SANS

www.sans.org

FORENSICS 526
MEMORY FORENSICS
IN-DEPTH

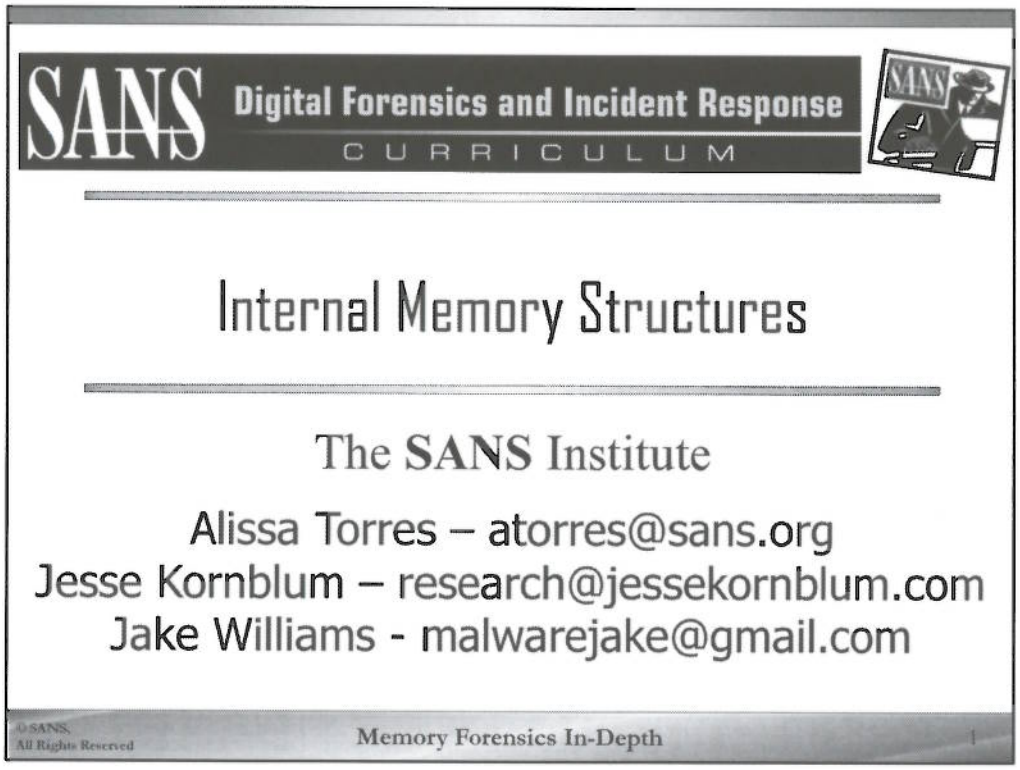
526.4
526.5

Internal Memory Structures
(Parts I & II)

Memory Analysis on Platforms Other than Windows

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

For526_4_5_A06_01



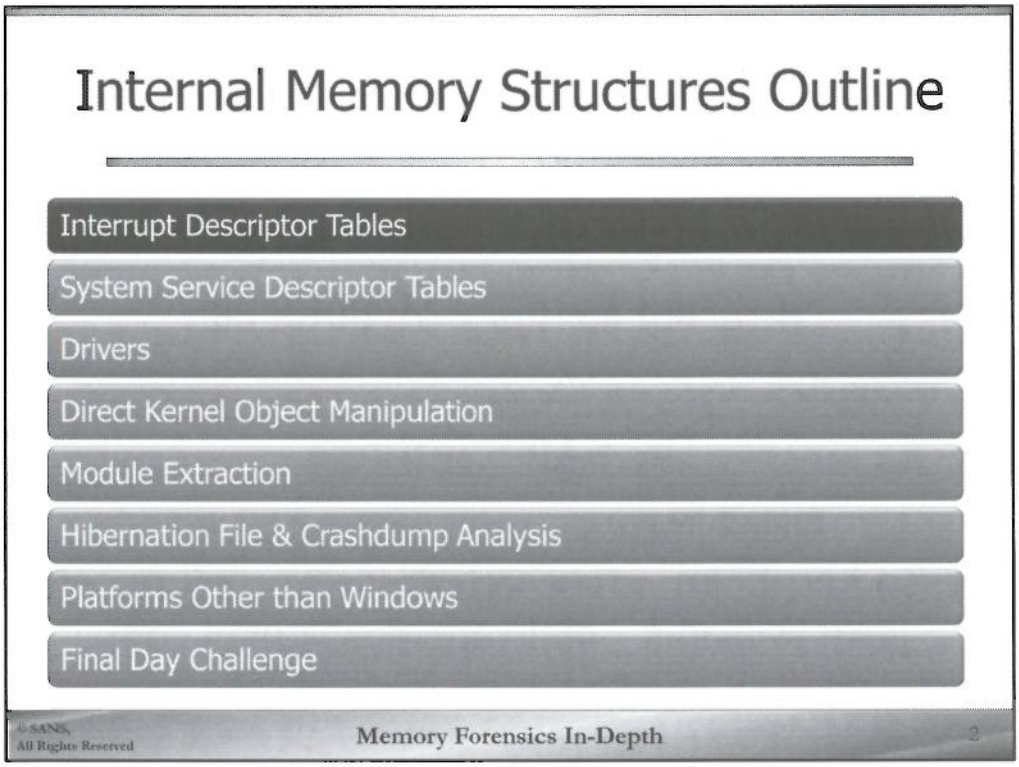
Welcome to Internal Memory Structures.

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

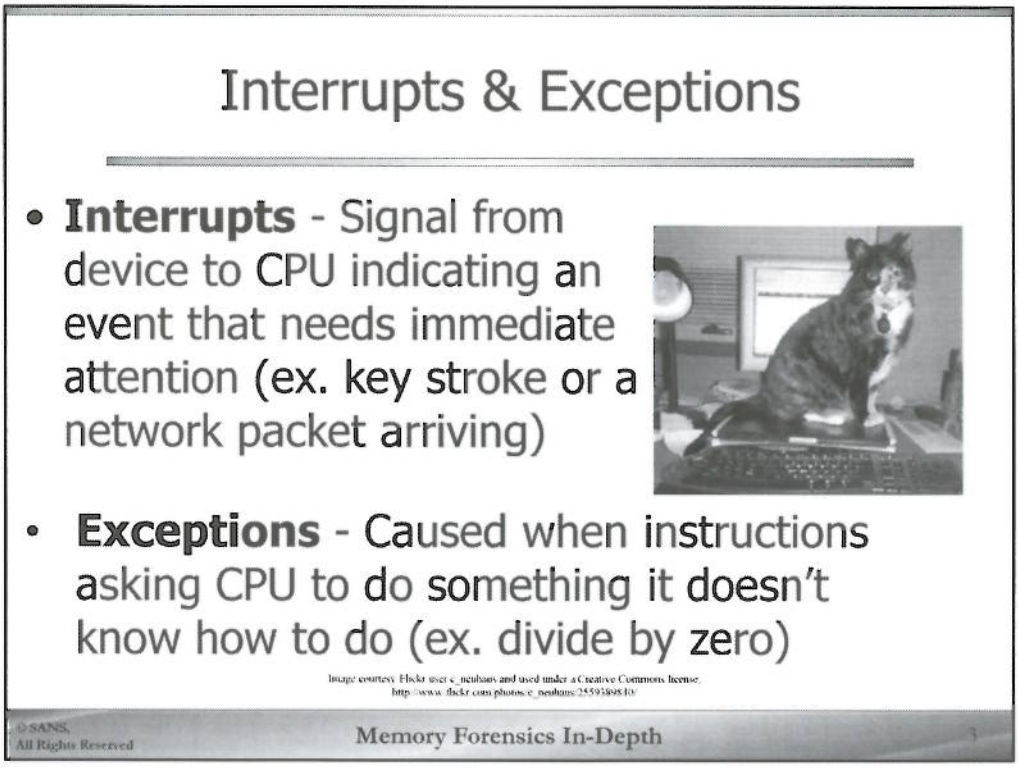
Jesse Kornblum – research@jessekornblum.com
http://twitter.com/jessekornblum

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

http://twitter.com/sansforensics

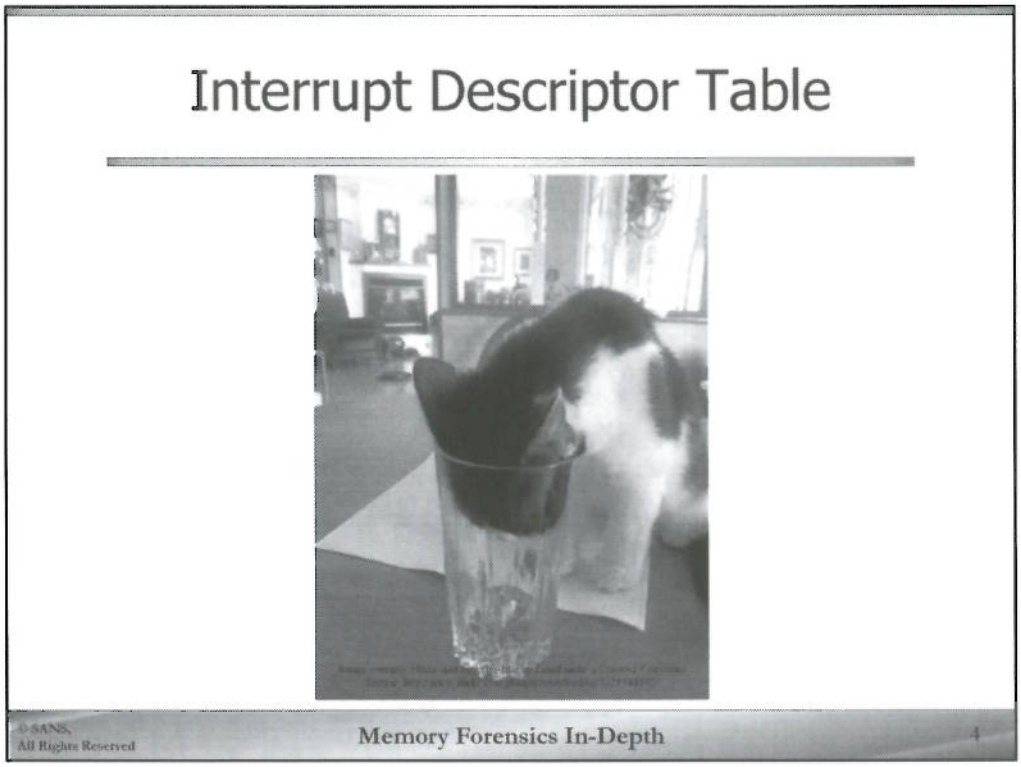


This page intentionally left blank.



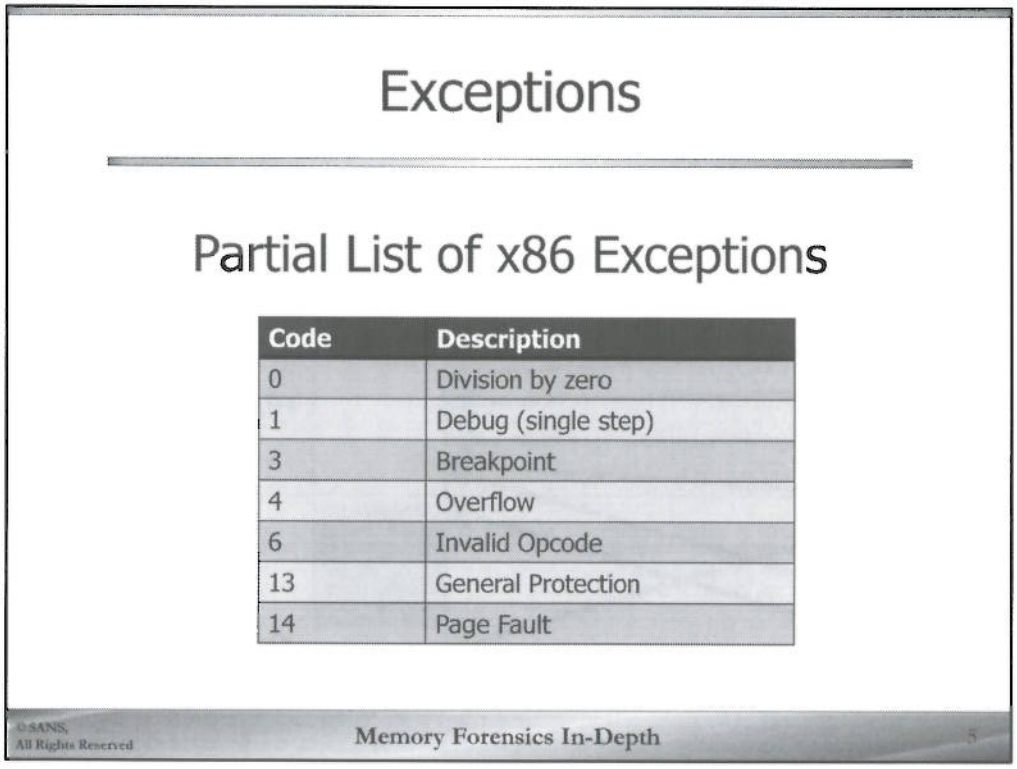
No matter what you plan on doing today, there will be interruptions. The same thing happens with your computer's operating system. Along with the normal flow of processing instructions, there are constant interruptions to the computer's activity. In this section we are going to discuss some of those interruptions, how the processor and operating system normally handle them, and how programs can affect the process, both legitimately and otherwise.

We are also going to cover how the operating system deals with calls from userland programs to services provided by the operating system. Although different from interrupts, the same kind of mechanisms, and subversions, are possible with system services.



The Interrupt Descriptor Table (IDT) is used to handle both interrupts and exceptions. Interrupts are things which happen externally to the processor, such as the user hitting a key on the keyboard or a network packet arriving. Exceptions are things which happen internally to the processor, such as an instruction to divide by zero. Exceptions can be repeated by executing the same sequence of instructions, but interrupts cannot. In either case, the operating system has to stop what it's doing--literally, interrupt it—save the current state, handle the interrupt, and then resume normal operation.

When an interrupt occurs, the processor looks up the function which will handle it using the IDT. The physical address of the IDT is stored in a processor register called idtr. Each entry in the IDT points to a function for handling the interrupt. Let's use the example of the user hitting a key on the keyboard. When this happens the keyboard sends a signal to the computer on I/O bus. A chip listening on the bus receives the signal and interrupts the processor. The processor then has to have the operating system figure out how to handle this keystroke. The processor could have any valid thread as the active thread at that instant. The processor saves the state of the current thread, looks up the interrupt number in the table and finds the appropriate function to handle this interrupt. If there is no function defined for this interrupt, the processor halts (crashes). The function referenced in IDT, in this case, would examine the keystroke (Was it CTRL-ALT_DEL? An ordinary letter?), figure out which process should receive that keystroke (which window had 'focus'?), and pass the input along to that program ("Hey notepad.exe, the user typed letter 'a').



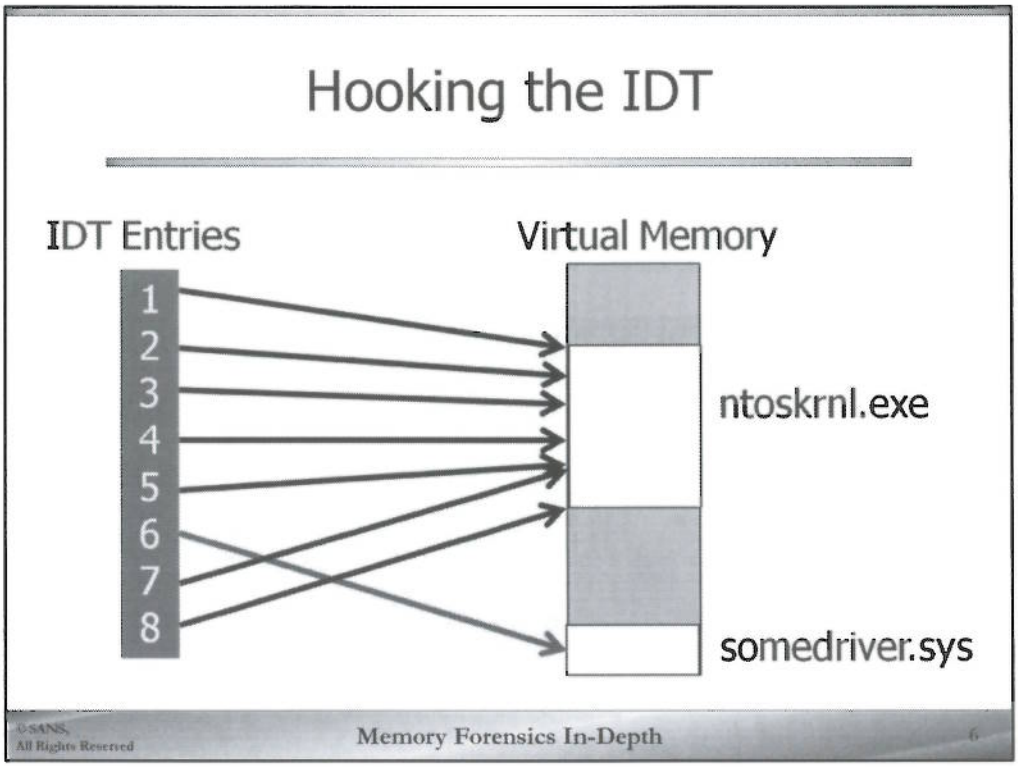
Exceptions are caused by software instructions which ask the processor to do things it doesn't know how to complete. For example, if you ask the processor to divide by zero, there is simply no right answer*. Like interrupts, exceptions are handled by looking up a handling function in the IDT. The first 32 entries at the start of the IDT are reserved by the processor for exceptions. That is, they are dictated by the x86 specification. A partial list of them is shown above.

The exceptions for "single step debugging" and a breakpoint are used for debugging a program. The former is used for single-stepping through a program. The user can have the processor execute a single instruction and then stop, returning control of the processor to the debugging program. The user can then view the effect of that single instruction and decide what to do next—single step again, change something, or simply continue execution unimpeded. It's a wonderful technique for seeing exactly what your program is doing. Likewise, when debugging a program the user can set a breakpoint in the code. The debugger will execute instructions up to the breakpoint, and then, as above, return control to the debugging program.

Some exceptions can be handled gracefully. For example, there is an exception generated when a page of virtual memory is unavailable. This condition, called a page fault, means that Windows must do the work necessary to make that page valid. The page may be backed by the disk—for example, if the processor is seeking part of an executable. Or it may be stored in the paging file. The processor doesn't know where that data is stored. Only Windows knows where the data is stored, and thus Windows has to make the page valid again.

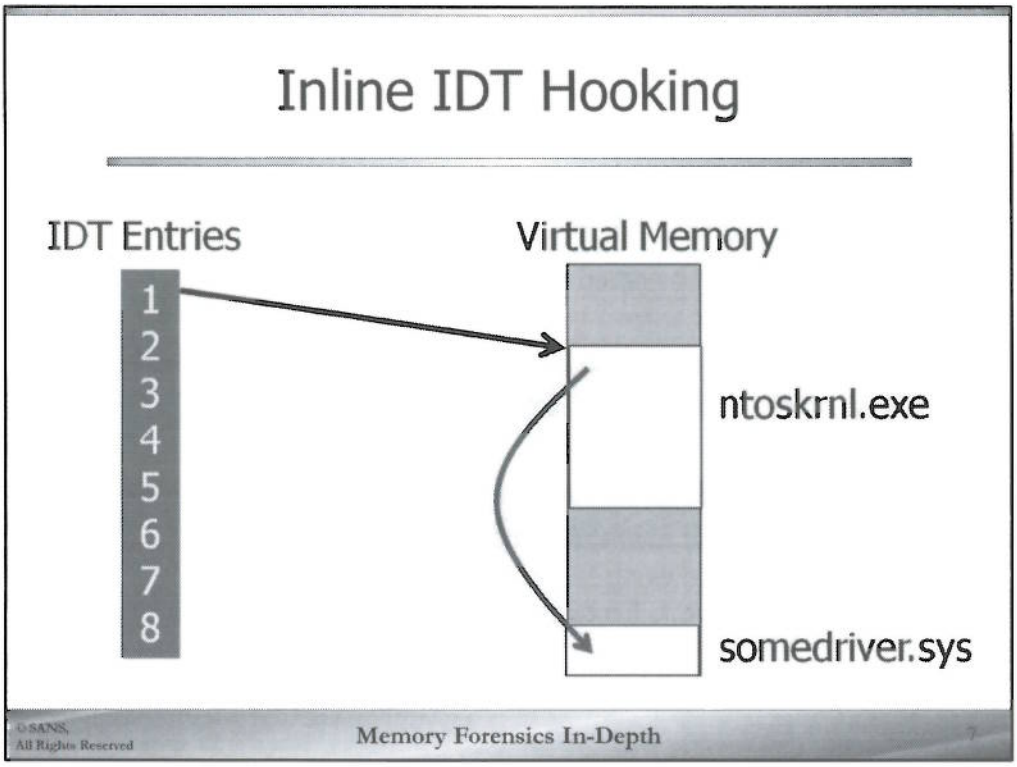
The list above is incomplete. There are many references for the full list of x86 exceptions, including Microsoft's, http://support.microsoft.com/kb/117389.

* Officially mathematicians will tell you that the result is "undefined," which they consider to be valid and correct. Like most results obtained from mathematicians, however, it is useless if you are trying to get something done in real life. This is why we keep mathematicians confined to academia and let engineers get on with building skyscrapers.



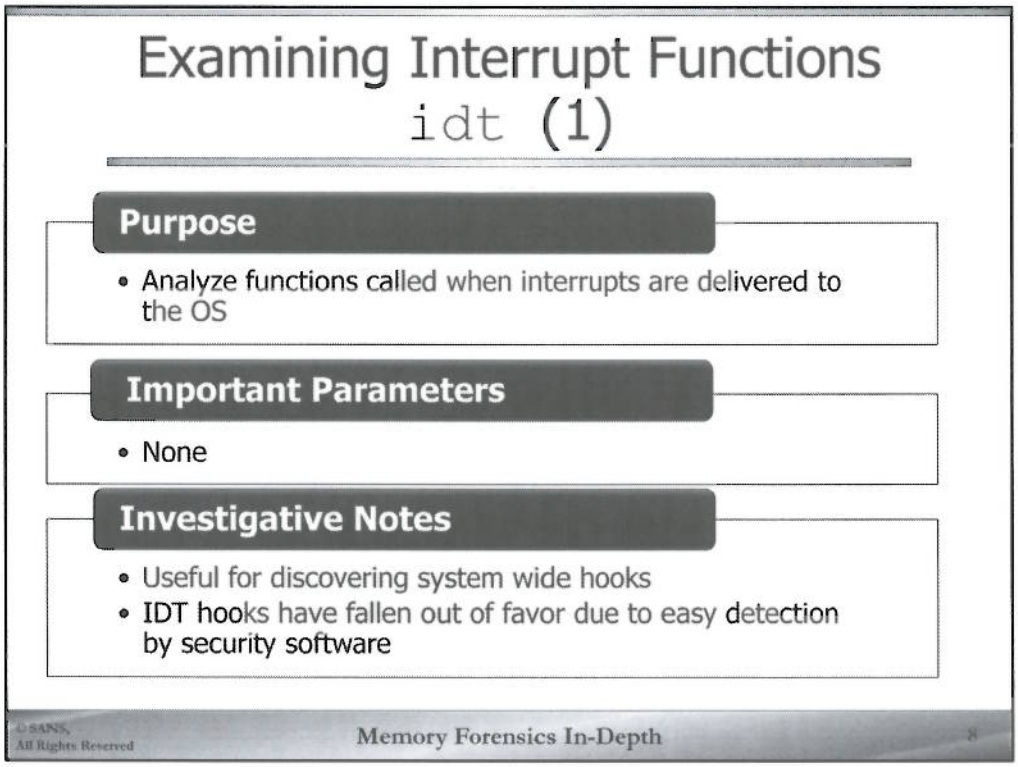
A malicious program could hook an entry in the IDT. When the interrupt occurs or the exception is generated, the processor would execute the new function pointed to by the IDT entry. This would be done, for example, to capture all of the keys typed on the keyboard, or all of the network packets received by the host.

Of course, changing a value in the IDT would be easily detectable. Every entry in the IDT should point into virtual address space of the Windows kernel, ntoskrnl.exe. If an entry did not point into the ntoskrnl.exe address space, we could immediately flag it as being suspicious.



To avoid being detected as hooking the IDT directly, malicious programmers can hook the IDT in-line. With an in-line hook, the malicious program does not modify the IDT itself. Instead it modifies the function which is referenced by the IDT. Usually this is in the form of adding a jump from that function to their own code. Again, this kind of technique can be detected. The code in ntoskrnl.exe should not jump into the code in any other module. It's more subtle, and thus requires more work to detect it, but still can be automated.

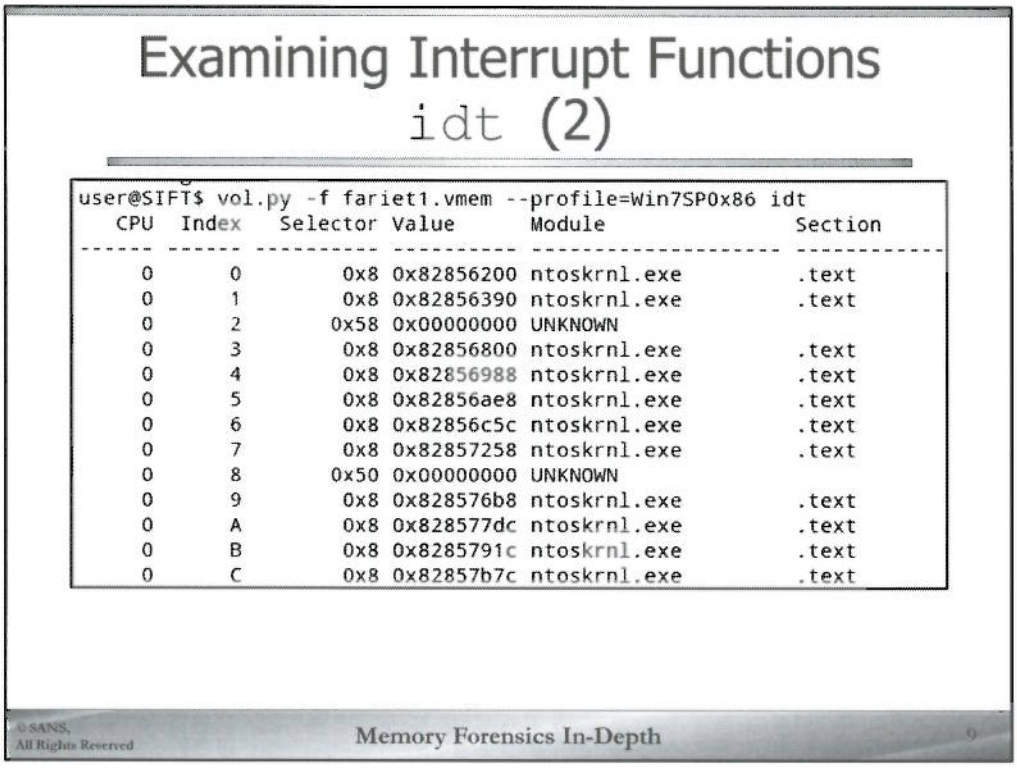
We're not going to dump out the IDT and have you look through it by hand, however. Instead we're going to do a hands-on exercise in the next section where we parse the IDT automatically and flag any suspicious values.



The idt plugin is used to enumerate the entries of the Interrupt Descriptor Table (IDT). The IDT is a table, maintained by the OS (and referenced by the CPU) that contains the addresses of functions to call when interrupts are delivered to the OS by the CPU. Historically, the IDT has been a favorite target of attackers due to the systemwide implications of a single hook. However, more recently hooking the IDT has fallen out of favor as hooks there are very easily detected by security software.

The idt plugin can detect hooks where pointers are replaced in the table itself. However, it is not useful in detecting inline hooks where code is replaced in the target module (such as ntoskrnl.exe). Most entries in the IDT should point to either ntoskrnl.exe or hal.dll.

The idt plugin is implemented in volatility/plugins/malware/idt.py

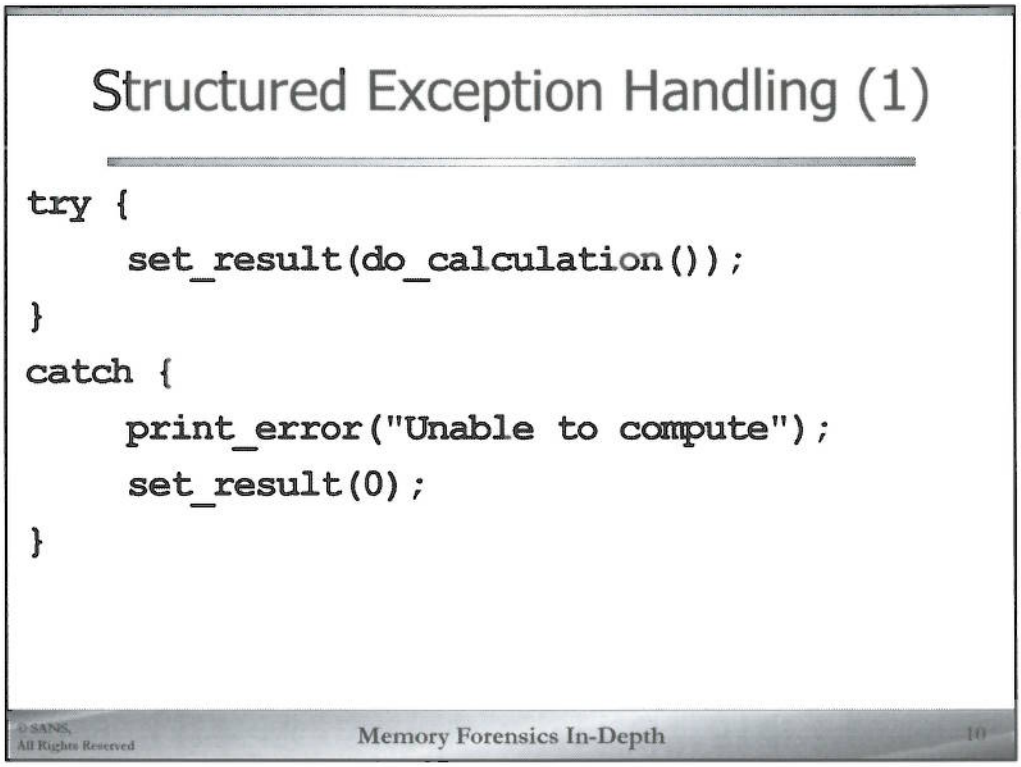


This slide shows the output of the idt plugin. Note that some entries (0x2, 0x8) have a module UNKNOWN. At first glance this might seem suspicious, however, note the address (Value field) of the entries marked UNKNOWN. They are all zeroes! This explains why volatility is unable to detect the module, there isn't one!

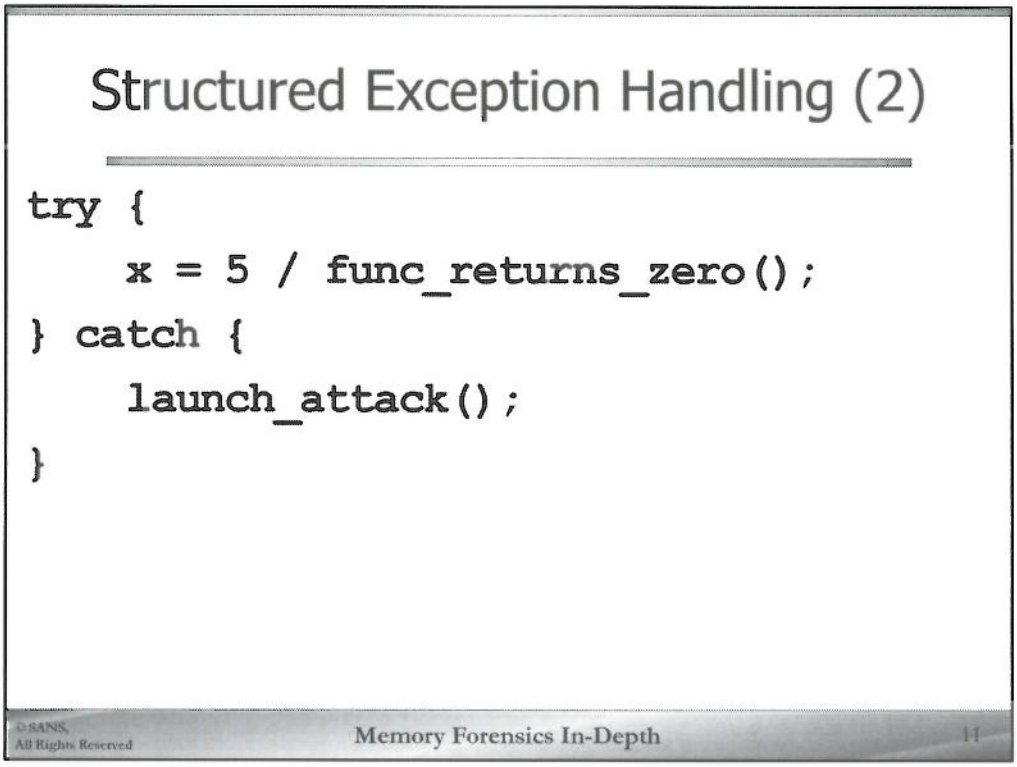
user@SIFT$ vol.py -f fariet1.vmem --profile=Win7SP0x86 idt

```
  CPU Index  Selector Value      Module               Section
------ ------ ---------- ---------- -------------------- ------------
     0      0        0x8 0x82856200 ntoskrnl.exe         .text
     0      1        0x8 0x82856390 ntoskrnl.exe         .text
     0      2       0x58 0x00000000 UNKNOWN
     0      3        0x8 0x82856800 ntoskrnl.exe         .text
     0      4        0x8 0x82856988 ntoskrnl.exe         .text
     0      5        0x8 0x82856ae8 ntoskrnl.exe         .text
     0      6        0x8 0x82856c5c ntoskrnl.exe         .text
     0      7        0x8 0x82857258 ntoskrnl.exe         .text
```

… output truncated …



Windows also offers Structured Exception Handling (SEH). With this technique, programmers can create a block of code which they try to execute. If an exception is generated while executing the block, it can be caught by an exception handler written into the program. This is a great technique for legitimate programmers. It helps them catch errors and handle them gracefully without the program crashing. For example, let's say the program calc.exe, a graphical calculator program, is attempting to do division. The user inputs five divided by zero. Our programmer could wrap the block which does the work of this division operation in a try statement. Immediately following that block is an exception handler, which tells the computer, that if an exception is generated, to catch it and display the value "ERROR" as the result. (You could argue that it might be better to check the arguments being passed to the processor to calculate, but this could get messy. For example, if the user submits (4 * 2) divided by (6^2 - 36). Might just be easier to pass the whole thing to the processor can catch any exceptions, regardless of how they're generated.)



As with all technologies, however, exception handling can be used maliciously. As you saw on the last page, the code to handle an exception is generally written, in the source code, immediately after the code which could generate the exception. When compiled, however, the code to handle exceptions is found in another part of the program. With SEH, the exceptions are still handled using the IDT. Eventually control is transferred back to the program, but not at the same location where it left.

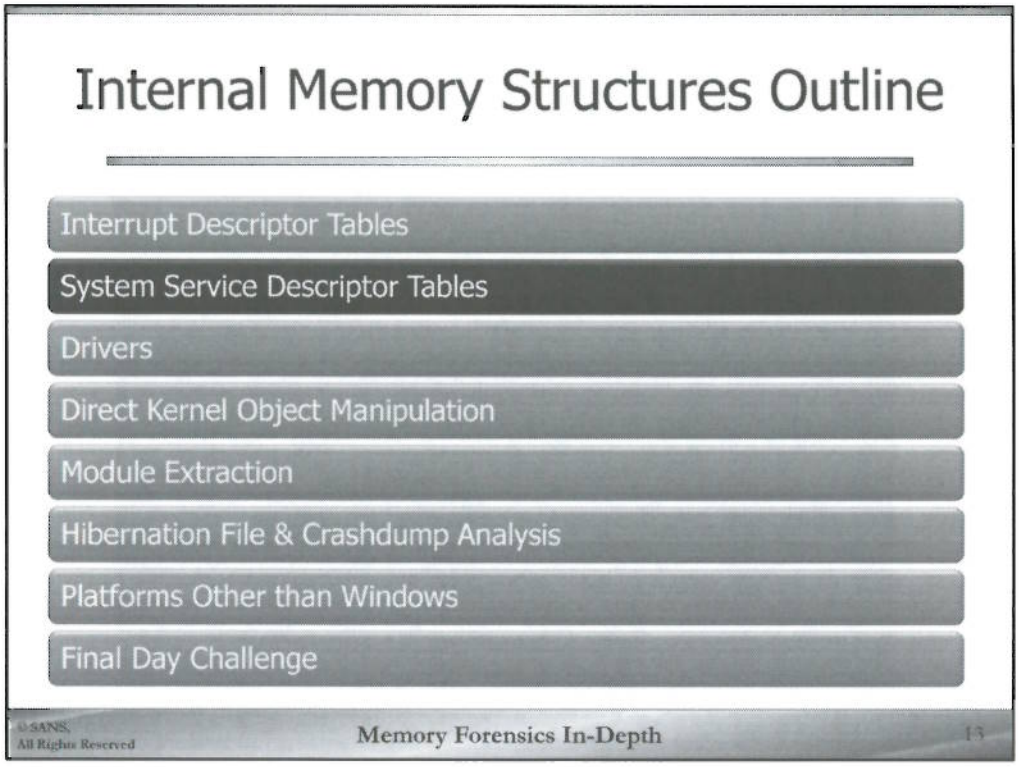
Malicious programmers can use this behavior to frustrate reverse engineering. During RE, the user is attempting to determine the program's flow. They can set up a condition which reliably generates an exception, and then use the exception handler to hide some part of what they're doing. For example, here's a bit of pseudo code:

```
initialize();
try {
    x = 5 / func_returns_zero();
} catch {
    launch_attack();
}
cleanup();
```

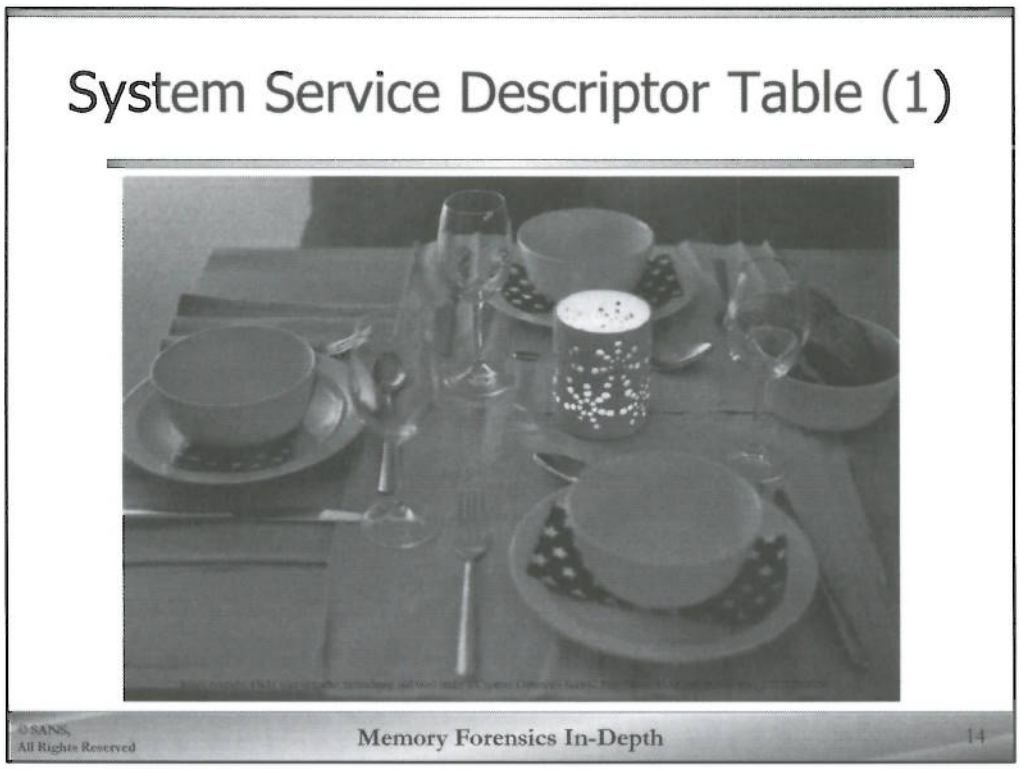
When this code is compiled, the attack code may not be anywhere near the main flow of the program's execution, which could look like this, in pseudo-assembly-code:

```
CALL initialize
CALL function_which_returns_zero
PUSH 5
PUSH result
CALL divide
CALL cleanup
```

The code to launch the attack will be elsewhere in the program. The RE will have to take the extra time to figure out what each of these functions does, that an exception is generated, track down the SEH functions, and *then* reverse engineer the attack. Anything which slows downs or befuddles the reverse engineer helps the attacker.

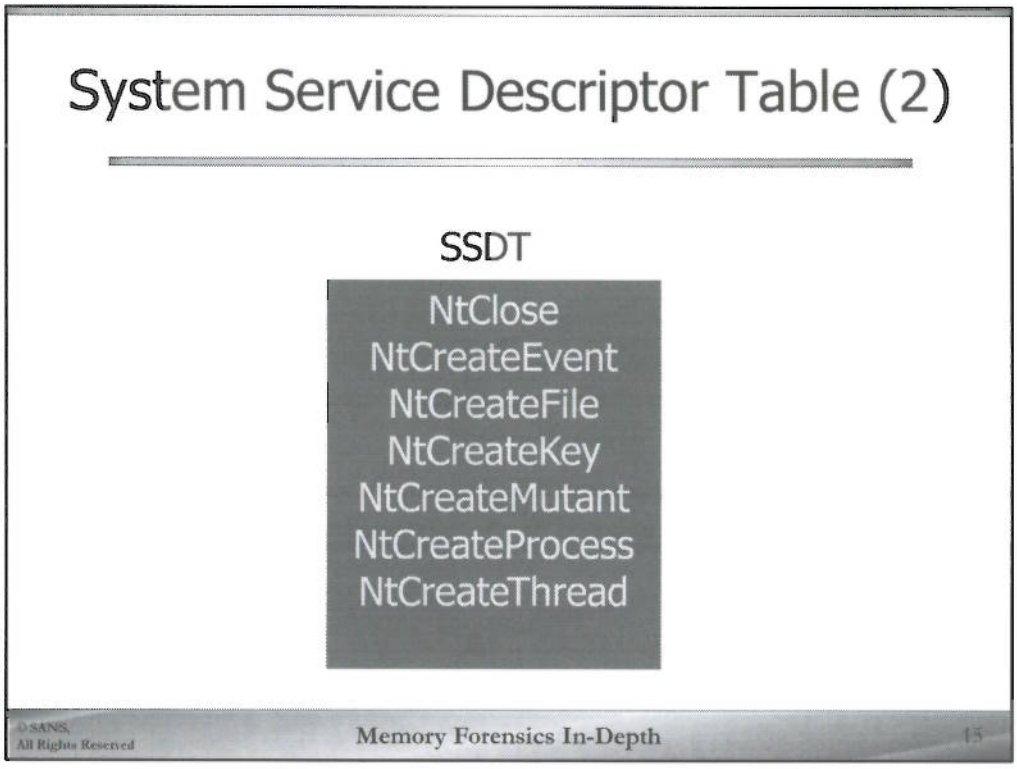


This page intentionally left blank.



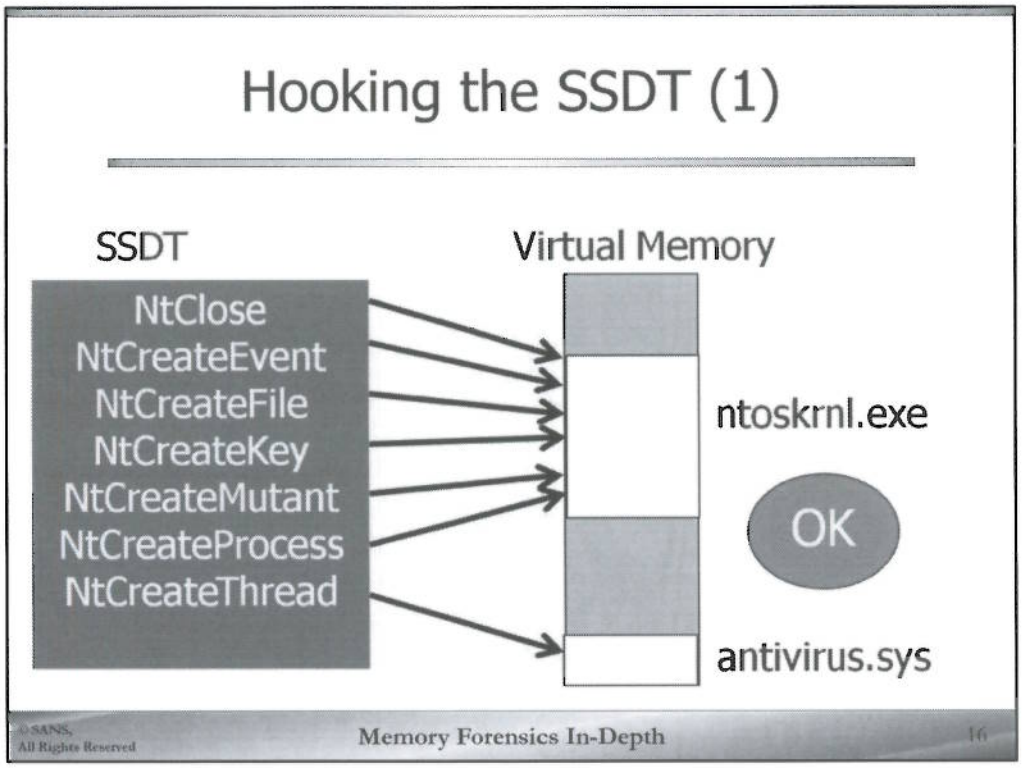
The System Service Descriptor Table (SSDT) is used to look up the addresses of functions for kernel support to userland function calls. The operating system provides a number of basic services to programs which require kernel level code to execute. For example, when the user requests to open a file using the API function CreateFile, the operating system must eventually create a kernel object, a FILE_OBJECT structure, and add it to the set of active kernel objects. The userland program calls the API function CreateFile in the ntdll.dll, but eventually that DLL must call up a kernel routine to do its work. In this case, the ntdll.dll calls the function NtCreateFile. The 'Nt' prefix denotes that it's part of the kernel but exposed to the outside world. (Other prefixes denote other levels of the kernel.)

On Windows there should be two SSDTs. One of the SSDTs is maintained by the kernel directly, either ntoskrnl.exe or ntkrnlpa.exe. (The latter denotes that PAE was running on the system.) This SSDT contains the basic functions used by the operating system, such as opening and closing files, reading and writing data to files, and so on. The other SSDT is used for the graphical user interface, and contains functions from win32k.sys.



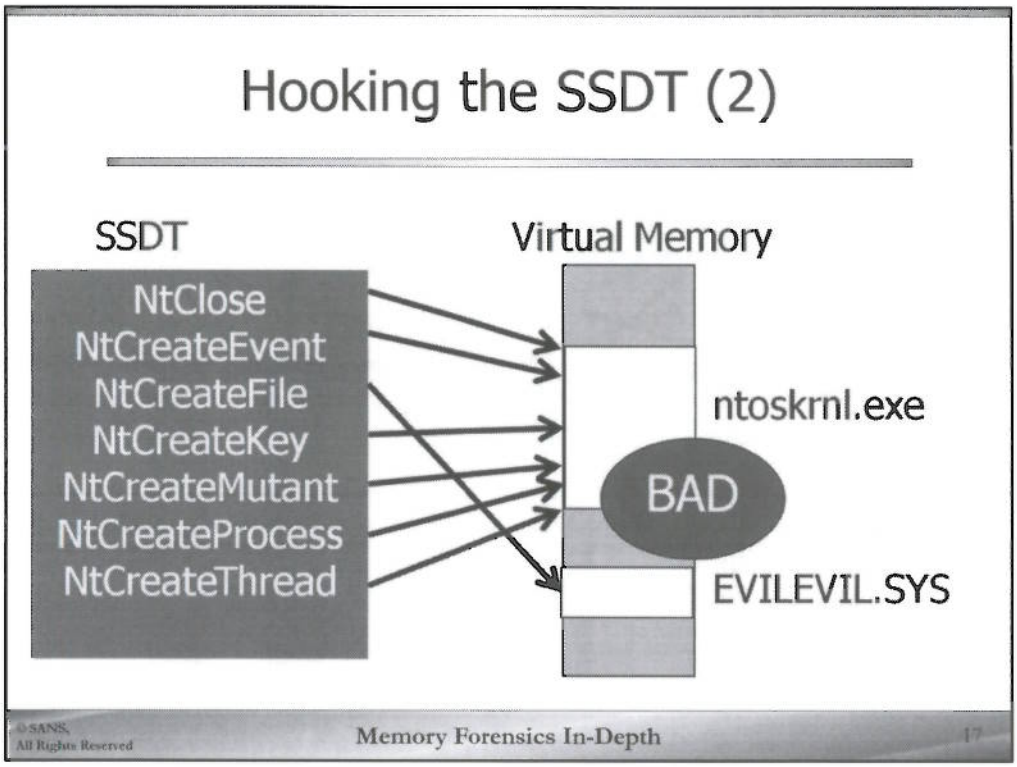
To get the addresses of these kernel functions, the kernel builds a list of functions, called the System Service Descriptor Table (SSDT). When one of these functions is needed, the userland code uses an instruction called SYSENTER. SYSENTER is a special instruction which transfers control of the system from the user program to the operating system. Arguments passed along with the SYSENTER instruction include which system call to execute.

When the SYSENTER instruction is executed, control of the processor is transferred to the operating system. Specifically, control goes to a kernel function, KiSystemService. That function figures out which system call was requested, looks it up in the SSDT, and executes the correct function.



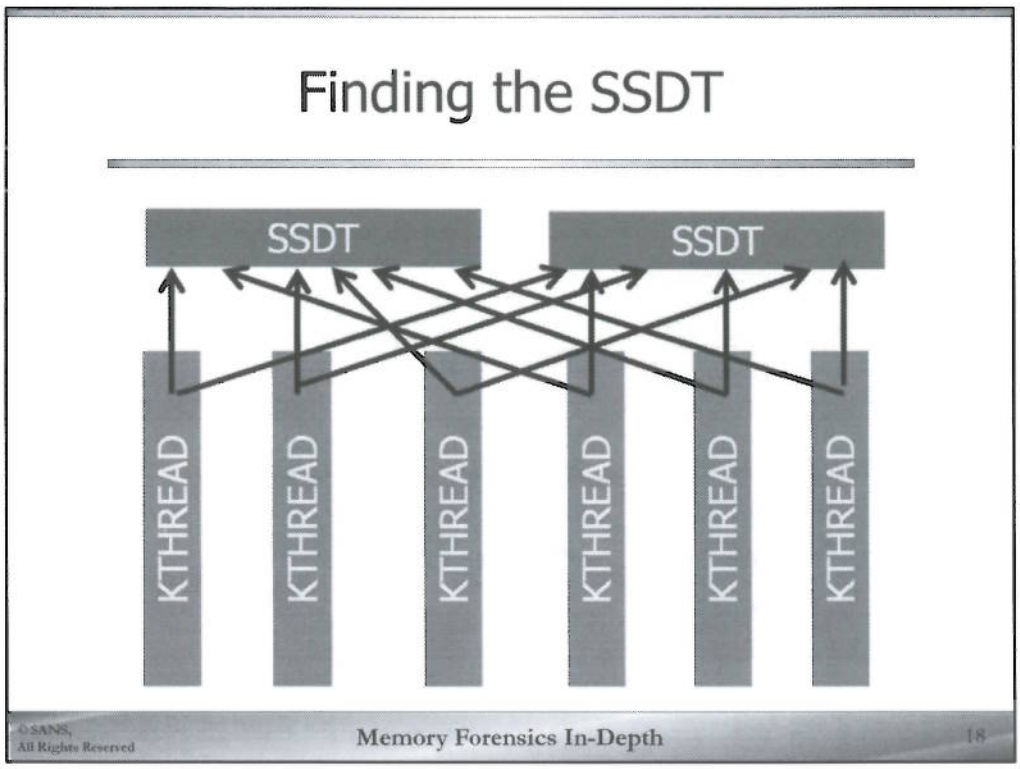
On a default Windows installation, every entry in the SSDT should point into one of the two legitimate modules. These are the kernel, ntoskrnl.exe, or win32k.sys.

Many programs, both legitimate and otherwise, will hook the entries in the SSDT. For example, an anti-virus program might want to hook the system call for starting a thread's execution. The A/V program could examine the code of that thread, compare it to known signatures of malicious software, and so on. Seeing hooks in the SSDT is normal; but be wary of where those hooks go! Each hook will lead to a function in another kernel module. (Well, ok, it *should* lead to a hook in a kernel module. If not, a call to that system call would crash the system.)



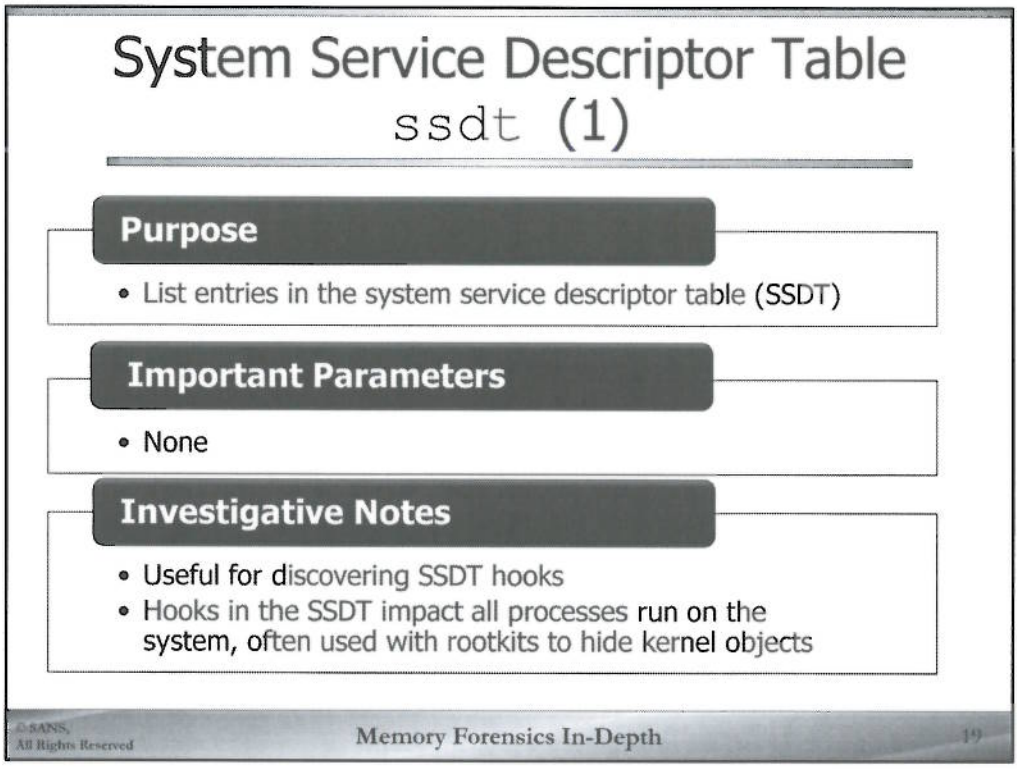
Malicious programmers, of course, would love to hook functions in the SSDT. For example, they could hook the system call to the NtCreateFile function. This function is used to open and read files on the disk. To hide files, a malicious programmer could hook this function in the SSDT. If the user attempted to open one of the malicious programmer's chosen files, they could program the machine to tell the userland program that the file does not exist. For any other request, the malicious code could just call the original function listed in the SSDT.

For example, let's say our programmer creates a function in the EVILEVIL.SYS driver called EvilNtCreateFile. This function takes the same arguments as the legitimate NtCreateFile. They then hook the value in the SSDT for NtCreateFile so that it points to EVILEVIL.SYS:EvilNtCreateFile. When a userland program calls CreateFile, the KiSystemService looks up the value in the SSDT for the address of NtCreateFile. That function finds the address of EVILEVIL.SYS:EvilNtCreateFile and calls it. When EvilNtCreateFile is called, it examines the filename requested to be opened. If that file is under the directory "C:\hidden", or whatever the malware author chooses, the function returns "File Not Found." If the file is not under that directory, this function calls the real NtCreateFile and returns the result of that function. In other words, calls which don't involve the hidden directory are allowed to proceed.



Pointers to the SSDTs are stored in the kernel structures for threads. When a userland program needs to make system calls, it's going to be done by a thread. Pointers to the SSDTs needed are stored in the KTHREAD structure, which is in turn part of the ETHREAD structure.

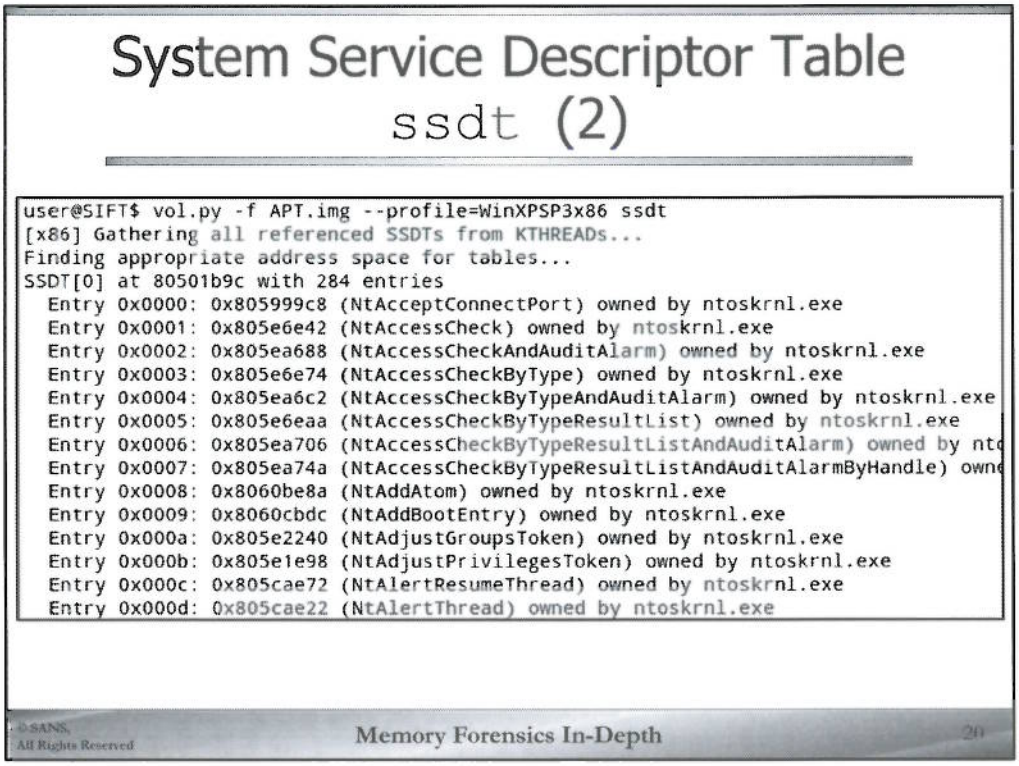
To find the SSDTs during memory forensics, we must examine every thread on the system and look at their KTHREAD structures. Each structure should point to one or both of the SSDTs. (It's possible to have a program which doesn't involve any system calls, but it wouldn't be very useful.)



