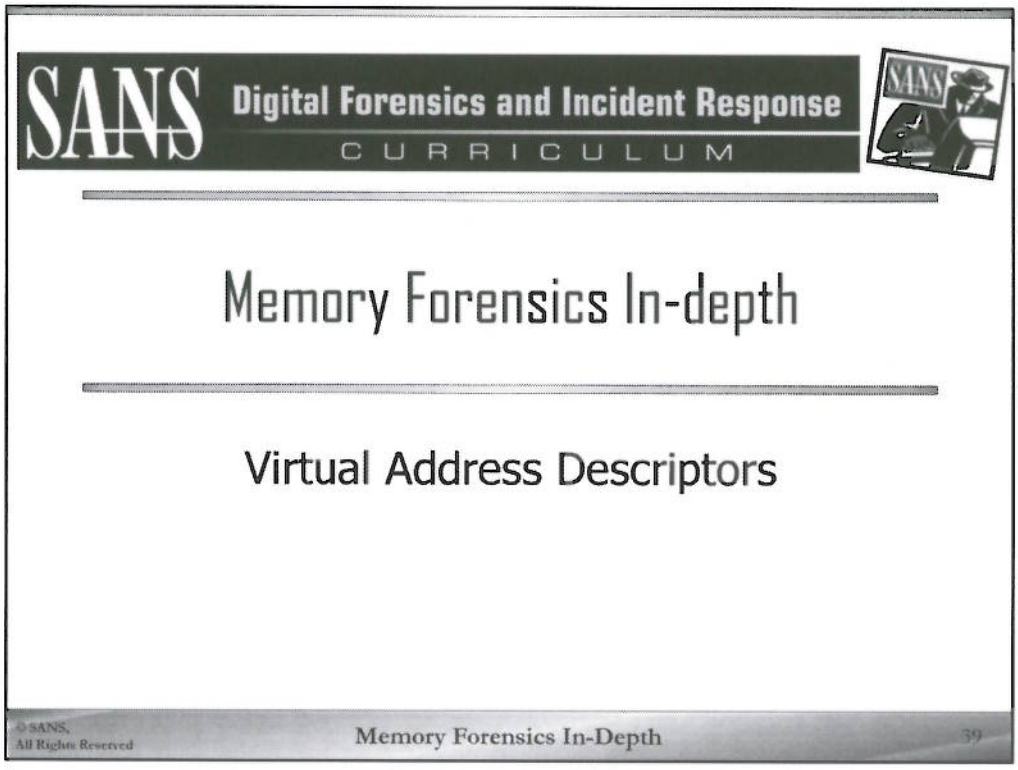
Traffic going from a high port on the local system to a high port on a remote system could be trouble, or it could be part of a legitimate service. Many popular and legitimate services use some ports. These include things like Spotify, Pandora, BitTorrent, and IRC chat (which may or may not be blessed depending on the environment!) But always be on the lookout for things like Poison Ivy. It defaults to communicating on port 3460.



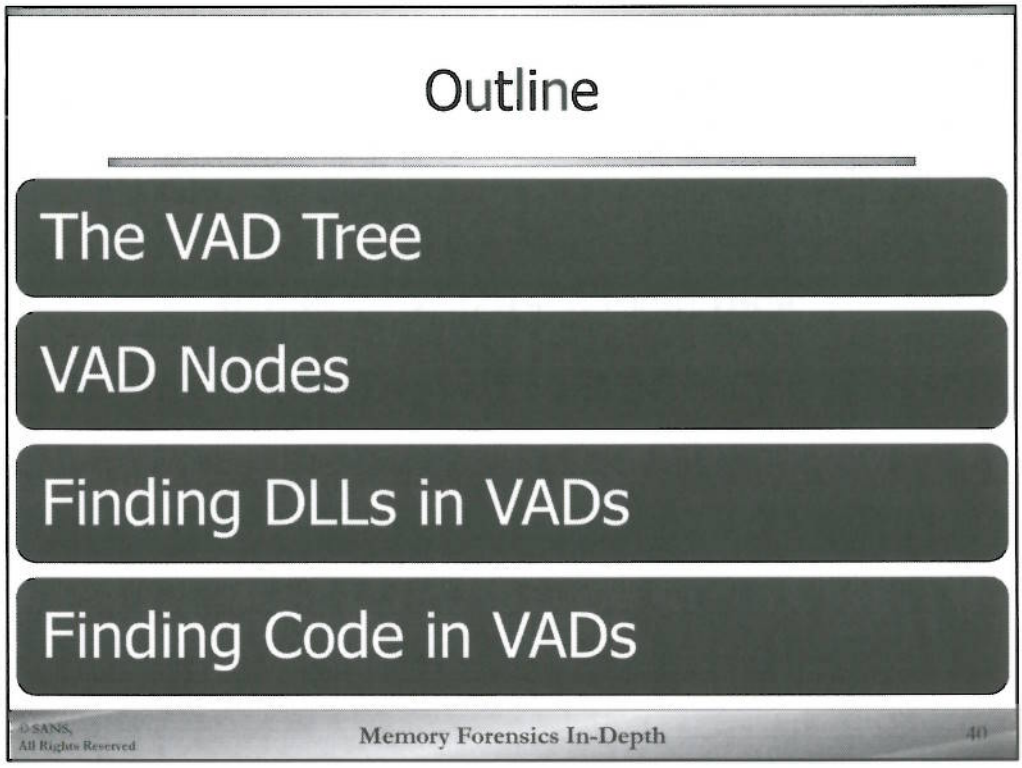
In this section we started with an overview of networking concepts, briefly describing the protocols involved, network stacks, addressing schemes, and port numbers. We put that knowledge to use by looking at the structures Windows XP and 2003 used to describe TCP and UDP sockets. A socket is the combination of a program binding to a port and waiting for other computers to connect on that port. The remote computer then communicates with the program. We both walked a list of sockets and did a brute force search for them. We then moved on to connections, which are only for TCP connections. We again both list walked and brute forced searched for these connections. Finally we discussed the changes made in Windows Vista with the Next Generation TCP/IP stack, and how this affected the socket and connection objects. We used a single plugin to scan for the new kinds of objects and display the results.



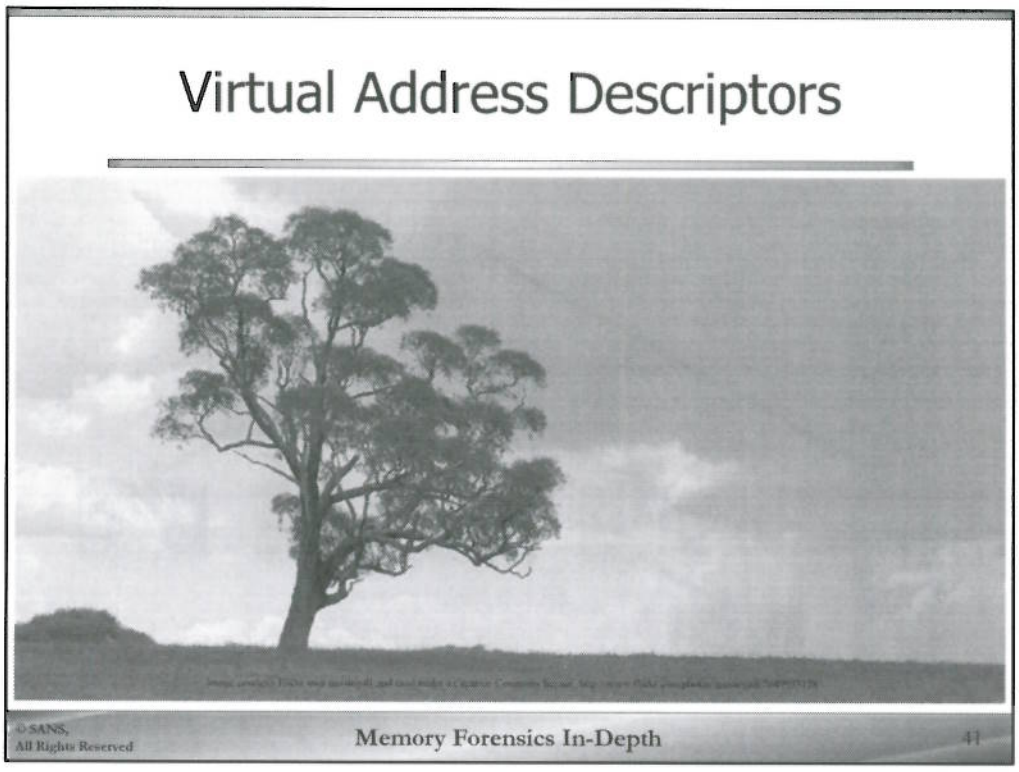
This page intentionally left blank.



This page intentionally left blank.

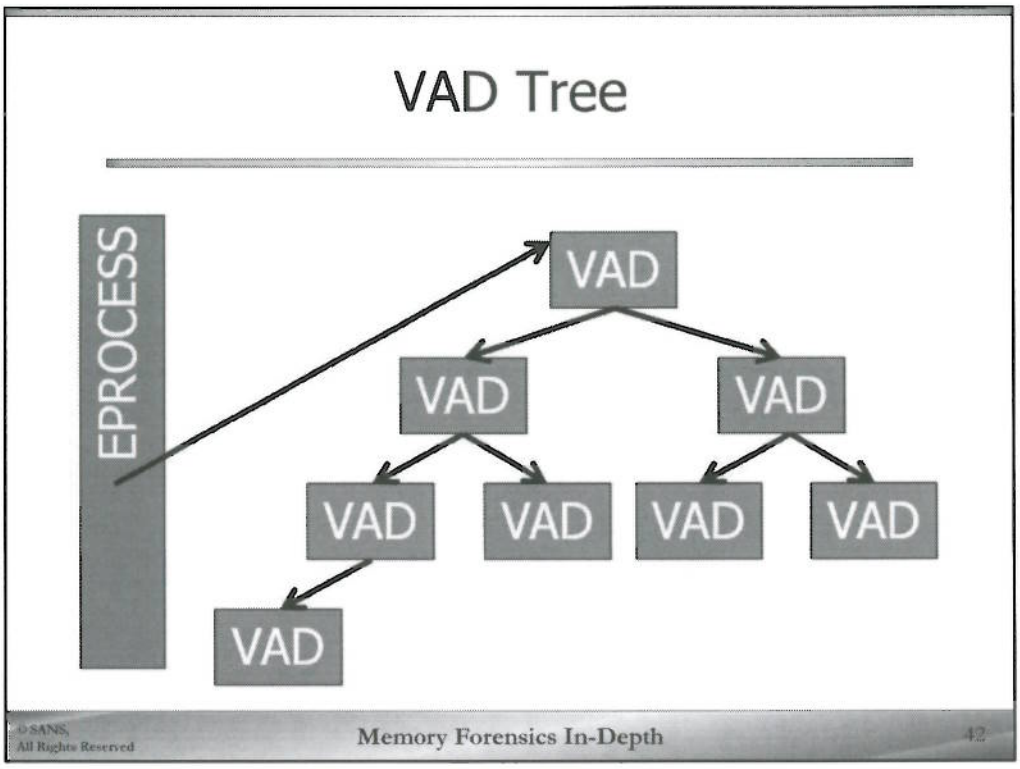


This page intentionally left blank.



Virtual Address Descriptors, or VADs, are how Windows keeps tracks of which ranges of virtual memory are allocated inside of a process, what they're allocated for, what permissions are allowed on that memory, and overall represent one of the best ways to find hidden evil inside of the operating system. In this section we're going to explore why Windows uses VADs, what data they hold, and how a forensic examiner can use them to find things which are amiss on the system.

The VADs are stored in a tree, with one tree per process, and hold the memory allocations for that process. In previous sections we have referred to how processes could allocate memory to hold malicious code or data. In this section we will see how to find those data and extract them. VADs are the road map for which kinds of data a processing is holding and where. Data can be "hidden" from the operating system, but it still has to be stored somewhere and made accessible for use. The VADs are the structure which ensures both of those states. Thus, it's the perfect mechanism for us to examine the memory of a process.



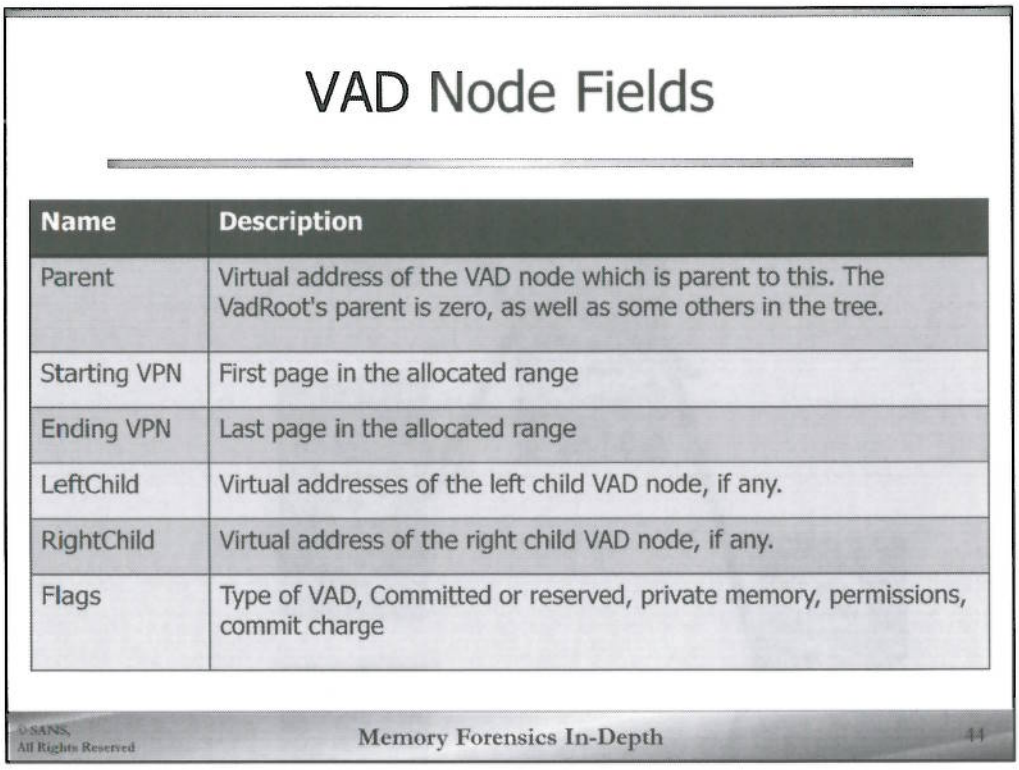
Each EPROCESS block has a member, VadRoot, which points to the node at the top of the VAD tree. Each node contains a range of virtual addresses, the permissions assigned to those pages, and pointers to left and right child nodes, if any. The nodes at the bottom of the tree do not have children, and so those pointers are zeros. Each node also has a pointer to its parent. We don't show this in the figure as we usually don't follow the parent pointers.

The VAD tree is a data structure called an AVL tree, which is a self-balancing binary tree. It was developed by two Soviet computer scientists, Adelson-Velskii and Landis; the name is the concatenation of their initials. In the average and worst case scenarios, this data structure always performs well. (Other data structures may perform much better in the average case, for example, but much worse in the worst case. You think of the AVL tree as "slow and steady wins the race.") When nodes are added or removed in the tree, the tree may need to be rebalanced. This happens well behind the scenes, and we will not delve into it. If you'd like to know more about AVL trees, check out the animation, complete with sound effects, at http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html.

The nature of the AVL tree is that each range under the left children of a node is lower than the current range. Each range under the right children is higher than the current range.

The ranges of memory are only for userspace. Kernel memory is handled by the kernel without using VADs. That being said, the kernel does *hold* VADs in userspace. That is, the kernel has memory allocated which appears in every process. Some of this is for the KUSER_SHARED_DATA structure which all processes can see. There is current research regarding the nature and purpose of this other kernel memory kept in user space.

VAD
VAD
VAD
VAD
VAD
VAD
VAD
VAD
VAD
PROCESS



Every VAD node has a basic set of fields. There are some VADs which have extra fields, which we'll get to in a few pages. The fields in every VAD include:

Parent: The virtual address of this node's parent in the VAD tree. The parent field of the VAD root is set to zero. Although not used to manage memory, the parent links make rebalancing the AVL tree much faster. As such, often times you will find VADs where the parent field is zero. This is completely normal.

Starting Virtual Page Number and Ending Virtual Page Number (VPN): These two values define the range of memory pages which are allocated for this VAD. The VPNs are stored as a page number, or in multiples of 0x1000 bytes. For example, the allocated memory range of 0x700000 to 0x799fff would be represented by a starting page of 0x700 and ending page of 0x799.

Left Child – The virtual address of the left child node, if any. If there is none, this value is zero. The nature of the AVL tree means that every Starting VPN and Ending VPN of the nodes under the left child is still less than the Starting VPN of this node. Continuing our example of a StartingVPN of 0x700, every node under the left child of this tree would have a Starting VPN and Ending VPN less than 0x700.

Right Child – The virtual address of the right child, if any. The opposite rule applies regarding the ranges here as under the left children. The Starting VPN of every node under the right child is guaranteed to be greater than the Ending VPN of this node, 0x7ff.

Flags – Holds several fields regarding the state of the VAD:

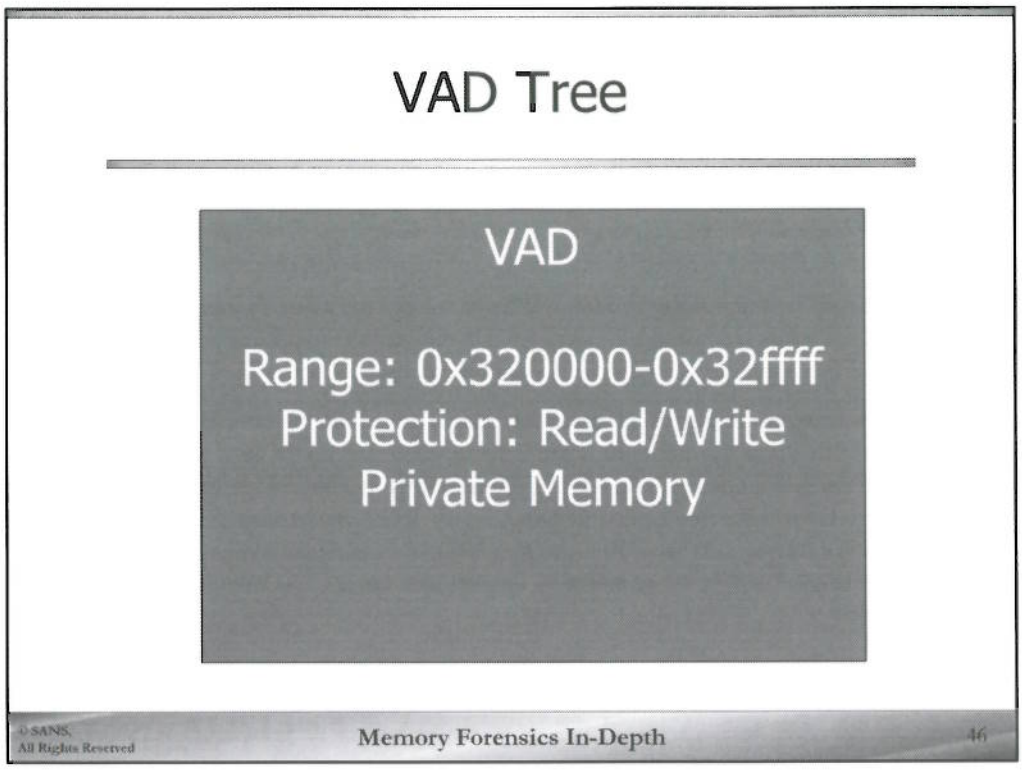
The type of VAD (short, regular, or long)

Whether this memory has been committed. Memory is reserved when a VAD is created. That is, the section of the virtual address space is blocked off from other threads attempting to create it. But resources are not committed to the memory range until the process requests to use it the first time. This lazy commitment system prevents resources from being allocated to things which never get used.

Whether this memory is private to this process or shared among more than one process. This happens often with DLLs, for example.

The permissions associated with the memory range. Whether or not the memory can be written to or executed, for example.

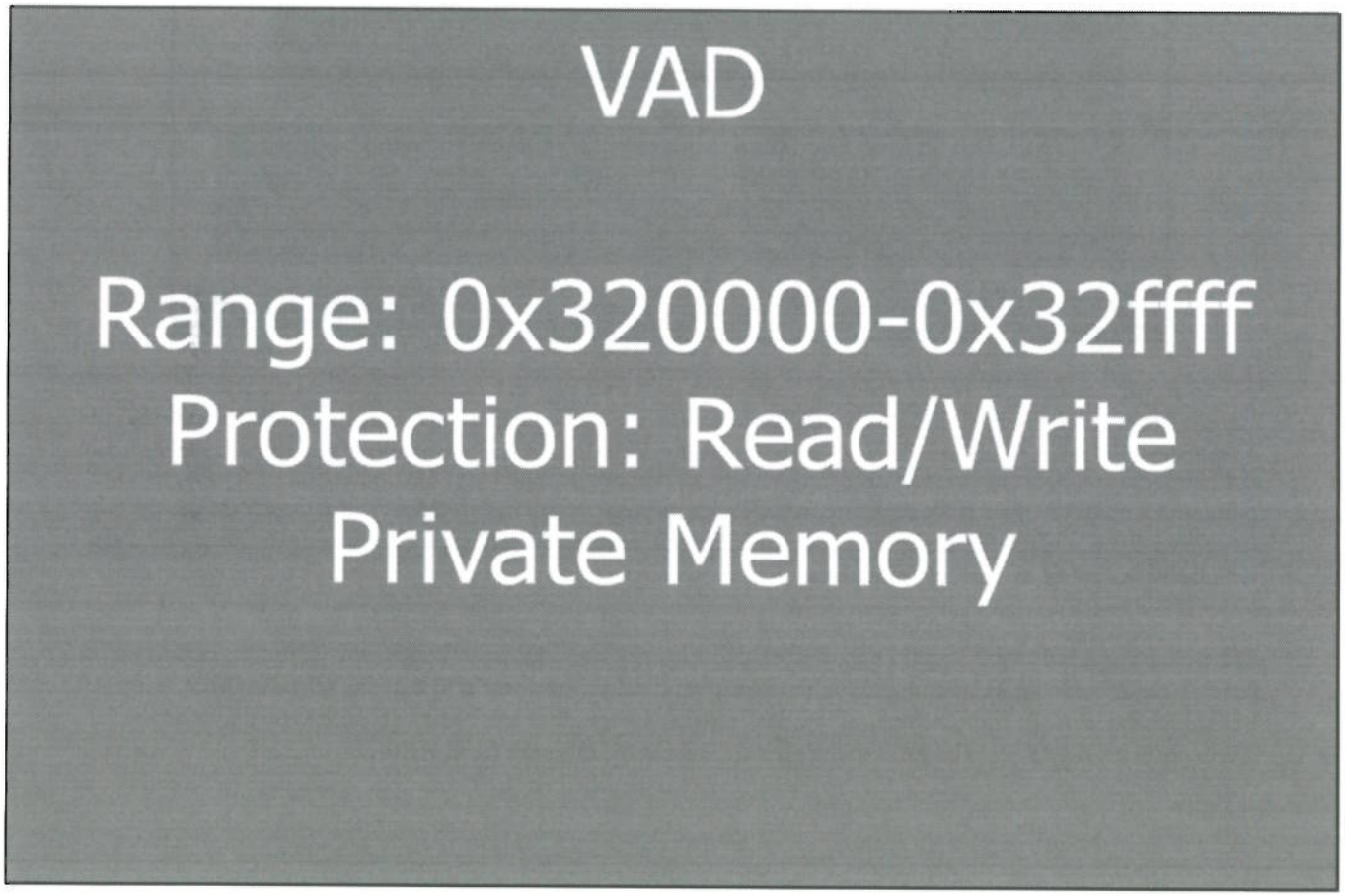
The commit charge. The commit charge is a measure of how much of the memory range is not backed by a file. In other words, how much of this memory range is being used and would need to be backed by the paging file. Each process, and the system as a whole, has a limit as to how much data can be stored in the paging file. If that limit is exceeded, the system runs out of virtual memory. The commit charge in the VAD is part of the overall commit charge accounting mechanism. From the examiner's perspective, the commit charge is a measure of how much data is actually being stored in the memory range. VADs with a large commit charge are probably holding data.

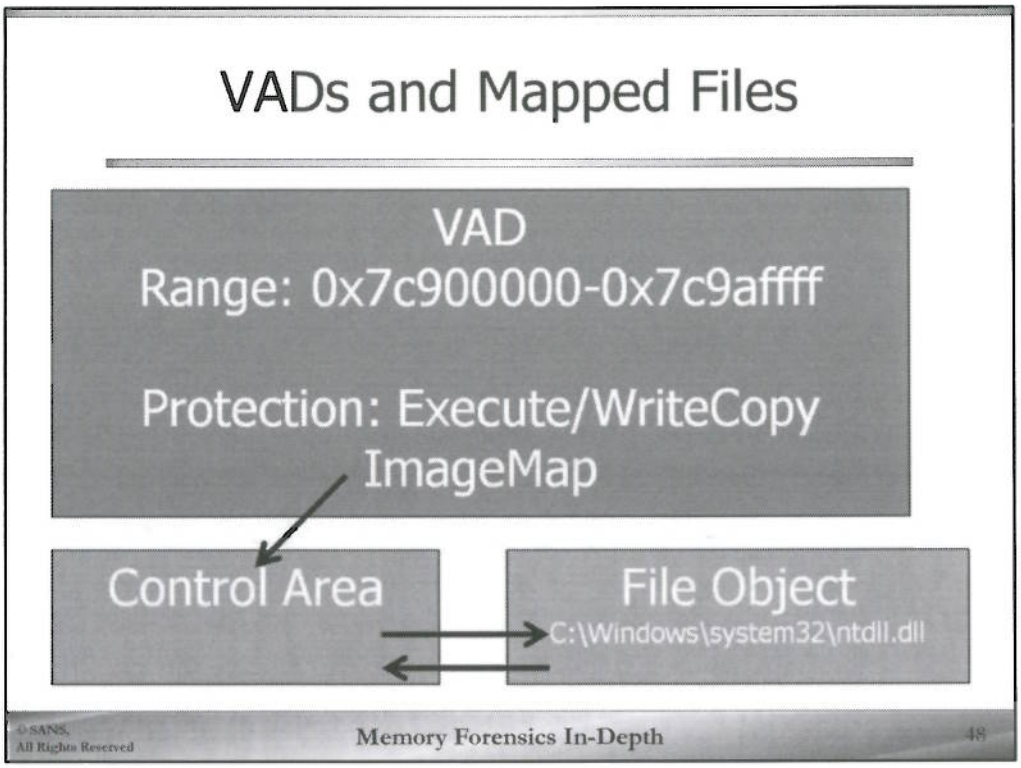


Ranges of memory which are not associated with a file on the disk are called Private Memory, and have a flag set in the VAD to indicate this. For example, if a process reserves ten pages of memory, loading a file from the disk, it would be given a memory range like 0x320000 to 0x32ffff. This range would be marked as Private Memory. But it would remain not committed until the first attempt to access this memory. When the process first attempted to access the memory, *then* Windows would create the page table entries necessary to store it, and actually reserve frames of physical memory to hold the data. Again, each frame is only committed as it's needed.

In the example above, the process has reserved the memory range 0x320000 to 0x32ffff, or ten pages of memory. The flags have been set such that this memory can be read or written, but not executed.

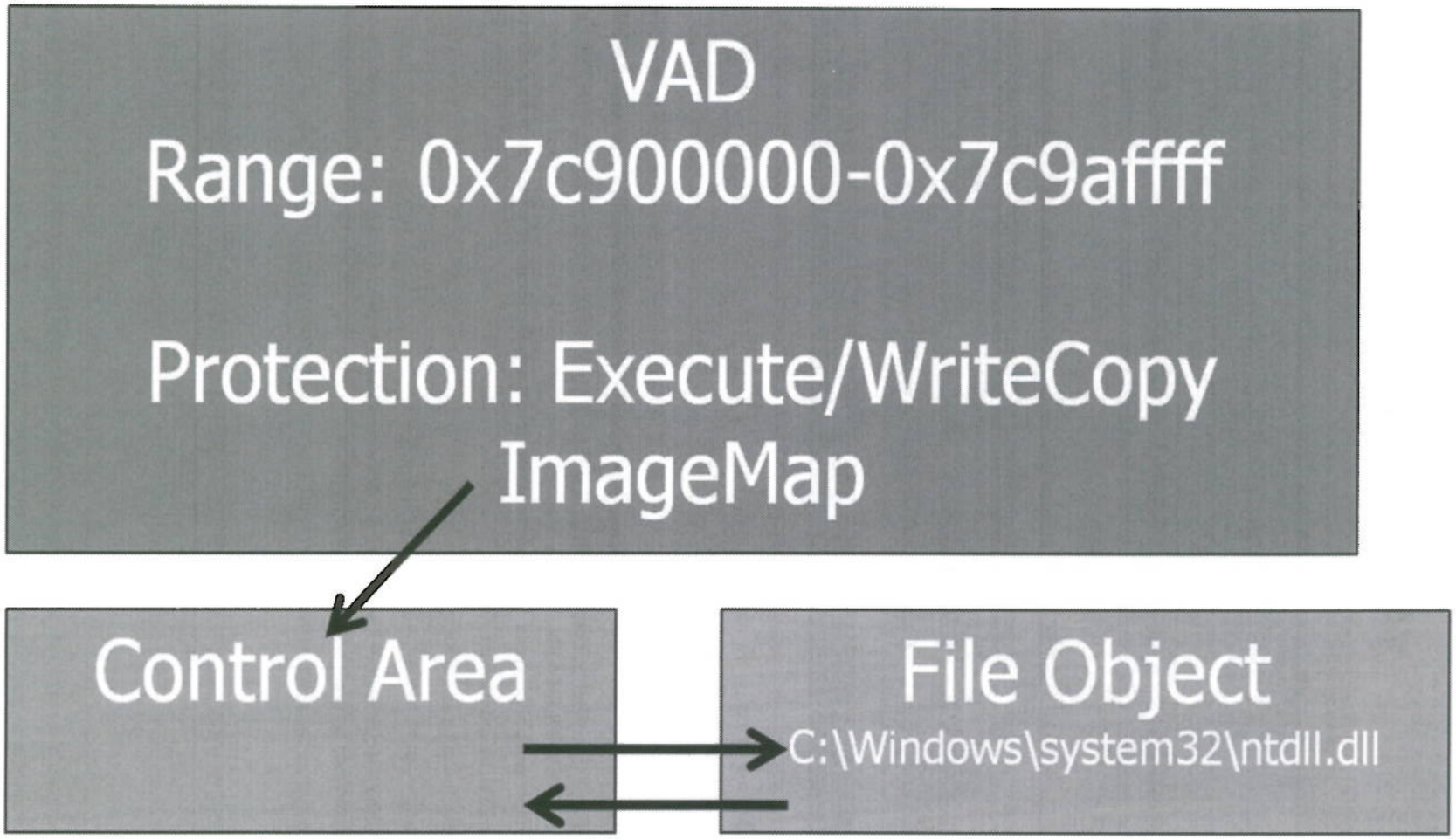
A word of caution here on the permissions, which will become important later on. Programmers can request memory with any permissions they like. If they ask for every permission, they will get it. For the sake of avoiding errors, many programmers *do* ask for every permission when they write their code. The result is that we see ranges of memory which are marked Execute/Read/Write. Often times there is not executable data on those pages. It is a common mistake to think that every executable range of memory contains executable code. We will see later on some tests which can be used to find executable ranges which contain executable code.

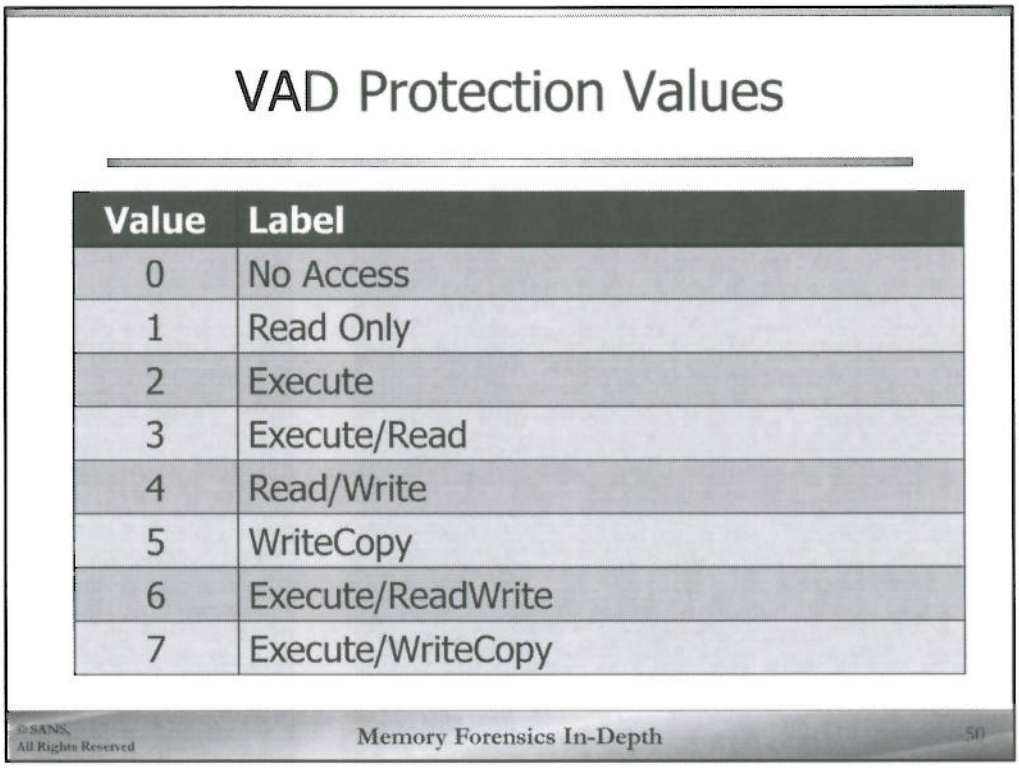

VAD
Range: 0x320000-0x32ffff
Protection: Read/Write
Private Memory



Ranges of memory which are mapped to files on the disk have a flag set in the VAD to indicate this. These VADs have the same fields as VADs which don't map to files, but their flags are slightly different. Instead of indicating they are private memory, the flags show that they are mapped to a file on the disk. There is then a pointer to a CONTROL_AREA structure, which in turns points to a FILE_OBJECT structure. You should remember FILE_OBJECTS from the section on kernel objects. They hold information about a file on the disk, which can be shared by multiple processes.

The permissions field for mapped DLLs and executables is usually Execute/WriteCopy. With this permission a process can execute the code and write to the memory. If the process writes to that memory, however, a copy of the memory is made. For example, if the DLL makes changes to itself, a copy of the frame in question is made. Such changes should be local to the process, not shared amongst processes.


VAD
Range: 0x7c900000-0x7c9affff
Protection: Execute/WriteCopy
ImageMap
Control Area
File Object
C:\Windows\system32\ntdll.dll



Here is the full list of values for the protection field of a VAD. The default version of Volatility displays the Protection field as one of these eight values. We've patched the SANS version of Volatility to display the strings you see in the chart.

0 – No Access. The memory cannot be accessed and any attempt to do so results in an exception. Such exceptions can be handled by a program, but if the programmer didn't specify how to handle them, they result in the program crashing.

1 – The memory can only be read. Attempts to write or execute the memory will generate an exception.

2 – The memory can only be executed.

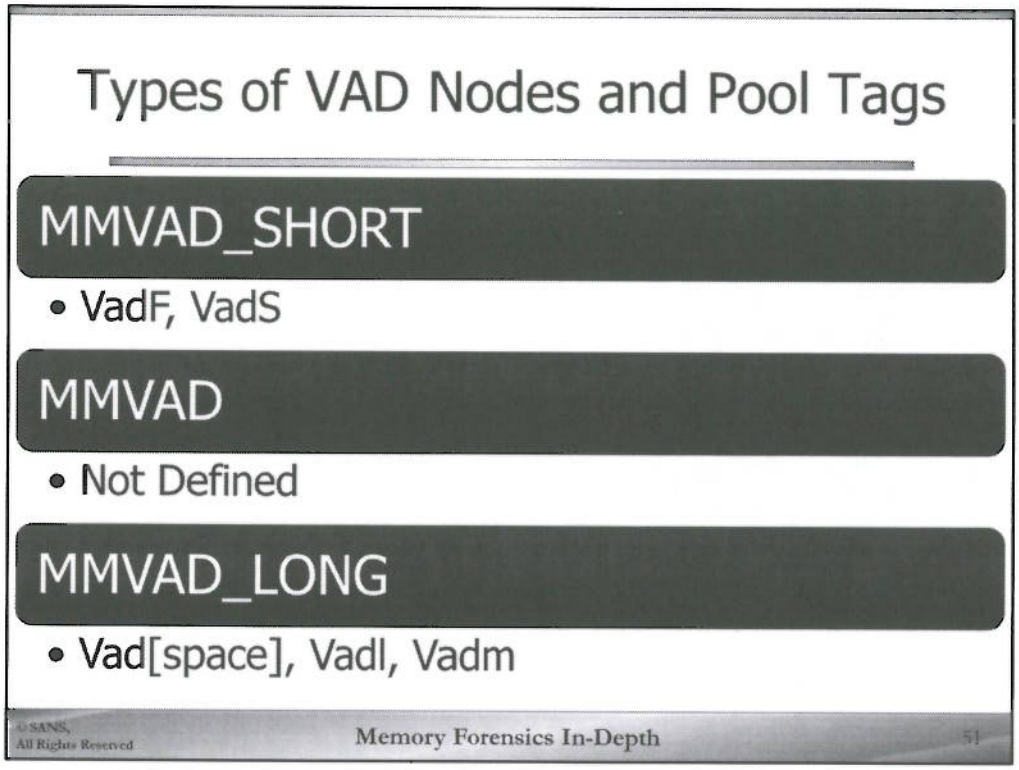
3 – The memory can be read or executed.

4 – The memory can be read or written to, but not executed.

5 – The memory can be written to, but if so a copy of it will be made.

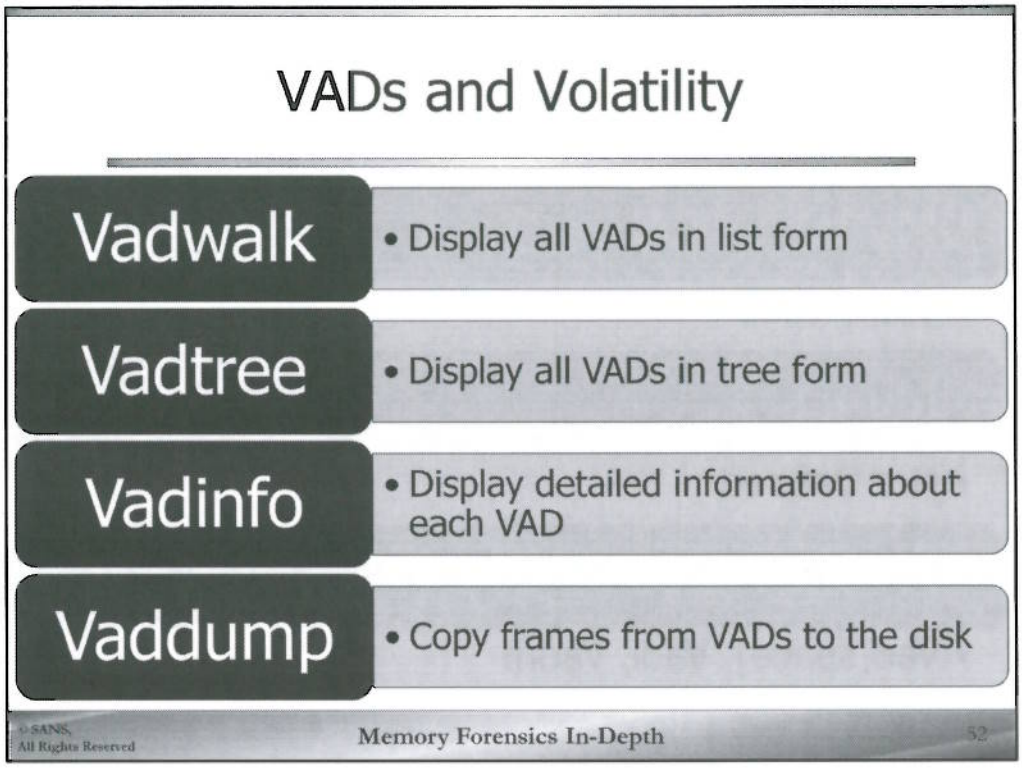
6 – The memory can be read, written, or executed.

7 – The memory can be read, written, or executed, but if written to, a copy will be made.



There are three kinds of VAD node structures. They are all stored in pool memory, and we know the type of VAD by the pool tag used on the pool header. (And the 'type' field in the VAD itself. When in doubt, we can treat a VAD structure as the shortest structure, check the type, and if necessary, re-cast it as one of the longer types.)

The smallest VAD structure is called MMVAD_SHORT. It only has fields for the memory range, parent, left and right children, and some flags. The second VAD structure is the MMVAD. It has the same fields as the MMVAD_SHORT, but also a pointer to a Control Area, which can point to a File Object. The third VAD structure is the MMVAD_LONG, which has everything in the MMVAD plus some more flags and other pointers. The short MMVAD structure uses the pool tags VadF or VadS. The long uses "Vad " (with a space at the end), Vadl, or Vadm. There do not appear to be any pool tags reserved for the MMVAD structure itself.



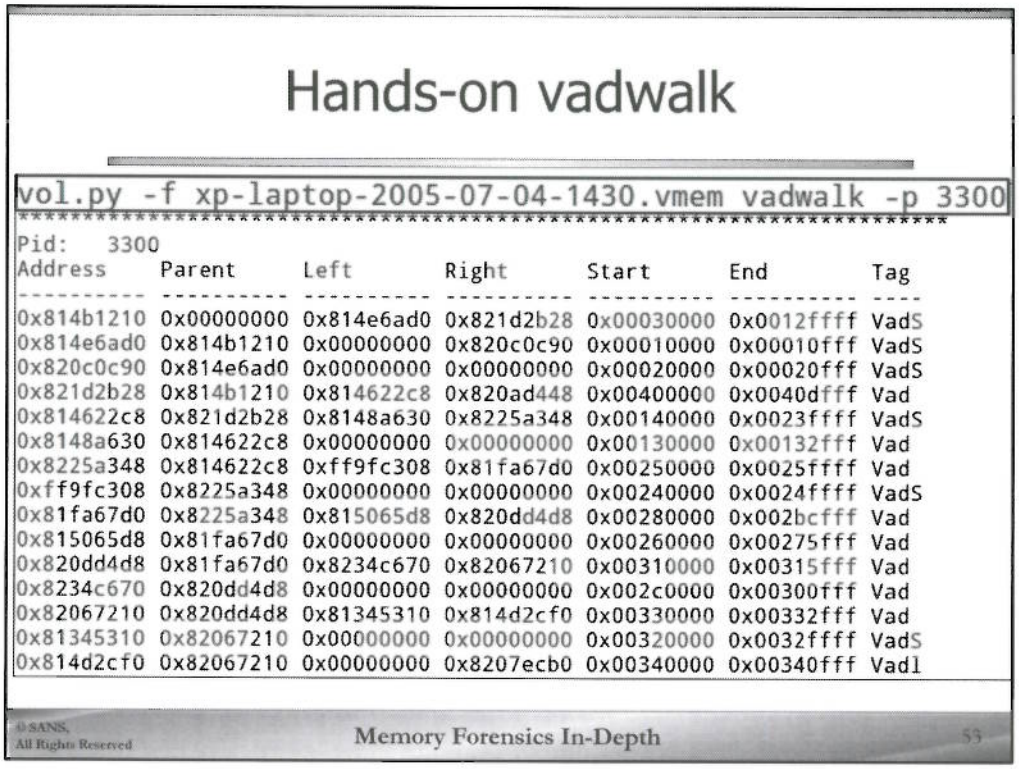
There are four Volatility plugins for looking at VADs:

vadwalk – Walk the VAD tree for each process or a subset of processes. For each VAD it displays the parent, memory range, left and right children, and the pool tag used on the entry.

vadtree – Walks the VAD tree like VAD walk, but attempts to display the result in a tree form. Unfortunately it's a one sided tree, and not terribly useful. That is, the output only appears to show the right-hand children from each VAD node.

vadinfo – Walks the VAD tree and displays detailed information about each VAD.

vaddump – Copies the frames committed by a VAD out to the disk. That is, dump the memory region associated with a VAD.



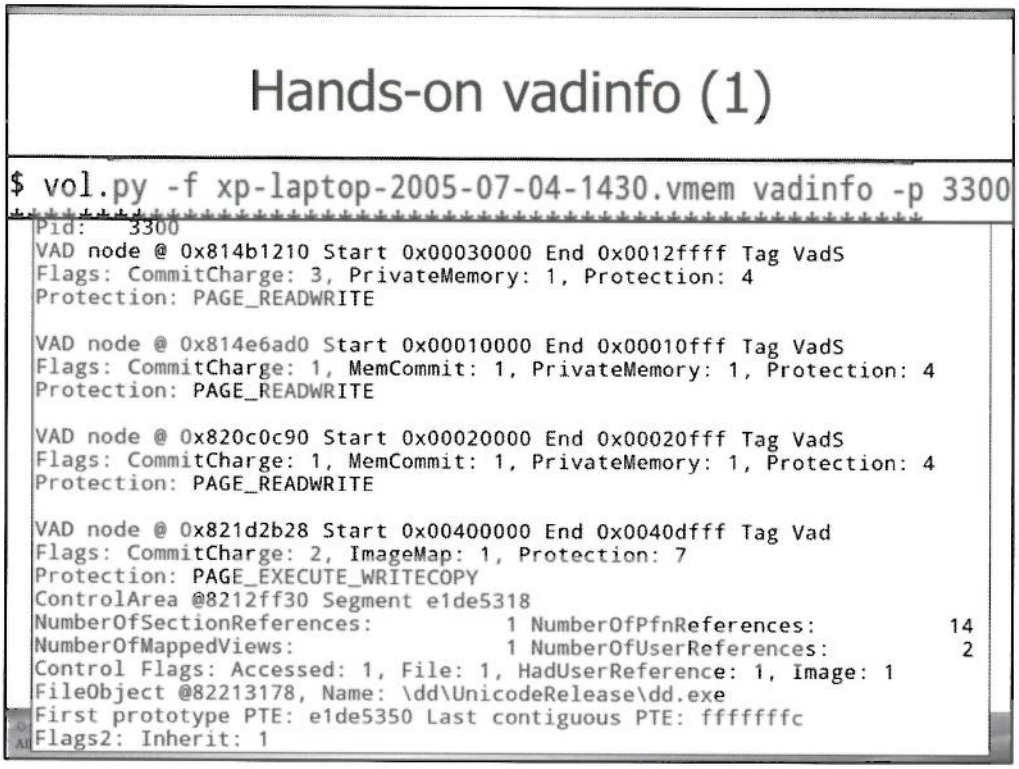
The first hands on exercise we're going to do with is with vadwalk using the xp laptop memory image. For simplicity we're only going to examine the VADs of the dd.exe process, pid 3300. Here's the command line to use:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
vadwalk --pid=3300
```

You should get the following output. We can now see all of the VADs for the dd.exe process. Although it gives us an overview of the process address space, it's not terribly interesting at this view. Let's move on to vadinfo.

```
**************************************************************************
Pid:   3300
Address  Parent   Left     Right    Start    End      Tag  Flags
814b1210 00000000 814e6ad0 821d2b28 00030000 0012ffff VadS
814e6ad0 814b1210 00000000 820c0c90 00010000 00010fff VadS
820c0c90 814e6ad0 00000000 00000000 00020000 00020fff VadS
821d2b28 814b1210 814622c8 820ad448 00400000 0040dfff Vad
814622c8 821d2b28 8148a630 8225a348 00140000 0023ffff VadS
8148a630 814622c8 00000000 00000000 00130000 00132fff Vad
8225a348 814622c8 ff9fc308 81fa67d0 00250000 0025ffff Vad
ff9fc308 8225a348 00000000 00000000 00240000 0024ffff VadS
81fa67d0 8225a348 815065d8 820dd4d8 00280000 002bcfff Vad
815065d8 81fa67d0 00000000 00000000 00260000 00275fff Vad
```

```
820dd4d8 81fa67d0 8234c670 82067210 00310000 00315fff Vad
8234c670 820dd4d8 00000000 00000000 002c0000 00300fff Vad
82067210 820dd4d8 81345310 814d2cf0 00330000 00332fff Vad
81345310 82067210 00000000 00000000 00320000 0032ffff VadS
814d2cf0 82067210 00000000 8207ecb0 00340000 00340fff Vadl
8207ecb0 814d2cf0 00000000 ff9b2250 00350000 00350fff Vadl
ff9b2250 8207ecb0 00000000 00000000 00360000 0036ffff VadS
820ad448 821d2b28 8204d1a0 81503448 7c900000 7c9affff Vad
8204d1a0 820ad448 81506608 00000000 7c800000 7c8f3fff Vad
81506608 8204d1a0 81fd0cf0 82107a98 10000000 10005fff Vad
81fd0cf0 81506608 00000000 815076a8 00410000 004d7fff Vad
815076a8 81fd0cf0 00000000 820c1518 004e0000 005e2fff Vad
820c1518 815076a8 00000000 00000000 005f0000 008effff Vad
82107a98 81506608 821240f0 00000000 7c000000 7c053fff Vad
821240f0 82107a98 820dd4a8 82177300 77c00000 77c07fff Vad
820dd4a8 821240f0 00000000 00000000 774e0000 7761cfff Vad
82177300 821240f0 82258580 00000000 77fe0000 77ff0fff Vad
82258580 82177300 815064d8 8203bfa8 77dd0000 77e6afff Vad
815064d8 82258580 8202f3c8 00000000 77d40000 77dcffff Vadl
8202f3c8 815064d8 00000000 00000000 77c10000 77c67fff Vad
8203bfa8 82258580 00000000 821f6bd8 77e70000 77f00fff Vad
821f6bd8 8203bfa8 00000000 00000000 77f10000 77f55fff Vad
81503448 820ad448 82258550 814ce3f0 7ffb0000 7ffd3fff Vad
82258550 81503448 00000000 00000000 7f6f0000 7f7effff Vad
814ce3f0 81503448 814d2cb0 00000000 7ffdf000 7ffdffff Vadl
814d2cb0 814ce3f0 00000000 00000000 7ffde000 7ffdefff Vadl
```



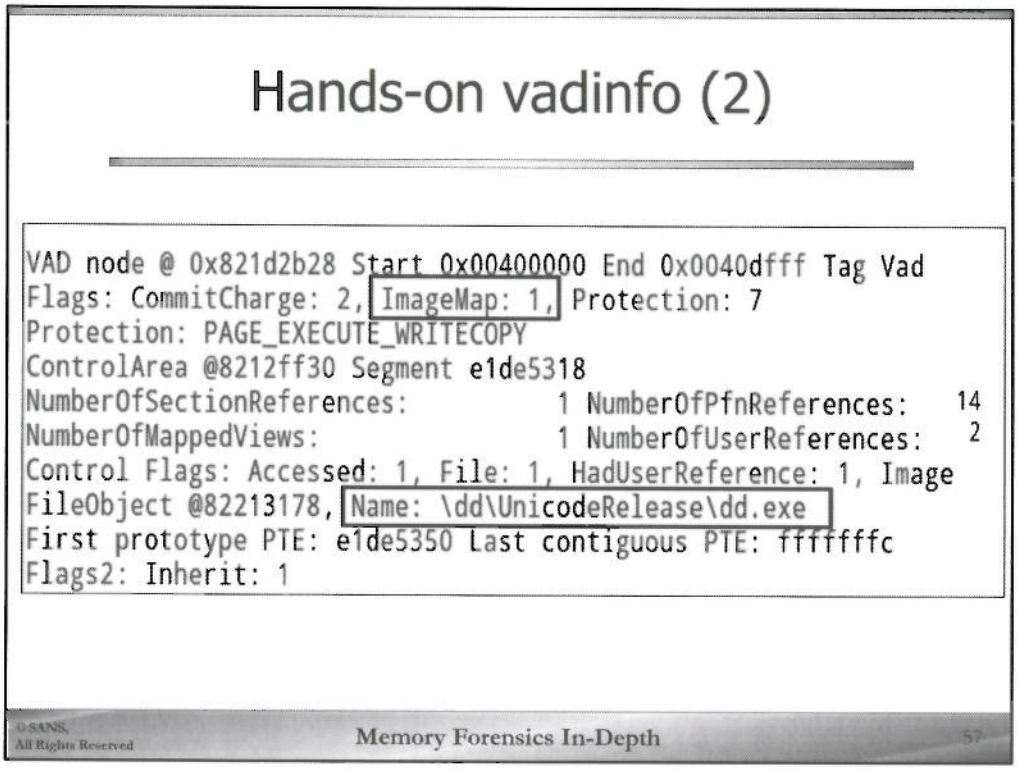
The vadinfo plugin will generate a LOT of output for each VAD, including parsing many of the flags and following pointers to some other structures. To run it, we repeat the same command line as before, but replacing the vadwalk command with vadinfo:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
vadinfo --pid=3300
```

This gives us output which starts with:

```
**************************************************************************
Pid:    3300
VAD node @814b1210 Start 00030000 End 0012ffff Tag VadS
Flags: PrivateMemory
Commit Charge: 3    Protection: Read/Write

VAD node @814e6ad0 Start 00010000 End 00010fff Tag VadS
Flags: MemCommit, PrivateMemory
Commit Charge: 1    Protection: Read/Write
```

```
VAD node @820c0c90 Start 00020000 End 00020fff Tag VadS
Flags: MemCommit, PrivateMemory
Commit Charge: 1   Protection: Read/Write

VAD node @821d2b28 Start 00400000 End 0040dfff Tag Vad
Flags: ImageMap
Commit Charge: 2   Protection: Execute/WriteCopy
ControlArea @8212ff30 Segment e1de5318
Dereference list: Flink 00000000, Blink 00000000
NumberOfSectionReferences:          1 NumberOfPfnReferences:          14
NumberOfMappedViews:                1 NumberOfUserReferences:          2
WaitingForDeletion Event:  00000000
Flags: Accessed, File, HadUserReference, Image
FileObject @8212ff54 FileBuffer @ e1c88690          , Name:
\dd\UnicodeRelease\dd.exe
First prototype PTE: e1de5350 Last contiguous PTE: fffffffc
Flags2: Inherit
File offset: 00000000
```

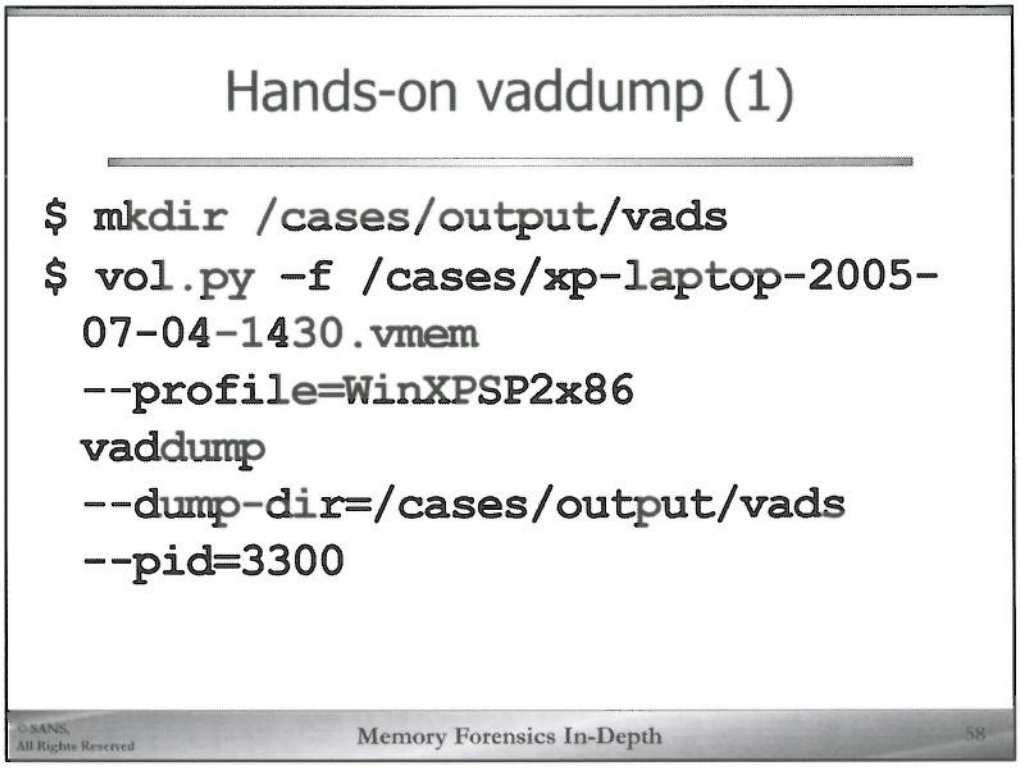


Let's look at the output of the vadinfo plugin.

The first three VADs all describe private memory regions. These are blocks of memory which the program has reserved and committed for its own use.

The fourth VAD is much more interesting. We can see the flags indicating this is an image map. It is mapped to a file. Next, its permissions were set to Execute/WriteCopy. This means that the memory can be executed, and any writes to this memory will result in a copy of the frame being made. Finally, we can see the filename of the mapped file. In this case it was the executable image itself, \dd\UnicodeRelease\dd.exe.

Looking through the rest of the vadinfo output, we can see other private memory ranges and other mapped files. The dd process is legitimate, and so all of the ranges you see are expected and normal. We can see that the process has reserved several blocks of memory. Some of these may be the heaps which Windows creates by default. There was also probably a buffer used to hold data captured from physical memory before writing it out to the disk.



We're now going to use the vaddump plugin to dump out the VADs for the dd.exe process. As with some of the other plugins which dump output to files, we must specify a directory where to dump these files. This is done with the --dump-dir command line flag. The full command line will be:

```
$ mkdir /cases/output/vads
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
vaddump --dump-dir=/home/sansforensics/output/vads --pid=3300
```

The output for this command should be minimal:

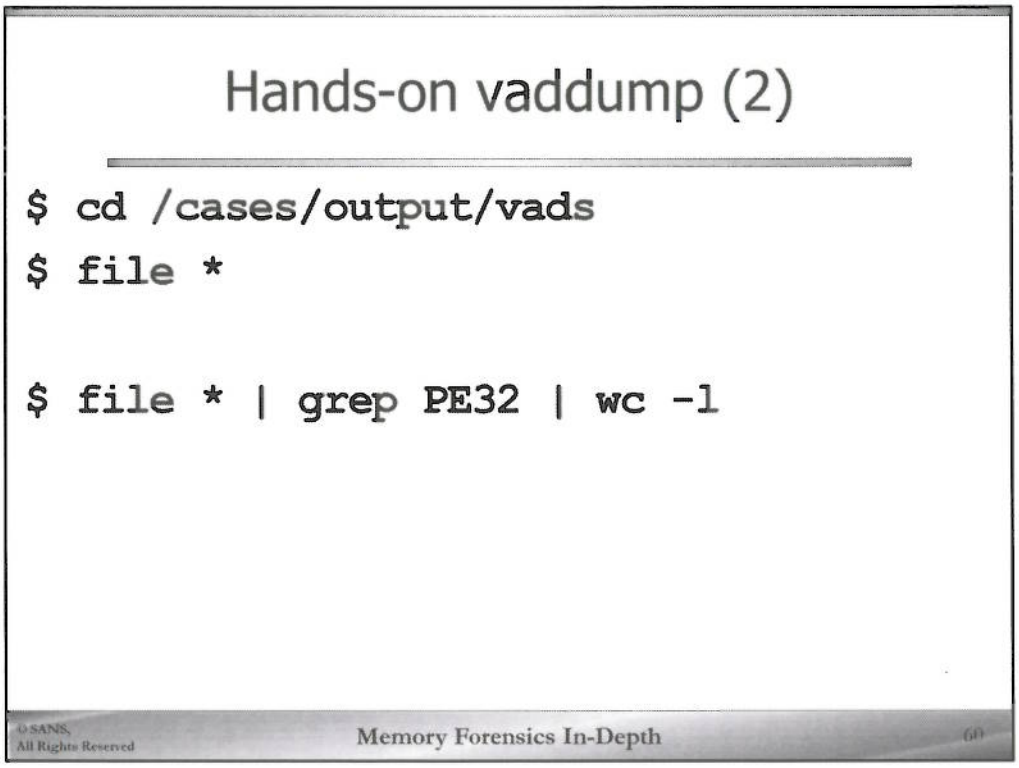
```
Pid:    3300
************************************************************************
```

But looking in the output directory we see lots of files, one for each VAD!

```
$ ls /cases/output/vads
```

```
dd.exe.1543870.00010000-00010fff.dmp  dd.exe.1543870.00330000-00332fff.dmp
dd.exe.1543870.77d40000-77dcffff.dmp
dd.exe.1543870.00020000-00020fff.dmp  dd.exe.1543870.00340000-00340fff.dmp
dd.exe.1543870.77dd0000-77e6afff.dmp
dd.exe.1543870.00030000-0012ffff.dmp  dd.exe.1543870.00350000-00350fff.dmp
dd.exe.1543870.77e70000-77f00fff.dmp
```

```
dd.exe.1543870.00130000-00132fff.dmp  dd.exe.1543870.00360000-0036ffff.dmp
dd.exe.1543870.77f10000-77f55fff.dmp
dd.exe.1543870.00140000-0023ffff.dmp  dd.exe.1543870.00400000-0040dfff.dmp
dd.exe.1543870.77fe0000-77ff0fff.dmp
dd.exe.1543870.00240000-0024ffff.dmp  dd.exe.1543870.00410000-004d7fff.dmp
dd.exe.1543870.7c000000-7c053fff.dmp
dd.exe.1543870.00250000-0025ffff.dmp  dd.exe.1543870.004e0000-005e2fff.dmp
dd.exe.1543870.7c800000-7c8f3fff.dmp
dd.exe.1543870.00260000-00275fff.dmp  dd.exe.1543870.005f0000-008effff.dmp
dd.exe.1543870.7c900000-7c9affff.dmp
dd.exe.1543870.00280000-002bcfff.dmp  dd.exe.1543870.10000000-10005fff.dmp
dd.exe.1543870.7f6f0000-7f7effff.dmp
dd.exe.1543870.002c0000-00300fff.dmp  dd.exe.1543870.774e0000-7761cfff.dmp
dd.exe.1543870.7ffb0000-7ffd3fff.dmp
dd.exe.1543870.00310000-00315fff.dmp  dd.exe.1543870.77c00000-77c07fff.dmp
dd.exe.1543870.7ffde000-7ffdefff.dmp
dd.exe.1543870.00320000-0032ffff.dmp  dd.exe.1543870.77c10000-77c67fff.dmp
dd.exe.1543870.7ffdf000-7ffdffff.dmp
```



We can go through these dumped out regions of memory using any tool we'd like. We could run strings on them, load them up in IDA Pro, whatever you desire. For now, let's do some simple tests to see if the frames we have dumped match the descriptions in the VAD information. Specifically, we're going to use the 'file' command to see which of these dumped files contains MZ/PE headers—which of them are Windows executables. We can change to the output directory:

```
$ cd /cases/output/vads
```

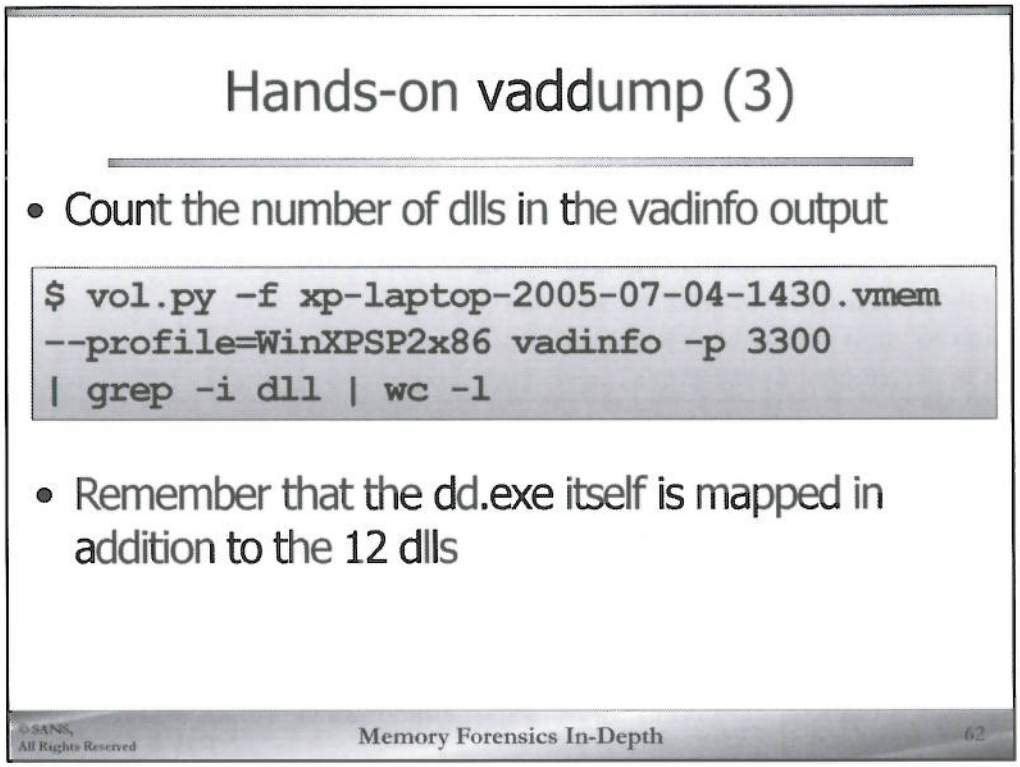
And then run file on everything in the directory:

```
$ file *

dd.exe.1543870.00010000-00010fff.dmp: data
dd.exe.1543870.00020000-00020fff.dmp: data
dd.exe.1543870.00030000-0012ffff.dmp: data
dd.exe.1543870.00130000-00132fff.dmp: data
dd.exe.1543870.00140000-0023ffff.dmp: data
dd.exe.1543870.00240000-0024ffff.dmp: data
dd.exe.1543870.00250000-0025ffff.dmp: data
dd.exe.1543870.00260000-00275fff.dmp: data
dd.exe.1543870.00280000-002bcfff.dmp: data
dd.exe.1543870.002c0000-00300fff.dmp: data
dd.exe.1543870.00310000-00315fff.dmp: DBase 3 index file
```

```
dd.exe.1543870.00320000-0032ffff.dmp: data
dd.exe.1543870.00330000-00332fff.dmp: data
dd.exe.1543870.00340000-00340fff.dmp: data
dd.exe.1543870.00350000-00350fff.dmp: data
dd.exe.1543870.00360000-0036ffff.dmp: data
dd.exe.1543870.00400000-0040dfff.dmp: PE32 executable (console) Intel 80386,
for MS Windows
dd.exe.1543870.00410000-004d7fff.dmp: data
dd.exe.1543870.004e0000-005e2fff.dmp: data
dd.exe.1543870.005f0000-008effff.dmp: data
dd.exe.1543870.10000000-10005fff.dmp: PE32 executable (DLL) (GUI) Intel 80386,
for MS Windows
dd.exe.1543870.774e0000-7761cfff.dmp: PE32 executable (DLL) (console) Intel
80386, for MS Windows
dd.exe.1543870.77c00000-77c07fff.dmp: PE32 executable (DLL) (GUI) Intel 80386,
for MS Windows
dd.exe.1543870.77c10000-77c67fff.dmp: PE32 executable (DLL) (GUI) Intel 80386,
for MS Windows
dd.exe.1543870.77d40000-77dcffff.dmp: PE32 executable (DLL) (GUI) Intel 80386,
for MS Windows
dd.exe.1543870.77dd0000-77e6afff.dmp: PE32 executable (DLL) (console) Intel
80386, for MS Windows
dd.exe.1543870.77e70000-77f00fff.dmp: PE32 executable (DLL) (console) Intel
80386, for MS Windows
dd.exe.1543870.77f10000-77f55fff.dmp: PE32 executable (DLL) (console) Intel
80386, for MS Windows
dd.exe.1543870.77fe0000-77ff0fff.dmp: PE32 executable (DLL) (console) Intel
80386, for MS Windows
dd.exe.1543870.7c000000-7c053fff.dmp: PE32 executable (DLL) (GUI) Intel 80386,
for MS Windows
dd.exe.1543870.7c800000-7c8f3fff.dmp: PE32 executable (DLL) (console) Intel
80386, for MS Windows
dd.exe.1543870.7c900000-7c9affff.dmp: PE32 executable (DLL) (console) Intel
80386, for MS Windows
dd.exe.1543870.7f6f0000-7f7effff.dmp: data
dd.exe.1543870.7ffb0000-7ffd3fff.dmp: data
dd.exe.1543870.7ffde000-7ffdefff.dmp: data
dd.exe.1543870.7ffdf000-7ffdffff.dmp: data
```

There are a total of 13 PE32 files. As a handy shortcut, you can pipe the output of file through grep and wc to get this count:

```
$ file * | grep PE32 | wc -l

13
```



We found a total of 13 memory regions in this process which contain PE32 executables. How many were there supposed to be? We can go back to the vadinfo output we had from before and count the number of mapped executables. Remember that Windows processes can map any file. Starting with Windows Vista, the operating system maps in a number of National Language Support (NLS) files. We have to be careful to only count the DLLs and executables.

To save time, we can pipe the output of vadinfo through grep and wc. First, we run the command with grep to make sure we have the right output:

```
$vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
vadinfo --pid=3300 | grep -i dll
FileObject @823cb1c4 FileBuffer @ e156b690      , Name:
\WINDOWS\system32\ntdll.dll
FileObject @820976b4 FileBuffer @ e1777958      , Name:
\WINDOWS\system32\kernel32.dll
FileObject @814d454c FileBuffer @ e1770310      , Name:
\dd\UnicodeRelease\getopt.dll
FileObject @814d4974 FileBuffer @ e25e3430      , Name:
\dd\UnicodeRelease\MSVCR70.dll
FileObject @8211cbbc FileBuffer @ e19124a0      , Name:
\WINDOWS\system32\version.dll
FileObject @821d202c FileBuffer @ e1929a48      , Name:
\WINDOWS\system32\ole32.dll
```

```
FileObject @820cd29c FileBuffer @ e1929a88          , Name:
\WINDOWS\system32\secur32.dll
FileObject @81f9a144 FileBuffer @ e1761778          , Name:
\WINDOWS\system32\advapi32.dll
FileObject @8212082c FileBuffer @ e17bfb08          , Name:
\WINDOWS\system32\user32.dll
FileObject @820d7904 FileBuffer @ e18800c0          , Name:
\WINDOWS\system32\msvcrt.dll
FileObject @821406c4 FileBuffer @ e1779b28          , Name:
\WINDOWS\system32\rpcrt4.dll
FileObject @8225dc8c FileBuffer @ e1969188          , Name:
\WINDOWS\system32\gdi32.dll
```

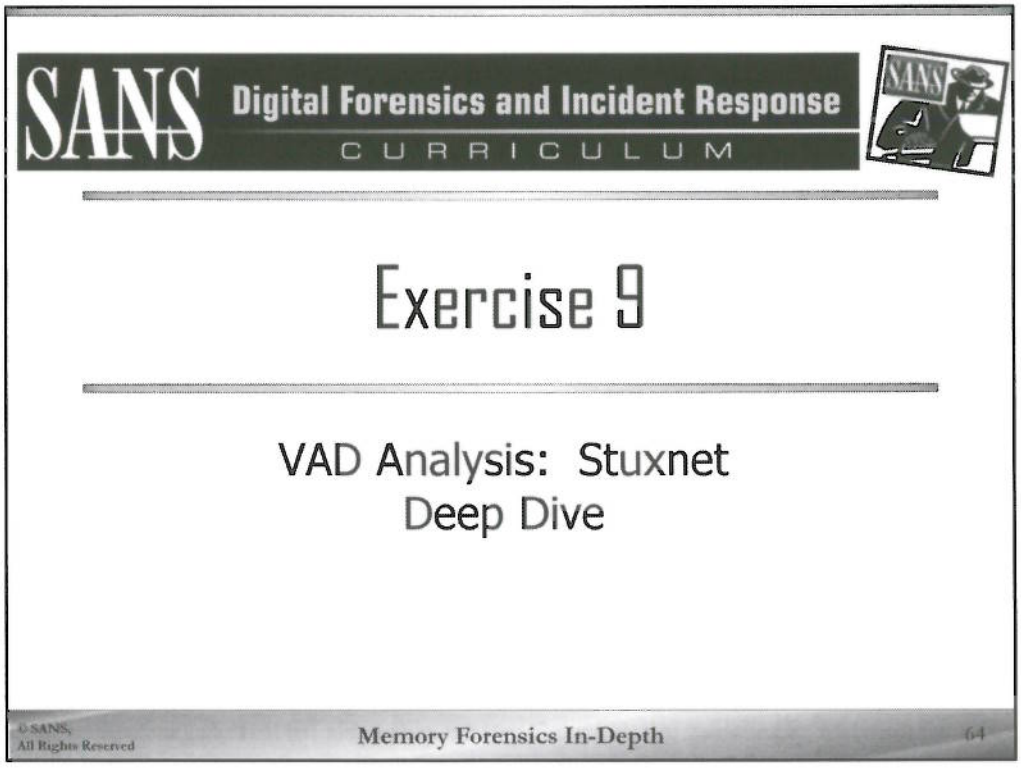
Yup, that looks right. Now pipe that through wc to see how many lines it is:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
vadinfo --pid=3300 | grep -i dll | wc -l
12
```

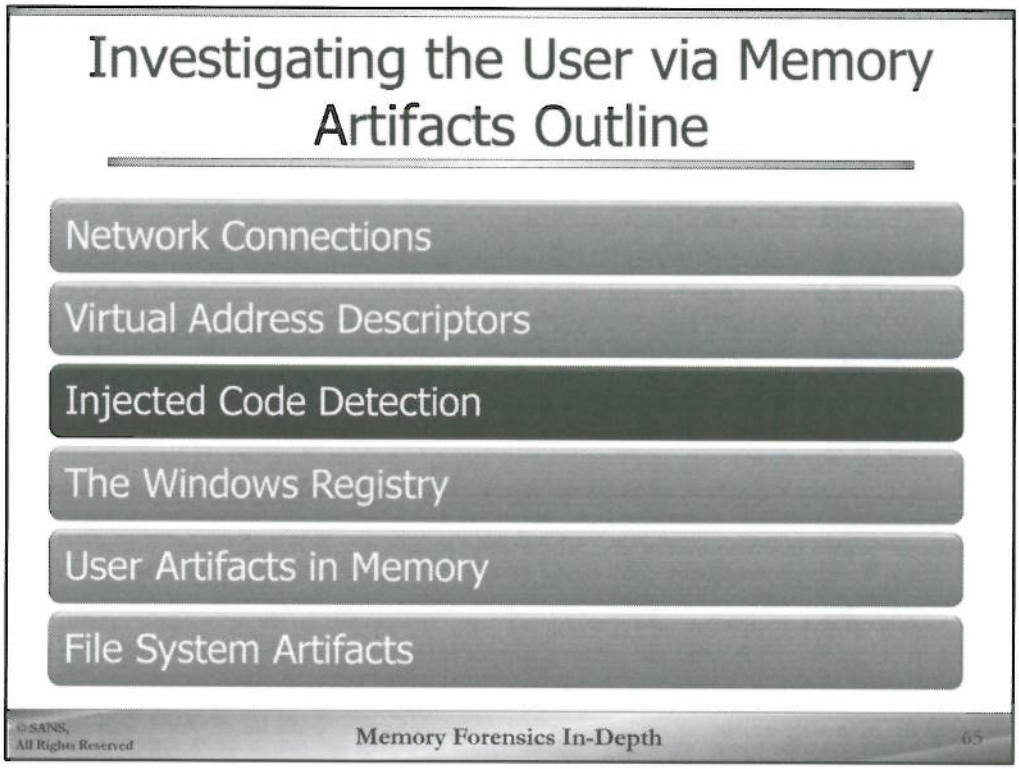
Only 12? But we found 13 executables. Ah, but we forgot the executable itself. Try again while grepping for exe. This time we have to be sure to include the period in the extension, in quotes. If we don't, we'll also get every line which includes "Execute".

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
vadinfo --pid=3300 | grep -i "\.exe"
FileObject @8212ff54 FileBuffer @ e1c88690          , Name:
\dd\UnicodeRelease\dd.exe
```

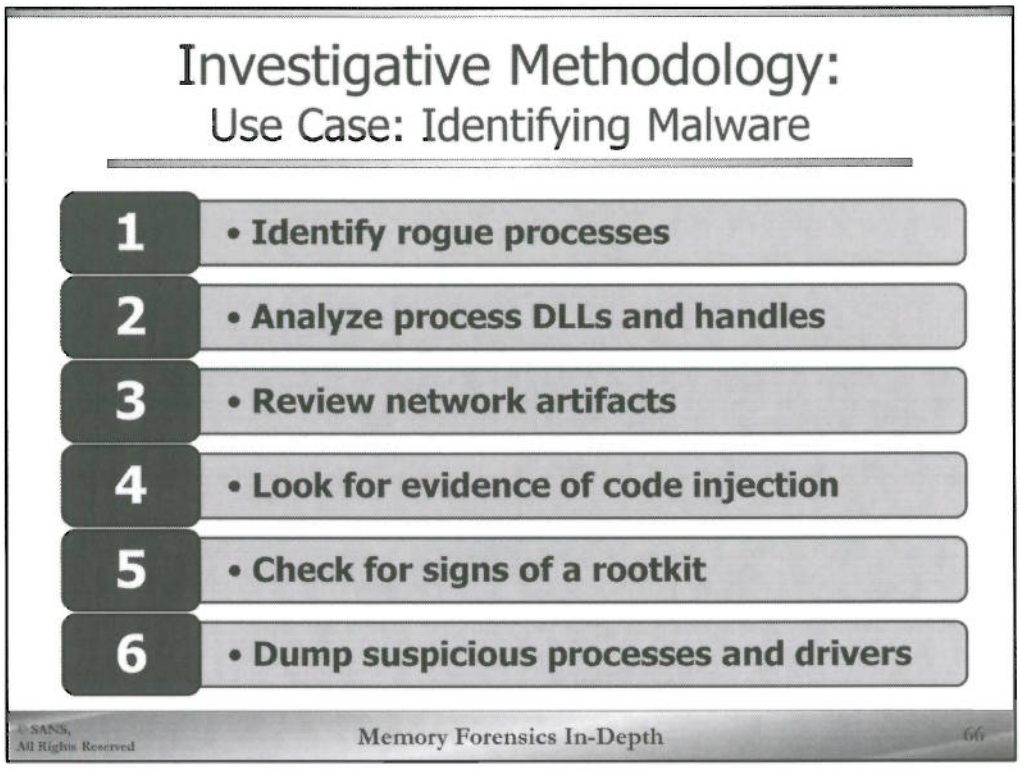
Just one executable. And so when we combine the 12 DLLs with one executable, that accounts for the 13 executables we found while looking at what we pulled out of the VADs.



This page intentionally left blank.

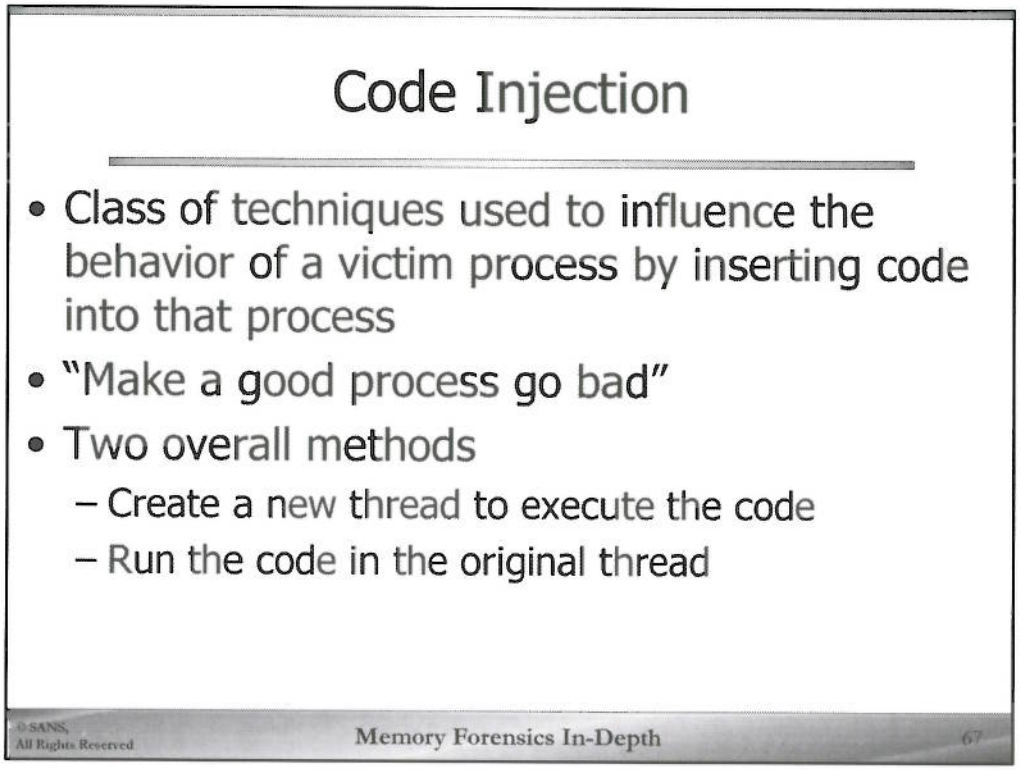


This page intentionally left blank.

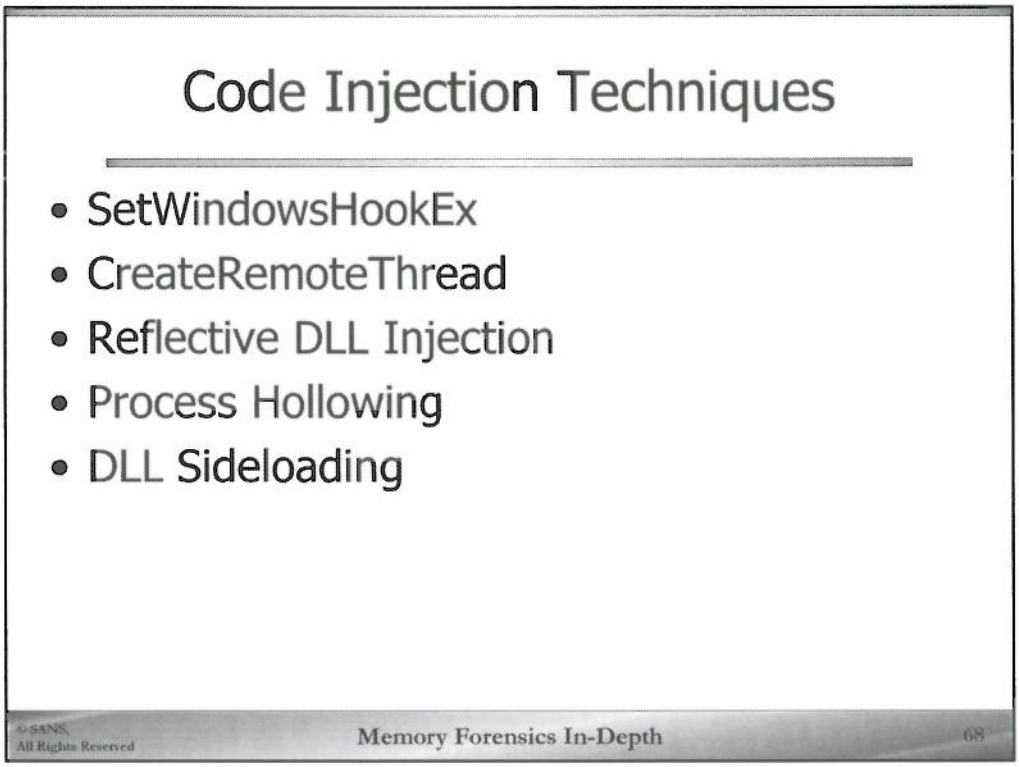


In a large majority of malware investigations, rogue processes may not be easy to pick out of a process listing or process scan. And unfortunately, not all systems that are compromised exhibit outbound network traffic. How else might an investigator spot malicious code on a system? Step 4 in our Investigative Methodology points to using "low-level" detailed analysis to spot signs of injected code, an obfuscation technique ubiquitous to today's malware attacks. By not running on the system as a standalone process, but as an injected dll from a legitimate process' memory section, malware authors may increase their chances of evading detection.

Walking through the Stuxnet exercise, we were able to identify sections of injected code through the deduction of what was expected to be in each VAD versus what the VAD node reported (i.e. no ImageMap flag in the VAD node but the memory section contains a PE file). Now, we will discuss the "art of dll injection" as well as some more advanced plugins, authored by Michael Hale Ligh, that will allow us to make easy work of what took us the good portion of an hour to work through by hand.

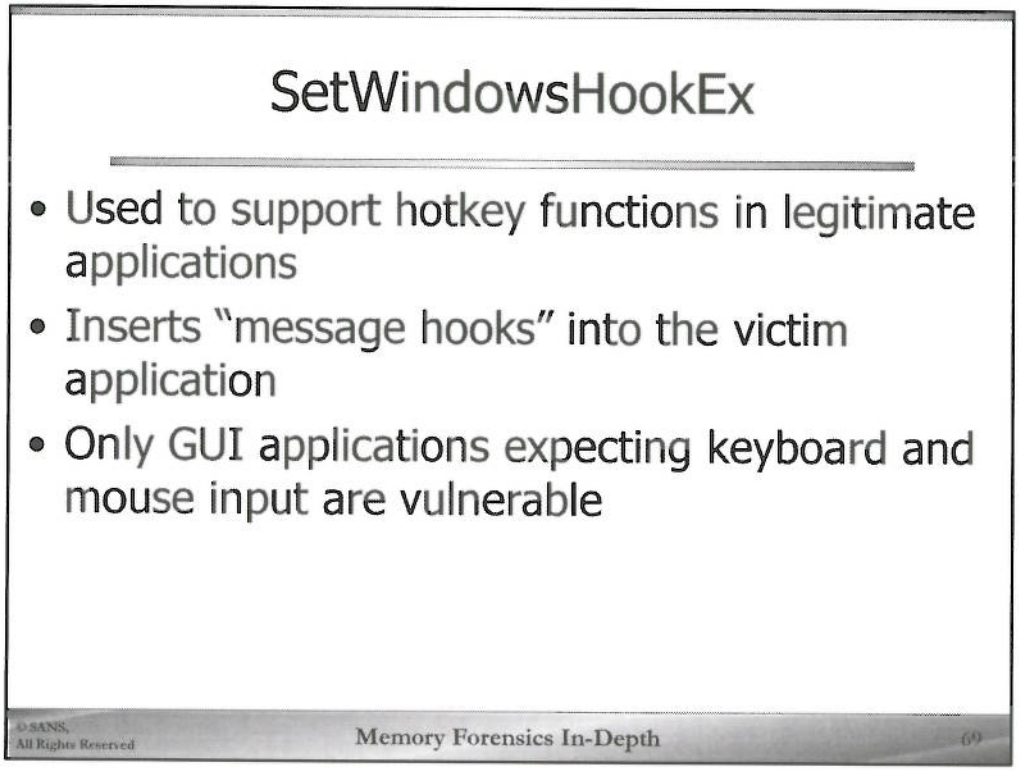


Code injection is a generic class of techniques used to influence the behavior of a victim process by inserting code into that process. This code is inserted into the victim process using a variety of techniques. Code injection techniques may create a new thread to execute the code or may run the code in the original thread (effectively neutering the victim program).

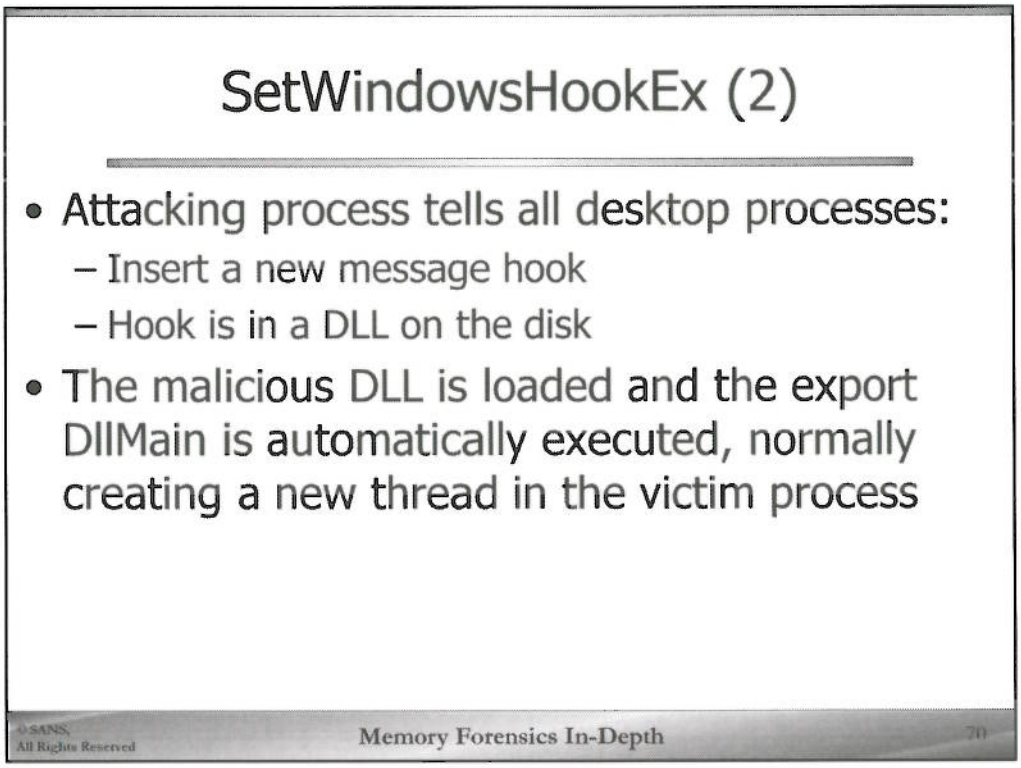


There are a number of techniques that can be used to injection code into a victim process. The following pages will include an overview of each of the techniques below, though this list is not comprehensive.

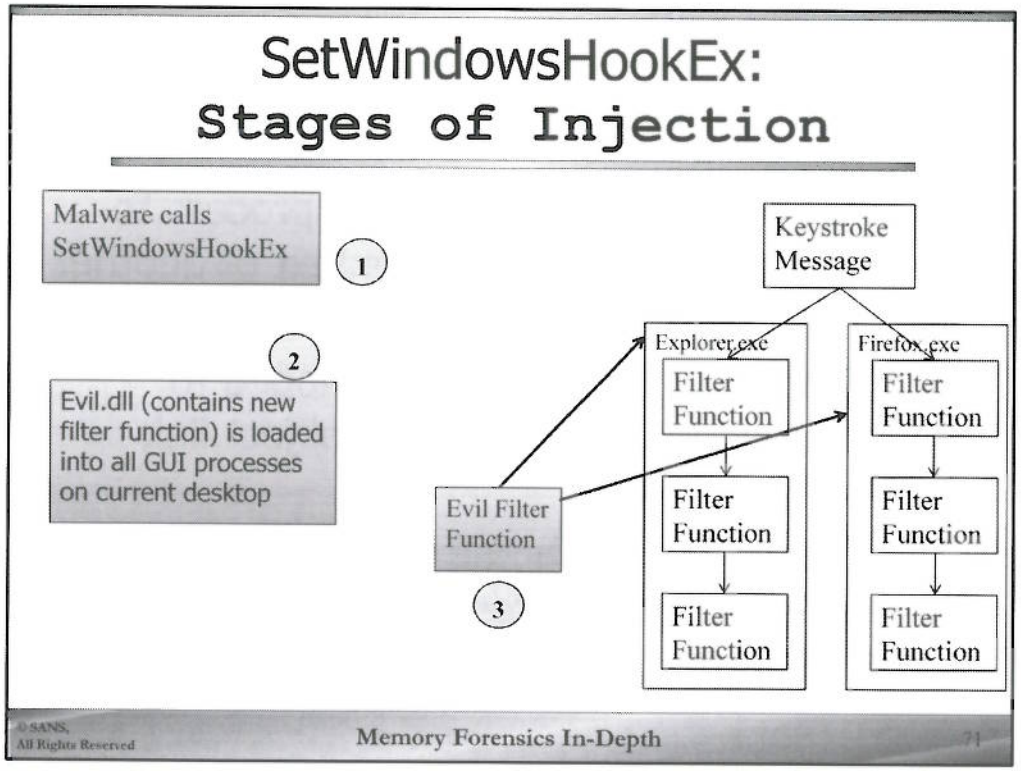
- CreateRemoteThread
- SetWindowsHookEx
- Reflective DLL Injection
- Process Hollowing
- DLL Sideloading



Despite the malicious sounding name, the SetWindowsHookEx API is used to support hotkey functions in legitimate applications. It supports hot keys by inserting "message hooks" into the victim application. Code can only be injected into GUI applications expecting keyboard and mouse input using this method. So processes like lsass.exe, services.exe, etc. that do not expect keyboard and mouse input cannot be injected using this method. Additionally, only processes attached to the same desktop as the injecting processes can be victimized. This is usually not an issue, but may be a consideration on application servers where many users create desktops (e.g. an RDP or Citrix based application server).

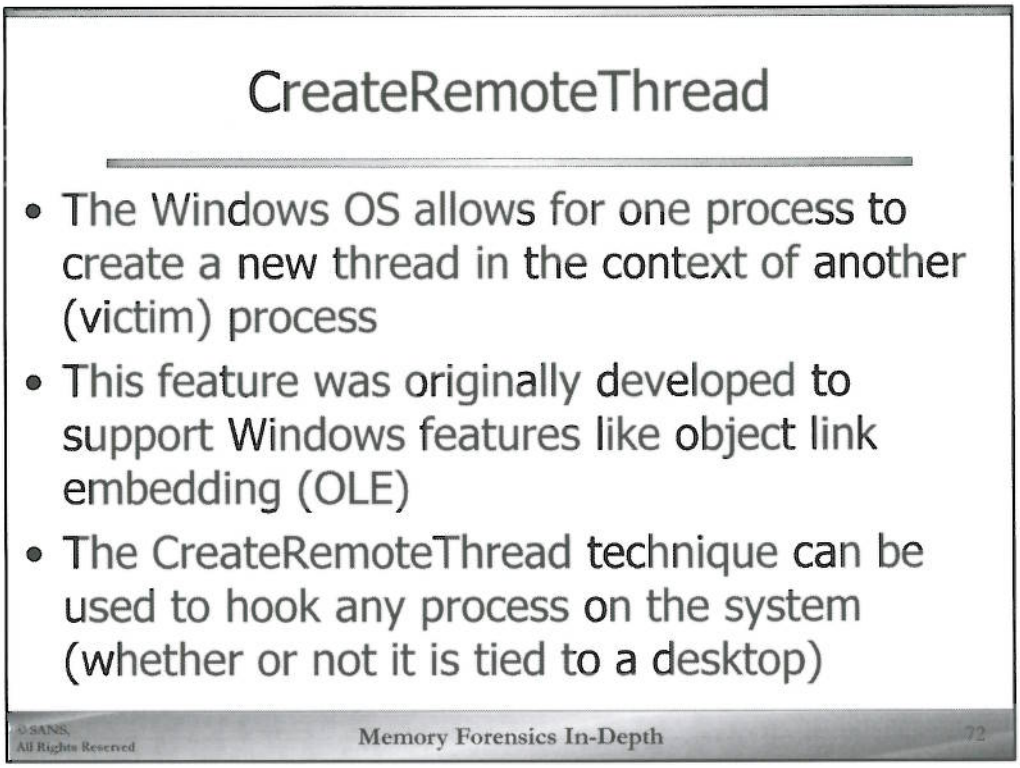


The attacking process tells all processes on the desktop that they should insert a new message hook. The hooking process tells the victim processes that the hook is in a DLL on the disk. The malicious DLL is loaded and the export DllMain is automatically executed, normally creating a new thread in the victim process.

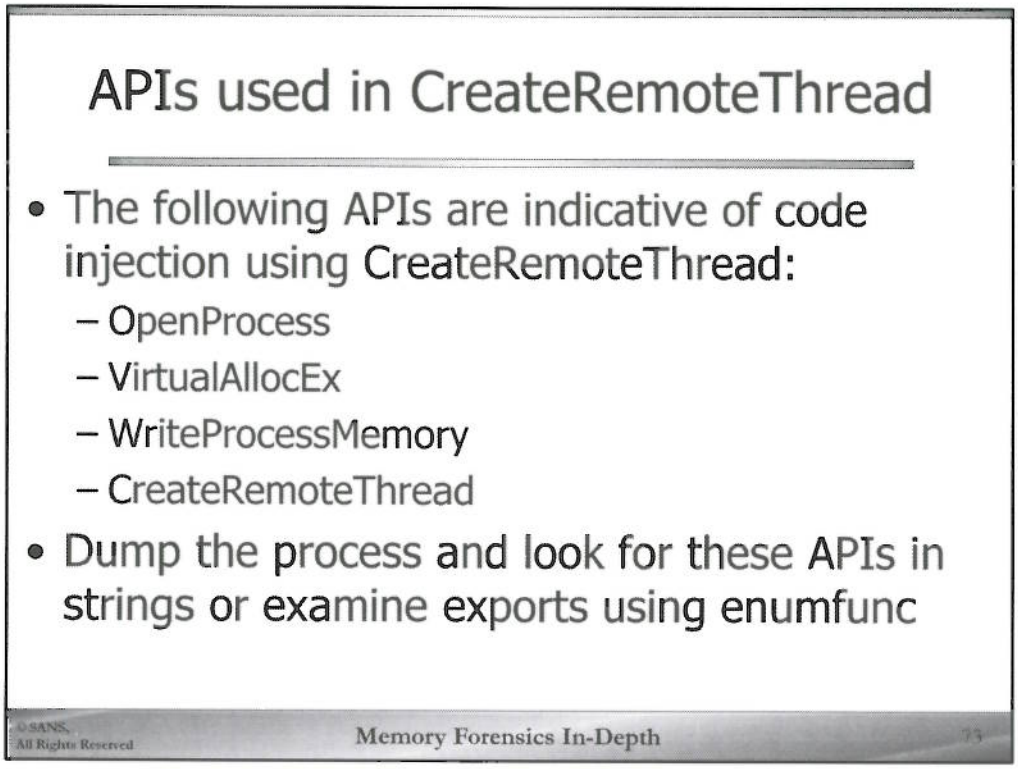


This diagram illustrates how SetWindowsHookEx functions to perform code injection. A malware process attached to a desktop calls SetWindowsHookEx and tells one or more processes attached to the desktop to load a new filter function. The malware must tell the process in which DLL to find the new filter function. The DLL containing the filter function will be malicious in nature. All DLLs execute a special export, DllMain, when loaded. Though the filter function will be installed in the desktop processes, the more interesting point from the attacker's perspective is that arbitrary code can be executed from DllMain in each process attached the desktop.