The ssdt plugin is used to enumerate the address of System Service Dispatch Table (SSDT) functions. These functions are the kernel APIs that are ultimately called when user space applications call APIs. Normally, entries in the SSDT should map to ntoskrnl.exe and win32k.sys. However, sometimes antivirus programs replace some entries in the SSDT to perform monitoring. Because this technique is often used by rootkits, you should carefully examine all SSDT hooks.

It should be noted that malware authors have previously implemented hooks using drivers with legitimate sounding names (particularly those of security software). Always cross reference the module name with the path on disk to ensure that the hooking module is legit.

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



The following shows the output of the ssdt plugin.

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

[x86] Gathering all referenced SSDTs from KTHREADs...

Finding appropriate address space for tables...

SSDT[0] at 80501b9c with 284 entries

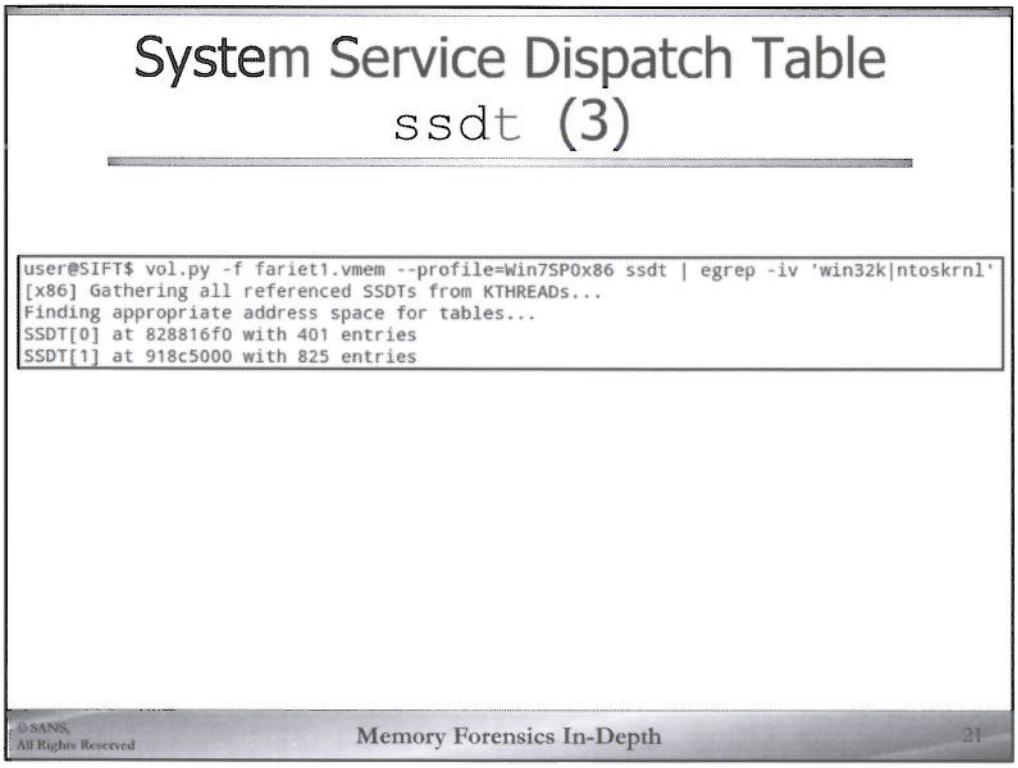
Entry 0x0000: 0x805999c8 (NtAcceptConnectPort) owned by ntoskrnl.exe

Entry 0x0001: 0x805e6e42 (NtAccessCheck) owned by ntoskrnl.exe

Entry 0x0002: 0x805ea688 (NtAccessCheckAndAuditAlarm) owned by ntoskrnl.exe

Entry 0x0003: 0x805e6e74 (NtAccessCheckByType) owned by ntoskrnl.exe

… output truncated …



Because entries in the SSDT should be in ntoskrnl.exe or win32k.sys, a common technique is to grep out these entries (as shown below). Note in this case that output has not been truncated. There are no entries in either SSDT that are not handled by ntoskrnl.exe or win32k.sys.

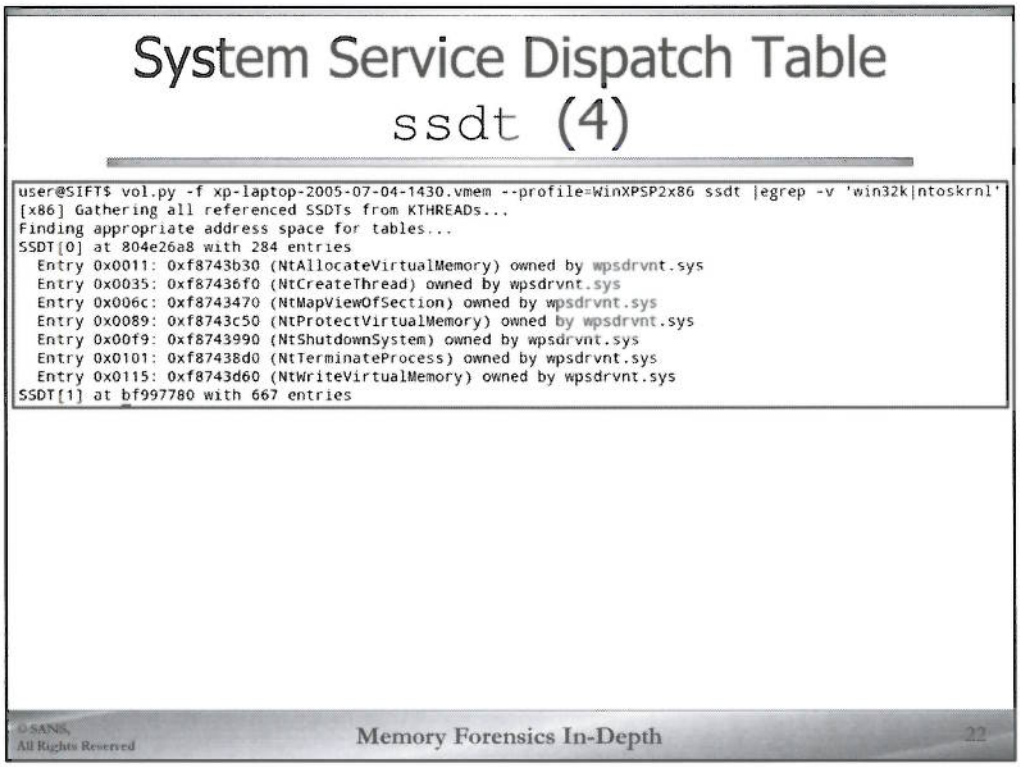
user@SIFT$ vol.py -f fariet1.vmem --profile=Win7SP0x86 ssdt | egrep -iv 'win32k|ntoskrnl'

[x86] Gathering all referenced SSDTs from KTHREADs...

Finding appropriate address space for tables...

SSDT[0] at 828816f0 with 401 entries

SSDT[1] at 918c5000 with 825 entries

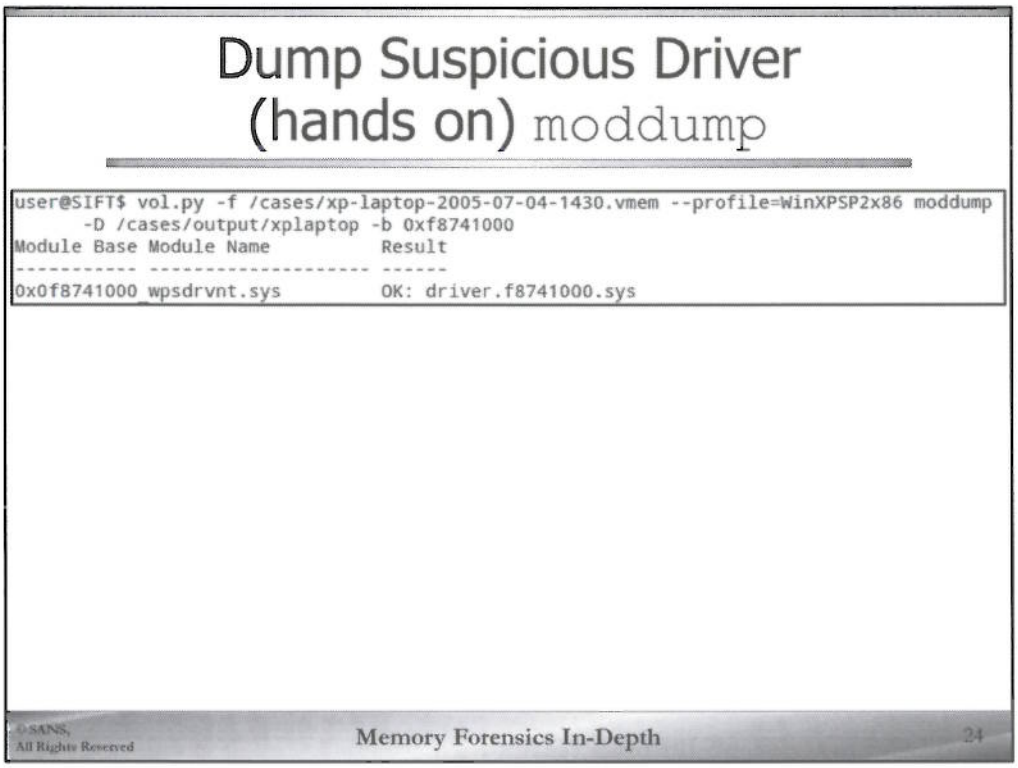


Now let's look at a memory image which has hooked functions, but hooked by a legitimate process. The XP laptop memory image has some hooked functions from an anti-virus program. Let's see what that looks like in the SSDT. We are going to immediately use the grep command to hide any entries which contain ntoskrnl or win32k. We're just looking for the entries which point outside of those modules—the entries which have been hooked.

Here's the command line:

```
user@SIFT$ vol.py -f xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
ssdt |egrep -v 'win32k|ntoskrnl'
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 804e26a8 with 284 entries
  Entry 0x0011: 0xf8743b30 (NtAllocateVirtualMemory) owned by wpsdrvnt.sys
  Entry 0x0035: 0xf87436f0 (NtCreateThread) owned by wpsdrvnt.sys
  Entry 0x006c: 0xf8743470 (NtMapViewOfSection) owned by wpsdrvnt.sys
  Entry 0x0089: 0xf8743c50 (NtProtectVirtualMemory) owned by wpsdrvnt.sys
  Entry 0x00f9: 0xf8743990 (NtShutdownSystem) owned by wpsdrvnt.sys
  Entry 0x0101: 0xf87438d0 (NtTerminateProcess) owned by wpsdrvnt.sys
  Entry 0x0115: 0xf8743d60 (NtWriteVirtualMemory) owned by wpsdrvnt.sys
SSDT[1] at bf997780 with 667 entries
```

There are seven functions which have been hooked, all by the module wpsdrvnt.sys. If that module name is familiar to you, that's great, but remember that just because the driver has the name of a legitimate driver, it doesn't make it legitimate! We can do two kinds of analysis on the driver names we get from the SSDT. First, we can search the web (or any other resource you have) for background on the name. If the name is associated with something automatically evil (e.g. EVILEVIL.SYS), you've found something suspicious. If the name is associated with something legitimate, you can test whether or not this driver actually is that legitimate thing. The second type of analysis is to dump out a sample of the driver and examine it yourself. You can run strings on it, submit it to Virus Total, reverse engineer it with IDA Pro, etc., etc.



Take a minute to search the web for the name "wpsdrvnt.sys". What do you find?

Hopefully you should find that the file wpsdrvnt.sys was associated with the Sygate Personal Firewall. That gives us a target to look for. The sample we're about to recover should match a legitimate firewall driver.

Let's grab a sample of this driver using the moddump command. To use that command, however, we'll need to know the module's base address in memory. We can get that using the modules plugin:

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

Which gives us the following output, with the header included as well:

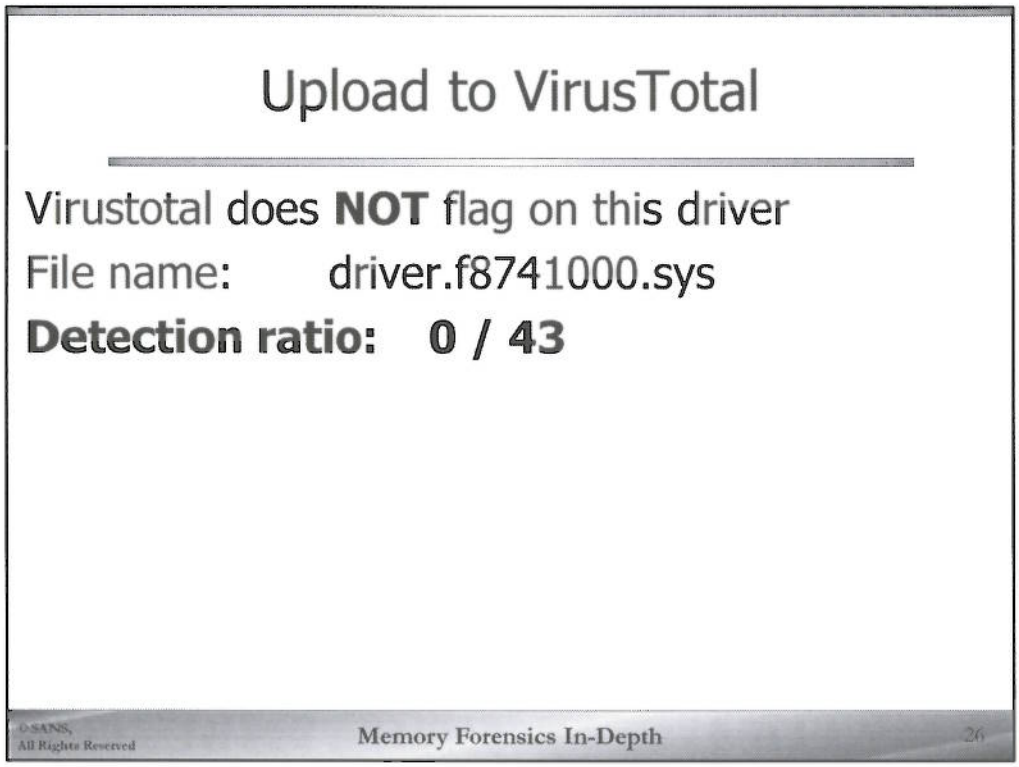
```
Offset(V)          File                                           Base         Size       Name
0x820d3398  \??\C:\WINDOWS\System32\drivers\wpsdrvnt.sys 0x00f8741000 0x009000  wpsdrvnt.sys
```

As a reminder, the values we are seeing are the virtual address of where information about the module is stored, 0x820d3398, then the base address of where the actual module is stored, 0xf8741000. When we call moddump, we need the base address, or 0xf8741000.

Here's the command line to dump the module:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
moddump --dump-dir=/cases/output/xplaptop -b 0xf8741000
```

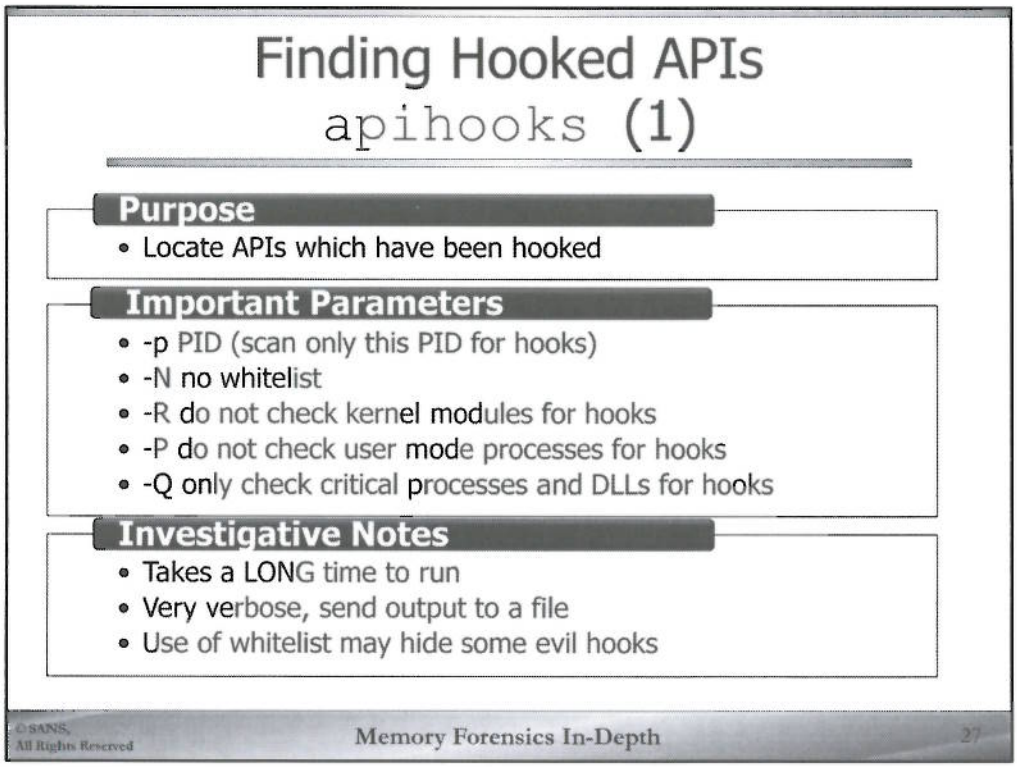
This will create the file driver.f8741000.sys, which we can examine.



Go ahead and submit this file to VirusTotal. Run strings on it. It certainly has the names of several kernel functions and system processes in there. This is a kernel driver, so it makes sense it would need these kinds of functions.

You can see the output of the VirusTotal scan at https://www.virustotal.com/file/ad49f1bedc94b0fcb3958d17f41c2db8976f2215972b930c8f4f6f0b11aeb92e/analysis/. You should find that zero of the VirusTotal engines report this file as being suspicious.

Of course, a negative result of a VirusTotal scan doesn't mean a file isn't malicious. But it is a data point in the analysis. (Especially when the memory image in question is so old. Presumably any malware from 2005 would be known by now!)



The apihooks plugin is used to enumerate hooked API functions. The plugin accepts the following options:

-p PID, --pid=PID Operate on these Process IDs (comma-separated)

-N, --no-whitelist No whitelist (show all hooks, can be verbose)

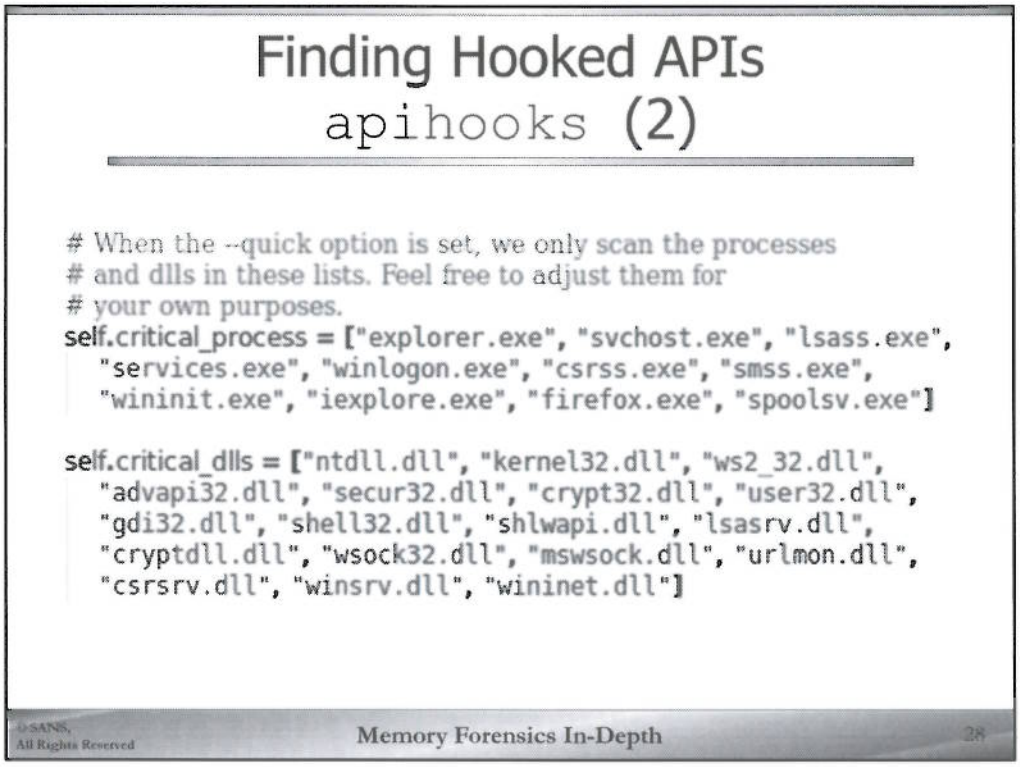
-R, --skip-kernel Skip kernel mode checks

-P, --skip-process Skip process checks

-Q, --quick Work faster by only analyzing critical processes and dlls

The plugin uses a whitelist (embedded in the apihooks.py code) to ignore certain hooks. This is probably what you want, as the whitelist is the result of a wealth of experience of DFIR pros. However, malware might hook some function and obscure it using the whitelist. For this reason, the –N option is offered. However, be advised that the apihooks module output is normally very verbose. With the –N option, the output is almost unusable (except by Windows internals experts), even on a clean system.

The apihooks plugin is implemented in volatility/plugins/malware/apihooks.py



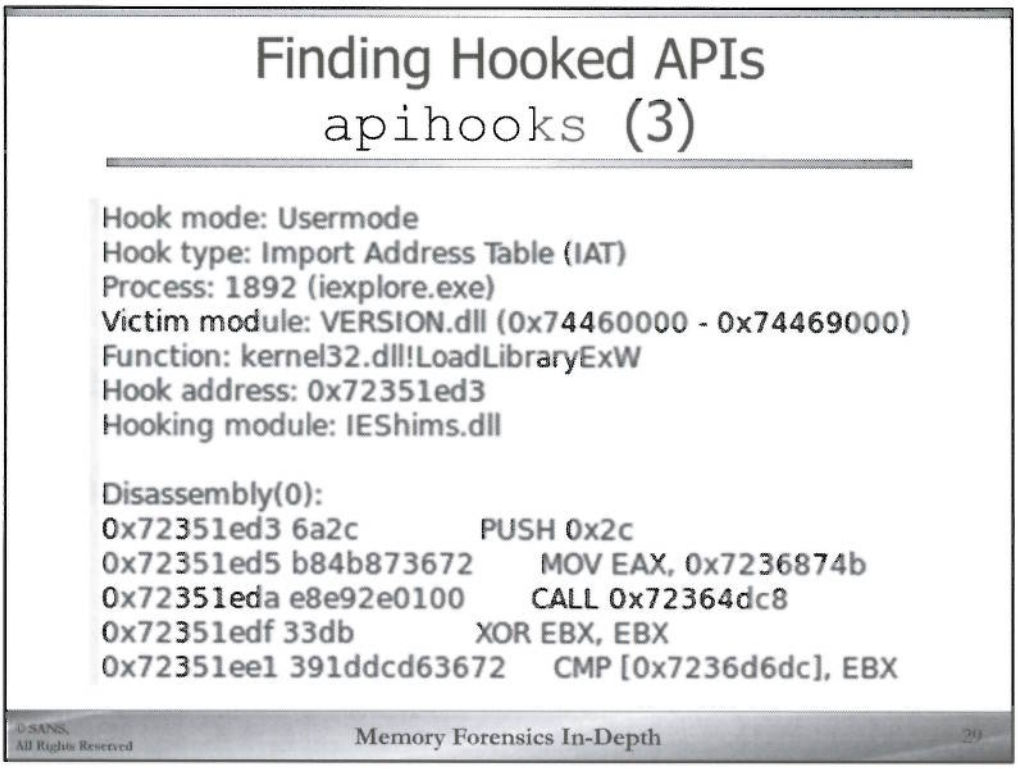
As of Volatlity 2.3.1, the DLLs and processes deemed critical to check for hooks are listed below. These are used with the –Q flag to perform quick checks of only those modules listed (rather than all processes and DLLs on the system). To change this list, you must perform a source code modification. The relevant portions to change are shown in the slide. Simply add the process or DLL name to the appropriate list (don't forget to enclose the name in quotes and add a comma).

Critical processes for apihooks (used with the –Q option):

explorer.exe, svchost.exe, lsass.exe, services.exe, winlogon.exe, csrss.exe, smss.exe, wininit.exe, explore.exe, firefox.exe, spoolsv.exe

Critical DLLs for apihooks (used with the –Q option):

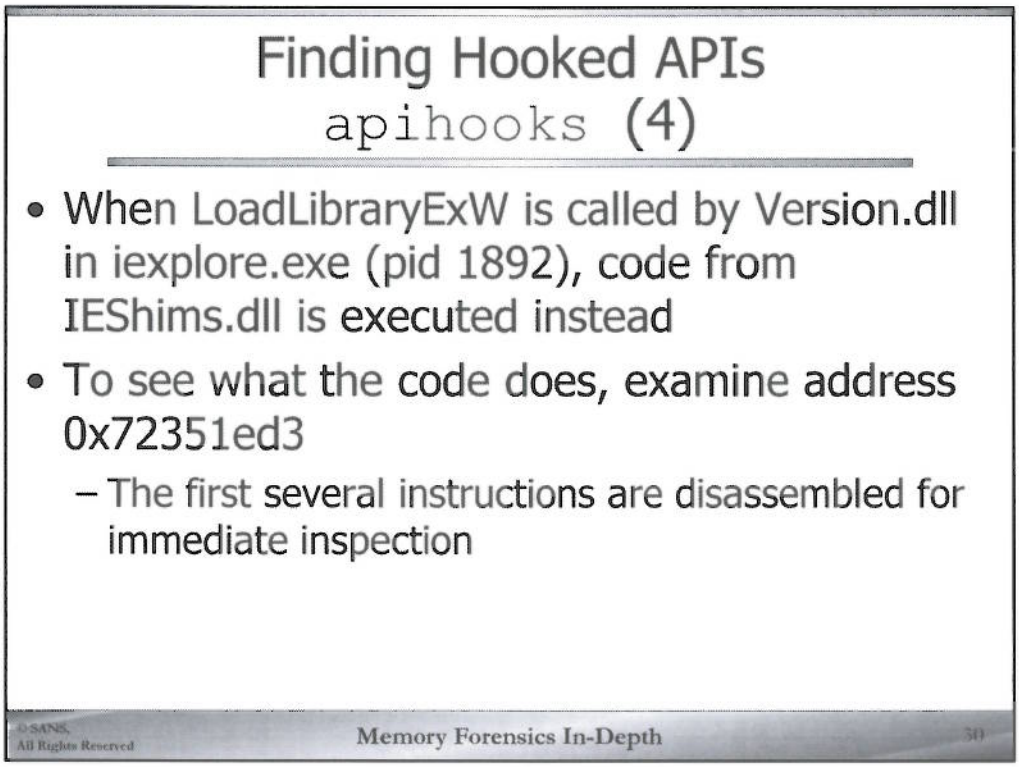
'ntdll.dll, kernel32.dll, ws2_32.dll, advapi32.dll, secur32.dll, crypt32.dll, user32.dll, gdi32.dll, shell32.dll, shlwapi.dll, lsasrv.dll, cryptdll.dll, wsock32.dll, mswsock.dll, urlmon.dll, csrsrv.dll, winsrv.dll, wininet.dll'



This slide shows sample output from the apihooks plugin.

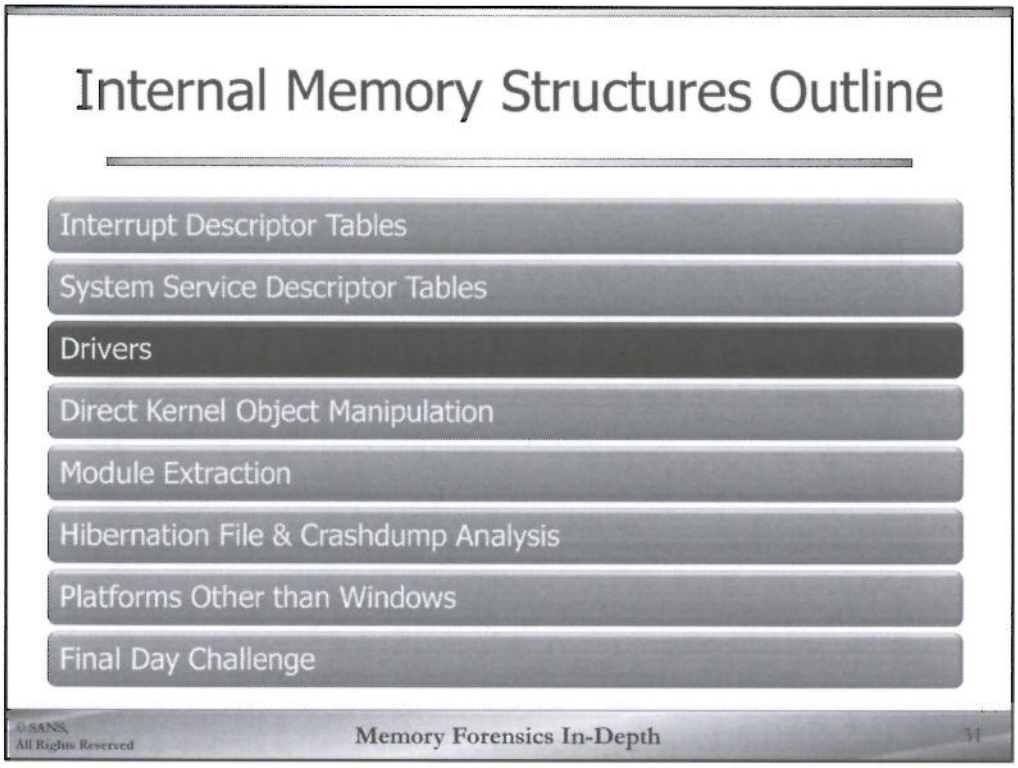
Hook mode: Usermode
Hook type: Import Address Table (IAT)
Process: 1892 (iexplore.exe)
Victim module: VERSION.dll (0x74460000 - 0x74469000)
Function: kernel32.dll!LoadLibraryExW
Hook address: 0x72351ed3
Hooking module: IEShims.dll

Disassembly(0):
0x72351ed3 6a2c PUSH 0x2c
0x72351ed5 b84b873672 MOV EAX, 0x7236874b
0x72351eda e8e92e0100 CALL 0x72364dc8
0x72351edf 33db XOR EBX, EBX
0x72351ee1 391ddcd63672 CMP [0x7236d6dc], EBX
0x72351ee7 7423 JZ 0x72351f0c
0x72351ee9 ff DB 0xff
0x72351eea 35 DB 0x35

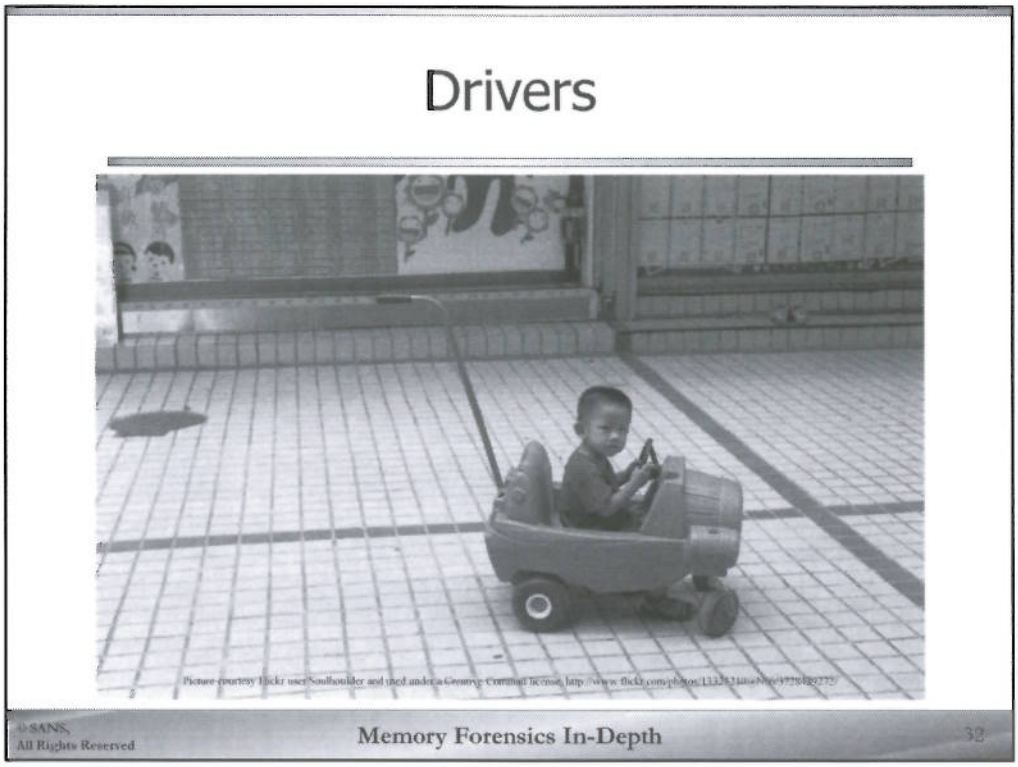


Making sense of apihooks plugin output.

Hook mode: Usermode
Hook type: Import Address Table (IAT)
Process: 1892 (iexplore.exe)
Victim module: VERSION.dll (0x74460000 - 0x74469000)
Function: kernel32.dll!LoadLibraryExW
Hook address: 0x72351ed3
Hooking module: IEShims.dll



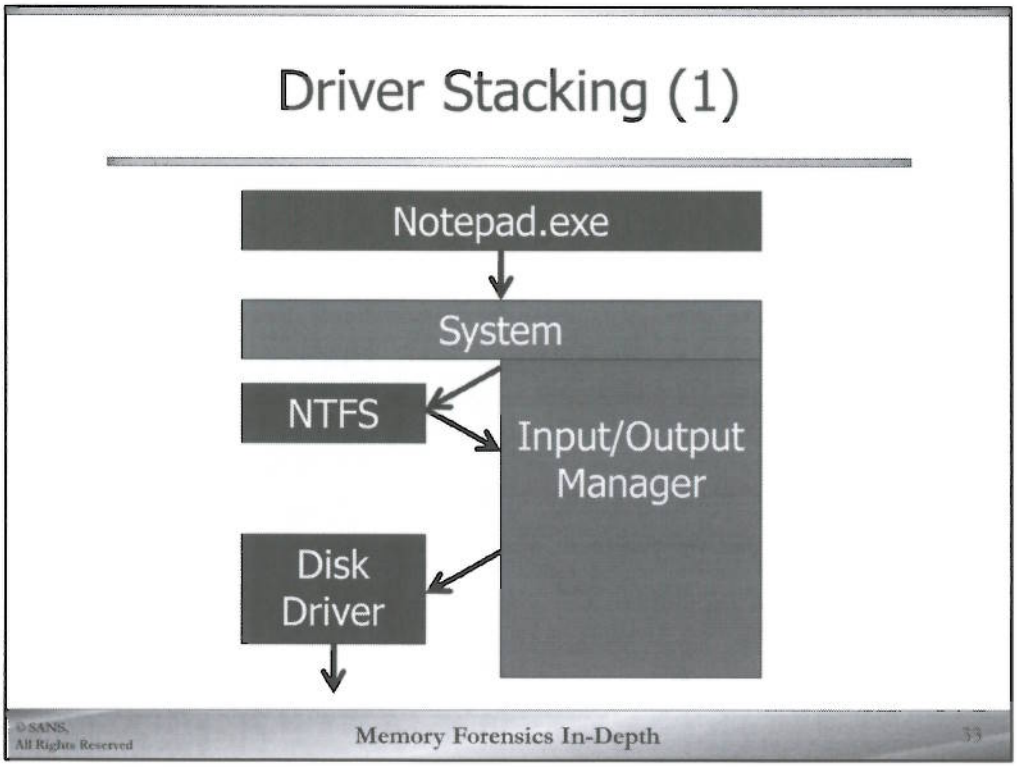
This page intentionally left blank.



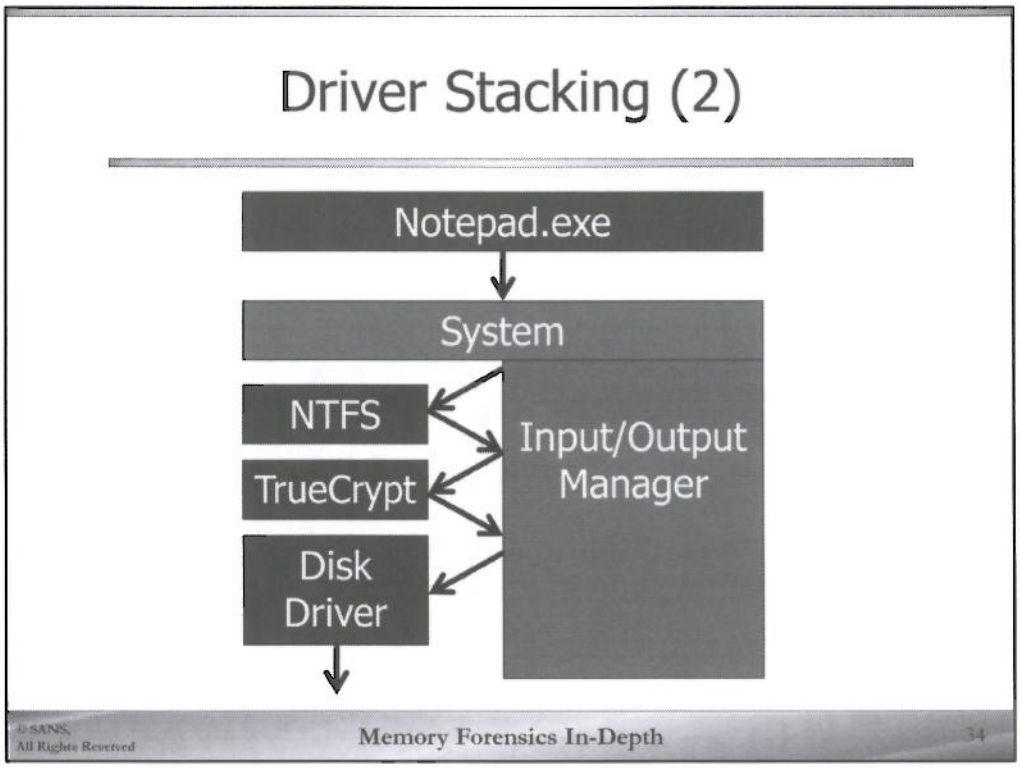
Drivers are modules, like DLLs or executables, but which run in kernel space. Their code executes as part of the system process and in the system's memory space. Some of them are used to communicate with hardware on the system, such as keyboards, mice, USB devices, or network cards. Others deal with parsing the information passed in from hardware. For example, the code to parse NTFS filesystems is in a driver. Full disk encryption tools use drivers to do the extra work of encrypting and decrypting data. These drivers can work in layers, each one passing data from one layer to the next. For example, the filesystems driver passes data to the NTFS driver, which passes it to the FDE driver, which passes it to the ATAPI driver for talking to the disk, and so on.

Along with these legitimate functions, however, malicious programs use drivers too. They are used to intercept communication with hardware devices (e.g. keyloggers or network monitors), create hidden storage areas on a disk, and hide processes, for example. Some progress has been made against malicious drivers through enhancements to Windows. Sixty-four bit versions of Windows since Vista, for example, require that all drivers be signed in order to run. Admittedly, advanced malware such as the Stuxnet worm *was* signed, but it's a step in the right direction. On 32-bit platforms, however, malicious drivers definitely remain a concern.

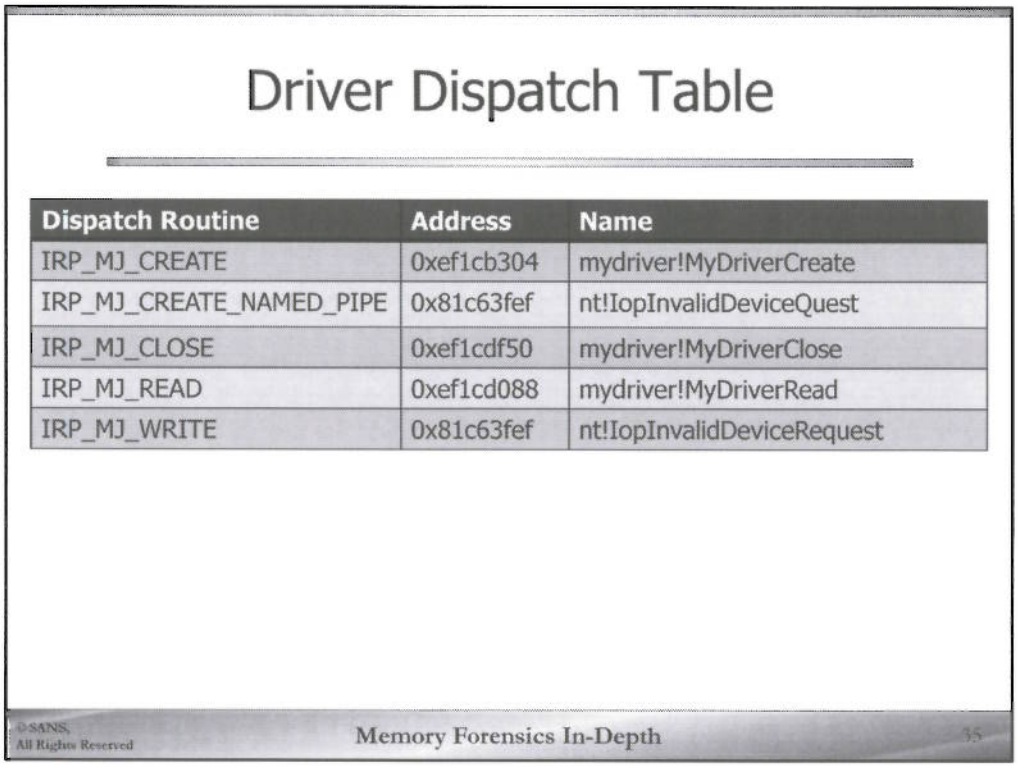
Windows maintains a list of loaded drivers. This doubly-linked list contains structures of the type LDR_DATA_TABLE_ENTRY. The kernel variable PsLoadedModuleList points to the head of this list. That variable, in turn, is found in the KDBG structure. The Volatility framework can parse the list of loaded modules using the plugin 'modules'. There are also two plugins for doing brute force searches for drivers and their associated device objects. As with processes, walking the list of drivers will only show us the drivers the operating system knew about. Any drivers which were unloaded or hidden will not be displayed. We'll have to use the brute force searching plugins for that.



Windows drivers have their operations stacked. While some drivers do talk to hardware devices, many do not. Instead these middleware drivers serve to interpret instruction from the layers above and below. When a user process, like notepad.exe, wants to read data from a file, it makes a request to the operating system. That request is handed over to the kernel, which passes it to the input/output manager, which in turn talks to the drivers. The request to read a particular file is handled by the driver which supports the filesystem in question. Let's say, for this system, that's the NTFS driver. That driver translates the read request from a filename into the logical block numbers on the disk. This request is then passed down the chain, going through the I/O manager to the disk driver, which can translate the request for logical block numbers into the hardware specific calls necessary to load data via the ATA or lower level interface.



If a new driver is loaded, it can be inserted into this stack. Continuing our example, if the user then loaded the TrueCrypt driver for doing full-disk encryption, that driver would be stacked between the NTFS driver and disk driver. From the perspective of the NTFS driver, nothing changes. It receives request from the operating system to read and write data. The requests for files are translated into read and write requests for logical blocks on the disk. Those requests are then passed down to the next driver in the stack. The TrueCrypt driver ensures that the data being passed up is decrypted before being presented to the NTFS driver. Similarly, on the way down, the TrueCrypt driver ensures that data being written are encrypted before being sent to the disk. The encryption and decryption process is transparent to the file system driver and to the user application.



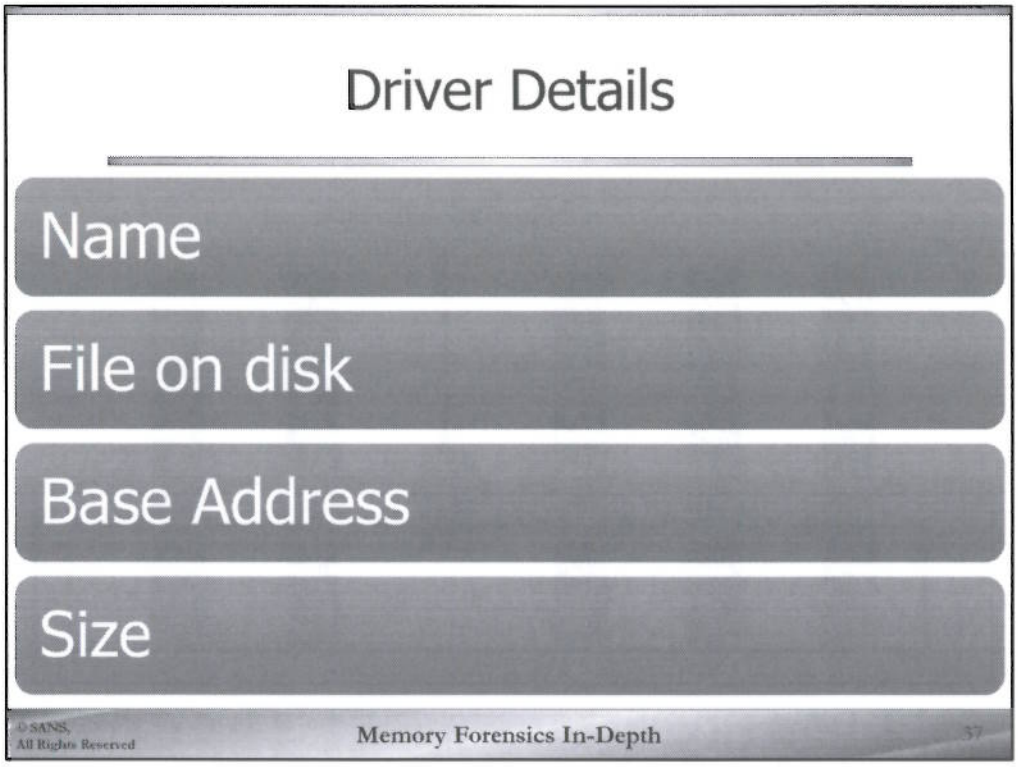
Every driver has to follow a set format in order to fit into the layered system. They all must have a function for being initialized and a function for adding a device. For example, the driver which handles USB devices will need to be called each time a USB device is connected to the system. This function is used to create the device objects, one for each device connected to the system, which are also maintained by the operating system.

Each driver must then have a table of the 28 possible dispatch routines. These routines cover the operation of what user programs and other drivers could ask this driver to do. For example, there are routines to create or open, read, write, control the power, and close. The driver must specify what function in the driver to execute when such a request is presented to it. The driver doesn't have to implement every function, however. The driver can refer requests it does not support to a special function in the kernel, IopInvalidDeviceQuest, which just returns an error message.

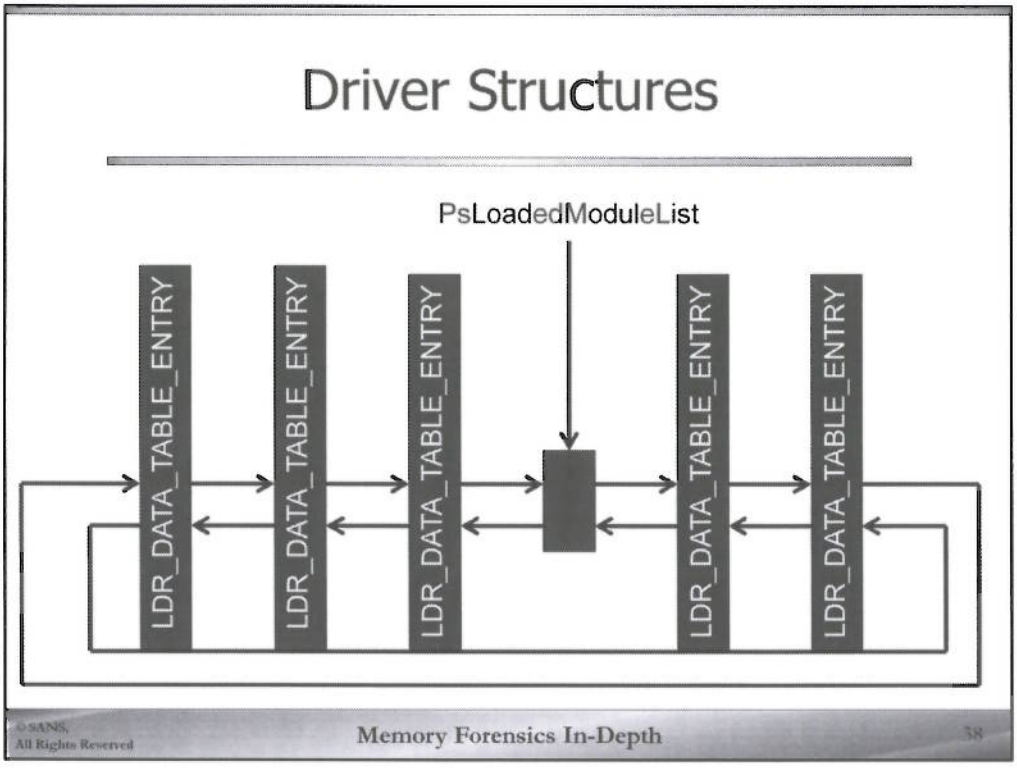
The example above shows a hypothetical driver, MyDriver.sys, which has been loaded into the system. The leftmost column contains five of the 28 dispatch requests. The middle column shows the address of the function which handles that request. Using the symbols provided by Microsoft and our driver's author, we can then compute the name of the function at that address, which is shown in the right-hand column. For example, our driver can handle a request to create or open a resource, using the function in the mydriver.sys code called MyDriverCreate. But a request to create a named pipe gets directed back to the kernel and the function IopInvalidDeviceRequest.

These 28 functions all require an Input/Output Request Packet (IRP) to be passed along with the command. The IRP is a standard form for handling such requests. The 28 dispatch routines are also called the IRP handlers. When a thread makes an I/O request, it generates an IRP, passes it up to the kernel, and adds the IRP to a list it maintains of IRPs it is waiting for. Since the thread can't do anything until the IRP is resolved, it goes into a wait state until it gets a response. We'll talk more about threads and their states in section 12.

There is other functionality a driver can implement, such as a system for "fast" dispatches which don't use IRPs. They instead rely on the caching system of Windows. For example, when Notepad needs to read part of a file from the disk, and that data has already been read by the operating system, a fast I/O request will grab the data from the cache rather than going out to the disk again.

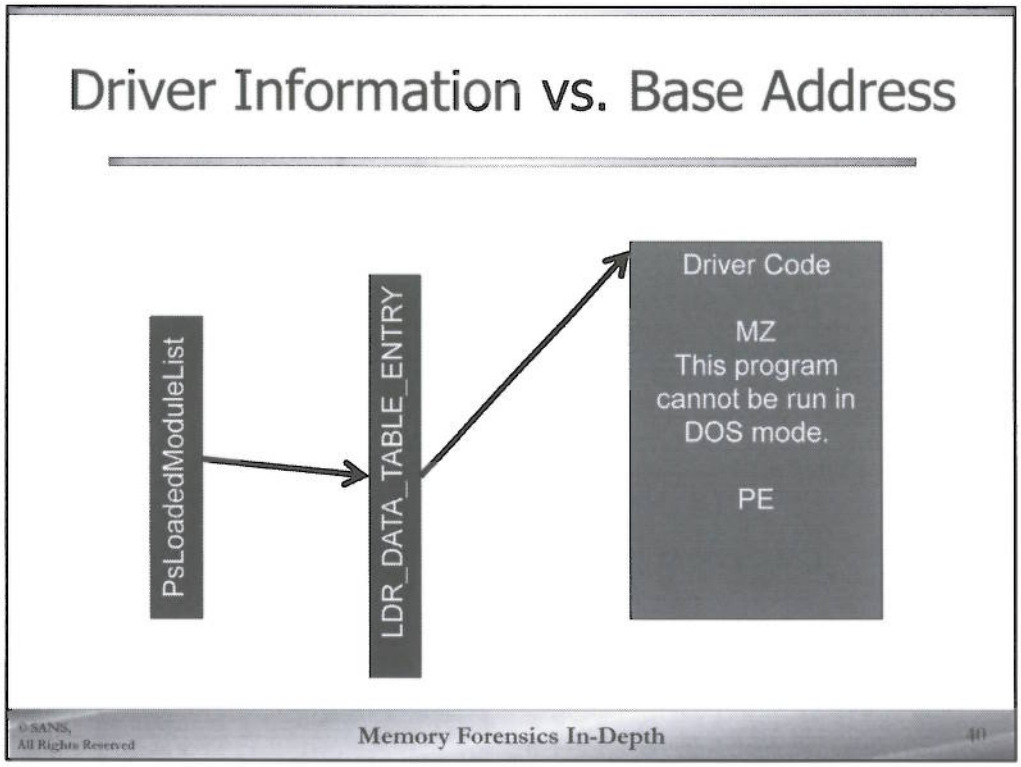


We can get several details about each driver loaded into the operating system. We get the name of the driver, which is sometimes different the filename on the disk. (We also get the full path to the driver, too). We also get the base virtual address where this driver would prefer to be loaded into memory. Like DLLs, drivers can be relocated in virtual memory. Finally, we get the size, in bytes, of the driver in memory.



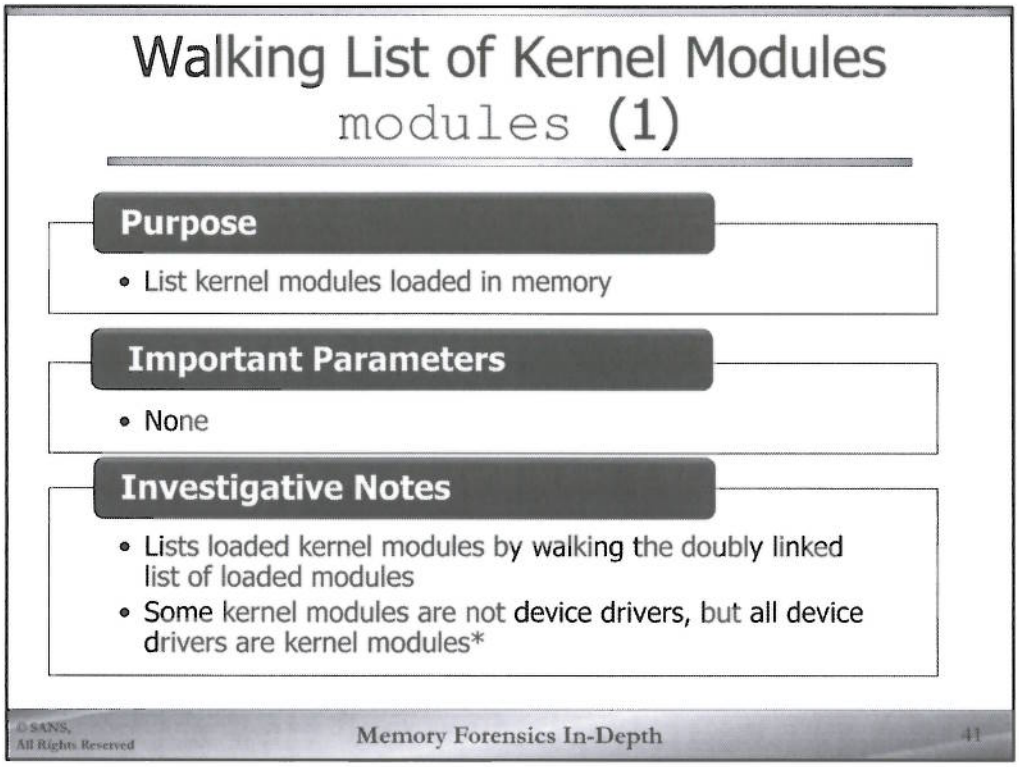
When we look at the list of loaded drivers, we starting with the PsLoadedModuleList kernel variable from the Kernel Data Debugging Block. This variable in turn points to a list of structures called LDR_DATA_TABLE_ENTRYs. (The list is actually doubly-circularly linked, like the list of processes. It's shown simplified above for clarity.) Each of these structures contains the base virtual address for the driver, where the code lives in memory.

PsLoadedModuleList
LDR_DATA_TABLE_TABLE
LDR_DATA_TABLE_TABLE
LDR_DATA_TABLE_TABLE
LDR_DATA_TABLE_TABLE
LDR_DATA_TABLE_TABLE



When we talked about processes, we were generally only concerned with the virtual and physical addresses of the EPROCESS blocks. When it comes to drivers, however, we will be discussing the location of the driver information in memory and the base address of the driver. That is, the driver information is the LDR_DATA_TABLE_ENTRY structures. But the base address of the driver is where the actual code lives.

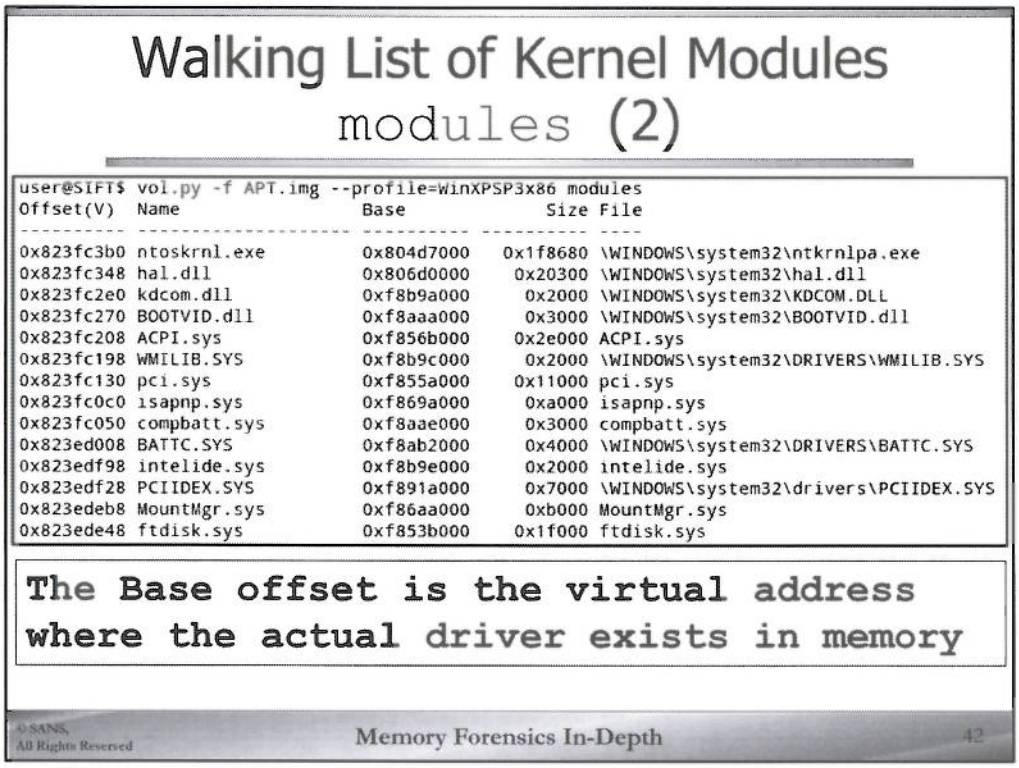
In the picture above, the driver information is in the center of the picture. The driver base address would be on the right-hand side--where the code lives. When we go to recover samples of drivers from memory, we will need the latter.



The modules plugin works by locating the head of the doubly linked list of LDR_DATA_TABLE_ENTRY structures maintained by the kernel. Malware may hide from this plugin by unlinking a module from this list. Note that all device drivers are loaded as kernel modules*. However, not all kernel modules have associated driver or device objects. Therefore, to get the most complete list of kernel code loaded, you should use the modules plugin (as opposed to driverscan or devicetree).

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

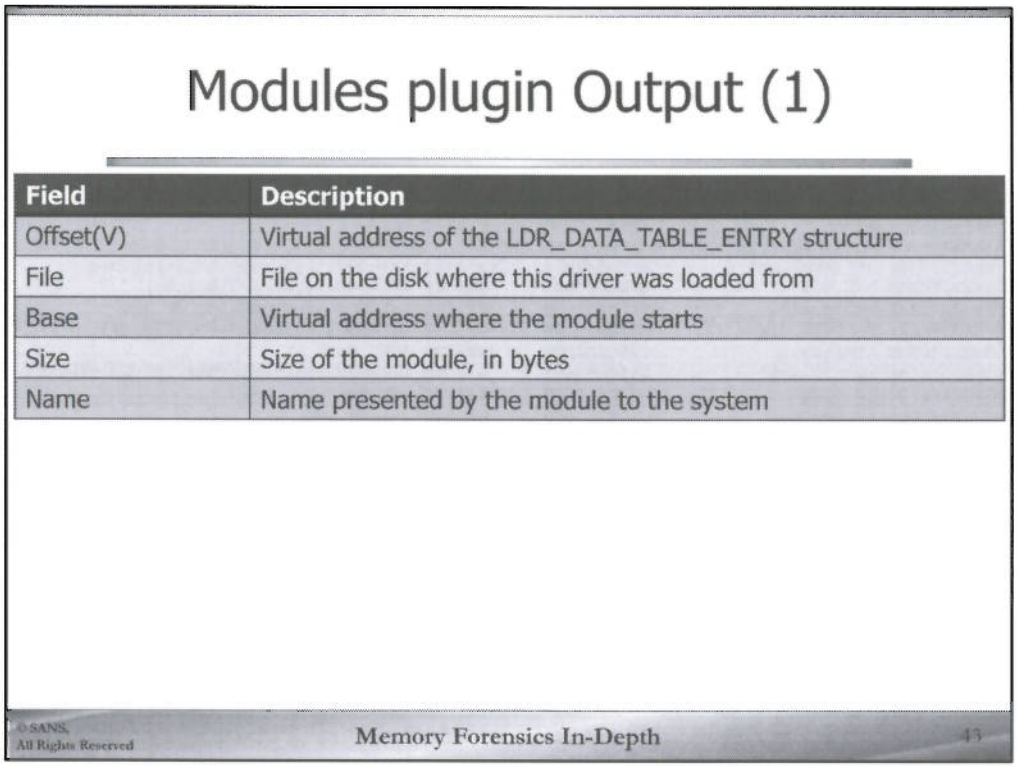
* Okay, well there are "user mode device drivers" written with UMDF. These are not typically seen in malware investigations. They also use a kernel module on the back end. If you are interested in these, read the MSDN documentation on UMDF.



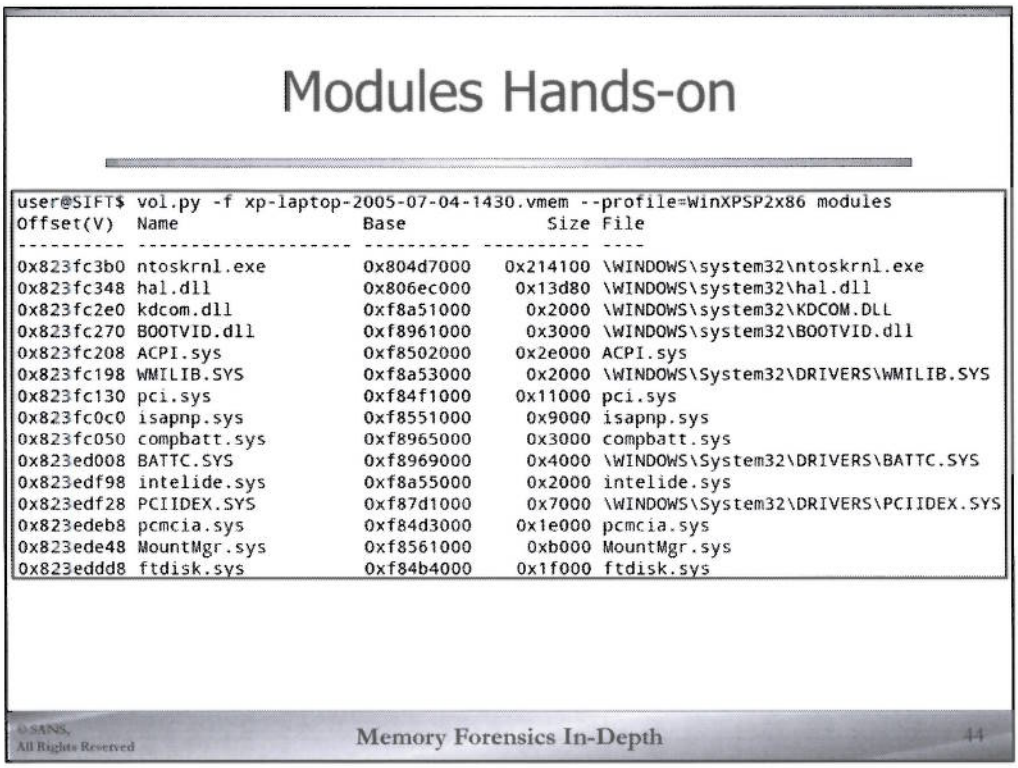
This shows the partial output of the modules plugin run against the APT.img memory capture. Note that when the full path to the file is not displayed (as is the case with ACPI.sys shown below), the path is %systemroot%\system32\drivers.

 # vol.py -f APT.img --profile=WinXPSP3x86 modules