Only those processes that expect keyboard and mouse input can be hooked using this technique. Processes such as lsass.exe and services.exe that do not interact with any desktop cannot be injected using this technique.



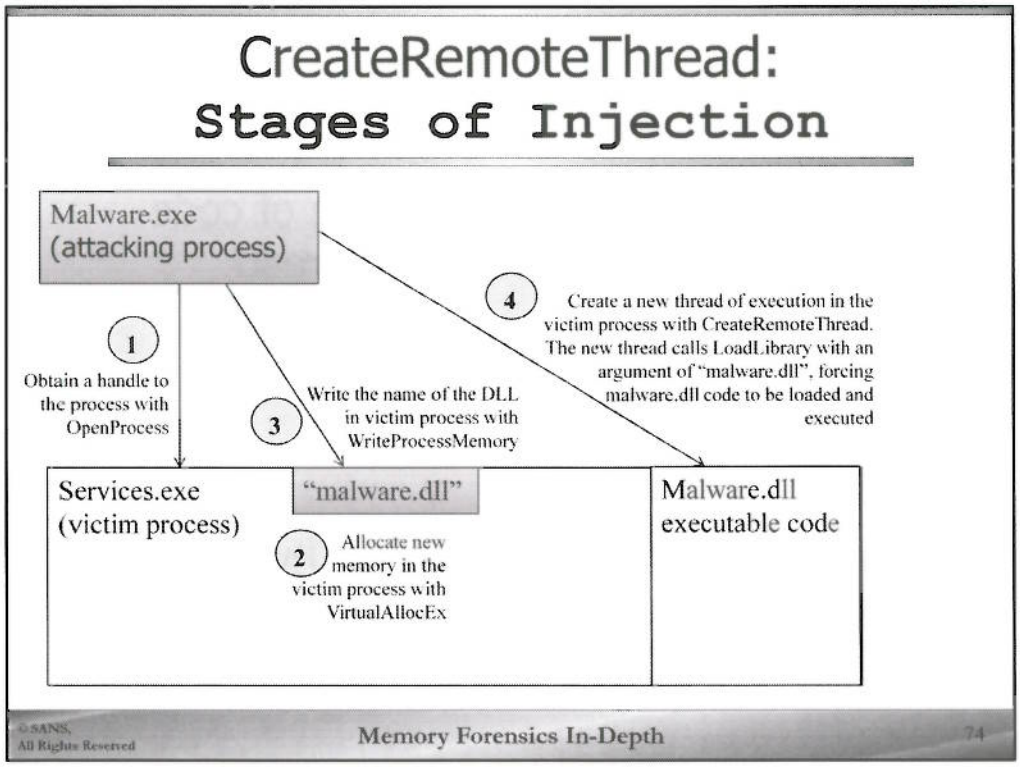
The Windows OS allows for one process to create a new thread in the context of another (victim) process. This feature was originally developed to support Windows features like object link embedding (OLE). This feature allows Unlike SetWindowsHookEx, the CreateRemoteThread technique can be used to hook any process on the system (whether or not it is tied to a desktop).



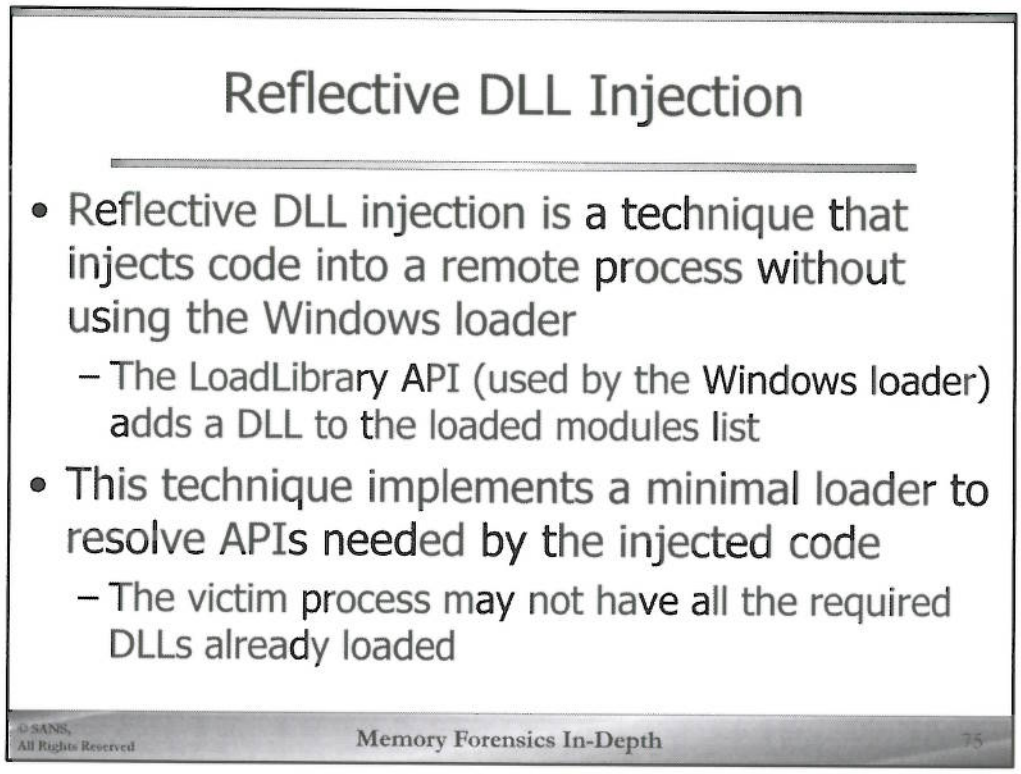
The following APIs are indicative of code injection using CreateRemoteThread:

- OpenProcess
- VirtualAllocEx
- WriteProcessMemory
- CreateRemoteThread

Detection is easy – just dump the injecting process and look for these APIs in strings or examine exports using enumfunc. This will only detect the injecting process – not the victim process. This can be problematic since often the attacking process will exit after injecting code into the victim process.

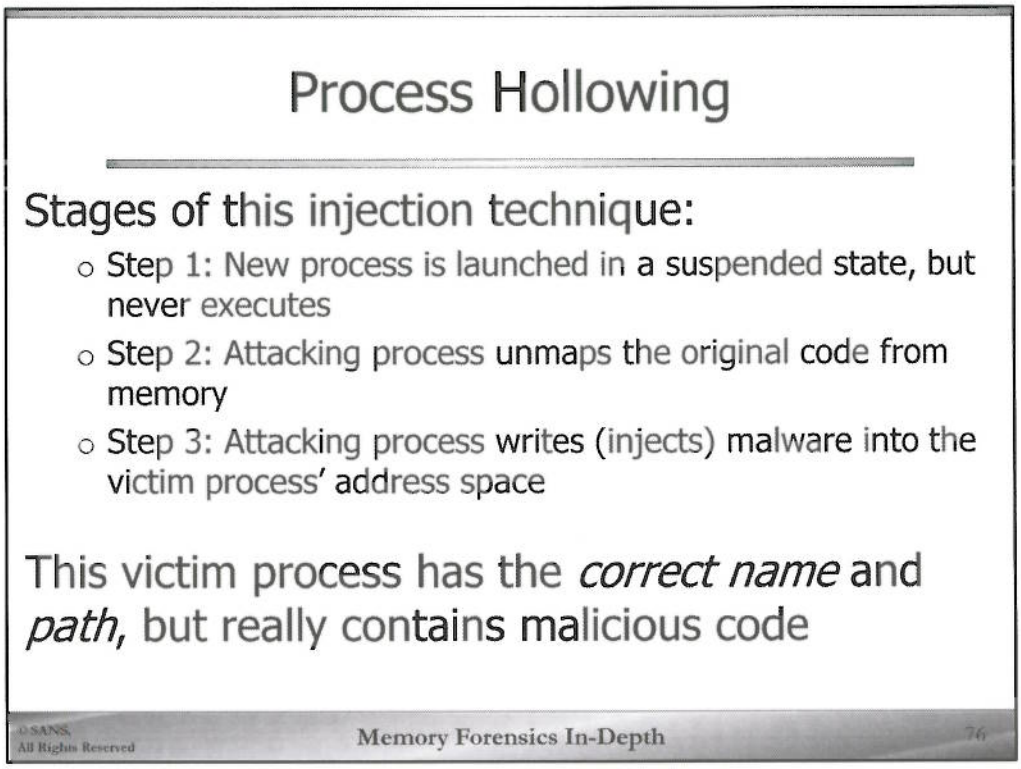


1. The malware first obtains a handle to the process with OpenProcess.
2. The malware allocates new memory in the victim process using VirtualAllocEx
3. The malware writes the name of the DLL in victim process with WriteProcessMemory. This is needed since LoadLibrary will require an argument in the same process address space.
4. The malware creates a new thread of execution in the victim process with CreateRemoteThread. The new thread calls LoadLibrary with an argument of "malware.dll", forcing malware.dll code to be loaded and executed. DllMain is automatically executed.
5. In most cases malware.exe will now exit since services.exe is running malicious code for the attacker.



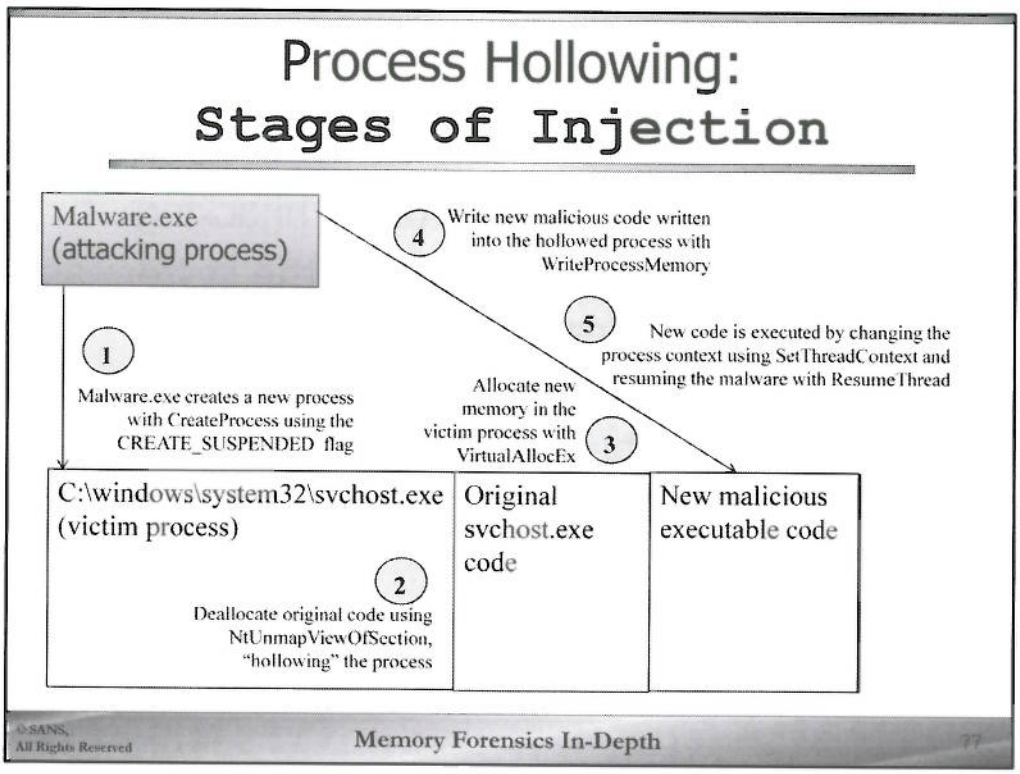
Reflective DLL injection is a technique that injects code into a remote process without using the Windows loader. The LoadLibrary API (used by the Windows loader) adds a DLL to the loaded modules list. This technique implements a minimal loader to resolve APIs needed by the injected code. The victim process may not have all the required DLLs already loaded.

This technique is used by Metasploit's Meterpreter module and because this code is open source, the functionality has been incorporated into many malware samples.

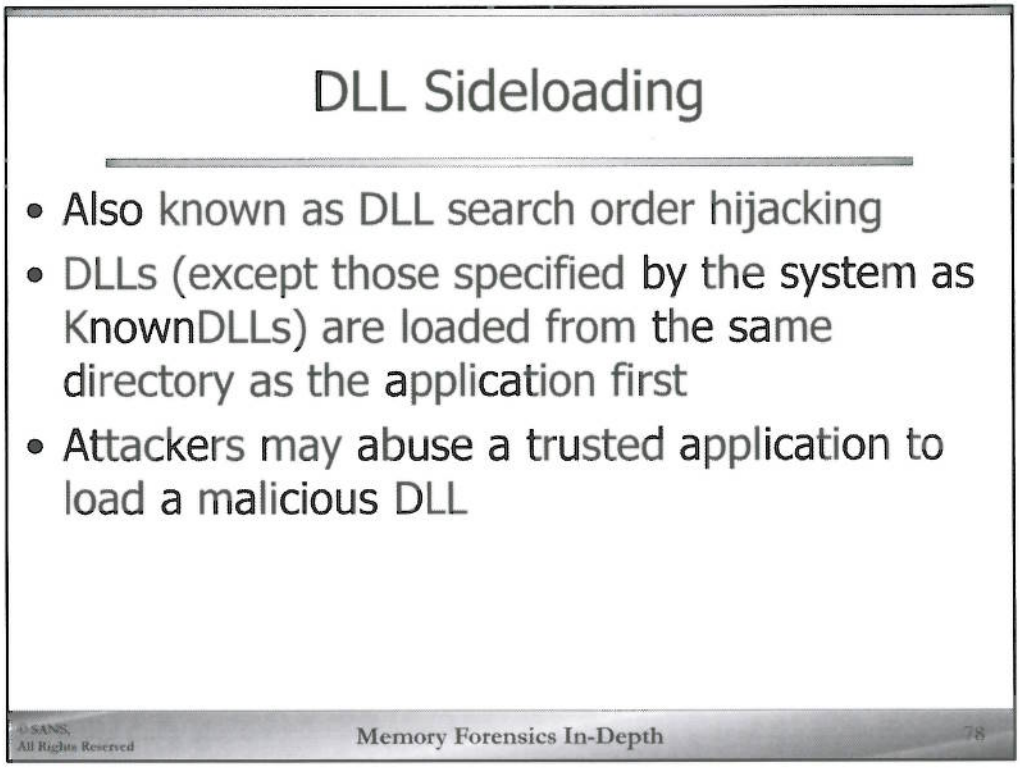


This technique injects code by first starting new process in a suspended state. Windows allows processes to be started in a suspended state so that the process object can be pre-allocated during idle times.

The new (suspended) process never actually executes – the attacking process unmaps the original code from memory and writes (injects) malware into the process' address space. This victim process has the correct name and path, but really contains malicious code. This is a huge advantage to the attacker in that the processes are better camouflaged from detection, even if the investigator is looking at both the process name and path.

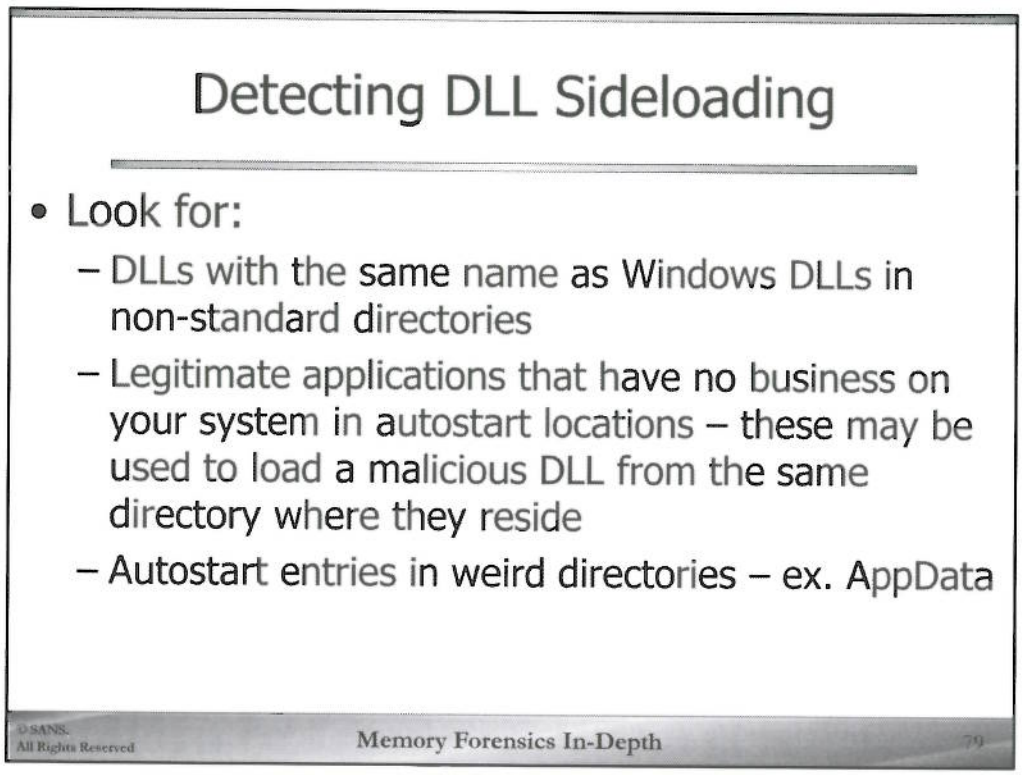


1. The malware first creates a new process using CreateProcess and uses the flag CREATE_SUSPENDED.
2. The malware deallocates the original victim process code using NtUnmapViewOfSection, effectively hollowing the process.
3. The malware allocates new memory in the victim process with VirtualAllocEx.
4. The malware writes new malicious code into the process in the area of the new memory allocation using the WriteProcessMemory API call.
5. The new code is executed by changing the process context using SetThreadContext. Given the new context, the thread is resumed by calling ResumeThread.
6. In most cases malware.exe will now exit since svchost.exe is running malicious code for the attacker.



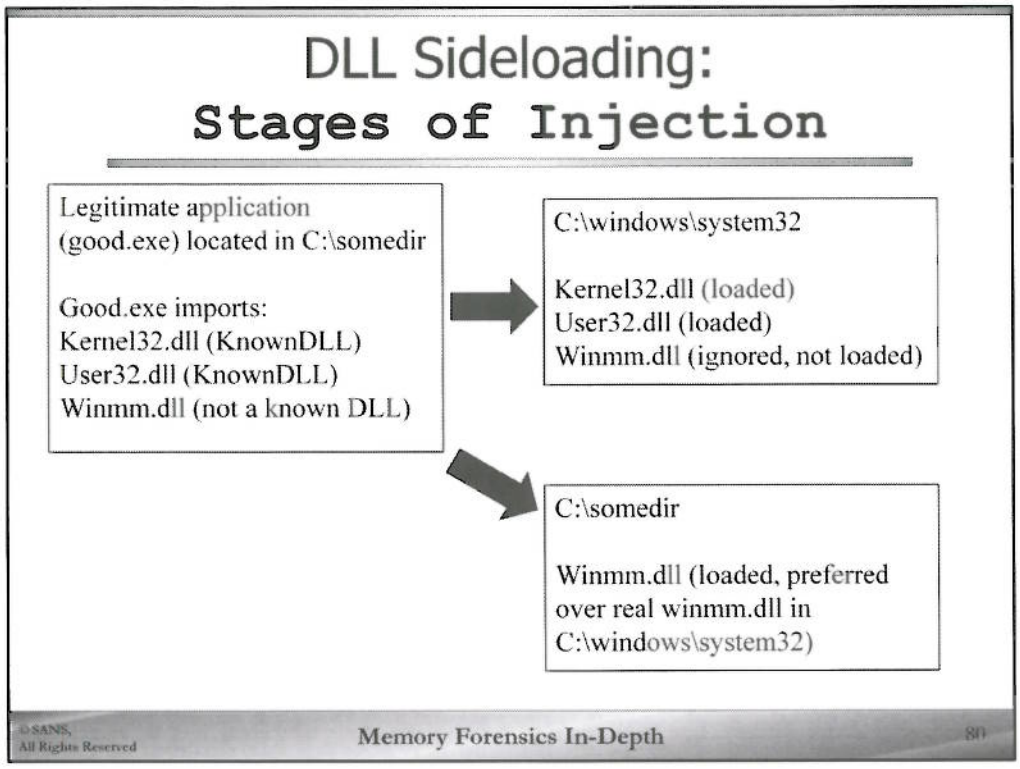
DLL sideloading is a special case of DLL search order hijacking. Although some DLLs are considered "known" (meaning they will always be loaded from C:\windows\system32), most are not and will be loaded following standard search order rules.

DLLs (except those specified by the system as KnownDLLs) are loaded from the same directory as the application first. Attackers may abuse a trusted application to load a malicious DLL. Most applications seen in the wild are digitally signed, which usually means that antivirus gives them a free pass. Antivirus however would be wise to examine the DLLs that the applications are loading to protect against DLL sideloading.

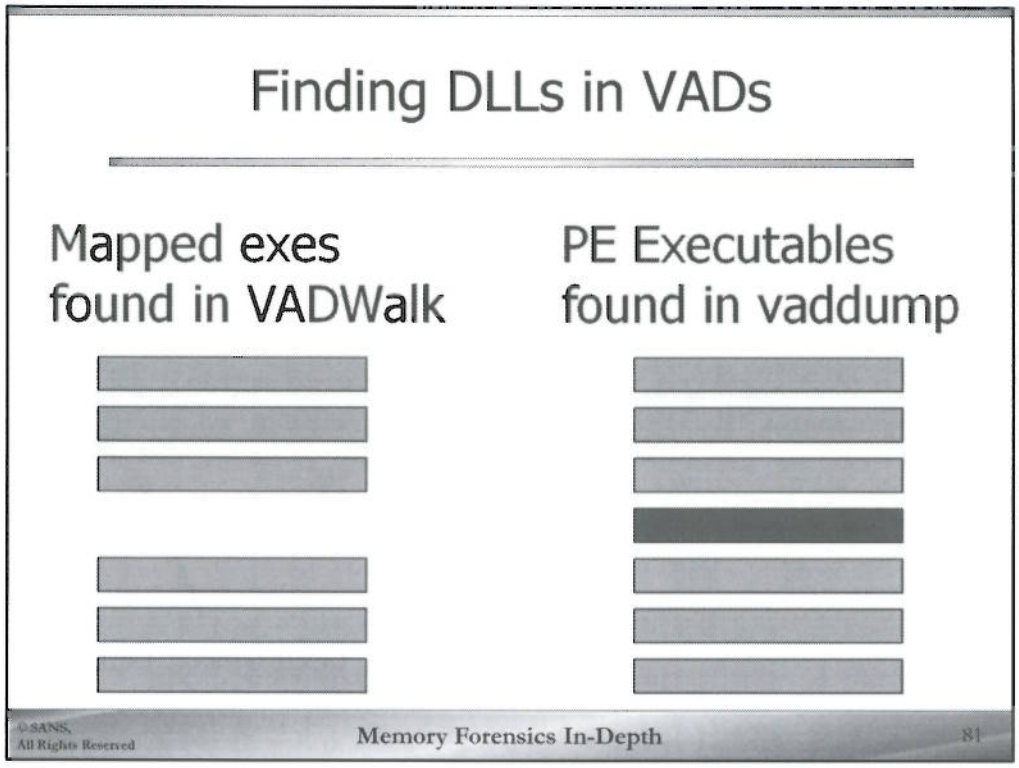


It should be noted that there is no sure fire way to detect DLL Sideloading. However some signs you can look for are:

- DLLs with the same name as Windows DLLs in non-standard directories
- Legitimate applications that have no business on your system in autostart locations – these may be used to load a malicious DLL from the same directory where they reside
- Autostart entries that launch applications located in weird directories – User AppData directories commonly seen



The diagram above shows how DLL sideloading abuses the DLL search order to load a malicious DLL with a legitimate name. In this example, Winmm.dll is loaded into good.exe. Most antivirus solutions miss this attack since they only look at good.exe, not the DLLs it loads. Kernel32.dll and User32.dll could not be loaded this way since Windows considers them "Known" DLLs. Such KnownDLLs will only be loaded from the system root (C:\windows\system32 on most machines).

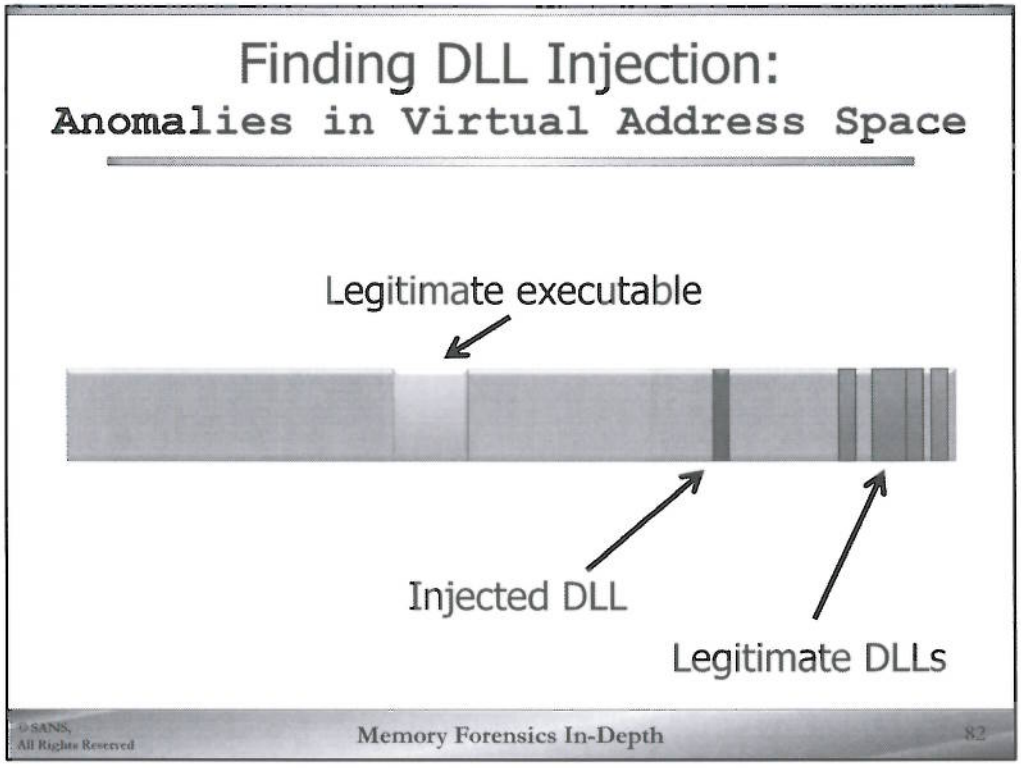


Using VADs to recover DLLs and executables is equivalent to what we've done before. We are dumping out the data pointed to by the FILE_OBJECT structures. We did this in previous modules with the procexedump and moddump plugins. The real magic of VADs is that we can find and dump out *any* region of virtual memory image in a process' memory space.

There are multiple malware techniques which this enables us to defeat. First, malware authors could hide their program as packed or encrypted data in the executable. When the program is run, it could allocate from memory, decompress the malicious program into that memory, and then execute it. By dumping out all of the VADs for a process, we would also dump out this newly allocated memory and the "hidden" malicious program as well. Or the program could download a second stage from the network, store that in allocated memory, and then execute it. Again, we're dumping the allocated memory out from the process and could see it right there. That memory region would not be listed as being mapped to a file.

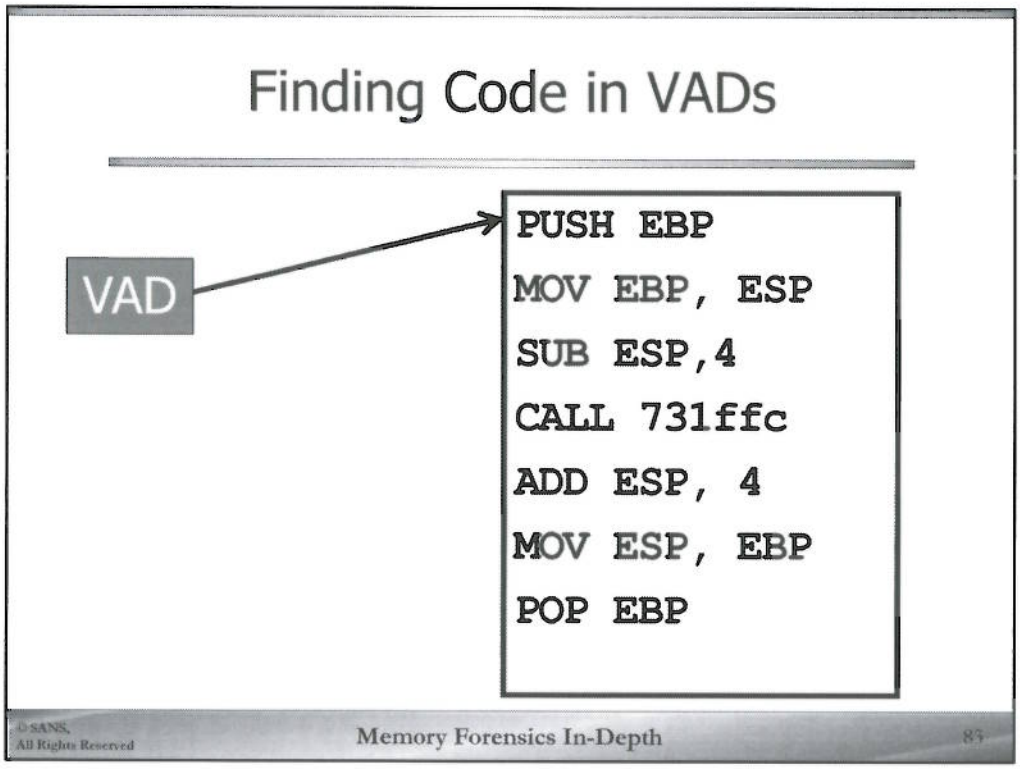
Thus, if we find a VAD which is not mapped to a file, but contains a PE32 header, we have a pretty good idea that we're looking at something malicious. Even better, the sample we've dumped out from the VAD can now be analyzed using strings, IDA Pro, or whatever tools you have at your disposal. These files can (and should) be scanned with anti-virus software. Or uploaded to Virus Total to scan them against multiple anti-virus engines.

Once again, the rootkit paradox rises to the occasion. We don't necessarily need to know what we're looking for. But the presence of a PE header in a VAD which is not mapped to a file is a dead giveaway that something is afoot!



Along with PE executables which are decompressed or downloaded into memory, DLLs can also be injected into another process. In a DLL injection attack, another process opens a handle to the victim process, loads the DLL into that process' memory space, and then creates a thread in the victim process which calls the newly loaded DLL. This has the advantage of hiding the DLL in the victim process and not needing to create a new, potentially suspicious process.

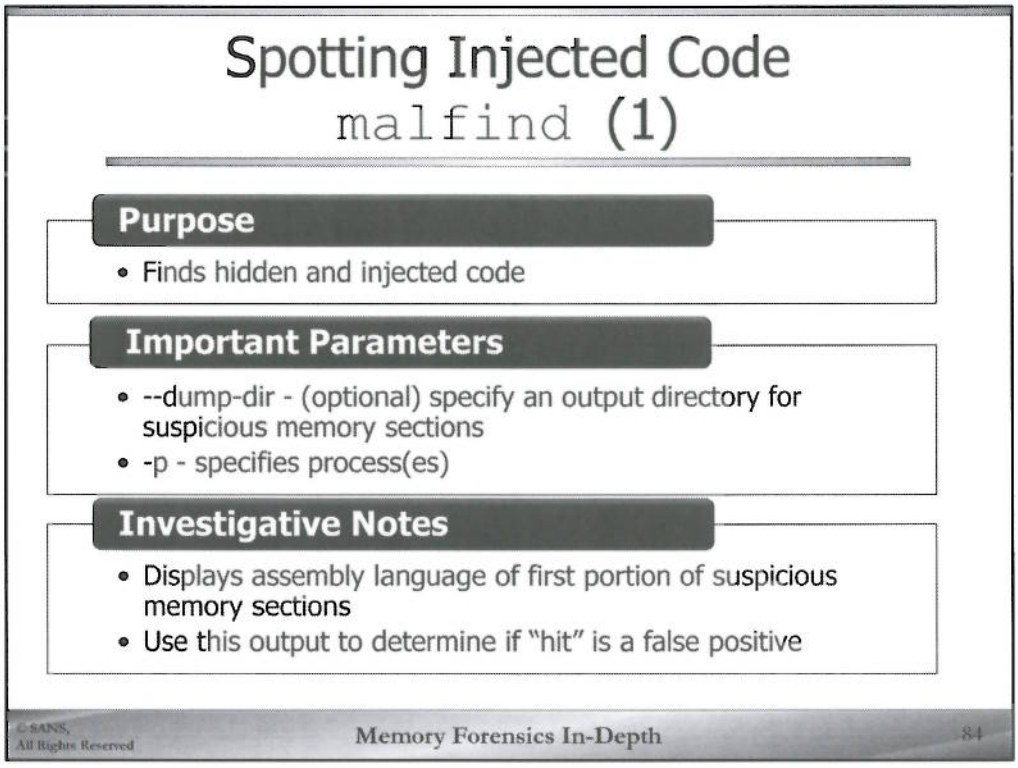
A walk of the victim's process VAD tree, however, will reveal the injected DLL. The DLL may be injected directly, or it may be mapped to a file on the disk. This DLL may not appear in the list of modules loaded by the executable, however, which is another way to detect this kind of attack. If the attacker did not use the legitimate Windows methods for loading a DLL, the injected DLL doesn't get added to the list of loaded modules.



Along with looking for entire DLLs in memory, it's also possible to find executable code in a VAD without a PE header. In the example given above about a program decompressing itself into a VAD, it's more than likely that the decompressed code will not have a PE header on it. The shell code will simply start at the beginning of the frame. There are still ways to find such code. Programs called disassemblers can convert the machine code used in programs back to assembly language. While the former is just a string of numbers, the latter is a series of opcodes which can be understood by humans*.

We can use a disassembler on the data recovered by a VAD to see if it parses correctly as assembly code. We can further use disassemblers to see if the code at the start of the VAD data matches well known compiler preambles—blocks of code which are often reused by standard compilers at the start of functions. In this manner we can start to automate the process. Rather than require the examiner to look through all of the recovered VADs, the computer can do it for us. In a few sections we will start to work with plugins which do that work for us. There is an automated solution, but for now, let's try it the manual way!

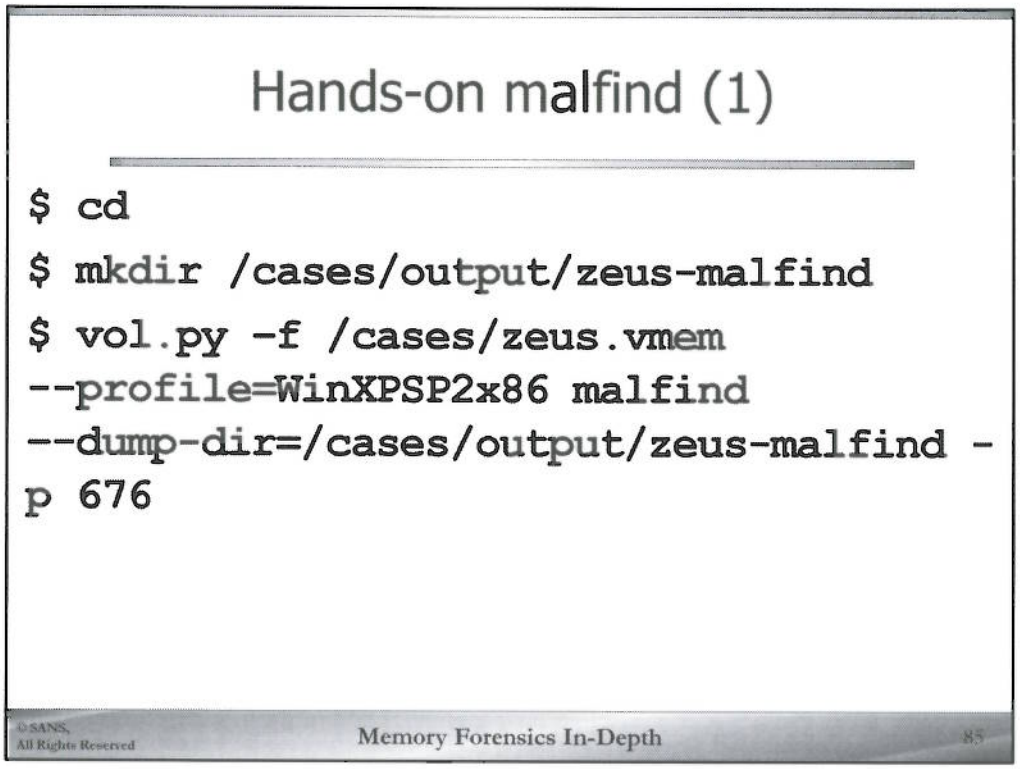
* If you ask programmers whether assembly can be parsed by humans you will get a variety of answers. Although it is immensely complicated, it *can* be understood.



Let's go in-depth with the malfind plugin. The malfind plugin searches for executable code in memory pages which are not backed by a file. It can also optionally search for strings in memory pages. For the complete reference on malfind, see http://code.google.com/p/volatility/wiki/CommandReference#malfind.

We did this same kind of work when we were looking at Virtual Address Descriptors. In that exercise we had to search, by hand, for VADs which contained executable code but didn't have a filename associated with them. The malfind plugin does that work for us. Let's see it in action by using it on the same memory image as we did for the VAD exercise.

As reminder, as with any kind of malicious software analysis, watch out for programs like Microsoft Security Essentials and others which look proactively for malware threats. They can detect the malware you are recovering from Windows memory images and may attempt to eliminate it. On one hand, this is a good thing—confirmation that the potential malware you are extracting is, in fact, malicious. But on the other hand, such programs are preventing you from conducting your own analysis. (Bonus question: how do such programs work? Are they hooking functions in the IDT? SSDT? Which ones?)



Let's start with malfind. We're going to use the zeus memory image, just as before. The malfind plugin can specify a directory to store its results with the --output-dir or -D flags. Though a dump directory was required in past versions of the malfind plugin, it is now optional. (You can also optionally limit malfind to single process with the --pid or -p flag. There are a few other options. Use malfind with the -h for more.) You'll make to make the output directory first:

```
$ cd
$ mkdir /cases/output/zeus-malfind
```

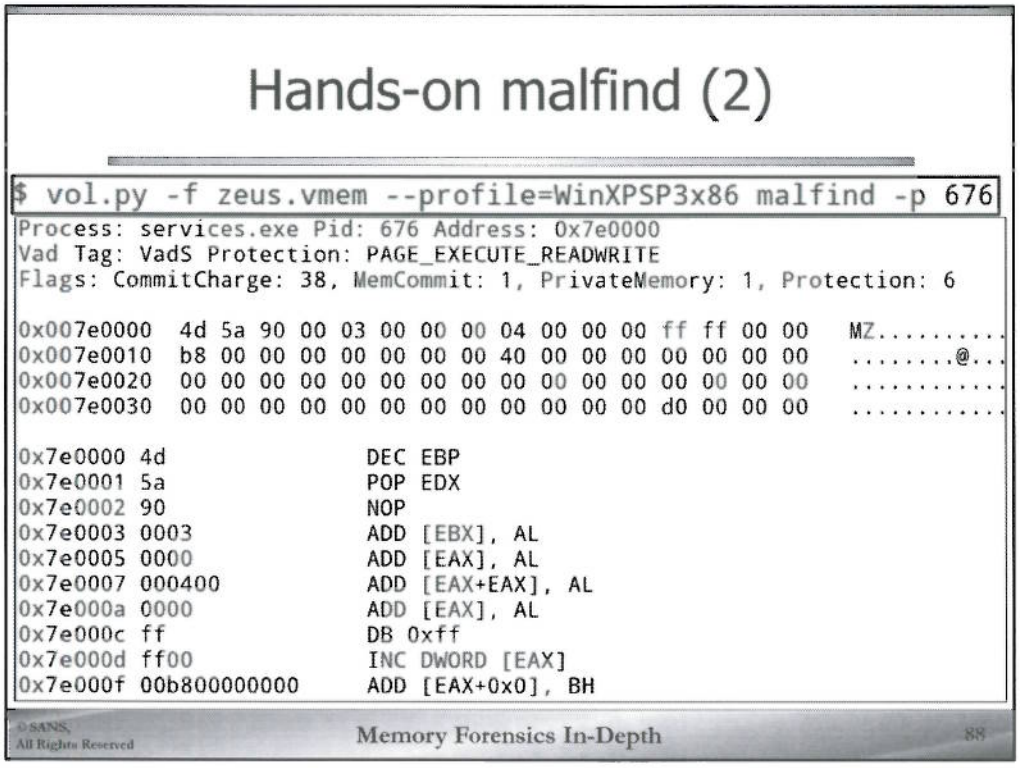
And then run the malfind plugin. We're going to limit it to a single process, services.exe, pid 676, for this example.

```
$ vol.py -f /cases/zeus.vmem --profile=WinXPSP2x86 malfind --dump-
dir=/cases/output/zeus-malfind -p 676
```

You should see this output:

```
Process: services.exe Pid: 676 Address: 0x7e0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 38, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x007e0000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x007e0010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x007e0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x007e0030  00 00 00 00 00 00 00 00 00 00 00 00 d0 00 00 00   ................

0x7e0000 4d              DEC EBP
0x7e0001 5a              POP EDX
0x7e0002 90              NOP
0x7e0003 0003            ADD [EBX], AL
0x7e0005 0000            ADD [EAX], AL
0x7e0007 000400          ADD [EAX+EAX], AL
0x7e000a 0000            ADD [EAX], AL
0x7e000c ff              DB 0xff
0x7e000d ff00            INC DWORD [EAX]
0x7e000f 00b800000000    ADD [EAX+0x0], BH
0x7e0015 0000            ADD [EAX], AL
0x7e0017 004000          ADD [EAX+0x0], AL
0x7e001a 0000            ADD [EAX], AL
0x7e001c 0000            ADD [EAX], AL
0x7e001e 0000            ADD [EAX], AL
0x7e0020 0000            ADD [EAX], AL
0x7e0022 0000            ADD [EAX], AL
0x7e0024 0000            ADD [EAX], AL
0x7e0026 0000            ADD [EAX], AL
0x7e0028 0000            ADD [EAX], AL
0x7e002a 0000            ADD [EAX], AL
0x7e002c 0000            ADD [EAX], AL
0x7e002e 0000            ADD [EAX], AL
0x7e0030 0000            ADD [EAX], AL
0x7e0032 0000            ADD [EAX], AL
0x7e0034 0000            ADD [EAX], AL
0x7e0036 0000            ADD [EAX], AL
0x7e0038 0000            ADD [EAX], AL
0x7e003a 0000            ADD [EAX], AL
0x7e003c d000            ROL BYTE [EAX], 0x1
0x7e003e 0000            ADD [EAX], AL
```

```
Process: services.exe Pid: 676 Address: 0x9e0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x009e0000  b8 35 00 00 00 e9 cd d7 f2 7b b8 91 00 00 00 e9
.5.......{......
0x009e0010  4f df f2 7b 8b ff 55 8b ec e9 ef 17 83 76 8b ff
O..{..U......v..
0x009e0020  55 8b ec e9 95 76 7e 76 8b ff 55 8b ec e9 be 53
U....v~v..U....S
0x009e0030  7f 76 8b ff 55 8b ec e9 d6 18 83 76 8b ff 55 8b
.v..U......v..U.

0x9e0000 b835000000       MOV EAX, 0x35
0x9e0005 e9cdd7f27b       JMP 0x7c90d7d7
0x9e000a b891000000       MOV EAX, 0x91
0x9e000f e94fdff27b       JMP 0x7c90df63
0x9e0014 8bff             MOV EDI, EDI
0x9e0016 55               PUSH EBP
0x9e0017 8bec             MOV EBP, ESP
0x9e0019 e9ef178376       JMP 0x7721180d
0x9e001e 8bff             MOV EDI, EDI
0x9e0020 55               PUSH EBP
0x9e0021 8bec             MOV EBP, ESP
0x9e0023 e995767e76       JMP 0x771c76bd
0x9e0028 8bff             MOV EDI, EDI
0x9e002a 55               PUSH EBP
0x9e002b 8bec             MOV EBP, ESP
0x9e002d e9be537f76       JMP 0x771d53f0
0x9e0032 8bff             MOV EDI, EDI
0x9e0034 55               PUSH EBP
0x9e0035 8bec             MOV EBP, ESP
0x9e0037 e9d6188376       JMP 0x77211912
0x9e003c 8bff             MOV EDI, EDI
0x9e003e 55               PUSH EBP
0x9e003f 8b               DB 0x8b
```



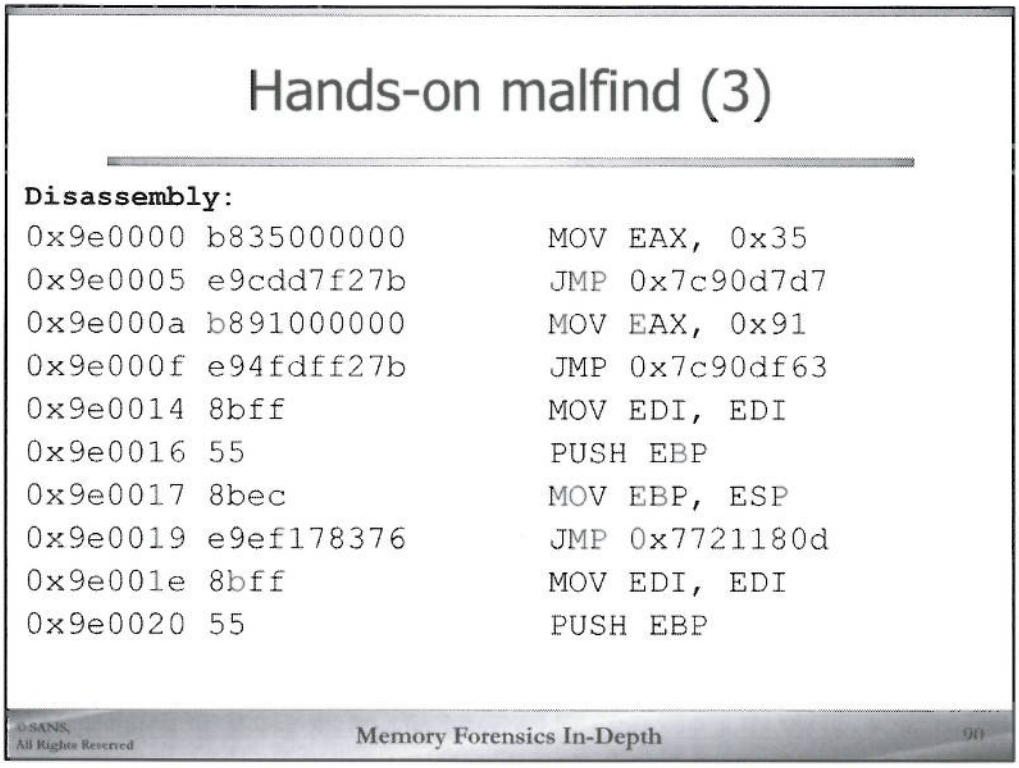
The plugin found two VADs in the services.exe process which it thought were suspicious. The first was:

```
Process: services.exe Pid: 676 Address: 0x7e0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 38, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x007e0000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x007e0010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x007e0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x007e0030  00 00 00 00 00 00 00 00 00 00 00 00 d0 00 00 00   ................

0x7e0000 4d              DEC EBP
0x7e0001 5a              POP EDX
0x7e0002 90              NOP
0x7e0003 0003            ADD [EBX], AL
0x7e0005 0000            ADD [EAX], AL
0x7e0007 000400          ADD [EAX+EAX], AL
0x7e000a 0000            ADD [EAX], AL
0x7e000c ff              DB 0xff
0x7e000d ff00            INC DWORD [EAX]
0x7e000f 00b800000000    ADD [EAX+0x0], BH
0x7e0015 0000            ADD [EAX], AL
0x7e0017 004000          ADD [EAX+0x0], AL
0x7e001a 0000            ADD [EAX], AL
```

```
0x7e001c 0000              ADD [EAX], AL
0x7e001e 0000              ADD [EAX], AL
0x7e0020 0000              ADD [EAX], AL
0x7e0022 0000              ADD [EAX], AL
0x7e0024 0000              ADD [EAX], AL
0x7e0026 0000              ADD [EAX], AL
0x7e0028 0000              ADD [EAX], AL
0x7e002a 0000              ADD [EAX], AL
0x7e002c 0000              ADD [EAX], AL
0x7e002e 0000              ADD [EAX], AL
0x7e0030 0000              ADD [EAX], AL
0x7e0032 0000              ADD [EAX], AL
0x7e0034 0000              ADD [EAX], AL
0x7e0036 0000              ADD [EAX], AL
0x7e0038 0000              ADD [EAX], AL
0x7e003a 0000              ADD [EAX], AL
0x7e003c d000              ROL BYTE [EAX], 0x1
0x7e003e 0000              ADD [EAX], AL
```

In this case malfind found a VAD in the process which contained a PE executable but wasn't backed by a file. The VAD covered the range of virtual memory from 0x7e0000 to 0x805fff00. (That range alone, by the way, is rather odd. It's a total of 0x7fe1ff00 bytes, or about 2GB of memory.) Following this header you can see a hex dump of the bytes at the start of the VAD. This contains an MZ header which is found at the start of such executables, along with the string "This program cannot be found in DOS mode". The plugin dumped the memory associated with that VAD to a file, services.exe.6015020.007e0000-00805fff.dmp, which you can reverse engineer. As a shortcut for this class, go ahead and submit it to Virus Total. You should find that this file is seen as very bad.