```
Offset(V)  Name                 Base       Size File
---------- -------------------- ---------- ---------- ----
0x823fc3b0 ntoskrnl.exe         0x804d7000 0x1f8680 \WINDOWS\system32\ntkrnlpa.exe
0x823fc348 hal.dll              0x806d0000 0x20300 \WINDOWS\system32\hal.dll
0x823fc2e0 kdcom.dll            0xf8b9a000 0x2000 \WINDOWS\system32\KDCOM.DLL
0x823fc270 BOOTVID.dll          0xf8aaa000 0x3000 \WINDOWS\system32\BOOTVID.dll
0x823fc208 ACPI.sys             0xf856b000 0x2e000 ACPI.sys
0x823fc198 WMILIB.SYS           0xf8b9c000 0x2000 \WINDOWS\system32\DRIVERS\WMILIB.SYS
0x823fc130 pci.sys              0xf855a000 0x11000 pci.sys
0x823fc0c0 isapnp.sys           0xf869a000 0xa000 isapnp.sys
0x823fc050 compbatt.sys         0xf8aae000 0x3000 compbatt.sys
0x823ed008 BATTC.SYS            0xf8ab2000 0x4000 \WINDOWS\system32\DRIVERS\BATTC.SYS
... output truncated ...
```



Now let's talk about how our tools display that information. The fields returned by the modules plugin are shown above. By default the plugin provides the virtual address of each LDR_DATA_TABLE_ENTRY, but you can have it provide the physical offset in the memory image by adding the -P flag. The LDR_DATA_TABLE_ENTRY structure contains information on each driver. The filename on the disk may not include a full path. Other paths may refer to the SystemRoot environment variable. But the vast majority of these entries will refer to C:\Windows\System32, where drivers are supposed to be stored.



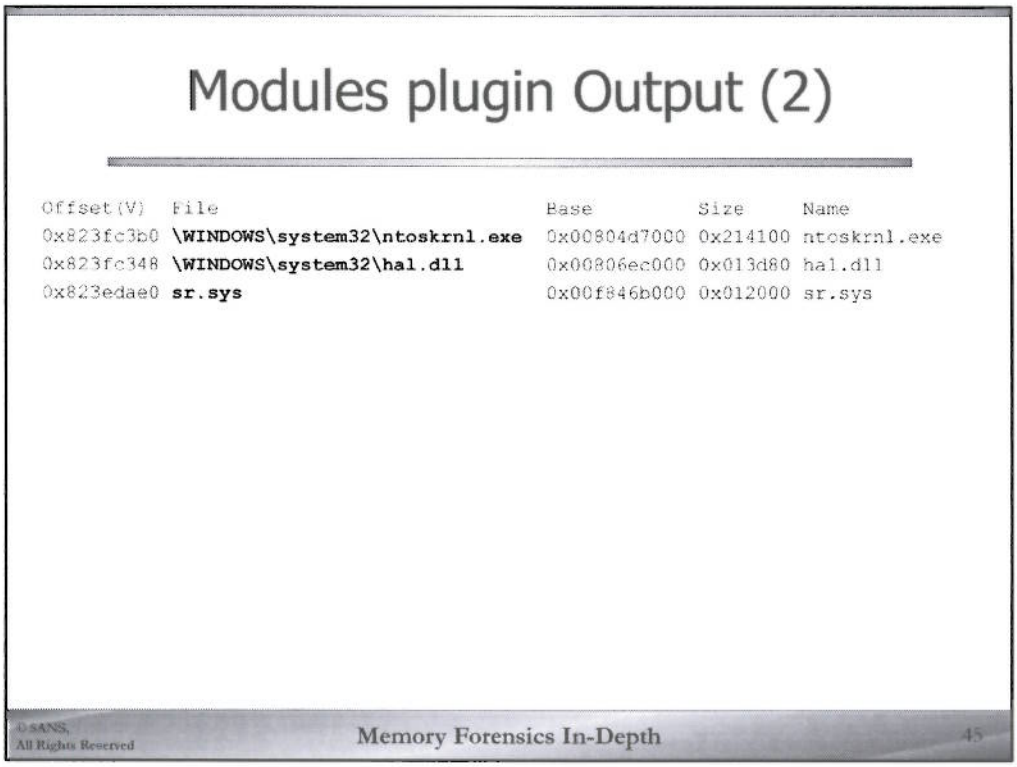
Let's take a look at some legitimate drivers. We are now going to use Volatility to walk the list of loaded drivers indicated by the PsLoadedModuleList variable using the 'modules' plugin. Along with the memory image filename, we need to specify the profile for the input memory image. Remember, if you don't know the profile, use the 'imageinfo' plugin to help you select the correct one.

Here's the command line for the modules plugin:

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

You should see output which starts with:

```
Offset(V)  File                                  Base          Size      Name
0x823fc3b0 \WINDOWS\system32\ntoskrnl.exe  0x00804d7000 0x214100 ntoskrnl.exe
0x823fc348 \WINDOWS\system32\hal.dll       0x00806ec000 0x013d80 hal.dll
0x823fc2e0 \WINDOWS\system32\KDCOM.DLL     0x00f8a51000 0x002000 kdcom.dll
0x823fc270 \WINDOWS\system32\BOOTVID.dll   0x00f8961000 0x003000 BOOTVID.dll
```

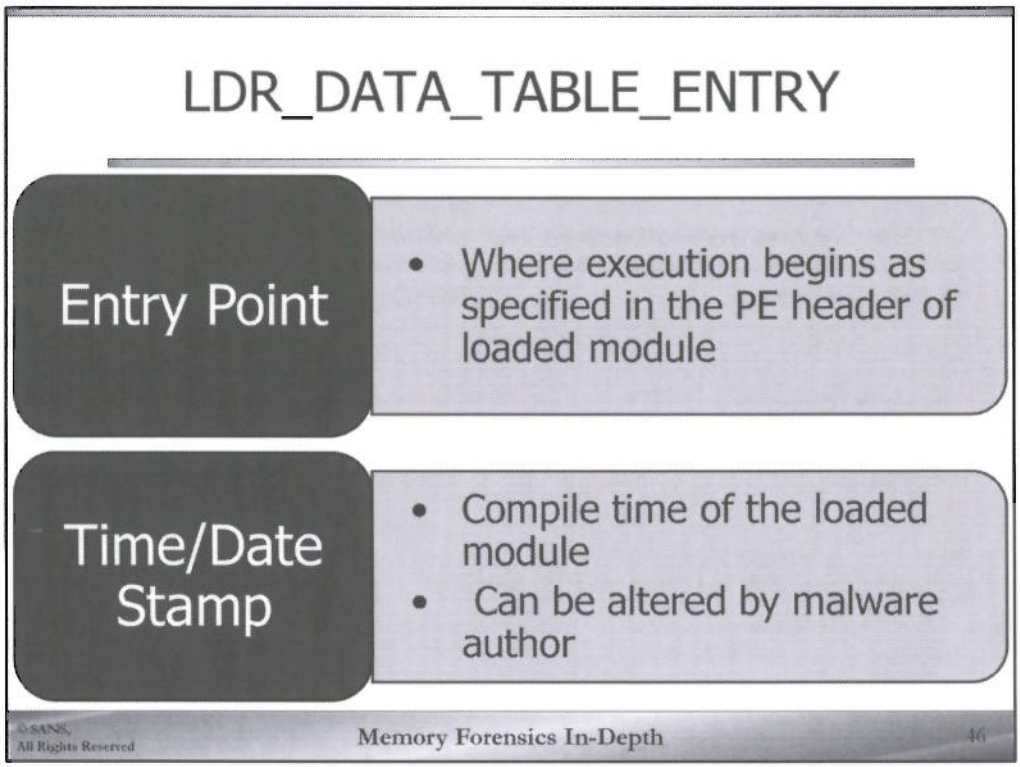



```
Offset(V)  File                                  Base          Size      Name
0x823fc3b0 \WINDOWS\system32\ntoskrnl.exe    0x00804d7000 0x214100 ntoskrnl.exe
0x823fc348 \WINDOWS\system32\hal.dll         0x00806ec000 0x013d80 hal.dll
0x823edae0 sr.sys                            0x00f846b000 0x012000 sr.sys
```

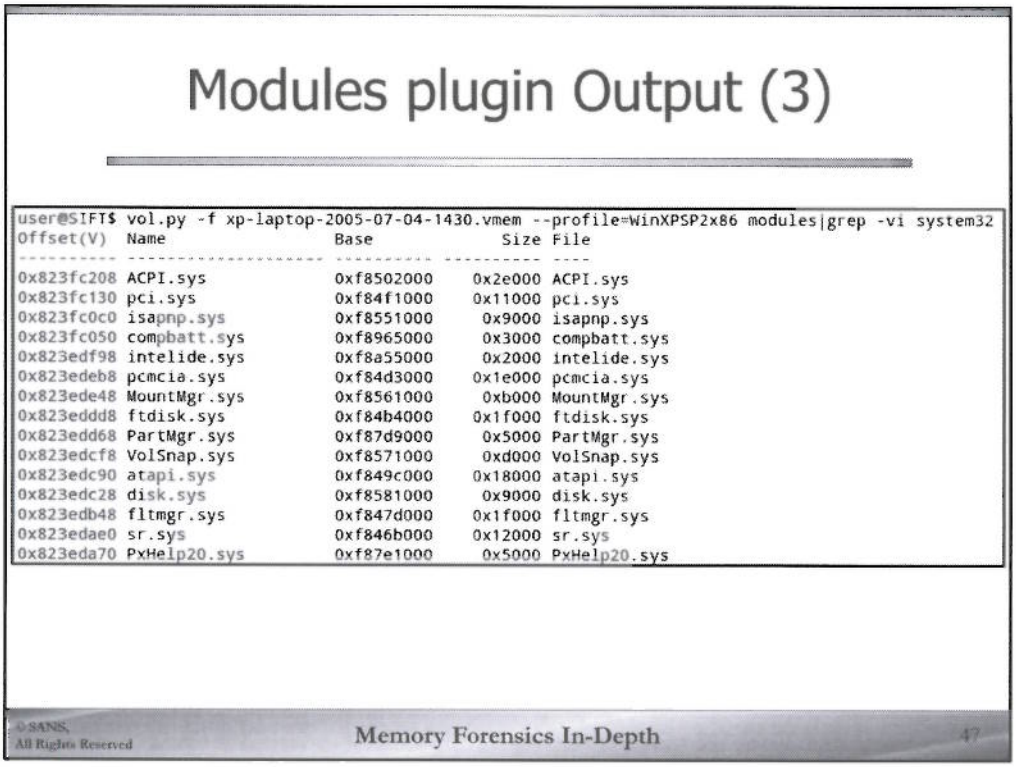
Looking at the output of 'modules' on the XP laptop image, you should see that the kernel itself is the first module in the list. The kernel is considered a module, just like all of the other drivers. You can even see which file on the disk contains the kernel. The next entry is the driver for the Hardware Abstraction Layer, or HAL, in hal.dll. This driver does the work of dealing with the processor and low-level system bus so that the operating system doesn't have to. You can read all about the other drivers in the Windows Internals book, or by doing searches on the Internet. You may find it helpful to restrict your searches to pages on microsoft.com.

For example, what is sr.sys?

Hopefully, after searching, you can find that it's part of the Windows System Restore service, which allows users to return to an earlier state of the machine (e.g. http://msdn.microsoft.com/en-us/library/bb521613(v=winembedded.51).aspx)



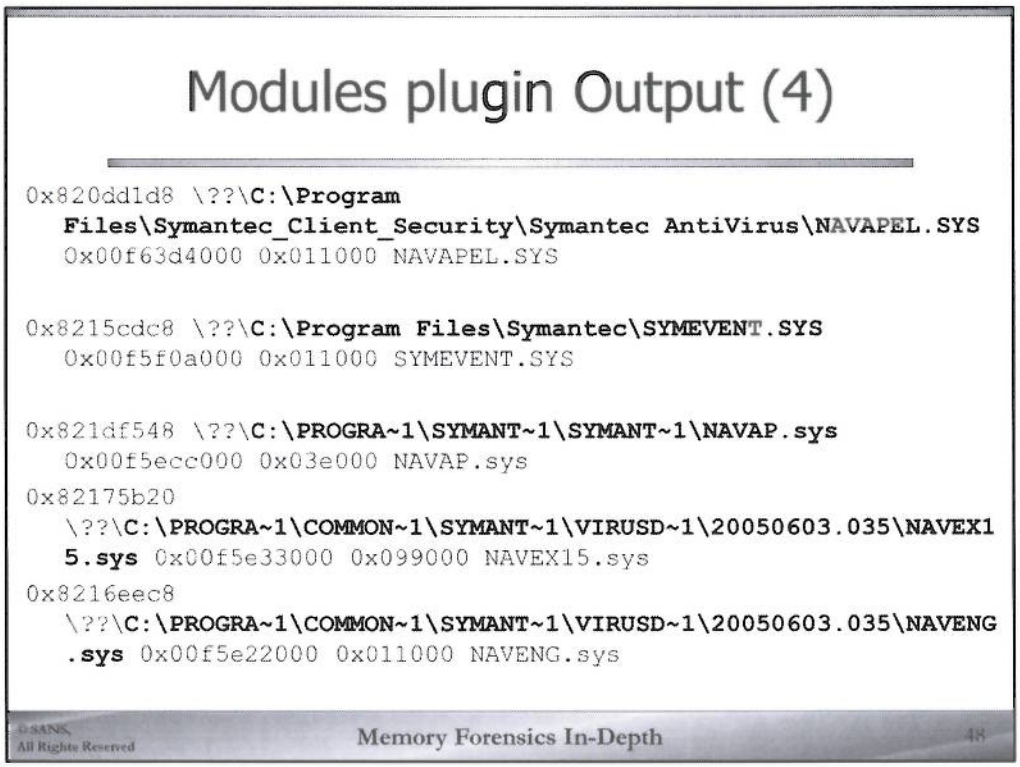
Along with the information which is displayed by the 'modules' plugin, the LDR_DATA_TABLE_ENTRY structure contains a few other pieces of data. The structure includes the entry point of the code, or the first instruction to be executed when the driver is run. This address is crucial for reverse engineering the driver. It tells us where to start disassembling code. The structure also contains the compile time/date stamp of the loaded module, but this field can be altered by the compiler. A malware author, especially, could alter this timestamp to throw you off the case. But maybe they *didn't* alter the timestamp. Maybe the bad guy always changes the timestamp to the same wacky value. It could be an indicator of a particular person or actor. But don't bet the farm on it.



Looking through the output some more, we can see there were a few drivers loaded from outside the System32 directory. You can get an easier look at these entries by running the plugin again, but this time using grep to remove any line which doesn't contain the string system32. Repeat the same command line, but add a pipe and 'grep -vi system32', like this:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
modules | grep -vi system32
```

The -v flag to grep causes it to display any line which does not match the specified pattern.



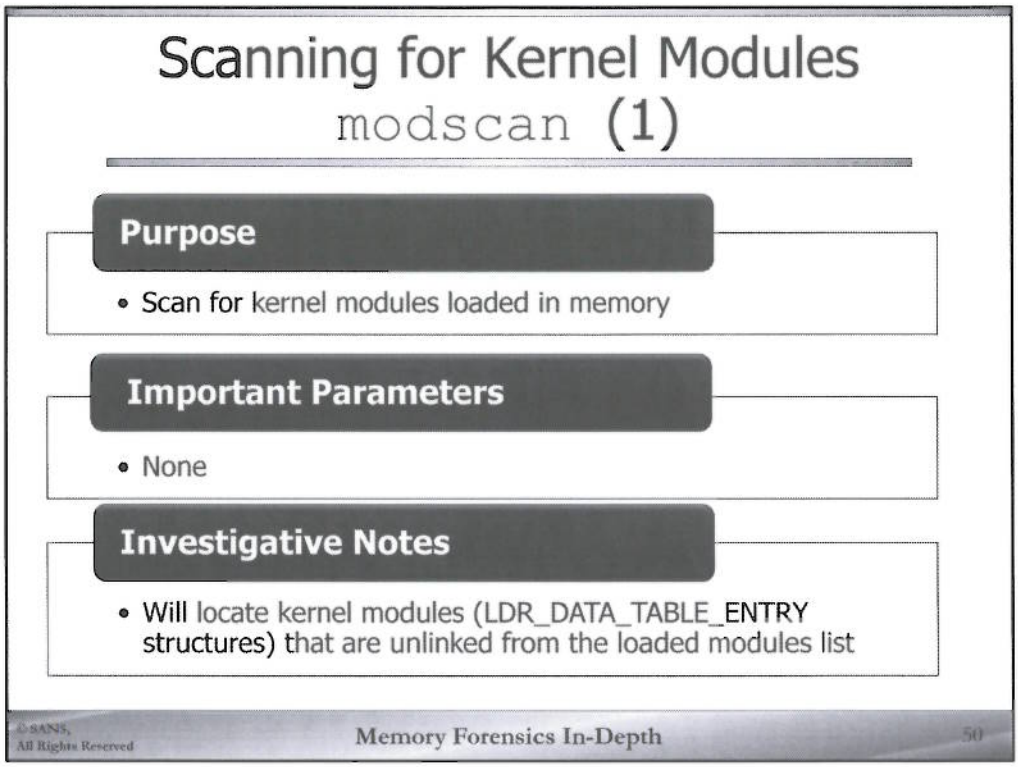
You should get the following:

```
Volatile Systems Volatility Framework 2.3
Offset(V)  Name                 Base             Size File
---------- -------------------- ---------- ---------- ----
0x823fc208 ACPI.sys             0xf8502000    0x2e000 ACPI.sys
0x823fc130 pci.sys              0xf84f1000    0x11000 pci.sys
0x823fc0c0 isapnp.sys           0xf8551000     0x9000 isapnp.sys
0x823fc050 compbatt.sys         0xf8965000     0x3000 compbatt.sys
0x823edf98 intelide.sys         0xf8a55000     0x2000 intelide.sys
0x823edeb8 pcmcia.sys           0xf84d3000    0x1e000 pcmcia.sys
0x823ede48 MountMgr.sys         0xf8561000     0xb000 MountMgr.sys
0x823eddd8 ftdisk.sys           0xf84b4000    0x1f000 ftdisk.sys
0x823edd68 PartMgr.sys          0xf87d9000     0x5000 PartMgr.sys
0x823edcf8 VolSnap.sys          0xf8571000     0xd000 VolSnap.sys
0x823edc90 atapi.sys            0xf849c000    0x18000 atapi.sys
0x823edc28 disk.sys             0xf8581000     0x9000 disk.sys
0x823edb48 fltmgr.sys           0xf847d000    0x1f000 fltmgr.sys
0x823edae0 sr.sys               0xf846b000    0x12000 sr.sys
0x823eda70 PxHelp20.sys         0xf87e1000     0x5000 PxHelp20.sys
0x823eda00 KSecDD.sys           0xf8454000    0x17000 KSecDD.sys
0x823ed998 Ntfs.sys             0xf83c7000    0x8d000 Ntfs.sys
0x823ed930 NDIS.sys             0xf839a000    0x2d000 NDIS.sys
0x823ed8c0 Teefer.sys           0xf837d000    0x1d000 Teefer.sys
0x823ed858 Mup.sys              0xf8362000    0x1b000 Mup.sys
```

```
0x823ed7e8 agp440.sys           0xf85a1000     0xb000 agp440.sys
0x820dd1d8 NAVAPEL.SYS          0xf63d4000    0x11000 \??\C:\Program
Files\Symantec_Client_Security\Symantec AntiVirus\NAVAPEL.SYS
0x8215cdc8 SYMEVENT.SYS         0xf5f0a000    0x11000 \??\C:\Program
Files\Symantec\SYMEVENT.SYS
0x821df548 NAVAP.sys            0xf5ecc000    0x3e000
\??\C:\PROGRA~1\SYMANT~1\SYMANT~1\NAVAP.sys
0x82175b20 NAVEX15.sys          0xf5e33000    0x99000
\??\C:\PROGRA~1\COMMON~1\SYMANT~1\VIRUSD~1\20050603.035\NAVEX15.sys
0x8216eec8 NAVENG.sys           0xf5e22000    0x11000
\??\C:\PROGRA~1\COMMON~1\SYMANT~1\VIRUSD~1\20050603.035\NAVENG.sys
```

Along with some basic system drivers for things like NTFS and ATAPI disks, we can also see five entries related to Symantec anti-virus.

We can also see that these drivers were loaded out of the Symantec directory in the Program Files directory. These drivers are intercepting calls to open and run executables, for example, searching for viruses. These are legitimate entries. But don't think some piece of malware might not try to camouflage itself by looking like one of these!



When a driver is loaded, it creates a number of device objects which correspond to resources it controls. These could be hardware devices, filesystems objects, and so on. If all of these objects go away, however, Windows will automatically unload the driver. This is done to reclaim the space in both virtual and physical memory. There is only so much room in the kernel's virtual address space, regardless of how much physical memory is present on the system!

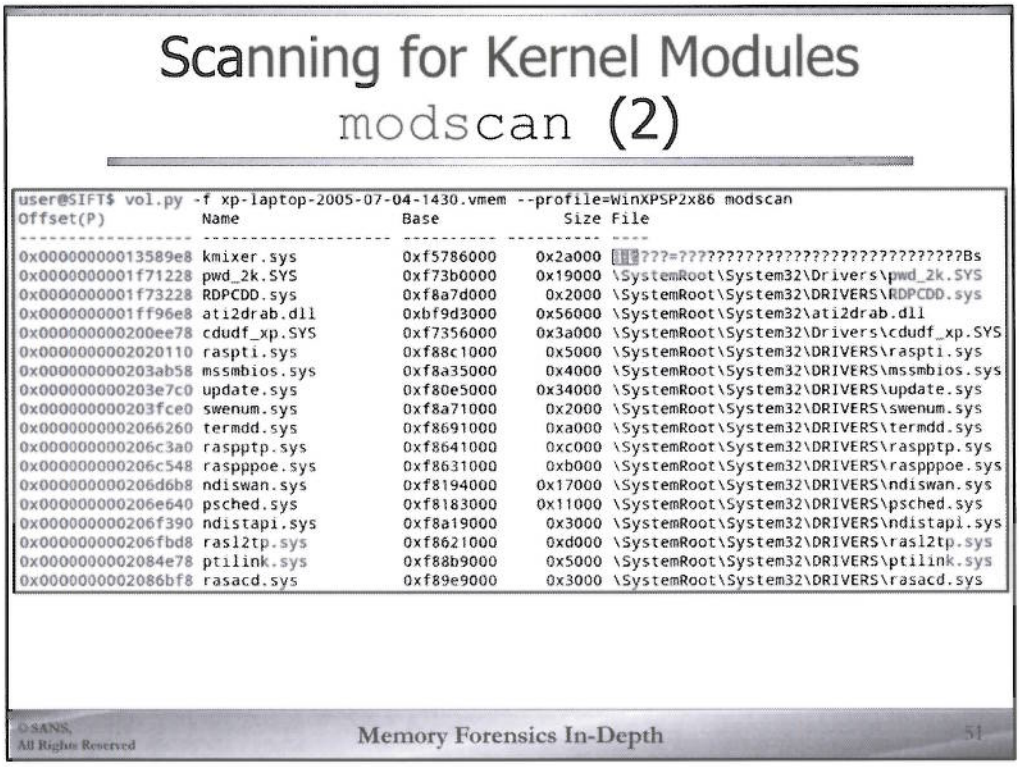
If a driver is unloaded, it is removed from the list of loaded kernel drivers on the system, and thus will no longer be visible in the output of the modules plugin. Just because it's been unloaded, however, doesn't mean that the LDR_DATA_TABLE_ENTRY structure will be destroyed. Such structures, like process information, will stay in memory until overwritten.

Because they're still in memory, we can scan for them and find them!

The modscan plugin searches kernel memory for pools with the tag 'MmLd'. Because is scans memory for these pool tags (rather than walking the LDR_DATA_TABLE_ENTRY list). Because some malware may unlink itself from this list, the loaded kernel module would not be present in the output of the modules plugin.

Because this module scans memory, duplicate entries are possible. This happens when a LDR_DATA_TABLE_ENTRY object is assigned to memory that is paged out and then paged back in at a different physical address and the old physical memory is not overwritten. Recently unloaded modules may also be located using modscan.

It is implemented in volatility/plugins/modscan.py



Below is example output from the **modscan** plugin. Unlike the pslist and psscan plugins for processes, there is no psxview like plugin for modules. You're left to find differences between the output lists yourself.

user@SIFT$ vol.py -f xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86 modscan

Offset(P) Name Base Size File

0x00000000013589e8 kmixer.sys 0xf5786000 0x2a000 ????=????????????????????????????????Bs

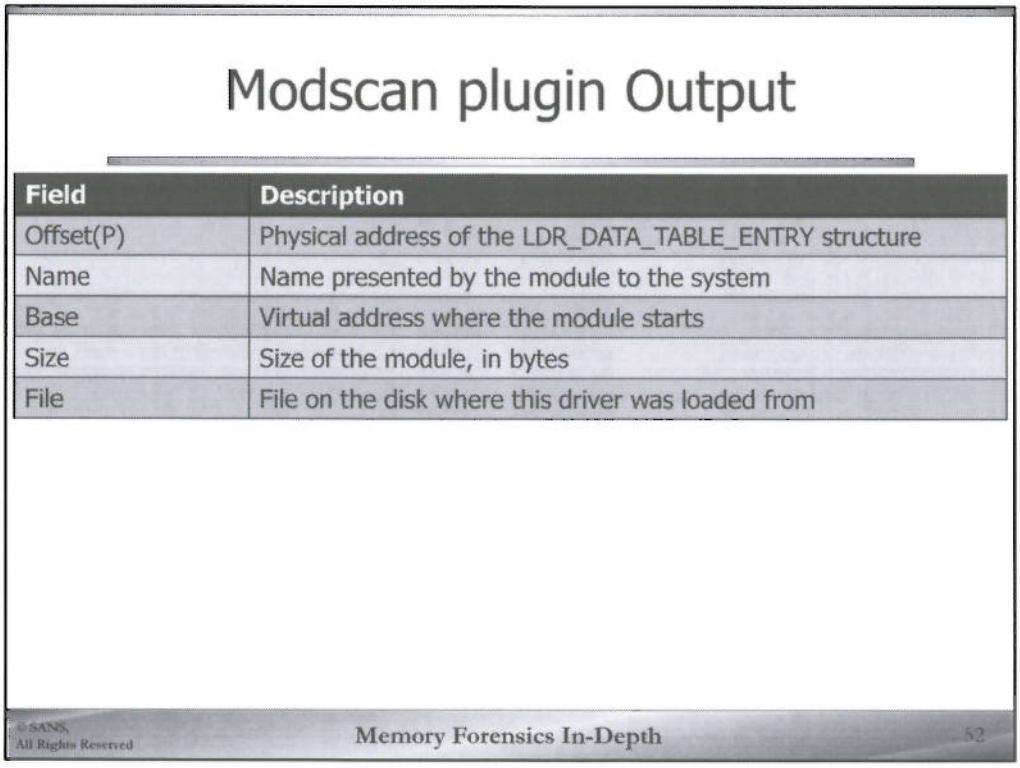
0x0000000001f71228 pwd_2k.SYS 0xf73b0000 0x19000 \SystemRoot\System32\Drivers\pwd_2k.SYS

0x0000000001f73228 RDPCDD.sys 0xf8a7d000 0x2000 \SystemRoot\System32\DRIVERS\RDPCDD.sys

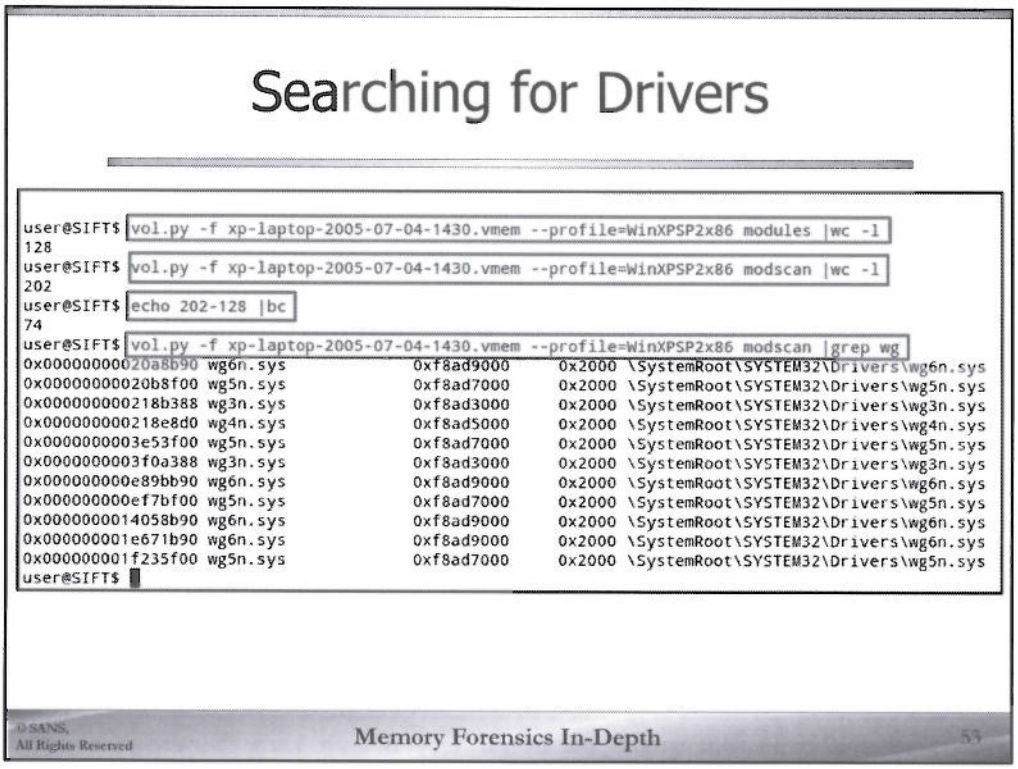
0x0000000001ff96e8 ati2drab.dll 0xbf9d3000 0x56000 \SystemRoot\System32\ati2drab.dll

0x000000000200ee78 cdudf_xp.SYS 0xf7356000 0x3a000 \SystemRoot\System32\Drivers\cdudf_xp.SYS

… output truncated …



The output of modscan is almost identical to the output of modules. The only difference is that the offset field is always the physical offset in the memory image. Because we're doing a brute force scan, the plugin does not see the virtual addresses of these structures.



Along with unloaded and hidden drivers, we're also going to find some duplicate entries in the module scan. Often times, a lot of them. We're going to do a hands on exercise to demonstrate just how many duplicates there can be. First we're going to run the modules plugin again, but this time piping it through wc, a program to count the number of lines in the output. (The -l flag is what makes this program count lines). It's the same command line we used before, but with an extra pipe at the end:

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

127

We can see there are 127 lines of output in the list of loaded modules.

We then repeat the process with modscan, like this:

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

202

And this time we see there 201 lines in the output. Doing a scan instead of walking the list yielded an additional 74 possible drivers!

(The bc program is a calculator program for *nix systems.)

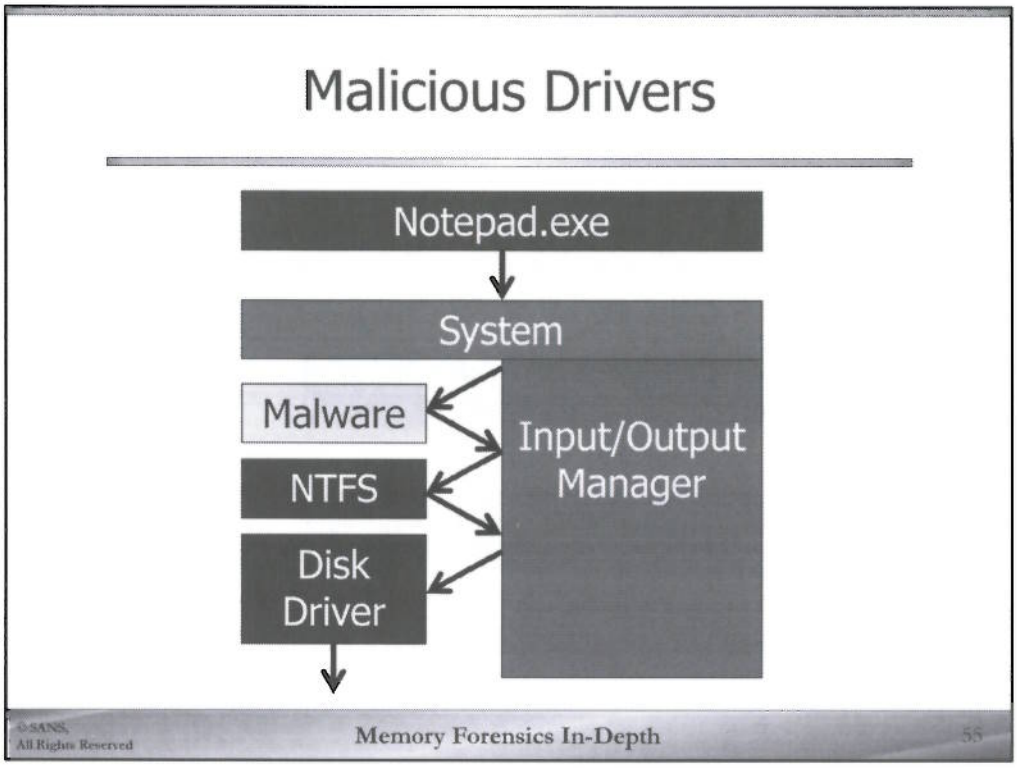
Before we get too excited about all of those extra results, we have to look for and weed out the duplicate values. And there are lots of them. For example, let's look for all of the drivers which have wg in the name:

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

You should get the following output:

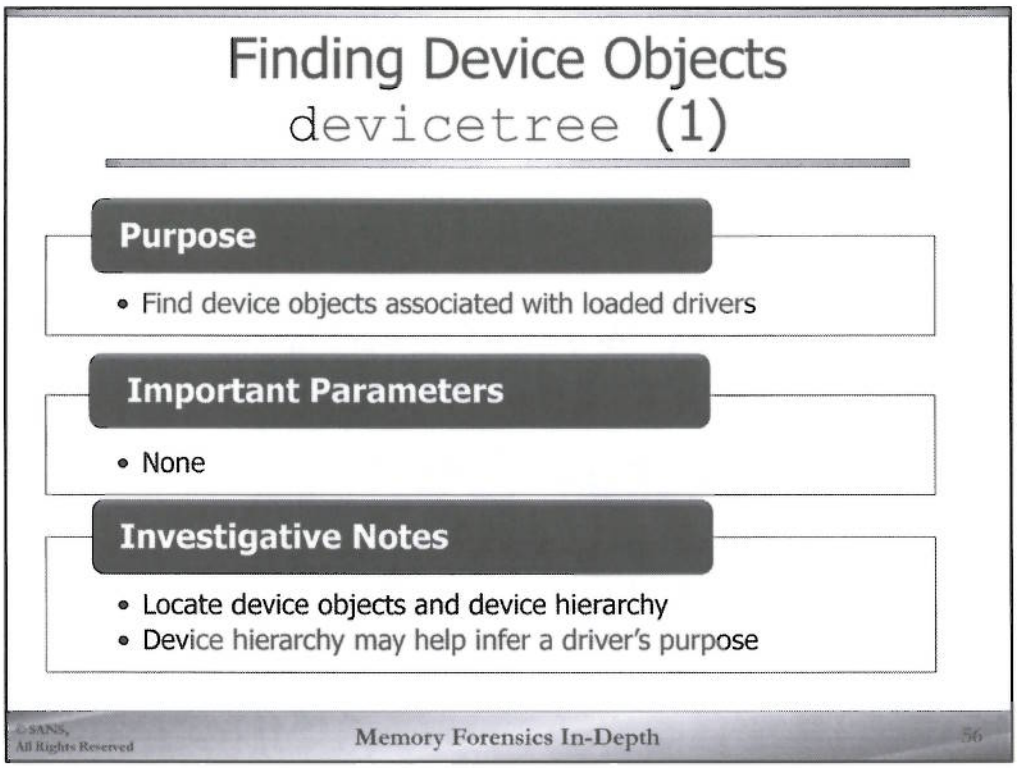
```
0x020a8b90 '\\SystemRoot\\SYSTEM32\\Drivers\\wg6n.sys'      0x00f8ad9000 0x002000 'wg6n.sys'
0x020b8f00 '\\SystemRoot\\SYSTEM32\\Drivers\\wg5n.sys'      0x00f8ad7000 0x002000 'wg5n.sys'
0x0218b388 '\\SystemRoot\\SYSTEM32\\Drivers\\wg3n.sys'      0x00f8ad3000 0x002000 'wg3n.sys'
0x0218e8d0 '\\SystemRoot\\SYSTEM32\\Drivers\\wg4n.sys'      0x00f8ad5000 0x002000 'wg4n.sys'
0x03e53f00 '\\SystemRoot\\SYSTEM32\\Drivers\\wg5n.sys'      0x00f8ad7000 0x002000 'wg5n.sys'
0x03f0a388 '\\SystemRoot\\SYSTEM32\\Drivers\\wg3n.sys'      0x00f8ad3000 0x002000 'wg3n.sys'
0x0e89bb90 '\\SystemRoot\\SYSTEM32\\Drivers\\wg6n.sys'      0x00f8ad9000 0x002000 'wg6n.sys'
0x0ef7bf00 '\\SystemRoot\\SYSTEM32\\Drivers\\wg5n.sys'      0x00f8ad7000 0x002000 'wg5n.sys'
0x14058b90 '\\SystemRoot\\SYSTEM32\\Drivers\\wg6n.sys'      0x00f8ad9000 0x002000 'wg6n.sys'
0x1e671b90 '\\SystemRoot\\SYSTEM32\\Drivers\\wg6n.sys'      0x00f8ad9000 0x002000 'wg6n.sys'
0x1f235f00 '\\SystemRoot\\SYSTEM32\\Drivers\\wg5n.sys'      0x00f8ad7000 0x002000 'wg5n.sys'
```

There are really only four distinct entries here, but several copies of each. These four drivers account for eleven entries in the modscan output. For example, wg3n.sys, which has identical virtual offsets 0x00f8ad3000 and size of 0x2000, can be found at offsets 0x21b8388 and 0x3f0a388. In memory images, duplicates happen. Don't be alarmed when you see them.



Now that we've talked about legitimate drivers, let's focus on how malicious drivers can fit into the picture.

A favorite technique of malware is to insert itself in the driver layers. There are many existing techniques to do this, and more are being invented all the time. The simplest method is to insert a malicious driver between two legitimate drivers. For example, if a piece of malware wanted to hide files on the disk, it could insert a driver between the NTFS driver and the operating system. When a user process, say notepad.exe, requests to read files on the disk, the resulting data from the NTFS driver can be intercepted and edited before being presented to the user. The "hidden" files will still be stored on the disk, but won't be viewable by the user.



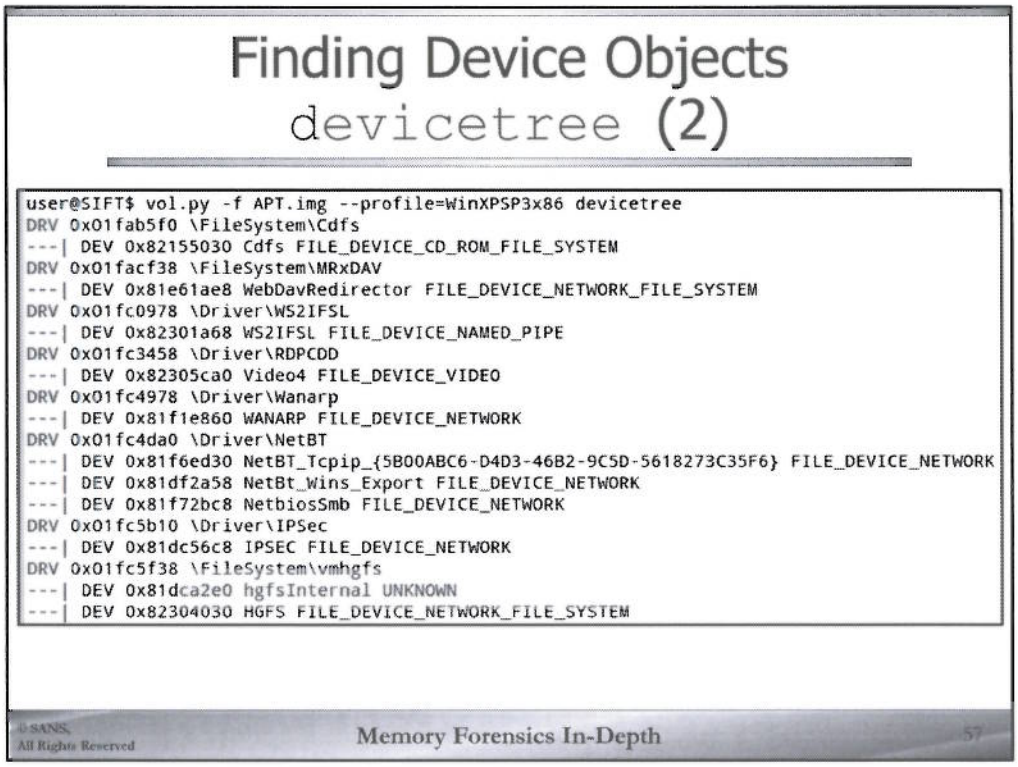
The devicetree plugin enumerates drivers and their associated device objects. Additionally, it shows the hierarchy of device objects. Some device objects have attached drivers. These drivers are the next layer.

The devicetree plugin inherits from the driverscan plugin as it's base class. It is implemented in volatility/plugins/malware/devicetree.py .

Note that the HidUsb driver has an unnamed device. This device in turn has an attached driver 'mouhid'. Finally, the mouhid driver has an attached driver Mouclass. This output shows the relationships between these drivers that might not otherwise be obvious simply by running the modules command.

```
DRV 0x023577e0 \Driver\HidUsb
---| DEV 0x822c0658 00000062 FILE_DEVICE_UNKNOWN
------| ATT 0x81f71020 - \Driver\mouhid FILE_DEVICE_MOUSE
---------| ATT 0x81f111d0 PointerClass3 - \Driver\Mouclass FILE_DEVICE_MOUSE
```

```
vol.py -f APT.img --profile=WinXPSP3x86   modules |egrep -i 'mouclass|hidusb|mouhid'
Volatility Foundation Volatility Framework 2.3.1
0x82331c50 mouclass.sys      0xf895a000    0x6000 \SystemRoot\system32\DRIVERS\mouclass.sys
0x8219ae70 hidusb.sys        0xf82ba000    0x3000 \SystemRoot\system32\DRIVERS\hidusb.sys
0x82197c18 mouhid.sys        0xf8167000    0x3000 \SystemRoot\system32\DRIVERS\mouhid.sys
```



Below is sample output for the devicetree command. The devicetree command enumerates driver objects. For each driver object located, it then checks for associated device objects. Identifying the device object may be useful for inferring the purpose of an unknown driver. Additionally, an attacker may change the name of a device driver (and it's signature) to hide from security software. However, sometimes attackers reuse code and continue to use the same device name. Because the devicetree plugin enumerates device objects, it gives investigators a view into this type of attack.

 # vol.py -f APT.img --profile=WinXPSP3x86 devicetree

DRV 0x01fab5f0 \FileSystem\Cdfs

---| DEV 0x82155030 Cdfs FILE_DEVICE_CD_ROM_FILE_SYSTEM

DRV 0x01facf38 \FileSystem\MRxDAV

---| DEV 0x81e61ae8 WebDavRedirector FILE_DEVICE_NETWORK_FILE_SYSTEM

DRV 0x01fc0978 \Driver\WS2IFSL

---| DEV 0x82301a68 WS2IFSL FILE_DEVICE_NAMED_PIPE

DRV 0x01fc3458 \Driver\RDPCDD

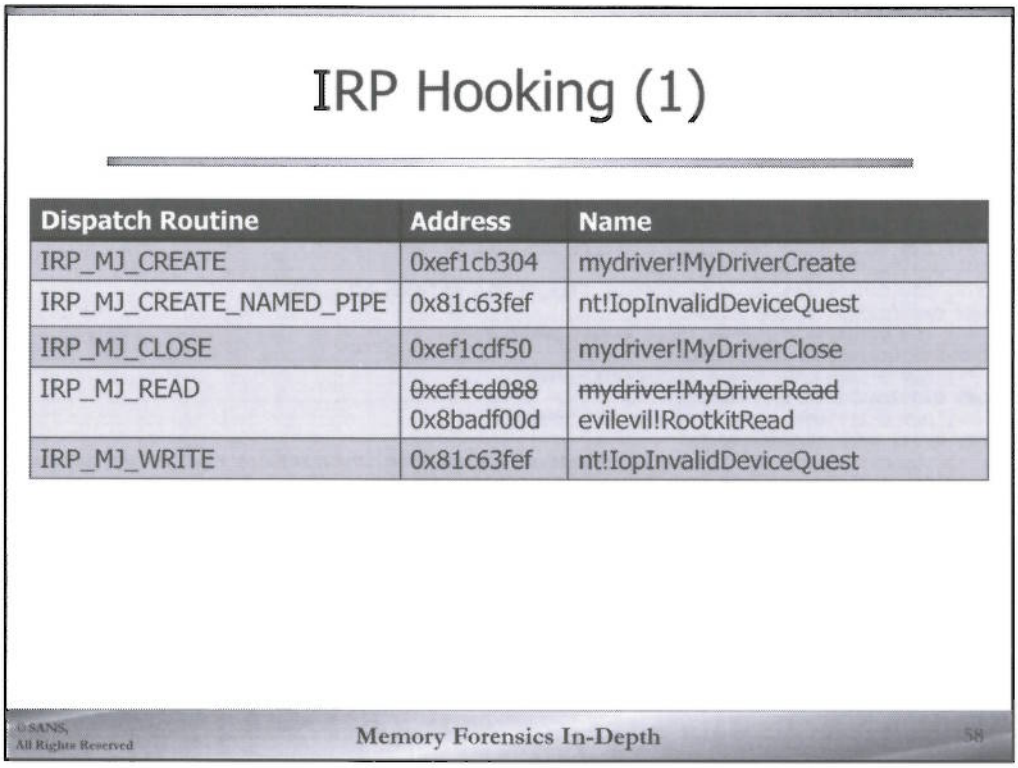
---| DEV 0x82305ca0 Video4 FILE_DEVICE_VIDEO

DRV 0x01fc4978 \Driver\Wanarp

---| DEV 0x81f1e860 WANARP FILE_DEVICE_NETWORK

DRV 0x01fc4da0 \Driver\NetBT

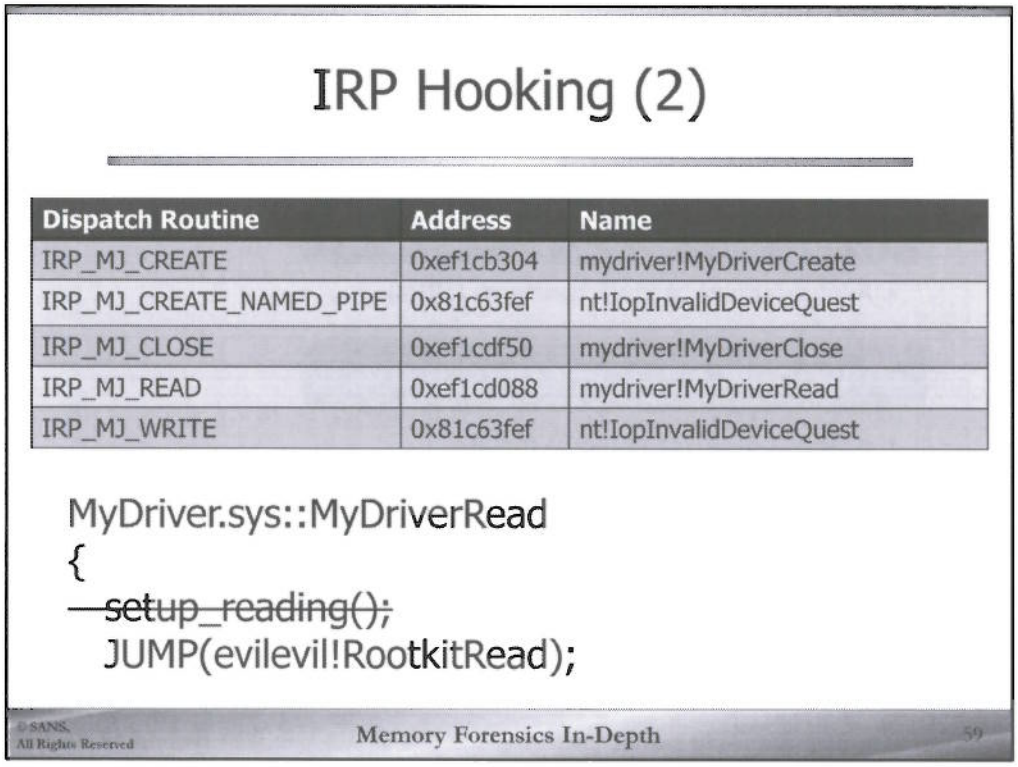
---| DEV 0x81f6ed30 NetBT_Tcpip_{5B00ABC6-D4D3-46B2-9C5D-5618273C35F6}



Inserting a driver can be rather obvious, and malware authors like to avoid detection.

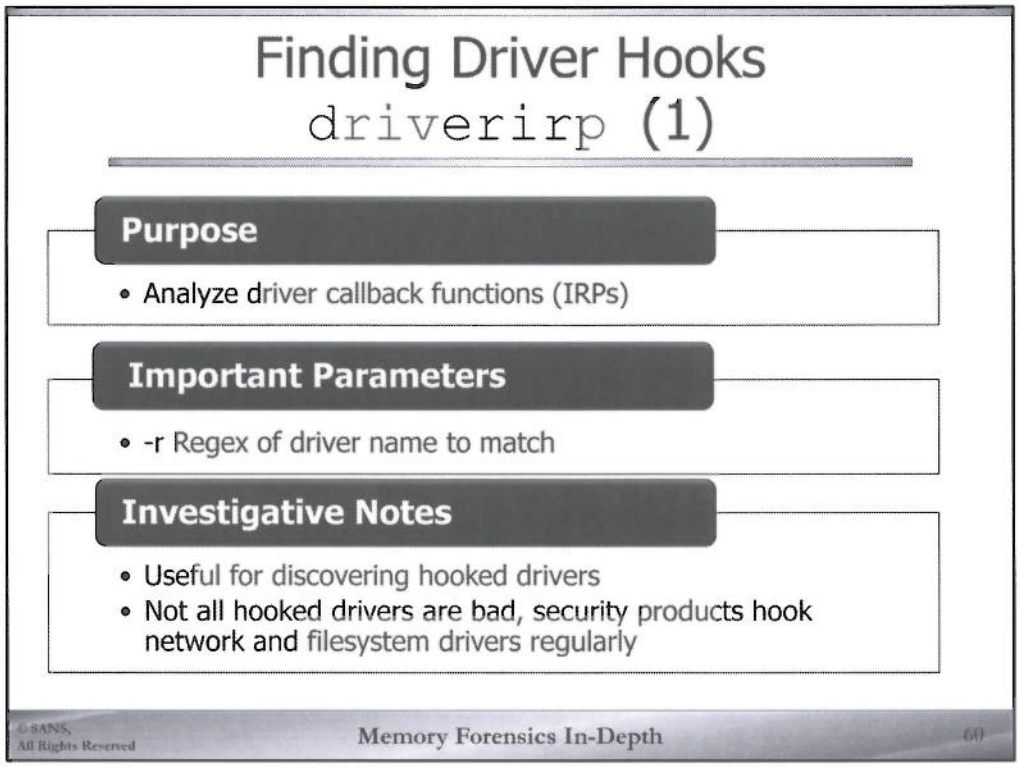
Another technique for hiding a driver is to alter the pointers in a driver's dispatch table. Instead of pointing to the legitimate function in the driver, they will change the pointer to one in another driver. For example, continuing our example of MyDriver.sys, a malicious programmer could overwrite the address of the function used to process IRP_MJ_READ requests with the address of another function.

This technique can also be detected. For each driver we examine, the functions to handle each IRP should be in the code loaded by that driver.



Instead of replacing the entry in the dispatch table, a malicious user could overwrite part of a legitimate code. The example above is stylized, but should get the point across. The real code is in machine language.

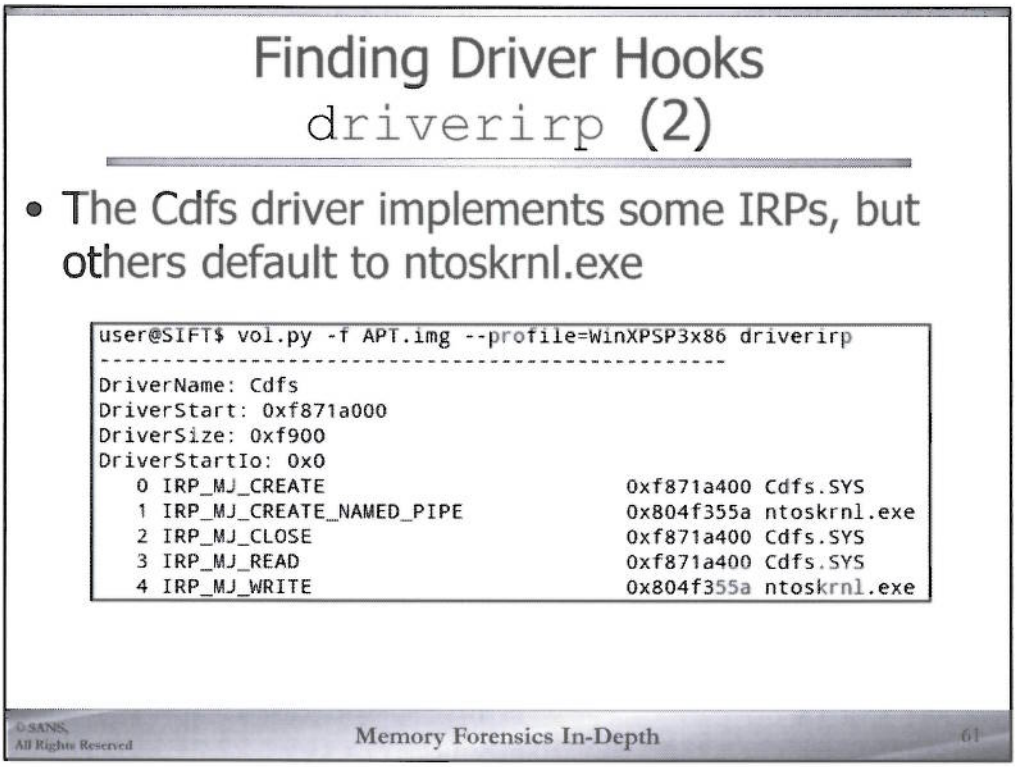
There are semi-automated tools for doing this kind of detection work. The one we will be discussing, driverirp, allows us to spot this type of trampoline hooking by translating the first code sections pointed to be each entry into assembly instructions.