You may be asking yourself, "Self, if this VAD reserved 2GB of virtual memory, why is the extracted file only 113KB?" Remember, the memory associated with this VAD was reserved, but not necessarily committed. Windows wouldn't actually create the structures for the whole allocation, or any of it, actually, until they were needed. Memory which was never used, and thus never committed, is not carved out. It's also possible that parts of this VAD would be in the paging file, but we'd have to get very low-level to figure out that. In high-value cases, you may have to get that low level!



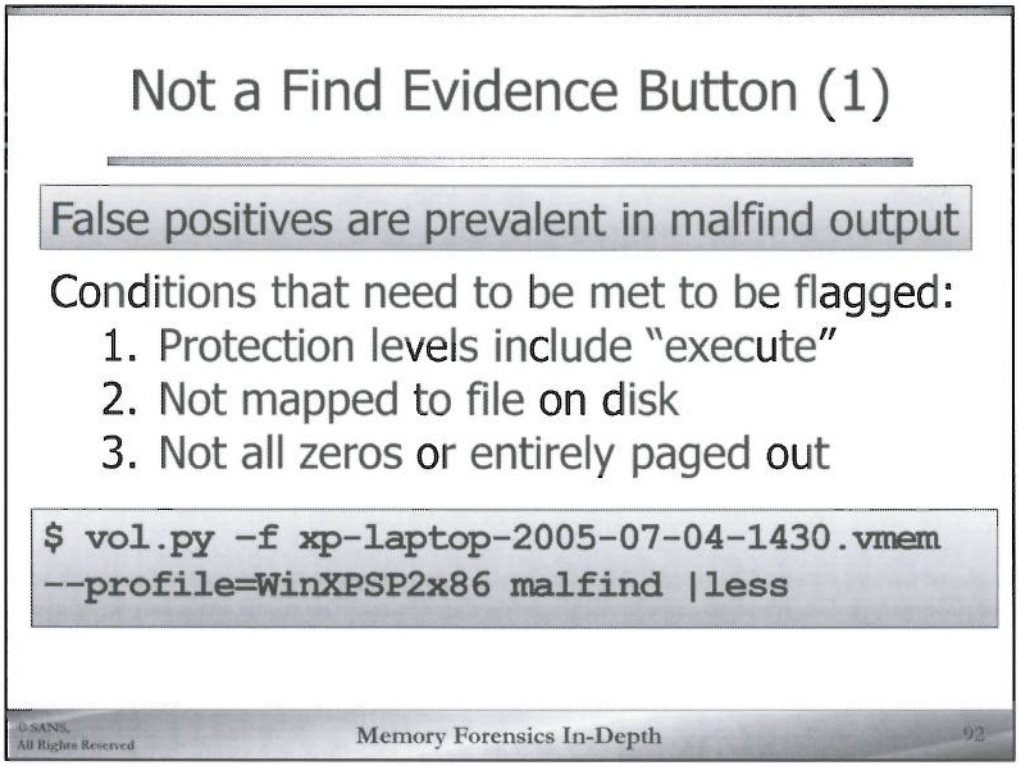
The other result from malfind is an example of the plugin recognizing executable code at the start of a VAD. This is also a suspicious behavior. It could be that a piece of malicious code was decompressed into here, or that the executable code was injected directly without a PE header. Regardless, such code is highly irregular and worthy of further examination. Here's what malfind reported:

```
Process: services.exe Pid: 676 Address: 0x9e0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x009e0000  b8 35 00 00 00 e9 cd d7 f2 7b b8 91 00 00 00 e9   .5.......{......
0x009e0010  4f df f2 7b 8b ff 55 8b ec e9 ef 17 83 76 8b ff   O..{..U......v..
0x009e0020  55 8b ec e9 95 76 7e 76 8b ff 55 8b ec e9 be 53   U....v~v..U....S
0x009e0030  7f 76 8b ff 55 8b ec e9 d6 18 83 76 8b ff 55 8b   .v..U......v..U.

Disassembly:
0x9e0000 b835000000       MOV EAX, 0x35
0x9e0005 e9cdd7f27b       JMP 0x7c90d7d7
0x9e000a b891000000       MOV EAX, 0x91
0x9e000f e94fdff27b                 JMP 0x7c90df63
0x9e0014 8bff                       MOV EDI, EDI
0x9e0016 55                         PUSH EBP
0x9e0017 8bec                       MOV EBP, ESP
0x9e0019 e9ef178376       JMP 0x7721180d
0x9e001e 8bff                       MOV EDI, EDI
```

```
0x9e0020 55                    PUSH EBP
0x9e0021 8bec                  MOV EBP, ESP
0x9e0023 e995767e76        JMP 0x771c76bd
0x9e0028 8bff                   MOV EDI, EDI
0x9e002a 55                    PUSH EBP
0x9e002b 8bec                  MOV EBP, ESP
0x9e002d e9be537f76        JMP 0x771d53f0
0x9e0032 8bff                  MOV EDI, EDI
0x9e0034 55                    PUSH EBP
0x9e0035 8bec                  MOV EBP, ESP
0x9e0037 e9d6188376        JMP 0x77211912
0x9e003c 8bff                   MOV EDI, EDI
0x9e003e 55                    PUSH EBP
0x9e003f 8b                    DB 0x8b
```

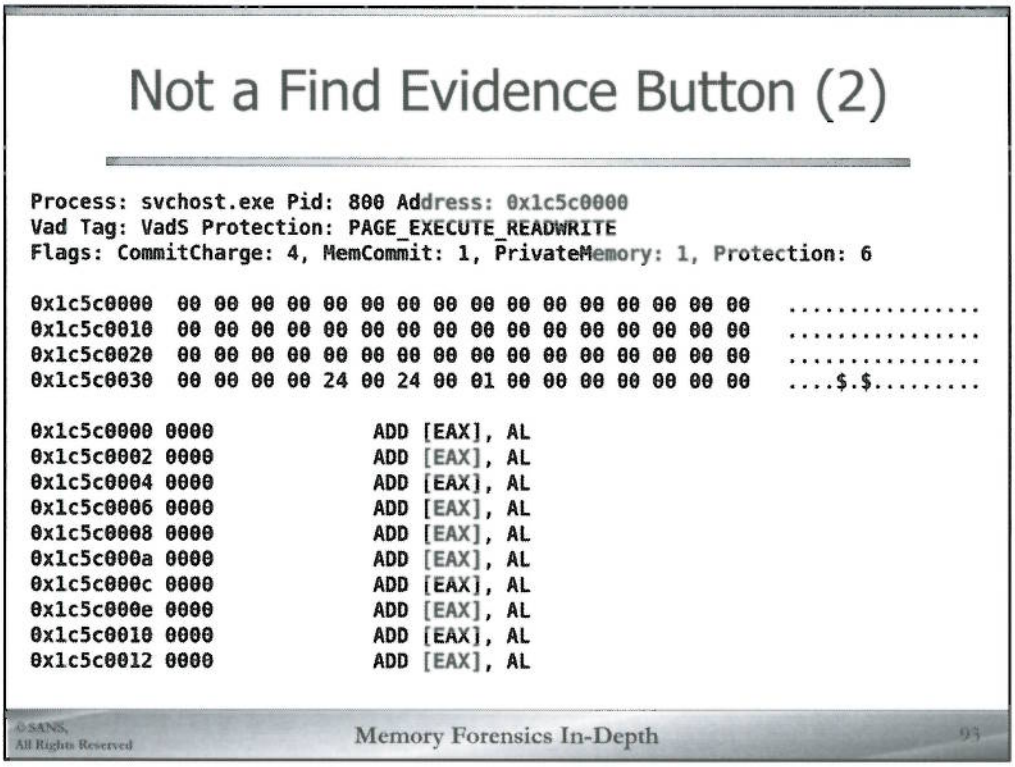
Here we can see a more reasonably sized VAD. The hex dump of those bytes doesn't make much sense. But the malfind plugin uses distorm to attempt to disassemble those bytes. The result, seen below, is reasonably sensible assembly code. Again, malfind has saved the memory pages for us, this time to the file services.exe.6015020.009e0000-009e0fff.dmp.

What do you get when you scan this file with VirusTotal? As of press time, no results. Virus total is generally not good at sections of raw code. Or it's possible that this VAD didn't actually contain anything bad. We'll see a better example of that on the next page.



Like every other tool in computer forensics, the MHL plugins are not a "Find Evidence" button. They won't solve a case for you. Although they can be helpful, they are not a replacement for understanding what was happening in a memory image. The plugins will often return or flag data which is innocuous, but *could* be suspicious. We spent so much time earlier in the course on what really matters so that you can correctly interpret the results. For example, let's use the malfind plugin on a memory image we know to be completely harmless, our old friend, the XP laptop image:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
malfind | less
```



The plugin will highlight seven "suspicious" VADs which it reports begin with executable code. They all look something like:

Process: svchost.exe Pid: 800 Address: 0x1c5c0000

Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE

Flags: CommitCharge: 4, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x1c5c0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....

0x1c5c0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....

0x1c5c0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....

0x1c5c0030 00 00 00 00 24 00 24 00 01 00 00 00 00 00 00 00 ....$.$.........

0x1c5c0000 0000 ADD [EAX], AL

0x1c5c0002 0000 ADD [EAX], AL

0x1c5c0004 0000 ADD [EAX], AL

0x1c5c0006 0000 ADD [EAX], AL

0x1c5c0008 0000 ADD [EAX], AL

0x1c5c000a 0000 ADD [EAX], AL

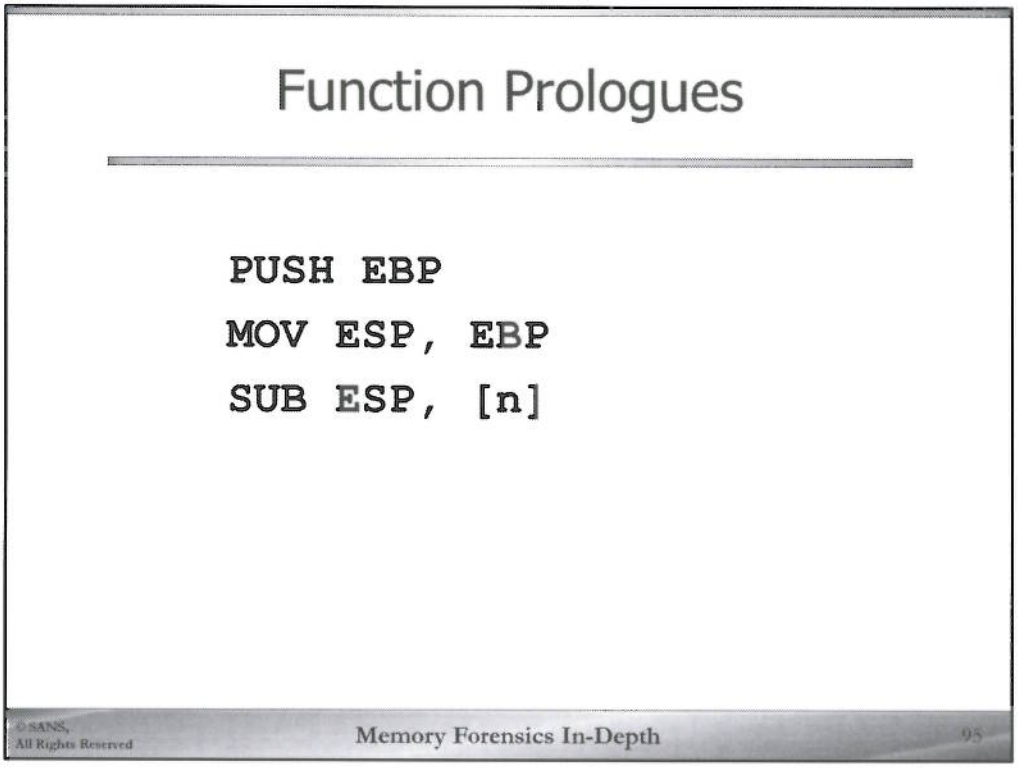
0x1c5c000c 0000 ADD [EAX], AL

0x1c5c000e 0000 ADD [EAX], AL

0x1c5c0010 0000 ADD [EAX], AL

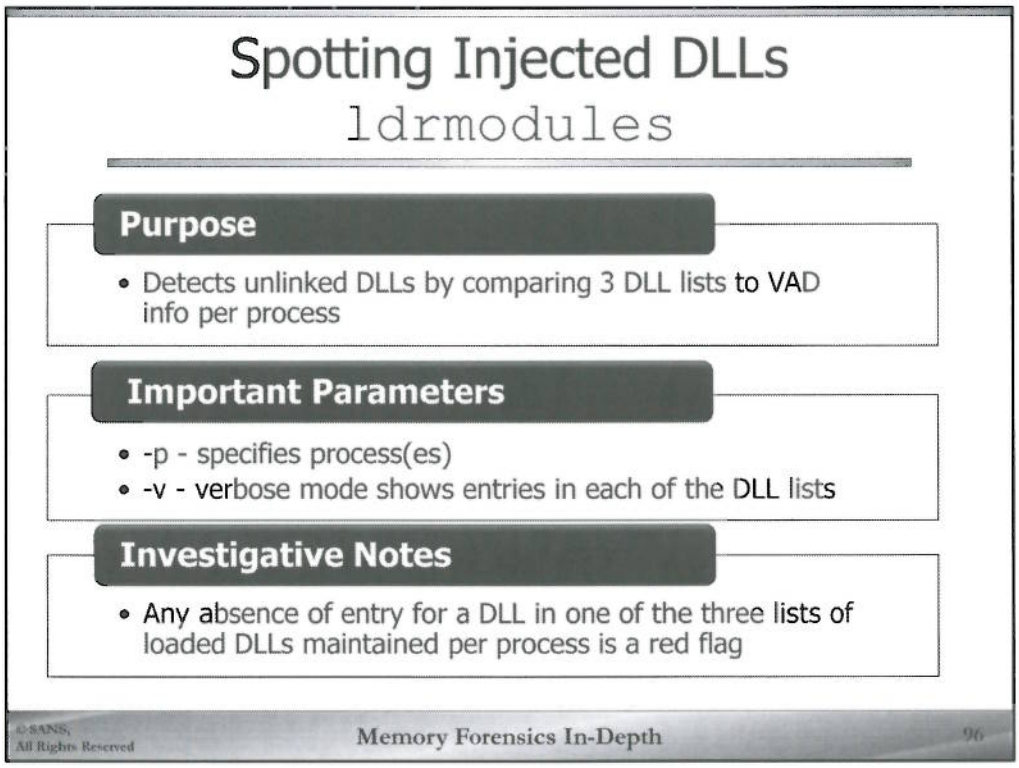
0x1c5c0012 0000 ADD [EAX], AL

You can see that the bytes at the start of the page *do* disassemble into code. Specifically these bytes, *which are all zeros*, disassemble into instructions to add one register to another, over and over again. It is certainly possible that a malware author did write code which was this… useless. But it's more likely that a block which starts off with all zeros is not executable code. Here the malfind plugin has done its job, but the reported data is not actually suspicious. Don't assume that because malfind found something that it is guaranteed to be something suspicious!

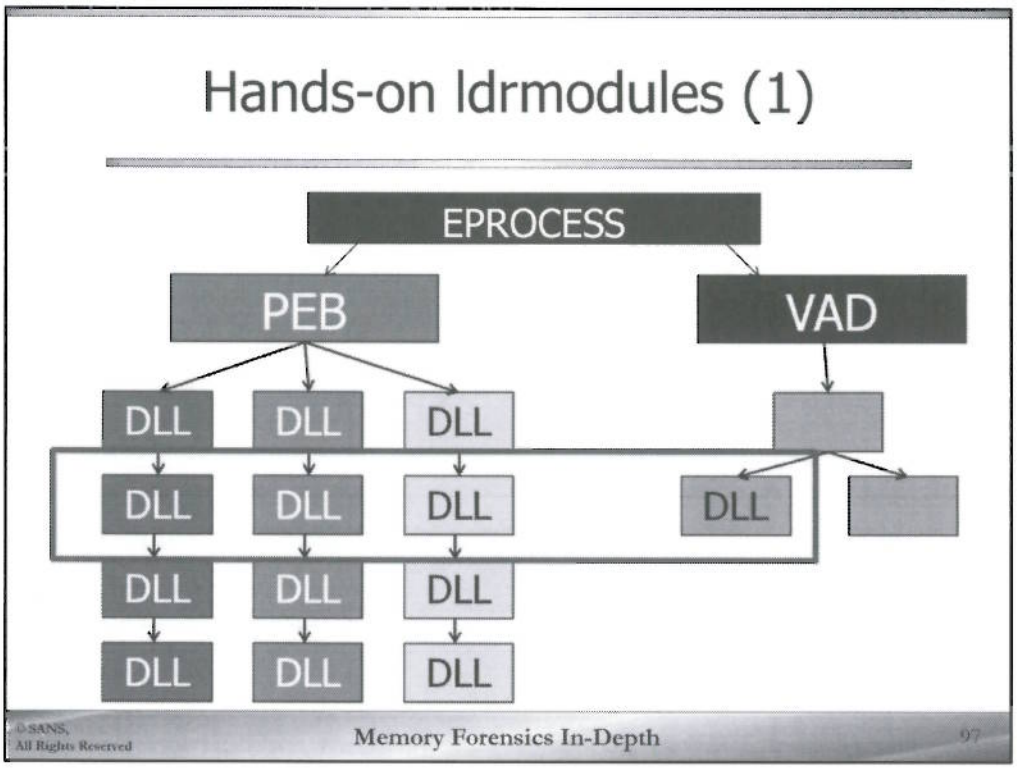


On the other hand, blocks of actual code tend to start with function prologues. These are small pieces of assembly language code which are often found at the start of individual functions. The exact instructions vary per compiler, but in general they have a standard form. They push the value of the base pointer, EBP, onto the stack, copy the base pointer in the stack pointer, ESP, and then change the value of the stack pointer in some manner. It could be an addition or a subtraction operation.

We're not going to cover assembly language in this course. In general memory forensics is about finding the malware in question, not analyzing it. You can learn MUCH more about function prologues and reverse engineering in FOR610.



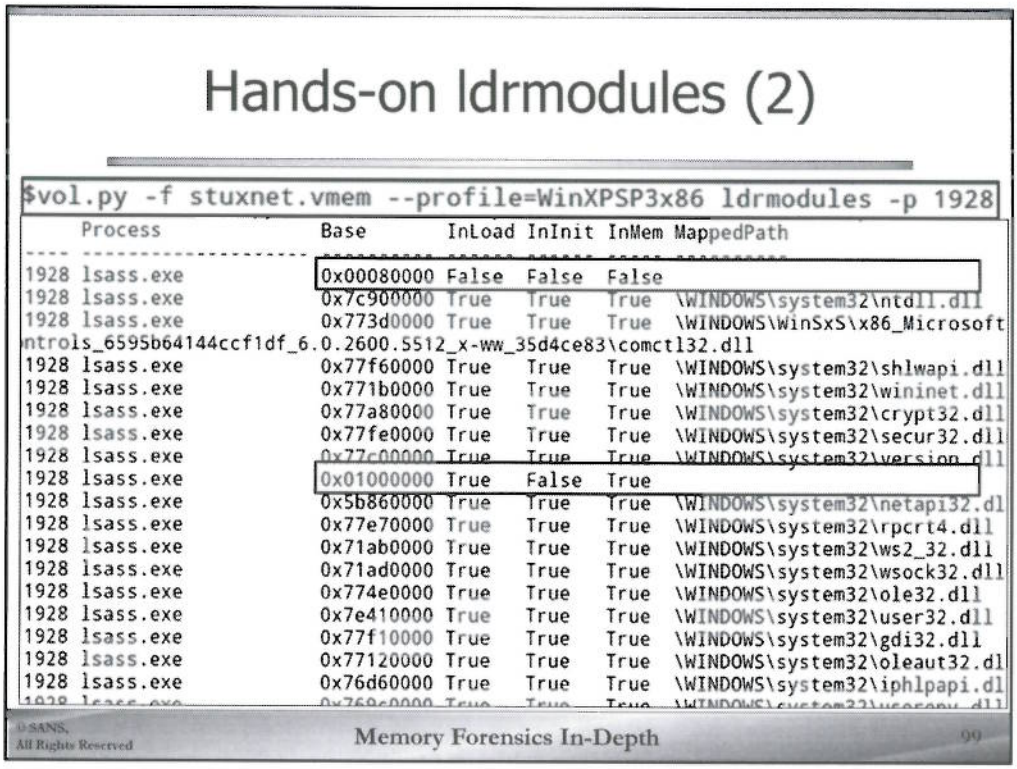
Malicious programmers will try to hide the DLLs they load into a program. To do this they have to eliminate information from the lists of DLLs loaded into each process. There are in fact three lists of DLLs linked to from the process' Process Environment Block (PEB). These lists are the InLoadOrderModuleList, the InMemoryOrderModuleList and the InInitializationOrderModuleList. Although malware can certainly eliminate entries from those lists, there is also the entry in the VAD indicating the mapped DLL. MHL wrote a plugin, **ldrmodules**, which compares the lists of DLLs loaded into a process with the mapped files in the VADs. The plugin indicates for each DLL if it is present in the lists referenced by the PEB, and the filename in the VAD.



Let's take a look at the memory image for the Stuxnet sample, and in particular the lsass.exe process, pid 1928. If we first examine the DLLs loaded into the process, using one of the lists referenced by the PEB, everything looks fine:

```
$ vol.py -f /cases/stuxnet.vmem --profile=WinXPSP3x86 dlllist -p 1928
Volatile Systems Volatility Framework 2.3
************************************************************************
lsass.exe pid:   1928
Command line : "C:\WINDOWS\\system32\\lsass.exe"
Service Pack 3

Base                  Size          LoadCount    Path
----------  ----------  ----------   ----
0x01000000   0x6000                  0xffff      C:\WINDOWS\system32\lsass.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
0x7e410000    0x91000        0xffff             C:\WINDOWS\system32\USER32.dll
0x77f10000    0x49000        0xffff             C:\WINDOWS\system32\GDI32.dll
0x00870000   0x138000        0x1
           C:\WINDOWS\system32\KERNEL32.DLL.ASLR.0360b7ab
0x76f20000    0x27000        0x2                C:\WINDOWS\system32\DNSAPI.dll
0x77c10000    0x58000        0x27               C:\WINDOWS\system32\msvcrt.dll
0x71ab0000    0x17000        0xa                C:\WINDOWS\system32\WS2_32.dll
0x71aa0000     0x8000        0x8                C:\WINDOWS\system32\WS2HELP.dll
0x76d60000    0x19000        0x2                C:\WINDOWS\system32\IPHLPAPI.DLL
0x5b860000    0x55000        0x2                C:\WINDOWS\system32\NETAPI32.dll
0x774e0000   0x13d000        0x5                C:\WINDOWS\system32\ole32.dll
0x77120000    0x8b000        0x4                C:\WINDOWS\system32\OLEAUT32.dll
0x76bf0000     0xb000        0x2                C:\WINDOWS\system32\PSAPI.DLL
0x7c9c0000   0x817000        0x2                C:\WINDOWS\system32\SHELL32.dll
0x77f60000    0x76000        0x8                C:\WINDOWS\system32\SHLWAPI.dll
0x769c0000    0xb4000        0x2                C:\WINDOWS\system32\USERENV.dll
0x77c00000     0x8000        0x2                C:\WINDOWS\system32\VERSION.dll
0x771b0000    0xaa000        0x2                C:\WINDOWS\system32\WININET.dll
0x77a80000    0x95000        0x2                C:\WINDOWS\system32\CRYPT32.dll
0x77b20000    0x12000        0x2                C:\WINDOWS\system32\MSASN1.dll
0x71ad0000     0x9000        0x2                C:\WINDOWS\system32\WSOCK32.dll
0x773d0000   0x103000        0x2
           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
```



But what happens if we use the ldrmodules plugin, we can see that there are some DLLs which have been removed from the PEB lists:

```
$ vol.py -f /cases/stuxnet.vmem --profile=WinXPSP3x86 ldrmodules -p 1928
lsass.exe pid:   1928
Command line : "C:\WINDOWS\\system32\\lsass.exe"
Service Pack 3

Pid      Process    Base       InLoad  InInit InMem         MappedPath
-------- ---------  -----      ------  ------- --------   ---------------
1928     lsass.exe  0x00080000 False   False   False
1928     lsass.exe  0x7c900000 True    True    True   \WINDOWS\system32\ntdll.dll
1928     lsass.exe  0x773d0000 True    True    True
\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-
ww_35d4ce83\comctl32.dll
1928     lsass.exe  0x77f60000 True    True    True   \WINDOWS\system32\shlwapi.dll
1928     lsass.exe  0x771b0000 True    True    True   \WINDOWS\system32\wininet.dll
1928     lsass.exe  0x77a80000 True    True    True   \WINDOWS\system32\crypt32.dll
1928     lsass.exe  0x77fe0000 True    True    True   \WINDOWS\system32\secur32.dll
```

```
 1928 lsass.exe  0x77c00000          True   True   True  \WINDOWS\system32\version.dll
1928 lsass.exe  0x01000000          True   False  True
1928 lsass.exe  0x5b860000          True   True   True \WINDOWS\system32\netapi32.dll
1928 lsass.exe  0x77e70000          True   True   True  \WINDOWS\system32\rpcrt4.dll
1928 lsass.exe  0x71ab0000          True   True   True  \WINDOWS\system32\ws2_32.dll
1928 lsass.exe  0x71ad0000          True   True   True  \WINDOWS\system32\wsock32.dll
1928 lsass.exe  0x774e0000          True   True   True  \WINDOWS\system32\ole32.dll
1928 lsass.exe  0x7e410000          True   True   True  \WINDOWS\system32\user32.dll
1928 lsass.exe  0x77f10000          True   True   True  \WINDOWS\system32\gdi32.dll
1928 lsass.exe  0x77120000          True   True   True \WINDOWS\system32\oleaut32.dll
1928 lsass.exe  0x76d60000          True   True   True \WINDOWS\system32\iphlpapi.dll
1928 lsass.exe  0x769c0000          True   True   True  \WINDOWS\system32\userenv.dll
1928 lsass.exe  0x7c800000          True   True   True \WINDOWS\system32\kernel32.dll
1928 lsass.exe  0x76bf0000          True   True   True  \WINDOWS\system32\psapi.dll
1928 lsass.exe  0x77c10000          True   True   True  \WINDOWS\system32\msvcrt.dll
1928 lsass.exe  0x77dd0000          True   True   True \WINDOWS\system32\advapi32.dll
1928 lsass.exe  0x7c9c0000          True   True   True  \WINDOWS\system32\shell32.dll
1928 lsass.exe  0x00870000          True   True   True
1928 lsass.exe  0x76f20000          True   True   True  \WINDOWS\system32\dnsapi.dll
1928 lsass.exe  0x5d090000          True   True   True \WINDOWS\system32\comctl32.dll
1928 lsass.exe  0x71aa0000          True   True   True  \WINDOWS\system32\ws2help.dll
1928 lsass.exe  0x77b20000          True   True   True  \WINDOWS\system32\msasn1.dll
```

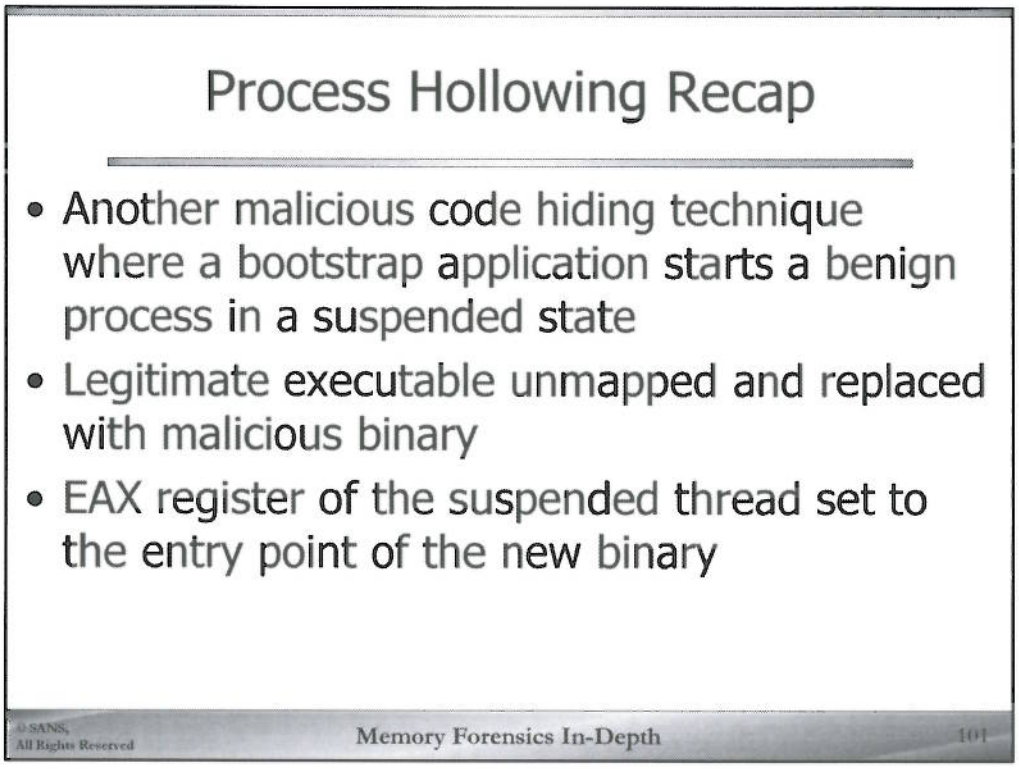
Here we can see three DLLs which are suspicious. The entries at 0x80000, 0x1000000, and 0x870000 are either missing from one of the PEB lists or have a blank filename in the VAD.

Let's keep digging here. We can get more information from the ldrmodules plugin by using the -v flag for verbose mode. This causes the plugin to display the filenames from each of the PEB lists for each DLL. This is also handy if the malware attempts to hide by changing the filename listed in the VAD. There shouldn't be any mismatches between the filename in the PEB lists and the filename in the VAD.

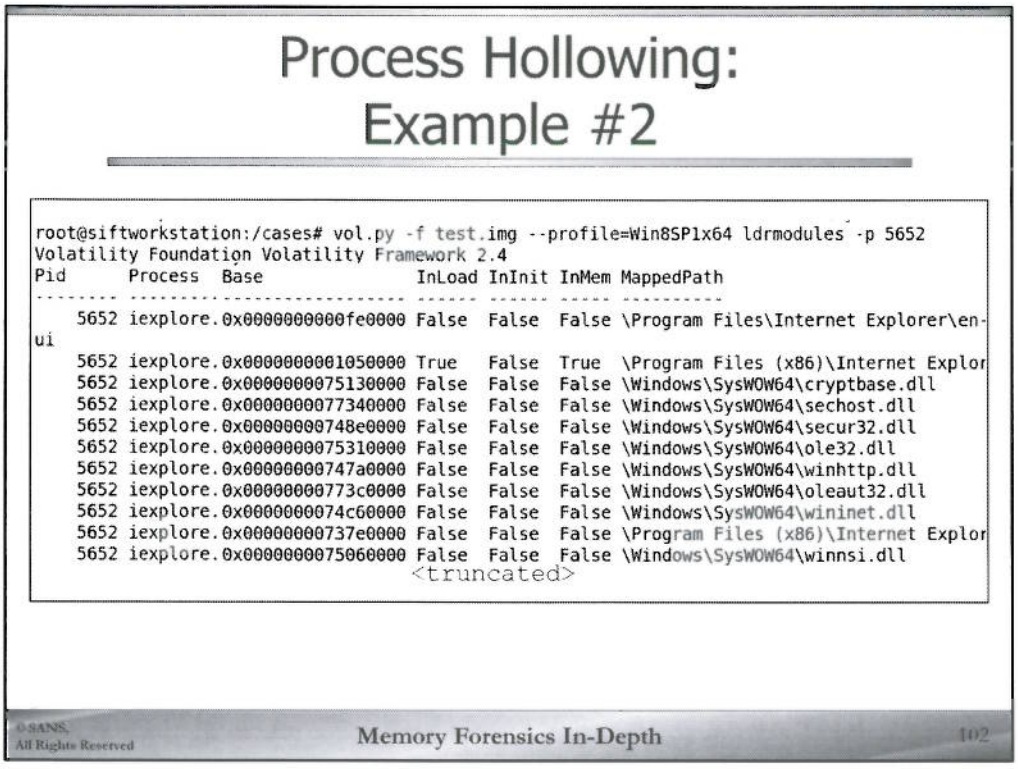
Running the plugin again, this time with the -v flag, shows us something interesting for the module at 0x100000:

```
$ vol.py -f /cases/stuxnet.vmem --profile=WinXPSP3x86 ldrmodules  -p 1928 -v

1928     lsass.exe            0x1000000    1        0        1        -
  Load Path: C:\WINDOWS\system32\lsass.exe : lsass.exe
  Mem Path:  C:\WINDOWS\system32\lsass.exe : lsass.exe
```

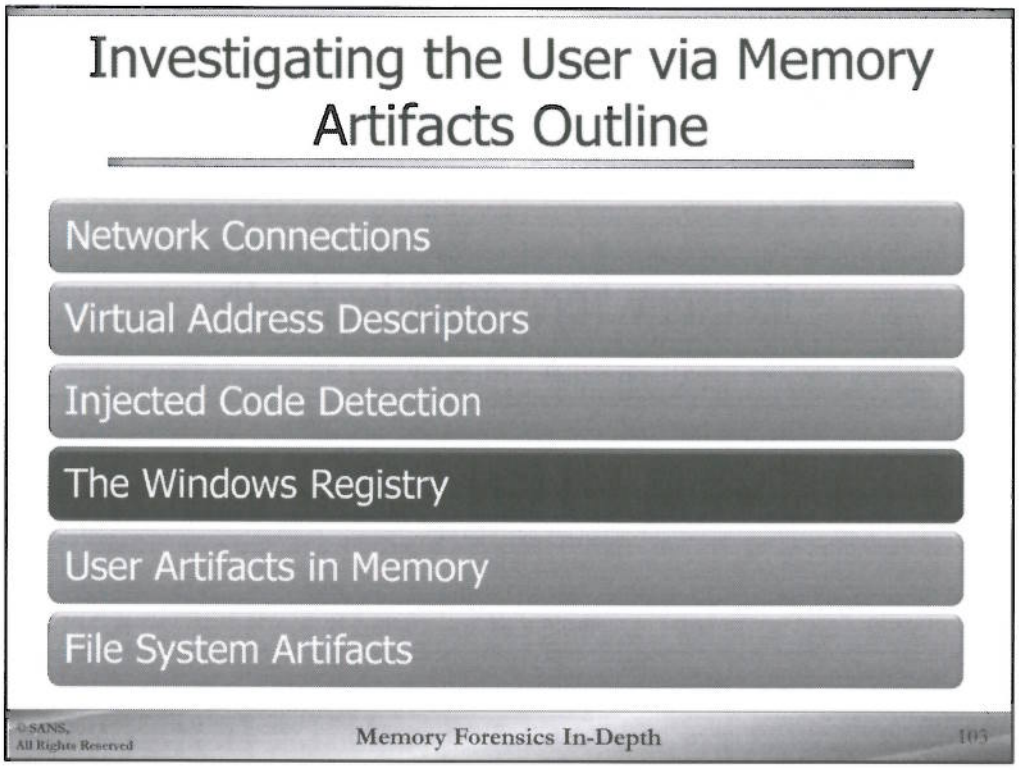
The file *looks* like the legitimate executable, but doesn't have a valid filename in the VAD. What happened? It's a hollow process, which we'll describe on the next page.



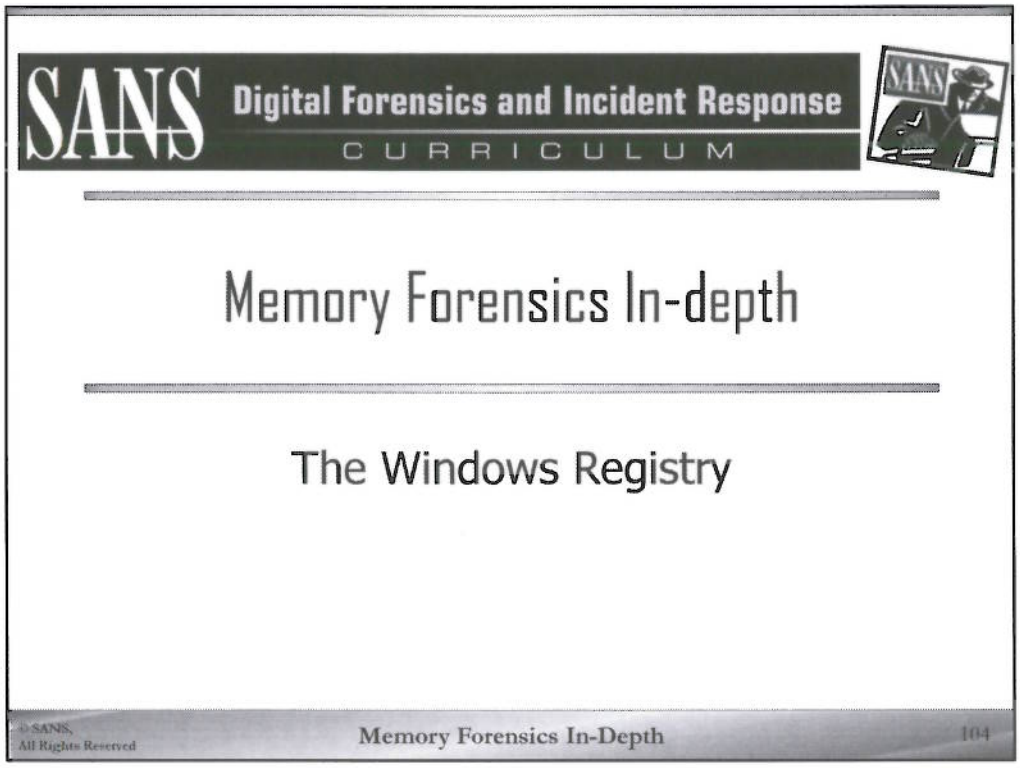
Process hollowing, as seen in the lsass process PID 1928 on the previous page, is another malicious code hiding technique. A bootstrap application launches a benign process in a suspended state. The legitimate executable is then unmapped and replaced with the malicious code to be hidden. The EAX register of the suspended thread of the process is set to the entry point of the malicious binary so as the process resumes, the entry point of the new binary is executed.



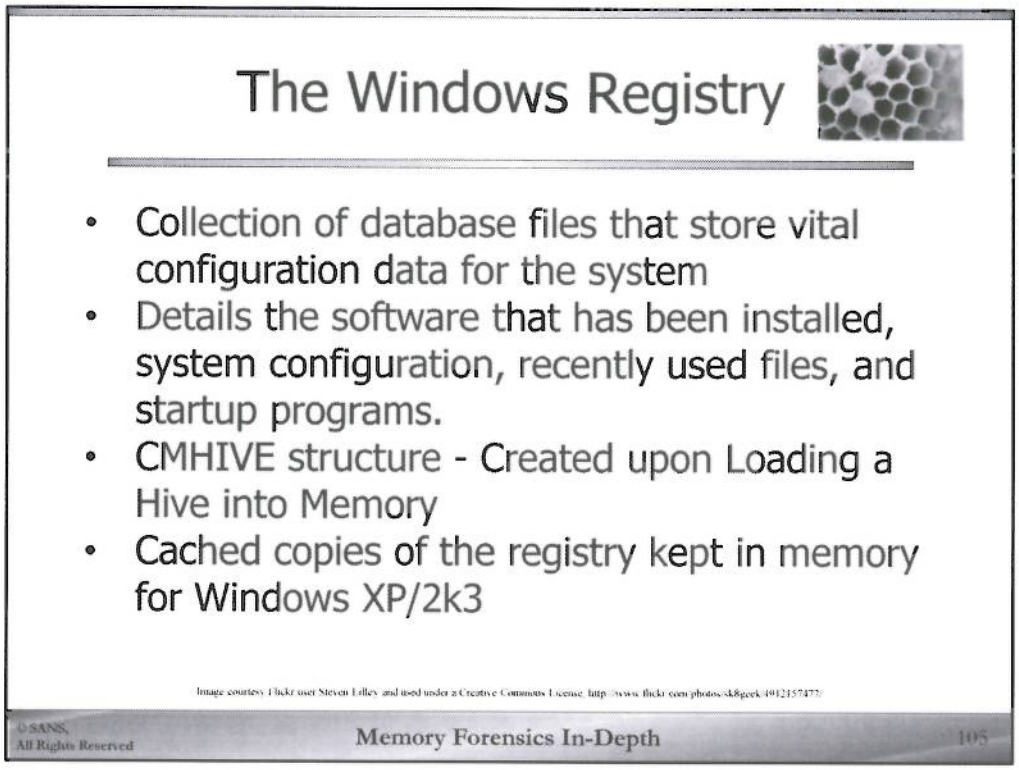
Another way process hollowing presents itself in ldrmodules output shows that suspicious virtual address ranges can have mapped paths still associated with them but not show up any of the three PEB lists of loaded dlls. In this example of a "Couponarific" adware variant, the dlls of the iexplore process have been unmapped from the PEB lists.



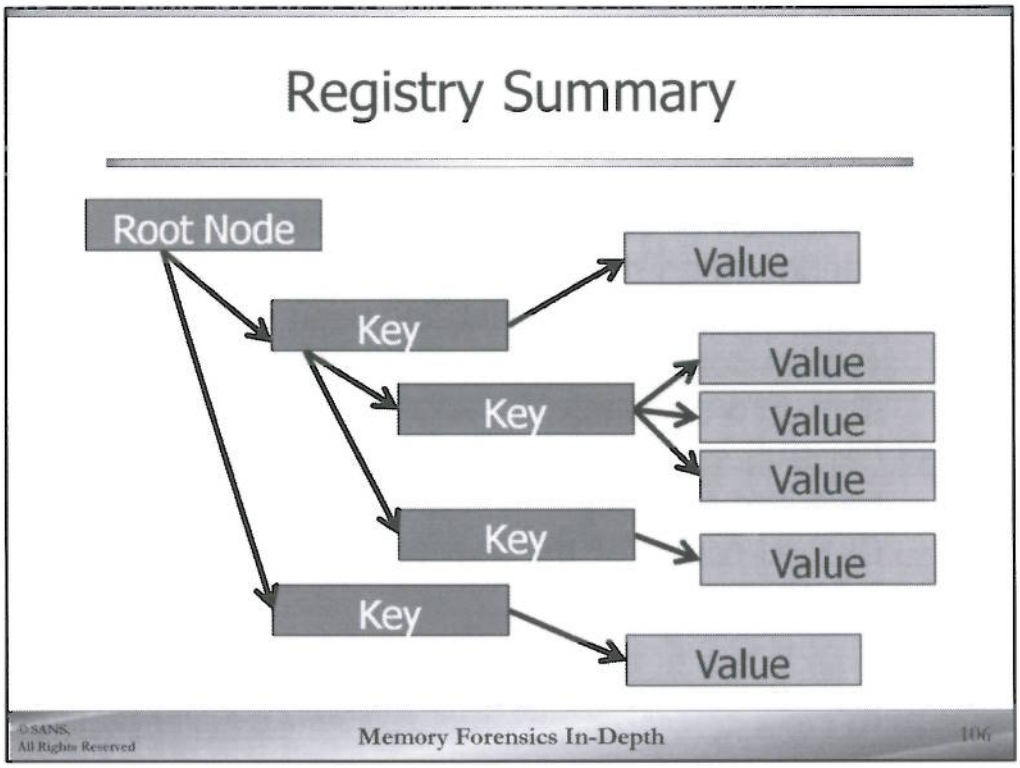
This page intentionally left blank.



This page intentionally left blank.



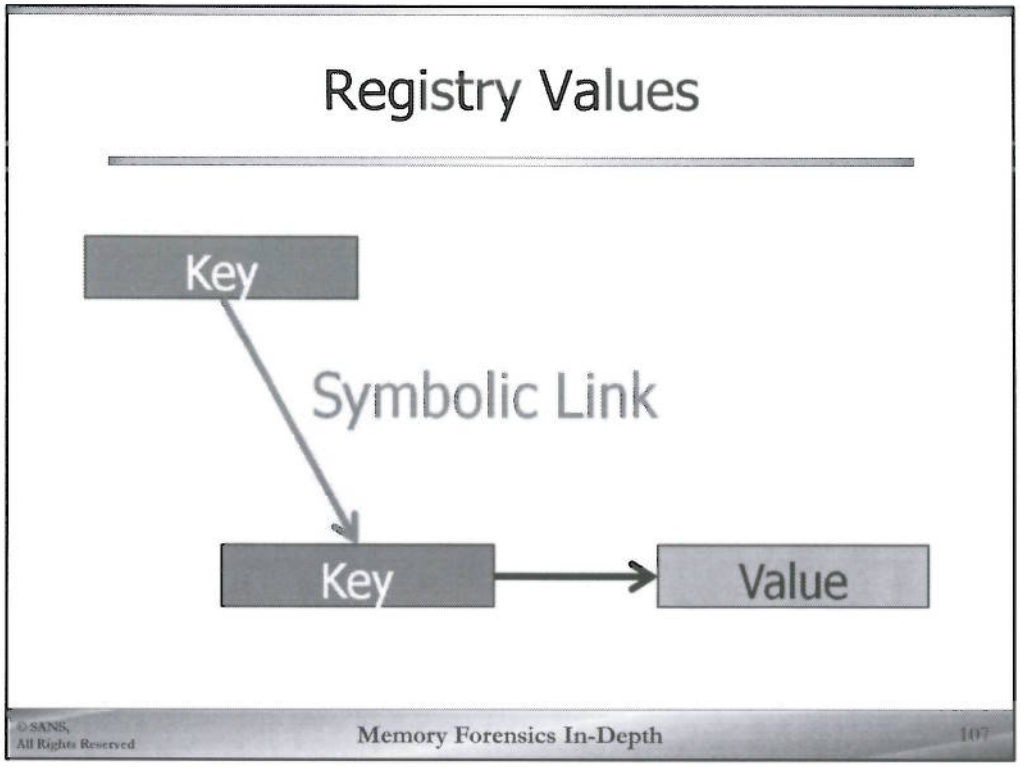
The Windows registry is a fantastic resource for traditional forensics, and even more so for memory forensics. In this section, we are going to discuss the parts of the Windows registry which are specific to memory forensics. There are entire books dedicated to registry forensics—we couldn't possibly cover all of it in this course. But we will do a basic review of how Windows uses the registry. We will then discuss the volatile hives, or the parts of the registry which are never written to the disk. Then we'll describe some useful registry artifacts you can find and how to extract them using the Volatility framework. These artifacts include password hashes, the USB keys which have been mounted on the system, and the wireless access points the system connected to.



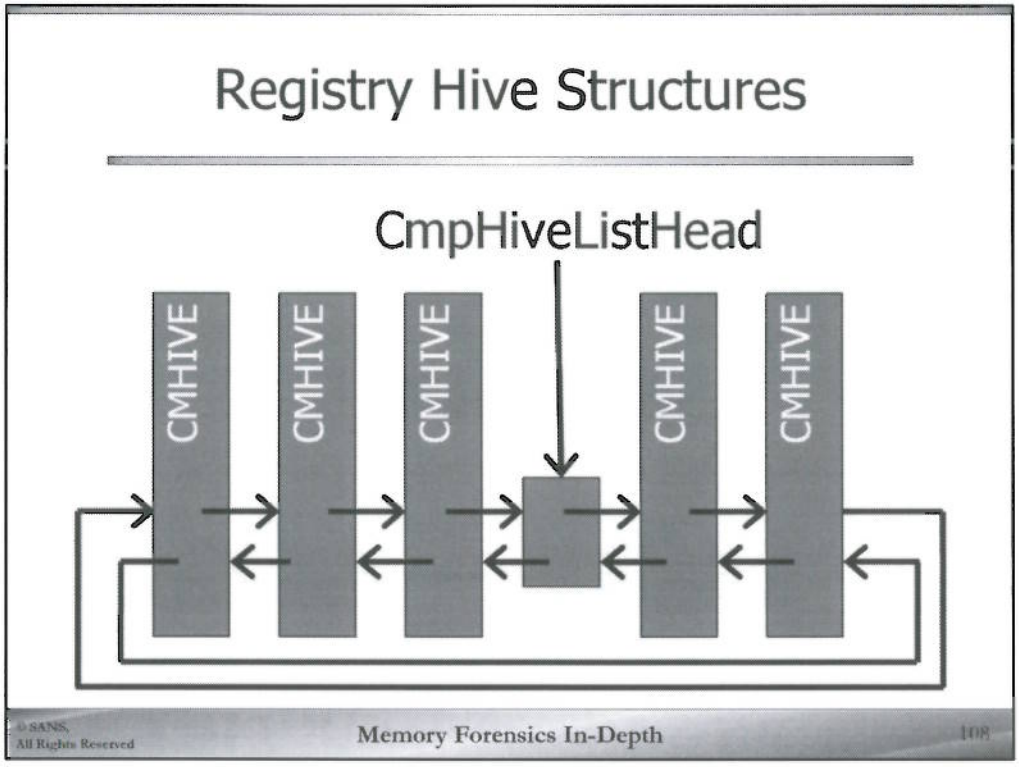
The registry is a hierarchical storage system used by Microsoft Windows. Used by both operating system components and ordinary programs, it provides a stable way to hold data for programs between runs and reboots. The registry is divided into parts called hives. Each hive holds a certain kind of information. Inside of each hive there is a root node which in turn points to a set of registry keys. Each key can point to zero or more values and zero or more subkeys. The result is that each key has a path, much like a file system, which defines it. For example, a key in the hive HKEY_LOCAL_MACHINE could have the path \SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces. Each entry between the slashes represents a key.

There are several hives common to all modern Windows systems. The ones you will deal with most often are HKEY_LOCAL_MACHINE (HKLM) and HKEY_CURRENT_USER (HKCU). These hives contain information about the system as a whole or a specific user, respectively.

Some of the registry hives are not stored on the disk. They are generated each time the machine boots. These hives include HKEY_CURRENT_CONFIG (HKCC). There is a set of keys under HKLM, HARDWARE, that is also dynamically generated. These keys describe the hardware found on the machine. Other important keys under the HKLM hive are the SAM, SECURITY, SYSTEM, and SOFTWARE hives. The SAM hive, or Security Accounts Manager, most notably for our purposes, contains the usernames and hashed passwords for all users on the system.



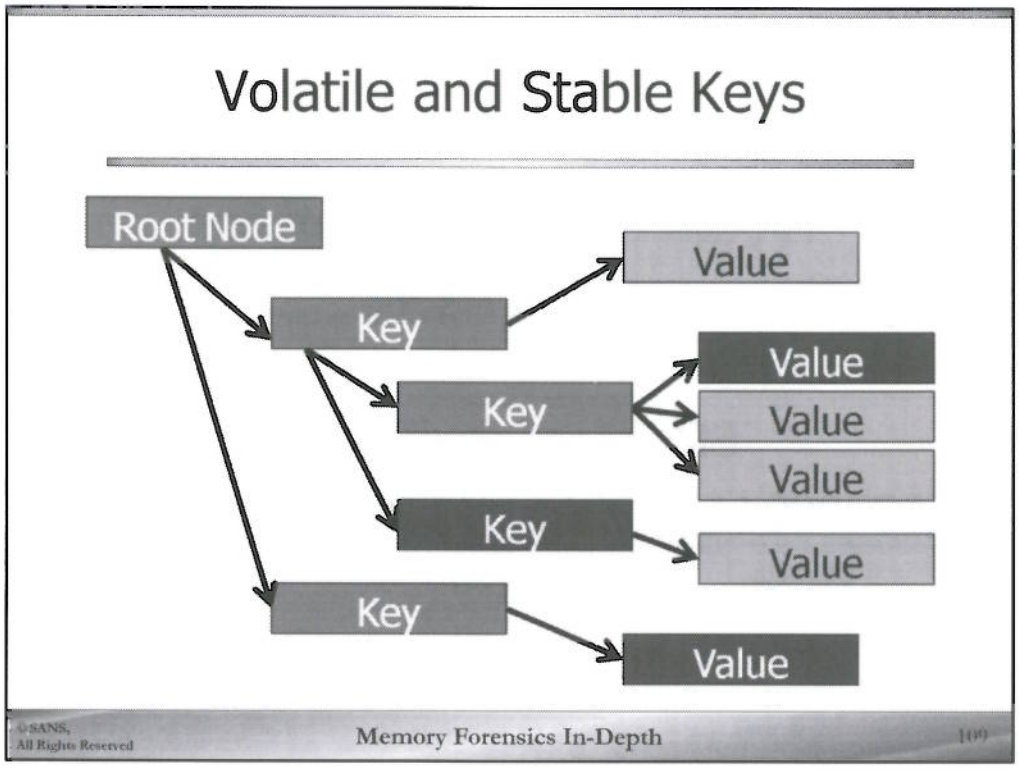
Registry values can contain no data (i.e., be a flag), a string, binary data, a number, a symbolic link, or a list of strings. Along with keys pointing directly to values and subkeys, registry keys can be symbolic links which point to other keys. A symbolic link is a special kind of value which points to another registry key.



Most references on the registry describe how the hives are stored on the disk. Their storage in memory is slightly different. The registry hive structures are maintained by the configuration manager. Each hive is described by a structure called CMHIVE (where CM stands for "configuration manager). These structures begin with a signature, which should be 0xbee0bee0. (Get it? Hives? Bees?) Technically the CMHIVE structure begins with an HHIVE structure, and the signature is the first element in the HHIVE structure, but regardless, you should still see that signature at the start of the structure. See the first part of this structure as seen in WindowsXPx86 shown below.

These CMHIVE structures are part of a doubly-linked list of structures. Just as we saw with processes, there is a list head for the registry hive structures. This list head is called CmpHiveListHead. These structures are all stored in pool memory and have the pool tag "CM10".

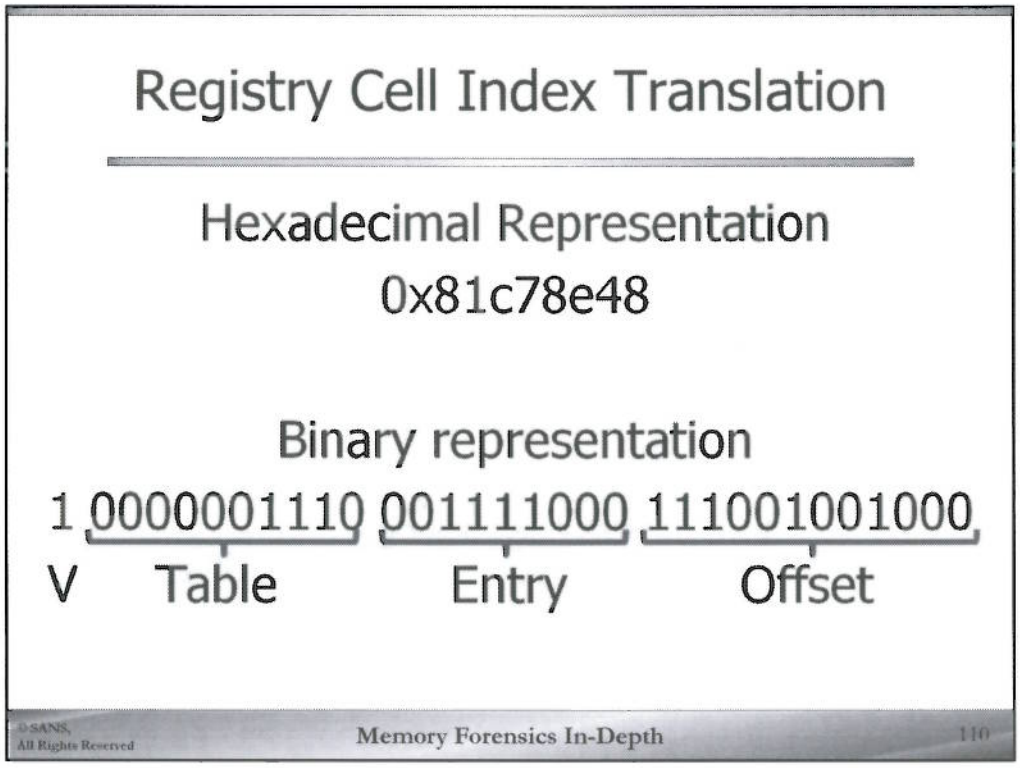
```
In [4]: dt("_CMHIVE")
'_CMHIVE' (1180 bytes)
0x0   : Hive                    ['_HHIVE']
0x210 : FileHandles               ['array', 3, ['pointer', ['void']]]
0x21c : NotifyList              ['_LIST_ENTRY']
0x224 : HiveList                ['_LIST_ENTRY'] ← Contains the flink and blink
0x22c : HiveLock                ['pointer', ['_FAST_MUTEX']]
0x230 : ViewLock                ['pointer', ['_FAST_MUTEX']]
0x234 : LRUViewListHead           ['_LIST_ENTRY']
0x23c : PinViewListHead         ['_LIST_ENTRY']
0x244 : FileObject              ['pointer', ['_FILE_OBJECT']]
0x248 : FileFullPath            ['_UNICODE_STRING']
0x250 : FileUserName              ['_UNICODE_STRING']
0x258 : MappedViews               ['unsigned short']
<output truncated>
```



Most of the registry hives are backed by files on the disk, but some are not. In addition, when changes are made to the registry, they are not immediately written out to the disk. For performance reasons, these values will remain in memory only for a little while. Every so often Windows will update the files on the disk with all of the changes to the registry in memory.

To record which registry keys and values are different in memory than on the disk, Windows will store those keys in the volatile portion of the hive. The other part of the hive is considered the stable part of the hive. Note that both parts of the hive are available to drivers and programs while the system is running. From their perspective, both the stable and volatile keys are part of the hive—there is no difference between them. In the picture above, we highlighted some keys as being volatile in red. Note that they are still part of the existing structure.

Although volatile keys are normally written back to the files on the disk, it is possible for a malicious programmer to modify keys and values such that they are *not* written back to the disk. In this case the programmer could make changes to the registry, such as altering access protections, logging policies, or even password hashes, which are not reflected on the disk. This kind of attack was proposed by Brendan Dolan-Gavitt in a paper called, "Forensic analysis of the Windows registry in memory", http://www.dfrws.org/2008/proceedings/p26-dolan-gavitt.pdf.



Each reference to a key, subkey, or value is made through a registry cell index. You can think of the cell index as a virtual address for the registry structure. The operating system (and we) have to translate these cell indexes into virtual addresses to determine where each data structure is stored. (Yes, these virtual addresses need to be translated again to find the physical addresses where the data live in a memory image.)

The process to translate a cell index into a location in the hive is different in memory than on the disk. When hives are stored on the disk, they are considered "Flat", and there is a simple equation to get the offset of any cell. The offset in the file is equal to the cell index plus 0x1000 + 4.

FileAddress = CellIndex + 0x1000 + 4

It is possible to have a Flat hive in memory, and in fact there is a flag for this condition.

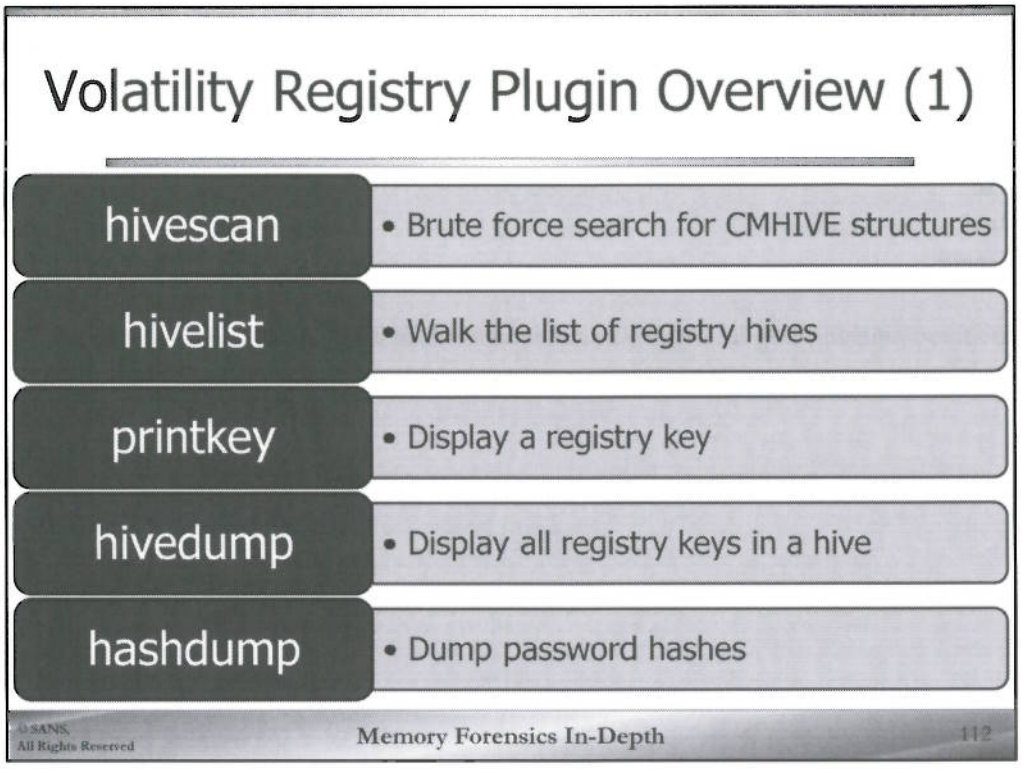
We will not cover all of the details of the in-memory cell index translation process. It's not something you should have to manually deal with during your investigations as your software tool should do it for you. If you need more details, however, check out http://moyix.blogspot.com/2008/02/cell-index-translation.html. Here is a brief overview:

Just like the virtual to physical address translation process has multiple stages with lookups, so does the cell index translation process. There are four parts to the cell index. The first bit indicates whether or not the index is in the stable or volatile part of the hive. A zero indicates the stable part of the hive; a one indicates the volatile part of the hive. Each part of the hive has a different Map value, or base address. The next ten bits of the cell index are the table index, which points into a series of tables. This is followed by nine bits for an index number of which entry in that table we want. The remaining twelve bits are an offset into that entry.

Let's look at an example. The cell index 0x81c78e48 in binary is 0b10000001110001111000111001001000. When we break this value into its component parts we see immediately that it is in the volatile part of the hive. After that:

Volatile = 1
Table = 0xe
Entry = 0x78
Offset = 0xe48

We can use these components to look up the virtual address which corresponds to the cell index.



There are a total of ten Volatility plugins which work with the Windows registry. The first is **hivescan**, which does a brute force search for CMHIVE structures stored in pool memory using the pool tag "CM10". This plugin is just a pool scanner for those pool tags.

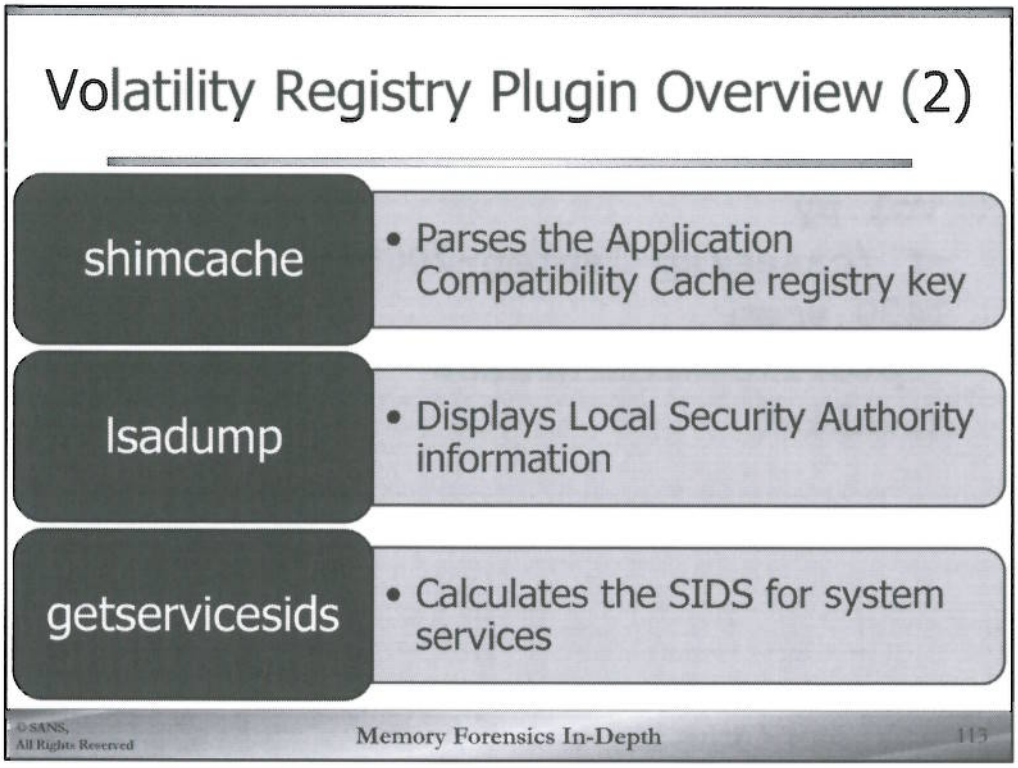
The second is **hivelist**. Given an initial offset of a registry hive, this plugin walks the links between the registry hives and displays the filename of each of them. If you don't supply an offset, this plugin calls hivescan and gets an offset from that output.

The **printkey** plugin will display the subkeys and values for any specified key name. If the key name exists in more than one hive, this plugin will display the keys from each hive.

The **hivedump** plugin will display all of the keys, but not values, from a hive. The hive to dump is specified with the virtual address in hexadecimal.

The **hashdump** plugin dumps out the Lanman (LM) and NTLanman (NTLM) password hashes for all users on the system. These hashes can be cracked using a variety of tools to find the user's passwords. Note that you must give this plugin the virtual addresses of the System and SAM hives using the -y and -s flags, respectively. You can get those addresses from the hivelist plugin.

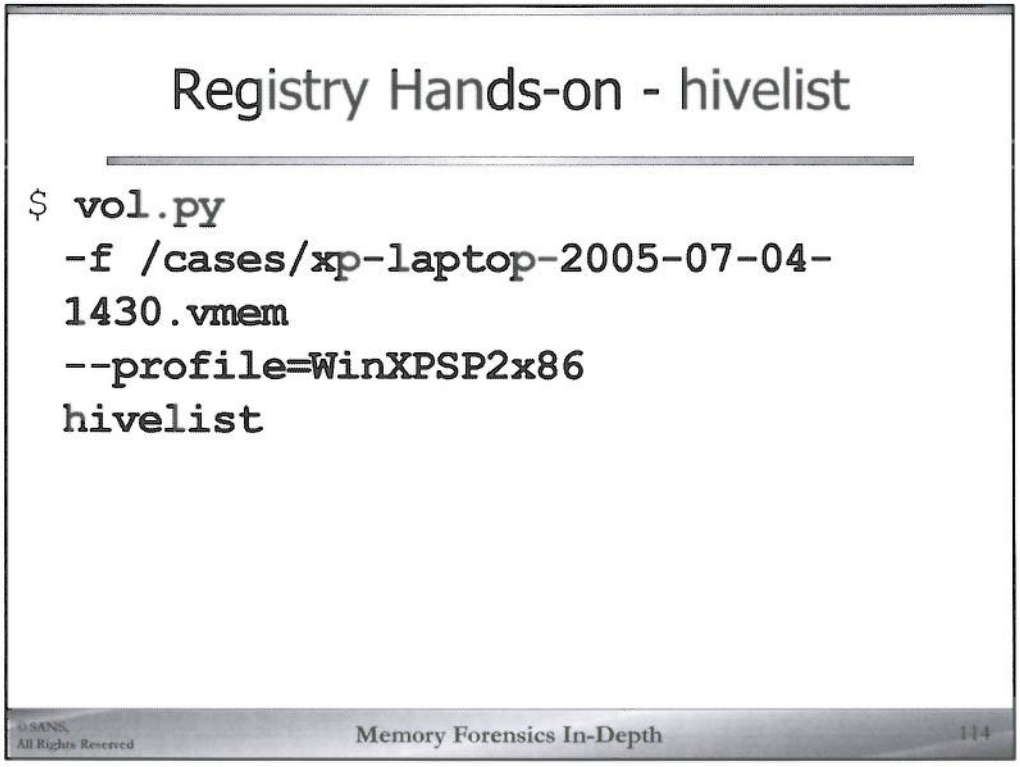
You will probably want to use the NTLM hashes from modern systems. Although LM hashes are still included on modern systems, they were disabled by default starting with Windows Vista. You should see that all users have the null LM hash, aad3b435b51404eeaad3b435b51404ee.



Shimcache, added to the Volatility framework as of version 2.3, parses the Application Compatibility Cache registry key. The values found here have been tied to evidence of execution. More on this key and its investigative usefulness can be found here at the Mandiant blog website: https://www.mandiant.com/blog/leveraging-application-compatibility-cache-forensic-investigations/

The **lsadump** plugin is useful on Windows XP systems for displaying information from the Local Security Authority (LSA). We are not covering it in this course, but you can read more at http://moyix.blogspot.com/2008/02/decrypting-lsa-secrets.html.

The final registry plugin in the Volatility 2.3 framework is **getservicesids** which calculates the service SIDS for the system.



Let's put all of this theory into practice with the Volatility plugins for examining Windows registry hives. We're going to use the xp-laptop memory image for these hands on exercises. First let's run the hivelist plugin to see what hives are available:

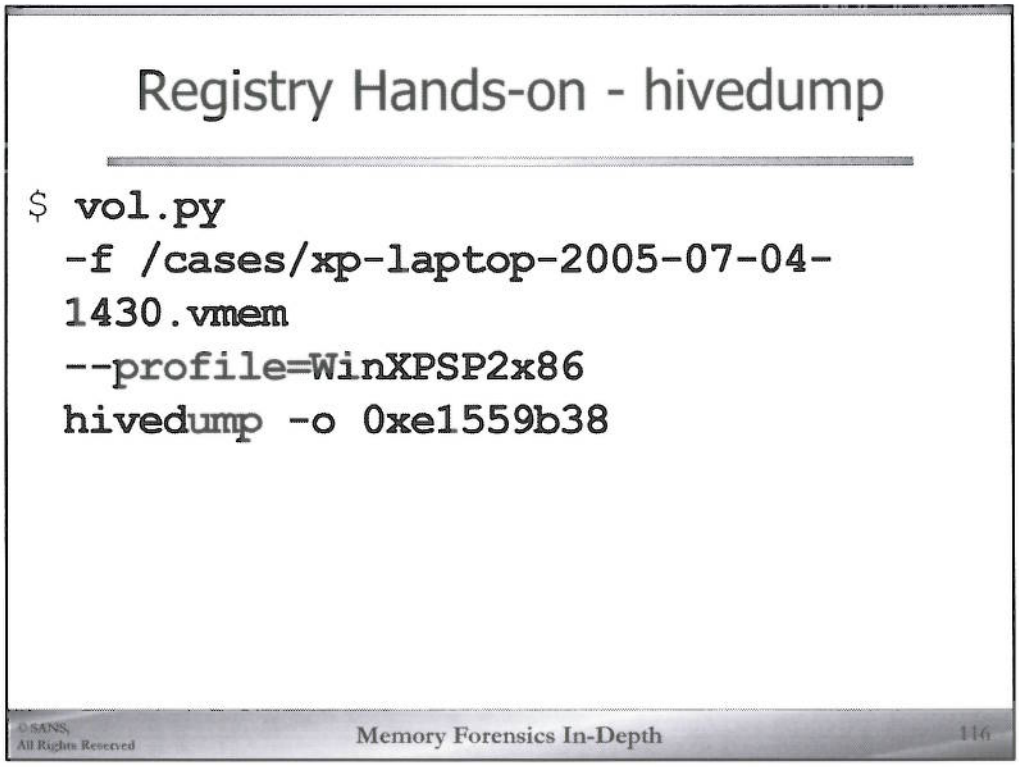
```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86 hivelist
```

Which should produce the following output:

```
Virtual            Physical         Name
0xe2610b60  0x14a99b60  \Device\HarddiskVolume1\Documents and
Settings\Sarah\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
0xe25f0578  0x17141578  \Device\HarddiskVolume1\Documents and
Settings\Sarah\NTUSER.DAT
0xe1d33008  0x0f12c008  \Device\HarddiskVolume1\Documents and
Settings\LocalService\Local Settings\Application
Data\Microsoft\Windows\UsrClass.dat
0xe1c73888  0x0efc5888  \Device\HarddiskVolume1\Documents and
Settings\LocalService\NTUSER.DAT
0xe1c04688  0x0e88e688  \Device\HarddiskVolume1\Documents and
Settings\NetworkService\Local Settings\Application
Data\Microsoft\Windows\UsrClass.dat
```

```
0xe1b70b60  0x0dff5b60  \Device\HarddiskVolume1\Documents and
Settings\NetworkService\NTUSER.DAT
0xe1658b60  0x0c748b60  \Device\HarddiskVolume1\WINDOWS\system32\config\software
0xe1a5a7e8  0x094bf7e8  \Device\HarddiskVolume1\WINDOWS\system32\config\default
0xe165cb60  0x0c6ecb60  \Device\HarddiskVolume1\WINDOWS\system32\config\SAM
0xe1a4f770  0x0948c770  \Device\HarddiskVolume1\WINDOWS\system32\config\SECURITY
0xe1559b38  0x02d64b38  [no name]
0xe1035b60  0x0283db60  \Device\HarddiskVolume1\WINDOWS\system32\config\system
0xe102e008  0x02837008  [no name]
0x8068d73c  0x0068d73c  [no name]
```

Here we can clearly see the virtual addresses and physical offsets for the various hives. Note that some of these hives are generated dynamically and do not have files on the disk.



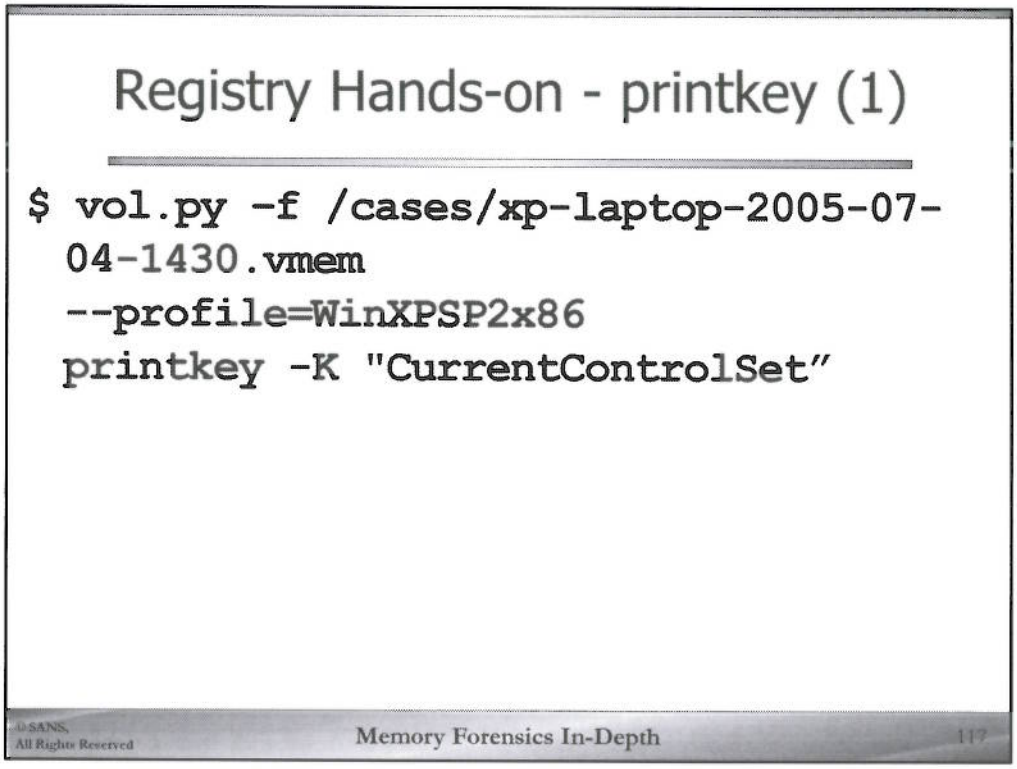
Let's look at one of those registry hives which doesn't have a filename. We'll use the hivedump plugin to do this. We'll use the -o flag to pass in the virtual address of the hive we want to dump, like this:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
hivedump -o 0xe1559b38
```

Which should generate output starting with:

```
Last Written         Key
2005-07-04 18:16:59  \HARDWARE
2005-07-04 18:16:59  \HARDWARE\ACPI
2005-07-04 18:16:59  \HARDWARE\ACPI\DSDT
2005-07-04 18:16:59  \HARDWARE\ACPI\DSDT\INT430
2005-07-04 18:16:59  \HARDWARE\ACPI\DSDT\INT430\SYSFexxx
2005-07-04 18:16:59  \HARDWARE\ACPI\DSDT\INT430\SYSFexxx\00001001
2005-07-04 18:16:59  \HARDWARE\ACPI\FACS
2005-07-04 18:16:59  \HARDWARE\ACPI\FADT
2005-07-04 18:16:59  \HARDWARE\ACPI\FADT\DELL__
2005-07-04 18:16:59  \HARDWARE\ACPI\FADT\DELL__\CPi_R__
2005-07-04 18:16:59  \HARDWARE\ACPI\FADT\DELL__\CPi_R__\27d30305
2005-07-04 18:16:59  \HARDWARE\ACPI\RSDT
2005-07-04 18:16:59  \HARDWARE\ACPI\RSDT\DELL__
2005-07-04 18:16:59  \HARDWARE\ACPI\RSDT\DELL__\CPi_R__
2005-07-04 18:16:59  \HARDWARE\ACPI\RSDT\DELL__\CPi_R__\27d30305
2005-07-04 18:16:59  \HARDWARE\DESCRIPTION
```

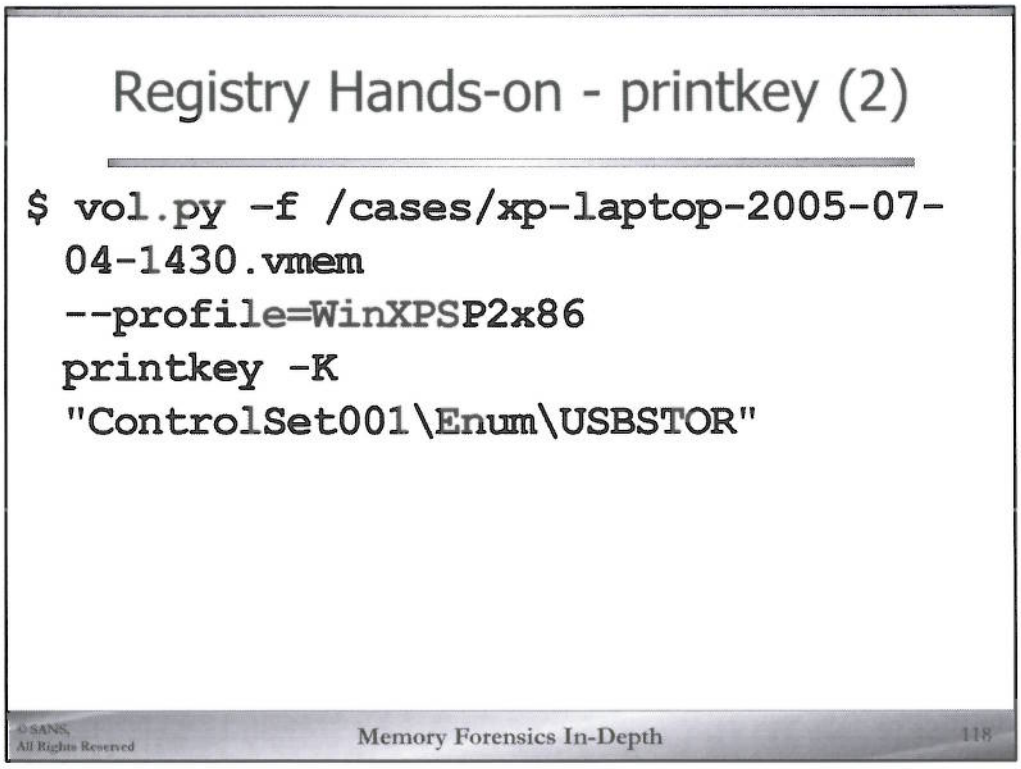
We've found the hardware hive!



We can use the printkey plugin to examine arbitrary keys and values in the registry. For example, let's take a look at the USB devices which were mounted as storage devices on the system. Information on such keys is kept the key System\CurrentControlSet\Enum\USBSTOR. Let's start digging.

First, let's take a look at the current control set. This is normally a symbolic link in the registry—it points to one of the existing control sets. We can use Volatility to tell us which one. Don't forget to enclose the key name in quotes. Although not necessary for this key name, it will be later on. Get in the habit now:

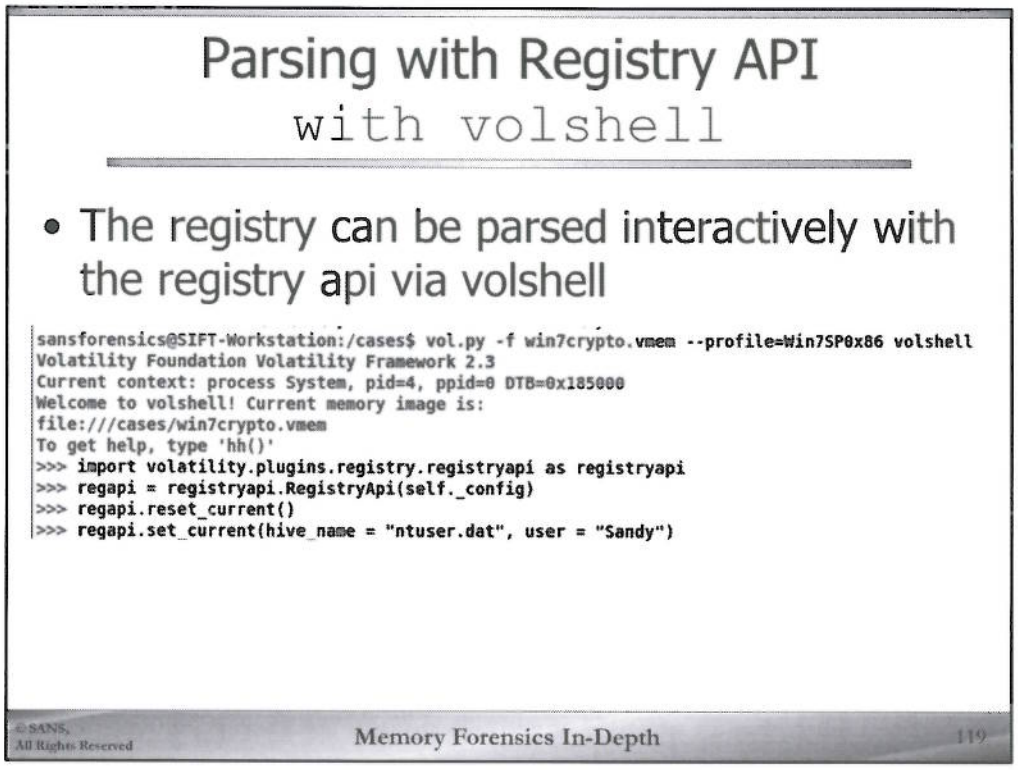
```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
printkey -K "CurrentControlSet"

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

----------------------------
Registry: \Device\HarddiskVolume1\WINDOWS\system32\config\system
Key name: CurrentControlSet (V)
Last updated: 2005-07-04 18:16:59

Subkeys:

Values:
REG_LINK      SymbolicLinkValue : (V)
\Registry\Machine\System\ControlSet001
```



We can see here that the Current Control Set points to ControlSet001. Ok, let's look for the USBSTOR key in that control set. We use the printkey plugin again, like this:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
printkey -K "ControlSet001\Enum\USBSTOR"

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

----------------------------
Registry: \Device\HarddiskVolume1\WINDOWS\system32\config\system
Key name: USBSTOR (S)
Last updated: 2005-06-25 16:50:48

Subkeys:
  (S) Disk&Ven_Generic&Prod_STORAGE_DEVICE&Rev_0.01
  (S) Disk&Ven_LEXAR&Prod_JUMPDRIVE_SPORT&Rev_1000