The driverirp plugin is used to enumerate IO request packet (IRP) handlers. These functions are used to perform asynchronous actions in the kernel. Most drivers operate asynchronously, meaning that they do not execute any threads. They only execute code when called upon by another kernel thread. IRP handlers are registers by the driver when loaded for common functions like read and write.

In general, the address for a driver's IRP handler should be in the driver's code section itself. Any IRP that is handled by code outside the driver has been hooked. This isn't always bad, as security products do this regularly. However, these modules with hooked functions should be given extra attention. Note that the IO manager will default to ntoskrnl.exe for any IRP not implemented in the driver.

The driverirp plugin inherits from the driverscan plugin as it's base class. It is implemented in volatility/plugins/malware/devicetree.py



The Cdfs driver implements some IRPs, but some default to ntoskrnl.exe as seen above in the driver's dispatch table. This is completely normal. What we are looking for here to detect malware are IRPs that do not belong to the driver, ntoskrnl.exe, or any security software. The most common functions to hook are IRP_MJ_CREATE, IRP_MJ_READ and IRP_MJ_WRITE.

vol.py -f APT.img --profile=WinXPSP3x86 driverirp

DriverName: Cdfs

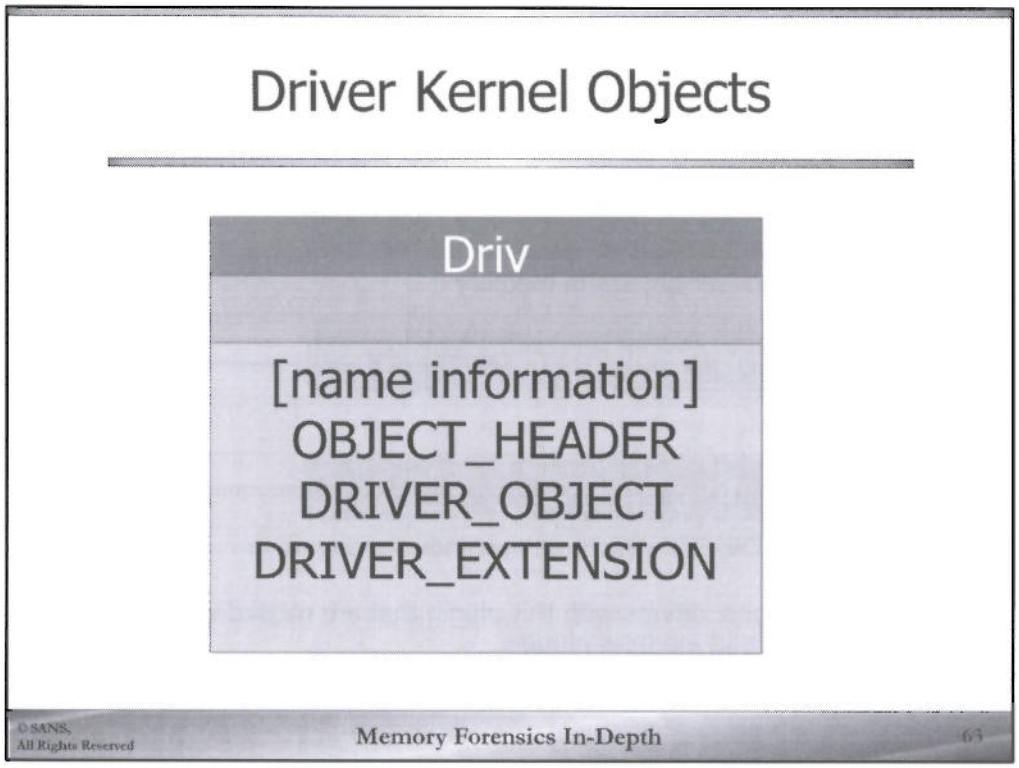
DriverStart: 0xf871a000

DriverSize: 0xf900

DriverStartIo: 0x0

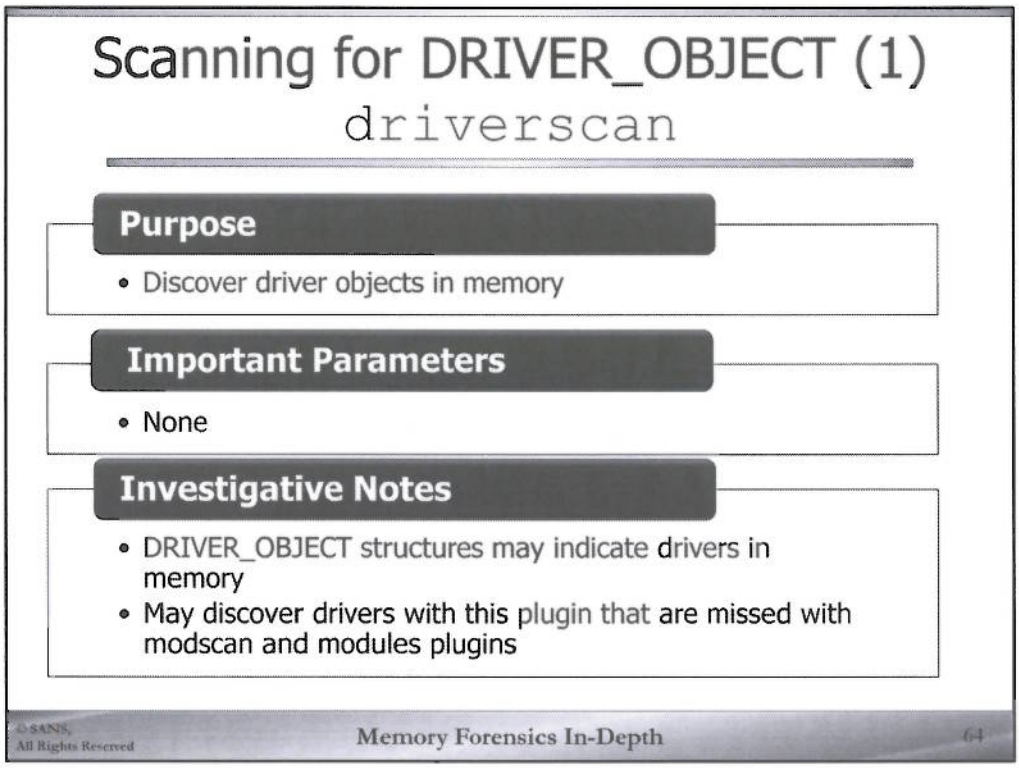
0 IRP_MJ_CREATE	0xf871a400 Cdfs.SYS
1 IRP_MJ_CREATE_NAMED_PIPE	0x804f355a ntoskrnl.exe
2 IRP_MJ_CLOSE	0xf871a400 Cdfs.SYS
3 IRP_MJ_READ	0xf871a400 Cdfs.SYS
4 IRP_MJ_WRITE	0x804f355a ntoskrnl.exe
5 IRP_MJ_QUERY_INFORMATION	0xf871a400 Cdfs.SYS
6 IRP_MJ_SET_INFORMATION	0xf871a400 Cdfs.SYS
7 IRP_MJ_QUERY_EA	0x804f355a ntoskrnl.exe
8 IRP_MJ_SET_EA	0x804f355a ntoskrnl.exe

```
 9 IRP_MJ_FLUSH_BUFFERS                  0x804f355a ntoskrnl.exe
10 IRP_MJ_QUERY_VOLUME_INFORMATION       0xf871a400 Cdfs.SYS
11 IRP_MJ_SET_VOLUME_INFORMATION         0x804f355a ntoskrnl.exe
12 IRP_MJ_DIRECTORY_CONTROL              0xf871a400 Cdfs.SYS
13 IRP_MJ_FILE_SYSTEM_CONTROL            0xf871a400 Cdfs.SYS
14 IRP_MJ_DEVICE_CONTROL                 0xf871a400 Cdfs.SYS
15 IRP_MJ_INTERNAL_DEVICE_CONTROL        0x804f355a ntoskrnl.exe
16 IRP_MJ_SHUTDOWN                       0xf871dc74 Cdfs.SYS
17 IRP_MJ_LOCK_CONTROL                   0xf871a400 Cdfs.SYS
18 IRP_MJ_CLEANUP                        0xf871a400 Cdfs.SYS
19 IRP_MJ_CREATE_MAILSLOT                0x804f355a ntoskrnl.exe
20 IRP_MJ_QUERY_SECURITY                 0x804f355a ntoskrnl.exe
21 IRP_MJ_SET_SECURITY                   0x804f355a ntoskrnl.exe
22 IRP_MJ_POWER                          0x804f355a ntoskrnl.exe
23 IRP_MJ_SYSTEM_CONTROL                 0x804f355a ntoskrnl.exe
24 IRP_MJ_DEVICE_CHANGE                  0x804f355a ntoskrnl.exe
25 IRP_MJ_QUERY_QUOTA                    0x804f355a ntoskrnl.exe
26 IRP_MJ_SET_QUOTA                      0x804f355a ntoskrnl.exe
27 IRP_MJ_PNP                            0xf871a400 Cdfs.SYS
---------------------------------------------------
... output truncated ...
```



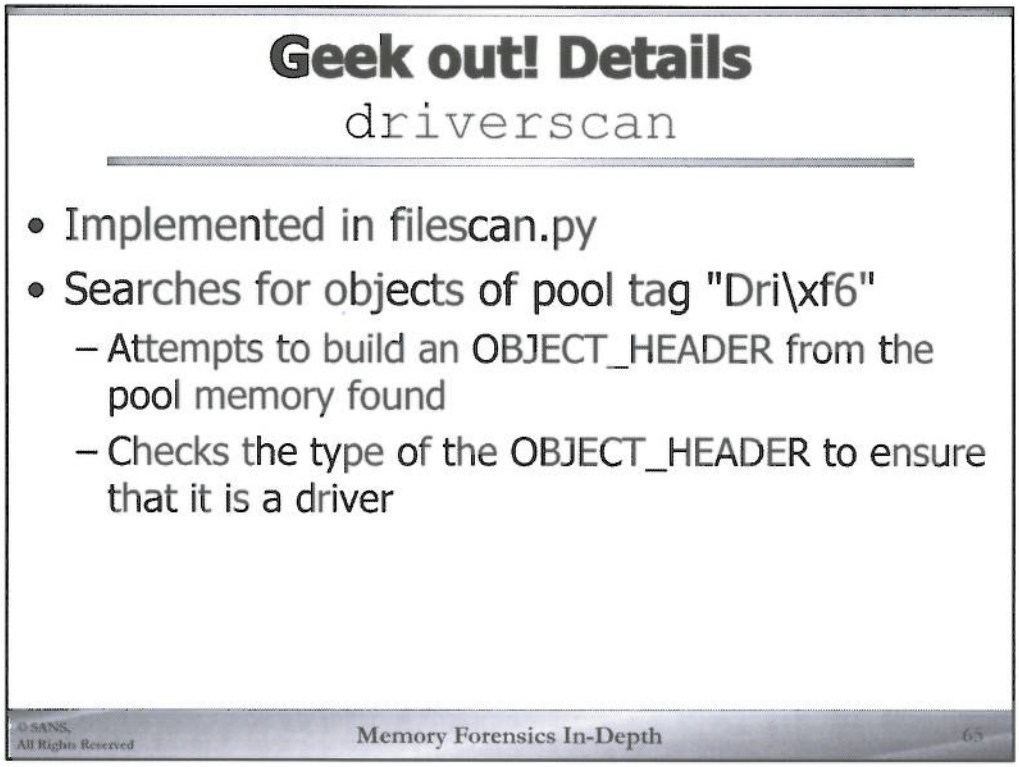
Along with searching for LDR_DATA_TABLE_ENTRY structures, we can also search for kernel objects in the pool memory. Driver information is stored in the non-paged pool with the pool tag "Driv" before the protection bit is set. With the PROTECTED_POOL bit set in the pool tag the actual hex bytes being searched are `44 72 69 f6`. The Volatility plugin 'driverscan' searches for and displays these structures.

There are three structures which are always found together and make up the driver information we can glean. First is an OBJECT_HEADER. Every object in the kernel has one of these structures, which gives the type of object, how many pointers and handles are open to that object, and where the object's name is stored. (The actual name data is above the object header, actually.) After the object header is a DRIVER_OBJECT structure, which contains information about the driver, the addresses of its functions, which devices it controls, and so on. Finally, the DRIVER_EXTENSION structure points to the Service Key Name, which is the registry key where this driver stores its configuration settings (i.e. HKLM\SYSTEM\[ControlSet]\Services).



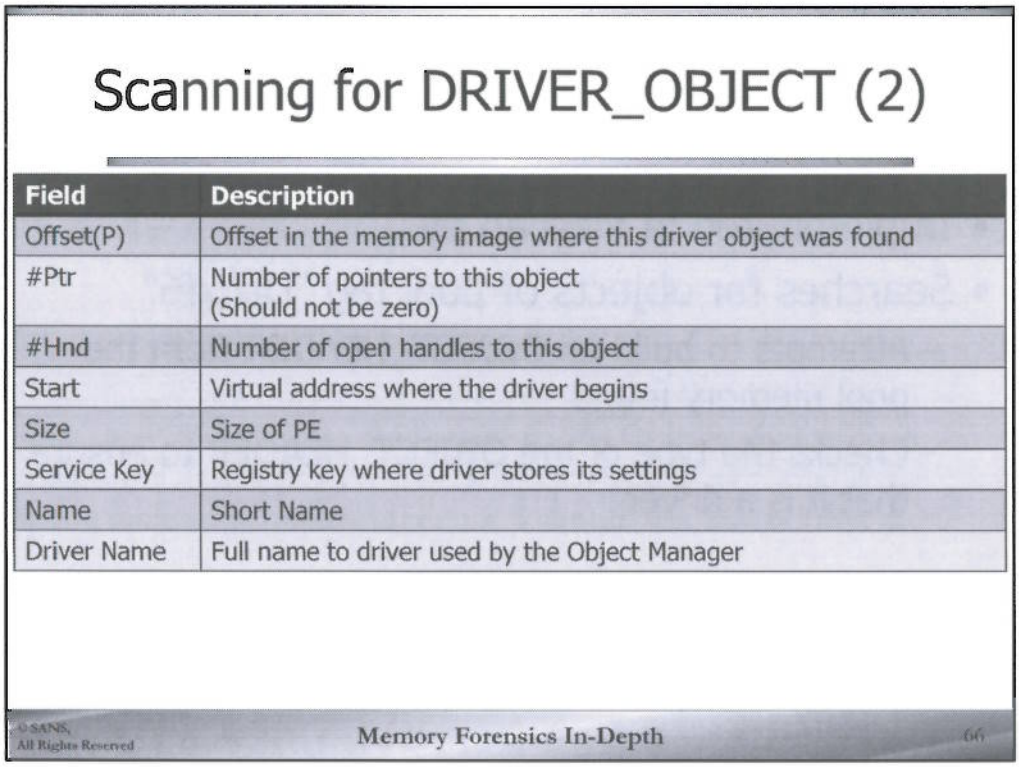
The driverscan plugin is used to locate DRIVER_OBJECT structures in kernel pool memory. A DRIVER_OBJECT may not be present for every module loaded in the kernel so you may observe less output with this plugin than with modules or modscan. Typically those drivers that rely on IRP related callbacks will have DRIVER_OBJECTS (think of a filesystem driver or a driver for a USB device). Those modules that only run (or contribute libraries to) kernel threads are unlikely to have DRIVER_OBJECTS.

The driverscan plugin is important because it offers yet another way to discover code running in the kernel. A rootkit developer loading a filter driver is likely to have instantiated a DRIVER_OBJECT. Even if the rootkit developer removed the driver from the loaded module list, driverscan still may be able to locate the DRIVER_OBJECT structure.

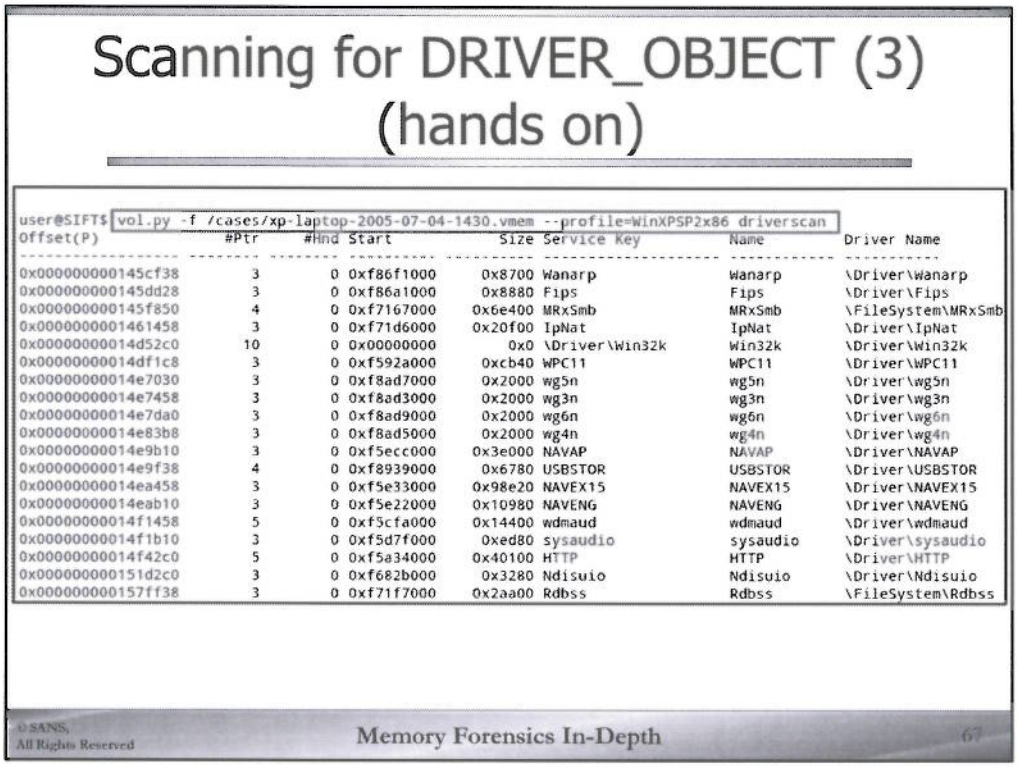


The driverscan plugin is implemented in the file volatility/plugins/filescan.py.

It searches memory for memory allocated with the "Dri\xf6" pool tag. When memory with that pool tag is located, it then builds an OBJECT_HEADER based on the position of the DRIVER_OBJECT. The OBJECT_HEADER is checked to see if the type of the OBJECT is a driver.



Here are the fields which the driverscan plugin outputs. Again, we don't get the virtual address where these structures were found because it's a brute-force search. We only get the physical offset in the memory image where they were found. Drivers which the operating system doesn't know about may also have zero pointers to them. If the number of pointers field is zero, the driver in question may be suspicious.



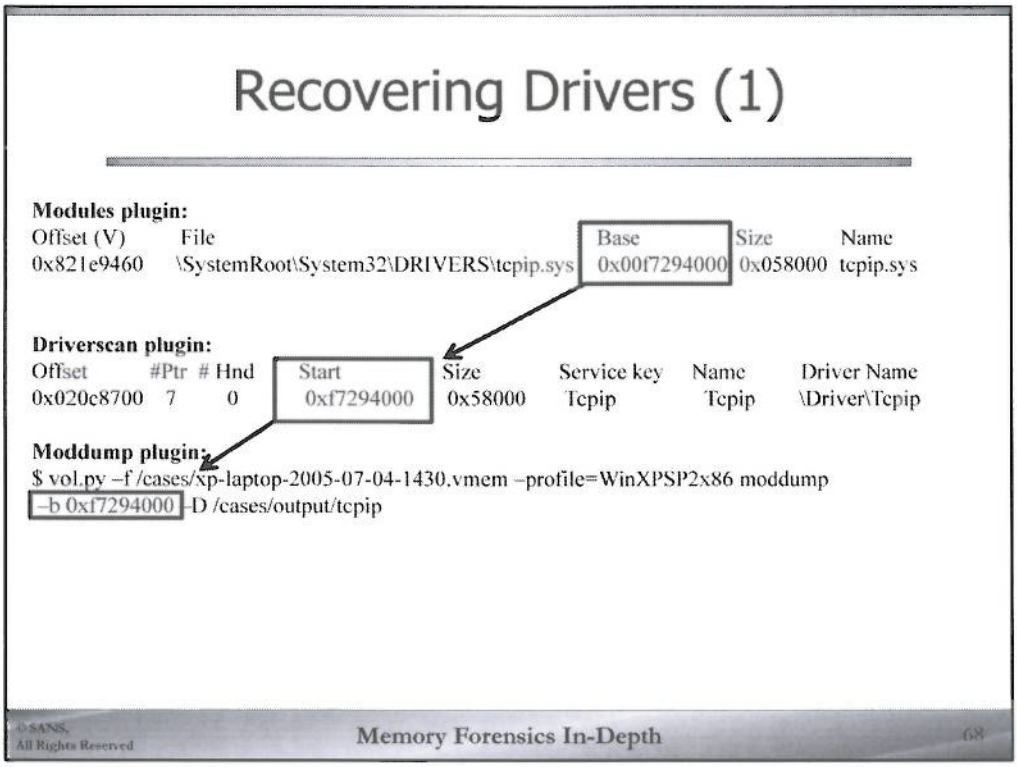
We're going to do a hands-on exercise now with the driverscan plugin. The command line is similar to the others we have used in this section:

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

You should see output which starts with:

Offset	Obj Type	#Ptr	#Hnd	Start	Size	Service key	Name	
0x0145cf38	0x823b25b8	3	0	0xf86f1000	34560	'Wanarp'	'Wanarp'	'\\Driver\\Wanarp'
0x0145dd28	0x823b25b8	3	0	0xf86a1000	34944	'Fips'	'Fips'	'\\Driver\\Fips'
0x0145f850	0x823b25b8	4	0	0xf7167000	451584	'MRxSmb'	'MRxSmb'	'\\FileSystem\\MRxSmb'
0x01461458	0x823b25b8	3	0	0xf71d6000	134912	'IpNat'	'IpNat'	'\\Driver\\IpNat'

Don't worry if you see a driver's name is blank. The unicode string which contained the name may have been paged out. Also, remember this is a brute force scan, and it may find data which isn't actually a driver object. It's unlikely, due to the pool header verification, but it is always a possibility.



Looking at all of these modules in memory is nice, but if we're going to do any kind of serious analysis we will need to pull out a copy of these drivers from memory. Thankfully, this is not hard, and Volatility includes a plugin, moddump, to do this for us. The plugin uses the virtual address of the driver's base address which must be specified in hexadecimal. You can get this value from the "Base" field in the modules or modscan plugins or the "Start" field in driverscan. For example, let's say we wanted to recover a sample of the TCPIP DLL, tcpip.sys, from the xp-laptop image.

We will need the virtual address of the base of the tcpip.sys driver, which we can get from either the modules, modscan, or driverscan plugins. Here's the output from modules:

```
Offset(V)  File                                    Base          Size      Name
0x821e9460 \SystemRoot\System32\DRIVERS\tcpip.sys  0x00f7294000 0x058000 tcpip.sys
```

This gives us the base address of 0xf7294000.

The moddump plugin also needs a command line flag specifying where it should write the output files. You can't use the tilde shortcut in this command line option. Instead you must specify the directory name. Thus, the full command line to dump out the tcpip.sys driver from the xp laptop memory image would be:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
moddump -b 0xf7294000 --dump-dir=/cases/output
```

The result is stored in the dump-directory as driver.[virtualaddress].sys. In this case we should get a file driver.f7294000.sys.

Modules plugin:

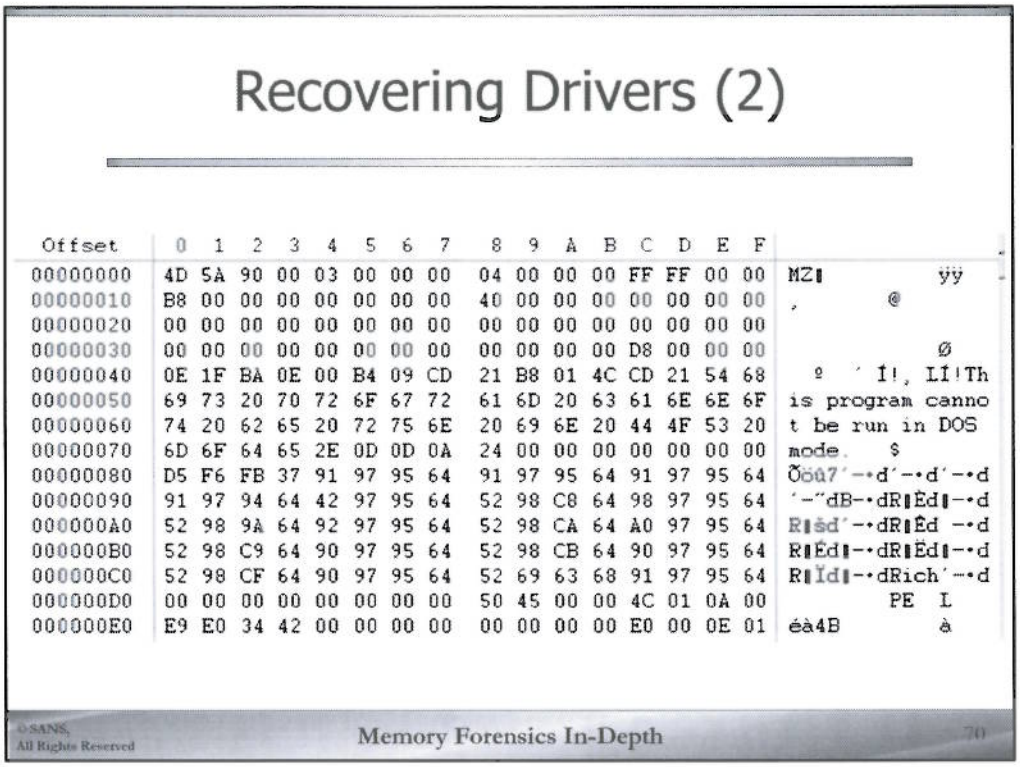
```
Offset(V)  File                                      Base         Size     Name
0x821e9460 \SystemRoot\System32\DRIVERS\tcpip.sys    0x00f7294000 0x058000 tcpip.sys
```

driverscan plugin:

```
Offset     Obj Type    #Ptr #Hnd Start      Size Service key          Name
0x020c8700 0x823b25b8    7    0 0xf7294000 359808 'Tcpip'             'Tcpip'
```

moddump plugin:

```
$ vol.py -f ~/Desktop/cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86 moddump -o
    0xf7294000 --dump-dir=/home/sansforensics/output
```



Let's take a look at the kind of data extracted from the memory image using the file command:

```
$ file /home/sansforensics/output/driver.f7294000.sys
/home/sansforensics/output/driver.f7294000.sys: PE32 executable (native)
Intel 80386, for MS Windows
```

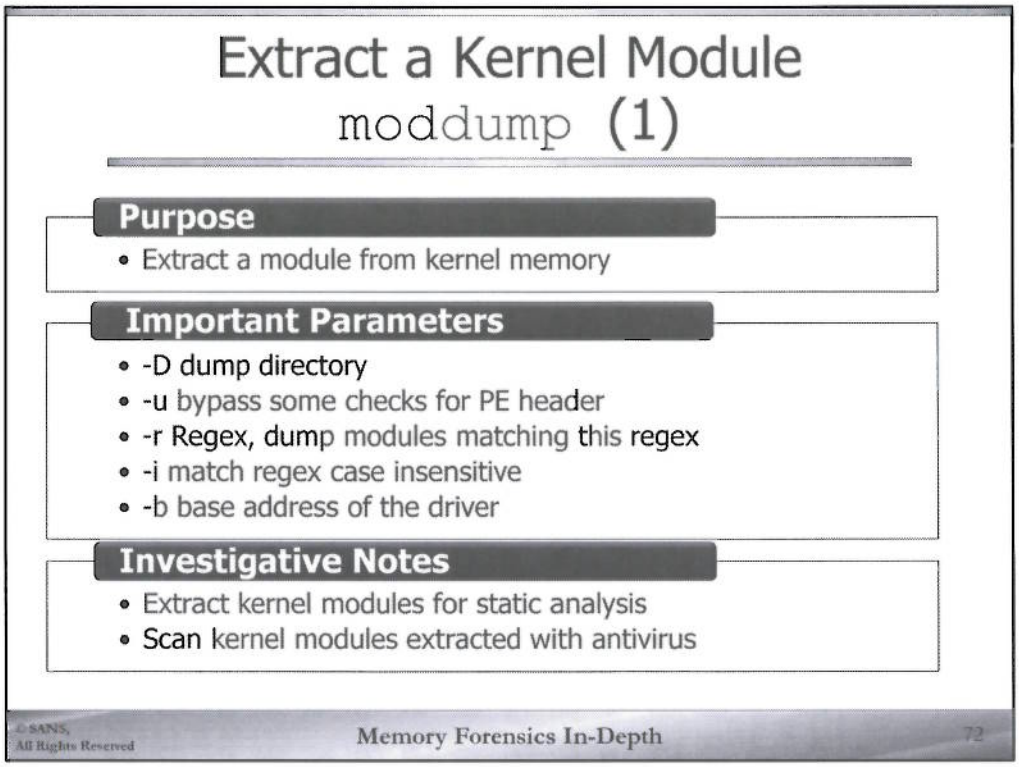
So far, so good! We extracted a Windows executable. Looking at the start of the file in a hex editor, above, we can clearly see the MZ and PE headers. (The PE header is at offset 0xd8.)

The file we've recovered is not the same as the original on the disk before being loaded by Windows. When executables are loaded, there are small changes made in the code. Functions get remapped, parts moved around, and so on. Although Volatility reverses some of these changes, it can't undo all of them. As a result, searching for this exact file in a list of known files, such as the NSRL, won't get a hit.

But this doesn't mean you can't use existing resources to save yourself some analysis time. Before you begin your own analysis, try looking up your recovered sample in the NSRL (you never know, it might work!), submitting it to Virus Total, or scanning it with your anti-virus software. There's no sense in repeating what's been done before.

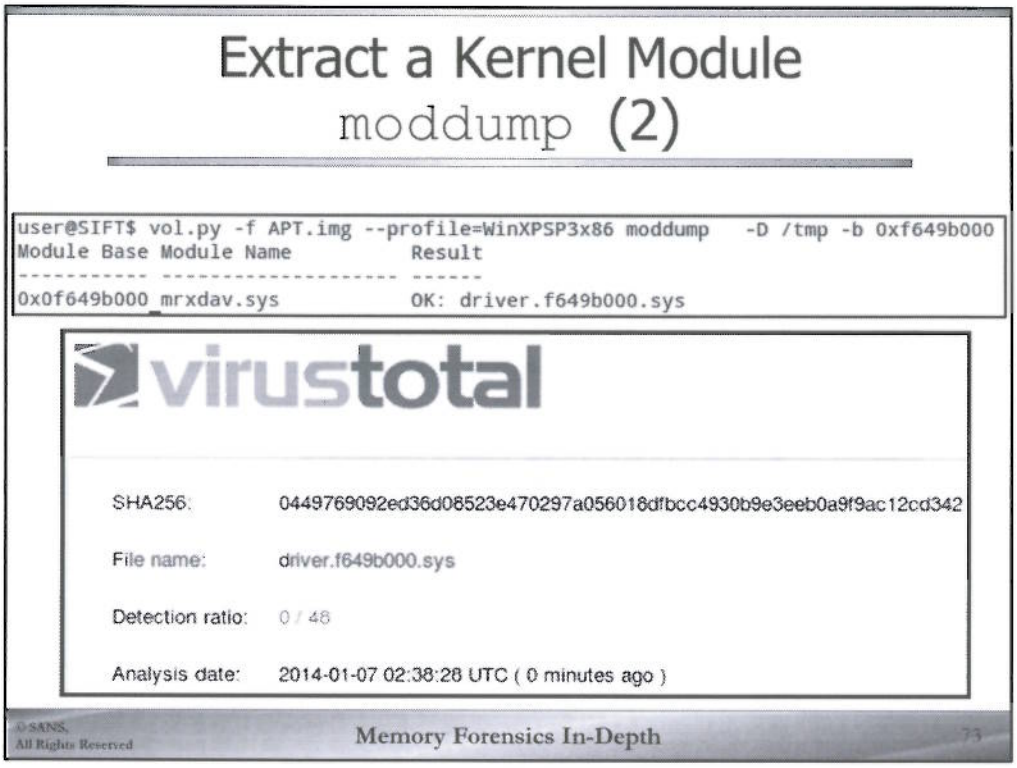
Virus Total, http://virustotal.com/, is a free, web-based service for scanning files against a few dozen anti-virus engines. You can get a report back quickly with the results of each anti-virus engine listed separately. Keep in mind that anything you submit to this site may be passed along to the anti-virus companies, and presumably other entities as well. Don't submit anything you don't want to be made public!

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ▮ ÿÿ
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸ @
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000030	00	00	00	00	00	00	00	00	00	00	00	00	D8	00	00	00	Ø
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	º ´ Í!¸ LÍ!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode. $
00000080	D5	F6	FB	37	91	97	95	64	91	97	95	64	91	97	95	64	Õöû7´-•d´-•d´-•d
00000090	91	97	94	64	42	97	95	64	52	98	C8	64	98	97	95	64	´-"dB-•dR▮Èd▮-•d
000000A0	52	98	9A	64	92	97	95	64	52	98	CA	64	A0	97	95	64	R▮šd´-•dR▮Êd -•d
000000B0	52	98	C9	64	90	97	95	64	52	98	CB	64	90	97	95	64	R▮Éd▮-•dR▮Ëd▮-•d
000000C0	52	98	CF	64	90	97	95	64	52	69	63	68	91	97	95	64	R▮Ïd▮-•dRich´-•d
000000D0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	0A	00	PE L
000000E0	E9	E0	34	42	00	00	00	00	00	00	00	00	E0	00	0E	01	éà4B à



The moddump plugin extracts modules from kernel memory. The plugin attempts to validate that the base address (-b) is valid by performing a number of checks for a valid PE header. This check can be bypassed by using the unsafe (-u) option. If you identify an unknown module using the modules or modscan plugin, you can extract it using the moddump plugin. This allows you to run strings against the extracted module or scan it using antivirus software.

It is implemented in volatility/plugins/moddump.py



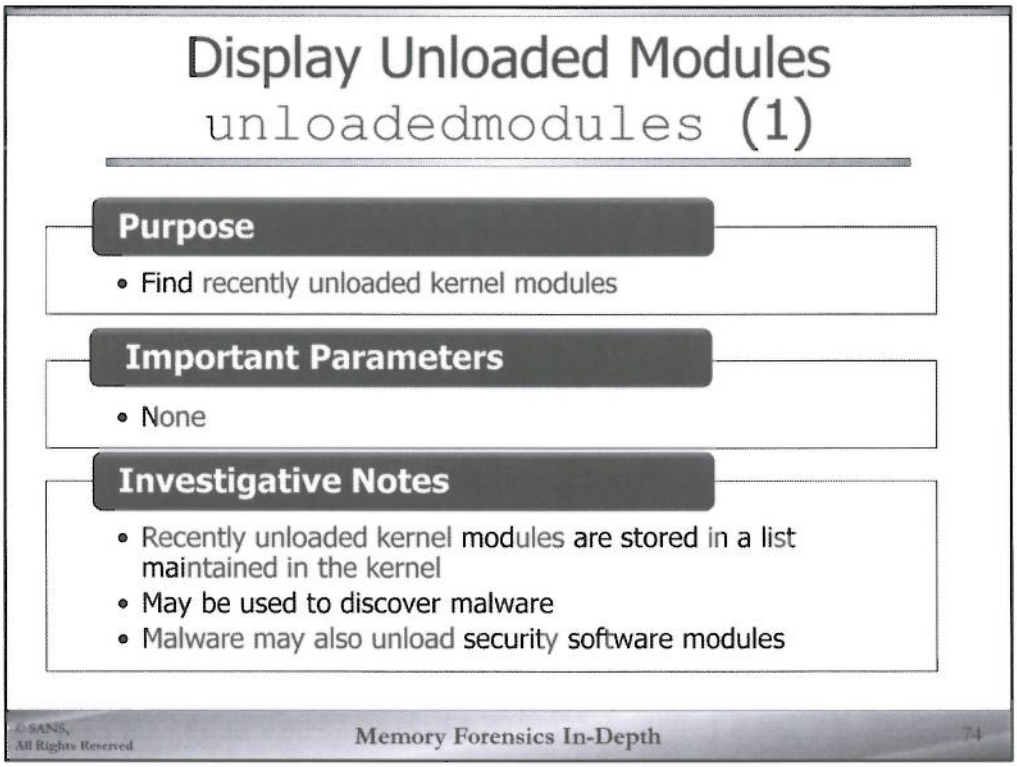
Wow, we just found a possibly suspicious kernel module using the modscan module. I was just reading a report on stuxnet and saw that it uses the driver mrxnet.sys. Then I saw the module mrxdav.sys. I wonder if this is a stuxnet variant? To find out, let's dump the module and scan it with virustotal.

 # vol.py -f APT.img --profile=WinXPSP3x86 moddump -D /tmp -b 0xf649b000

Module Base Module Name Result

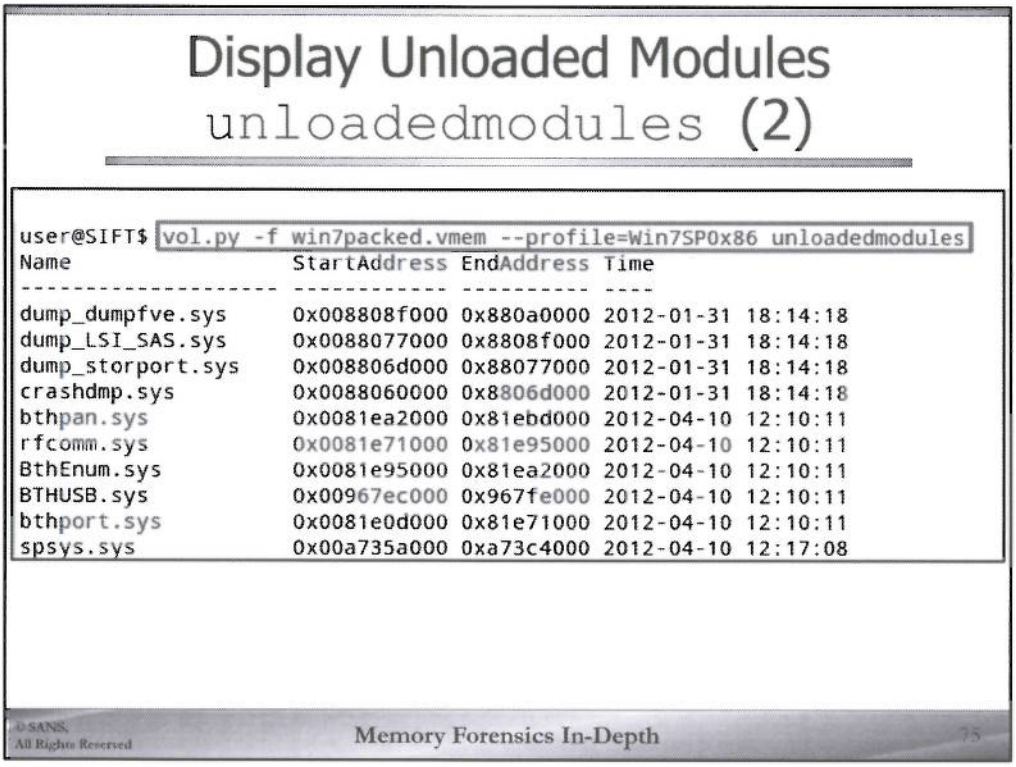
---------- -------------------- ------

0x0f649b000 mrxdav.sys OK: driver.f649b000.sys



The unloadedmodules plugin is used to enumerate kernel modules that have been recently unloaded. The Windows kernel maintains a list of those modules that have recently been unloaded (for some definition of recent). This is useful for debugging since recently unloaded kernel modules may have leave dangling pointers, causing a bugcheck (or BSOD).

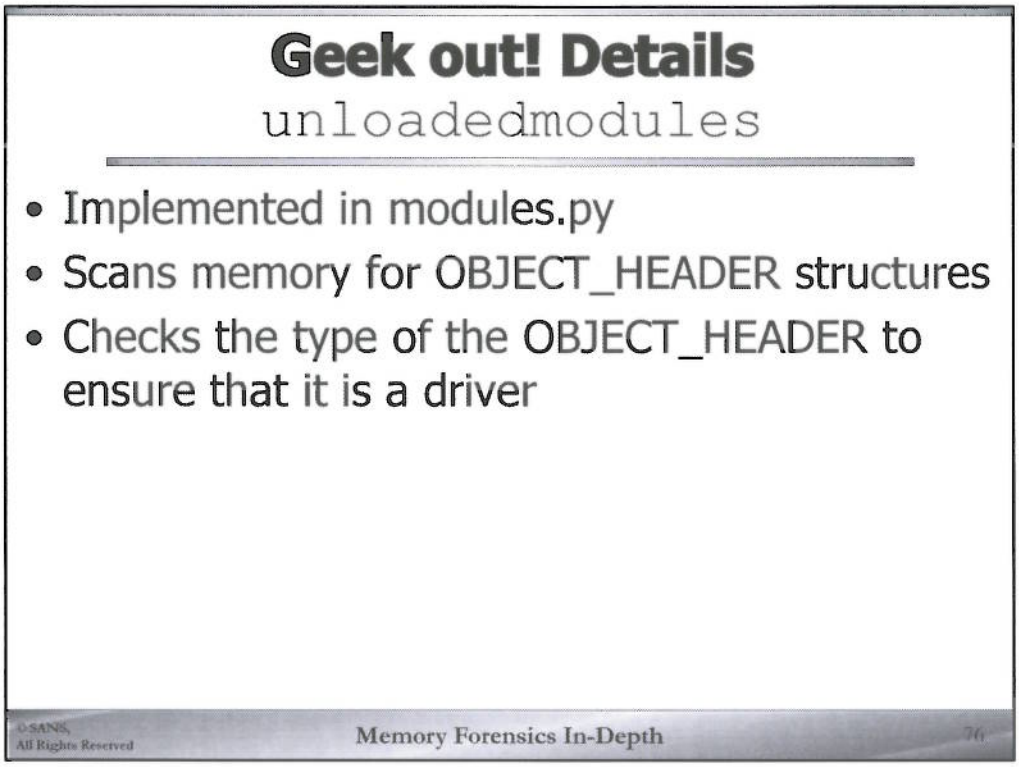
If an attacker uses a kernel module to perform some action (like dumping physical memory or hiding a process) and then unloads the module, the module will be in the recently unloaded modules list for some time. Similarly, if an attacker unloads a security software filter driver so she won't be caught performing some malicious action, this driver will also be in the recently unloaded modules list.



Sample output from the unloadedmodules plugin is shown below:

user@SIFT$ vol.py -f win7packed.vmem --profile=Win7SP0x86 unloadedmodules

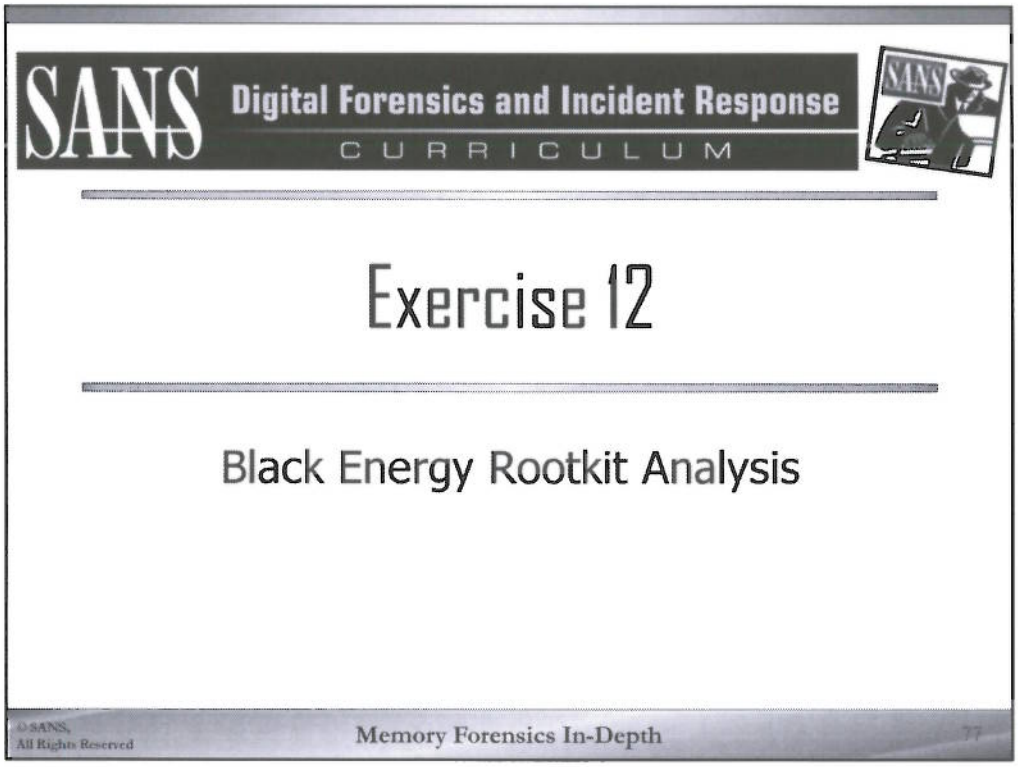
```
Name                 StartAddress EndAddress Time
-------------------- ------------ ---------- ----
dump_dumpfve.sys     0x008808f000 0x880a0000 2012-01-31 18:14:18
dump_LSI_SAS.sys     0x0088077000 0x8808f000 2012-01-31 18:14:18
dump_storport.sys    0x008806d000 0x88077000 2012-01-31 18:14:18
crashdmp.sys         0x0088060000 0x8806d000 2012-01-31 18:14:18
bthpan.sys           0x0081ea2000 0x81ebd000 2012-04-10 12:10:11
rfcomm.sys           0x0081e71000 0x81e95000 2012-04-10 12:10:11
BthEnum.sys          0x0081e95000 0x81ea2000 2012-04-10 12:10:11
BTHUSB.sys           0x00967ec000 0x967fe000 2012-04-10 12:10:11
bthport.sys          0x0081e0d000 0x81e71000 2012-04-10 12:10:11
spsys.sys            0x00a735a000 0xa73c4000 2012-04-10 12:17:08
```



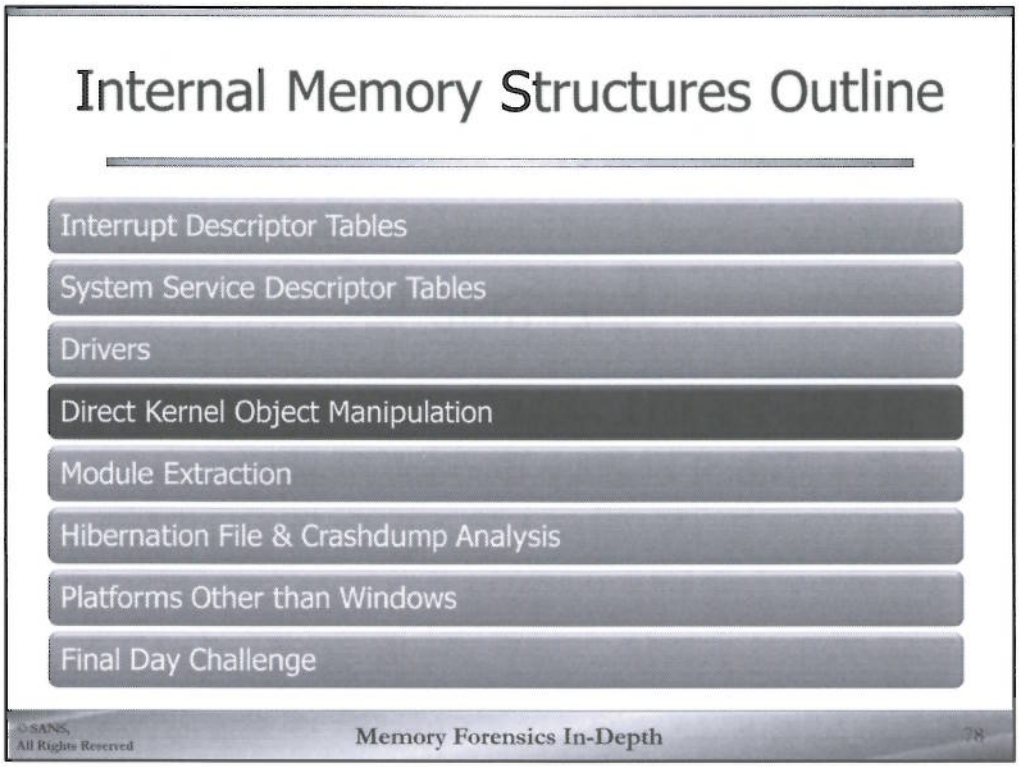
Like many plugins, the unloadedmodules plugin relies heavily on the KDBG. Remember that the whole reason that Windows tracks unloaded modules is for debugging errors. Therefore, it makes sense that KDBG would have a link to the recently unloaded modules list.

To locate the unloaded modules list, first the KDBG is referenced. Then one of the members, MmUnloadedDrivers is double dereferenced to find the address of the unloaded modules list. This list is not circularly linked, but the KDBG contains a pointer to the last entry in the list. The list contains members of the type _UNLOADED_DRIVER. A sample definition for this struct is shown below.

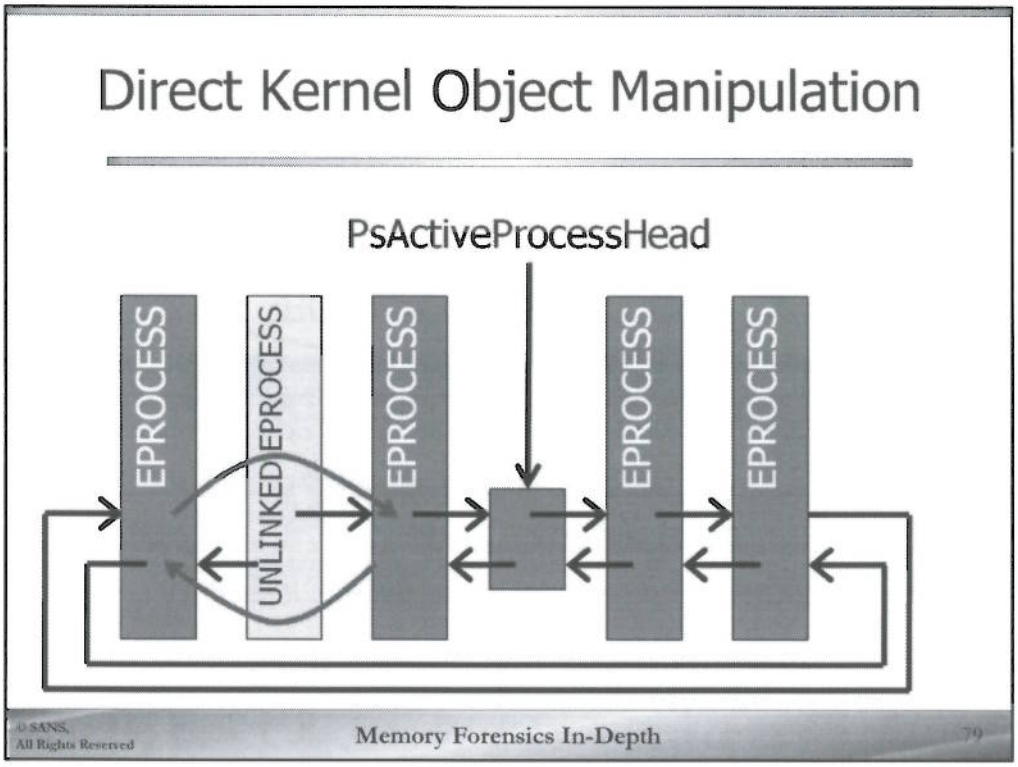
```
struct _UNLOADED_DRIVER {
        _UNICODE_STRING Name;
        uint StartAddress;
        uint EndAddress;
        WinTimeStamp CurrentTime;
};
```



This page intentionally left blank.



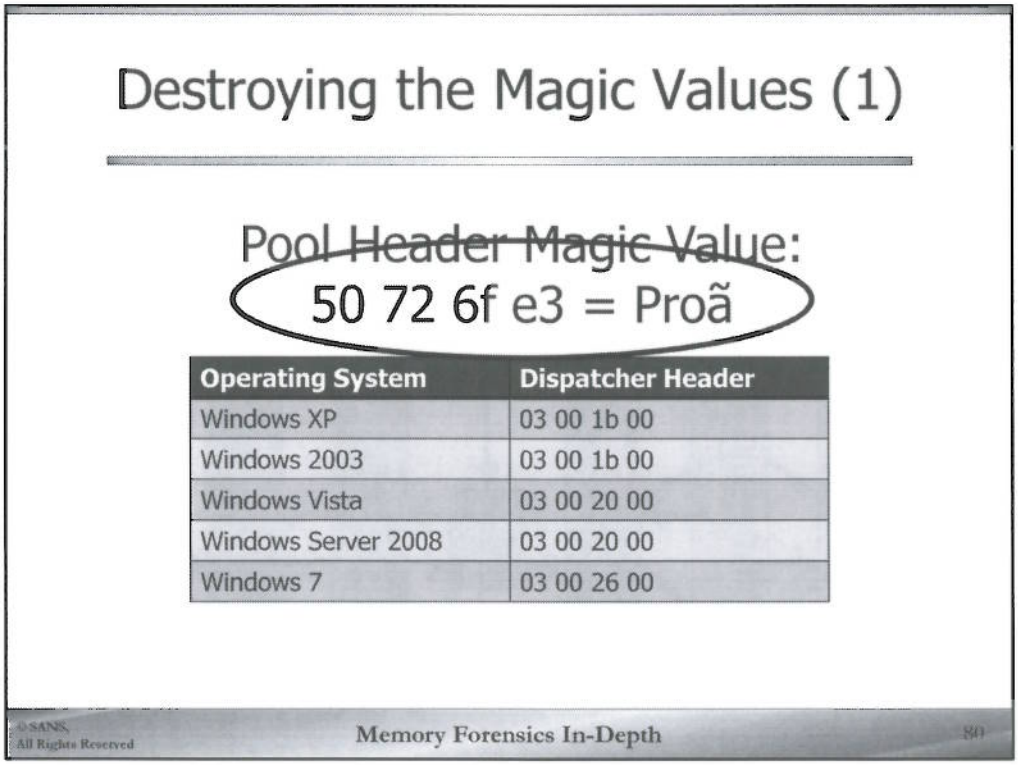
This page intentionally left blank.



Earlier we briefly mentioned how a malicious user could unlink a process from the list of active processes. They could redirect the active process links to skip over a hidden process and instead point to each other. The unlinked process wouldn't show up in the list of active processes. We still found such processes during our brute force searching, but it's hidden from a walk of the process list.

This is an example of Direct Kernel Object Manipulation (DKOM). Whomever was making this change was deliberately making changes to kernel objects without using the Windows APIs. It's dangerous because Windows will still attempt to use the modified data structures. When all of those pointers don't quite line up anymore—when not everything is consistent—what happens. In computer science terms, the results are "undetermined", which is the academic way to denote, "Not only do we not know, but there is no way on earth to know."

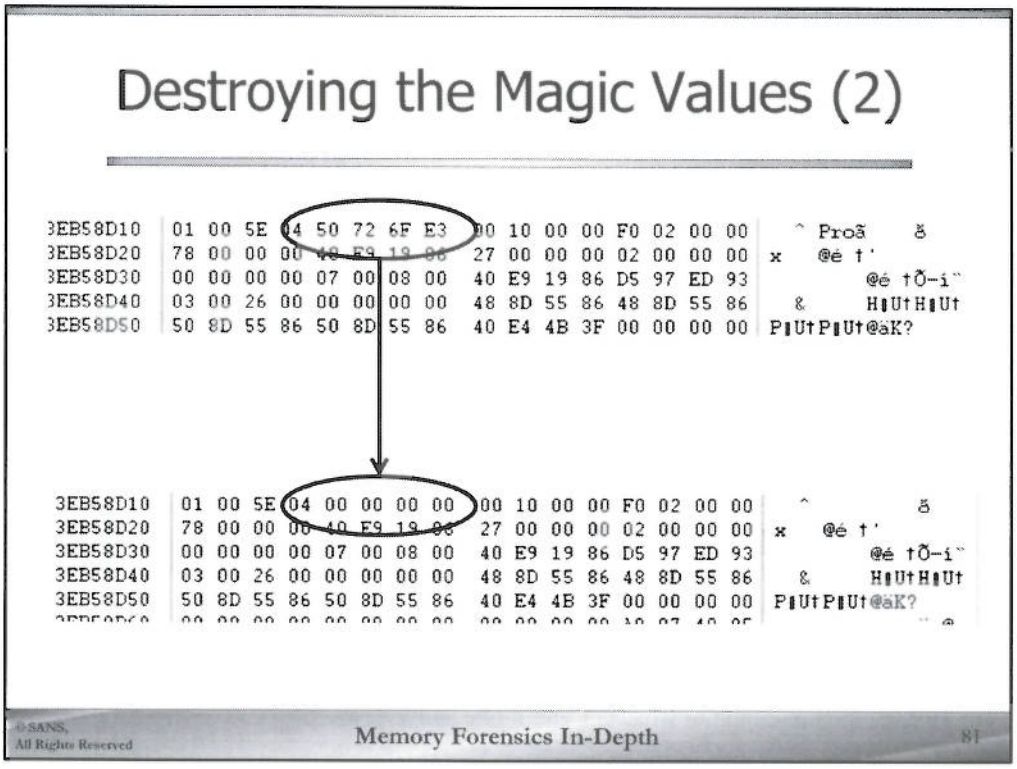
Such changes may affect the stability of the operating system, but do not necessarily affect our memory forensics techniques. We were still able to find the unlinked process during our brute force search. Although the pointers between processes had been changed, each process object still had the magic values which denoted them as processes. Our search method used those magic values to identify and validate potential processes.



As it turns out, those magic values aren't necessarily so magic. We used two magic values found at the start of each EPROCESS block to identify them as such. The first was the protected pool tag and the second was the dispatcher header. The value of the dispatcher header varied slightly per operating system, but there were only three legal values for our target operating systems.

As it turns out, changing the bytes in the pool header while the process is running has no effect on the system. Nothing happens. The operating system continues to function, the process continues to run, and the stability of the system is not impacted. The only real consequence is that the process is no longer visible to most memory forensics tools. Volatility, like many others, attempts to use the pool header to validate potential processes. If the header isn't there, Volatility doesn't include the potential process in the list of actual processes.

Normally these changes would be made by a driver which--as the name suggests--is directly modifying the kernel structures. Of course, that wouldn't be much fun to watch, so we're going to do it manually.



If you're in the live class, your instructor will now demonstrate three different kinds of DKOM.

1. A change will affects the values displayed by memory forensics tools but doesn't affect the operating system.
2. A change which affects the artifacts found by memory forensics tools but doesn't affect the operating system.
3. A change which affects the operating system.

If you're not in a live class, you can try this yourself. You'll need a virtual machine running Windows 7, and both Volatility and a hex editor on the host system. In class we'll be using a 32-bit version of Windows 7.

Here are the steps your instructor is doing:

1. Start the virtual machine.
2. Run Notepad in the VM.
3. Suspend the virtual machine. The VM software should save the memory of the machine to the disk as a single file. VMware, for example, saves the memory to a file in the VM's directory with a .vmem extension. If you've taken some snapshots with the VM, there may be a dash and some hex digits following the VM's name. For example, for the virtual machine "foo", you may have to look for "foo-a5734d2a.vmem".
4. Run the Volatility plugin 'psscan' on the VM's memory image file. This should be something like:

```
$ vol.py -f /home/user/virtual-machines/foo/foo-a57434d2a.vmem --
profile=Win7SP1x86 psscan
```

Find the entry for the notepad process in the psscan output. Note that we can see this valid process in the brute force scan for processes. Note the offset of the EPROCESS block. For example:

```
0x3eb58d40 notepad.exe        2792    352 0x3f4be440 2013-03-21 17:14:51
```

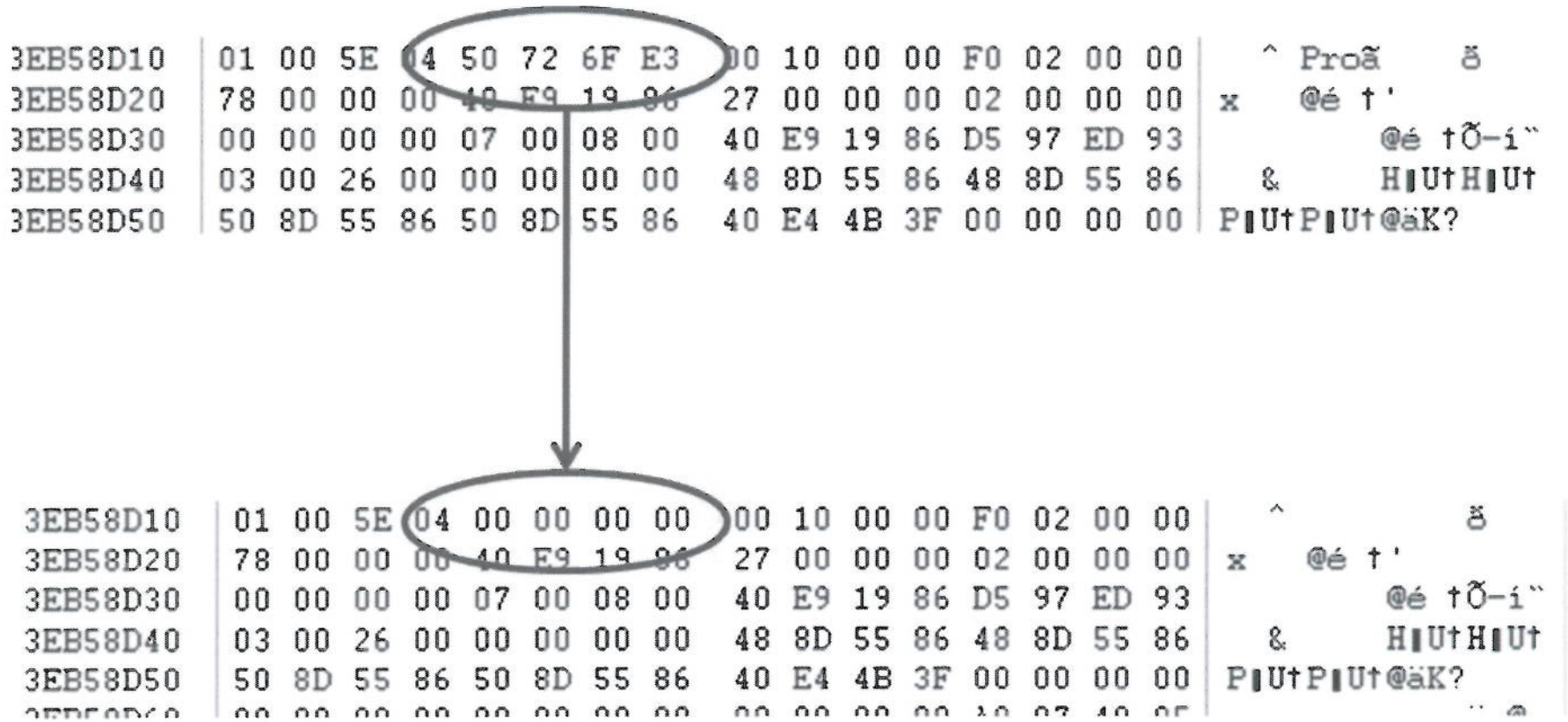
The first value, 0x3eb58d40, is the physical offset of the EPROCESS block in the file. The value on your system will be different.

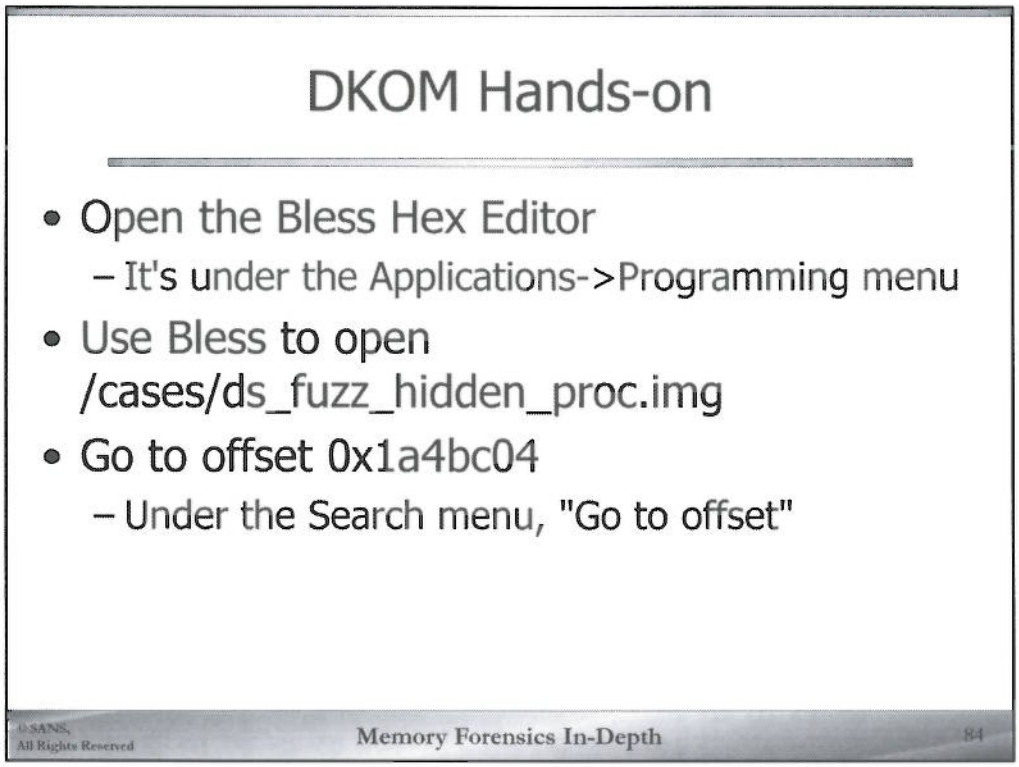
If your host is Microsoft Windows, you can try one of the products listed at http://en.wikipedia.org/wiki/Comparison_of_hex_editors. The screenshot on the previous page was generated from WinHex, which is a commercial product. Although you can (and should) try the free evaluation version, the evaluation version does not let you save large files.

5. Using your hex editor, open the memory image file and go to the offset of the Notepad process' EPROCESS block. You should see the pool tag for the process, the bytes 50 72 6f e3. (The offset reported by Volatility should take you to the Dispatcher Header, but the Pool Tag, slightly above, will be easier to recognize.)
6. Find the timestamp value in the EPROCESS block. It will be 0xa0 bytes after the offset of the EPROCESS block.
7. Change the timestamp to whatever you'd like and save the file.
8. Run the Volatility psscan command again. You should see the timestamp change.
9. Restart the virtual machine. Notice how nothing changes!

10. Suspend the virtual machine again and open the memory image in your hex editor.
11. Replace the Proa header with zeros, as shown on the previous page, and save the file.
12. Run the psscan plugin again. You should see that the notepad process has disappeared from the list!
13. Restart the virtual machine. Again, notice that everything still works just fine. We've created a hidden process!

14. Suspend the virtual machine once more and open the memory image in your hex editor.
15. Replace the dispatcher header with zeros and save the file.
16. When you restart the virtual machine again, it should immediately crash.

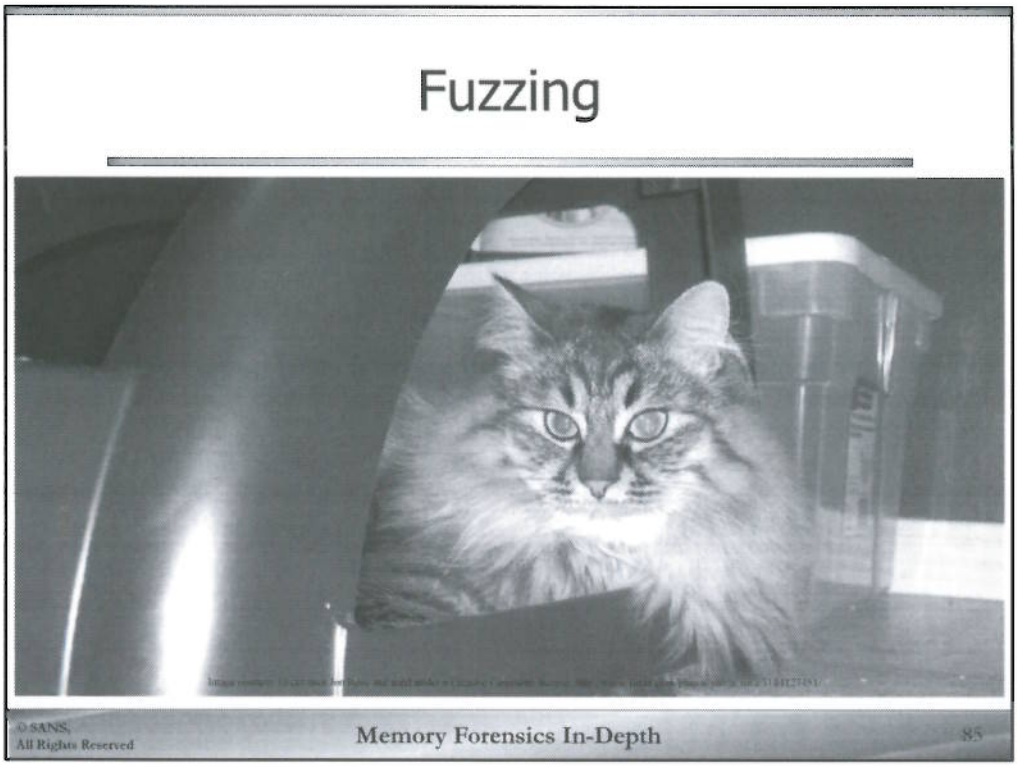

3EB58D10 01 00 5E 04 50 72 6F E3 00 10 00 00 F0 02 00 00 ^ Proã ð
3EB58D20 78 00 00 00 40 E9 19 86 27 00 00 00 02 00 00 00 x @é †'
3EB58D30 00 00 00 00 07 00 08 00 40 E9 19 86 D5 97 ED 93 @é †Õ-í"
3EB58D40 03 00 26 00 00 00 00 00 48 8D 55 86 48 8D 55 86 & H Ut H Ut
3EB58D50 50 8D 55 86 50 8D 55 86 40 E4 4B 3F 00 00 00 00 P Ut P Ut @äK?
3EB58D10 01 00 5E 04 00 00 00 00 00 10 00 00 F0 02 00 00 ^ ð
3EB58D20 78 00 00 00 40 E9 19 86 27 00 00 00 02 00 00 00 x @é †'
3EB58D30 00 00 00 00 07 00 08 00 40 E9 19 86 D5 97 ED 93 @é †Õ-í"
3EB58D40 03 00 26 00 00 00 00 00 48 8D 55 86 48 8D 55 86 & H Ut H Ut
3EB58D50 50 8D 55 86 50 8D 55 86 40 E4 4B 3F 00 00 00 00 P Ut P Ut @äK?



If you don't have the required tools to do DKOM, you can still see the effects of destroying the dispatcher header. One of the memory images in your cases folder, ds_fuzz_hidden_proc.img, has had a process hidden using the DKOM technique described on the previous page. To see it, open up the Bless Hex Editor on your SIFT workstation. Use Bless to open the file /home/sansforensics/Desktop/cases/ds_fuzz_hidden_proc.img. (You can drag and drop the memory image into the Bless window to open it.)

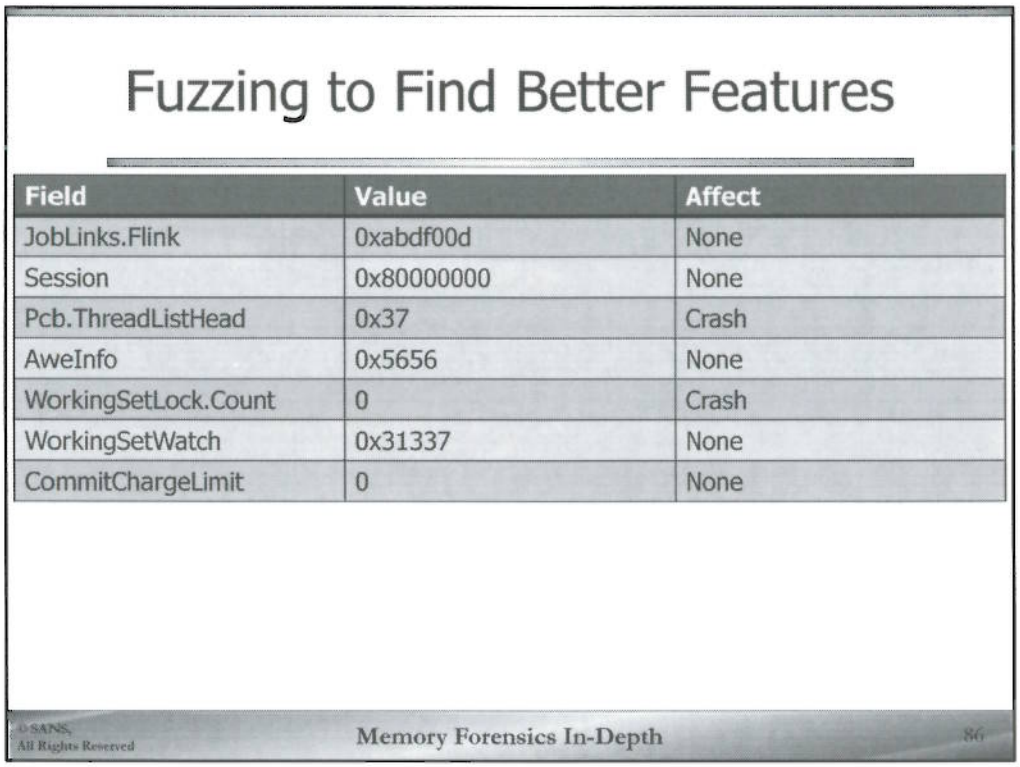
In the ds_fuzz_hidden_proc.img memory image, navigate to offset 0x1a4bc04. The command to go to an offset is under the search menu. The input window for the offset to go to appears at the bottom of the screen.

At this offset you should see the start of an EPROCESS block—the pool header bytes 50 72 6f e3. But where the dispatcher header should be, there are only zeros! Look further down and you should see the process name, network_listene.



Although some features of each kernel object can be changed without consequences, many cannot. There are some features which, if you change them, cause the operating system to crash. When doing forensics, we would like to use such features in our searching. If there is some aspect of an EPROCESS which cannot be changed lest the system go down, we should be searching for that feature and using it to validate potential EPROCESS blocks.

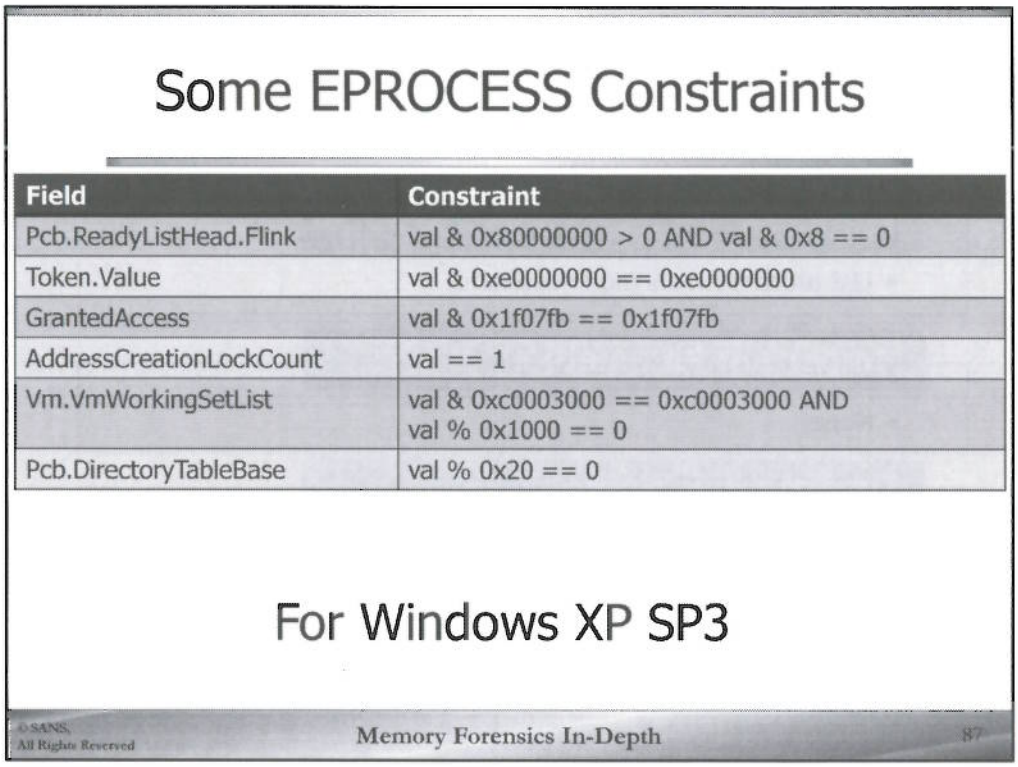
Finding those features can be done using a process called fuzzing. Although generally thought of as a method for finding weaknesses in a piece of software, it can also be used to identify features useful for forensic analysis. Specifically we identify the parts of structures which cannot be changed lest they crash the operating system. We then search for those features to guarantee we find the structures we want.



Here's a sample of what fuzzing would looks like. To fuzz the EPROCESS block, we set up an environment with a running operating system. We then change some value in the EPROCESS structure at random.

We record which value we change and what we changed it to, but don't care which one or what value we choose. For example, we could set the WorkingSetLock.Count value in the EPROCESS to 0x31337. Or the WorkingSetWatch to zero. We then see if the system crashes or not. As it turns out, setting the WorkingSetWatch value to 0x31337 has no effect on the operating system. It doesn't affect the system. But changing the WorkingSetLock value causes the system to crash. We record both results.

After repeating this process many, many times, we get a large data set indicating which values for which fields crash the system. Those are fields and values we are most interested in. If altering those values means the system will crash, an attacker cannot mess with them and hope to maintain the system. Thus forensic examiners should search for those fields and their legitimate values.



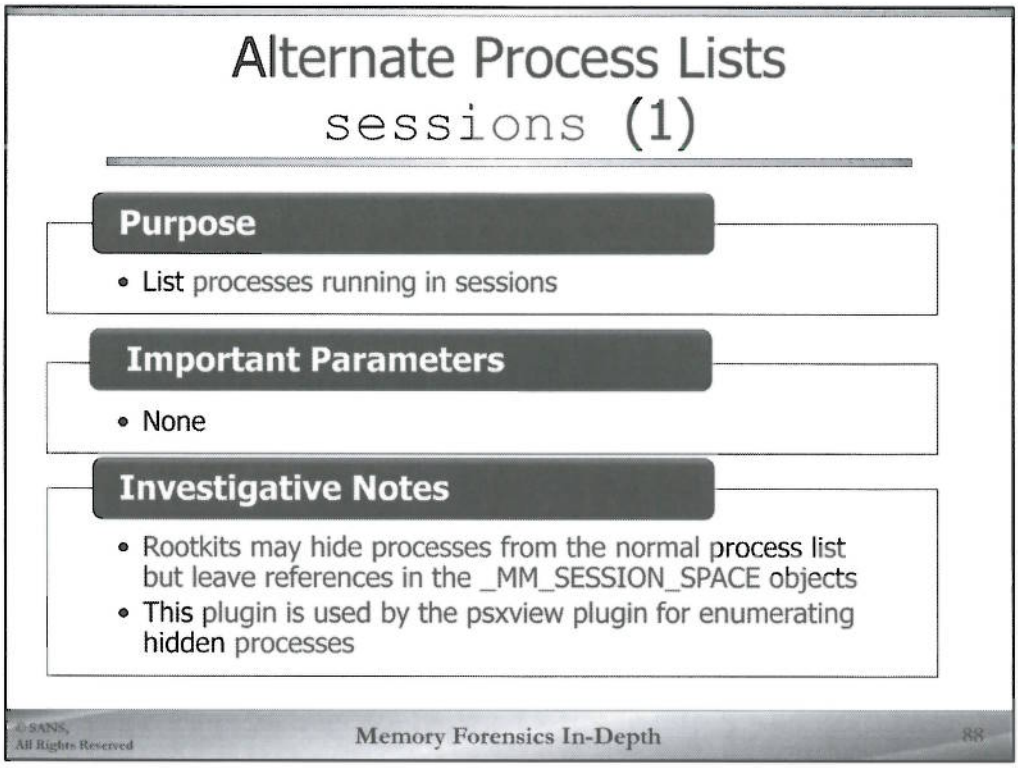
Using the raw data, there is an algorithm we can use to compute the limits on the values which cause the operating system to crash. The result is a series of constraints on values in the EPROCESS block which must be met for the system to run and for the block to be a valid EPROCESS. For example, the AddressCreationLockCount must be one. A subset of the conditions found for Windows XP Service Pack 3 is shown in the picture above.

This method was invented by Brendan Dolan-Gavitt, aka moyix. He published a great paper on this topic* and it was featured on his blog at http://moyix.blogspot.com/2010/07/plugin-post-robust-process-scanner.html. Moyix created a system to do this fuzzing automatically and used it to find these constraints and more for Windows XP Service Pack 3.

Thankfully, moyix wrote a Volatility plugin which used his work to search for processes. That plugin was called psscan3, and you can find a copy at http://www.cc.gatech.edu/~brendan/volatility/dl/psscan3.py. Unfortunately, the plugin was developed for an earlier version of Volatility and does not work with version 2.0.

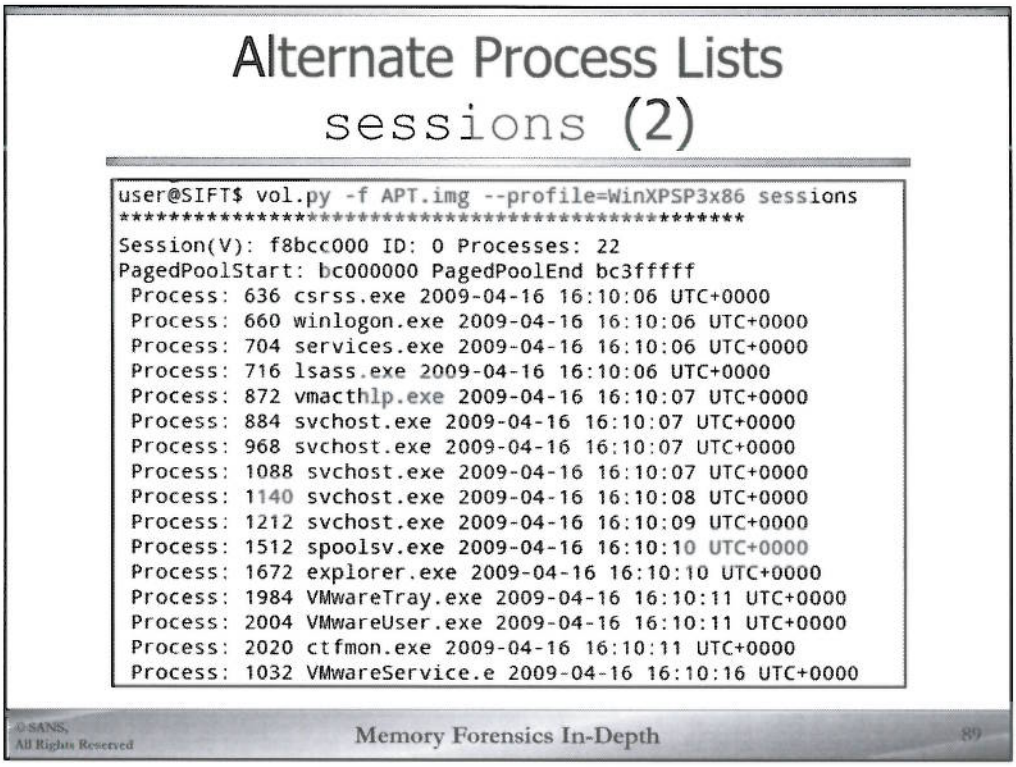
The other thing to note here is that the constraints which worked for Windows XP Service Pack 3 may not necessarily work for other service packs or versions of Windows. This research to find constraints will need to be repeated for all of the other versions. To date, nobody has done that work.

- Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin, *Robust Signatures for Kernel Data Structures*, ACM CCS 2009, http://www.cc.gatech.edu/~brendan/ccs09_siggen.pdf



An additional way to enumerate processes and potentially spot those that have been unlinked is through sessions. In Windows Vista and later, when a user logs on, a new session is created. (In Windows XP and 2003, there is just one session (Session 0) -that is shared between system services and user programs.[1]) All processes then launched from this session are tracked in a different linked list associated with the Session ID. In addition, there is a value maintained in the EPROCESS structure of the process that tags it with the associated session. The **sessions** plugin is used to locate _MM_SESSION_SPACE structures, which contains the pointer to the linked list of session processes. In addition to locating processes, the sessions plugin will also locate drivers mapped into the session and memory pools allocated to the session.

The sessions plugin is defined in the file volatility/plugins/gui/sessions.py



vol.py -f APT.img --profile=WinXPSP3x86 sessions

****

Session(V): f8bcc000 ID: 0 Processes: 22

PagedPoolStart: bc000000 PagedPoolEnd bc3fffff

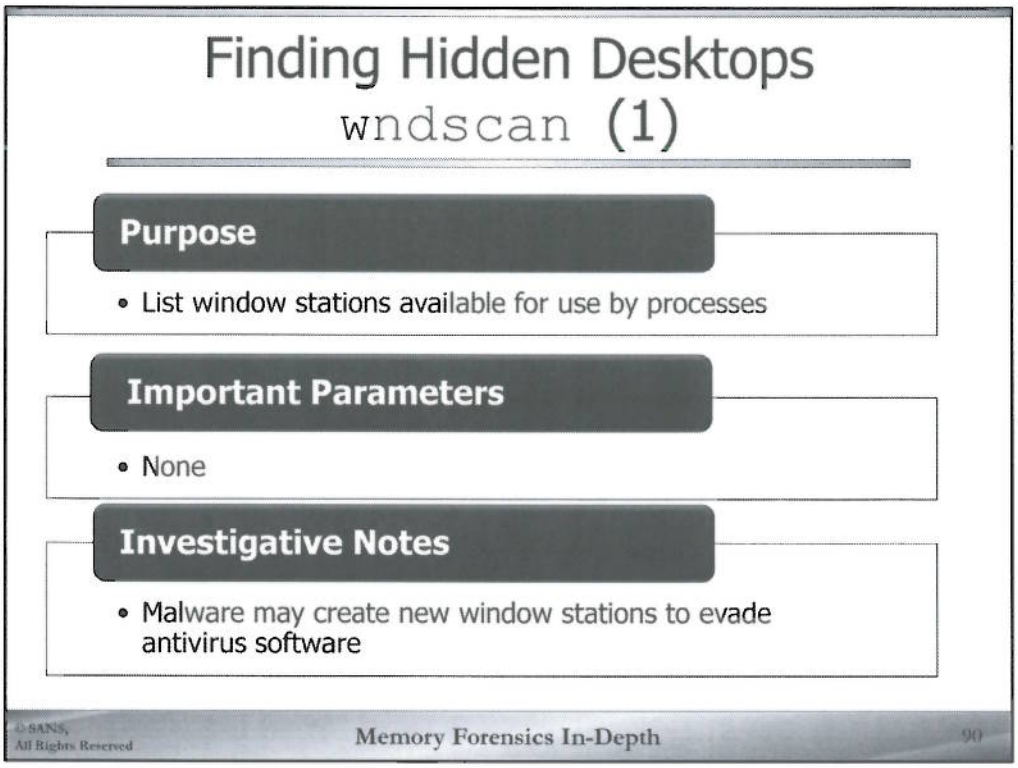
Process: 636 csrss.exe 2009-04-16 16:10:06 UTC+0000

Process: 660 winlogon.exe 2009-04-16 16:10:06 UTC+0000

Process: 704 services.exe 2009-04-16 16:10:06 UTC+0000

… output truncated …

[1] MoVP 1.1 Logon Sessions, Processes, and Images. http://volatility-labs.blogspot.com/2012/09/movp-11-logon-sessions-processes-and.html.



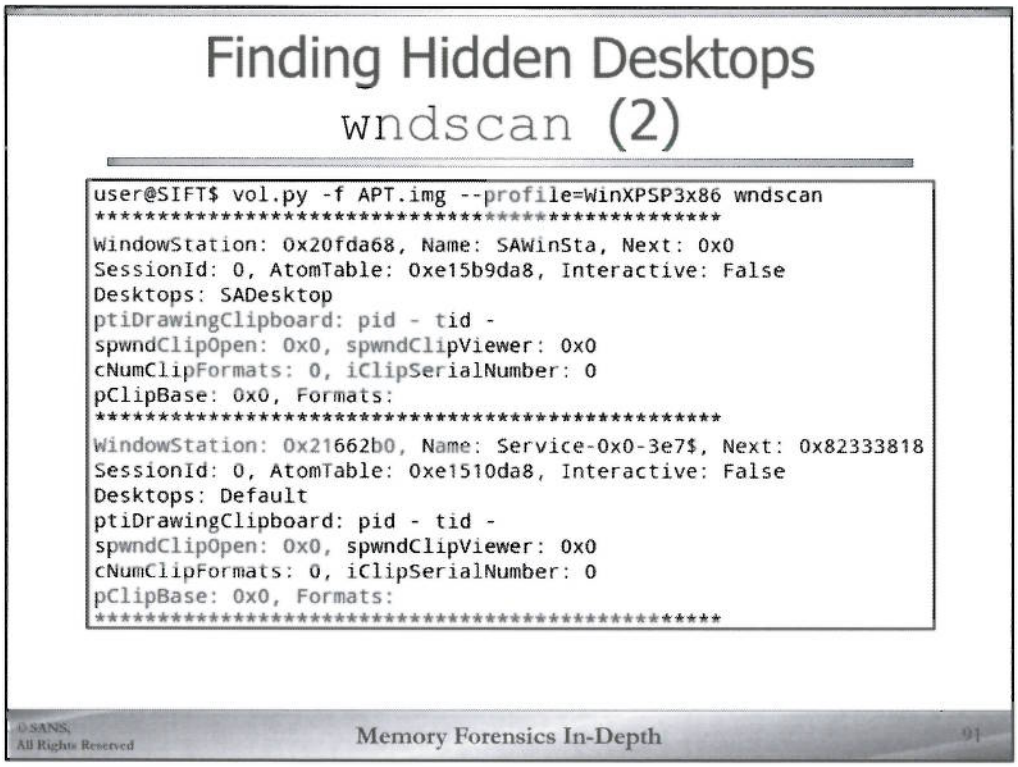
The wndscan plugin scans through pool memory to locate tagWINDOWSTATION objects. These objects are allocated with the specific pool tag 'Win\xe4' . When examining the output, you should know that the user's desktop is normally located on WinSta0. Some Windows API functions require a windows station (a desktop) to connect to. Malware may create new desktops to connect to in order to evade antivirus.

Matt Burrough from Microsoft has an excellent description of how the clipboard works and how the tagWINDOWSTATION objects are related to clipboard operation located here:

http://blogs.msdn.com/b/ntdebugging/archive/2012/03/29/how-the-clipboard-works-part-2.aspx

Michael Hale Ligh wrote an excellent blog post on locating malware in the GUI.

http://volatility-labs.blogspot.com/2012/09/movp-12-window-stations-and-clipboard.html



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

****

WindowStation: 0x20fda68, Name: SAWinSta, Next: 0x0

SessionId: 0, AtomTable: 0xe15b9da8, Interactive: False

Desktops: SADesktop

ptiDrawingClipboard: pid - tid -

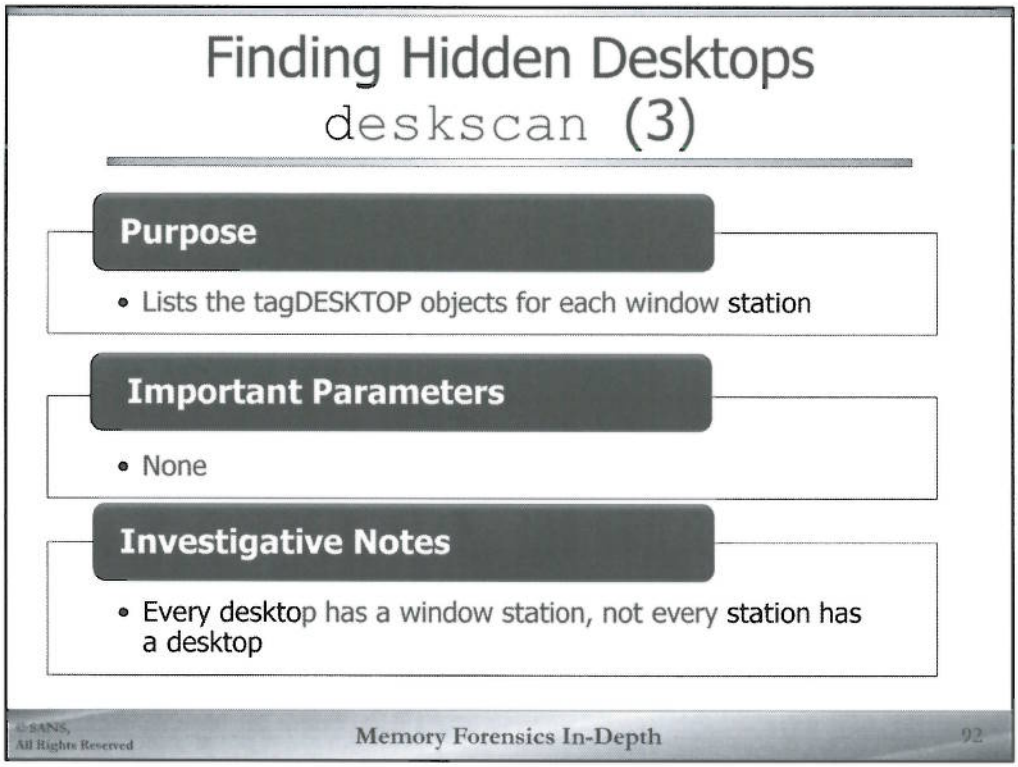
spwndClipOpen: 0x0, spwndClipViewer: 0x0

cNumClipFormats: 0, iClipSerialNumber: 0

pClipBase: 0x0, Formats:

****

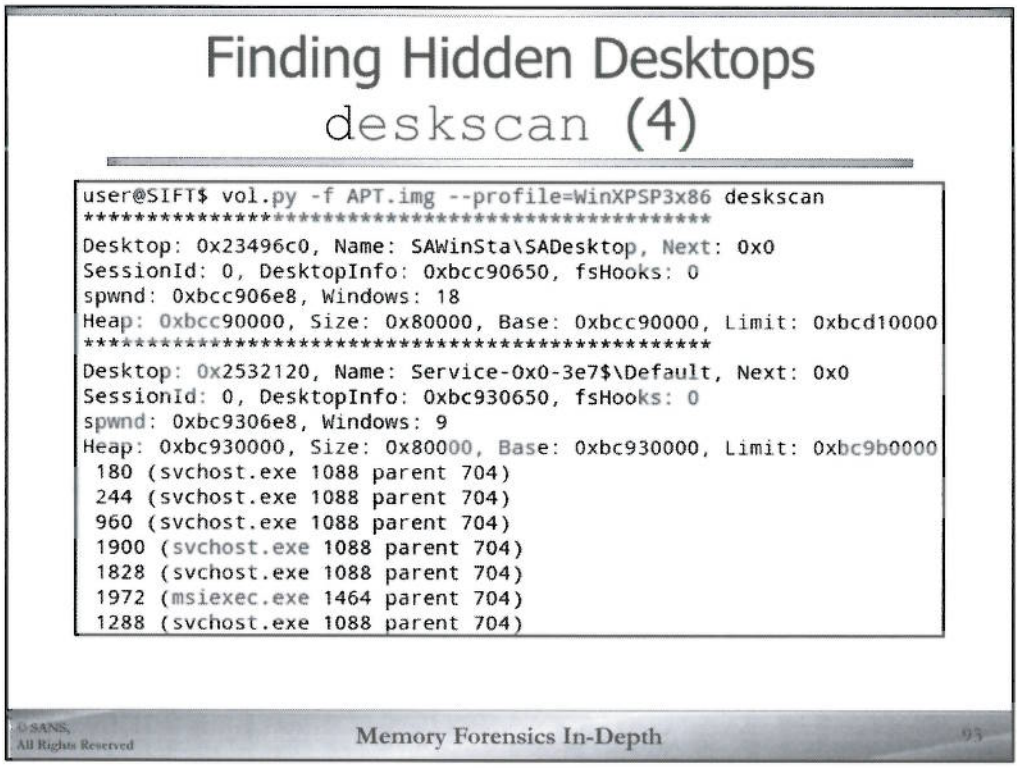
… output truncated ….



The deskscan enumerates desktops for each window station found. The plugin enumerates the tagDESKTOP objects, referenced from the output discovered via wndscan.

Note that the WinSta0\Default desktop is the user's standard desktop. We should expect user processes to be located on this desktop.

```
vol.py -f APT.img --profile=WinXPSP3x86 deskscan
Volatility Foundation Volatility Framework 2.3.1
**************************************************
Desktop: 0x23496c0, Name: SAWinSta\SADesktop, Next: 0x0
SessionId: 0, DesktopInfo: 0xbcc90650, fsHooks: 0
spwnd: 0xbcc906e8, Windows: 18
Heap: 0xbcc90000, Size: 0x80000, Base: 0xbcc90000, Limit: 0xbcd10000
**************************************************
Desktop: 0x20accf8, Name: WinSta0\Default, Next: 0x8219b038
SessionId: 0, DesktopInfo: 0xbc630650, fsHooks: 2168
spwnd: 0xbc6306e8, Windows: 141
Heap: 0xbc630000, Size: 0x300000, Base: 0xbc630000, Limit: 0xbc930000
 500 (MIRAgent.exe 456 parent 840)
 1444 (iexplore.exe 796 parent 884)
 1716 (cmd.exe 840 parent 1672)
 308 (explorer.exe 1672 parent 1624)
```



vol.py -f APT.img --profile=WinXPSP3x86 deskscan

****

Desktop: 0x23496c0, Name: SAWinSta\SADesktop, Next: 0x0

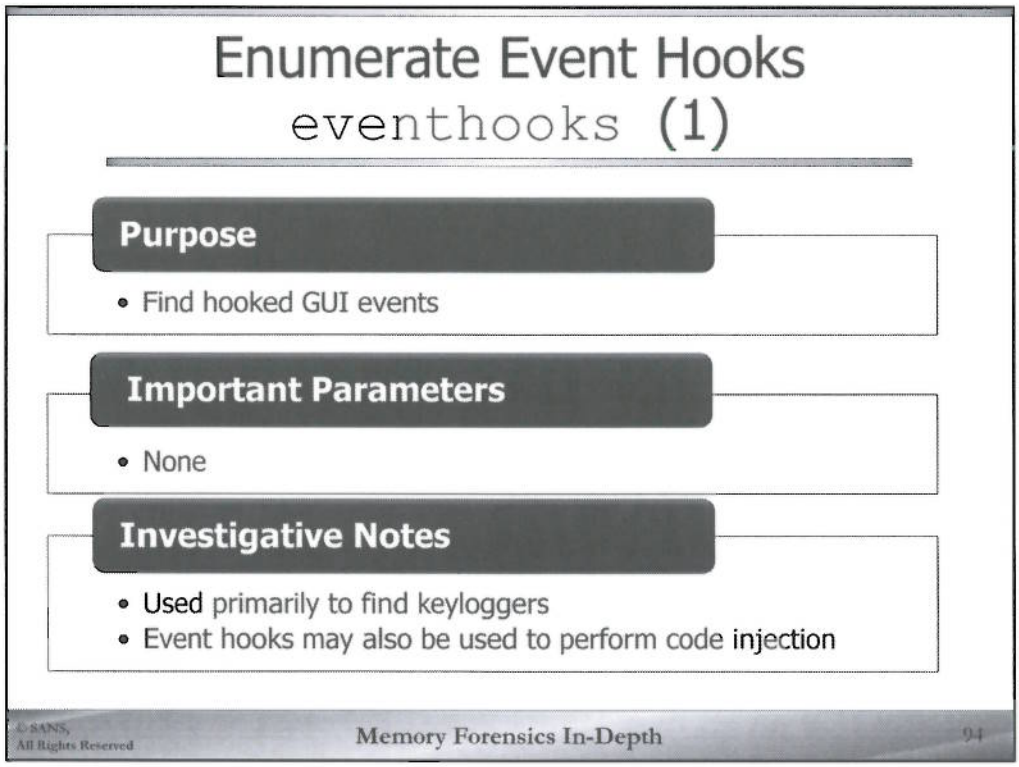
SessionId: 0, DesktopInfo: 0xbcc90650, fsHooks: 0

spwnd: 0xbcc906e8, Windows: 18

Heap: 0xbcc90000, Size: 0x80000, Base: 0xbcc90000, Limit: 0xbcd10000

****

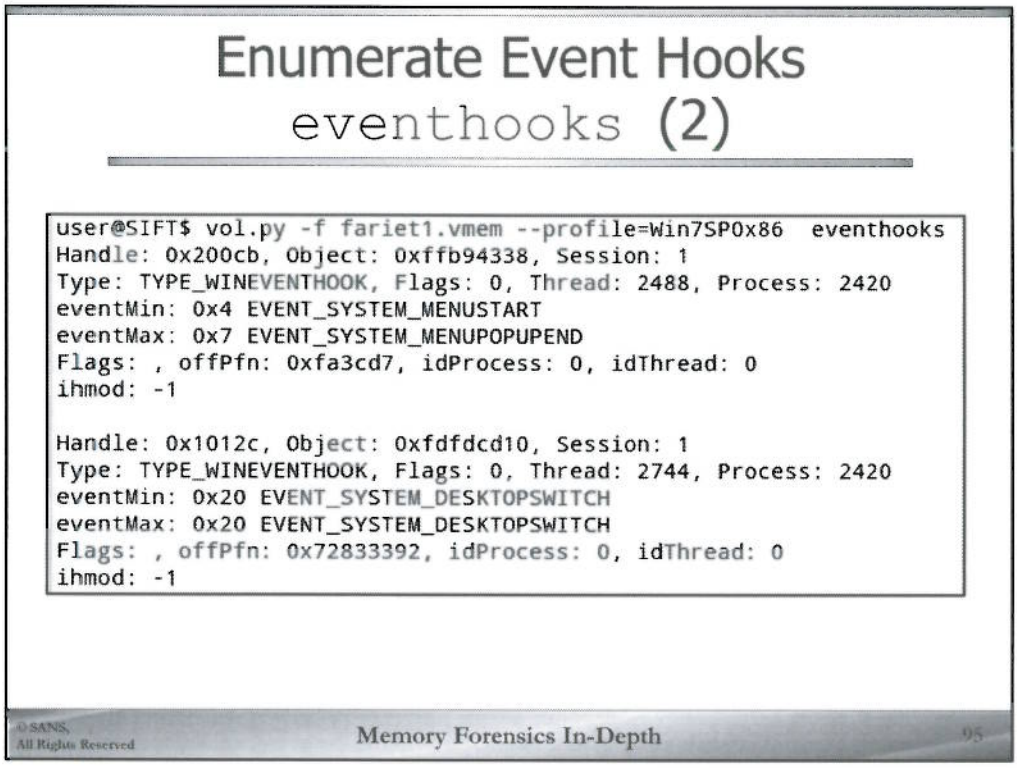
… output truncated …



The eventhooks plugin is used to identify hooks that have been set in the Windows GUI. These hooks are usually set using the Windows APIs SetWindowsHook and SetWindowsHookEx. Hook types of the most significance for malware are WH_KEYBOARD, WH_MOUSE, and WH_CBT. The latter monitors all GUI interaction (including keyboard and mouse) so it is particularly dangerous. Note that hooks can be set on a per-process or per-desktop basis. Event hooks are only valid for GUI processes, so this technique couldn't be used for instance to inject code into lsass.exe.

The eventhooks plugin lists the types of hooks that have been set, the process that the hook is set in, and the module containing the hook callback function. The hook callback function is invoked whenever an event of the proper type is received in the GUI for the hooked process. It is normal for this plugin to return data. Carefully examine the module name for the hooked function.

The code for this plugin is located in the file volatility/plugins/gui/eventhooks.py



user@SIFT$ vol.py -f fariet1.vmem --profile=Win7SP0x86 eventhooks

Handle: 0x200cb, Object: 0xffb94338, Session: 1

Type: TYPE_WINEVENTHOOK, Flags: 0, Thread: 2488, Process: 2420

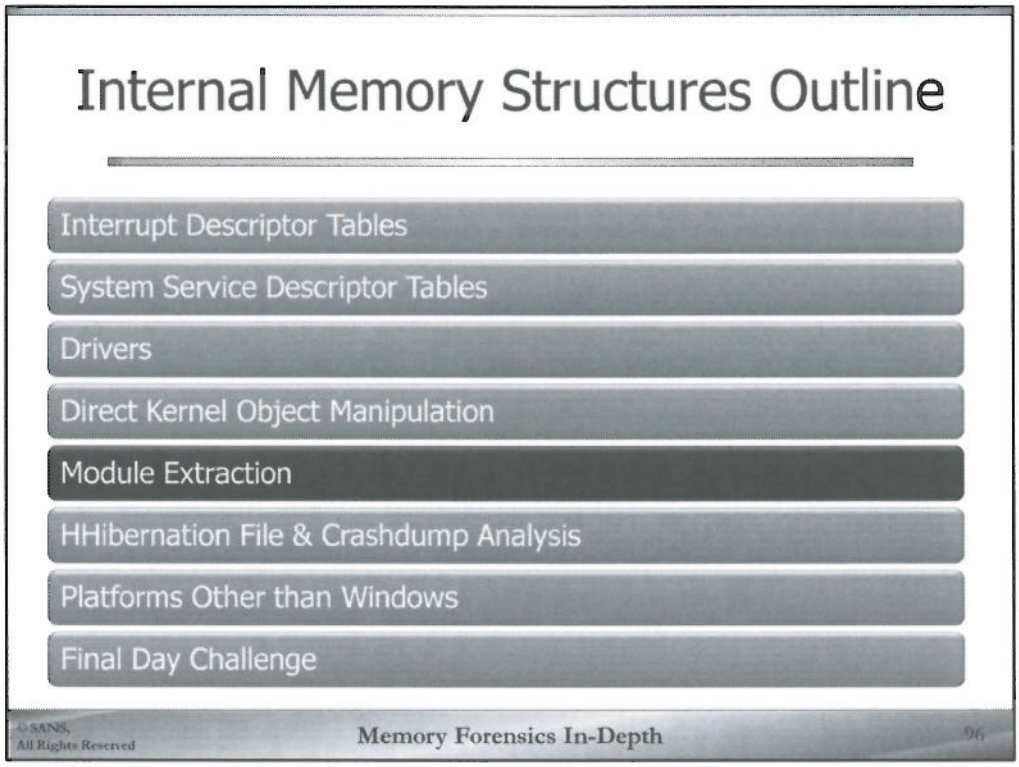
eventMin: 0x4 EVENT_SYSTEM_MENUSTART

eventMax: 0x7 EVENT_SYSTEM_MENUPOPUPEND

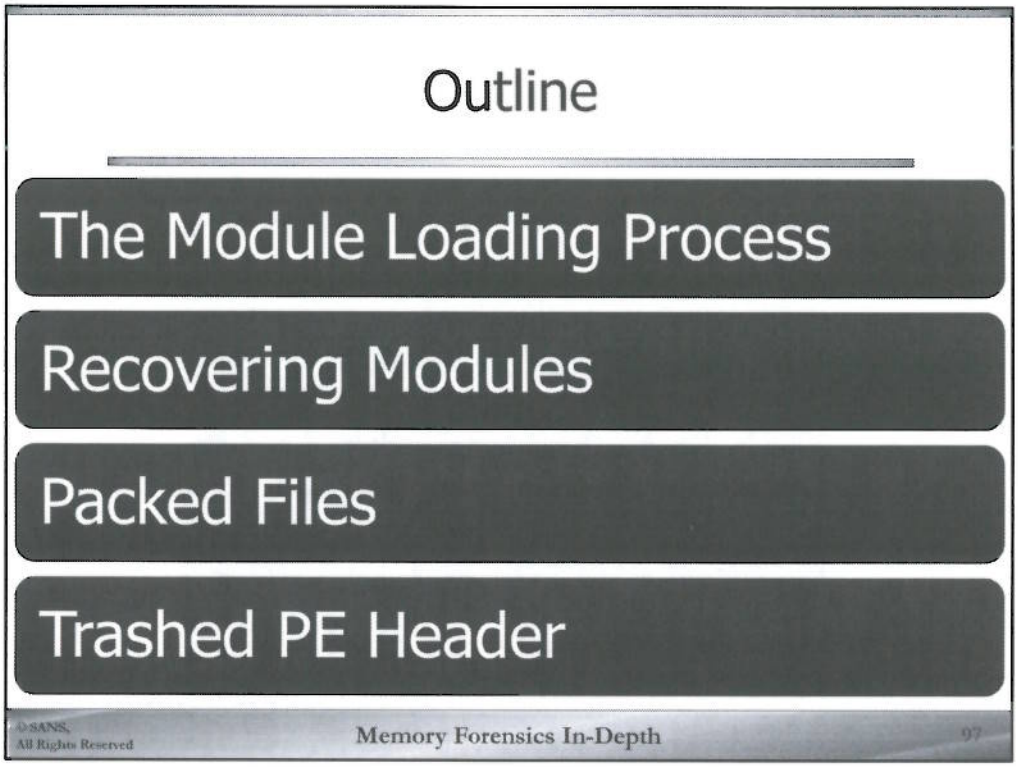
Flags: , offPfn: 0xfa3cd7, idProcess: 0, idThread: 0

ihmod: -1

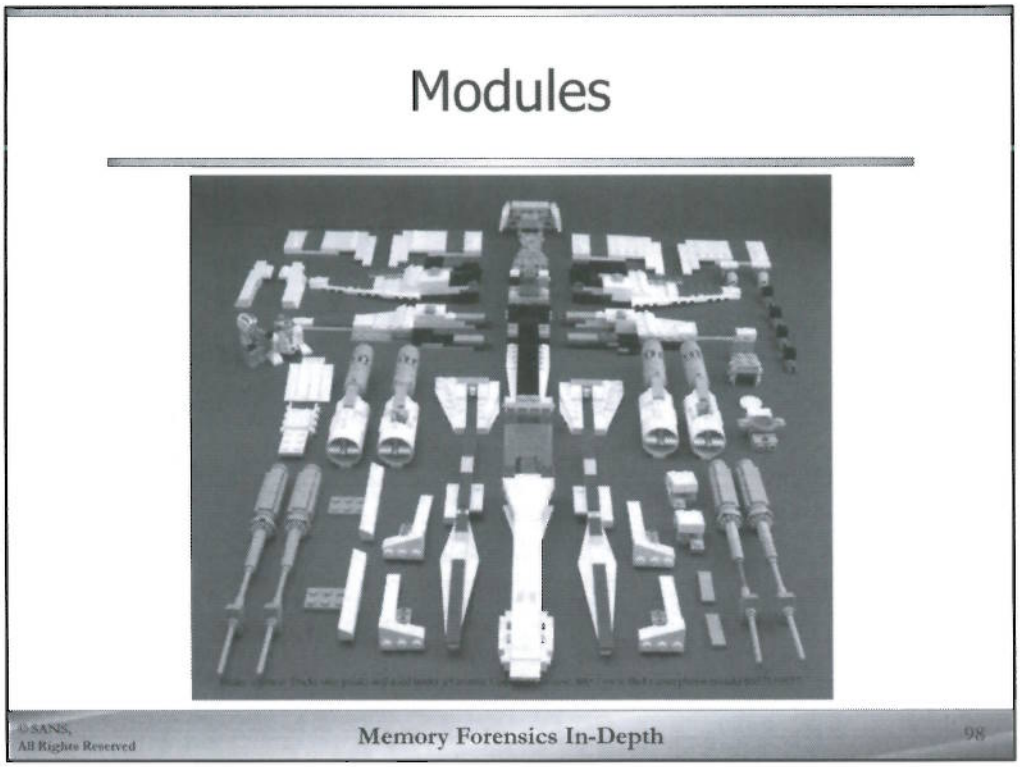
…. Output truncated …



This page intentionally left blank.



This page intentionally left blank.

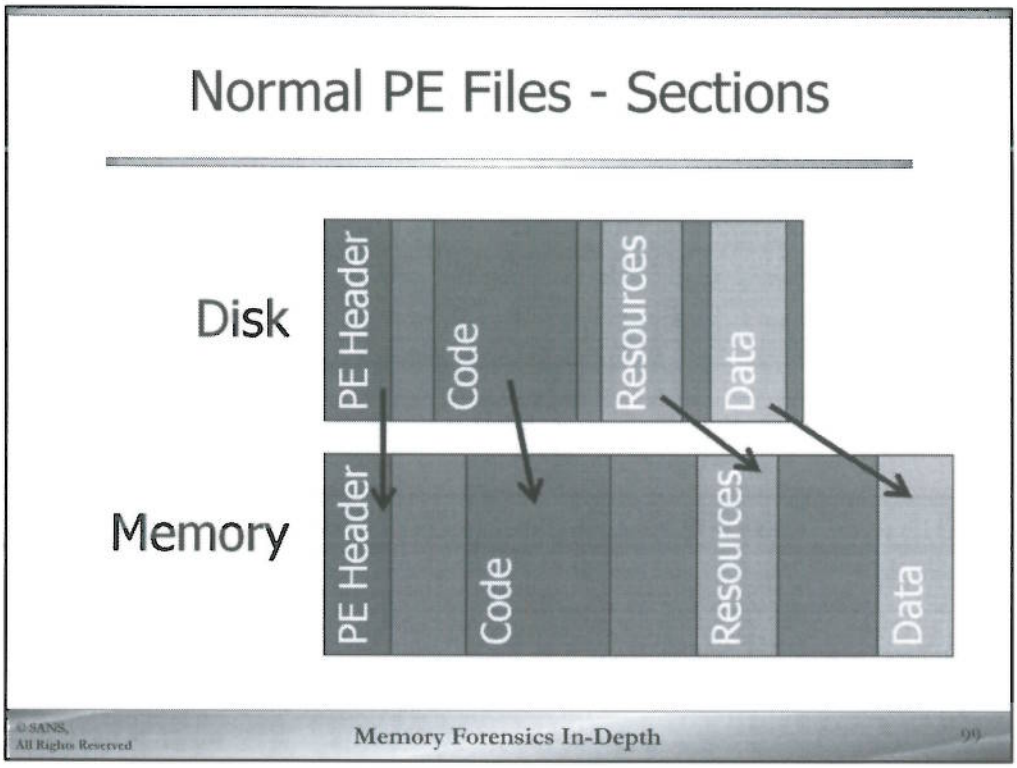


All PE executables loaded into memory are considered modules: Processes, DLLs, and drivers. The Portable Executable format (PE) was introduced in Windows 3.1 and hasn't changed much since then. Although there is a formal specification for how PE's work [1], a much more readable overview was given by Matt Pietrek in a 2002 article in MSDN magazine, "An In-Depth Look into the Win32 Portable Executable File Format", **http://msdn.microsoft.com/en-us/magazine/cc301805.aspx.**

Until now, we've talked about looking at modules in memory images to learn more about them. In this block of instruction we are going to recover modules from memory images. We'll start with normal, legitimate modules and how they're loaded into memory and used. We're going to do a hands-on exercise to see how we can recover modules and compare them to their known good programs. After that we'll discuss what packers and malicious programs do. Finally, we'll describe some techniques for detecting and dealing with those files.

[1] Microsoft PE and COFF Specification, http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

(Yes, PE executable is redundant, like PIN number or ATM machine. Go with it.)

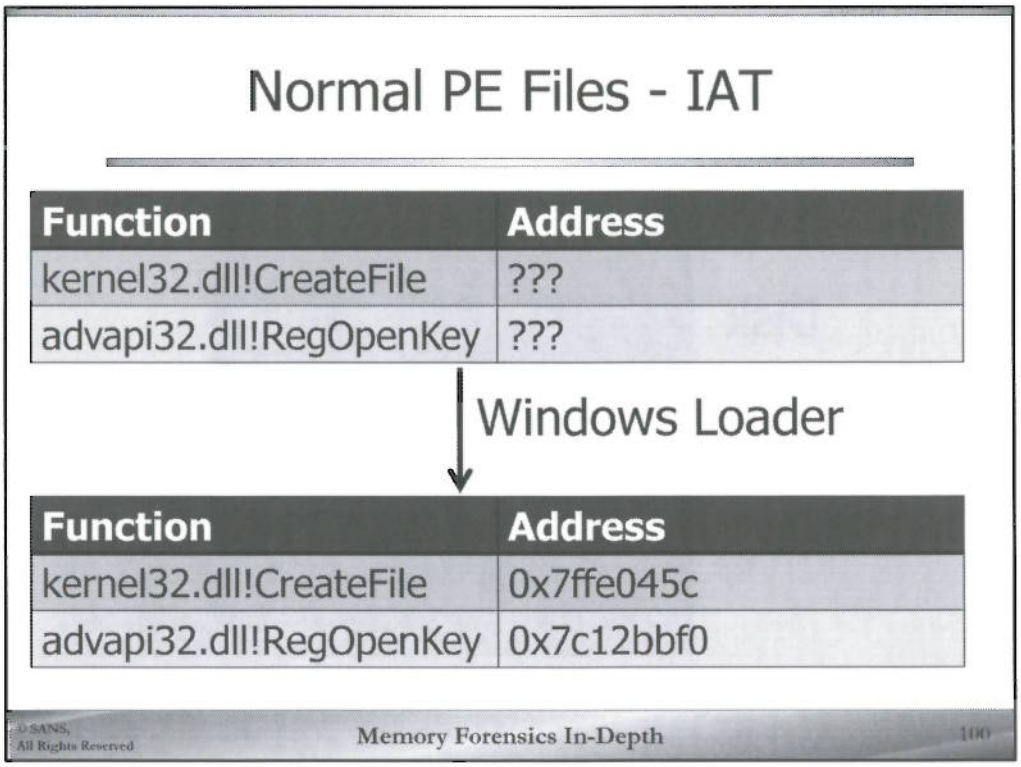


A PE file on the disk contains a header and then one or more sections. These sections can contain code, resources (such as icons, dialog boxes), and data.

When a PE is loaded into memory, the Windows loader parses the header to see what needs to be brought into memory and what needs to happen to that data before the program can be run. There are three main types of changes made from the data on the disk to the data we find in memory.

First, each section of the PE file is loaded into memory. The PE header specifies, for each section, its location on the disk, its size on the disk, its desired location in memory, and its size in memory. The alignment of these sections is different on the disk than it is in memory. On the disk, the sections must be aligned to sectors, which are 512 (0x200) bytes long. In memory, the sections must be page aligned, or every 0x1000 bytes. The different sections must be on different pages because they could have different permissions. While the executable code part of the PE needs executable permissions, the data does not (and should not) have such permissions.

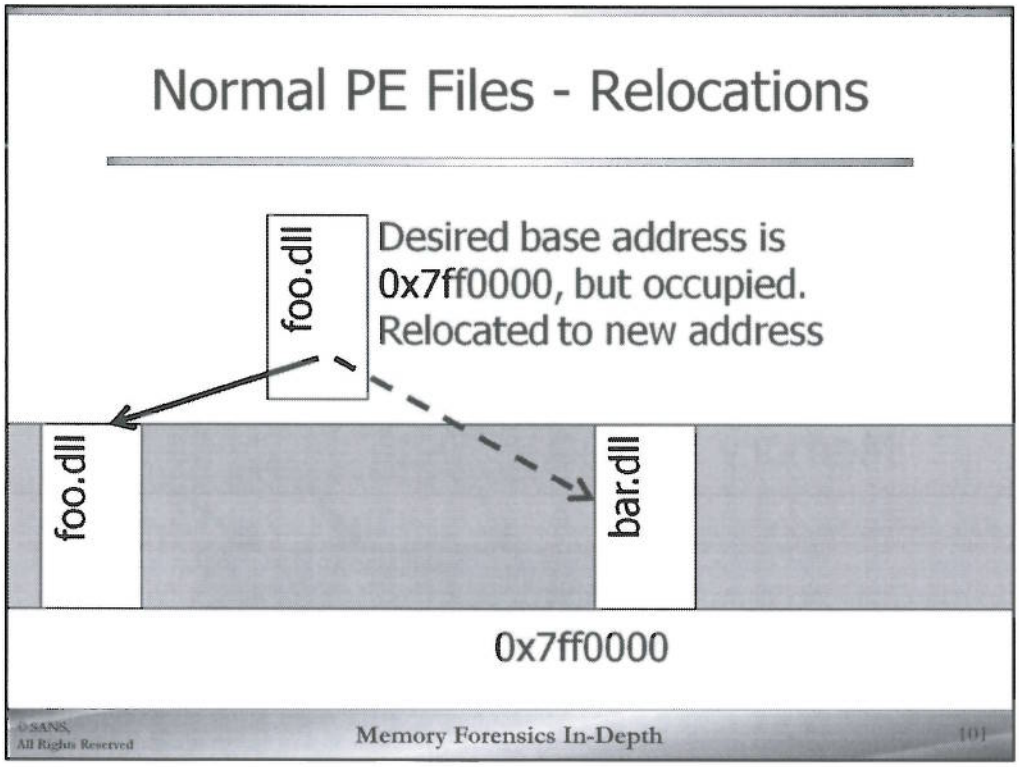
The result is that there is some "slack space" inserted between sections in the virtual address space as compared to the file on the disk.



The second major change between PE files on the disk and PE files in memory is that the functions imported from other modules need to be found and mapped for this PE.

When a program is compiled, there are always functions which need to be imported from other modules, usually DLLs. For example, the program Notepad needs to open files from the disk, and thus imports the function CreateFile from the DLL kernel32.dll. The compiler does not know what the virtual address of the CreateFile function will be when the program is run. The kernel32 DLL could be loaded at any virtual address in userspace. There's no way for the compiler to know where ahead of time.

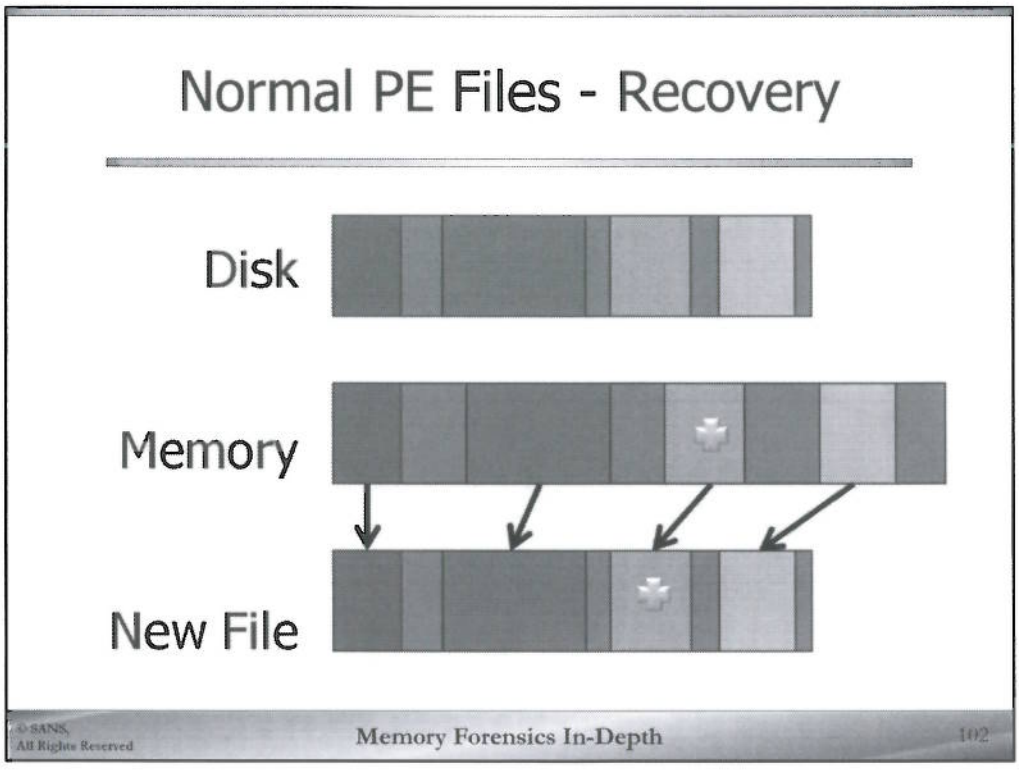
To make sure Notepad uses the correct address for imported functions, calls to such functions are routed through a lookup table. Notepad looks up the correct virtual address in a table compiled into Notepad. When the program is run, Windows must update these tables with the current virtual addresses of all of the imported functions. The compiler includes this table, called the Import Address Table (IAT), which lists all of the imported functions in the program. When the PE is loaded, the Windows loader must parse the IAT and update the virtual addresses for the all of the functions used by the program. Each entry in the IAT lists the function being imported. Using the table saves time during the update process. Instead of having to update every instance in the program where such functions are called, the loader only has to update one entry in the table.



The third major change to a PE file during loading happens if the PE needs to be relocated. When PE executables are compiled, a virtual address is chosen as the "preferred" virtual address for the base of the executable. This will be the virtual address where the start of the PE is stored in RAM. References in the PE file are then based on that virtual address. Let's say a PE chooses 0x7ccc0000 as its base address. If a function in that PE begins 0x5000 bytes into the file, the actual call in the PE file will be "call 0x7ccc5000". But when this PE is loaded, if there is already another PE file at the preferred base address, this PE will need to be relocated. It can't be loaded into its preferred address and will need to be relocated to another address.

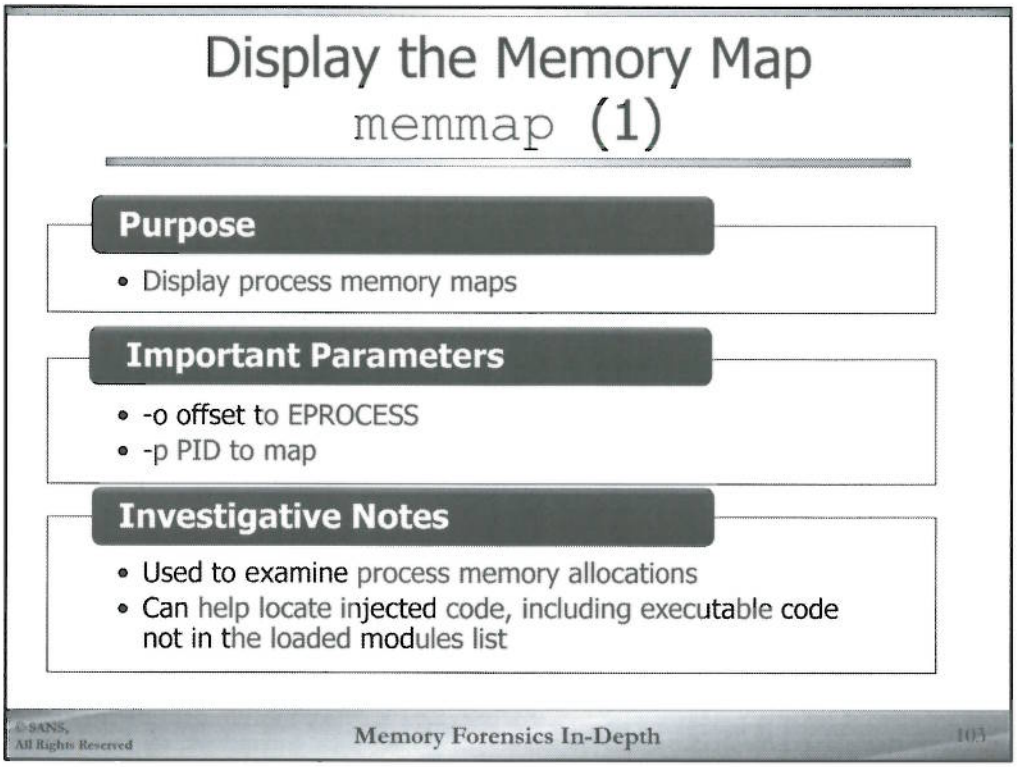
When a PE needs to be relocated, all of these hardcoded virtual addresses will need to be changed. Windows does this by embedding in the PE a list of the hardcoded virtual addresses that need to be changed when the PE is rebased. To rebase the PE, Windows calculates the difference between the preferred virtual address and the actual virtual address where the PE will be loaded. This difference, or delta, is then applied to each reference in the program. Continuing our example, if our PE needed to be relocated to a base address of 0x7bbb0000, Windows would calculate the delta of 0x7ccc0000 – 0x7bbb0000, which is 0x111000. This value would then be subtracted from each hard coded address in the PE. The call above, originally to 0x7ccc5000, would be changed to 0x7ccc5000 – 0x1110000, or 0x7bbb5000.

This relocation process takes time and slows down how long it takes to load and run executables. It also negates any benefits of having a shared DLL. That is, if a PE can be loaded into the same virtual address space for multiple processes, those processes can share the frames where the DLL is loaded. If notepad.exe and calc.exe both need our DLL and can load it at 0x7ccc0000, then only one set of frames is needed. But if the DLL has to be rebased for one of those it programs, then two copies of the DLL are needed in memory. The second copy of the DLL has the rebased hard coded addresses. Microsoft takes care to ensure that all of its DLLs have base addresses and sizes which do not overlap in virtual memory, but problems arise when third party vendors are creating DLLs.



From a forensics point of view, the PE header is a fantastic resource. The header is loaded into memory with the other parts of the file. As mentioned above, the header details, for each section, where it goes in memory, its size in memory, where it goes on the disk, and its size on the disk. We can therefore use the header to identify each section in virtual memory and copy it back out to a new file on the disk. In effect, we can recreate the file almost as it existed on the disk.

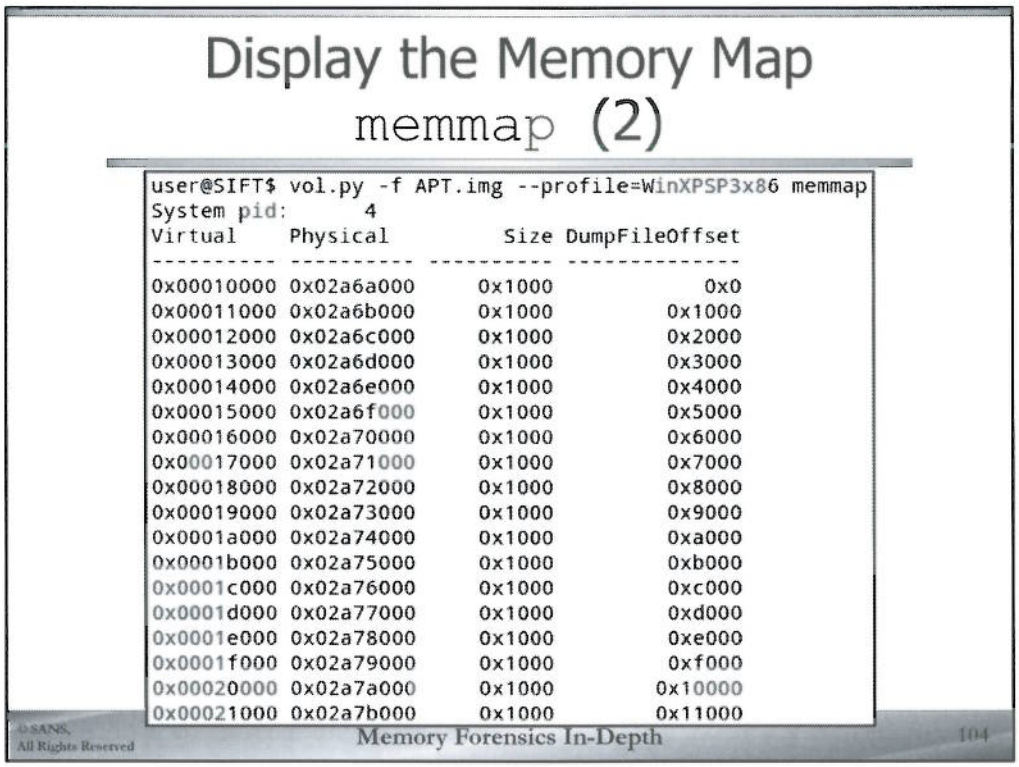
We say "almost" as it existed on the disk because the recovered file is not the same as the original. We cannot undo the changes made to the PE by the Windows loader. We can't recover the original addresses replaced in the Import Address Table. We also don't undo the relocations, if any, done to the program. We also may not be able to recover all of the pages of the executable from a memory image. The Windows loader is lazy, although computer scientists would call it efficient. It doesn't bring all of the pages of a PE file into memory unless they are going to be needed. In some cases, this means entire sections are not loaded into memory. Although that helps programs appear to start faster, it means we get less of the original executable during our recovery.



The memmap plugin is used to display a memory map of a given process (or all processes). This is similar to a listing of loaded DLLs obtained using the dlllist plugin. However, dlllist only walks the list of loaded modules for a process. Therefore, code injected using reflective DLL injection (the technique used by Meterpreter) would not be listed in the output of dlllist. However, the memory used by such code is still allocated to the process and as such would be present in the output of the memmap plugin.

One of the great things about the memmap plugin is that it gives both the virtual address of a page in a process and the physical address of the page in the dump file. Although we've already covered how to examine memory from a process using volshell, this can be useful for examining the physical memory using a hex editor.

The code for this plugin is located in the file volatility/plugins/taskmod.py. The plugin takes all the same options as the dlllist plugin because it inherits code from it (also located in taskmod.py).



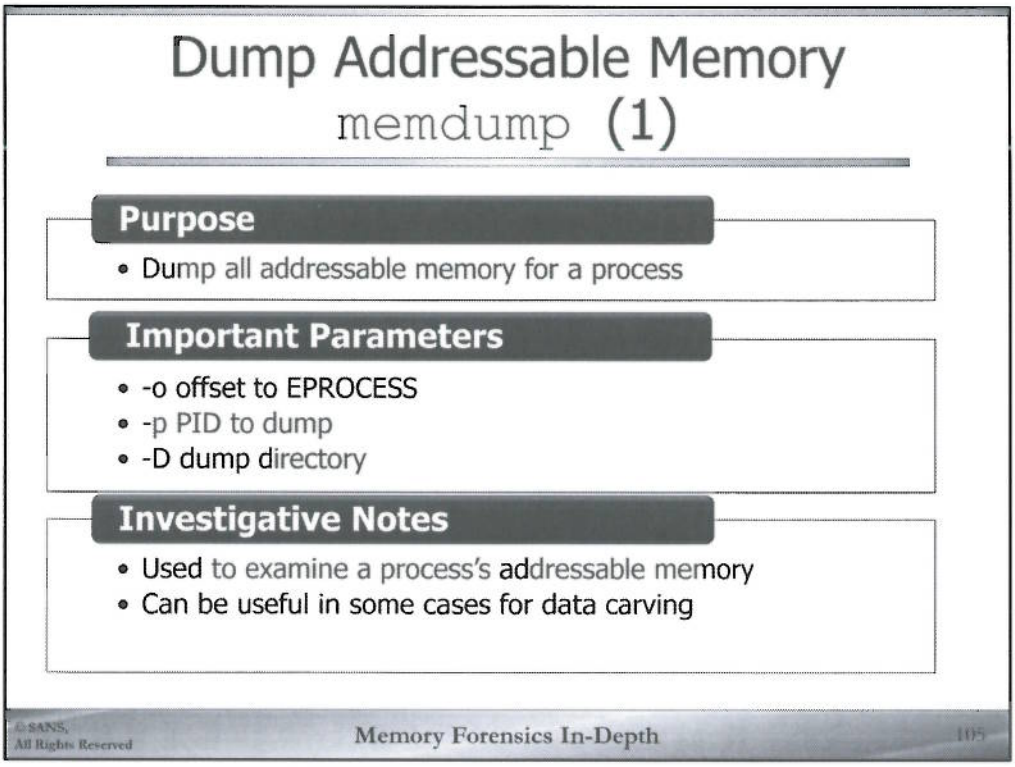
user@SIFT$ vol.py -f APT.img --profile=WinXPSP3x86 memmap

System pid: 4

Virtual	Physical	Size	DumpFileOffset
0x00010000	0x02a6a000	0x1000	0x0
0x00011000	0x02a6b000	0x1000	0x1000
0x00012000	0x02a6c000	0x1000	0x2000
0x00013000	0x02a6d000	0x1000	0x3000
0x00014000	0x02a6e000	0x1000	0x4000
0x00015000	0x02a6f000	0x1000	0x5000

…. Snip …

The virtual and physical addresses are shown in the first two fields. These fields are fairly self explanatory. The DumpFileOffset field specifies the offset to this memory page in the dump file created by the memdump plugin. This allows the investigator to map specific offsets in the dump file to known addresses in virtual memory.

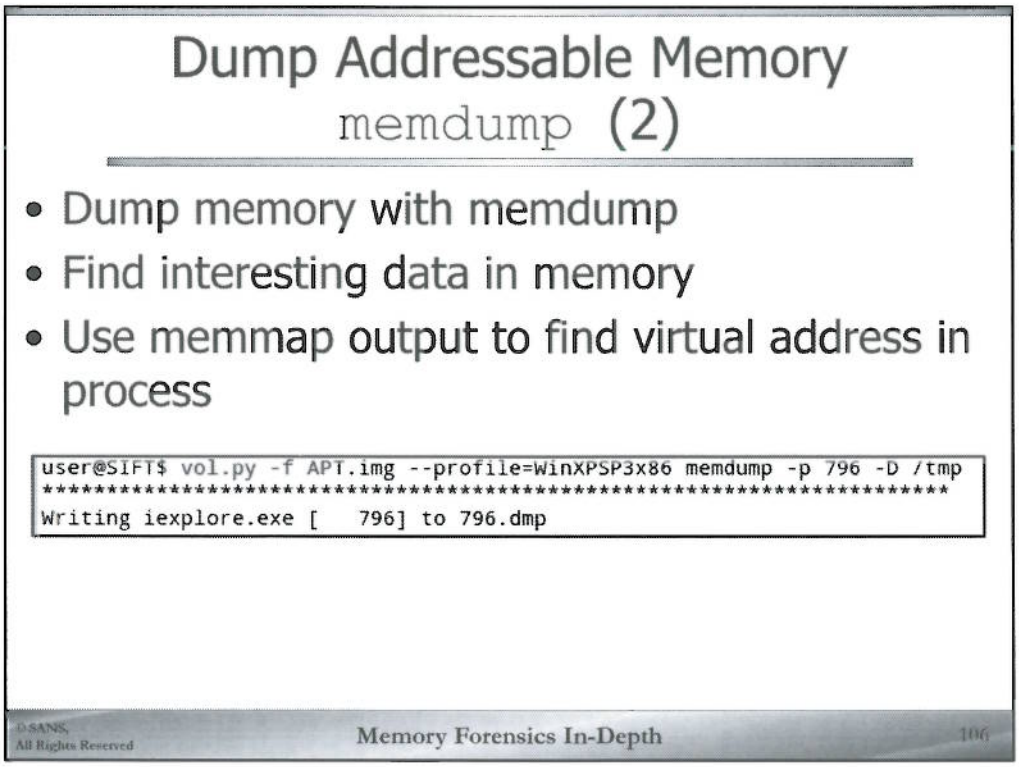


The memdump plugin is used to dump all addressable memory from a process to an output file. This memory includes loaded modules, heap, and stack memory. It also includes memory mapped files. This memory can be a treasure trove of data. Note that the output from the memmap plugin can be used to find the memory in the map file.

Consider the case of a system memory dump containing a private browsing session. Simply carving for HTML fragments (as we did with page_brute) may turn up some data. However, for data that breaks a page boundary (usually 4k) may be discontiguous in physical memory. However, the data is much more likely to be contiguous in virtual memory spaces.

Another example of when this is useful is in encryption programs. Some poorly written encryption programs may not overwrite the last plaintext buffer allocated with zeroes (or other garbage data). Memory dumps of a process may also be used to discover commands sent to malware and data exfiltrated. The sky is the limit, use your imagination and ingenuity. Carving from virtual memory space is almost always more effective than carving from a physical memory dump. Of course, carving from a physical dump will also include unallocated memory (which is not present in the output of the memdump plugin).

The code for this plugin is located in the file volatility/plugins/taskmod.py. The plugin takes all the same options as the memmap plugin because it inherits code from it (also located in taskmod.py).

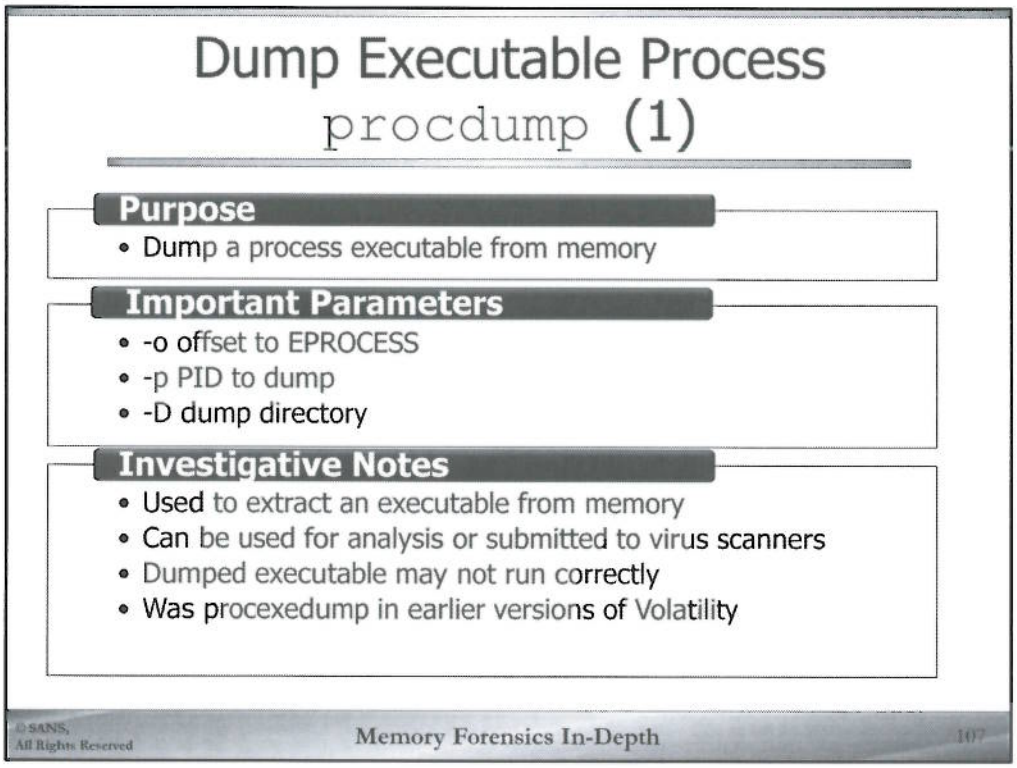


The following command creates a dump file containing the memory of the iexplore.exe process. When searching for interesting strings in the dump file, use the '-tx' option to print the offset of the string in hex. This will allow translation back to the virtual address in memory using the output from the memmap plugin.

vol.py --profile=WinXPSP3x86 -f APT.img memdump -p 796 -D /tmp

****

Writing iexplore.exe [796] to 796.dmp

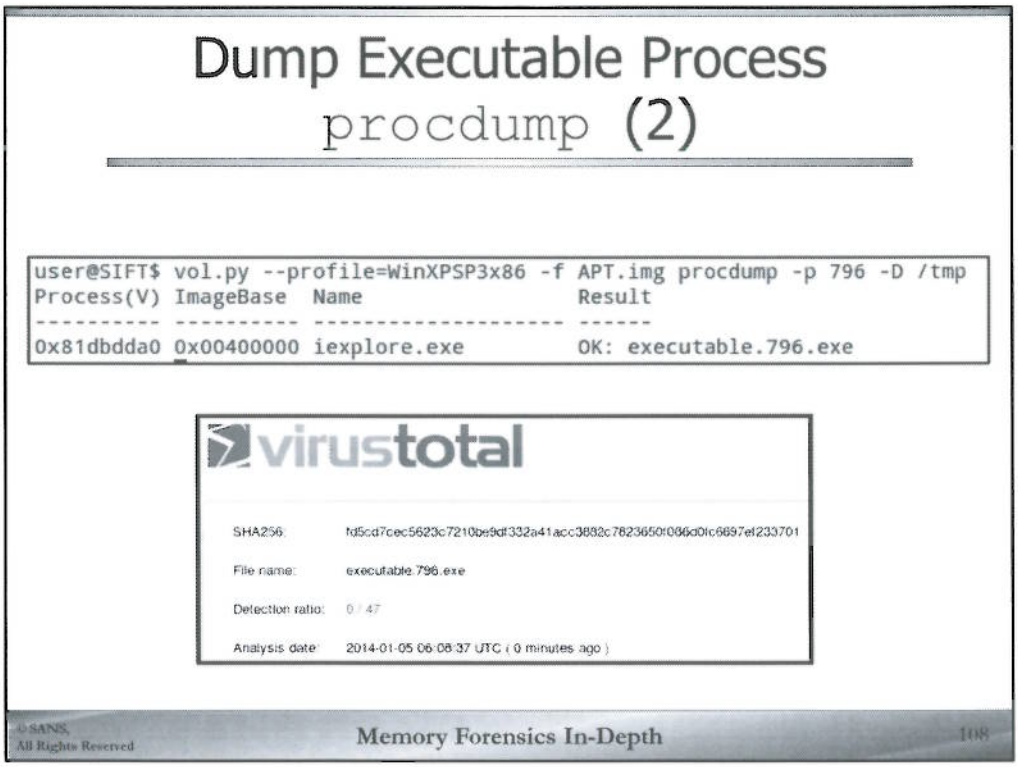


The procdump plugin is used to extract an executable from a physical memory dump. This plugin does not extract all memory from the process (as is the case with memdump). Rather, it merely extracts the memory associated with the executable running the process. In an earlier lab, you learned how to dump the memory associated with a driver. Although the same could be done for an executable, our technique would not take into account the relocations performed by the Windows loader (recall that executable file sections are loaded starting at page boundaries).

This plugin uses better heuristics than our volshell dumping method. It examines the PE header and discovers the proper alignment for the binary on disk. This results in a binary being output that is more like what was originally found on disk than a sample dumped using our volshell method. This extra space present only in memory is often referred to as "slack space."

This extracted executable can be submitted to online virus scanning engines for analysis. However, hash based analysis (e.g. NSRL) is unreliable. Further, the extracted file may not execute correctly. Both of these are due to changes made by the Windows loader that cannot be reliably undone by the plugin.

The code for this plugin is located in the file volatility/plugins/procdump.py.



Files extracted from memory using procdump can be uploaded to scanning sites, such as VirusTotal for analysis. It is generally safer to upload the file dumped from memory than the original file. This allows antivirus engines to scan the file for known malicious indicators without tipping the adversary based on the hash. However, you do still lose control of the file, so ensure that you have permission to upload the file.

This command dumps the iexplore.exe process to disk in the /tmp directory.

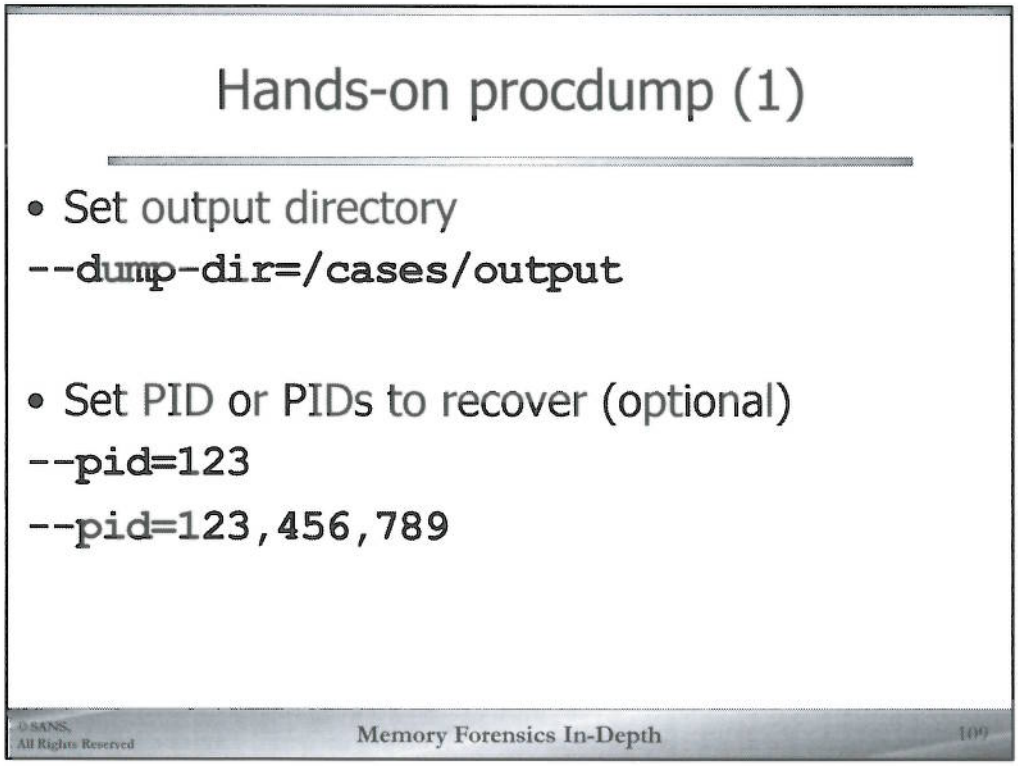
vol.py --profile=WinXPSP3x86 -f APT.img procdump -p 796 -D /tmp

Volatility Foundation Volatility Framework 2.3.1

Process(V) ImageBase Name Result

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

0x81dbdda0 0x00400000 iexplore.exe OK: executable.796.exe



Remember that when run in its default option, the procdump plugin creates a new file on the disk which, due to its parsing of the PE header, should be as close as possible to the original PE file before it was loaded into memory. For each section, procdump attempts to read the pages from memory and write them out to the specified location in the new file. If you get lucky, sometimes the resulting executable will *run* on a new system. But don't count on it.

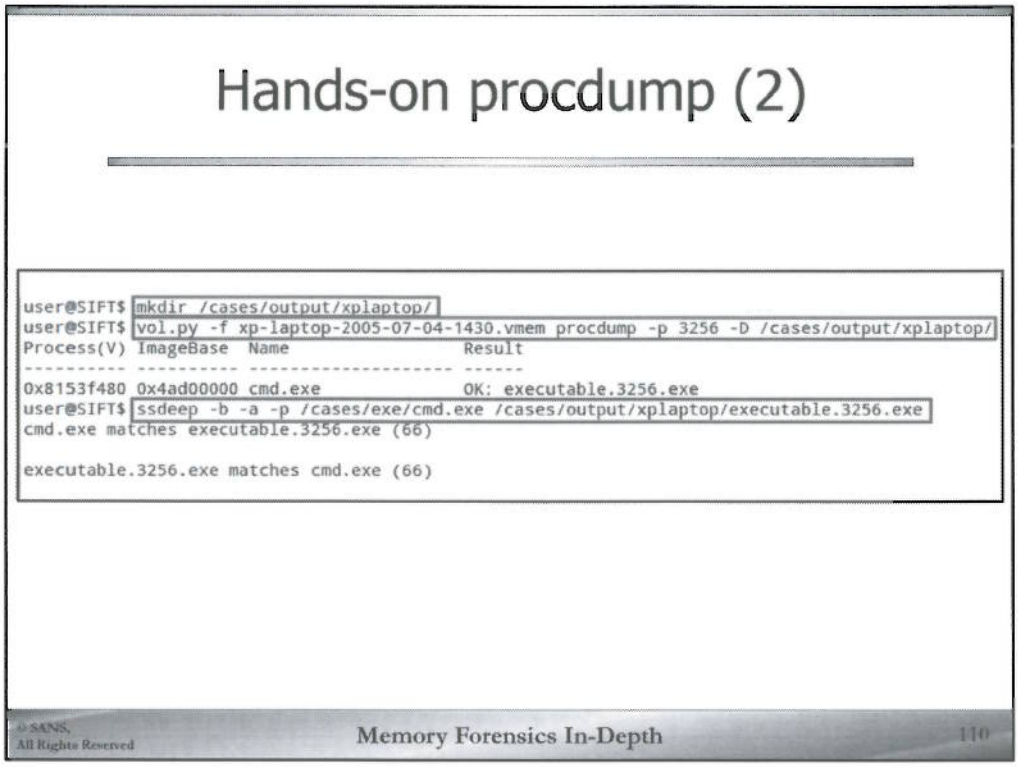
This plugin can recover all of the executables from a memory image, or you can limit it to a set of process id numbers. Here are the important command line options for this plugin:

--dump-dir – Sets the directory where recovered executables will be written. This directory must already exist.

--pid – Sets the PID or comma separated list of PIDs you wish to recover. This flag is optional. If omitted, procdump will recover the executable for every pid.

-v – Verbose mode

As with other Volatility plugins you still need to specify the filename and profile with -f and --profile, respectively.



We're going to start with the Command Prompt process, pid 3256, from the xp-laptop memory image.

First, let's create a directory in the output directory to hold this file.

```
$ mkdir /cases/output/xplaptop
```

Next, we call the procdump plugin on the xp laptop image:

```
$ vol.py -f /cases/xp-laptop-2005-07-04-1430.vmem --profile=WinXPSP2x86
procdump --pid=3256 --dump-dir=/cases/output/xplaptop
```

You should get this output:

```
**************************************************************************
Dumping cmd.exe, pid:   3256 output: executable.3256.exe
```

If you look in the /cases/output/xplaptop directory, you should see the file executable.3256.exe

Let's examine this file we've recovered. The recovered file is ~/output/executable.2308.exe.

First, let's see what kind of file we got. We change to the output directory and run the 'file' command on it to get started.

```
$ cd /cases/output/xplaptop
$ file executable.3256.exe
output/executable.3256.exe: PE32 executable (console) Intel 80386, for MS
Windows
```

We recovered a Windows executable. Good!

For your convenience we've included a copy of the Windows XP Command Prompt program in your cases directory*. It's under ~/Desktop/cases/exe/cmd.exe. First, let's compare the MD5 hash of the recovered file back to the original:

```
$ md5deep /cases/output/xplaptop/*
ff8a9a332a9471e1bf8d5cebb941fc66  /cases/output/executable.3256.exe
$ md5deep /cases/exe/cmd.exe
eeb024f2c81f0d55936fb825d21a91d6  /cases/Desktop/cases/exe/cmd.exe
```

The hashes don't match. That's to be expected. As noted above, we don't recover the original version of the program, but rather a version which has changes made to the import address table and potential relocations too. But the recovered file *should* be similar to the original file. Let's compare the recovered file to the original using fuzzy hashing. We use three command line flags:

-b – Bare filenames. Omit all path information

-a – Display all potential matches. This will be helpful later on when we're working with multiple files

-p – Pretty print all comparisons. Compare the files on the command line against each other

```
$ ssdeep -b -a -p ~/Desktop/cases/exe/cmd.exe ~/output/executable.3256.exe
executable.3256.exe matches cmd.exe (66)
cmd.exe matches executable.3256.exe (66)
```

Excellent! The file we recovered is a fuzzy match back to the original Command Prompt program.

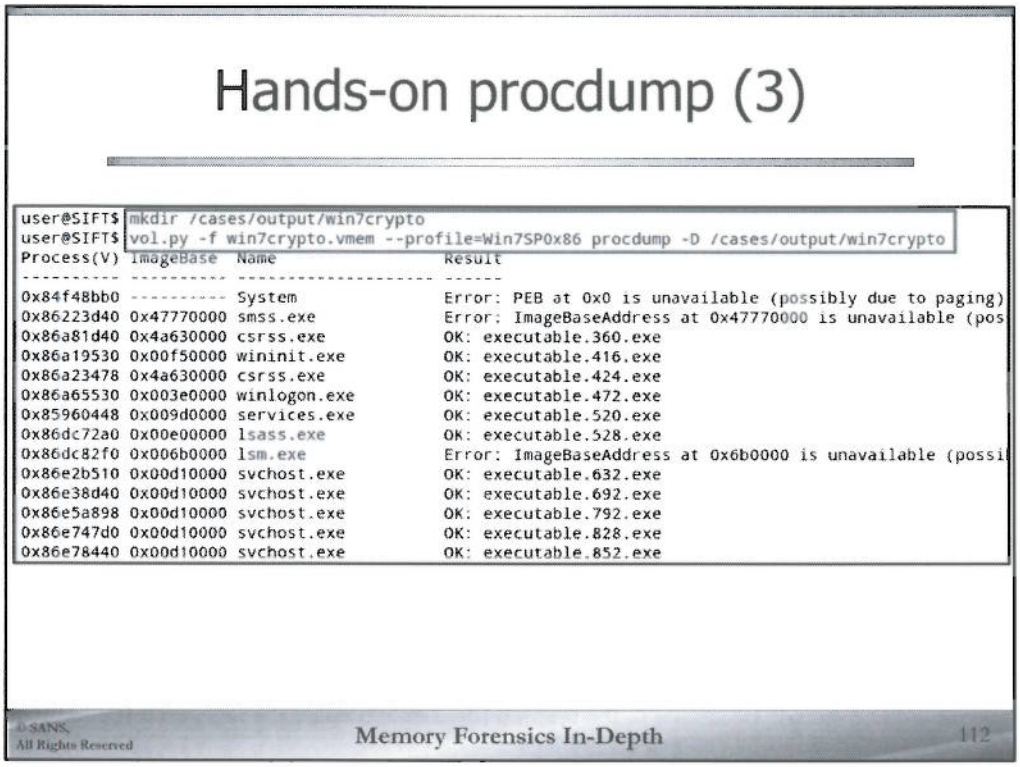
* Don't take our word for it that the file in ~/Desktop/cases/exe is a known good file. Try looking it up in the National Software Reference Library (NSRL)

```
$ md5deep ~/Desktop/cases/exe/cmd.exe | nsrllookup -s nsrl.kyr.us
```

Nsrllookup is a Free program for searching for hashes in large data sets. It was written to support the NSRL. Kyrus maintains a free nsrllookup server you can use. The program reports only the hashes of unknown programs. You can see the results of known hashes by adding a -k flag onto that command line, like this:

```
$ md5deep ~/Desktop/cases/exe/cmd.exe | nsrllookup -s nsrl.kyr.us -k
eeb024f2c81f0d55936fb825d21a91d6
/home/sansforensics/Desktop/cases/exe/cmd.exe
```

See http://nsrlquery.sourceforge.net/ for more details.



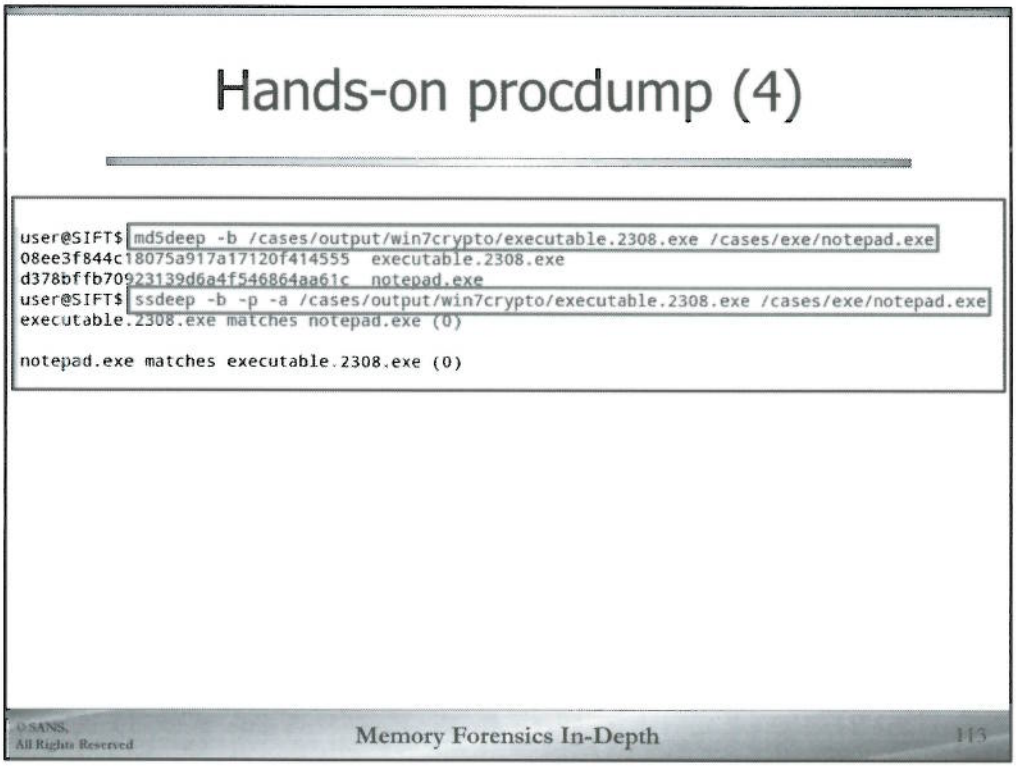
Let's run procexedump again, this time recovering all of the executables from the Windows 7 crypto memory image. We'll need to make another output directory so that we don't confuse the recovered executables from the xp laptop system and the Windows 7 system.

```
$ cd
$ mkdir output/win7crypto
$ vol.py -f /cases/win7crypto.vmem --profile=Win7SP0x86 procexedump --dump-
dir=/cases/output/win7crypto
```

You should get output which starts like:

```
*************************************************
Error: PEB not memory resident for process [4]
*************************************************
Error: ImageBaseAddress not memory resident for process [268]
*************************************************
Dumping csrss.exe, pid:    360 output: executable.360.exe
*************************************************
Dumping wininit.exe, pid:    416 output: executable.416.exe
*************************************************
Dumping csrss.exe, pid:    424 output: executable.424.exe
*************************************************
Dumping winlogon.exe, pid:    472 output: executable.472.exe
```

Notice in the output that several processes could not be recovered. Some part of their structure was not available and we couldn't even get to the PE header.



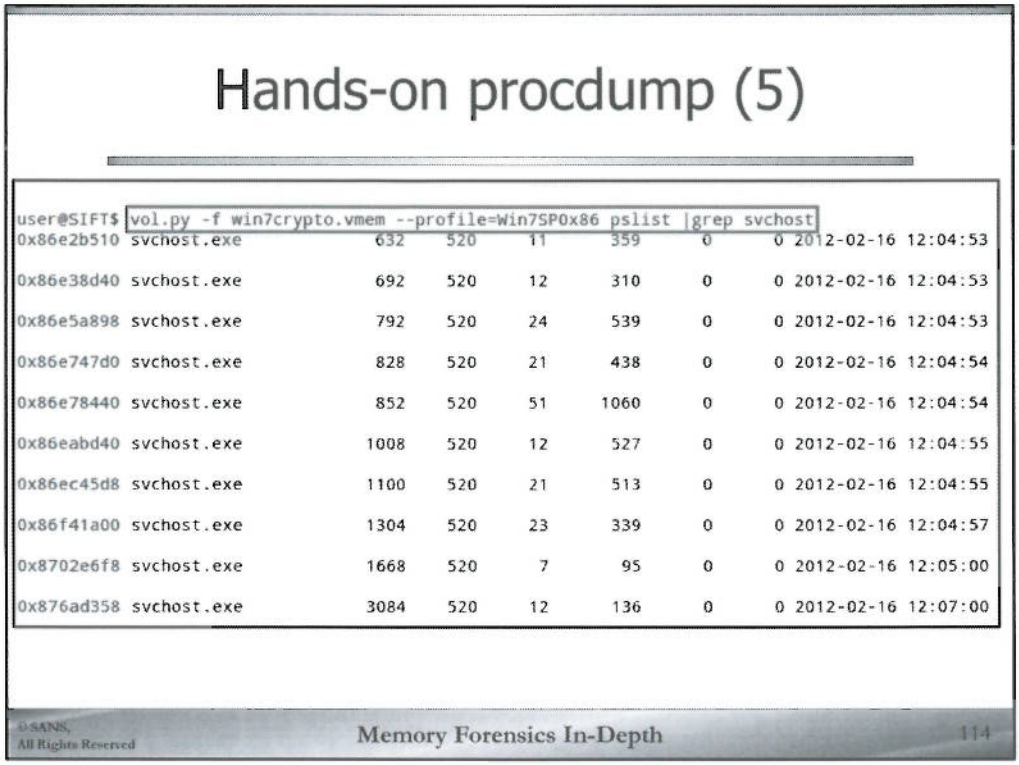
The first thing we're going to do is look at the Notepad.exe process, pid 2308. (If you lose track, you can always use pstree or pslist to remember a process id number). We've included a copy of the Windows 7 notepad.exe file in your images/exe directory. Again, we can compute the MD5 of the recovered program and the original and see that they are different:

```
$ md5deep -b /cases/output/win7crypto/executable.2308.exe
/cases/exe/notepad.exe
d378bffb70923139d6a4f546864aa61c  notepad.exe
08ee3f844c18075a917a17120f414555  executable.2308.exe
```

But as it turns out, these files do not match via fuzzy hashing either!

```
$ ssdeep -b -a -p /cases/output/win7crypto/executable.2308.exe
/cases/exe/notepad.exe
executable.2308.exe matches notepad.exe (0)

notepad.exe matches executable.2308.exe (0)
```

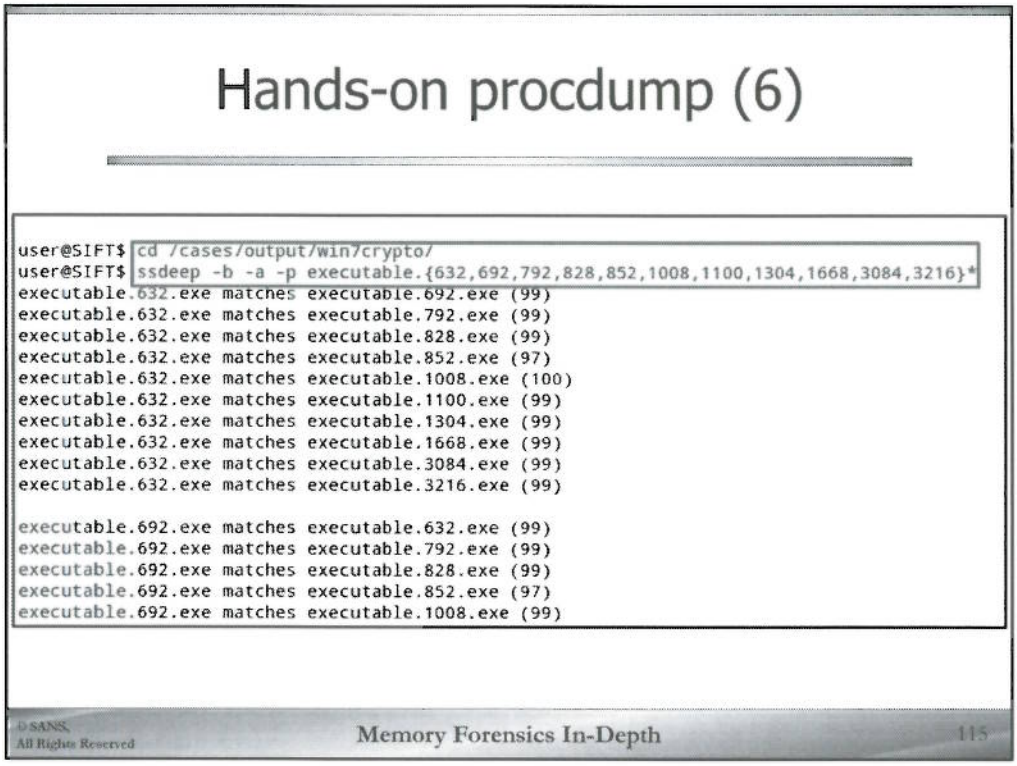
What happened? It turns out we weren't able to get several of the pages while attempting to recover the notepad.exe process. A large portion of the recovered file we got was nothing but zeros. There are lots of ways to see this. Try opening up the recovered file and the original in the Bless hex editor. Look at offset 0xcc00, and see how there is nothing but zeros until offset 0x27c00. You can also run the procexedump with the -v flag for verbose output. This will give you some idea of how many pages could not be obtained during the process recovery.



Malware authors will often attempt to hide their programs by naming them the same as a legitimate system process. There are several techniques for finding such programs. The one we have just done is to compare the modules we can extract from a memory image to the known good on the disk. Another method, which works for programs which run multiple times on the same system, is to compare the recovered files to each other. A process like svchost.exe is an ideal candidate for camouflaging malware. It legitimately runs several times on any given any system. Who would notice one more instance of it?

In the Windows 7 memory image we just examined, there were 11 running processes called svchost.exe. Were they all legitimate? Let's take a look. If they were all legitimate, then all of the copies we recovered should either be identical or very close to it. First, let's identify those eleven processes:

```
$ vol.py -f /cases/win7crypto.vmem --profile=Win7SP0x86 pslist | grep svchost

0x86e2b510 svchost.exe        632    520     11      359 2012-02-16 12:04:53
0x86e38d40 svchost.exe        692    520     12      310 2012-02-16 12:04:53
0x86e5a898 svchost.exe        792    520     24      539 2012-02-16 12:04:53
0x86e747d0 svchost.exe        828    520     21      438 2012-02-16 12:04:54
0x86e78440 svchost.exe        852    520     51     1060 2012-02-16 12:04:54
0x86eabd40 svchost.exe       1008    520     12      527 2012-02-16 12:04:55
0x86ec45d8 svchost.exe       1100    520     21      513 2012-02-16 12:04:55
0x86f41a00 svchost.exe       1304    520     23      339 2012-02-16 12:04:57
0x8702e6f8 svchost.exe       1668    520      7       95 2012-02-16 12:05:00
0x876ad358 svchost.exe       3084    520     12      136 2012-02-16 12:07:00
0x850bd708 svchost.exe       3216    520      9      304 2012-02-16 12:07:04
```



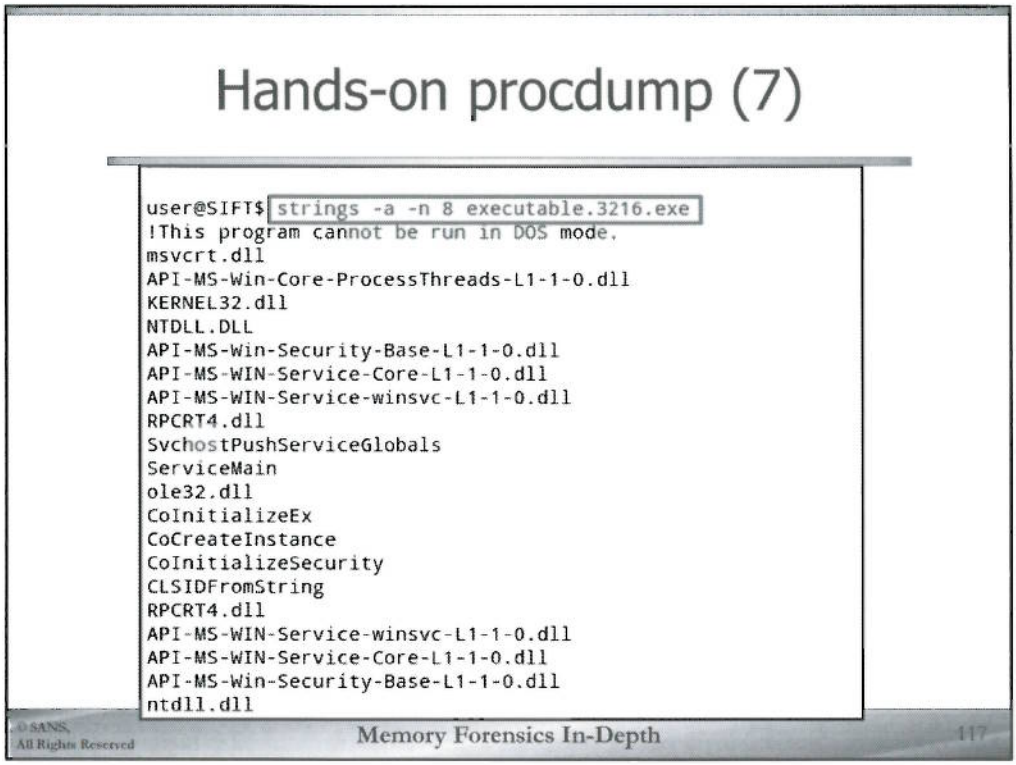
The pids are 632,692,792,828,852,1008,1100,1304,1668,3084, and 3216.

First, let's see if the modules we recovered for those PIDs are all the same size:

```
$ cd /cases/output/win7crypto
$ ls -l executable.{632,692,792,828,852,1008,1100,1304,1668,3084,3216}*
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.1008.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.1100.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.1304.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.1668.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.3084.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.3216.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.632.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.692.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.792.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.828.exe
-rw-r--r--  1 sansforensics  staff  20992 Mar  8 15:08 executable.852.exe
```

Yes, they are all the same size, a good sign. What about MD5 values?