Values:
```

Here we can see the two drives which were connected to the system at some point! There appears to have been a generic USB storage device and a Lexar Jumpdrive Sport. We could even run printkey again on one of those keys and get even more details, including the device serial numbers.



In Section 2, you were introduced to volshell, the interactive python shell that allowed you to enumerate memory structures and populate them with memory image specific values based on virtual offsets. Volshell can also be used to parse the Windows registry hives from a memory image.

Run the following to import the registry api libraries into your instance of volshell:

$vol.py -f win7crypto.vmem --profile=Win7SP0x86 volshell

Volatility Foundation Volatility Framework 2.3

Current context: process System, pid=4, ppid=0 DTB=0x185000

Welcome to volshell! Current memory image is:

file:///cases/win7crypto.vmem

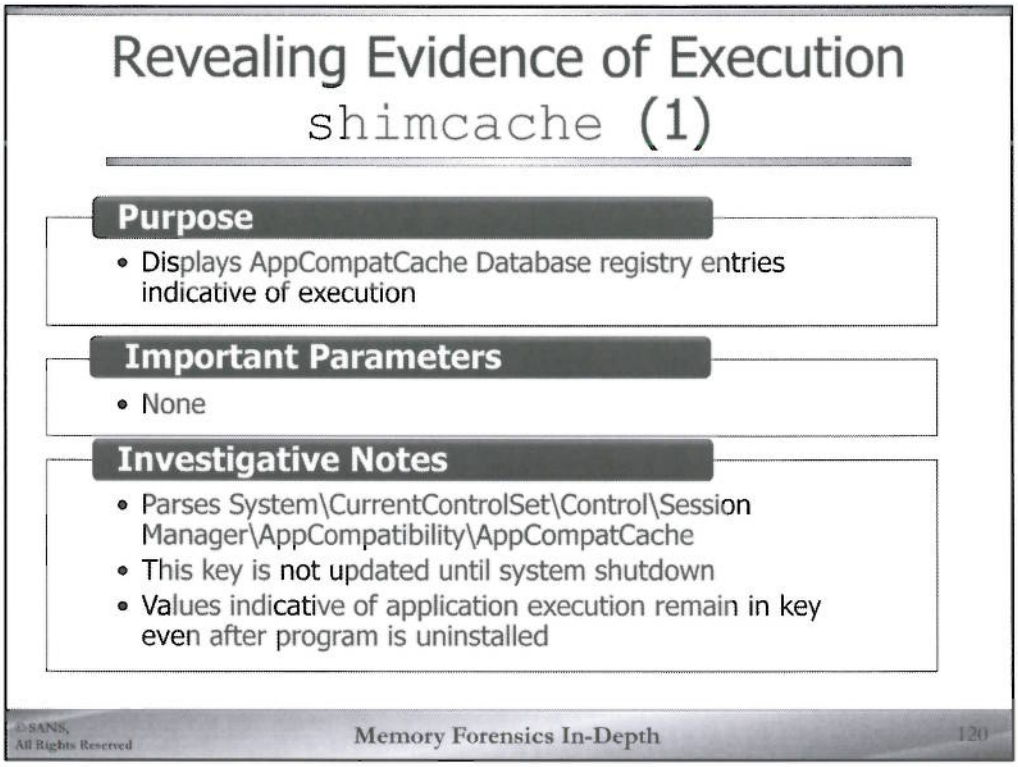
To get help, type 'hh()'

>>> import volatility.plugins.registry.registryapi as registryapi
>>> regapi = registryapi.RegistryApi(self._config)

>>> regapi.reset_current()

>>> regapi.set_current(hive_name = "ntuser.dat", user = "Sandy")

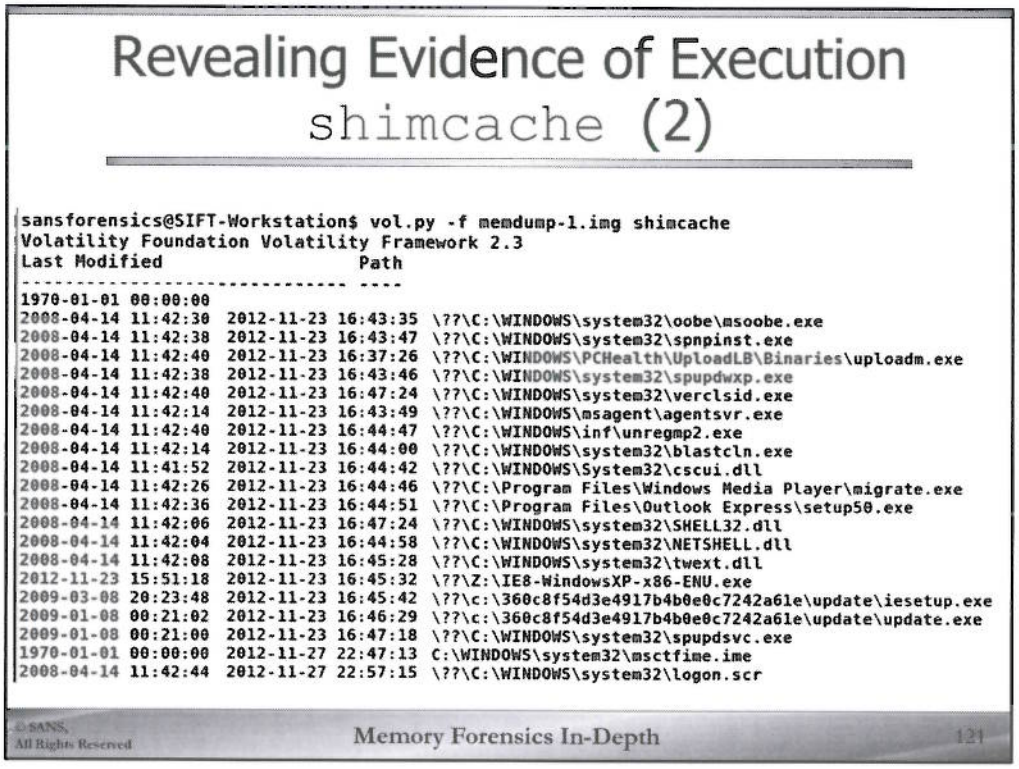
>>> >>> for keypath in regapi.reg_enum_key("ntuser.dat", "Software\\Microsoft\\Windows\\Explorer"):
... print keypath



When an application is executed, Windows records the execution path and file name in the Windows registry under the AppCompatCache key (HLKM\SYSTEM\CurrentControlSet\Control\Session Manager\AppCompatibility\AppCompatCache) which tracks the compatibility needs of an application. For example, the AppCompatCache would record if a program required a "shim" and needed to be run in XP compatibility mode on a Windows 7 system. Much research has been done on this key and how it is populated over the last few years and largely points to the values for each application being created upon first execution on the system.[1] This key is updated upon system shutdown and new entries are maintained in cache until then. Therefore if you are investigating a system that has required uptime (and cannot be rebooted), you should expect to not see the most recent new program execution entries in the **shimcache** plugin output.

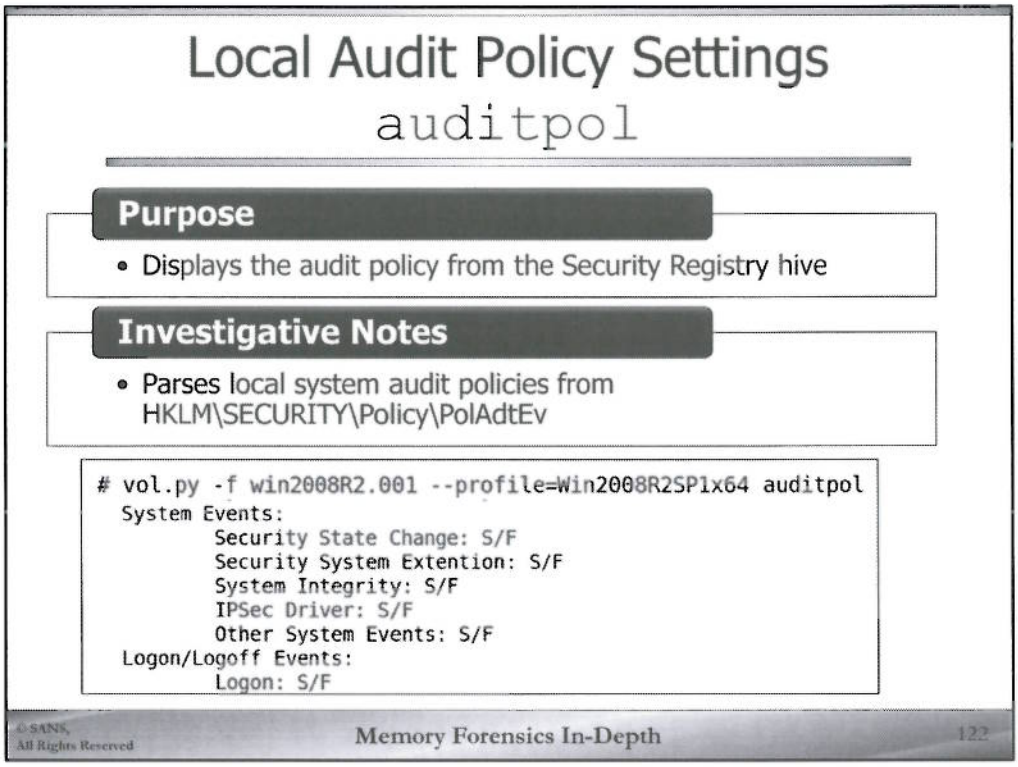
This key has useful application in both malware investigations and user investigations due to the fact that few, if any, privacy cleaners remove these values, nor does uninstalling a program remove evidence of its execution from this registry key.

[1] https://dl.mandiant.com/EE/library/Whitepaper_ShimCacheParser.pdf

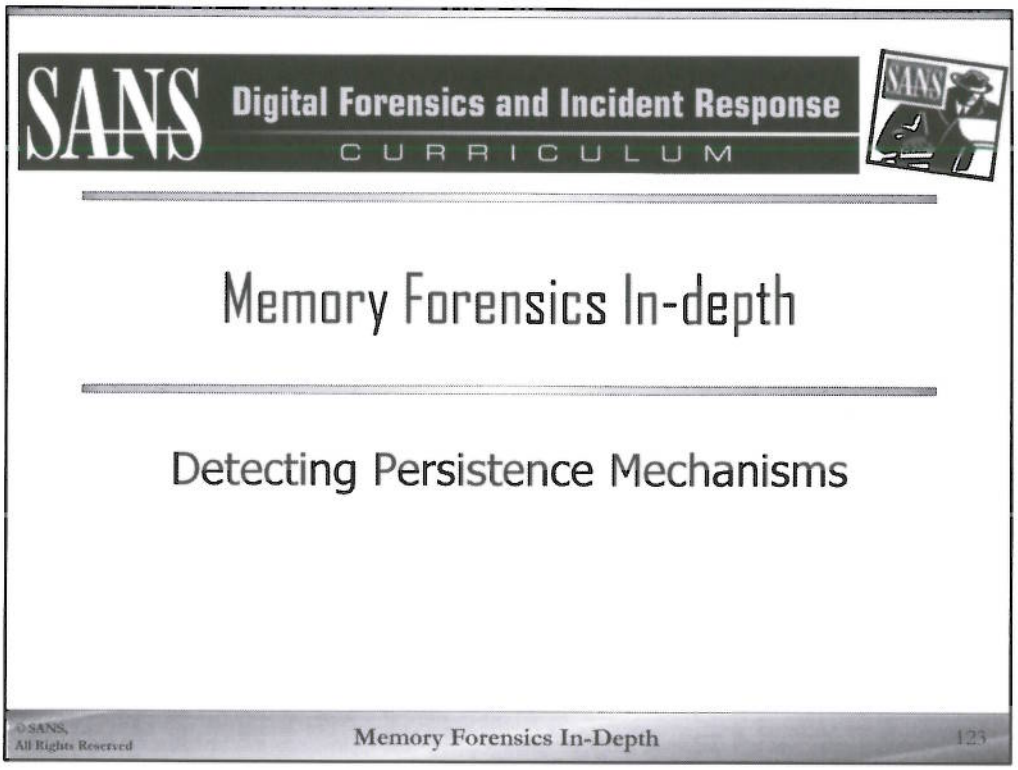


AppCompatCache exhibits differing behaviors based on Windows version. Above is shimcache output from an XP system, therefore there are three columns. The output from the **shimcache** plugin displays the following information: the first column is the last modified date/time for the file, the second column is the last modified date/timestamp for the Application Compatibility Cache entry (indicative of when the program was either first run or entry modified) and the third column provides the full path of the executable.

When running **shimcache** against a Windows 2003, Vista, 2008 or Win7 image, the output for shimcache just displays last modified date/timestamp of the entry and file path.



Using the auditpol plugin from Volatility, an examiner can enumerate the local audit policy in effect at the time the system memory was acquired. Prior to moving to volume analysis of the target system, it helps to understand what was being logged, therefore what the eventlogs will contain.



This page intentionally left blank.

Persistence Mechanisms

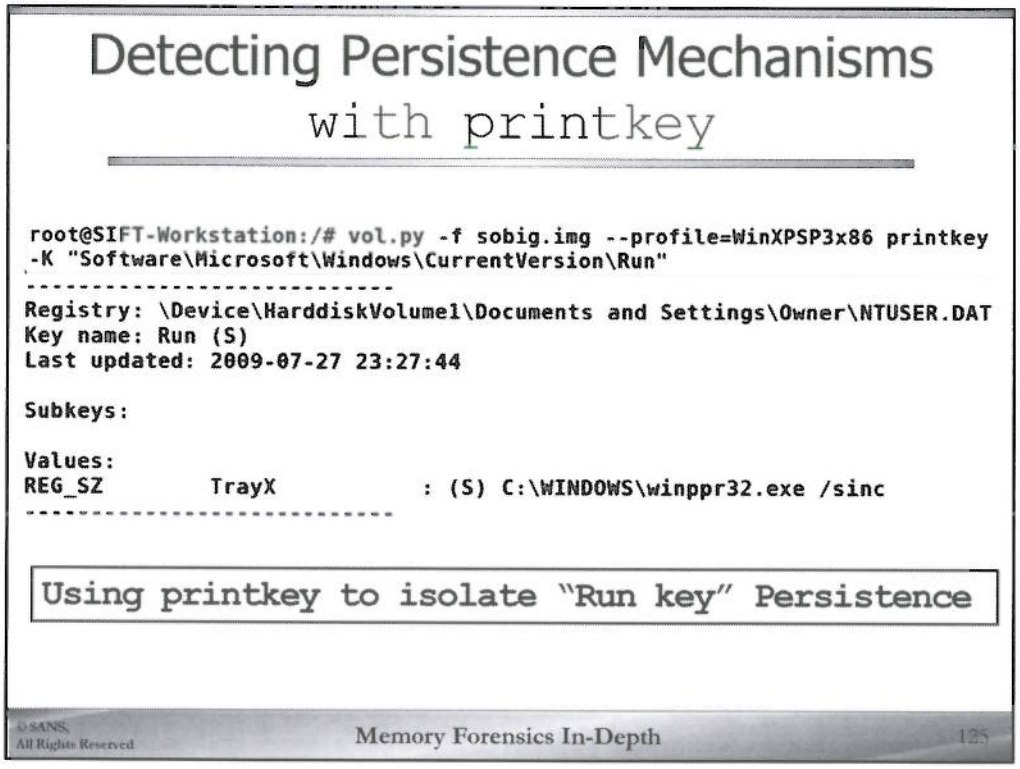
The Malware Predicament

- Persistence allows malware the ability to "survive" reboot
- Yet, there are trade-offs when the adversary employs persistence techniques
- Common persistence techniques include:
  - AutoRun keys
  - Creation/Hijacking Services
  - Scheduled Tasks
  - DLL Search Order Hijacking

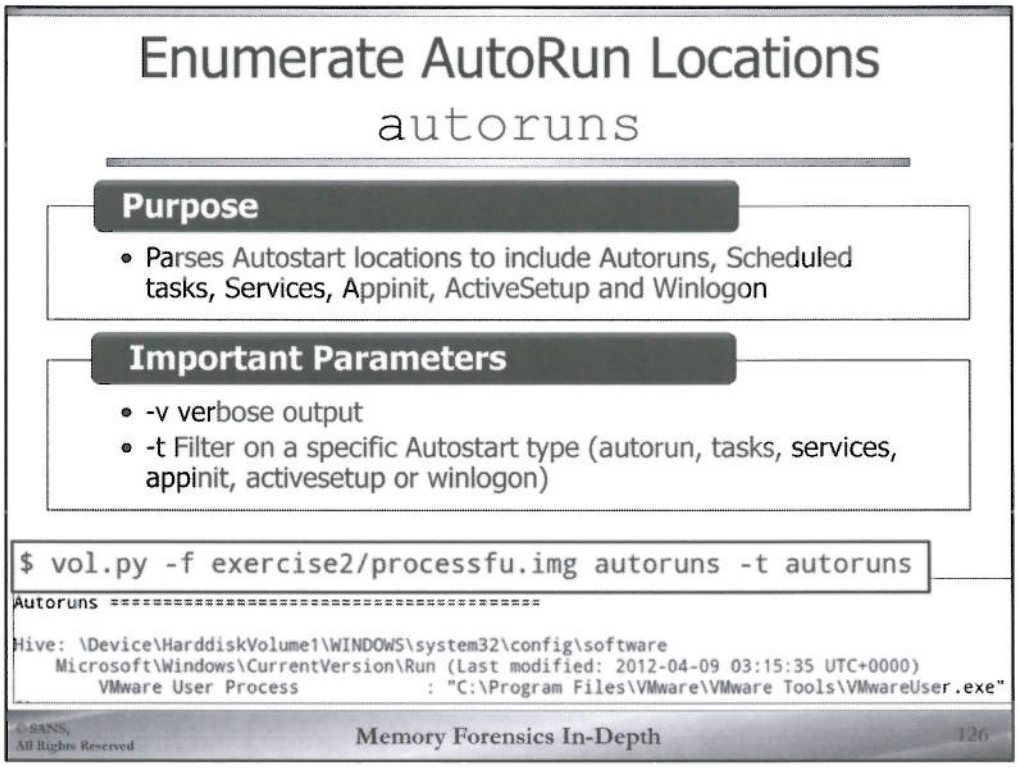
"if I persist, they will know I was here…"

In order for malware to survive a reboot and live to "beacon another day", it must gain persistence on the target system. For memory-only malware, instantiating a persistence mechanism is the first step to one day being detected by host-based forensics. So there is a trade-off between survivability and obscurity. Nonetheless, despite many of the persistence mechanisms employed by malware being quite well known, they are still commonly employed by today's malware variants. According to Mandiant's M-Trends 2010, 76% of APT malware samples used service persistence, 21% used Registry Run keys and 3% made use of other techniques. [1] It would seem that if more investigators knew the most common place to look for persistence mechanisms, we could add this to our 6 step investigative methodology as a tried and true method for spotting evil. The common use of the registry for triggering malware execution makes it a fantastic place for investigators to look while examining a system for malicious code infection.

[1] Mandiant M-Trends 2010: The Advanced Persistent Threat. https://www.mandiant.com/resources/mandiant-reports/



Malware frequently uses the Autorun keys in the Windows registry in order to achieve persistence and guarantee execution upon boot up or user logon. An investigator can enumerate the commonly used location inside the registry by using the **printkey** plugin. The SysInternals Autoruns tool is one of the best resources for a comprehensive list of locations in the registry that have been seen being used by malware for persistence.



The **Autoruns** plugin, written by Thomas Chopitea, parses the following locations in the Windows Registry and the file system to identify common locations for persistence mechanisms:

Software hive

Microsoft\Windows\CurrentVersion\Run, RunOnce

Microsoft\Windows\CurrentVersion\RunServices,

Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,

Wow6432Node\Microsoft\Windows\CurrentVersion\Run, RunOnce

Wow6432Node\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,

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

NTUSER.DAT hives

Software\Microsoft\Windows\CurrentVersion\Run, RunOnce

,Software\Microsoft\Windows\CurrentVersion\RunServices,

Software\Microsoft\Windows\CurrentVersion\RunServicesOnce,

Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,

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

Software\Microsoft\Windows NT\CurrentVersion\Run,

Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run,

Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run

Winlogon & AppInit

Microsoft\Windows NT\CurrentVersion\Winlogon (value AppInit_DLLs)

Microsoft\Windows NT\CurrentVersion\Winlogon\Notify

Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit

Microsoft\Windows NT\CurrentVersion\Winlogon\VmApplet

Microsoft\Windows NT\CurrentVersion\Winlogon\Shell

Microsoft\Windows NT\CurrentVersion\Winlogon\TaskMan

Microsoft\Windows NT\CurrentVersion\Winlogon\System

Services

CurrentControlSet\Services

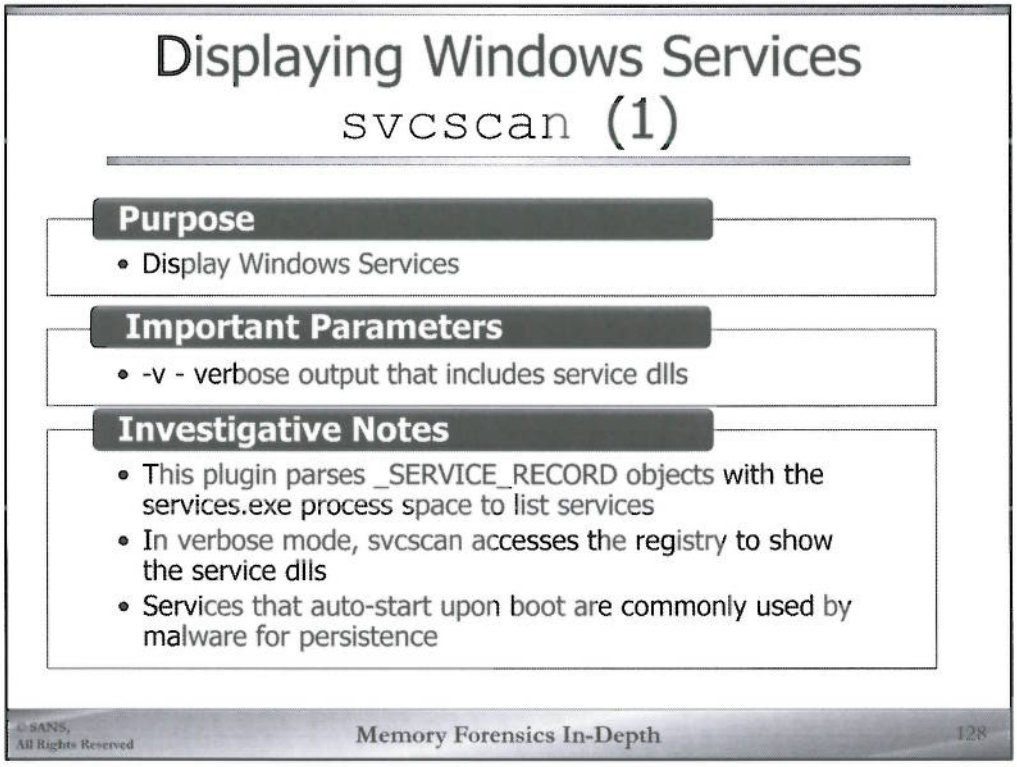
Scheduled Tasks

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

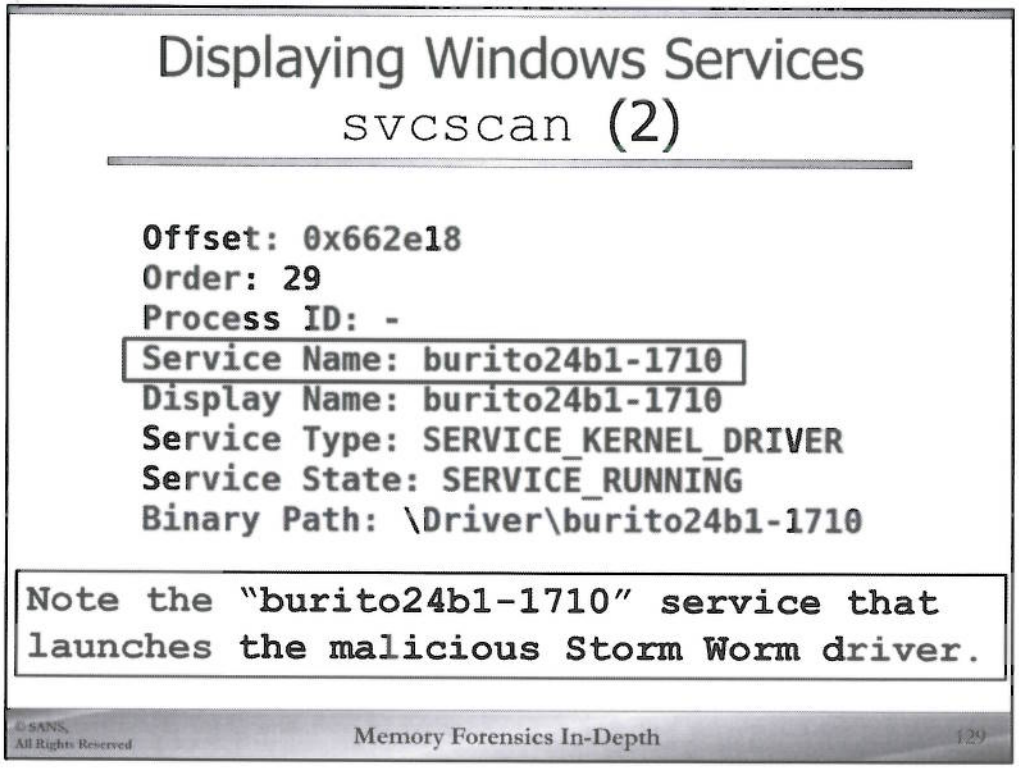
Active Setup

Microsoft\Active Setup\Installed Components

For more information on the Autoruns plugin (fourth place winner of the 2014 Volatility plugin contest), see https://github.com/tomchop/volatility-autoruns



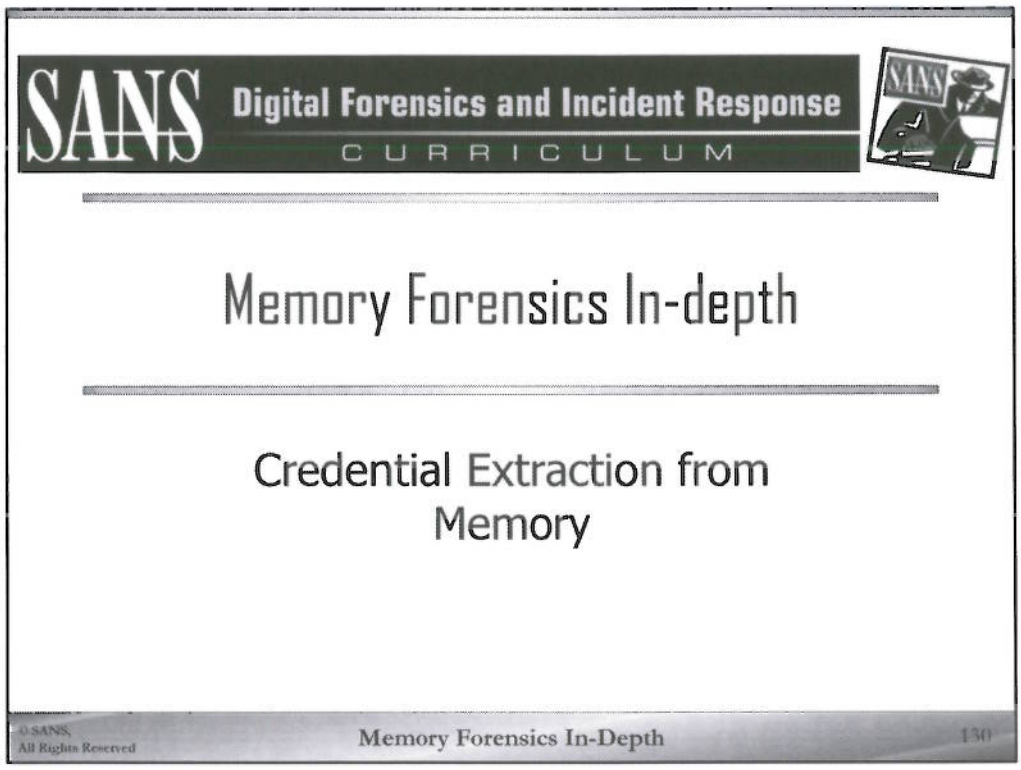
Another common way malware achieves persistence is through the creation or hijacking of Windows services. The Volatility plugin, **svcscan**, identifies services by walking the process list to find services.exe. Within the process memory of services.exe, svcscan scans for _SERVICE_RECORD objects that hold the values for the services themselves. Though svcscan is commonly thought to be purely a registry scanner, it only accesses the registry in verbose mode in order to extract the service dlls.



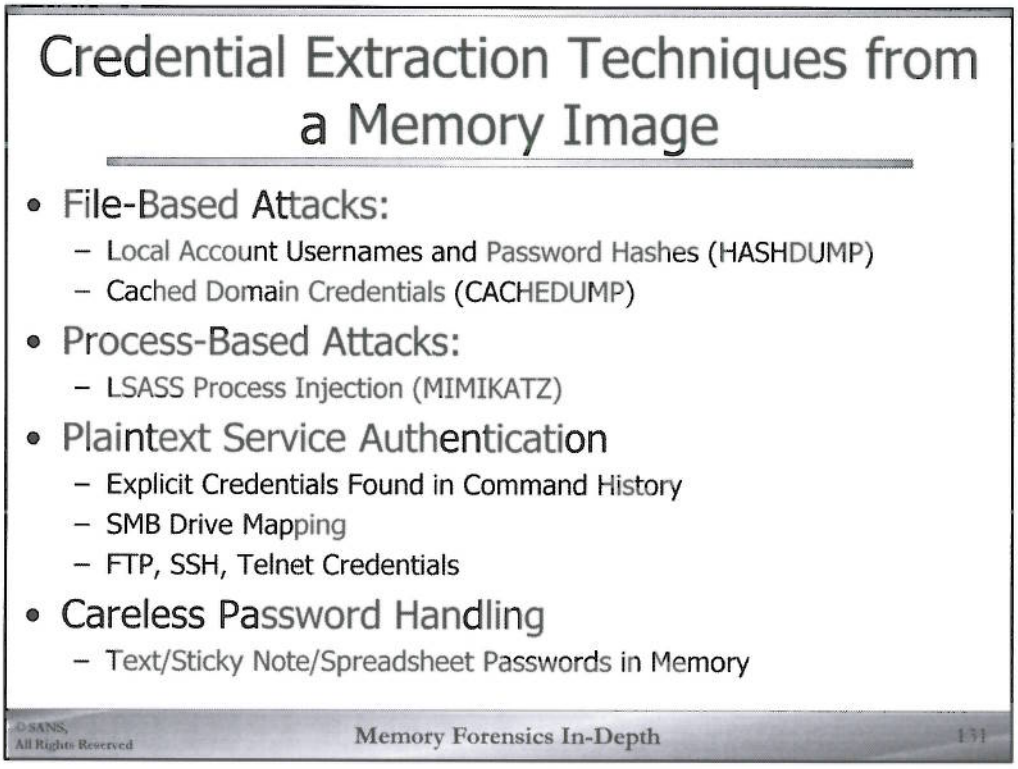
Volatility offers an efficient way of parsing the System hive to enumerate the Services key, representing Windows Services that have been created on the target system. Services can be found as individual subkeys under the following registry key: "SYSTEM\CurrentControlSet\Service". Malware frequently will instantiate a new service or hijack an existing Windows services in order to achieve persistence. If a Windows service has a start value of 2, it is set to "auto load" on start up.[1] In the instance of a hijacked service, the malicious dll or driver that is being called will typically gain execution and then call the legitimate service dll so that the service does not appear to be subverted.

In looking at the output, it is notable that there is no binary path for services that are not running.

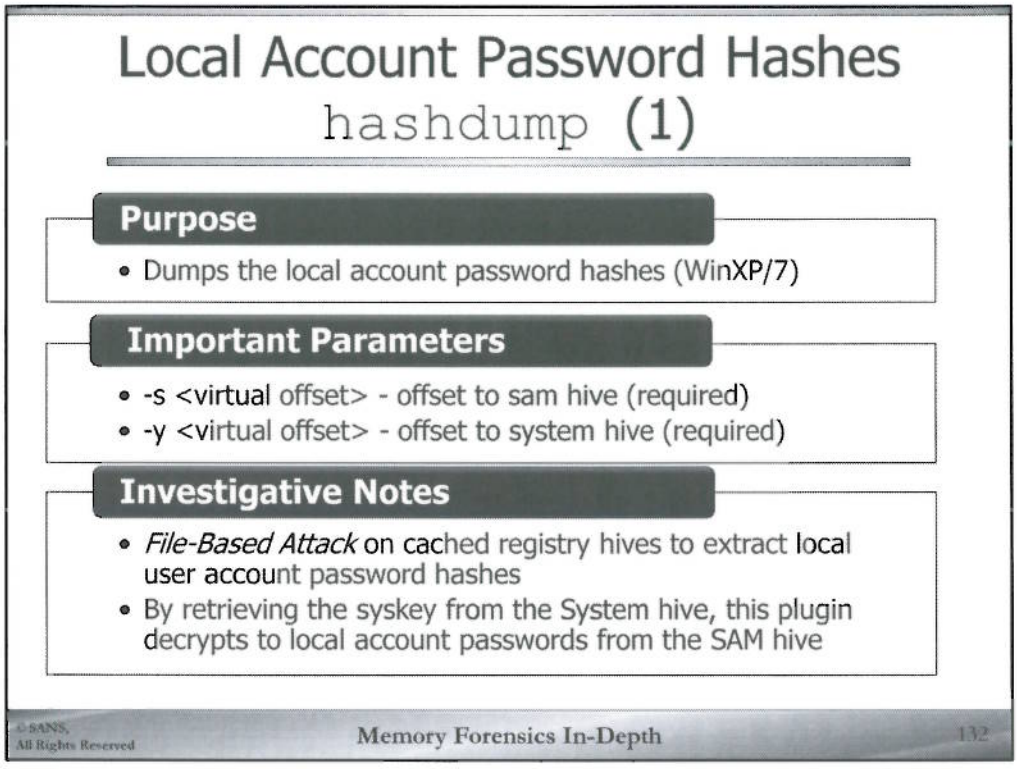
[1] "CurrentControlSet\Services Subkey Entries" http://support.microsoft.com/kb/103000



This page intentionally left blank.



There are multiple ways to extract user credentials from a system memory image, many that mimic tools that were first used for offensive post-exploitation methods. We will cover file-based and process-based credential attacks as well as harvesting username and passwords from network packets and memory mapped files.

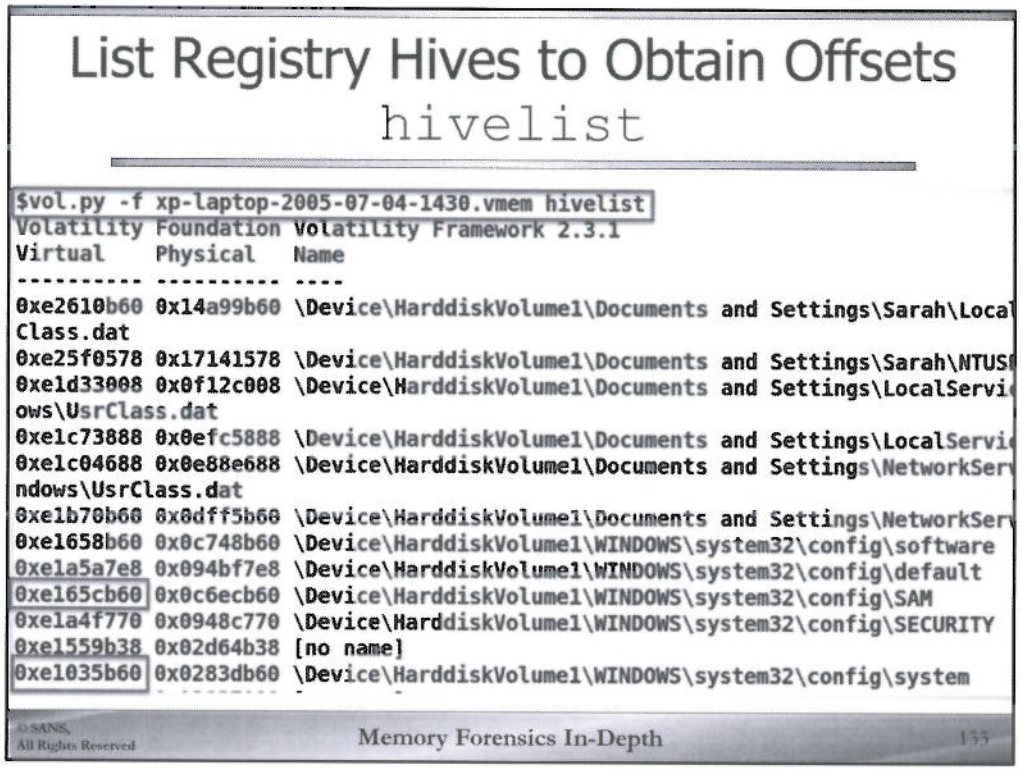


Why should pen testers (and attackers) have all of the fun?

Using the Volatility plugin, hashdump, on versions of Windows prior to Win8, investigators can dump the local account password hashes from a memory image. Similar to how the post-exploitation module works in Metasploit, hashdump uses the syskey from the System hive to decrypt the password hashes from the SAM hive.

Why would investigators need to have access to passwords?

When conducting an intrusion investigation, if a system has been deemed "owned" or compromised by an attacker, it should be assumed that the credentials have been harvested from the box. Identifying these compromised credentials involves extracting them yourself (or at least enumerating them in some manner). Though hashdump only provides local account passwords and no visibility into the cached domain credentials, there is still value to investigators in gaining access to them. If passwords are not in accordance with security best practices or you are able to ascertain that all local administrator passwords across multiple systems are the same, you have some recommendations that you can provide to the customer. In addition, in user investigations, any password that the target used (even for a local account) has a good chance of being reused in other parts of the investigation.



In order to run hashdump successfully, we need to first obtain the virtual address offsets for the System and Sam hives. Therefore, the first step in running hashdump is to run hivelist and obtain these offsets. This is shown in the slide above as a reminder that it is first column in the hivelist output that contains the virtual offsets.

Here are the relevant lines from hivelist:

```
Virtual                  Physical                 Name
0xe165cb60   0x0c6ecb60  \Device\HarddiskVolume1\WINDOWS\system32\config\SAM
0xe1035b60   0x0283db60
\Device\HarddiskVolume1\WINDOWS\system32\config\system
```

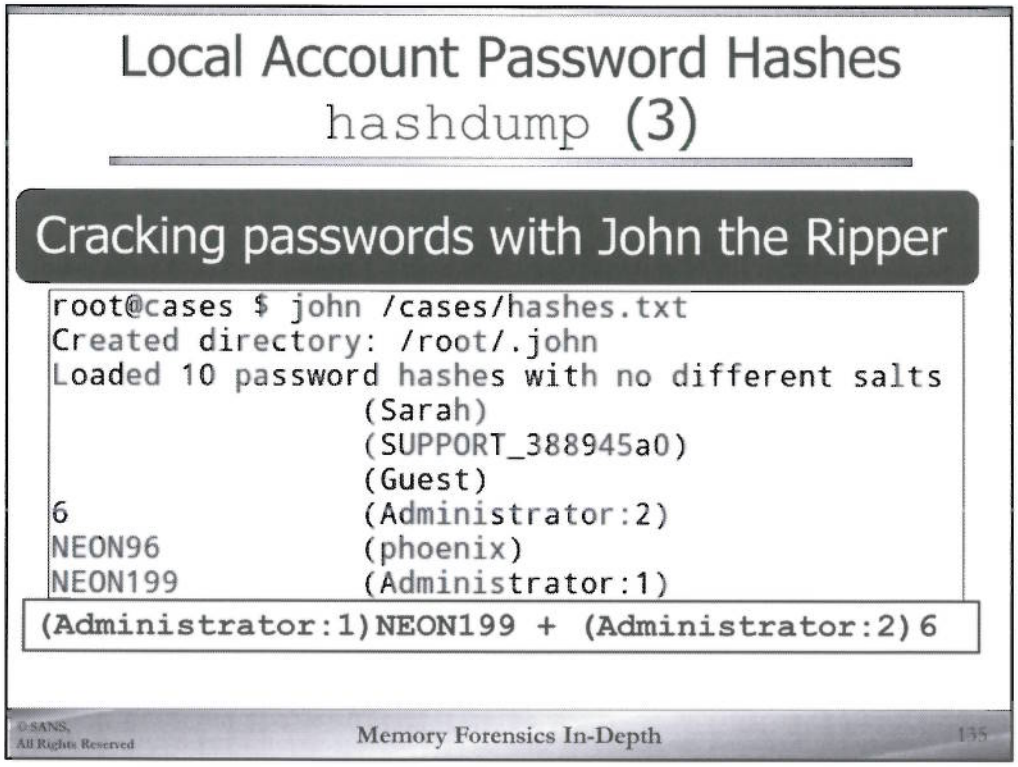
And thus the command line to dump the hashes will be:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
hashdump -y 0xe1035b60 -s 0xe165cb60
```

Which gives the following output:

```
Administrator:500:08f3a52bdd35f179c81667e9d738c5d9:ed88cccbc08d1c18bcded317112555
f4:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:ddd4c9c883a8ecb2078f88d729ba2e67:e78d693bc40f92a534197dc1d3a6d
34f:::
SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:8bfd47482583168a0ae5ab020e
1186a9:::
phoenix:1003:07b8418e83fad948aad3b435b51404ee:53905140b80b6d8cbe1ab5953f7c1c51:::
ASPNET:1004:2b5f618079400df84f9346ce3e830467:aef73a8bb65a0f01d9470fadc55a411c:::
Sarah:1006:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
```

The **hashdump** plugin output requires some interpretation. The image above explains each of the fields, separated by colons. The first field is the local account username, and second field is the RID (Relative Identifier) for that account. Immediately following the RID is the LM hash, if stored locally and the NT hash, separated from each other with a colon. In looking at many LM and NT hashes, you come to recognize the hashes that indicate null passwords. The LM and NT hash of two accounts shown above, Guest and Sarah, are examples of accounts that have blank passwords.



We can immediately see there are two non-trivial accounts on the system, phoenix and Sarah. We now have the LM and NTLM hashes for each account and can attempt to crack them.

There are lots of ways to crack LM and NTLM hashes. Your SIFT workstation also has a program called John the Ripper which can be used to crack passwords. It's slow, using brute force to find matches for the password hashes. You'll need the output of the hashdump command we ran on the previous page saved to a text file. Then run John the Ripper on that file:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
hashdump -y 0xe1035b60 -s 0xe165cb60 > hashes.txt
$ john hashes.txt
```

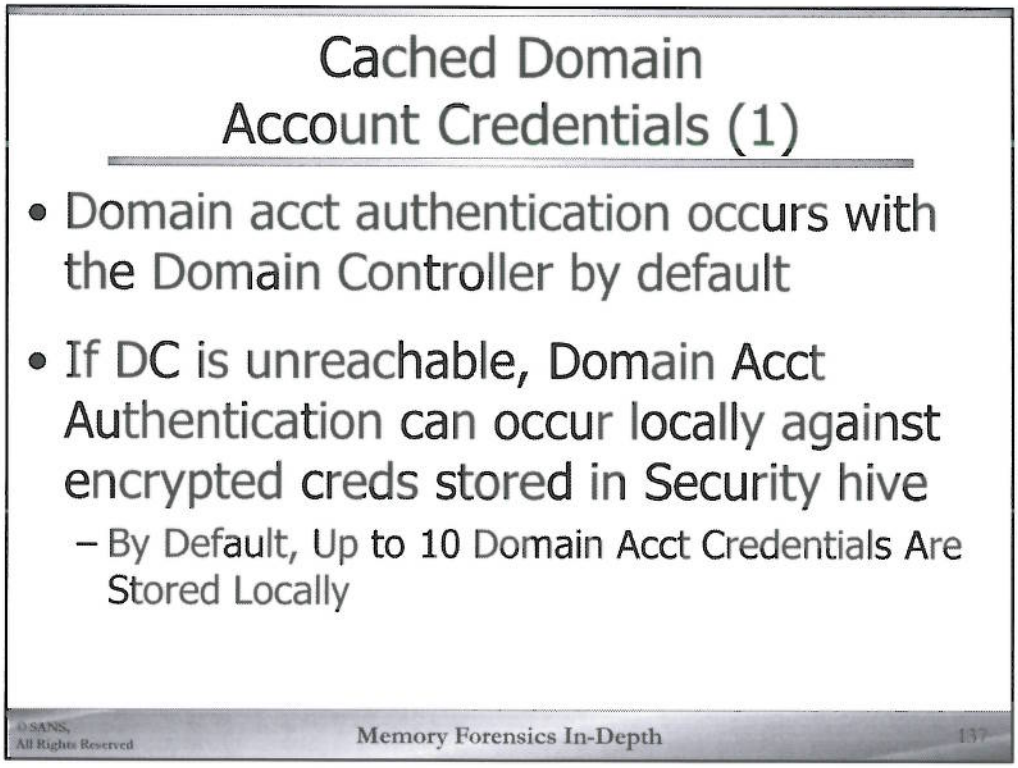
It's slow, but you will see the passwords eventually appear. Remember, the LM hashes are broken into two halves which John is brute forcing separately. You will see results for each half with (Username:half). In this case, (Administrator:1) would be the first half of the Administrator password. Extra credit: What is the password to the HelpAssistant account?

In addition to local cracking programs, you can use an online site, such at http://crackstation.net/ or http://www.insidepro.com/hashes.php?lang=eng. We'll use the former in our example.

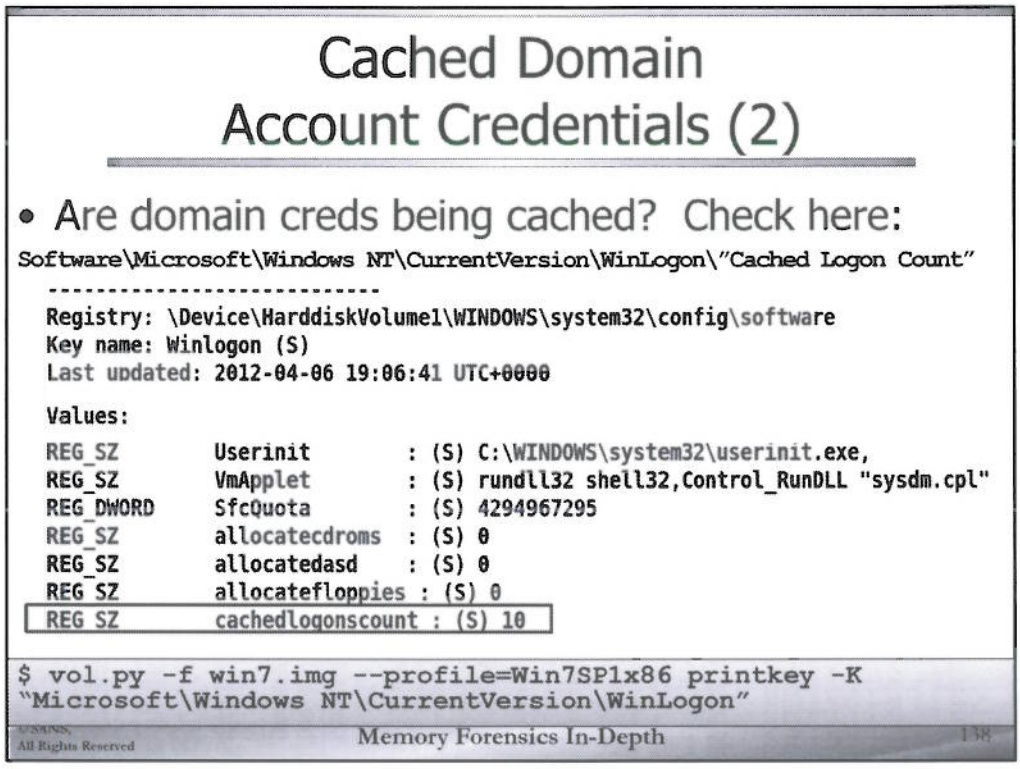
Go to https://crackstation.net/ and submit the LM hash for Sarah, aad3b435b51404eeaad3b435b51404ee. The site should tell you that it's the empty-password hash. The user Sarah had no password. In general, these sites are hit or miss. Either they can tell you the password immediately, or not at all.

Then submit the LM hash for the phoenix account, 07b8418e83fad948aad3b435b51404ee. Hopefully the site should tell you that the password was Neon96. Victory! See how easy it is to crack Windows passwords?

NOTE: Some websites will block the classroom's IP address if too many students visit the site simultaneously. Don't be surprised if you see such an error message.

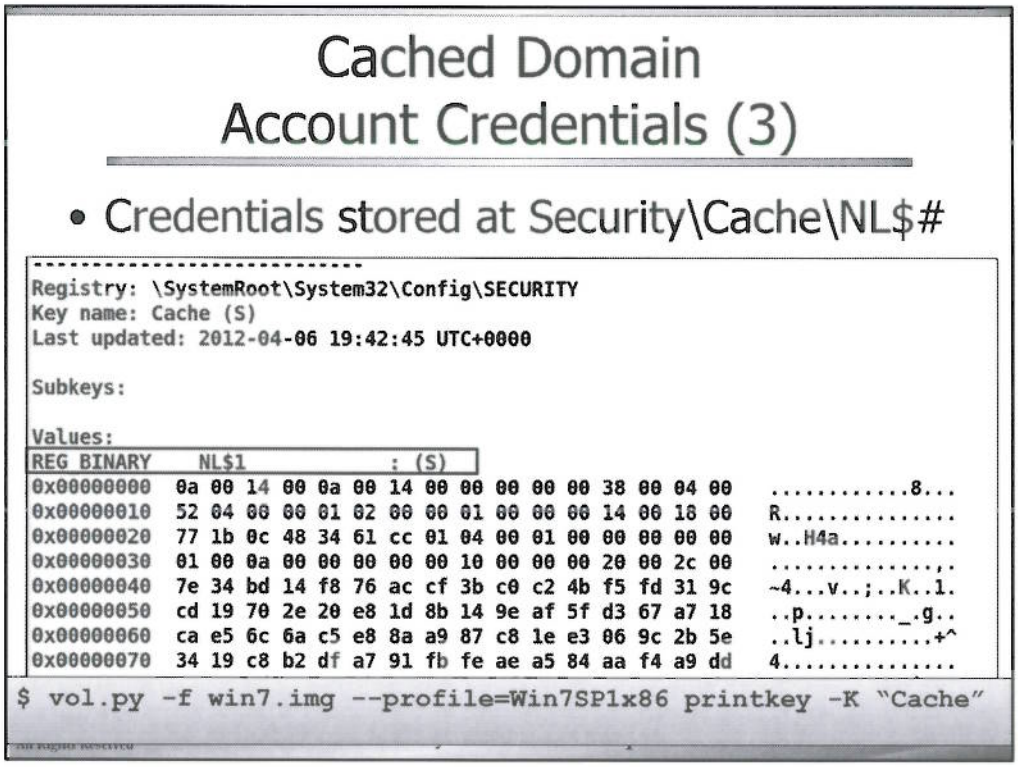


Many security professionals are surprised to find out that, although domain account authentication occurs on the Domain Controller by default, local authentication for domain account is possible under certain conditions. If the DC is unreachable, domain accounts can be authenticated can occur locally against credentials stored in the Security registry hive. By default, the most recent ten domain account credentials are stored in an encrypted form locally.

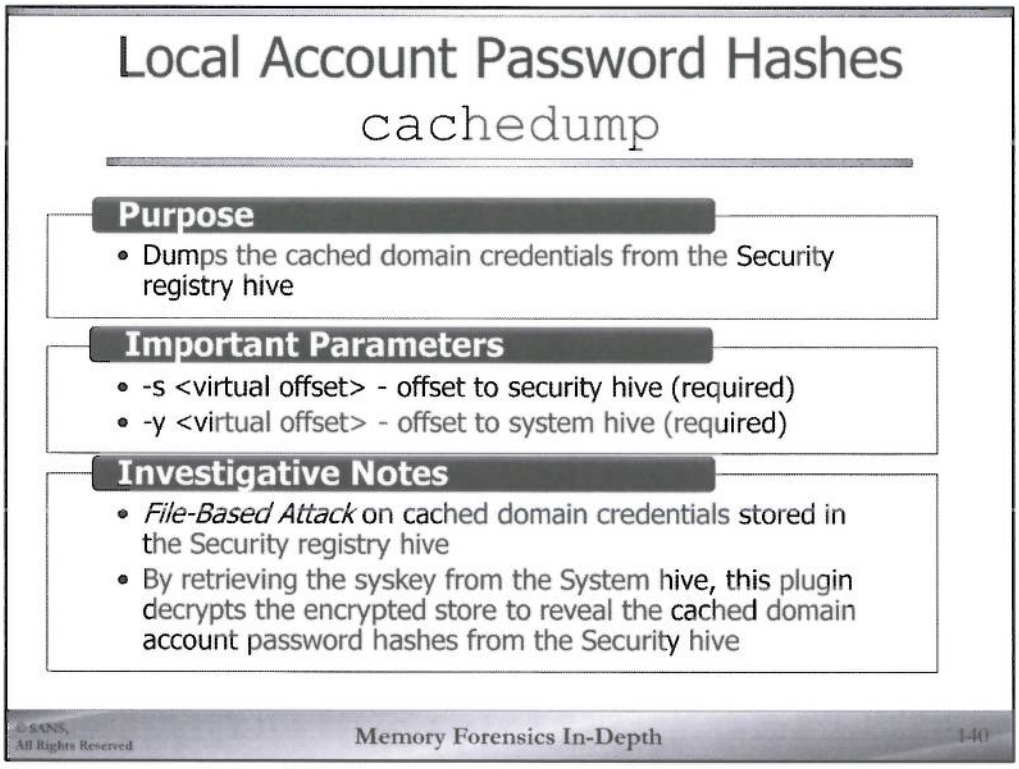


Whether systems in an enterprise are caching domain credentials can be configured via group policy. In order to determine from a memory image if the target system is configured to cache domain credentials, the examiner may use the printkey plugin to extract the value located at the following registry path:

"Software\Microsoft\Windows NT\CurrentVersion\WinLogon\Cached Logon Count"



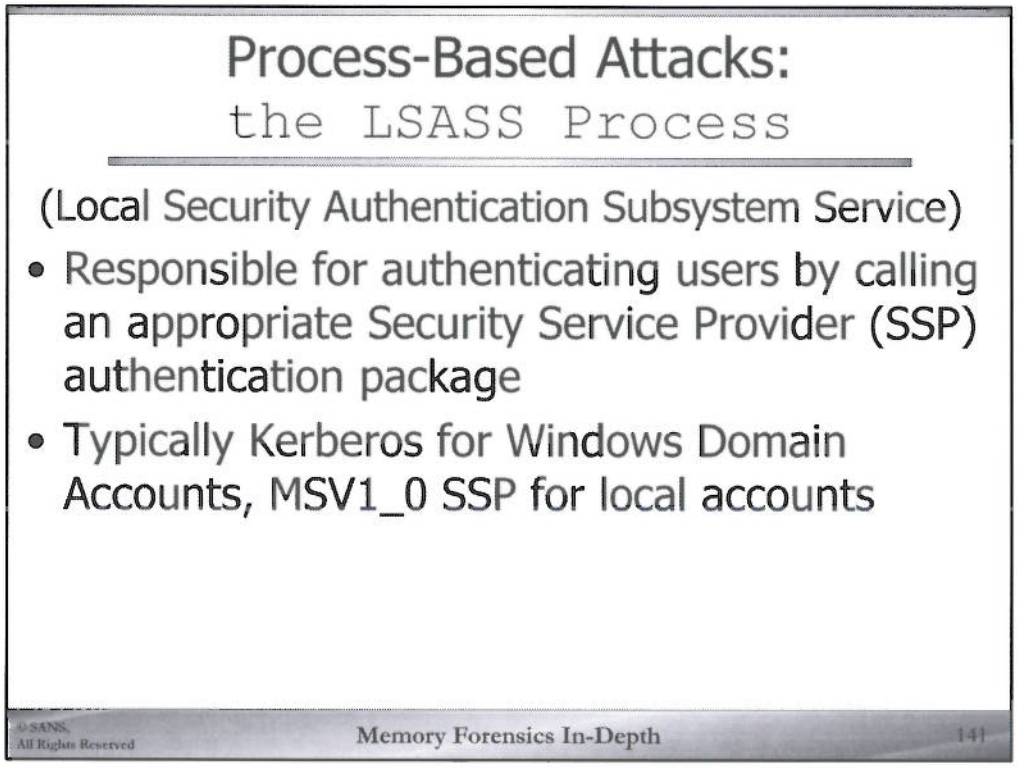
If they are being stored locally, these cached domain credentials are held in the Security registry hive. Shown above are the encrypted credentials for the most recently used domain account to login to the system.



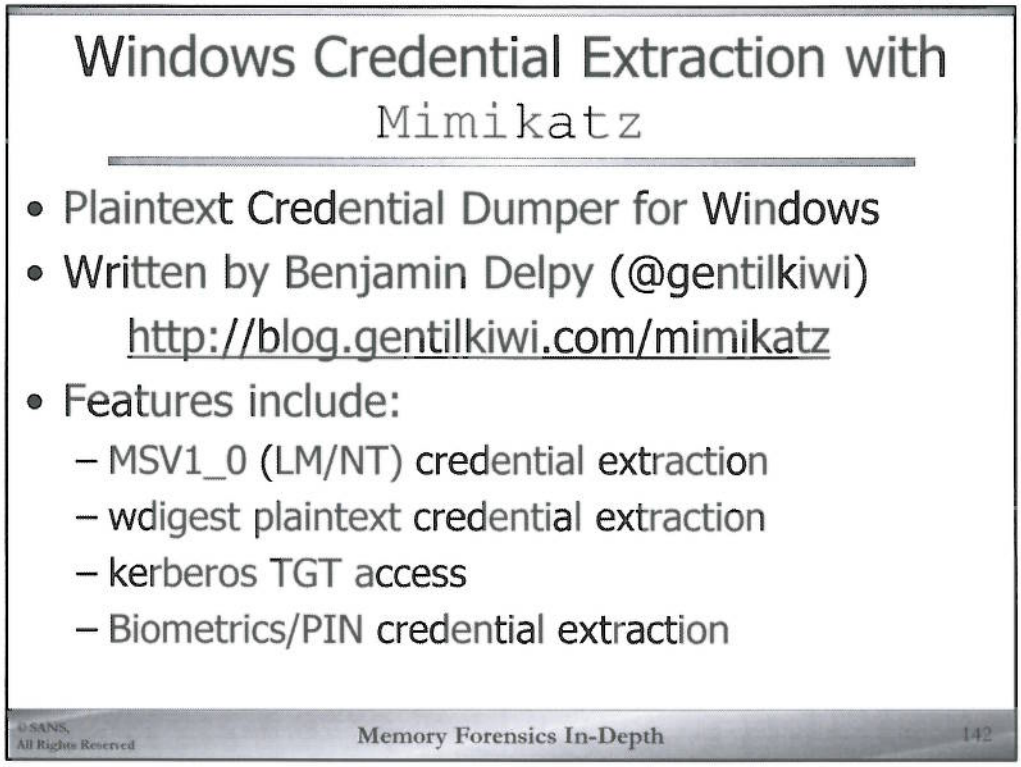
The cachedump plugin, developed by Brendan Dolan Gavitt, was included in an older version of Volatility and added back in Vol2.4. The required parameters include the virtual offsets for the System and the Security registry hives. For additional information on the extraction these hashes, Read more here: http://moyix.blogspot.com/2008/02/cached-domain-credentials.html

Output from this plugin is in the following format:

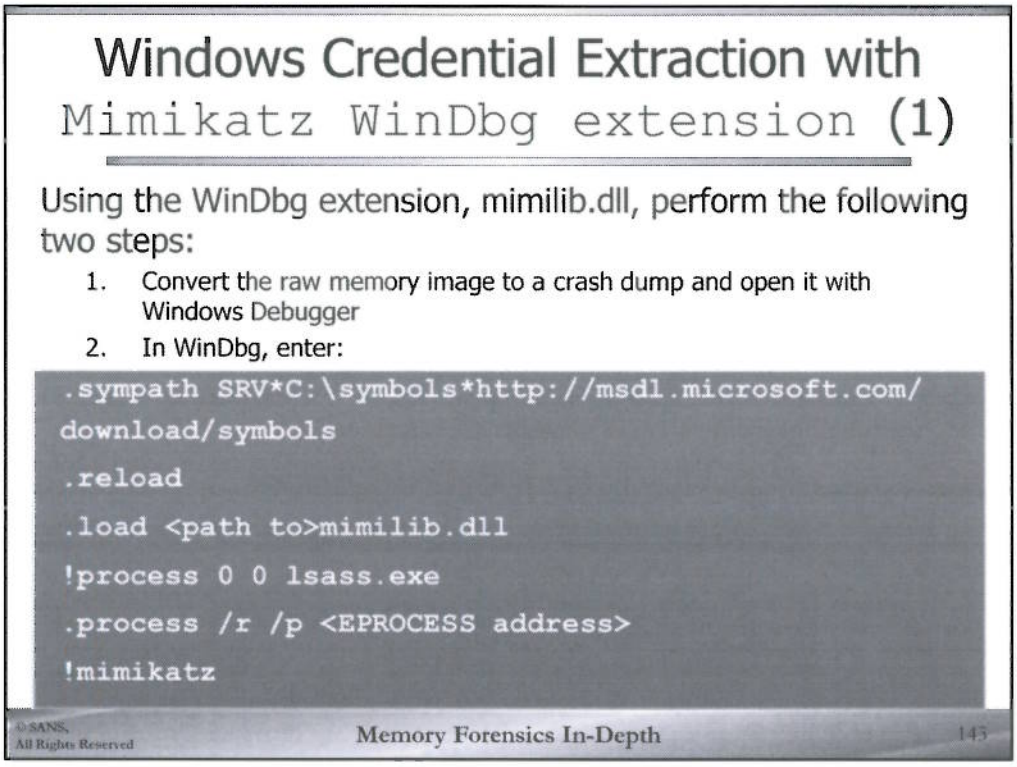
```
Domain Account: Password Hash: Domain
```



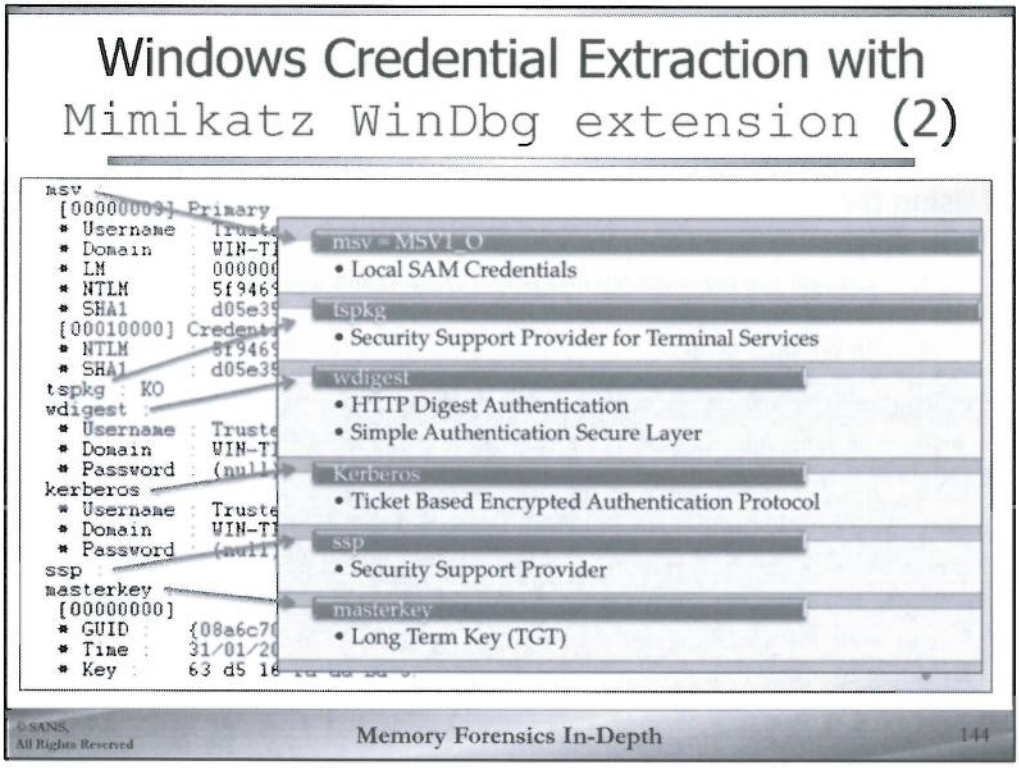
Although we are parsing a memory image as we perform file-based credential harvesting, we have process address space available as well. It is the access to the lsass process address space which allows us to conduct our next technique for extracting credentials - this time resulting in plaintext password extraction. The lsass (local security authentication subsystem service) is responsible for authentication of user credentials either locally or across the network on the domain controller and actually maintains the active user's password in its process address space in plaintext.



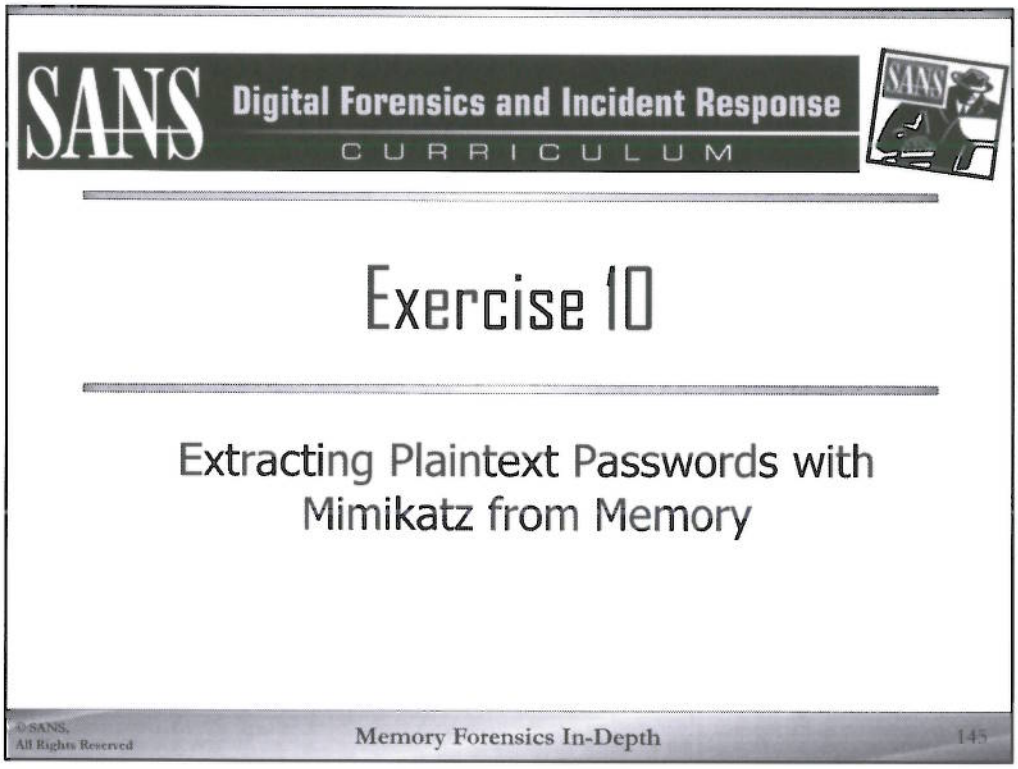
Mimikatz is a plaintext credential harvester, written by Benjamin Delpy, that is designed to run on a live system. It injects into the lsass process and extracts the active session credentials stored in the process address space. A Metasploit post-exploitation module has been developed to include mimikatz functionality, allowing easy access to remote system credentials for pentesters. In the summer of 2014, Delpy created a Windows Debugger extension that allows this functionality to run against a memory image of a system.



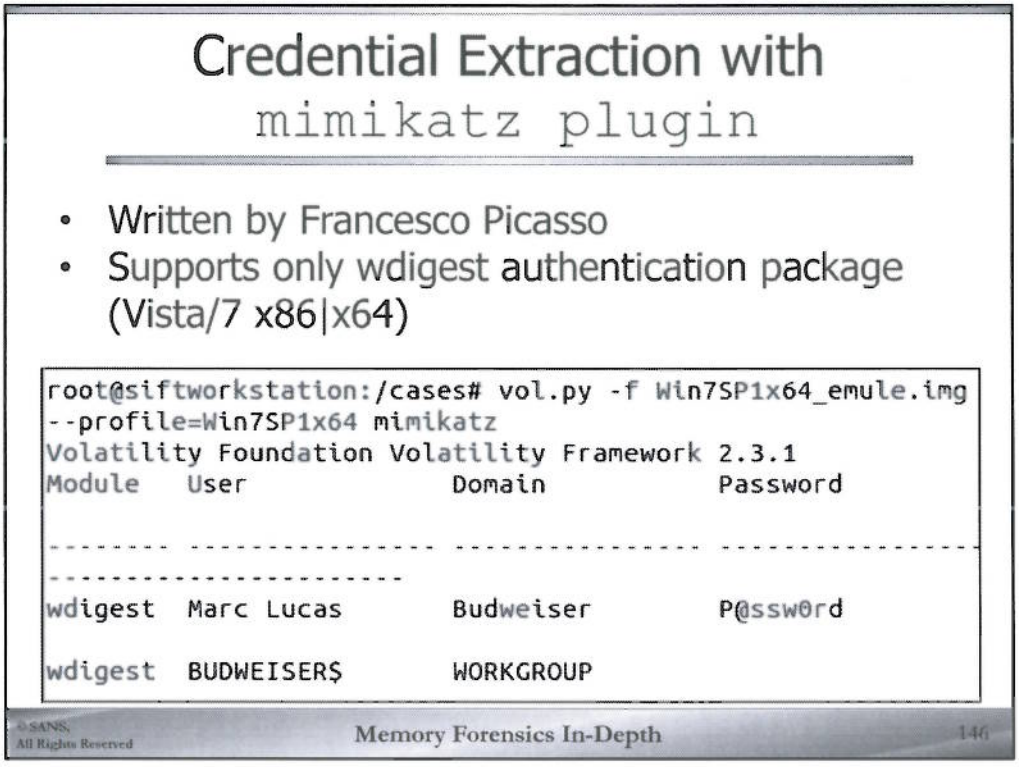
The mimikatz WinDbg extension is located in the FOR526 Win8.1 VM in the c:\tools directory. Upon loading the correct driver, x86 or x64, into WinDbg, and making current context the lsass.exe process, the extension can be launched by typing "!mimikatz".



The above screenshot provides some translation for the credentials extracted using the Mimikatz Windows Debugger extension. The upcoming exercise will have you extract the plaintext passwords from a memory image in order to access a zip file that a user has made the mistake of reusing the same password as his user account to protect.



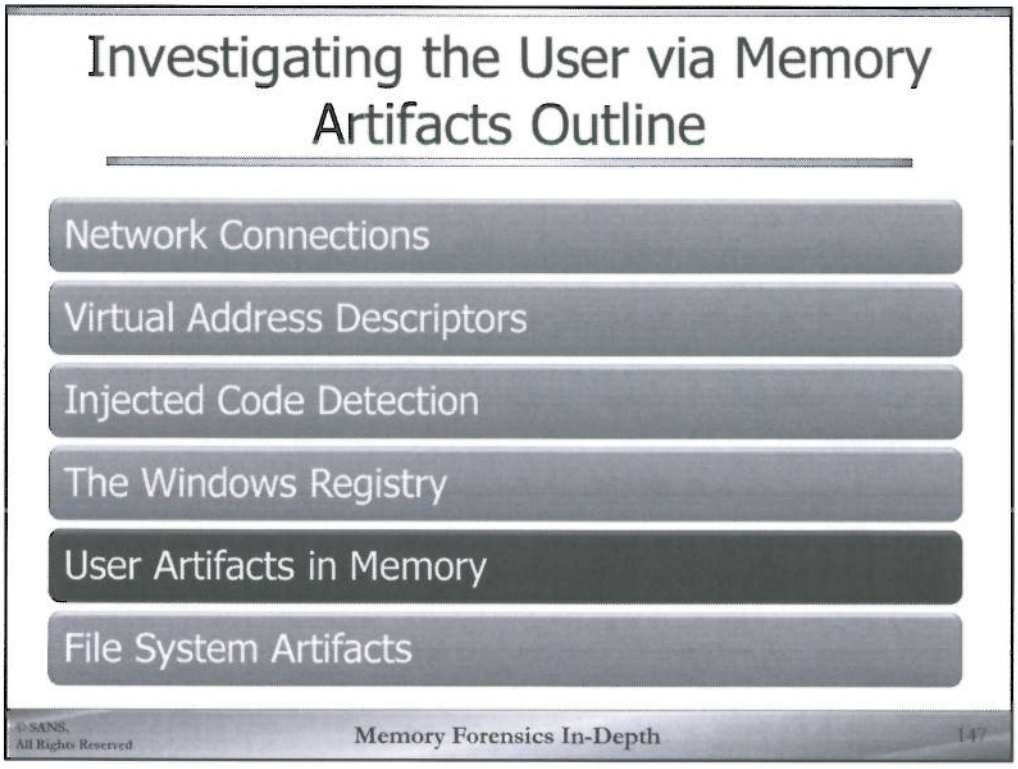
This page intentionally left blank.



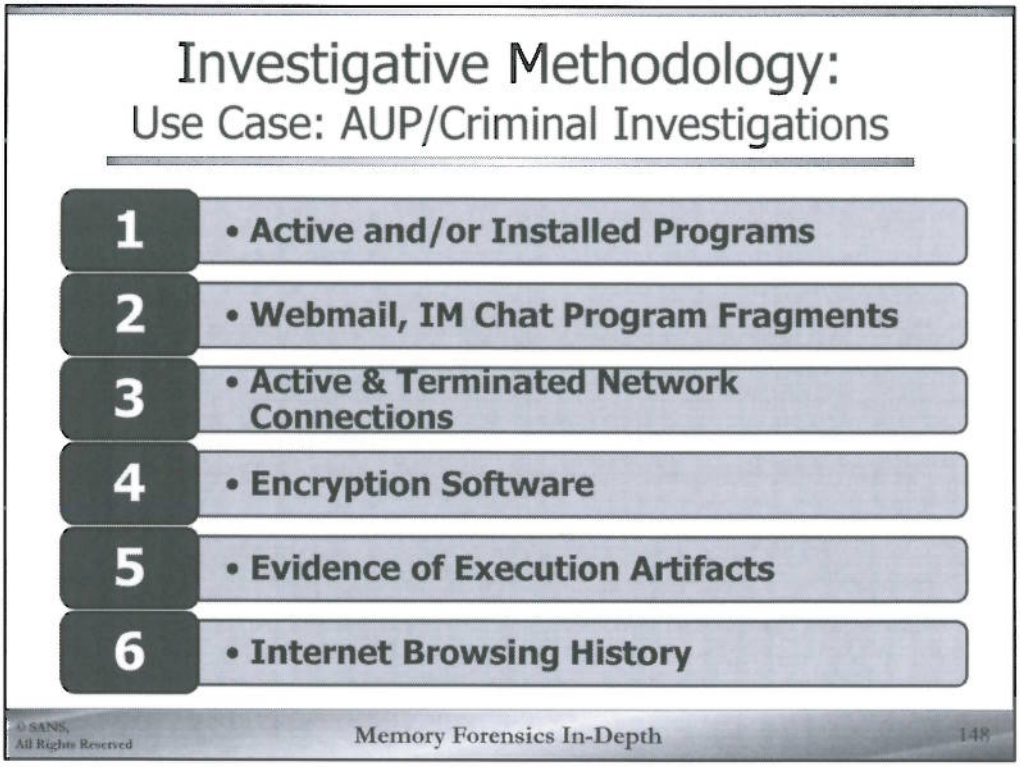
Having to convert a raw memory image to a crashdump in order to load it into Windows Debugger is a great deal of work, making us ask, "Is there an easy button for plaintext password extraction?" Due to the work of Francesco Picasso, we now have a Volatility plugin that extracts plaintext passwords from lsass process. It is not as comprehensive as the mimikatz WinDbg extension and supports only the wdigest authentication package on Windows Vista and 7, both x86 and x64 versions.

For more information on this plugin, go here:

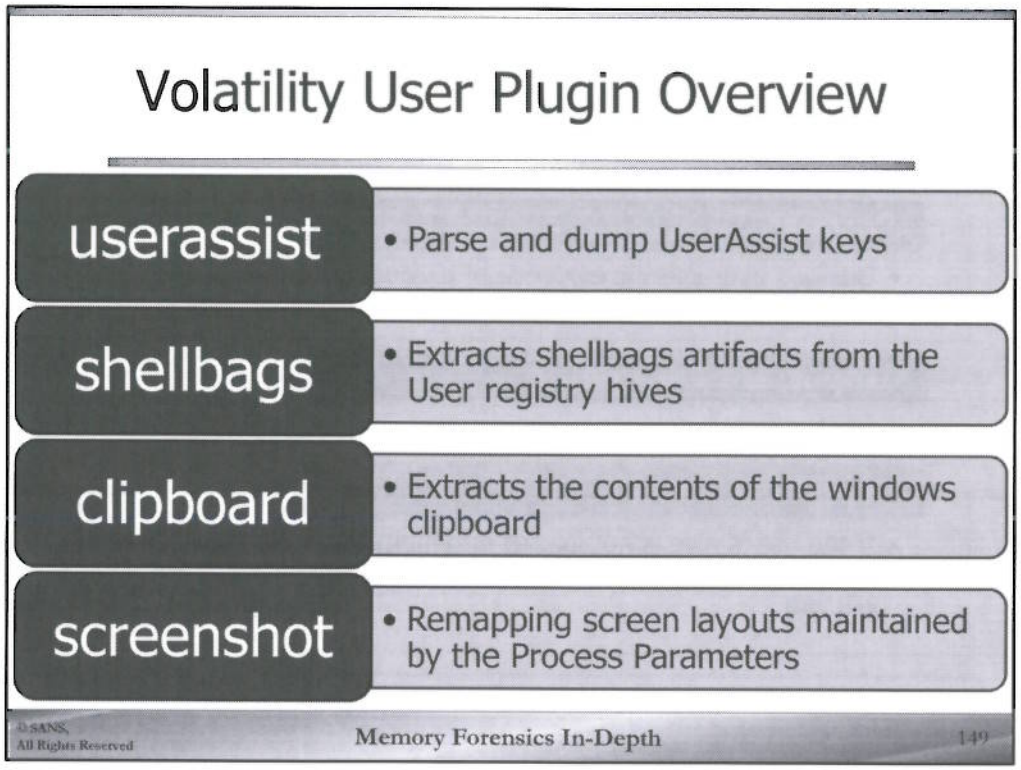
http://blog.digital-forensics.it/2014/03/et-voila-le-mimikatz-offline.html



This page intentionally left blank.

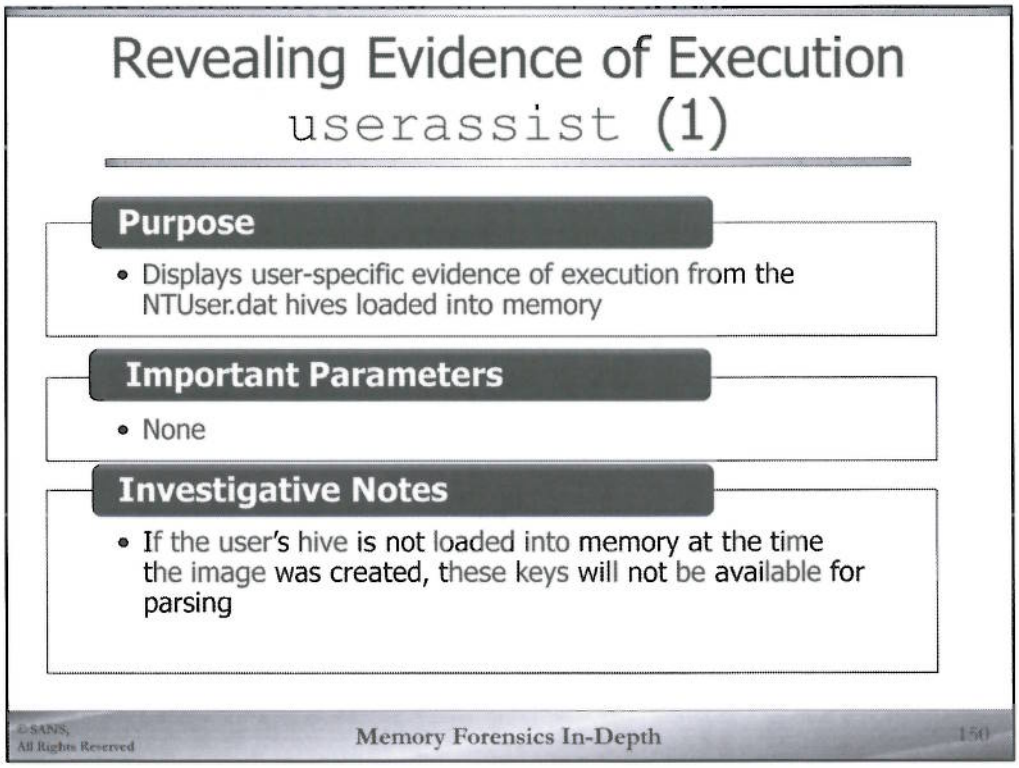


In "Cases Other Than Malware", investigators can use many key artifacts found in memory to recreate past or present user activity on the target system. In this section, we will discuss some Volatility plugins that target user-specific artifacts, whether it be browsing history or evidence of file knowledge or execution. Our goal is to make memory forensics accessible and applicable for use in ALL type of forensics investigations. If employee or criminal investigations are part of your daily job, this part of the course was specifically designed for you.

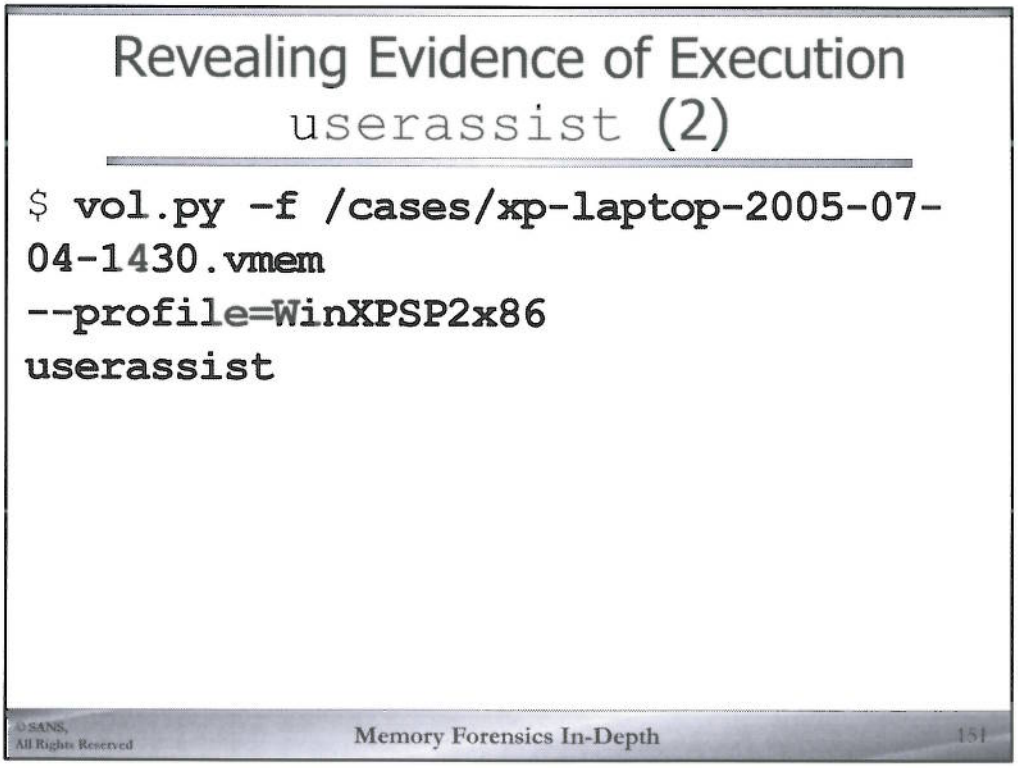


The **userassist** plugin, written by Jamie Levy, displays data from the User Assist keys. These keys, maintained by Windows Explorer, describe how often some applications on the system have been executed. They are found under HKEY_CURRENT_USER\Software\MicrosoftWindows\CurrentVersion\Explorer\UserAssist and obfuscated using ROT13. This plugin simplifies the process of viewing the data.

The **shellbags** plugin parses these Windows artifacts from the user hives in memory. Shellbags are the configuration settings maintained by Windows that track user's viewing preferences for Windows explorer directories. These artifacts can point to user knowledge and access of specific files or directories, as well as providing insight into directory structures of removable devices.



The userassist plugin written by Jamie Levy, one of the core Volatility developers, displays userassist keys from NTUser.dat hives loaded into memory. These registry keys are obfuscated with ROT13 and have a strange run count that starts with 6 (for versions of Windows other than Windows 7) - details that an investigator needs to know if he is parsing the registry keys by hand. The userassist plugin accounts for both of these peculiarities and presents the data in an unobfuscated manner.



The Userassist keys can be used as evidence of execution and can point to how a program was launched, either via a shortcut or from the executable itself. Each value is preceded with "UEME_" and then one of the following identifying tags, "RunPATH", "RunCPL", or "RunPIDL", that indicate how the executable was launched.

RunPATH: Launched by the absolute path of the executable. Most likely, the user traversed via Windows Explorer to the .exe.

RunCPL: Control Panel applet launched

RunPIDL: Launched from a shortcut to the actual file, such a LNK file.

As seen from the output of the following command below, the user, Sarah, ran the program "cmd.exe" three times, with the last time of execution being 2005-07-04 18:20:58 UTC.

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
userassist
```

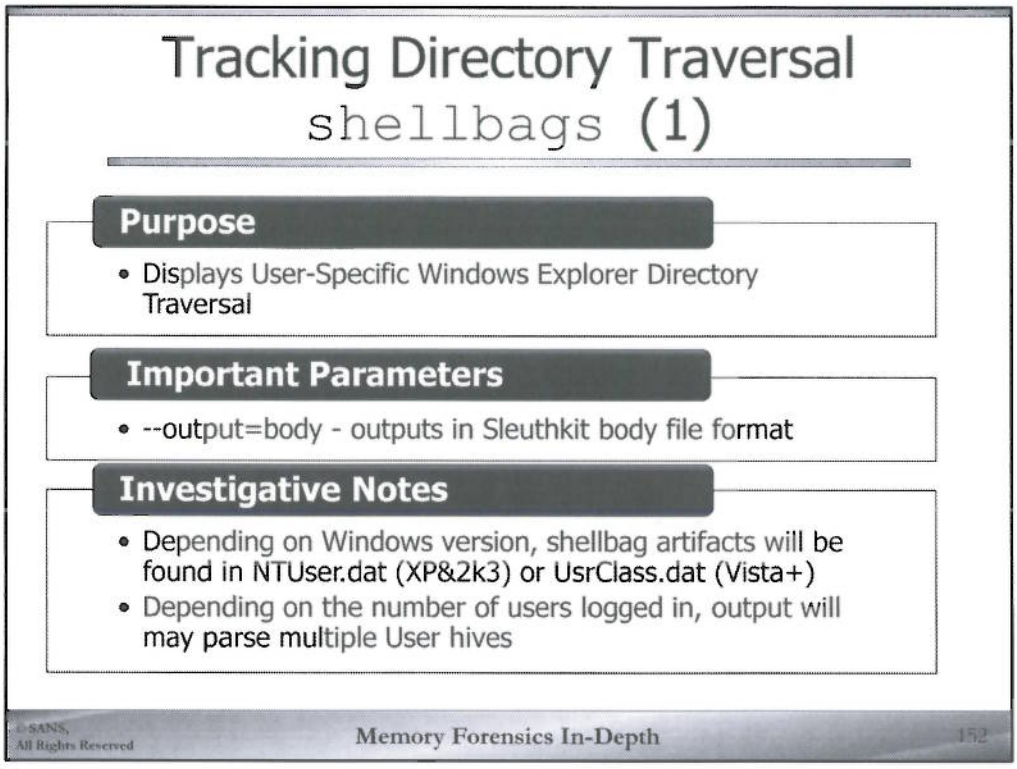
Volatility Foundation Volatility Framework 2.3.1

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

Key name: Count

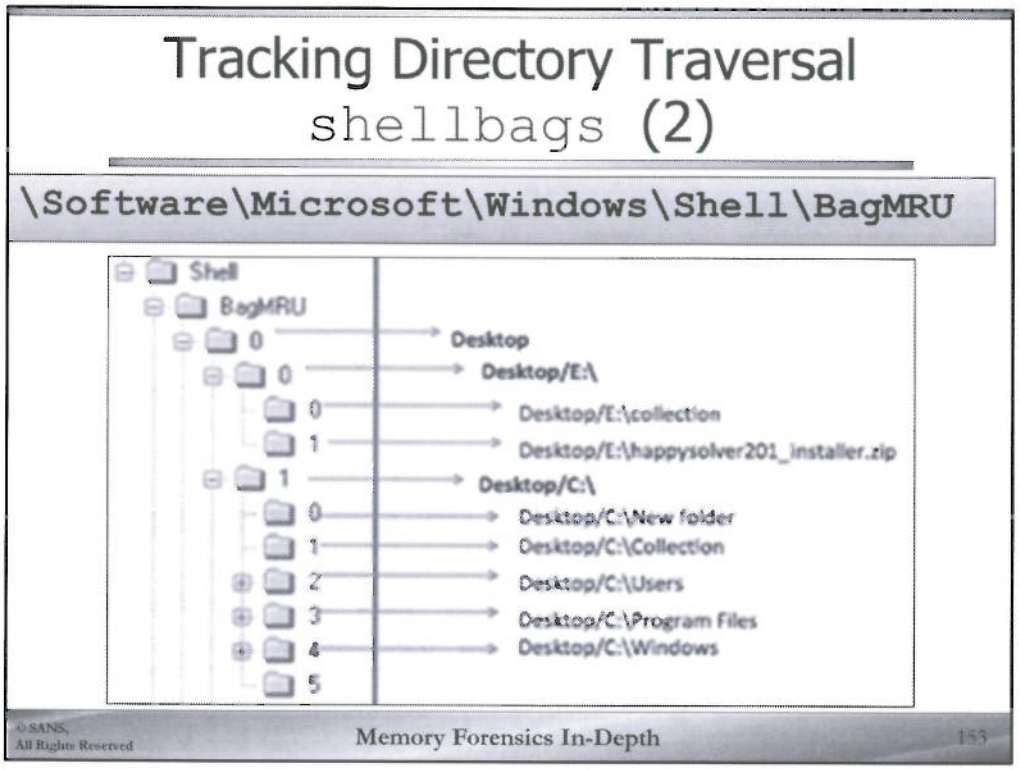
Last updated: 2005-07-04 18:25:27 UTC+0000

<Truncated>



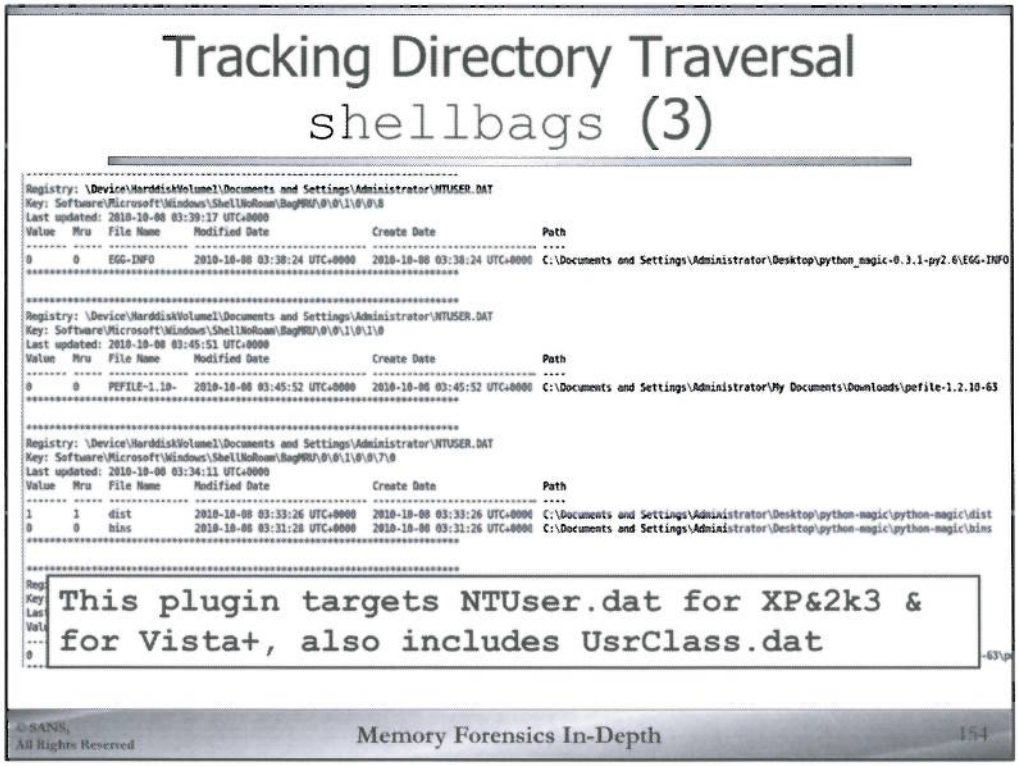
In most type of investigations, gaining visibility into where the user browsed on the local machine, removable devices or network share can be invaluable in substantiating his/her knowledge and access to directories. As investigators, we can track user-specific access to directories via Windows Explorer by parsing shellbags artifacts from the user registry hive.

The Volatility plugin, **shellbags**, parses the Windows version-specific user hive to extract these values.

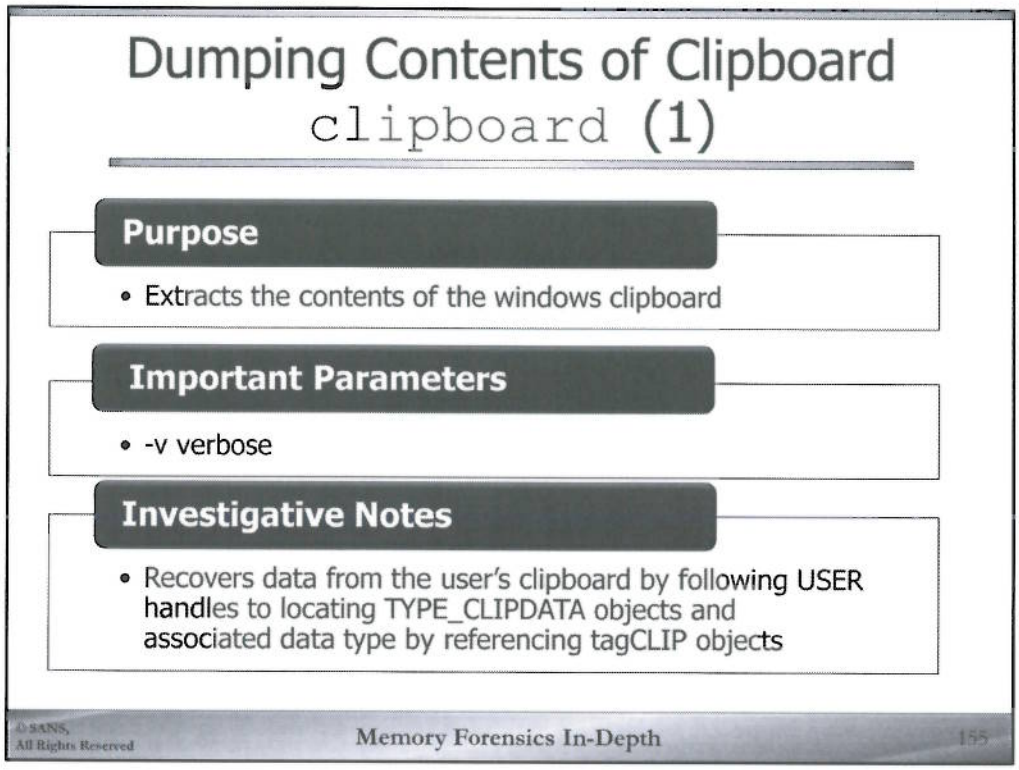


Here is a visual mapping of the reconstruction performed by the **shellbag** plugin. The BagMRU keys contain the directory tree structure representative of user-specific traversing via Windows Explorer. Corresponding Bag keys contain MACE time/date stamps for each directory. Shellbag data should be included user investigations to include intellectual property theft cases to show evidence of directory/file knowledge and potentially evidence of data copying. As shown above, the entire structure of removable devices, in addition to the local machine directory structure, is shown and could be used to prove that an employee copied sensitive data from a corporate network.

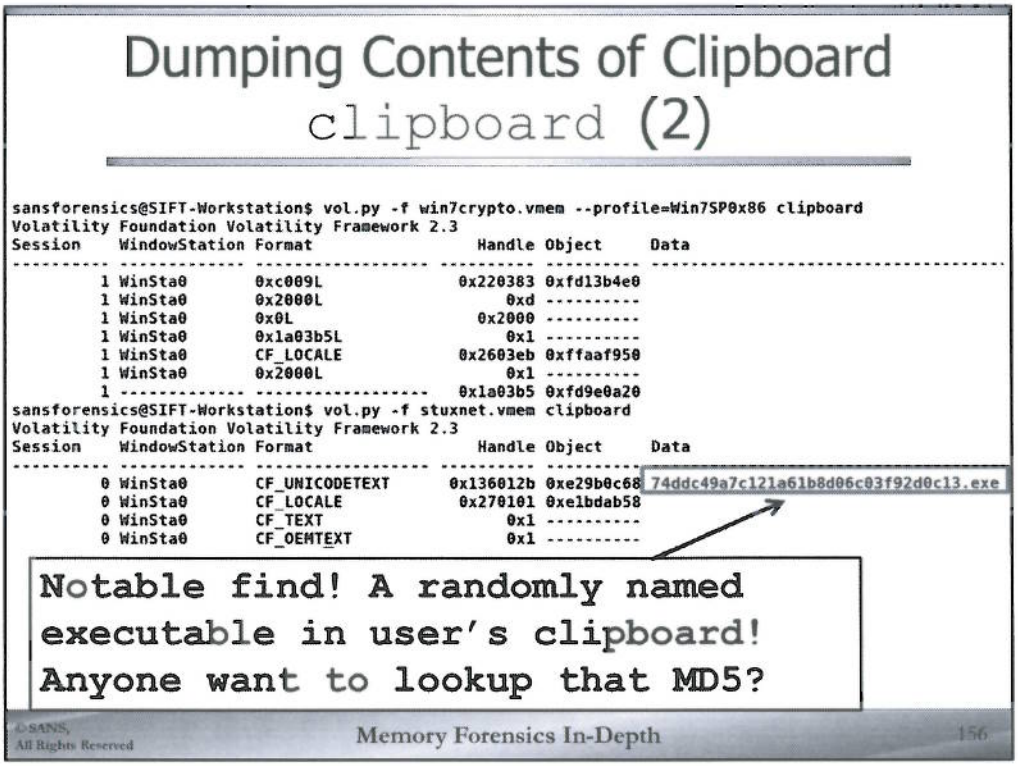
In intrusion investigations, shellbag artifacts become key evidence of attacker activity if he/she connected to the system via RDP (remote desktop session). Activity done in an RDP session creates the same artifacts when Windows Explorer is used to browse directories as a local logon.



Some interesting user directory traversal artifacts can be seen in the truncated **shellbags** output above. We can see the directory structure of the Administrator's desktop from the stuxnet.vmem file and allows for some user profiling. Clearly, based on his subdirectories on his desktop, this user is python programmer or researcher. Clearly, this output is from a Windows XP or Windows 2003 system due to the user registry hive that is being parsed, in addition to the file path "C:\Documents and Settings".

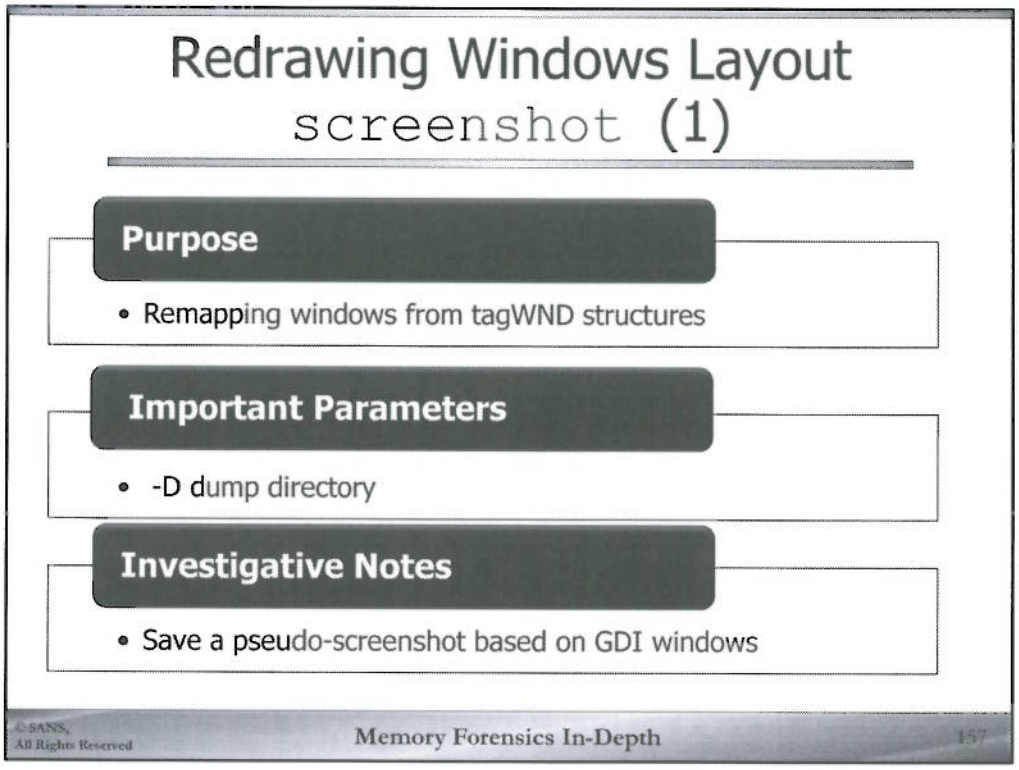


There is something almost magical about being able to access a user's clipboard days, even months or years, after they populated it with a Ctrl-C. The **clipboard** plugin gives us this power. By following USER handle table (making use of the **userhandles** plugin functions), it is able to locate TYPE_CLIPDATA objects that reference "clipped data" and then apply the associated data type by referencing tagCLIP objects pointed to by tagWINDOWSTATION.pClipBase.

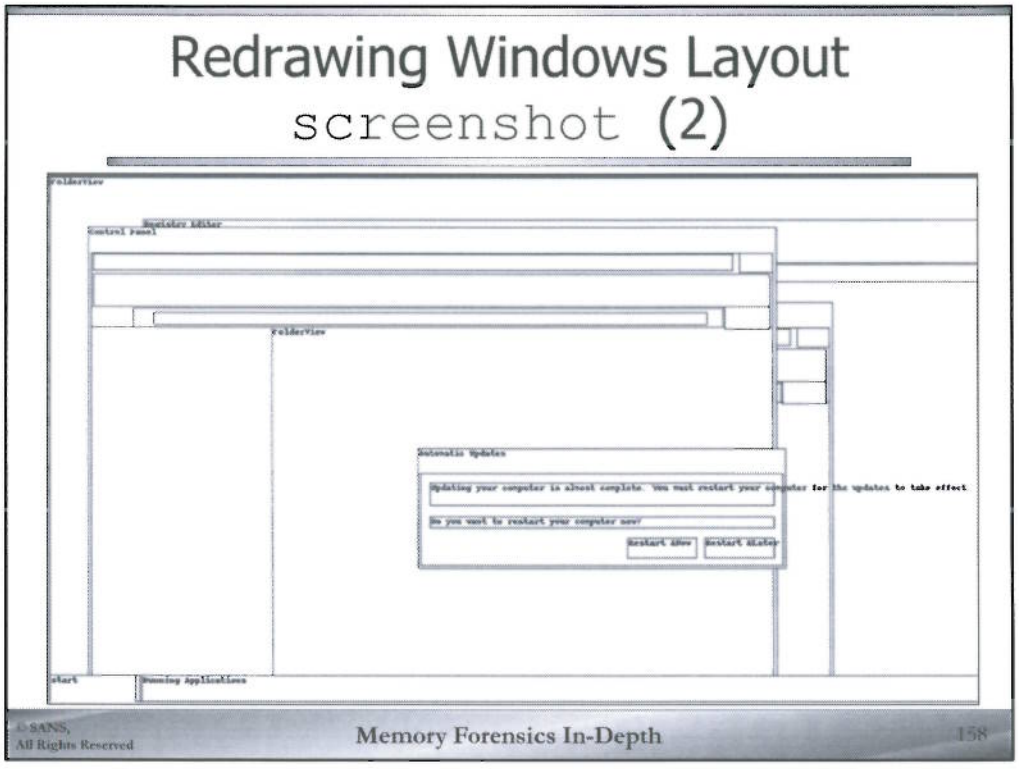


The underworkings of the Windows clipboard function are quite complex. There are many different formats of data that applications can copy into the clipboard, with some standard types (bitmap, Unicode or ANSI text, TIFF) and some that are custom to specific applications. To populate the clipboard, an application uses the OpenClipboard function and then empties the clipboard to prep it for its use through the EmptyClipboard function. It then populates the clipboard using SetClipboardData in one of the accepted formats. There are actually several different formats of the clipped data stored in memory for best compatibility with a destination application.

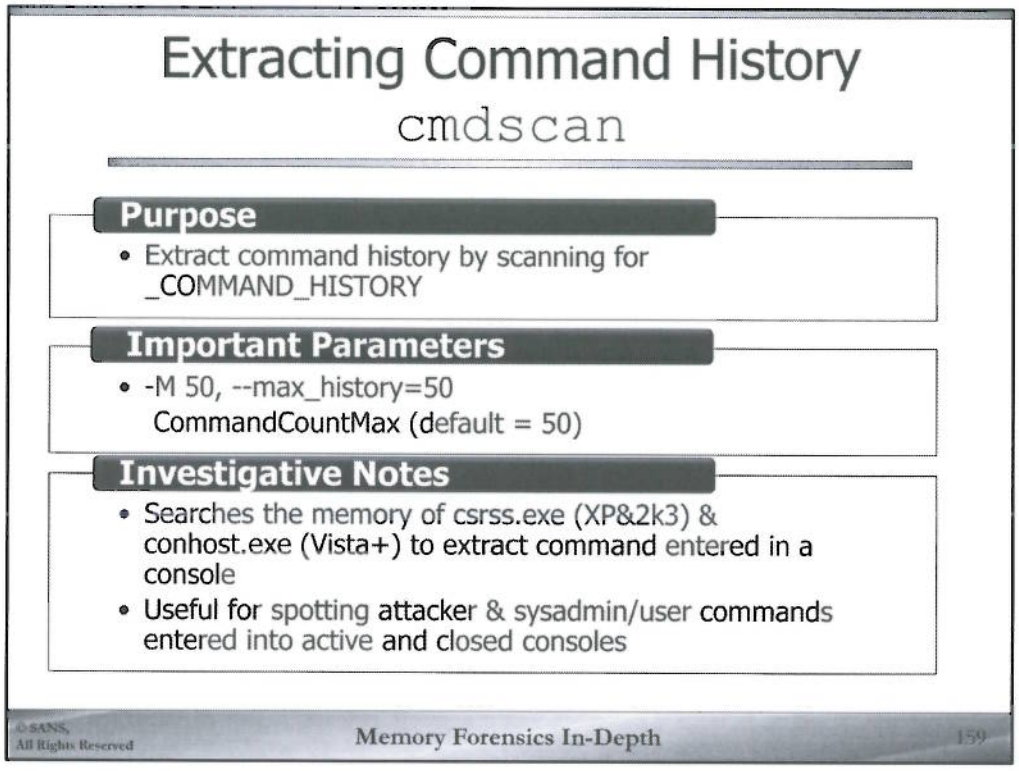
You can see evidence of the multiple formats of the clipboard data in the clipboard plugin output above. You can see that the populating application only filled the UNICODETEXT format. The contents of that clipboard are quite interesting, as it appears to be a executable file with a MD5 filename.



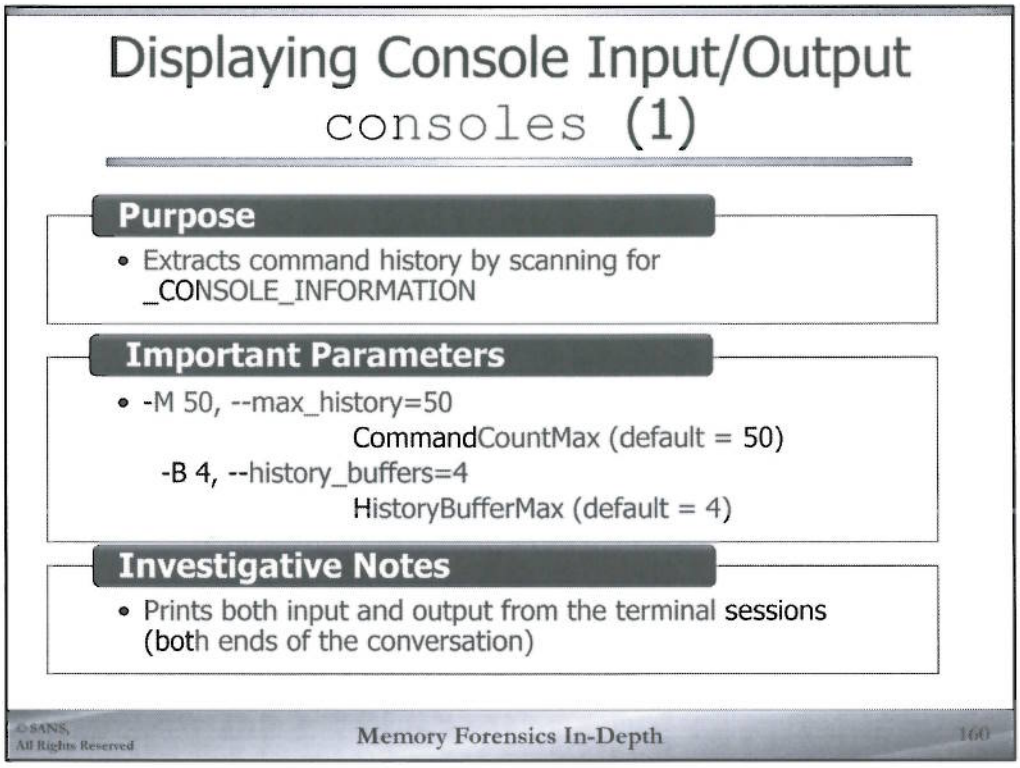
Getting a frame layout of the active desktop windows at the time the system memory is dumped may prove relevant to some user or intrusion investigations. The screenshot plugin offers the examiner this insight by redrawing the windows using the screen layout coordinates from the tagWND structure. Though most of the desktops redrawn and extracted by this plugin will not have active windows since there is only one interactive window station per user logged in, you are typically able to get one populated screenshot .png file per interactive session.



Here is the **screenshot** plugin output, redrawing the user's desktop from a hiberfil.sys file that has been converted to a raw memory dump. As you can see from the Windows Updates window, the system had completed automatic updates prior to hibernating.



As discussed in the unstructured memory analysis portion of this class, the csrss process and conhost process track commands enter into a terminal/console. By searching through the memory of these processes, commands can be extracted from a memory image that were once type on the target system in a console window. Volatility makes this easy for investigators with the **cmdscan** plugin. This plugin works by searches for a known constant value of MaxHistory and using it to identify the structure COMMAND_HISTORY within the process memory. One of the options available for this plugin is the manipulation of the MaxHistory (-M #) as this value can be changed by the user, thereby changing the value that cmdscan uses to locate the COMMAND_HISTORY. The investigator can manipulate this MaxHistory if the user set it to anything other than 50.

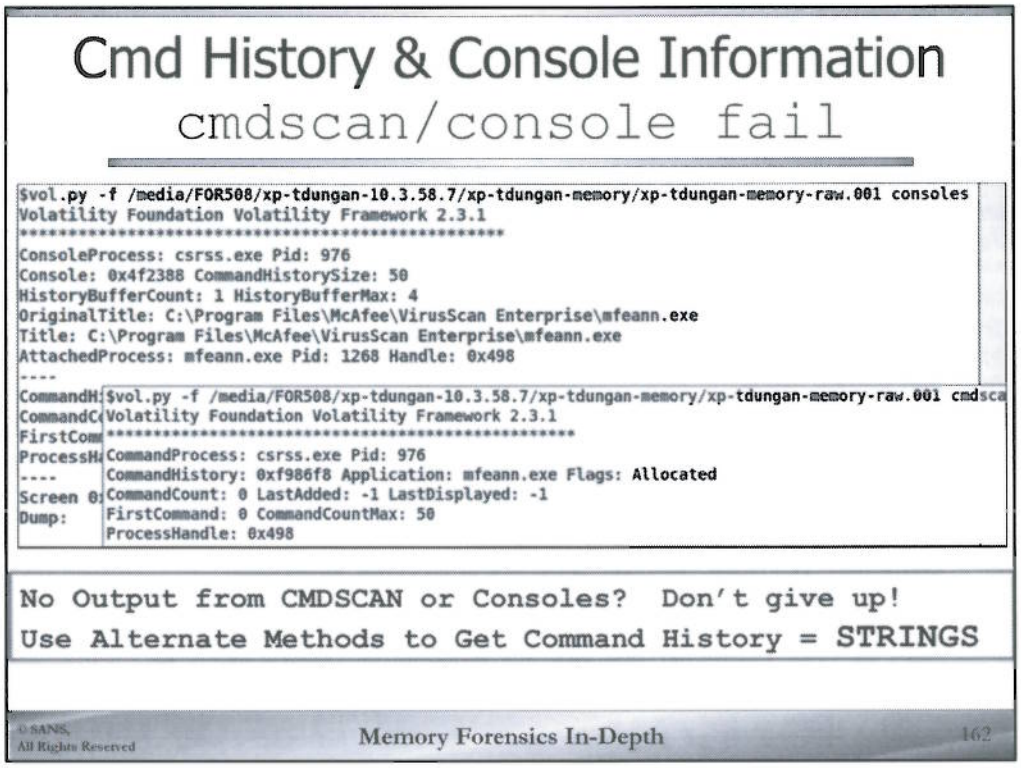


Consoles, like cmdscan, gives the investigator visibility into activity of open or terminated consoles. Unlike cmdscan though, the consoles plugin extracts the input and output of a console. This plugin in works by identifying and parsing the _CONSOLE_INFORMATION object by using known values such as MaxHistory and HistoryBufferMax. If these are known to be changed from the default, the investigators can manually enter different values that the plugin can use to locate the _CONSOLE_INFORMATION object.

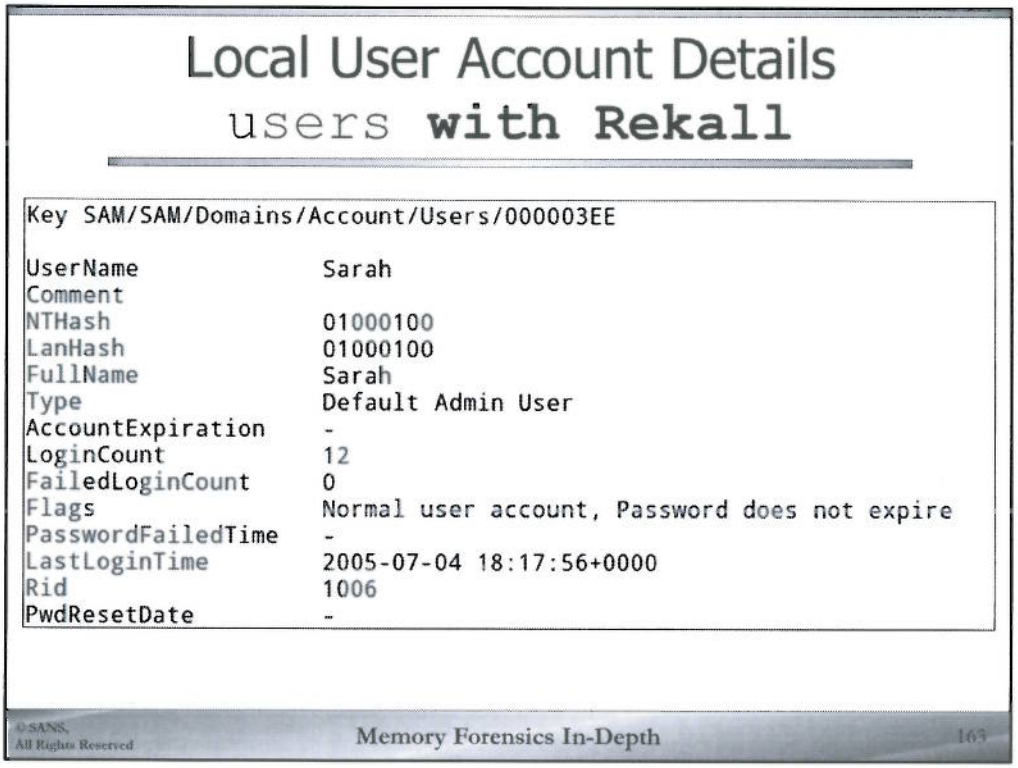
Displaying Console Input/Output
consoles (2)

```
sansforensics@SIFT-Workstation:/cases$ vol.py -f memdump-1.img consoles
Volatility Foundation Volatility Framework 2.3
**************************************************
CommandHistory: 0x10f39f0 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 2 LastAdded: 1 LastDisplayed: 1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x620
Cmd #0 at 0x4fcfe8: cd Desktop
Cmd #1 at 0x4f2370: mdd.exe -o memdump-1.img
C:\Documents and Settings\Administrator\Desktop>mdd.exe -o memdump-1.img
  -> mdd
  -> ManTech Physical Memory Dump Utility
     Copyright (C) 2008 ManTech Security & Mission Assurance

  -> This program comes with ABSOLUTELY NO WARRANTY; for details use option
     This is free software, and you are welcome to redistribute it
     under certain conditions; use option `-c' for details.

  -> Dumping 511.48 MB of physical memory to file 'memdump-1.img'.
```

Above is an example of **consoles** output, showing the execution of mdd.exe in a command shell. With consoles, the investigator has access to what the user/attacker saw in the window in response to their commands.

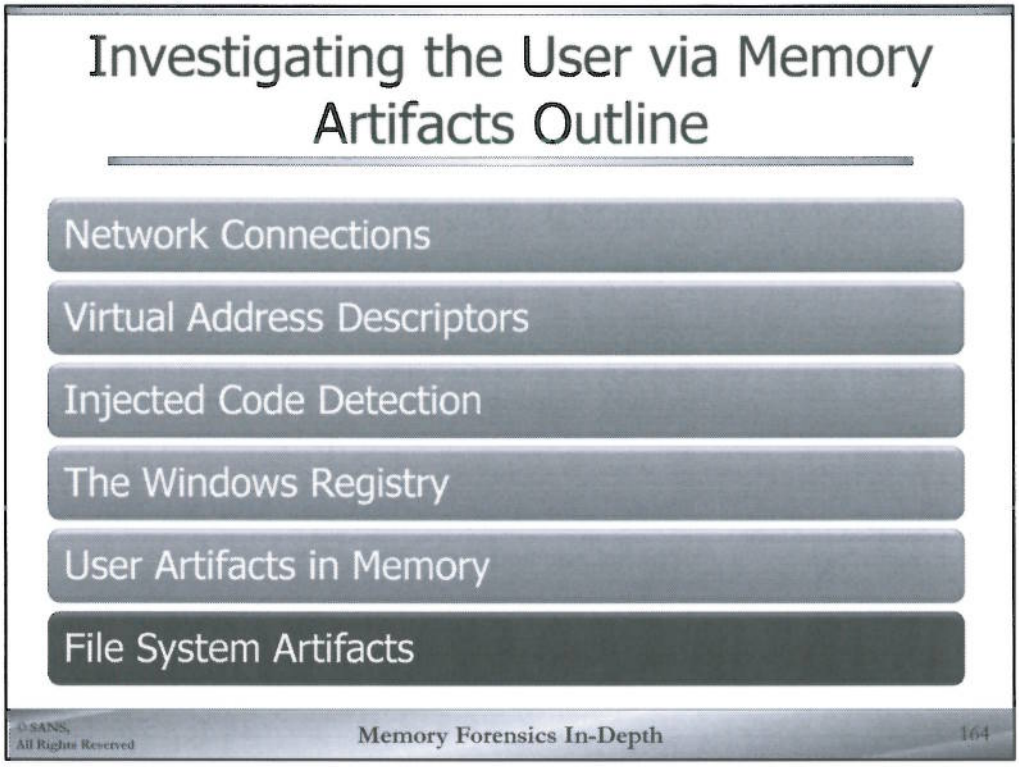


There are times when both cmdscan and consoles fail to produce any notable output. In these instances, the examiner may be able to gain insight into what was entered in a terminal windows on the target system by extracting the process address space of the csrss.exe (WindowsXP/2k3) or the conhost.exe/cmd.exe processes (Vista+) using memdump. Then by running strings against the resultant .dmp file, commands and relevant terminal input may be uncovered.

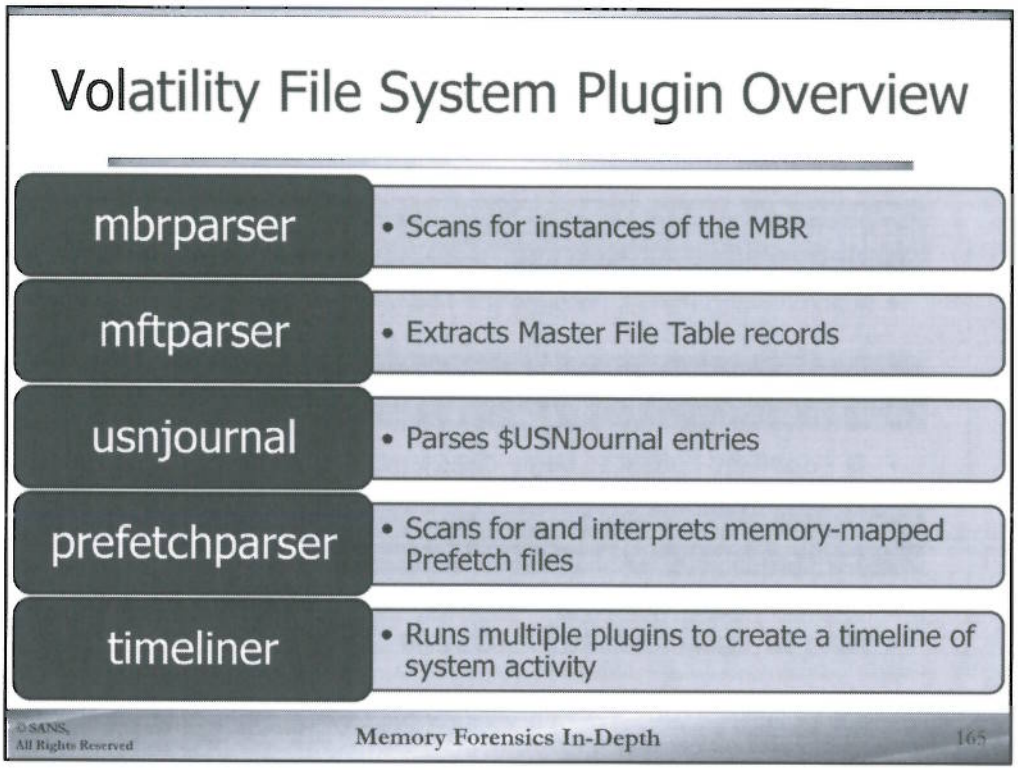


Easy access to the local user account details is available using Rekall's **users** plugin. Relevant time/datestamps pulled from the SAM registry hive include Last Login Time, Time/Date of last Password Change and Login Count.

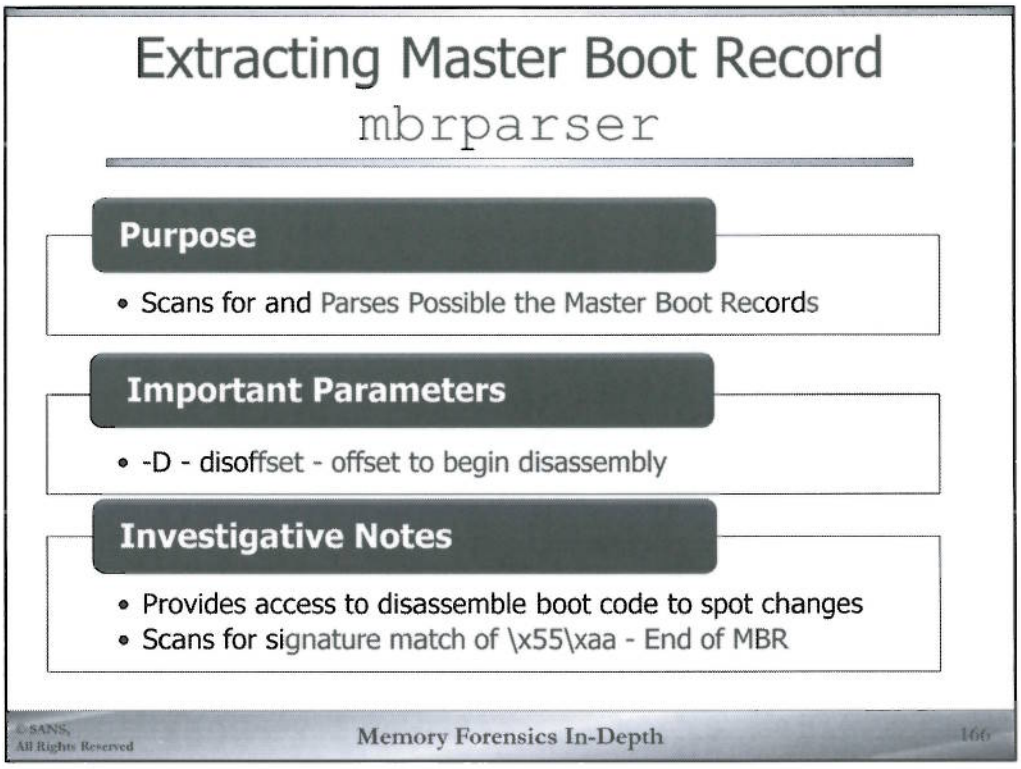
```
[1] xp-laptop-2005-07-04-1430.vmem 21:09:45> users
----------------------------------------> users()
**************************************************
Key SAM/SAM/Domains/Account/Users/000001F4

UserName        Administrator
Comment         Built-in account for administering the computer/domain
NTHash          01000100dd2323146b4e50fdbdd53ed4af7579aa
LanHash         0100010002fcfbdff8efd4d58ac09f2a069ad0e3
FullName
Type            Default Admin User
AccountExpiration  -
LoginCount      0
FailedLoginCount   0
Flags           Normal user account, Password does not expire
PasswordFailedTime  -
LastLoginTime      -
Rid             500
PwdResetDate       2004-05-04 23:16:42+0000
**************************************************
<Output Truncated>
```



This page intentionally left blank.

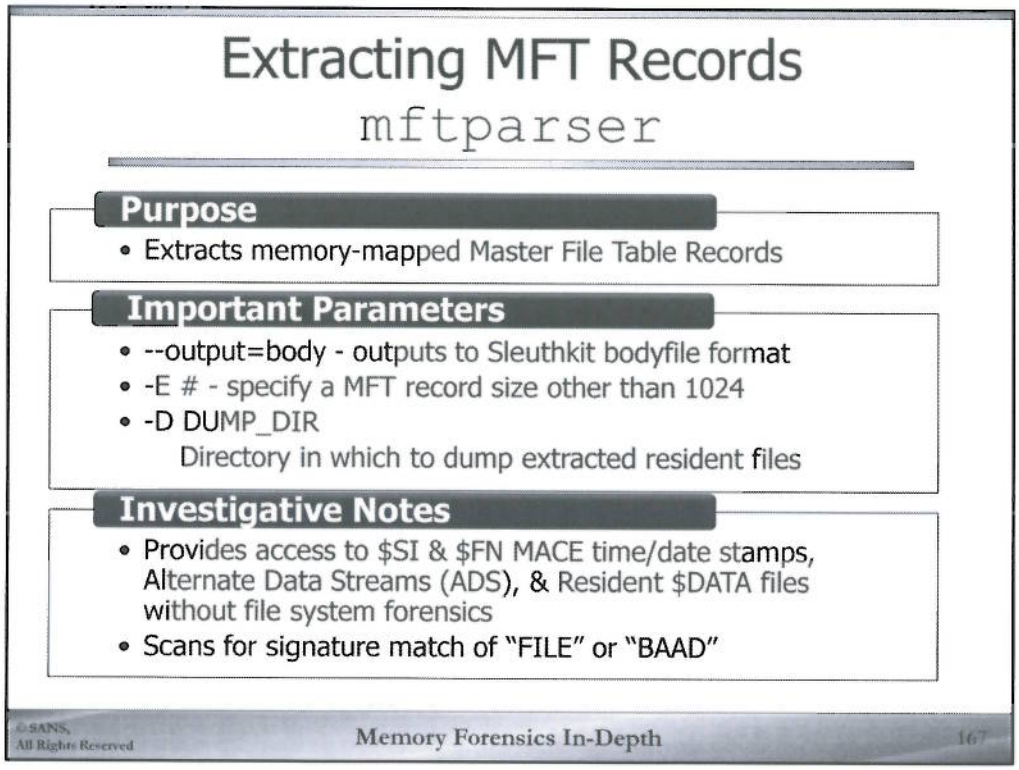


This page intentionally left blank.



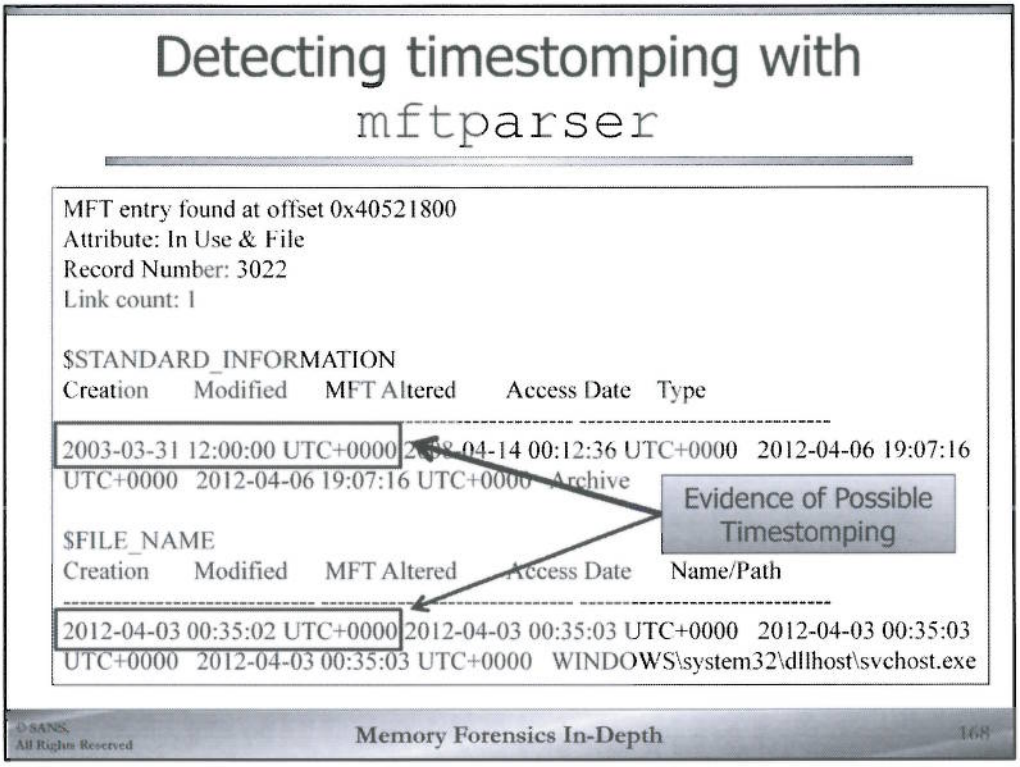
The Master Boot Record has a set size of 512 bytes and known layout with 440 bytes of bootcode, a 64 bytes partition table and a final two byte footer of \x55\xaa. Volatility plugin, **mbrparser**, not only displays possible Master Boot Record matches in the system memory image, it provides bootcode disassembly. The output of this plugin include hash values of both the bootcode and the entire MBR, which can be used to both establish a baseline and spot deviations. In comparing the baseline bootcode hash to that of a potentially compromised system, the investigator may be able to detect MBR infection or modified bootcode.

One point to note, if disk encryption is in use on the target system, the MBR may be skipped over during boot, the result of redirection to the encryption software's bootcode. It is recommended an examiner understand how encryption software is implemented in order to identify MBR infections.

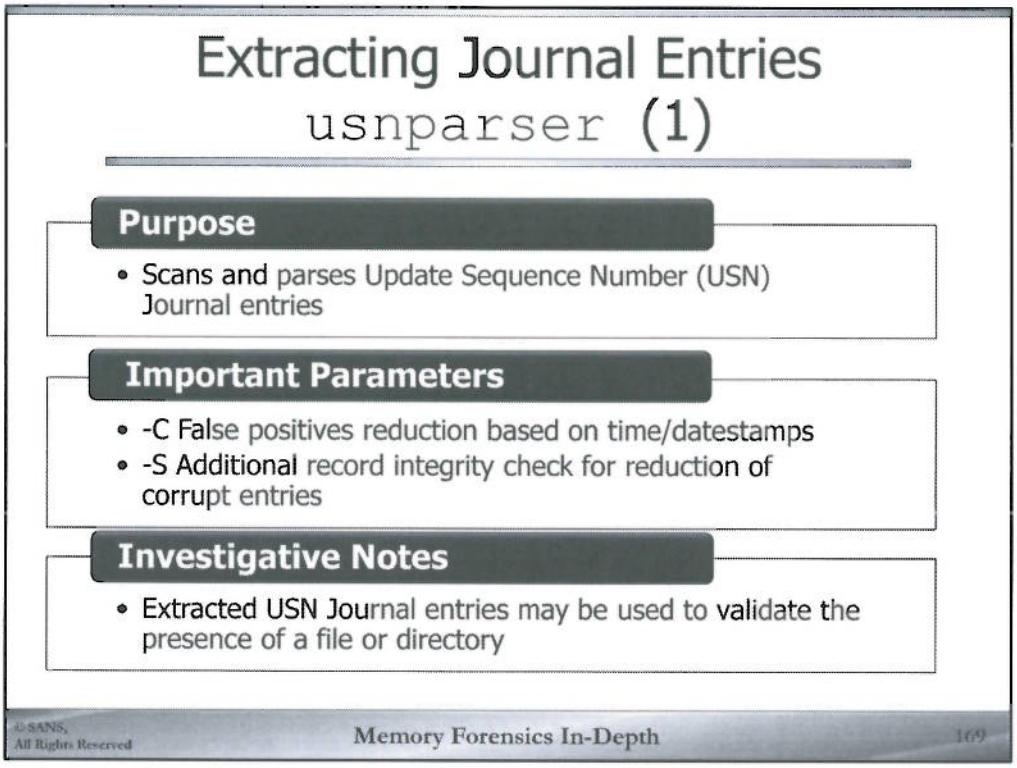


MFTParser is a fantastic plugin that allows an investigator to gain access to MFT Records **without** the volume image of the target system. Though it takes a long time to run, gaining access to the file system metadata can provide forensic artifacts such as creation or modified time of a file or directory. Timestomping is easy to detect as well due to access to $SI and $FN time/date stamps.
Additional options built into this tool include the ability to extract Resident $DATA files and Alternate Data Streams (additional $DATA attributes) by specifying a dump directory.

```
**************************************************************************
**************************************************************************
MFT entry found at offset 0x1ffaa800
Attribute: In Use & File
Record Number: 39930
Link count: 1

$STANDARD_INFORMATION
Creation                  Modified                  MFT Altered               Access Date               Type
------------------------- ------------------------- ------------------------- ------------------------- ----
2004-05-06 22:58:28 UTC+0000 2005-04-07 03:12:20 UTC+0000  2005-04-07 03:12:20 UTC+0000  2005-04-21
22:00:51 UTC+0000  Compressed & Archive & Content not indexed

$FILE_NAME
Creation                  Modified                  MFT Altered               Access Date               Name/Path
------------------------- ------------------------- ------------------------- ------------------------- ---------
2005-04-21 22:00:51 UTC+0000 2005-04-21 22:00:51 UTC+0000  2005-04-21 22:00:51 UTC+0000  2005-04-21
22:00:51 UTC+0000  A0047164.ini

$DATA
Non-Resident

**************************************************************************
```



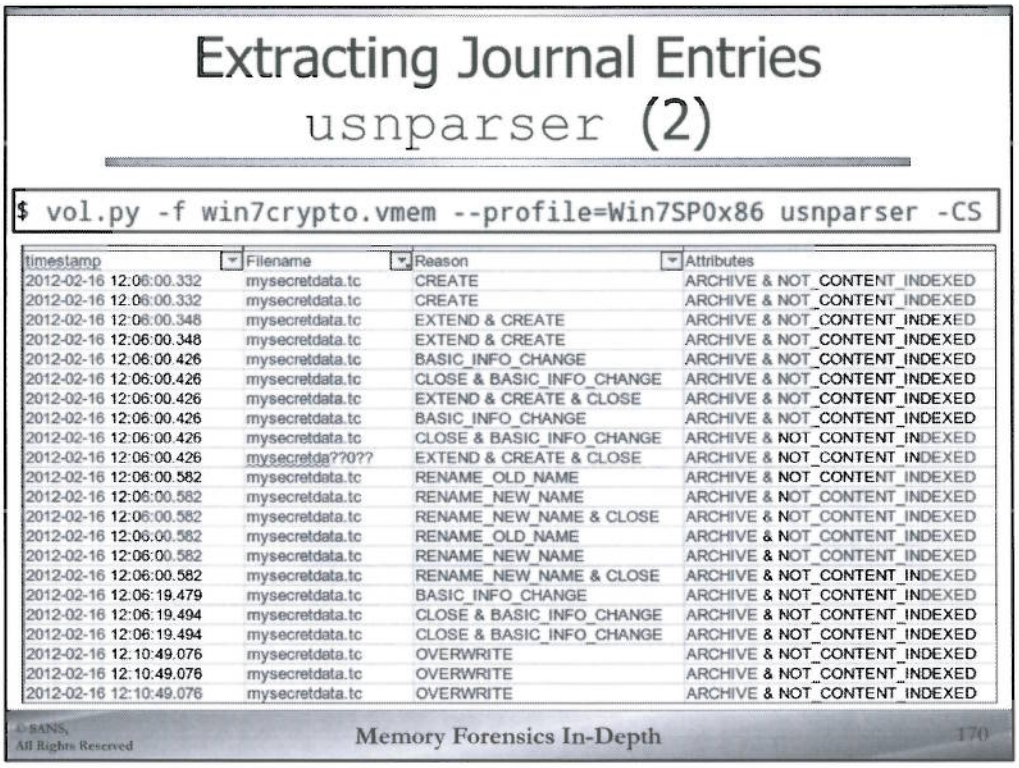
Based on the MFT Record for this "svchost.exe" file extracted from the target system's memory capture using Volatility's MFTParser plugin, a common anti-forensics technique, timestomping, is easy to spot. As shown here, when the $Standard_Information attribute time/date stamps predate the $File_Name attribute time/date stamps, it is indicative of anomalous behavior typically seen when time/date stamps are manipulated through the Windows API. This can be done in an automated fashion, as a "feature" of malicious code, or manually by the attacker, through post-exploitation activity.



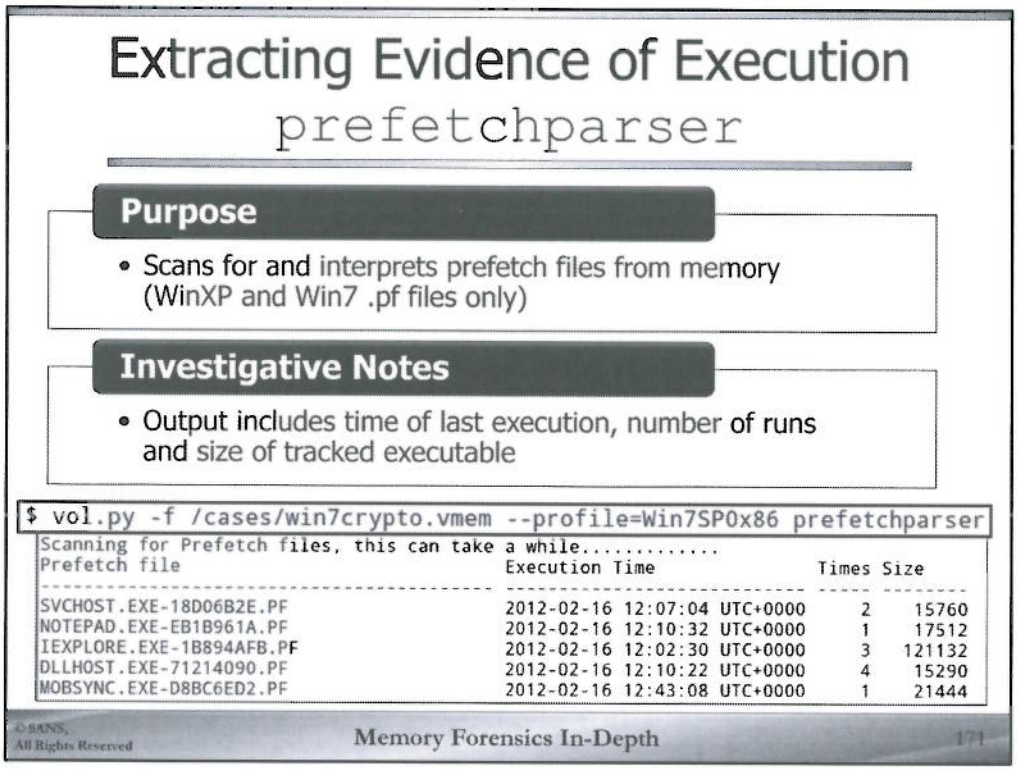
The USNParser, authored by Tom Spencer, specifically parses for $USN Journal entries from memory. There are numerous options available for this plugin, show below, to include outputting to body, text or csv format.

```
-T, --timestamp      Print timestamps instead of human-readable dates
 -X, --unixtime      Use Unix Epoch 32-bit timestamps instead of native
                     Windows 64-bit timestamps (loses subsecond accuracy).
                     DOES NOT imply -T above.
 -C, --check         Don't show entries with timestamps outside of Unix
                     epoch range to reduce corrupt entries
 -S, --strict        Enable stricter checks on record integrity to further
                     reduce corrupt entries
 -O, --offset        Show the physical offset for each record
 -R RECORDTYPE, --recordtype=RECORDTYPE
                     Force version of USN record (2 or 3) to search for. In
                     testing so far all OS's seem to use version 2 records
                     in memory (even 8.1/2012r2 which purport to use R3).
                     As such, default is R2.
 -U, --unicode       Show unicode (utf-8) filenames. Be aware that due to
                     corrupted records there will likely be strange
                     characters in some places. Using -C and -S can help
                     cut this down.
```

Module Output Options: body, csv, text



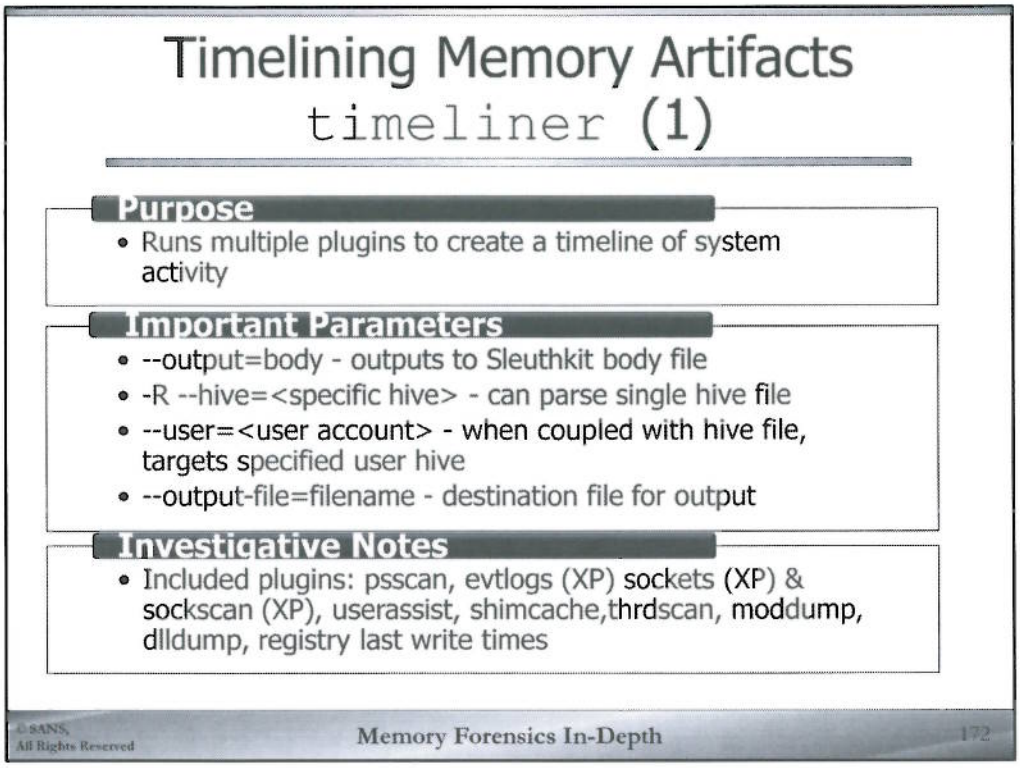
This example shows the output of **usnparser** in csv format, making use of the "--output=csv --output-file=/cases/win7crypto.csv" options. The win7crypto.vmem memory image makes reference to the truecrypt file "mysecretdata.tc", as you may remember seeing reference to in the bulk_extractor exercise. Filtering the usnparser output on the filename, after sorting by date, we are able to see the creation of the .tc file as well as some file accesses and metadata changes.



The prefetchparser plugin was written by Dave Lassalle, winner of the 2014 Volatility plugin contest with his Forensic Suite that includes Firefox and Chrome browser artifact parsers. We have added his contributions, to include prefetchparser, to the FOR526 SIFT volatility 2.4 main plugin directory.

vol.py --plugins=. -f /cases/stuxnet.vmem prefetchparser
Scanning for Prefetch files, this can take a while.....

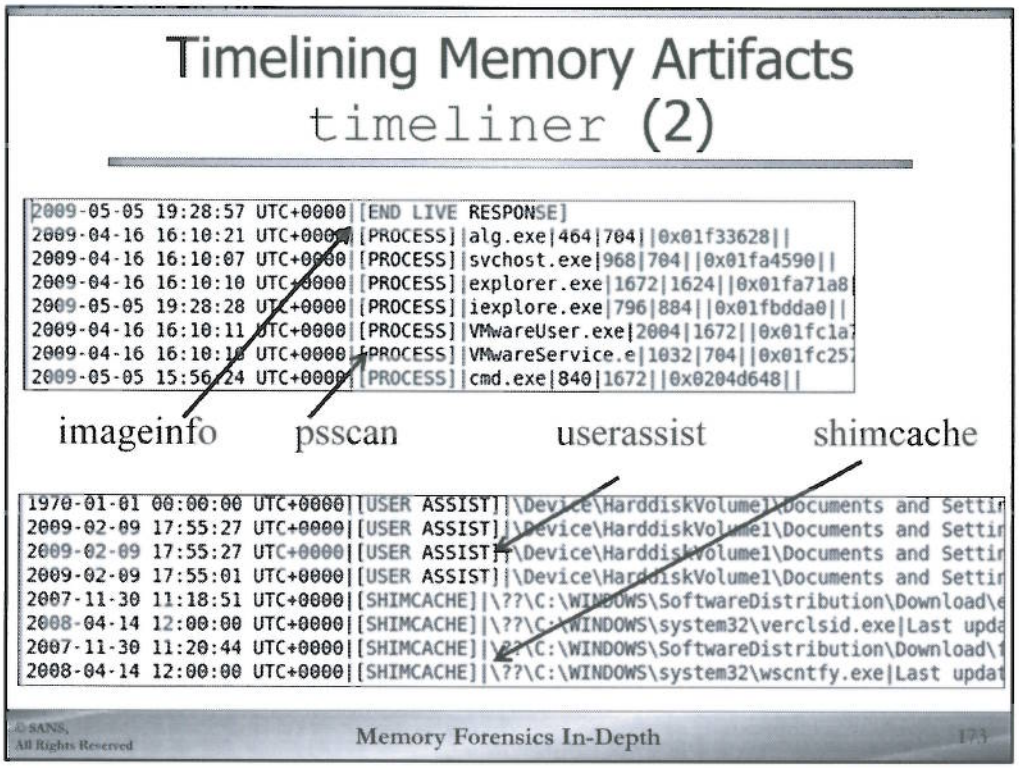
Prefetch file	Execution Time	Times	Size
PROCEXP.EXE-28BEBBF2.PF	2011-06-03 04:25:51 UTC+0000	1	13606
74DDC49A7C121A61B8D06C03F92D0-2DC88F6.PF	2011-06-03 04:26:46 UTC+0000	1	14016
VERCLSID.EXE-3667BD89.PF	2011-06-03 04:25:59 UTC+0000	51	20118
IMMUNITYDEBUGGER.EXE-16086EFB.PF	2010-10-31 16:48:12 UTC+0000	2	90230
IPCONFIG.EXE-2395F30B.PF	2011-06-03 04:21:27 UTC+0000	9	22044
LSASS.EXE-20DB6D1B.PF	2011-06-03 04:26:55 UTC+0000	5	16146
CMD.EXE-87B4001.PF	2011-06-03 04:21:27 UTC+0000	21	16392
WUAUCLT.EXE-399A8E72.PF	2011-06-03 04:21:28 UTC+0000	17	20946
NOTEPAD.EXE-336351A9.PF	2010-10-31 16:48:32 UTC+0000	1	11502
WMIPRVSE.EXE-28F301A9.PF	2011-06-03 04:26:00 UTC+0000	9	42138
UPDATER.EXE-7ACF858.PF	2010-10-31 16:36:19 UTC+0000	1	11204
PYTHON.EXE-2D18E9AA.PF	2010-10-31 16:55:15 UTC+0000	3	66994
LOGON.SCR-151EFAEA.PF	2010-10-31 13:59:47 UTC+0000	9	6042
PROCMON.EXE-25EFF911.PF	2011-06-03 04:25:56 UTC+0000	1	51398



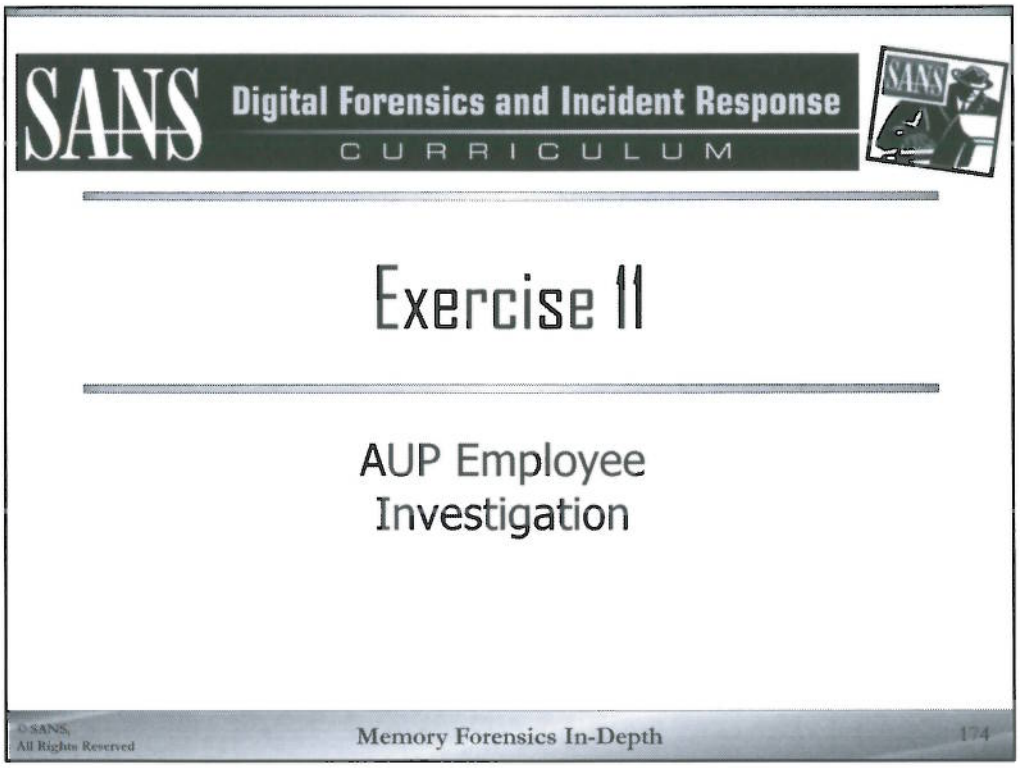
The timeliner plugin consists of many different plugins being run and output to the same file of a specified format. The plugins run by the Volatility 2.3 timeliner include imageinfo, psscan, evtlogs (XP/2k3), shimcache, userassist and optionally, registry lastwrite time extraction. You can limit the registry parsing to specific hive files and users making use of --hive and --user. For example:

```
$ vol.py -f APT.img timeliner -R --hive=user.dat --user=demo --output-
file=/cases/demo.txt
```

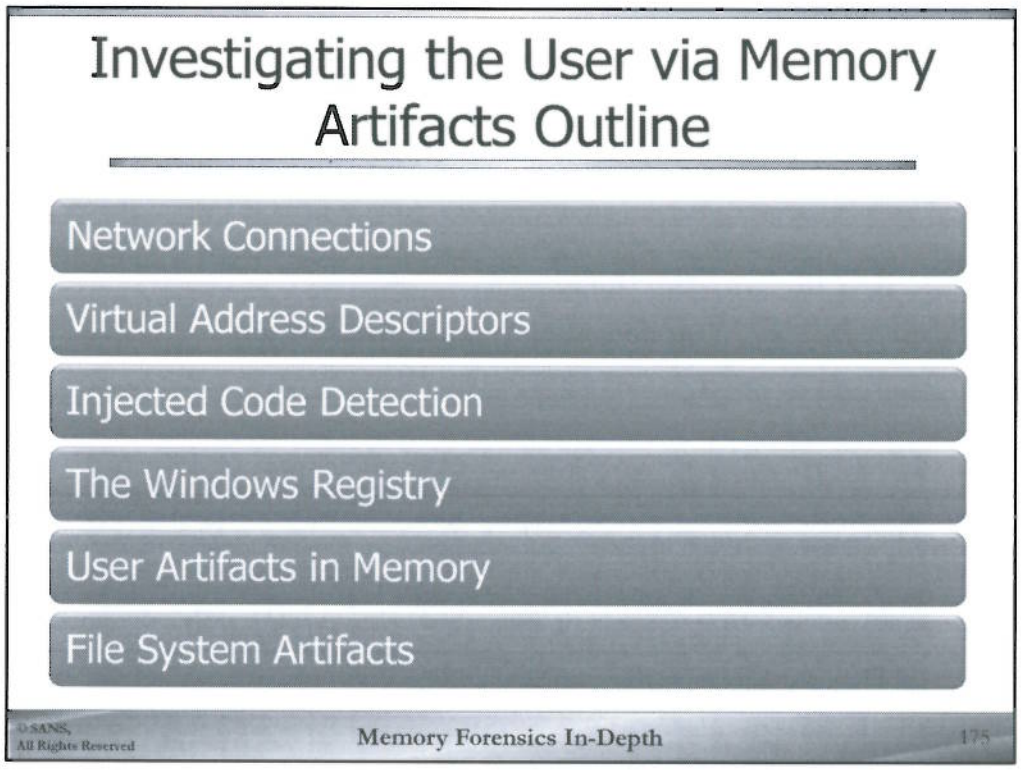
The output formats that are supported consist of Sleuthkit body file, Excel (requires openpyxl) and text file. With the body file format, the timeliner output can be added to a log2timeline bodyfile for a more complete picture of changes occurring on the target system in both the filesystem and memory.



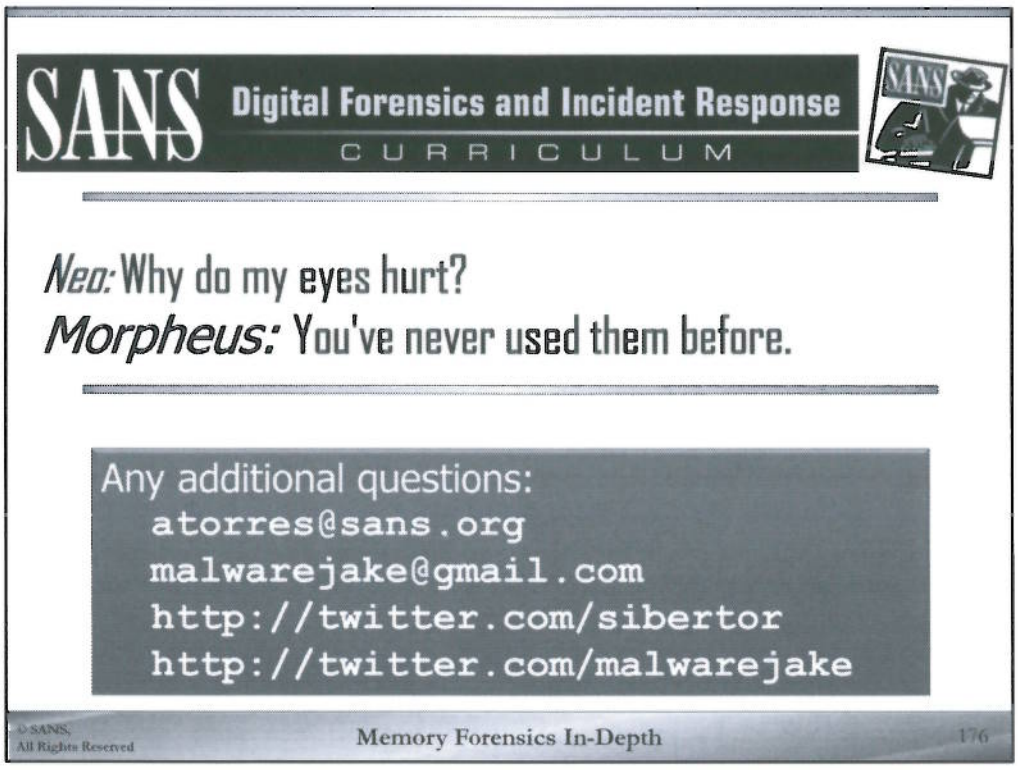
This is sample output from the timeliner plugin run in default mode against the APT.img memory image in your /cases directory. The first line "END LIVE RESPONSE" is pulled from the Imageinfo Creation Time/Datestamp in UTC. A list of processes is generated from



This page intentionally left blank.



This page intentionally left blank.



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