```
$ md5deep -b
executable.{632,692,792,828,852,1008,1100,1304,1668,3084,3216}*
98061b0a6ac49819c41ea6ca9476ce9  executable.828.exe
825a82df6baeeb1421e9c0708d82da05  executable.632.exe
6b56b983c7e28800726c45de7d062d4b  executable.792.exe
825a82df6baeeb1421e9c0708d82da05  executable.692.exe
d93b15c1262eca0c684fab15d9b70d45  executable.852.exe
825a82df6baeeb1421e9c0708d82da05  executable.1008.exe
825a82df6baeeb1421e9c0708d82da05  executable.1100.exe
825a82df6baeeb1421e9c0708d82da05  executable.1304.exe
825a82df6baeeb1421e9c0708d82da05  executable.1668.exe
11bfed8167ac0e3ffdf76927d8bfb6d0  executable.3084.exe
8afadda32a42b5a2ab0f34f388ef60d0  executable.3216.exe
```



Some of them *are* identical, but not all. Finally, let's run our test of fuzzy hashing:

```
$ ssdeep -b -a -p
executable.{632,692,792,828,852,1008,1100,1304,1668,3084,3216}*
```

This will generate a LOT of output. For each block you should see a comparison of one file against all of the other files, like this:

```
executable.1668.exe matches executable.632.exe (100)
executable.1668.exe matches executable.692.exe (100)
executable.1668.exe matches executable.792.exe (82)
executable.1668.exe matches executable.828.exe (82)
executable.1668.exe matches executable.852.exe (80)
executable.1668.exe matches executable.1008.exe (100)
executable.1668.exe matches executable.1100.exe (100)
executable.1668.exe matches executable.1304.exe (100)
executable.1668.exe matches executable.3084.exe (82)
executable.1668.exe matches executable.3216.exe (82)
```

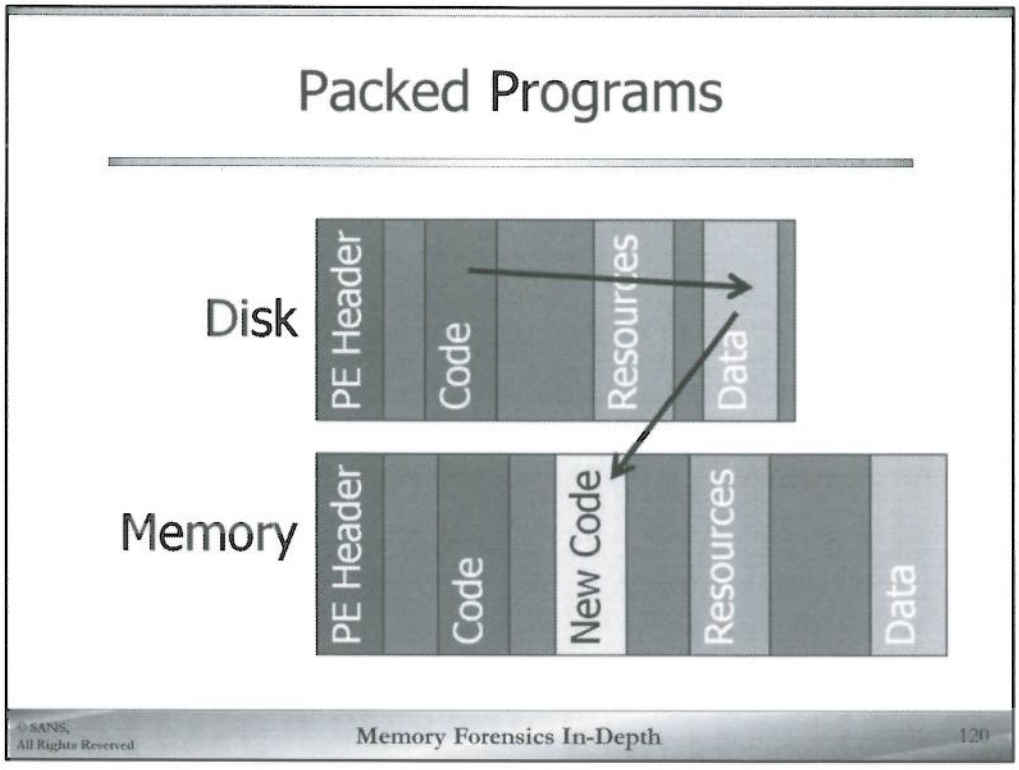
This file matched against all of the others. Looking at the rest of the blocks, they all matched too. This is a good sign. If one of them didn't match against the others, it would mean that version is different. That could be because that version of the module was relocated, or had been partially paged out. Or it could mean it was malware.

You can do some basic tests for the latter by looking at the program with reverse engineering tools. This isn't an RE class, but here's an easy technique to get a rough idea of what a program does. Run 'strings' on the

program. This prints out all of the ASCII strings in the program. For example, looking at one of our recovered svchost programs:

```
$ strings -a -n  8 executable.3216.exe
!This program cannot be run in DOS mode.
Rich
.text
`.data
.rsrc
@.reloc
msvcrt.dll
API-MS-Win-Core-ProcessThreads-L1-1-0.dll
KERNEL32.dll
NTDLL.DLL
API-MS-Win-Security-Base-L1-1-0.dll
API-MS-WIN-Service-Core-L1-1-0.dll
API-MS-WIN-Service-winsvc-L1-1-0.dll
RPCRT4.dll

ole32.dll
CoInitializeEx
CoCreateInstance
CoInitializeSecurity
CLSIDFromString
RPCRT4.dll
API-MS-WIN-Service-winsvc-L1-1-0.dll
API-MS-WIN-Service-Core-L1-1-0.dll
API-MS-Win-Security-Base-L1-1-0.dll
ntdll.dll
KERNEL32.dll
API-MS-Win-Core-ProcessThreads-L1-1-0.dll
msvcrt.dll

...

svchost.pdb
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Copyright (c) Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
    version="5.1.0.0"
    processorArchitecture="x86"
    name="Microsoft.Windows.Services.SvcHost"
    type="win32"
<description>Host Process for Windows Services</description>
```

The first few lines are things you should see at the start of every executable. There's a DOS stub, a legacy of the PE file format, followed by the section names of this PE file. Then there are a lot of function names, perhaps imported functions, and eventually the string svchost.pdb, which corresponds to a debugging file created when svchost was compiled. Looking at strings is a little like looking at goat entrails. You're not going to get a definitive answer, but it can get you started!

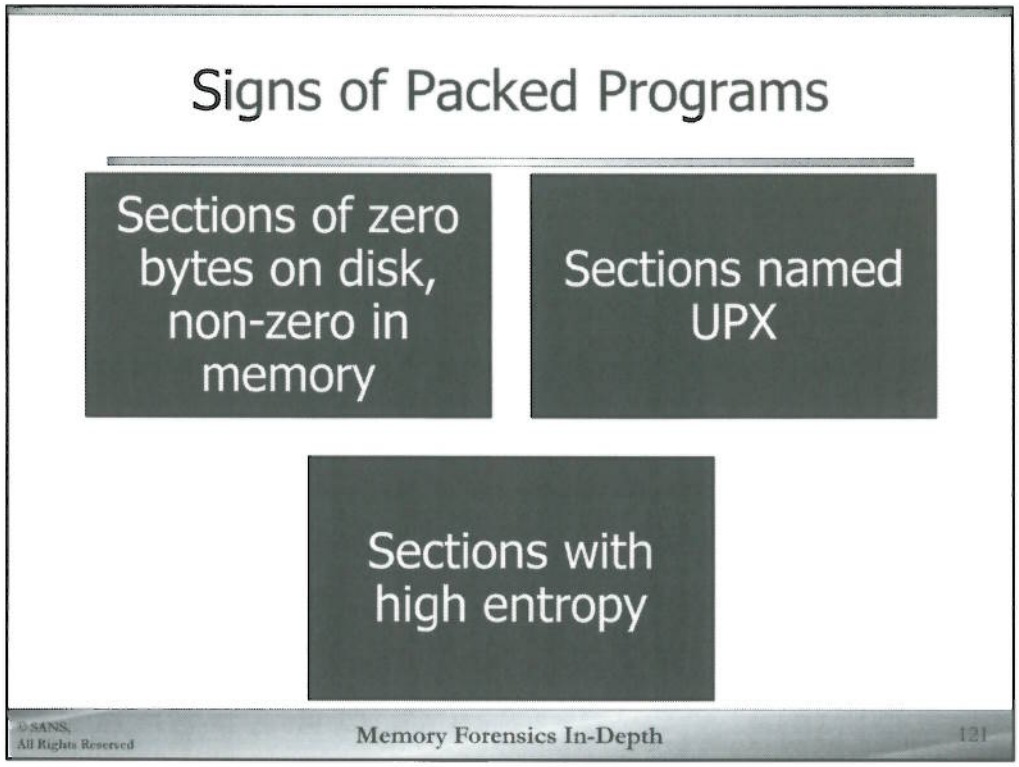


Now let's talk about non-standard PE files. The first kind we're going to discuss are packed programs. Packed programs store part or all of their code in a packed, and sometimes encrypted section of the PE file. When these programs are executed, an unpacking routine processes the packed data and writes an unpacked version to another section. Execution is then transferred to the newly unpacked code.

This technique was originally used to compress the size of legitimate executables, and is still used to protect the contents of legitimate programs against reverse engineering. Of course, any technology can be used by both the good guys and the bad guys. Malicious programmers use packing to bypass anti-virus systems and other system safeguards. They also want to frustrate reverse engineering.

The PE header of a packed program can be immediately obvious. To have a section which will hold the unpacked data, the programs which created packed executables will add a section to the PE file which is zero bytes on the disk, but a non-trivial number of bytes in memory. (Some legitimate Windows programs will create small sections in memory which don't exist on the disk.)

The picture above gives an example of what a packed program looks like. The original code in the program, when executed, reads the data in the file, decompresses it to a new section, and then executes it.

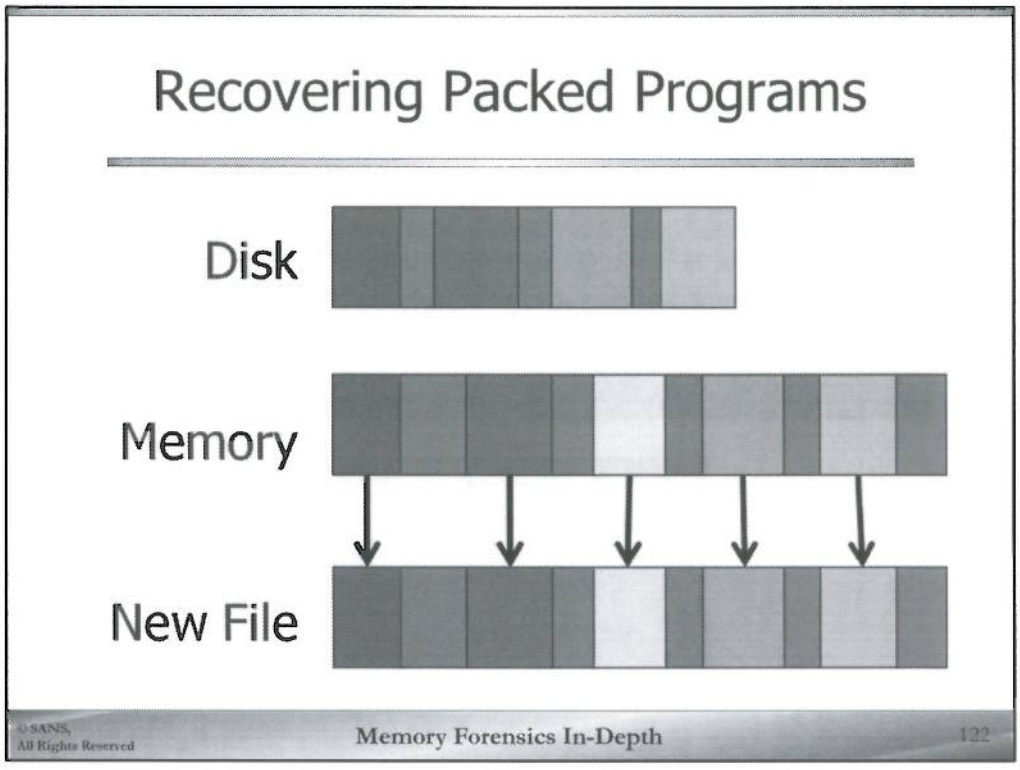


Although packers can work in an infinite number of ways, there are some telltale signs we can use to detect them. For example, any section which is zero bytes on the disk and a non-trivial number of bytes in memory is immediately suspect. This section represents where the unpacked data of a program will be stored in memory.

Another indicator is to look for section names which start with "UPX". One of the most popular packing programs is the Ultimate Packer for Executables (UPX). It's Free and open source. You can download a copy at http://upx.sourceforge.net/. When UPX creates packed executables it creates sections named UPX0 and UPX1. (Programs produced by Microsoft's Visual C compiler have names like .code and .reloc.) So finding programs with sections whose names begin with UPX is a pretty good indicator of a packed program.

UPX is well known, and as far as packers go, fairly easy to unpack. As it turns out, however, many other packers *also* use section names which begin with UPX. These packers do this to confuse reverse engineers as to which packer is being used on any particular program. Although it's easy to see *a* packer is being used, it's difficult to unpack a program statically without knowing *which* packer is being used. Packing programs have many techniques for obfuscating data. They change often and are designed to frustrate reverse engineering.

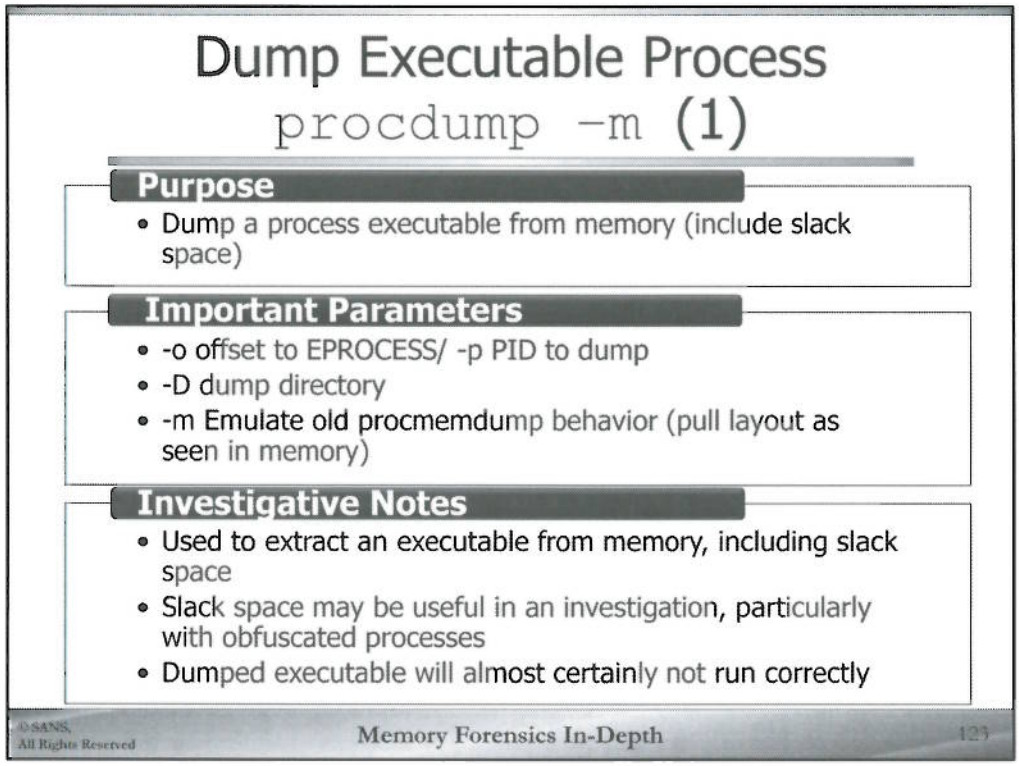
We can also look for sections which have high entropy. Entropy, in the information theory sense of the word, is the degree of randomness between bytes. ASCII text, for example, is restricted to the letters between A-z and punctuation symbols. Each byte can only be so different from the ones which precede it. It has a fairly low entropy. Compiled code has a higher entropy, as it can include more possible bytes than ASCII text. Packed or encrypted data, however, has an even higher entropy. We can measure the entropy of any block of data. In a PE file we would expect to find code and ASCII text. But if the entropy is higher than what we expect, we may be looking at a block of compressed or encrypted data. Such a finding could indicate a packed program.



Recovering a packed program presents both a huge opportunity and a challenge for the forensic examiner. We can recover all of the original sections as they existed on the disk. But what about the extra section of data which came from the decompressed data? Obviously we don't want to ignore it. That section contains the unpacked code which we are most interested in. But we can't put this data into the original PE file—there is no place for it.

One solution is to recover the previously packed data into a new file. This file can then be analyzed using tools like IDA Pro. It's not simple, but does work, and works quite well.

Another solution is to dump out the process memory space of the entire executable, without attempting to put the sections back into the format of the PE file. The Volatility plugin procmemdump does this for you. It works just like procexedump, but creates a file which holds the contents of virtual memory for the module.

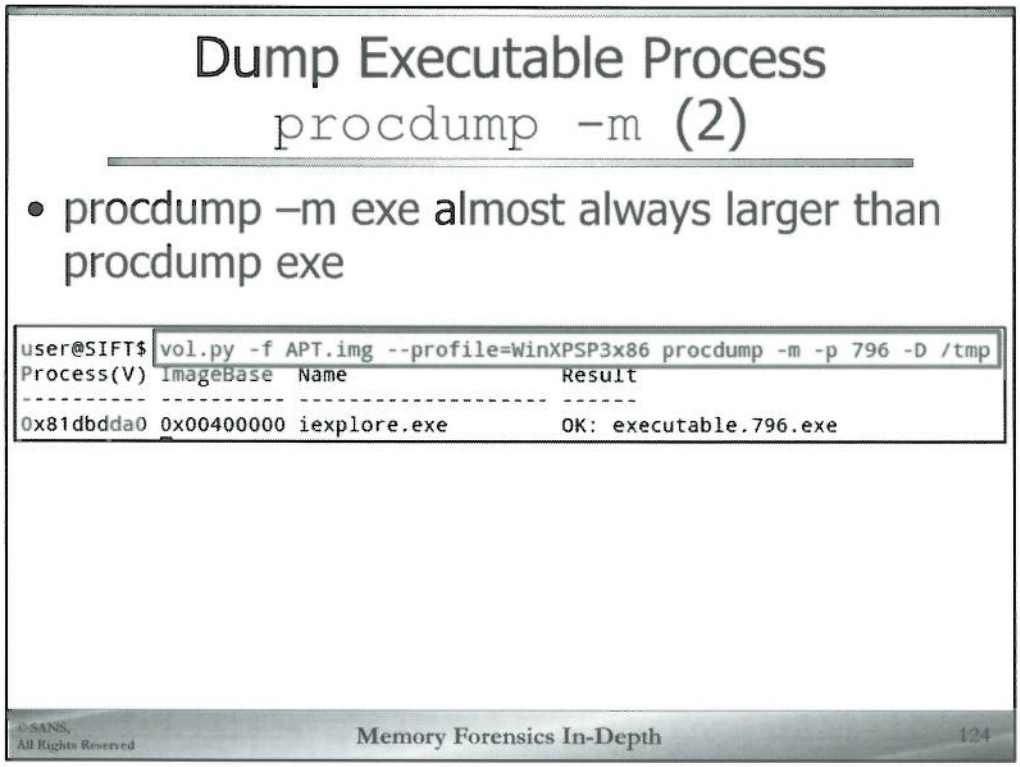


The procdump plugin with the -m option is used to extract an executable from a physical memory dump. Using this option, slack space is handled between sections is handled. The default run of procdump plugin ignores slack space in the dump while the 'procdump -m' plugin captures it.

There is a trade space for obtaining slack space. Some PE analysis tools can parse procdump output but are not be able to analyze the output of 'procdump –m' due to the addition of slack space. Additionally, if file scanners are looking for a file of a particular size, procmemdump will output a differently sized file. The obvious argument for dumping slack space is that it may contain additional data (particularly true for .bss sections). The 'procdump –m' plugin operates very similarly to the volshell lab dumping method.

Note that if PE headers have been modified, you may have to specify the '-u' command line option to force volatility to try to dump the executable even though some safety checks cannot be validated. If this still fails, there's always volshell!

The code for this plugin is located in the file volatility/plugins/procdump.py.



Note that the file executable file created with 'procdump -m' is larger than that created with the default run of procdump. This is because of the slack space.

 # vol.py -f /cases/bootcamp/APT.img procdump -m -D /tmp/dump -p 796

Process(V) ImageBase Name Result

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

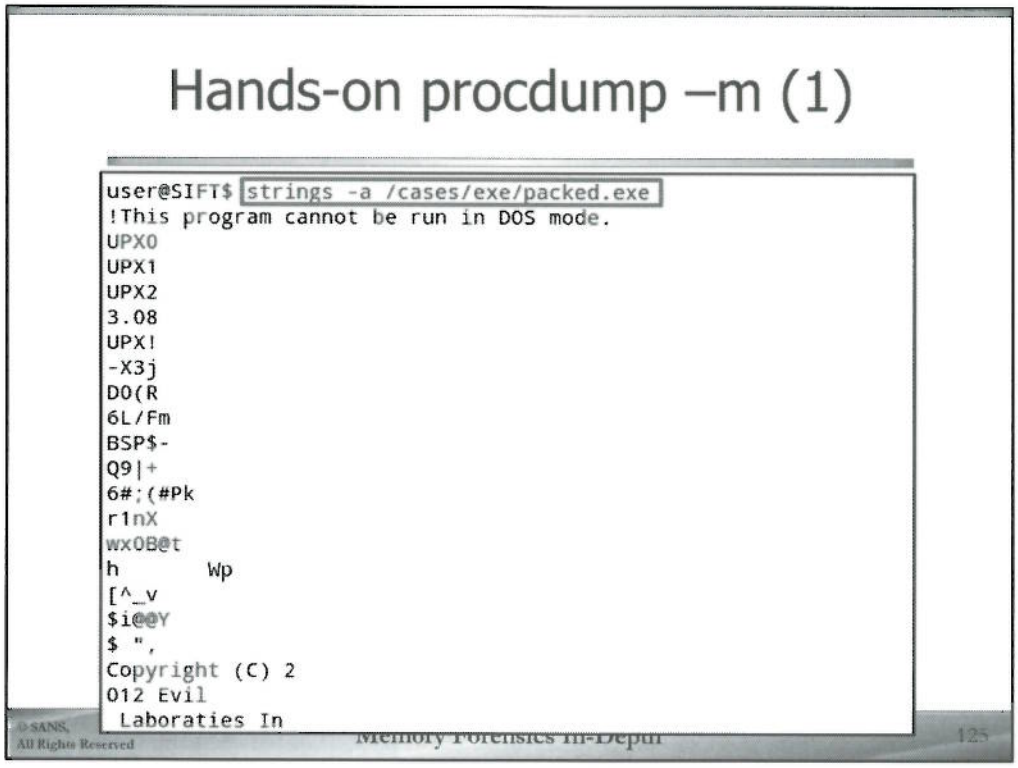
0x81dbdda0 0x00400000 iexplore.exe OK: executable.796.exe

 # ls -l /tmp/executable.796.exe

-rw-rw-rw- 1 root root 625664 2014-01-05 06:07 /tmp/executable.796.exe

 # ls -l /tmp/dump/executable.796.exe

-rw-rw-rw- 1 root root 634880 2014-01-05 06:43 /tmp/dump/executable.796.exe

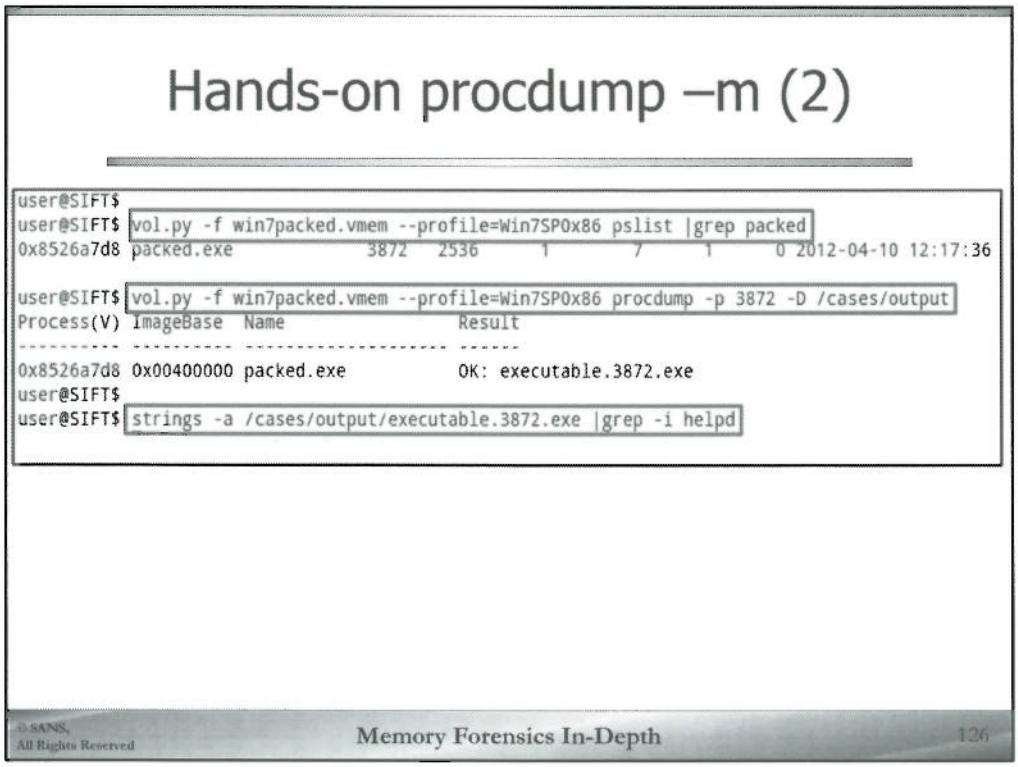


Let's walk through an example of this together. In your cases directory you should find an executable called packed.exe. This program is an example of a Windows program which has been packed. You can run this program, and it works just fine, but it's been obfuscated against reverse engineering. When run, this program unpacks itself into another section and then executes the unpacked code. In particular, this program displays a bunch of information, including a phone number for the help desk. If you look at the strings for this program, however, you won't see any of these strings or the phone number. Let's run strings on the program to see what we can see. We'll use the 'less' pager to show us one screen of text at a time.

```
$ strings -a /cases/exe/packed.exe | less
```

```
You should see output like this:
!This program cannot be run in DOS mode.
UPX0
UPX1
… output truncated …
```

Although there are hints of meaningful data in there, there isn't much to go on. On the other hand, those strings UPX0, UPX1, and UPX2 are indicative of *something* afoot. The UPX packer is a common packing program. Programs compressed by UPX have sections with those names. (Granted, many other packers use those section names too, in an attempt to confuse reverse engineers, too. But the presence of those strings lets us know immediately that this program is probably packed somehow.)



Now let's look at the memory image win7packed.vmem, which contains a running instance of this process. This system is running Windows 7, SP0, and so uses the profile Win7SP0x86. You will first find the pid of this process, which you can get using any of the process plugins. For example:

$ vol.py –f /cases/win7packed.vmem --profile=Win7SP0x86 pslist | grep packed

0x8526a7d8 packed.exe 3872 2536 1 7 2012-04-10 12:17:36

We can see the process packed.exe was running as pid 3872. If we use procexedump on this process, we don't get any more information than we did running strings on the original executable.

$ vol.py –f /cases/win7packed.vmem --profile=Win7SP0x86 procdump --dump-dir=/cases/output -p 3872

Dumping packed.exe, pid: 3872 output: executable.3872.exe

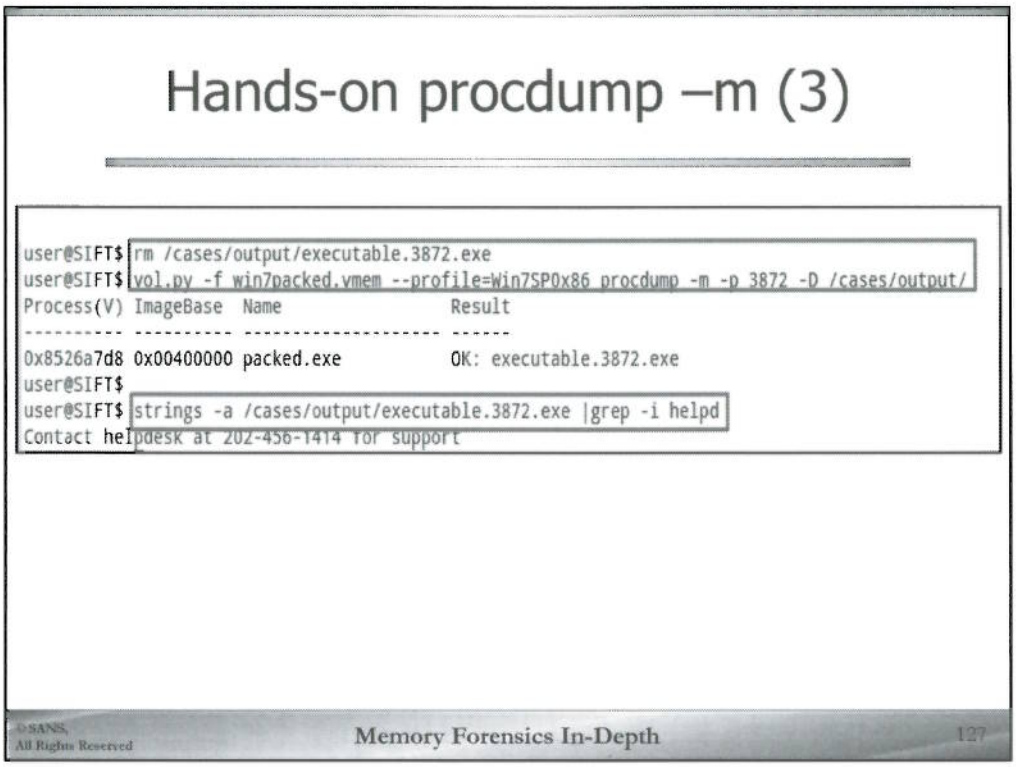
$ strings ~/output/executable.3872.exe | grep -i helpd

[no output]

But if we use procdump -m, and dump out the whole address space, we get quite a bit more! First, we must remove the existing recovered file:

$ rm –f /cases/output/executable.3872.exe

And now with procdump -m:



$ vol.py –f /cases/win7packed.vmem --profile=Win7SP0x86 procdump –m --dump-dir=/cases/output -p 3872

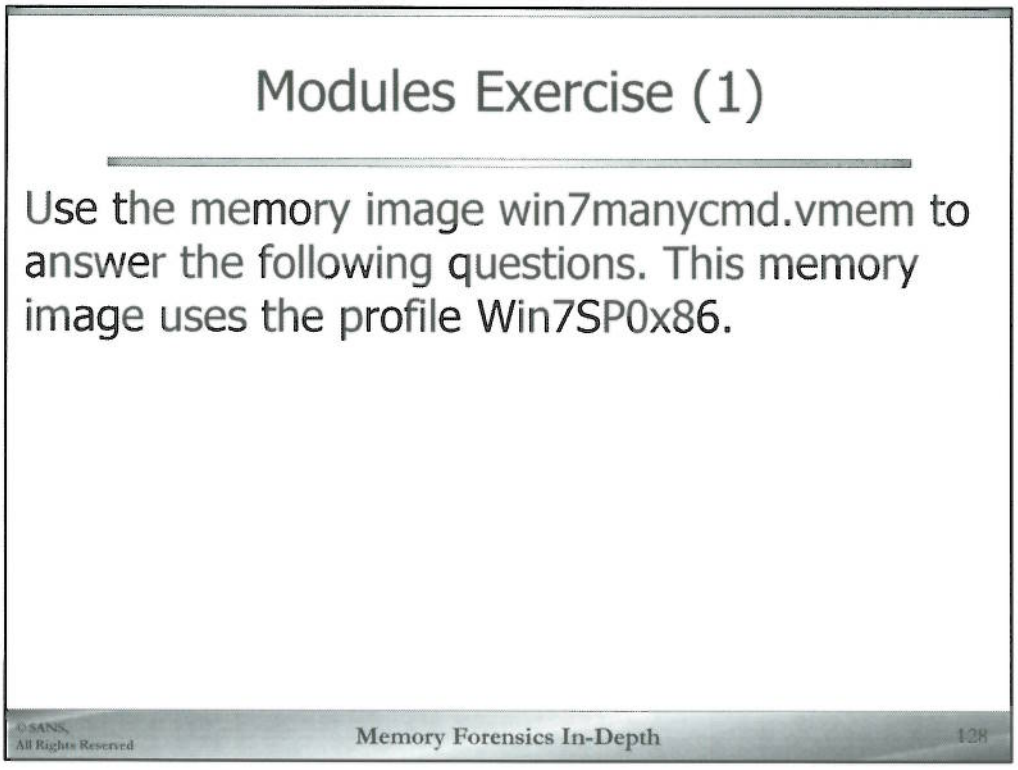
Dumping packed.exe, pid: 3872 output: executable.3872.exe

$ strings –a /cases/output/executable.3872.exe | grep -i helpd

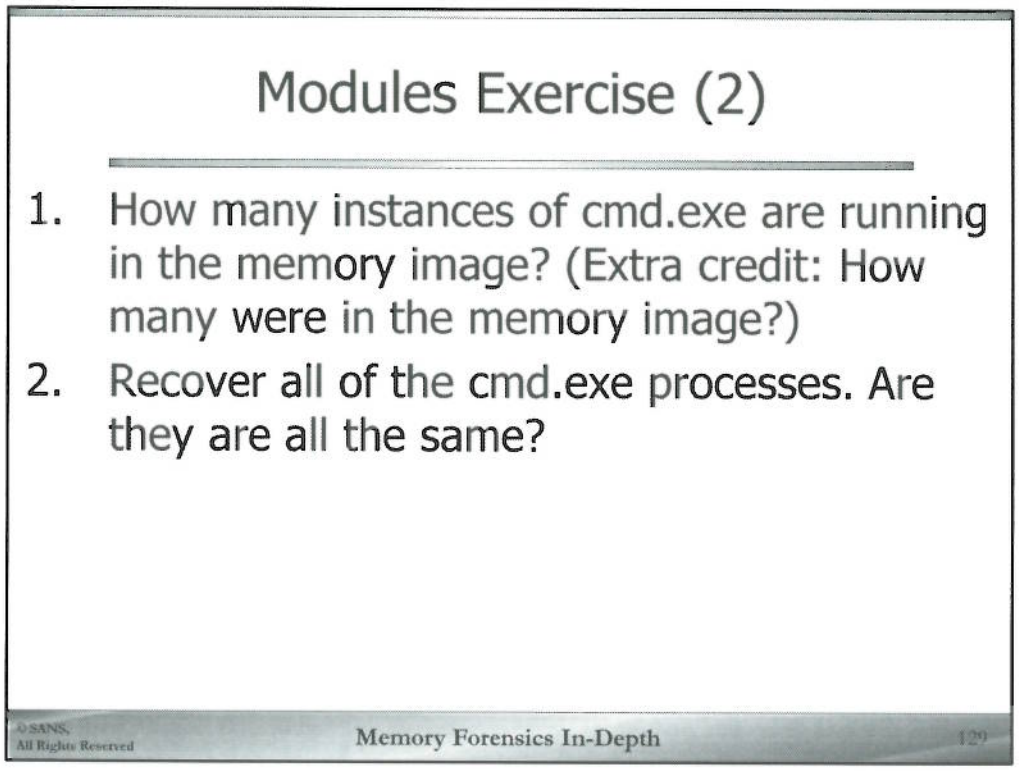
Contact helpdesk at 202-456-1414 for support

Victory!

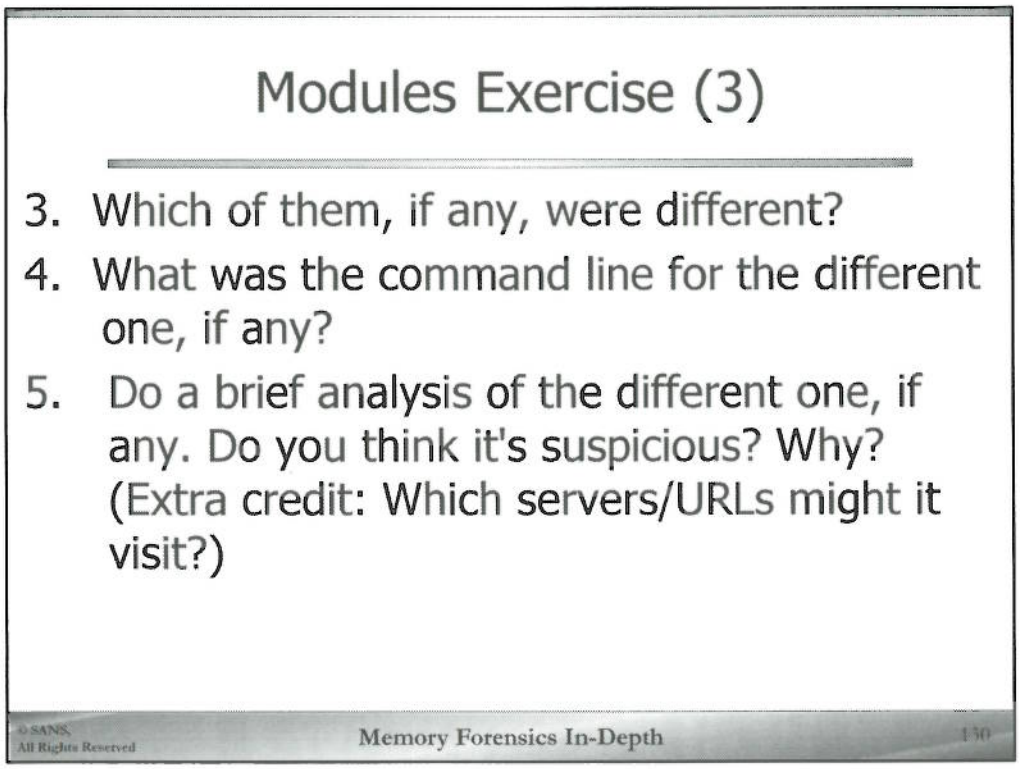
(Bonus questions for those with extra time: 1. Who does that phone number connect to? 2. What do the strings in this program say about your parent?)



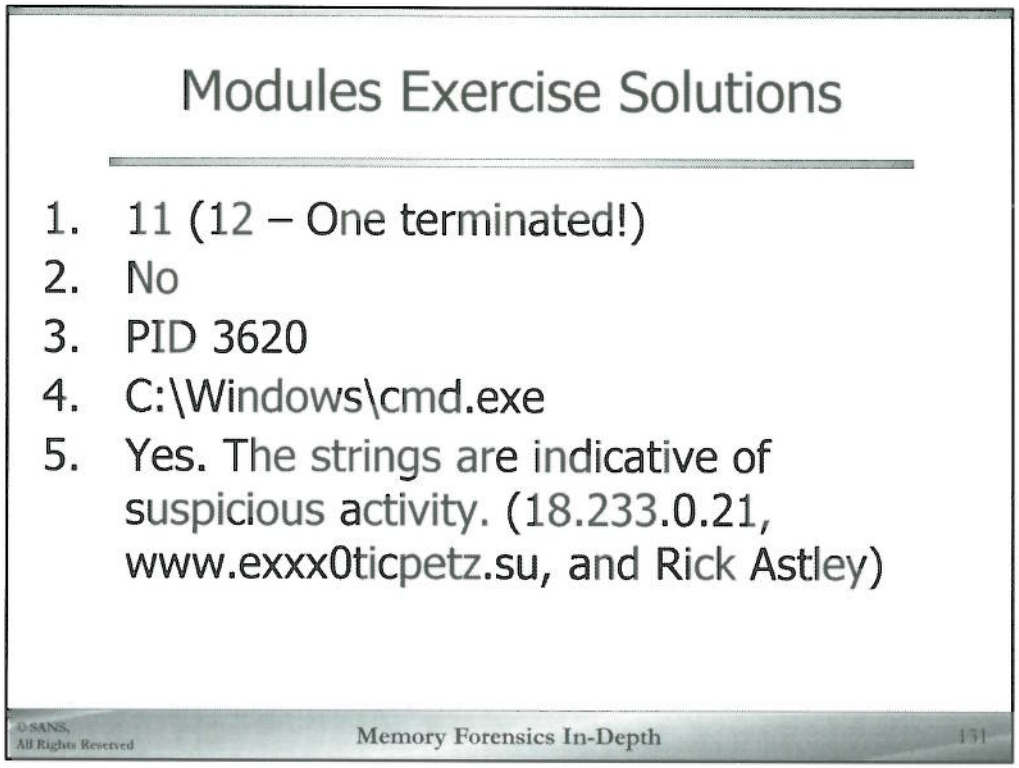
This page intentionally left blank.



This page intentionally left blank.



This page intentionally left blank.



1. How many instances of cmd.exe were running in the memory image?

   You can see the instances of cmd.exe using a number of plugins. Here's an example using pslist. We've used the grep command to make the output easier to read:

```
$ vol.py -f /cases/win7manycmd.vmem --profile=Win7SP0x86 pslist | grep cmd
0x872e7590 cmd.exe          3456   2440      1       23 2012-03-08 21:10:29
0x861644c0 cmd.exe          3696   3456      1       22 2012-03-08 21:10:40
0x87018b10 cmd.exe          3732   2440      1       19 2012-03-08 21:10:47
0x86cc9030 cmd.exe          3764   2440      1       19 2012-03-08 21:10:50
0x84fe46e0 cmd.exe          3796   2440      1       22 2012-03-08 21:10:51
0x973fb7e8 cmd.exe          3828   2440      1       23 2012-03-08 21:10:53
0x86e62960 cmd.exe          3856   3828      1       23 2012-03-08 21:10:56
0x86ea14e8 cmd.exe          3864   3856      1       23 2012-03-08 21:10:57
0x86fda030 cmd.exe          3872   3864      1       23 2012-03-08 21:10:57
0x86c40d40 cmd.exe          2560   3872      1       22 2012-03-08 21:11:07
0x86e3dd40 cmd.exe          3620   2440      1        6 2012-03-08 21:12:37
0x871a1938 cmd.exe          2416   1512      0 -------- 2012-03-08 21:12:40
   2012-03-08 21:12:41

$ vol.py -f /cases/win7manycmd.vmem --profile=Win7SP0x86 pslist | grep cmd
   | wc -l
12
```

2. Recover all of the cmd.exe processes. Are they all the same?

Let's use procexedump to extract just the cmd.exe processes:

```
$ mkdir /cases/output/win7manycmd
$ vol.py -f /cases/win7manycmd.vmem --profile=Win7SP0x86 procdump --dump-
    dir=/cases/output/win7manycmd --
    pid=3456,3696,3732,3764,3796,3828,3856,3864,3872,2560,3620,2416
```

And then compare them:

```
$ cd /cases/output/win7manycmd
$ ls -l
```

```
total 5936
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.2560.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3456.exe
-rw-r--r--  1 sansforensics  staff    6144 Mar  9 07:30 executable.3620.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3696.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3732.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3764.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3796.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3828.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3856.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3864.exe
-rw-r--r--  1 sansforensics  staff  301568 Mar  9 07:30 executable.3872.exe
```

They are NOT all the same size! Hmmm….

3. Which of them, if any, were different?

Let's jump straight to fuzzy hashing. We're most concerned with the one which is a different size, 3620:

```
$ ssdeep -b -a -p *
```

```
...
executable.3620.exe matches executable.2560.exe (0)
executable.3620.exe matches executable.3456.exe (0)
executable.3620.exe matches executable.3696.exe (0)
executable.3620.exe matches executable.3732.exe (0)
executable.3620.exe matches executable.3764.exe (0)
executable.3620.exe matches executable.3796.exe (0)
executable.3620.exe matches executable.3828.exe (0)
executable.3620.exe matches executable.3856.exe (0)
executable.3620.exe matches executable.3864.exe (0)
executable.3620.exe matches executable.3872.exe (0)
```

It doesn't match ANY of the others. (The others all seem to match each other, which is good.)

4. What was the command line for the different process? Let's take a look:

```
$ vol.py -f /cases/win7manycmd.vmem --profile=Win7SP0x86 dlllist --
   pid=3620
pid:   3620
cmd.exe
EPROCESS virtual: 0x86e3dd40
EPROCESS  offset: 0x3de3dd40
      Started at: 2012-03-08 21:12:37
    Command line: "C:\Windows\cmd.exe"
 Image Path Name: C:\Windows\cmd.exe
 Environment off: 0x0c74b7f0
```

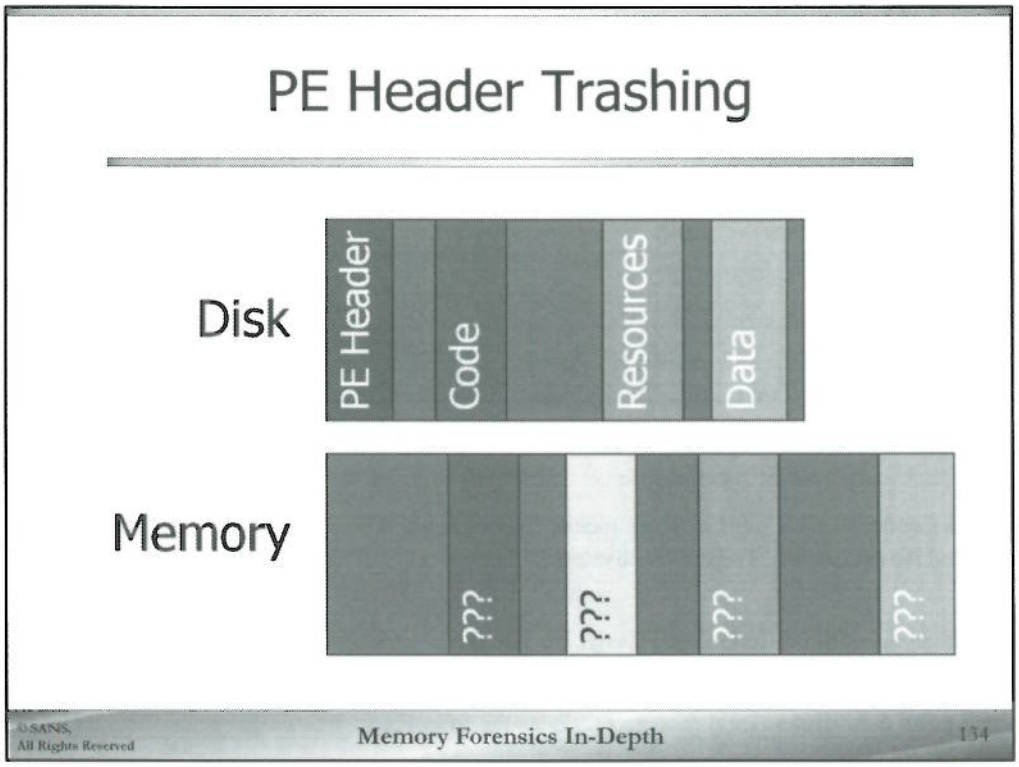
Well, that looks normal– HEY! Just a cotton pickin' minute there! The command prompt process isn't run out of C:\Windows! It's run out of C:\Windows\System32.

5. Do a brief analysis of the different one, if any. Do you think it's suspicious? Why?

Running strings on the different cmd process produces the following:

```
$ strings -a executable.3620.exe:
Copyright (C) 2010 EvilEvil Laboratories Inc. All Rights Reserved.
Connecting to command and control server
SUCCESS
Connection Failed.
Drone available for DDOS attacks
www.exxx0ticpetz.su
18.233.0.21
Press any key to compromise host
http://www.youtube.com/watch?v=dQw4w9WgXcQ
LoadLibraryA
RegOpenKeyW
HKCU
Cracking passwords...
done!
uuname
Username = %s
passname
Password = %s
```

None of that looks good! (If you look at this program in a disassembler or IDA Pro, however, the program doesn't actually do anything. The program displays those strings and then waits.)

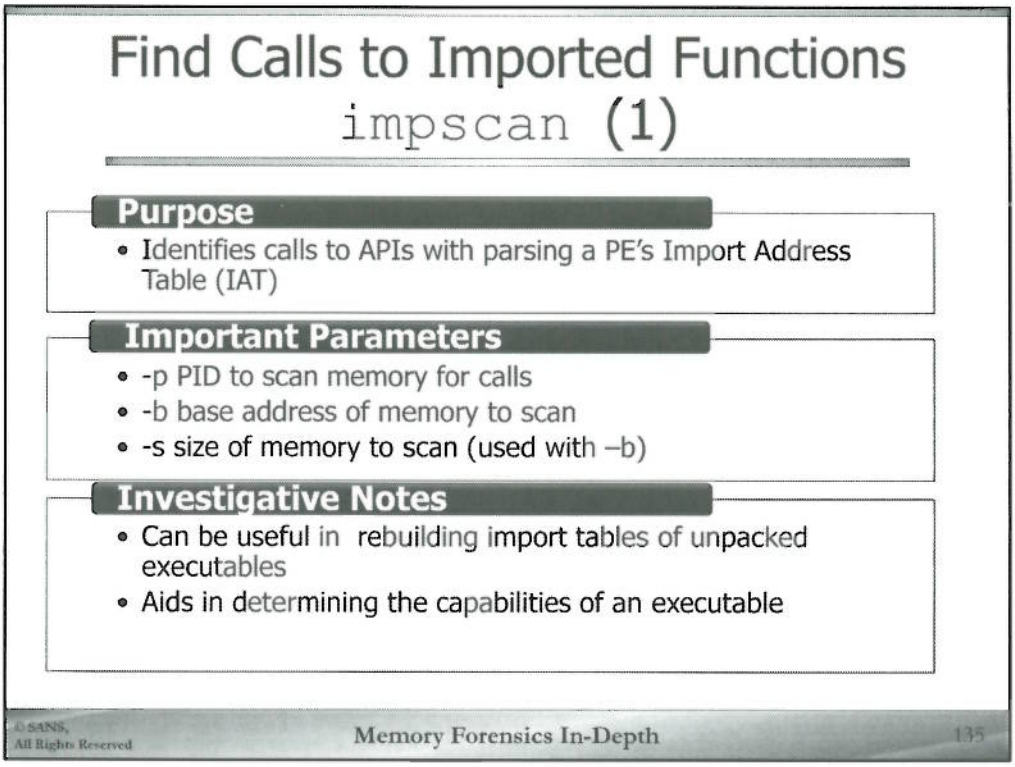


As part of an effort to defeat memory forensics, some malicious programs trash their PE header when they execute. More frustratingly, some just change important values in the PE header to nonsense. For example, once a program is loaded into memory, it could change the copy of the PE header in memory so that every section was absolutely enormous. When a memory forensics program attempts to use the PE header to copy out the sections, it gets lost attempting to copy the bogus data.

Certainly trashing the PE header does make our job more difficult. We can still see the program's code and other data in memory. We can usually automatically find the breaks between those blocks which make up the sections on the disk. But having to recreate a PE header is painful. Attempting to reverse engineer blobs of binary code without knowing where they were based makes the process more time consuming (and manual), but it is still possible.

But at the same time, the trashing of the PE headers is a clear example of the rootkit paradox. Every module should have a PE header, and that header should be valid. If it's not, then something is clearly wrong with that process. We can immediately and automatically detect errors or absurd values in the headers. If found, we've immediately found a piece of malware, and being found is the last thing a malware author wants!

Even if the PE header has been damaged, we can recover the contents of the module, as we would for a packed program, by copying out the virtual address space reserved for that module. The information on that reservation is contained in the Virtual Address Descriptor, or VAD, which we will cover in depth in the next section.

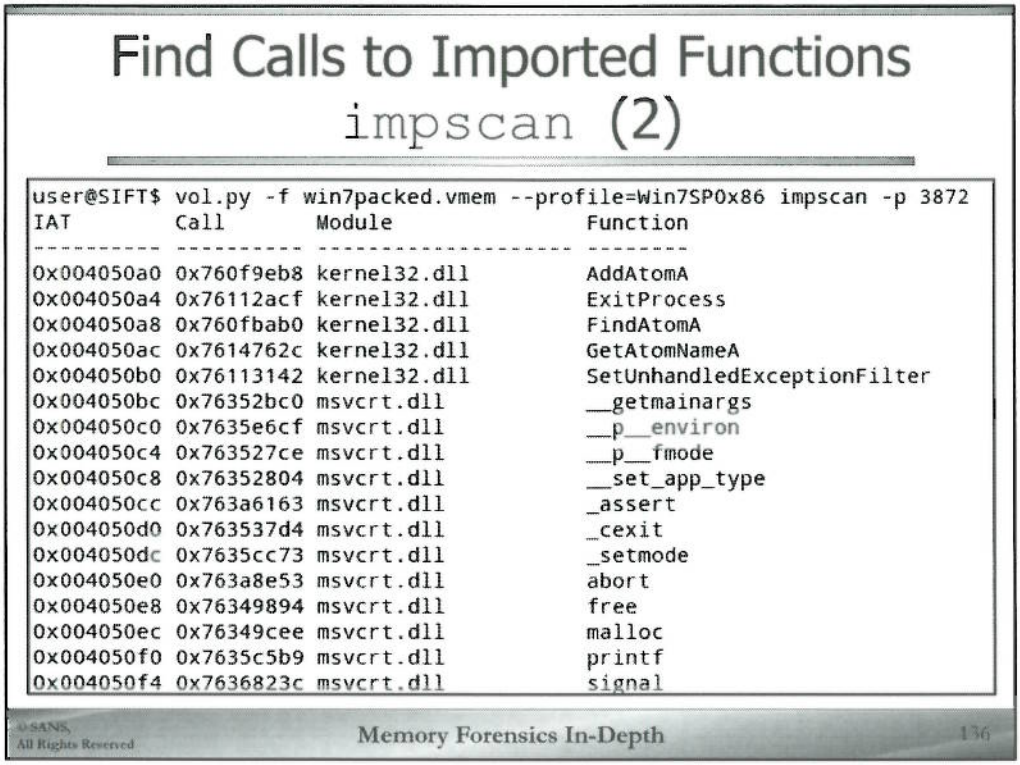


The impscan plugin is used to discover calls to imported functions. This is very useful for malware analysts trying to determine the capabilities of an executable. It is also sometimes useful for rebuilding the import tables of unpacked executables that are dumped from memory. Both of these functions are beyond the scope of this course, however we will quickly introduce two imports to be on the lookout for.

The first import to look for is CreateRemoteThead. This import is almost always indicative of code injection. The second import to look for is WriteProcessMemory. This import is sometimes used for code injection. In other cases it is used to modify the memory of another process (think hooking). You want to be on the lookout for both of these actions. Both APIs have legitimate purposes, but both are often used by malware authors. Both imports are located in kernel32.dll.

Note that impscan does not currently work on WoW64 processes. A WoW64 process is a 32 bit process running on a 64 bit OS.

The code for this plugin is located in the file volatility/plugins/malware/impscan.py



Sample output from the impscan plugin is shown below:

user@SIFT$ vol.py -f win7packed.vmem --profile=Win7SP0x86 impscan -p 3872

IAT Call Module Function

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

0x004050a0 0x760f9eb8 kernel32.dll AddAtomA

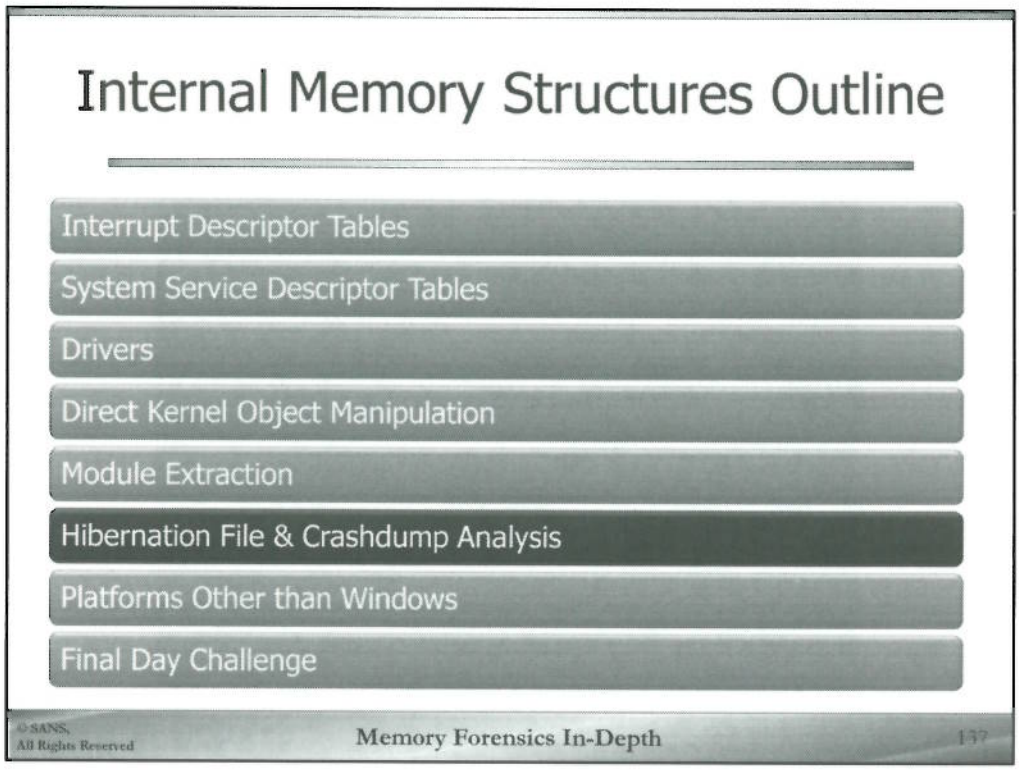
0x004050a4 0x76112acf kernel32.dll ExitProcess

0x004050a8 0x760fbab0 kernel32.dll FindAtomA

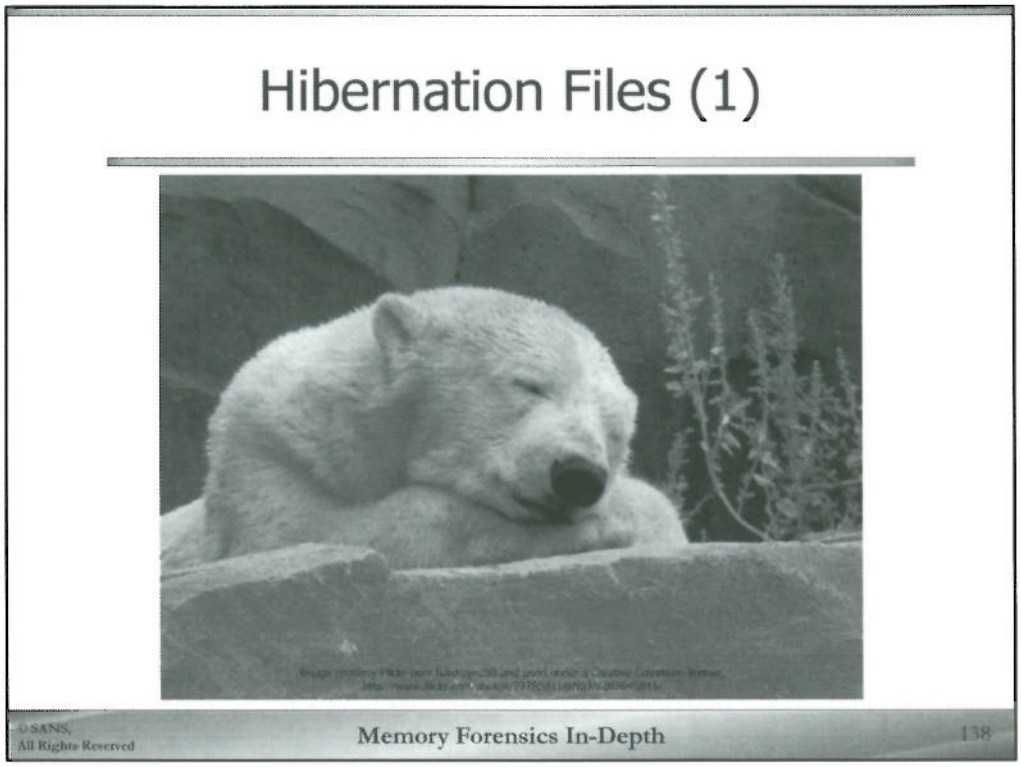
0x004050ac 0x7614762c kernel32.dll GetAtomNameA

0x004050b0 0x76113142 kernel32.dll SetUnhandledExceptionFilter

… output truncated …



This page intentionally left blank.



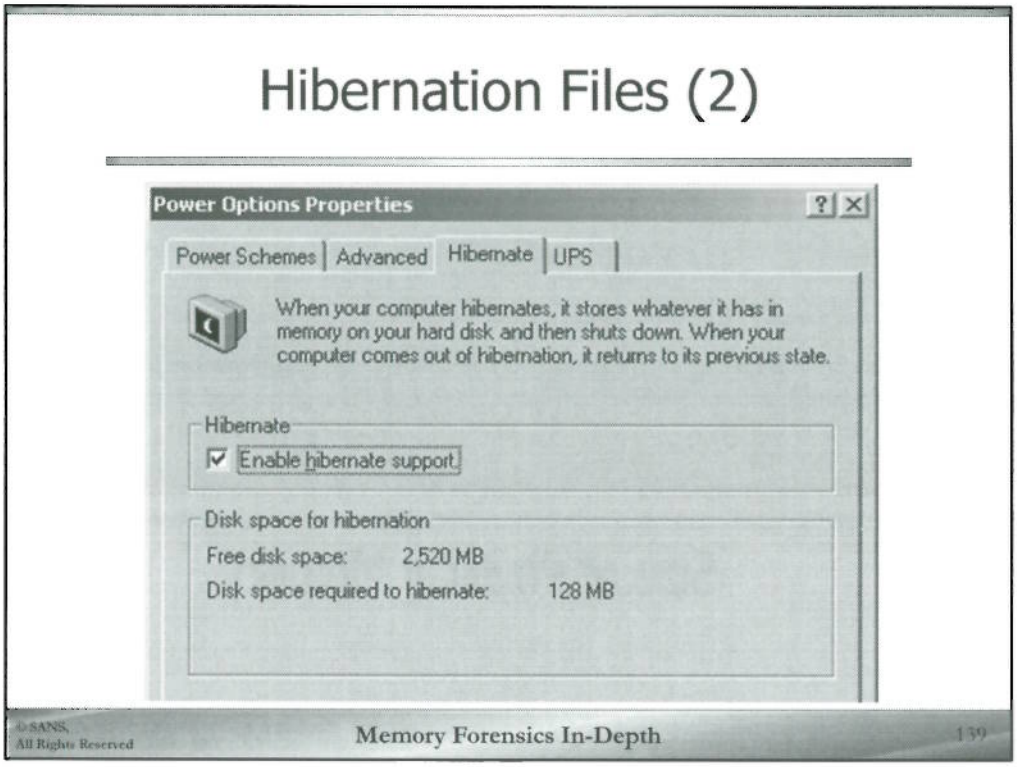
Without question the best source of existing Windows memory images come from hibernation files. Hibernation was a feature added in Windows 2000 which allows the machine to save the current state to the disk. This allows the user to return to the same state when the power is turned on the next time. It was originally intended for laptop users who wanted to extend their battery life.

The saved state includes a compressed, serialized memory image and even the state of the processor itself. (Remember how RFC 3227 included preserving the contents of the processor registers? The hibernation file has that information!) All of the state information is saved to a single file on the disk, usually called hiberfil.sys, but we'll refer to it as the hibernation file. A serialized memory image is one where the full contents of memory have been obtained. Unlike imaging memory with a software program, no data is missed because it was moving while the image was being captured.

At the start of the hibernation process, each process is notified and given a chance to do anything it wants. For example, a program could display a dialog box, "Are you sure you want to hibernate now and interrupt your file transfer?". But programs could also wipe sensitive data from memory, such as encryption keys.

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

Prior to Windows 7, unfortunately, hibernation was not supported on any Windows system with more than 4GB of RAM. This restriction applied to both 32 and 64 bit versions of the operating systems. See http://support.microsoft.com/kb/888575/ for more details on why Microsoft imposed this limitation.

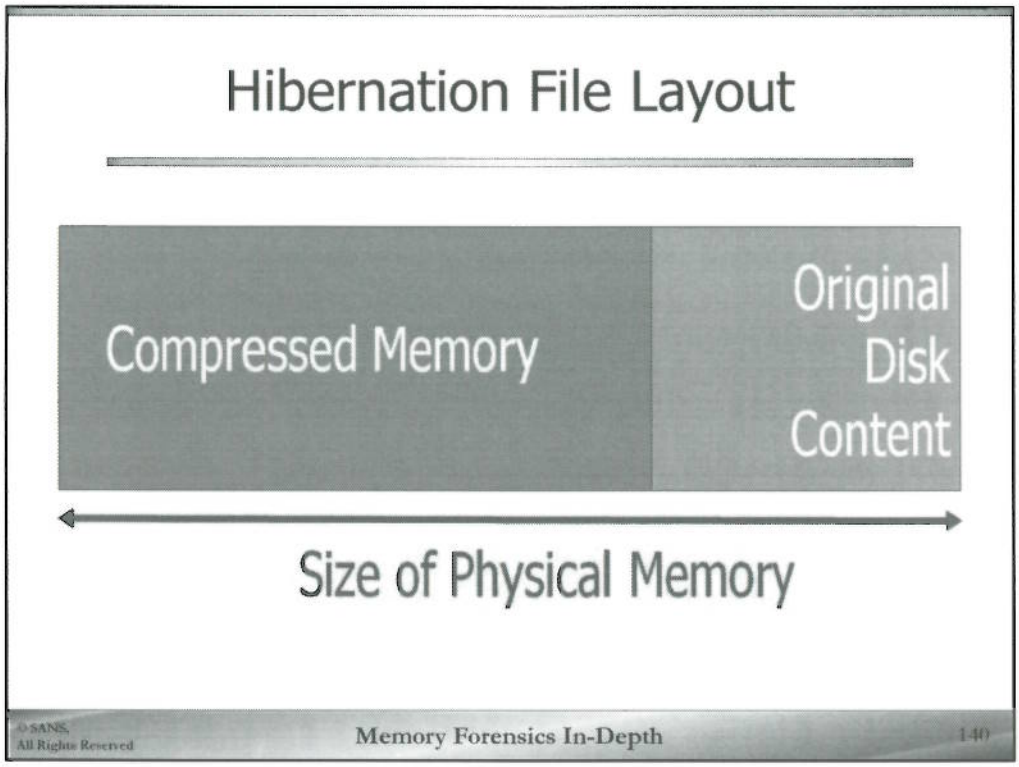


Although introduced with Windows 2000, Microsoft did not enable hibernation by default until Windows Vista. (Computer manufactures, usually laptop makers, would sometimes enable hibernation on the products.) The picture above shows the dialog box, under the Control Panel's power options, for enabling hibernation on Windows XP and Windows 2000.

Hibernation is different from "Sleep". When a Windows operating system is put to sleep, devices like the display and hard drive are shut down to save power. But the state of the system is still kept in memory and the memory is still powered. Windows is able to return from sleep very quickly, usually in just a few seconds. With hibernation, all power is terminated and the system is shutdown. When power is turned on again, the system is booted into a custom boot-loader, which in return restores the same state.

With the introduction of Vista, Microsoft introduced a "hybrid sleep" mode. In such a mode, the computer can enter sleep after a short amount of time, say a few minutes. If the computer is not woken after a longer period time, such as an hour, it is hibernated. The hybrid sleep mode was intended for desktop computers, and may only be enabled by default for such systems, but it can be enabled on any Windows computer.

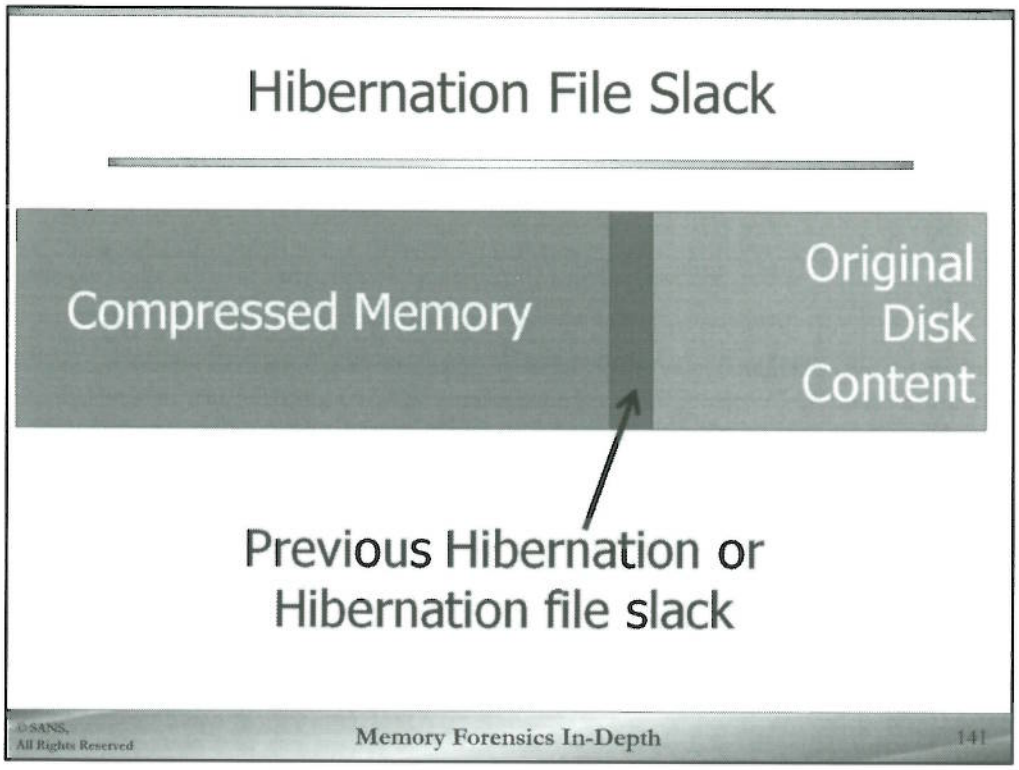
Sleep and hibernation are sometimes referred to using their codes from the Advanced Configuration and Power Interface standard (ACPI). This standard is used to define the power states of modern computers. The state S0 is when the computer is running. Sleep is state S3 and hibernation as S4.



When hibernation is enabled, earlier versions of Windows reserved space on the disk equal to the size of physical memory. As of Windows 7, though, the size of the hibernation file size has been set to 75% of your memory size by default. This file, normally C:\hiberfil.sys, will hold the saved state of the system when it is hibernated. Nothing is written to this file until the system is hibernated the first time. If the system has never been hibernated, you will find the original contents of the disk in the hibernation file.

During normal operation, Windows keeps an open file handle to the hibernation file. The hibernation file, like the paging file, cannot be read by another process while the system is running. It can be read either by mounting the drive on another system, or by parsing the raw filesystem.

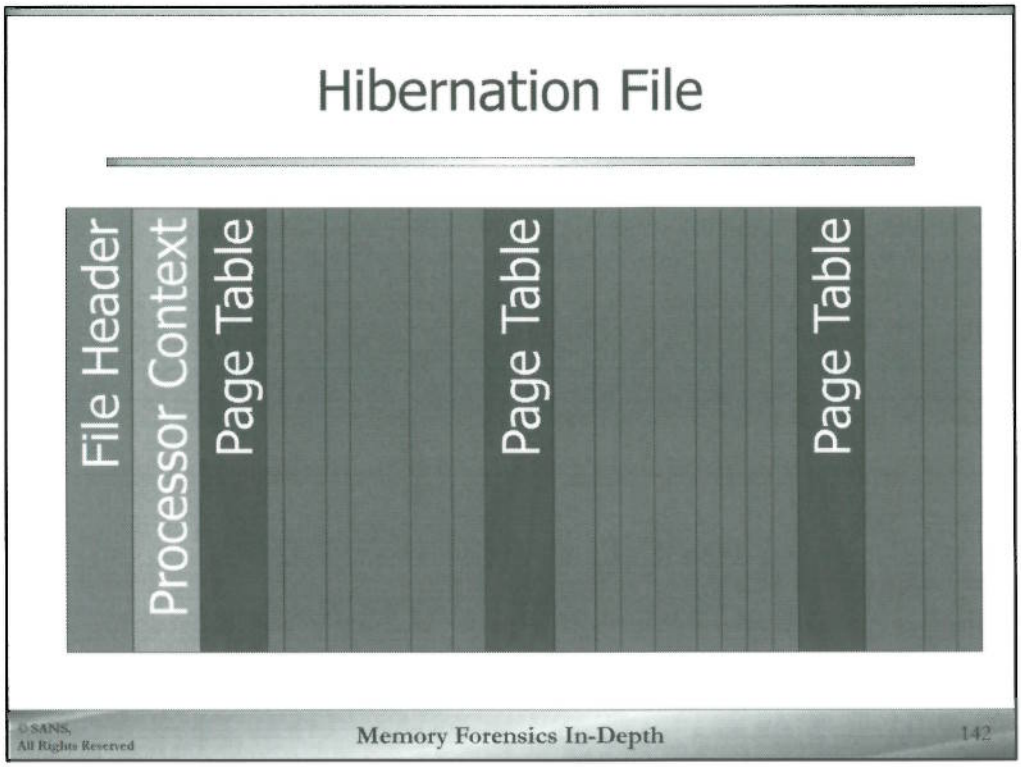
When the system is hibernated, Windows writes the system state to the hibernation file. The file contains a header, the processor state, unpacking information, and the compressed frames of physical memory. Because the frames are compressed, the saved state will almost never take up the entire size of the file. There may still be some of the original disk content at the end of the hibernation file.



It is theoretically possible to have hibernation file slack. That is, data from a previous hibernation which is not overwritten. This data could offer a second look back at the system. Unfortunately such data is not generally useful.

First, hibernation file slack does not occur very often. Hibernation files are usually about the same size, so the amount of leftover data would be small. We would need quite a bit of file data in order to get the table which tells us where in memory such things would go. Finally, we wouldn't have data like the DirectoryTableBase variable, for example, which would allow us to do lookups.

On the other hand, hibernation file slack does exist, can be decompressed, and could contain valuable strings. For this reason, tools like bulk_extractor can and should be used on hibernation files. There could be e-mail addresses, phone numbers, SSNs, you name it in there.

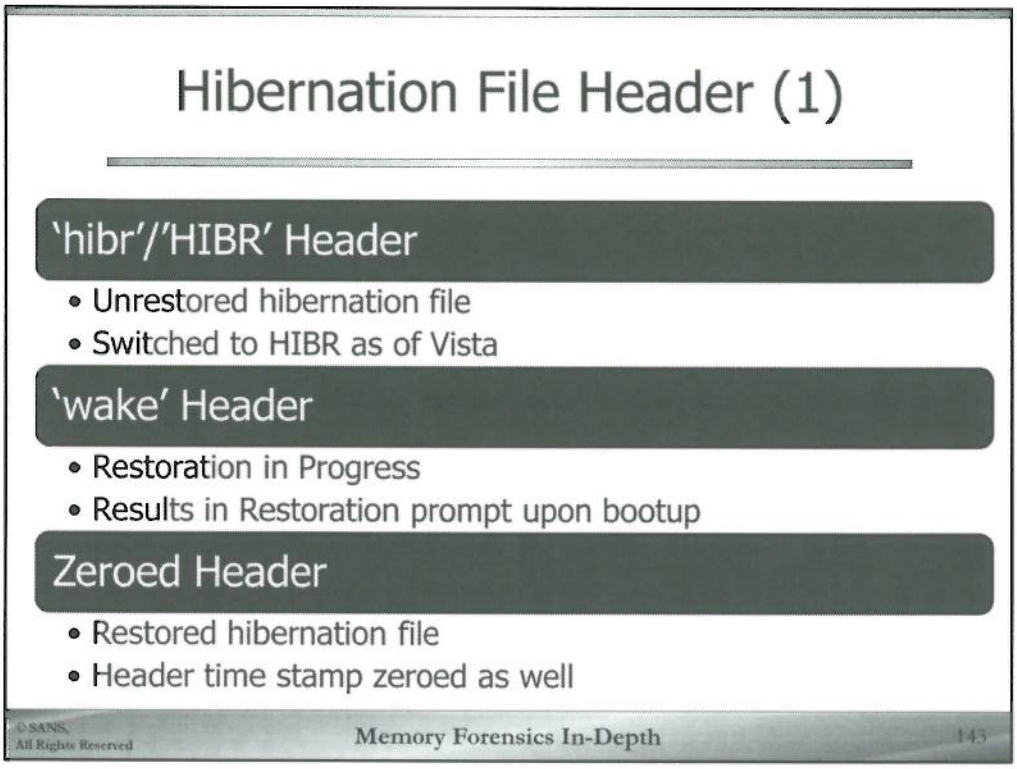


The hibernation file starts with a header which identifies it as a hibernation file and gives some basic information about the file. This includes the date and time the system was hibernated, the size of physical memory, and performance information about the hibernation process. Because the size of the header structure has increased between versions of Windows, we can use the header size to tell us the version of Windows which created it.

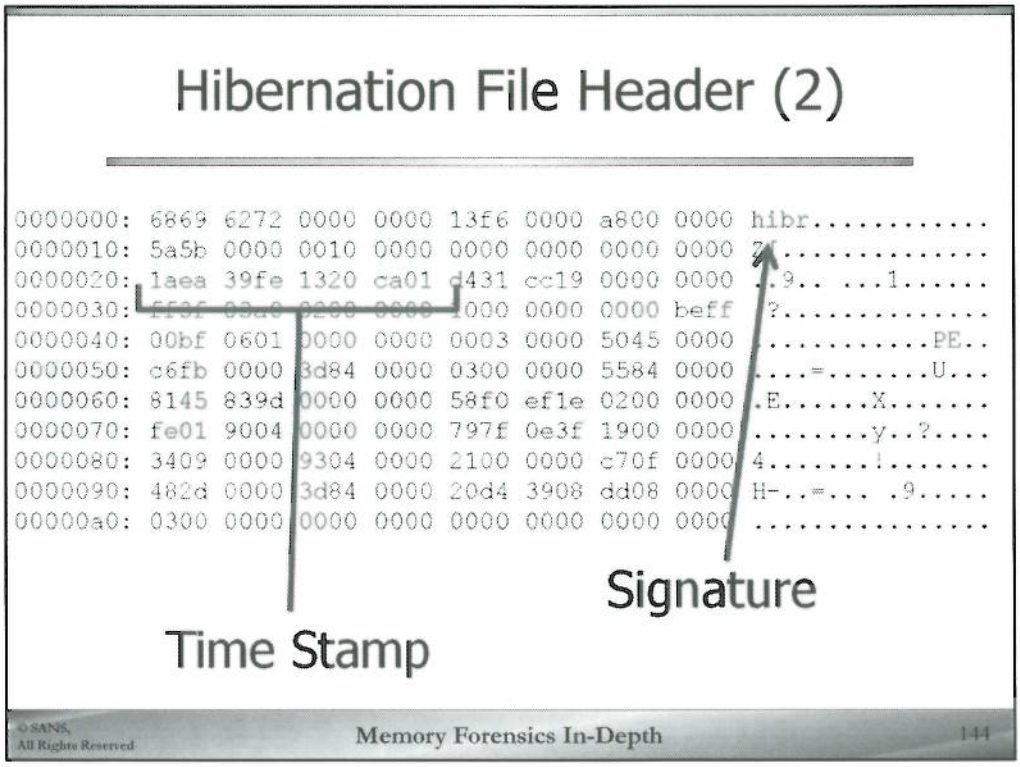
The header is followed by the state of the processor. When Windows resumes execution, it starts exactly where it was before the system was hibernated. This means the very instruction which was next when the system was hibernated is the first to be executed when the system resumes.

Along with the processor state, there is a list of frames which were not being used by the operating system at the time of hibernation. Windows uses these frames to store and execute the code to decompress and restore the rest of the system state. If these frames were not specifically listed, the restored state could overwrite the decompression routines before the decompression was complete.

After the processor state and free frame list comes a series of page tables and compressed data. The page tables describe ranges of memory frames. For each range it gives the starting and ending frames in memory. The tables are then followed by the compressed frames themselves. Windows uses the page tables to determine where each decompressed frame should be stored in physical memory.



This page intentionally left blank.

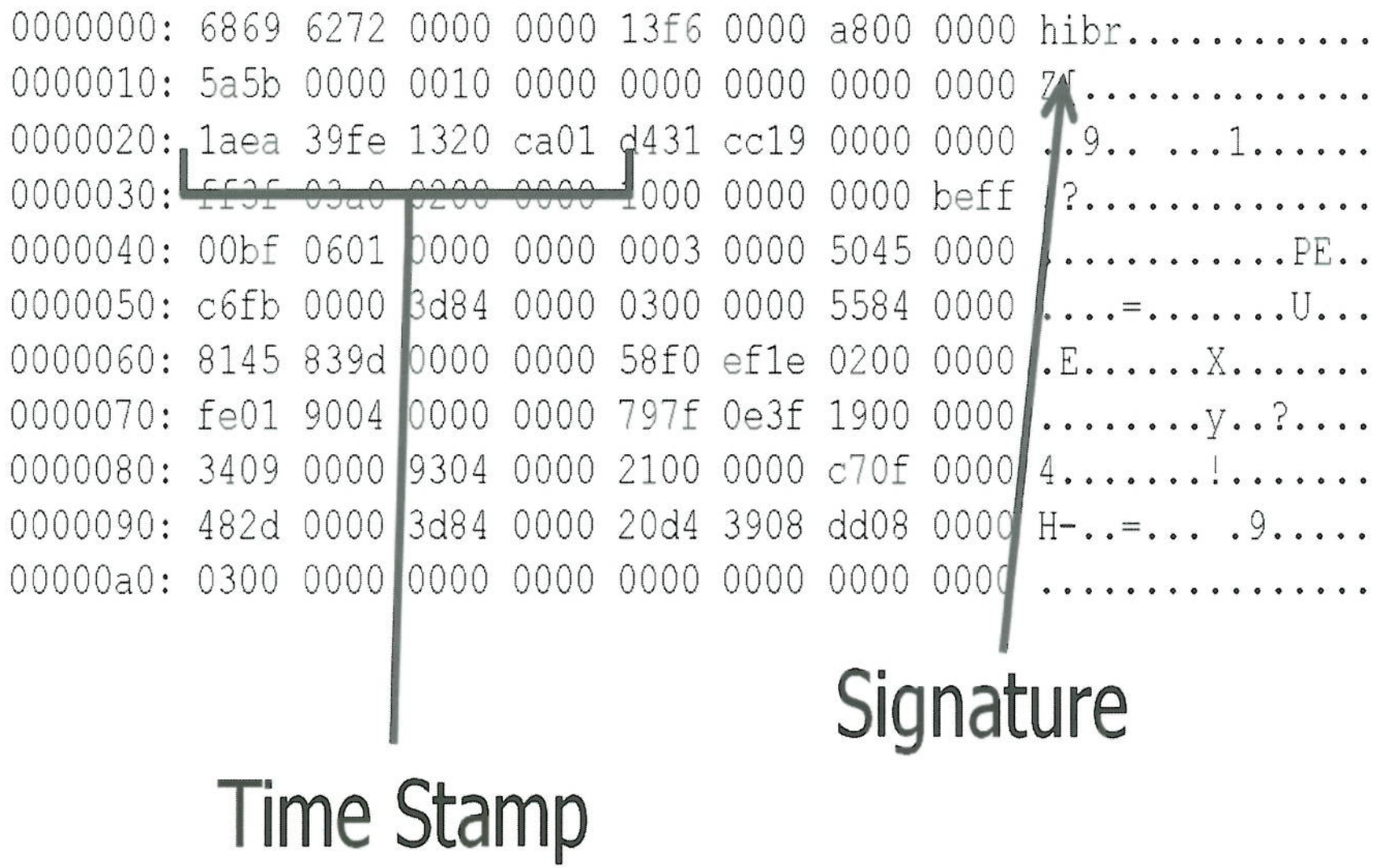


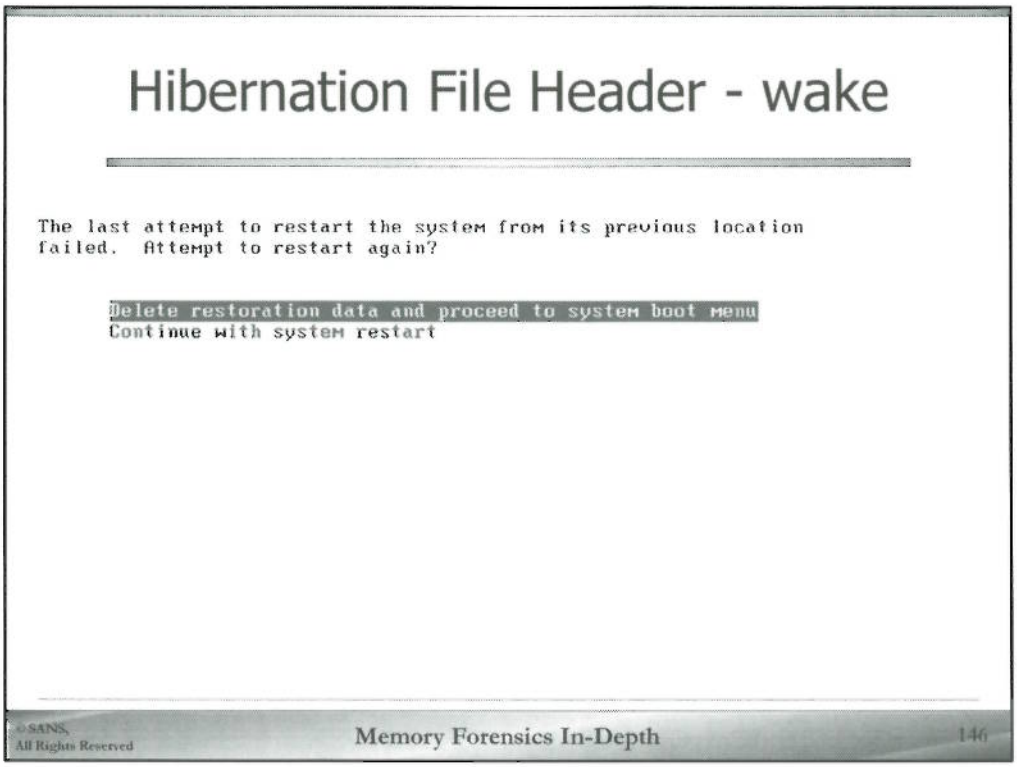
The header of an unrestored hibernation file starts with a four byte signature, hibr or HIBR, depending on the OS. Windows switched to uppercase signatures with Windows Vista.

The picture above shows the header from a Windows XP system. The lowercase signature tells us it came from a pre-Vista system. The size of the header structure, at offset 0xc, is 0xa8 bytes. That's what tells us it came from an XP system and not a Windows 2000 or Server 2003 system.

Note the timestamp at offset 0x20. This was the time, in UTC, when the system was hibernated. (Recognizing Windows timestamps can be a bit of an art. They are eight bytes, and generally end with something like c701 to cb01. Because Windows is a little endian operating system, the bytes are actually the most significant bytes in the value.) In this case the timestamp value ix 0x1ca2013fe39ea1a, or 01:48:37 on 30 Aug 2009.

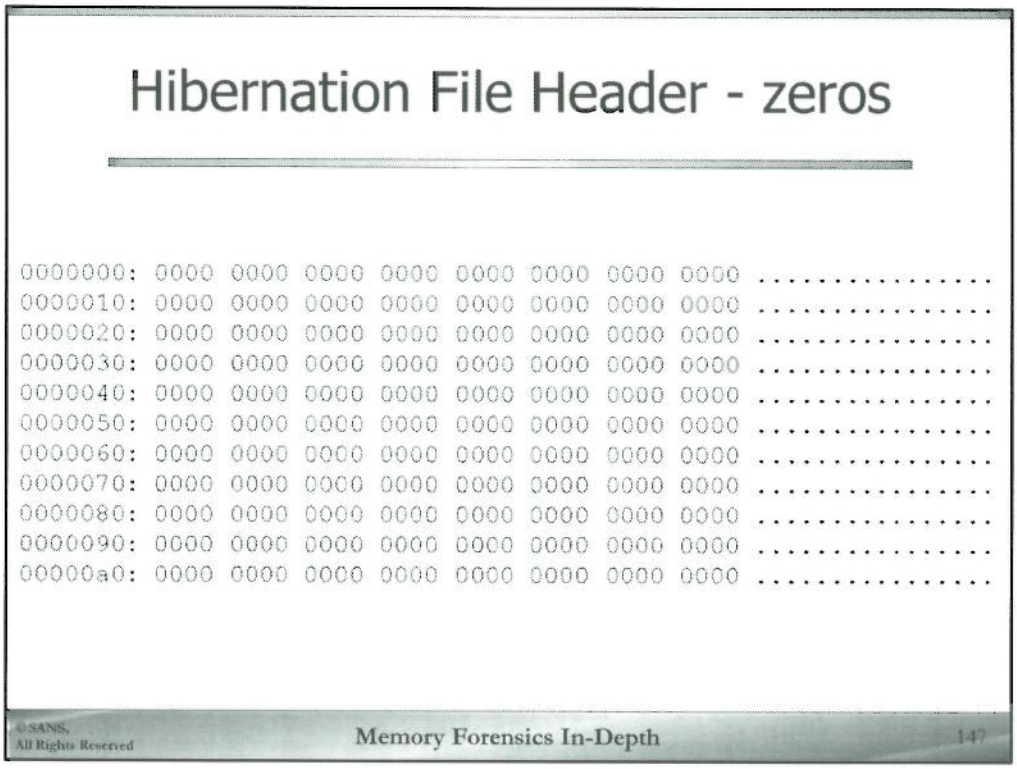
(For some additional details on timestamps and how to recognize them, check out http://blogs.msdn.com/b/oldnewthing/archive/2003/09/05/54806.aspx.)


0000000: 6869 6272 0000 0000 13f6 0000 a800 0000 hibr............
0000010: 5a5b 0000 0010 0000 0000 0000 0000 0000 Z[..............
0000020: 1aea 39fe 1320 ca01 d431 cc19 0000 0000 ..9.. ...1......
0000030: ff3f 03a0 0200 0000 1000 0000 0000 beff .?..............
0000040: 00bf 0601 0000 0000 0003 0000 5045 0000 ....PE..
0000050: c6fb 0000 3d84 0000 0300 0000 5584 0000 ....=.......U...
0000060: 8145 839d 0000 0000 58f0 ef1e 0200 0000 .E......X.......
0000070: fe01 9004 0000 0000 797f 0e3f 1900 0000 ....y..?....
0000080: 3409 0000 9304 0000 2100 0000 c70f 0000 4.......!.......
0000090: 482d 0000 3d84 0000 20d4 3908 dd08 0000 H-..=... .9.....
00000a0: 0300 0000 0000 0000 0000 0000 0000 0000 ....
Time Stamp
Signature



When the system is rebooted, the hibernation file is restored. The frames are decompressed and put back into RAM. The PCR is used to set the state of the processor, and then execution is resumed.

While this is happening, Windows changes the file header from 'hibr' to 'wake' (or 'WAKE'). You can make this happen by pulling the plug (or powering off the VM) during the restore operation. If something goes wrong during the decompression process and the system is rebooted, Windows gives the user a choice about trying the decompression again or just booting the system. The choice is shown in the dialog box above.



Once the hibernation file has been fully restored, Windows wipes the header with zeros. The remainder of the file, including all of the compressed frames of memory, is not altered. With the header zeroed out, Windows does not restore the hibernation file when the system boots. This doesn't mean we can't get anything of value from the hibernation file. There are some missing pointers, such as the size of the processor context record and the location of the start of the compressed memory pages. But with a little brute force searching, we can find the start of the compressed memory pages and recover the memory image manually.

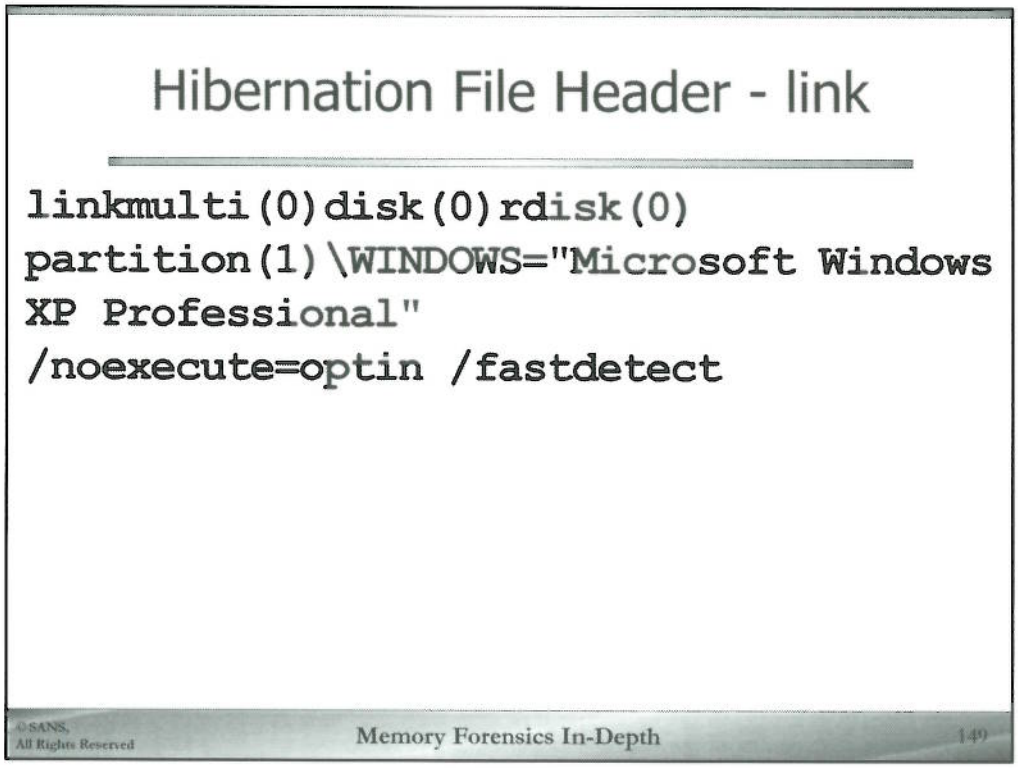
Unfortunately, when the hibernation file is restored we lose the timestamp information from the header. When the header is overwritten, obviously it wipes out the stamp from the file. The change to the file also updates the NTFS timestamp for the hibernation file as well. Don't despair! The system time is still present in the compressed memory image, which is intact. We can determine the time the system was hibernated from that time stamp.

It is theoretically possible to restore a hibernation file more than once. That is, hibernate the system, copy off the hibernation file and paging file, restore the system, do stuff, shut the system down, copy back the hibernation file and paging file, and then attempt to restart the system. At press time, there was no published method to do this. Attempts to do this using a virtual machine resulted in bizarre, undocumented errors.

Windows can be configured to not wipe the header, and in fact restore the same hibernation file again and again, but this is uncommon in consumer devices. The mode to do this is Hibernate Once Restart Many (HORM) mode. This was a feature of Windows XP embedded which allowed the system to be booted multiple times using the same hibernation file. HORM mode depends on the Enhanced Write Filter (EWF) which is not included with other editions of Windows XP. There are methods to enable HORM on Windows XP, but they are cumbersome at best. The best indicator of HORM mode is the presence of a file resmany.dat in the root directory.

For more on HORM, see http://blogs.msdn.com/b/embedded/archive/2006/11/03/what-is-horm-and-how-can-you-use-it.aspx and http://msdn.microsoft.com/en-us/library/ms932932%28WinEmbedded.5%29.aspx.

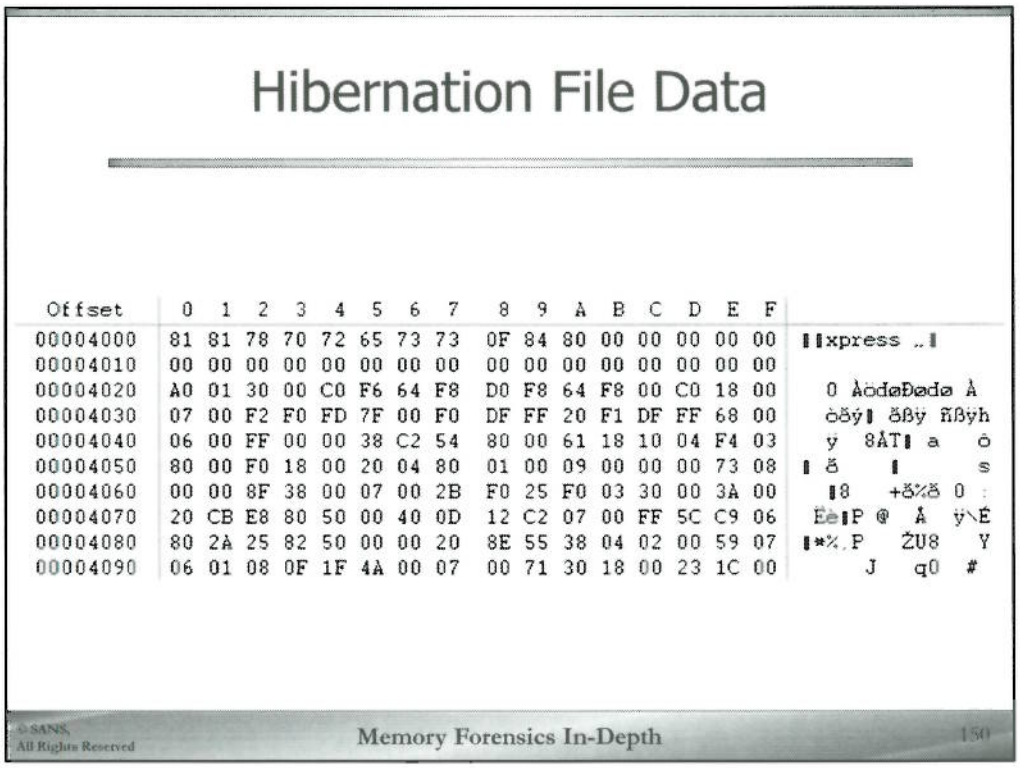

0000000: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000010: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000020: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000030: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000040: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000050: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000060: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000070: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000080: 0000 0000 0000 0000 0000 0000 0000 0000 ....
0000090: 0000 0000 0000 0000 0000 0000 0000 0000 ....
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000 ....



Finally, there is a fourth valid value for the signature in the hibernation file header. The value "link" means the header contains a pointer to another hibernation file to be used. The value is stored as an ARC path. There is no space between the signature and path, meaning the hibernation file could start like:

linkmulti(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect

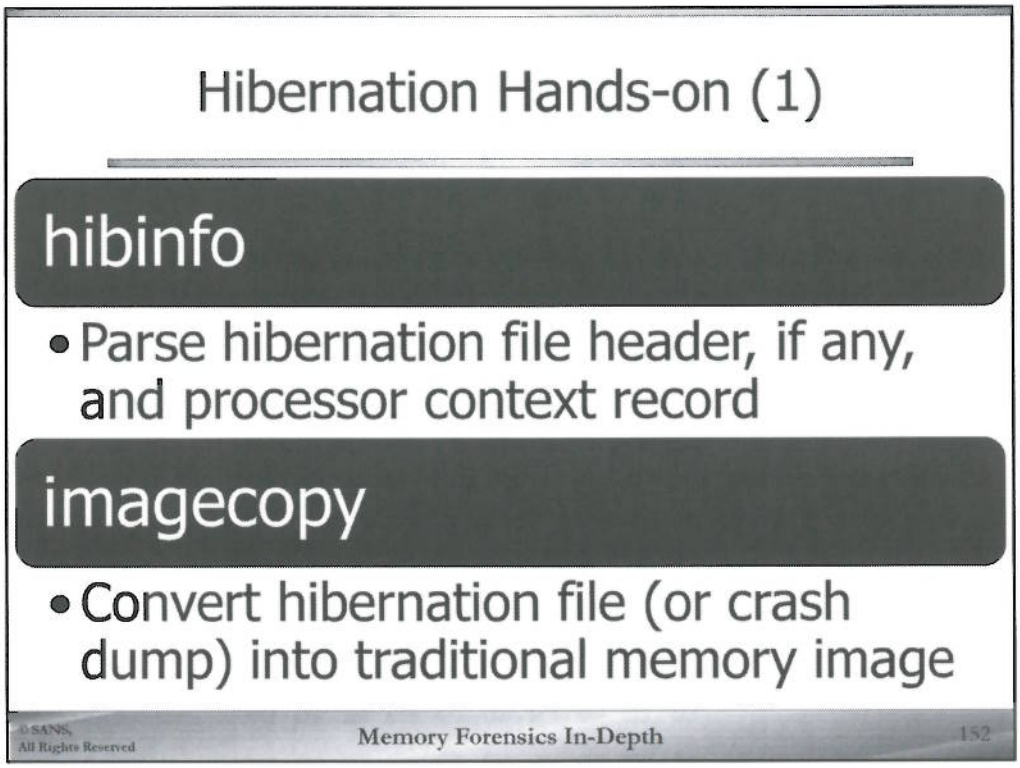
For more details on the link header, see http://blogs.msdn.com/b/ntdebugging/archive/2007/06/28/how-windows-starts-up-part-the-second.aspx.



The compressed memory frames are stored in blocks, each of which begins with an "XPRESS" header. The picture above shows a sample block of compressed data. You can see that the exact header is, in hex, 81 81 78 70 72 65 73 73, or \x81\x81xpress.

The data compression algorithm is referred to as xpress, but has never been formally documented by Microsoft. The algorithm has, however, been reverse engineered. The original published version was by Matthieu Suiche and part of a tool he wrote called Sandman. Soon after that publication, X-Ways, a forensic software vendor, introduced the capability to decompress hibernation files into one of their products. Suiche objected, on the grounds that X-Ways had used his code without permission. As proof, Suiche pointed to the X-Ways binary, which contained Suiche's birthdate in hexadecimal. It turns out Suiche had embedded the value in a vestigial part of his code. When X-Ways copied it, they inadvertently copied the birth date too!

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00004000	81	81	78	70	72	65	73	73	0F	84	80	00	00	00	00	00	▮▮xpress „▮
00004010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00004020	A0	01	30	00	C0	F6	64	F8	D0	F8	64	F8	00	C0	18	00	0 ÀödøÐødø À
00004030	07	00	F2	F0	FD	7F	00	F0	DF	FF	20	F1	DF	FF	68	00	òðý▮ ðßÿ ñßÿh
00004040	06	00	FF	00	00	38	C2	54	80	00	61	18	10	04	F4	03	ÿ 8ÂT▮ a ô
00004050	80	00	F0	18	00	20	04	80	01	00	09	00	00	00	73	08	▮ ð ▮ s
00004060	00	00	8F	38	00	07	00	2B	F0	25	F0	03	30	00	3A	00	▮8 +ð%ð 0 :
00004070	20	CB	E8	80	50	00	40	0D	12	C2	07	00	FF	5C	C9	06	Ëè▮P @ Â ÿ\É
00004080	80	2A	25	82	50	00	00	20	8E	55	38	04	02	00	59	07	▮*%‚P ŽU8 Y
00004090	06	01	08	0F	1F	4A	00	07	00	71	30	18	00	23	1C	00	J q0 #

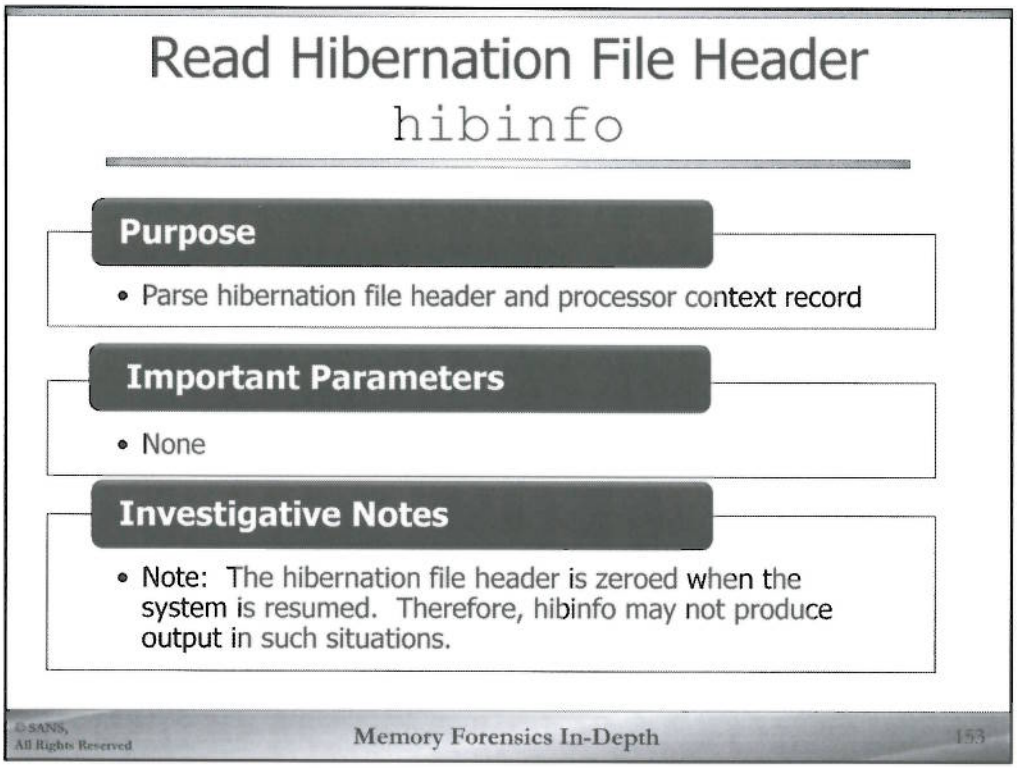


The hibernation file can be used directly by Volatility. In fact, you can use a hibernation file in Volatility just like any other memory image. Volatility uses what the authors call stackable address layers to do its processing. The hibernation decompression routines are just another layer in the process. When decompressing, however, the framework will be much slower. It has to decompress each block before it can be read. But the process is completely transparent to the user.

You could, in theory, convert a memory image to a hibernation file, but it wouldn't have a valid processor context record. To restore a hibernation file you must know the state of the processor exactly.

Volatility has two plugins for dealing with hibernation files directly. The first is hibinfo, which attempts to parse the hibernation file header and dump out meaningful information. The second is the imagecopy plugin, which can decompress a hibernation file back to a traditional memory image.

Remember when using any Volatility plugin you must still specify a profile. With hibernation files, it's often much faster to get the profile information from the hibinfo plugin rather than the imageinfo plugin.

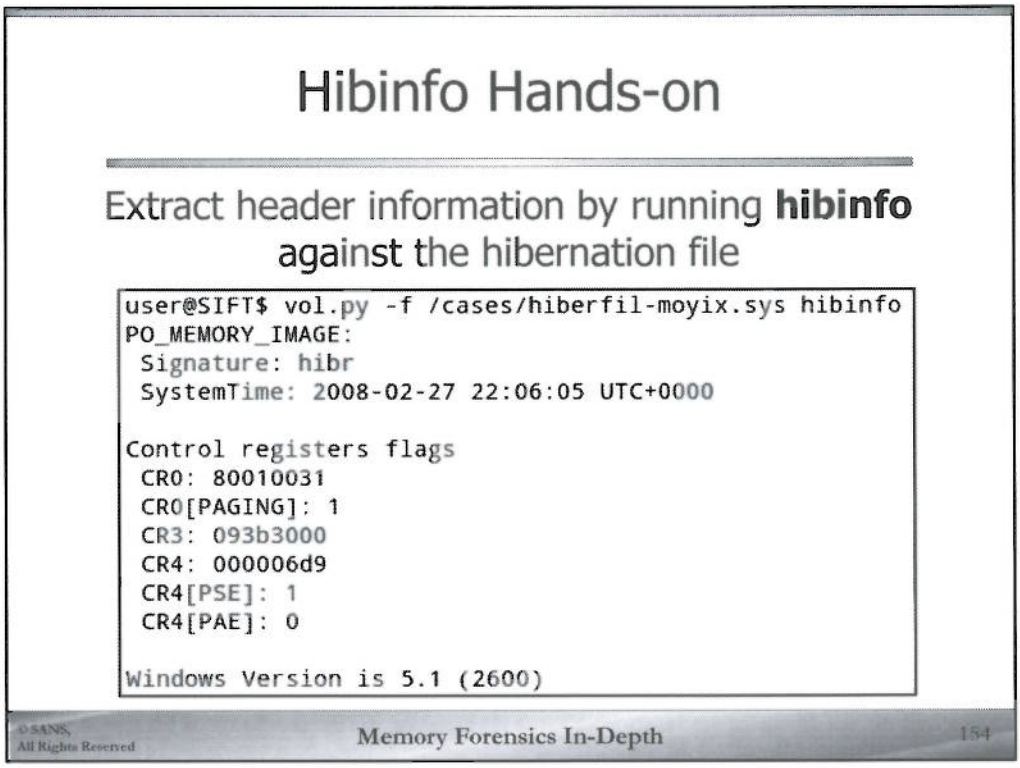


The hibernation file can be used directly by Volatility. In fact, you can use a hibernation file in Volatility just like any other memory image. Volatility uses what the authors call stackable address layers to do its processing. The hibernation decompression routines are just another layer in the process. When decompressing, however, the framework will be much slower. It has to decompress each block before it can be read. But the process is completely transparent to the user.

You could, in theory, convert a memory image to a hibernation file, but it wouldn't have a valid processor context record. To restore a hibernation file you must know the state of the processor exactly.

Volatility has two plugins for dealing with hibernation files directly. The first is hibinfo, which attempts to parse the hibernation file header and dump out meaningful information. The second is the imagecopy plugin, which can decompress a hibernation file back to a traditional memory image.

Remember when using any Volatility plugin you must still specify a profile. With hibernation files, it's often much faster to get the profile information from the hibinfo plugin rather than the imageinfo plugin.



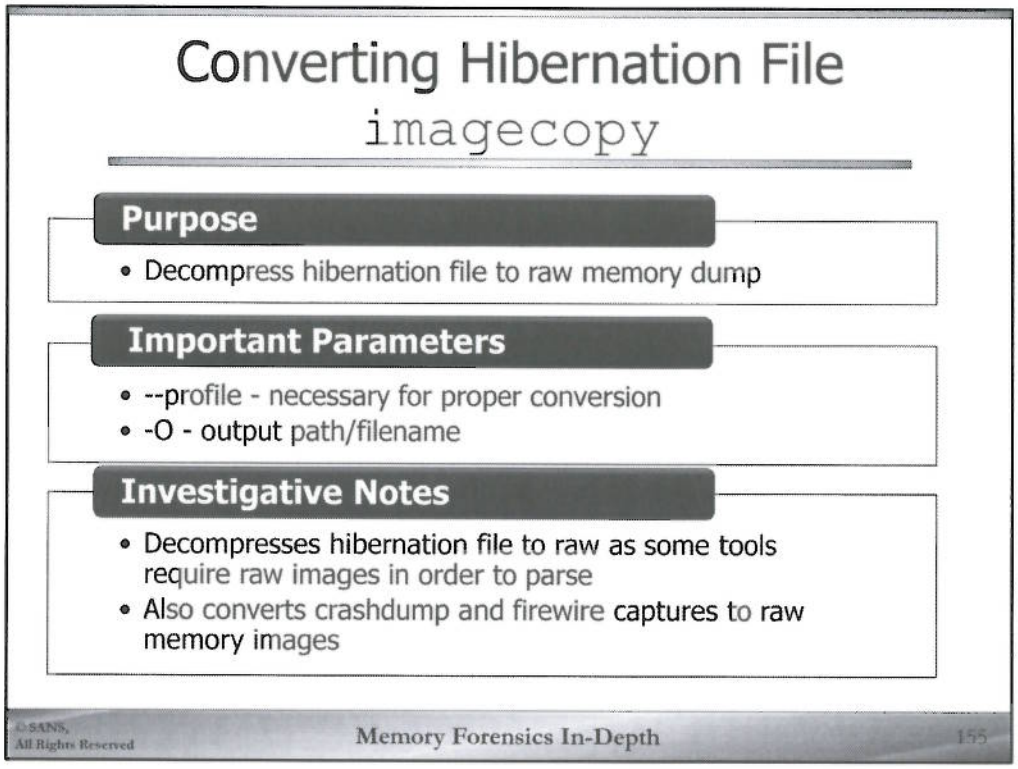
Let's try the Volatility hibernation file commands together. First, we'll start with hibinfo. We're going to use this plugin on a hibernation file to get out the basic information we'll need to decompress it. Here's the command line you'll want to use:

```
$ vol.py -f /cases/hiberfil-moyix.sys hibinfo
```

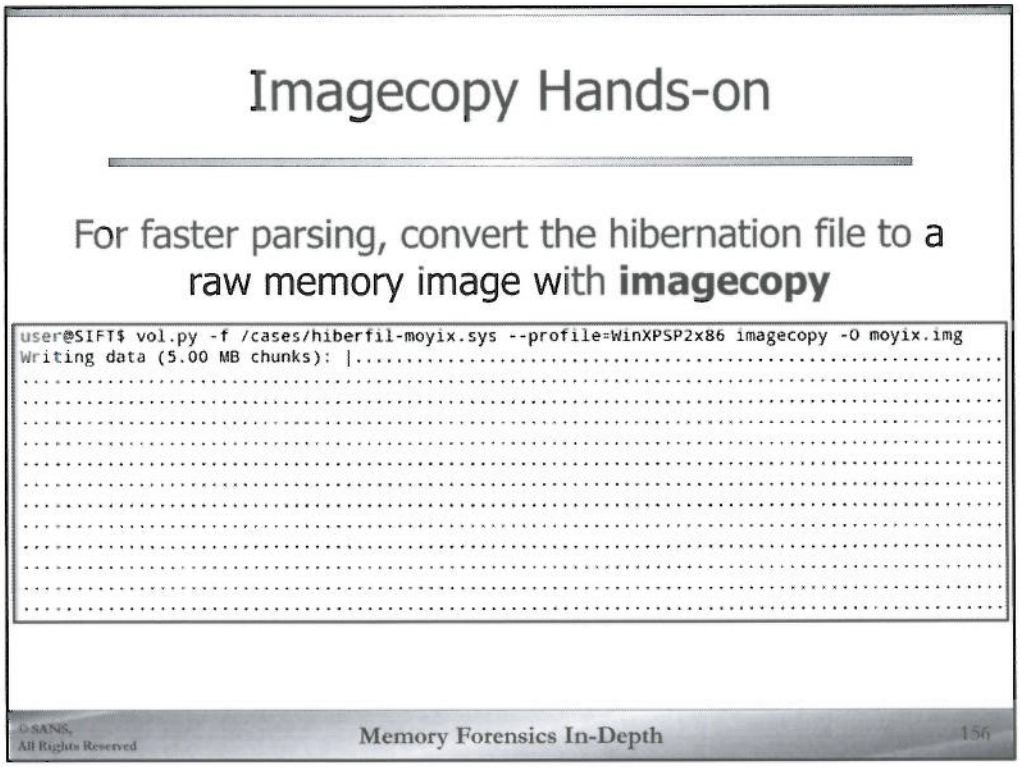
You should see the following:

```
Volatile Systems Volatility Framework 2.3.1
PO_MEMORY_IMAGE:
 Signature: hibr
 SystemTime: 2008-02-27 22:06:05 UTC+0000

Control registers flags
 CR0: 80010031
 CR0[PAGING]: 1
 CR3: 093b3000
 CR4: 000006d9
 CR4[PSE]: 1
 CR4[PAE]: 0

Windows Version is 5.1 (2600)
```

We can see the hibernation file signature, hibr, and the date and time the system was hibernated. At the bottom, we also get the Windows version which created this file. There are many references for the major, minor, and build numbers, including Wikipedia, http://en.wikipedia.org/wiki/Windows_NT#Releases. Looking up the build 2600 in the table we see that this image came from a Windows XP system.



This page intentionally left blank.



At this point we could run any Volatility command we wanted on this hibernation file. There is a gotcha, however. Because Volatility needs to know how to decompress this file, we must specify a profile before running any commands. For example, if you run the imageinfo plugin without a profile, it takes about an hour to execute. Adding the profile flag, however, produces a faster result, but still far too long to demonstrate in this course. You can always get the profile by running the imageinfo plugin on the hibernation file.

```
$ vol.py -f /cases/hiberfil-moyix.sys imageinfo
```

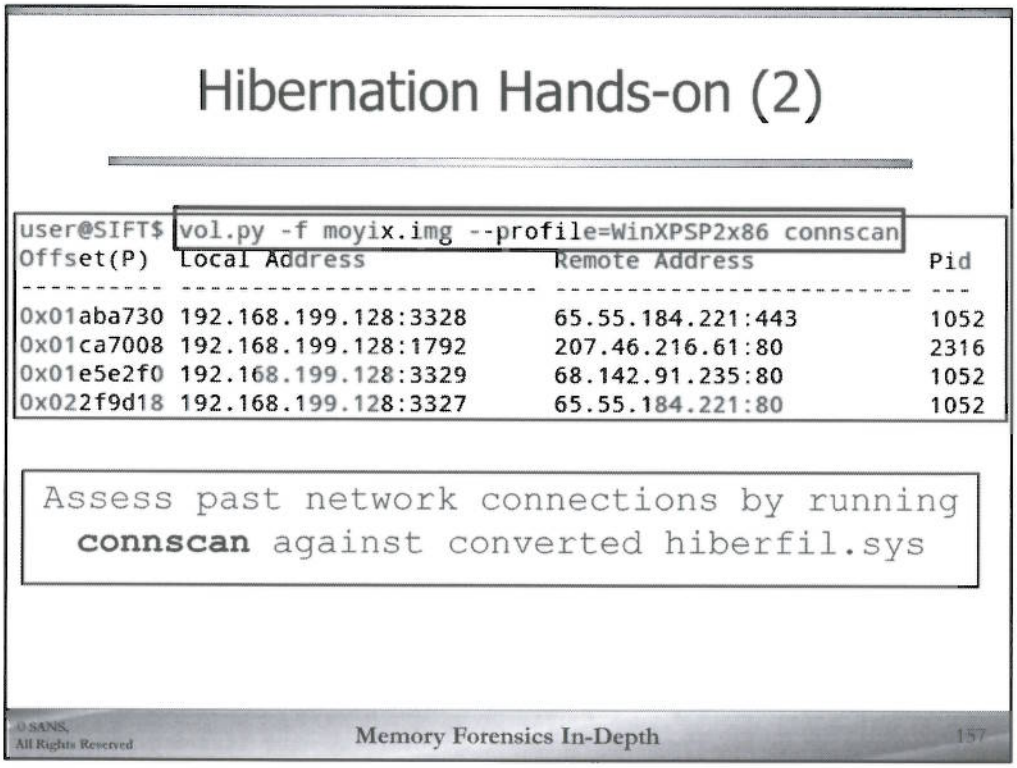
Instead, we can copy out the decompressed memory image to a new file using the imagecopy plugin. We need to specify an output file to write the decompressed file to using the -O or --output-image flag. Here's the full command line you'll want to execute:

```
$ vol.py -f /cases/hiberfil-moyix.sys --output-image= moyix.img --
profile=WinXPSP2x86 imagecopy
```

You will see a lot of progress dots before the process completes. You may also want to grab a cup of coffee. Once it does complete, you can run Volatility plugins on the decompressed file.

For example, try running the pstree plugin on the decompressed memory image.

```
$ vol.py -f /cases/moyix.img --profile=WinXPSP2x86 pstree
```



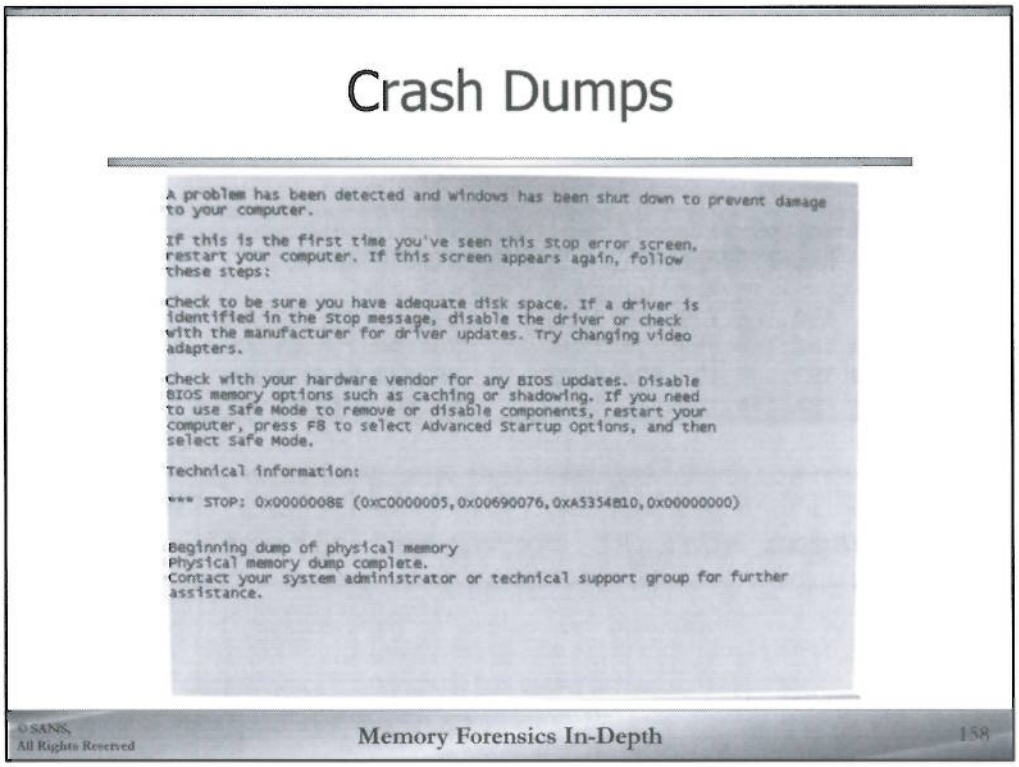
And finally, the real magic. Let's see who this computer was talking to back then. Let's do a scan for connection objects.

There were no active connections. You can verify this by running the **connections** plugin.)

```
$ vol.py -f /cases/moyix.img --profile=WinXPSP2x86 connscan
```

```
 Offset     Local Address             Remote Address            Pid
---------- ------------------------- ------------------------- ------
0x01aba730 192.168.199.128:3328      65.55.184.221:443          1052
0x01ca7008 192.168.199.128:1792      207.46.216.61:80           2316
0x01e5e2f0 192.168.199.128:3329      68.142.91.235:80           1052
0x022f9d18 192.168.199.128:3327      65.55.184.221:80           1052
```

Don't forget that you can get the full content of these network communications with bulk extractor!



Crash dumps were created for debugging Windows drivers. When the operating system crashes, it can write out a crash dump file. This file lists what went wrong, details about the problem, and what it was trying to do at the time. The result can be used by developers to diagnose the problem. (Hopefully fix it too!) Some of these details are also displayed in the infamous Blue Screen of Death. Operating system crashes should occur only from bugs in kernel drivers or other code running at the kernel level. When a user program crashes, it *shouldn't* take down the operating system. Due to compromises Microsoft made when writing Windows, however, for performance reasons there is a lot of user mode functionality which is included in the kernel. For example, the vulnerability which Microsoft addressed in MS11-087, and which was subsequently used by the Duqu malware, involved the processing of fonts in kernel mode. Although an operating system function, certainly, such processing should probably be done in user mode.

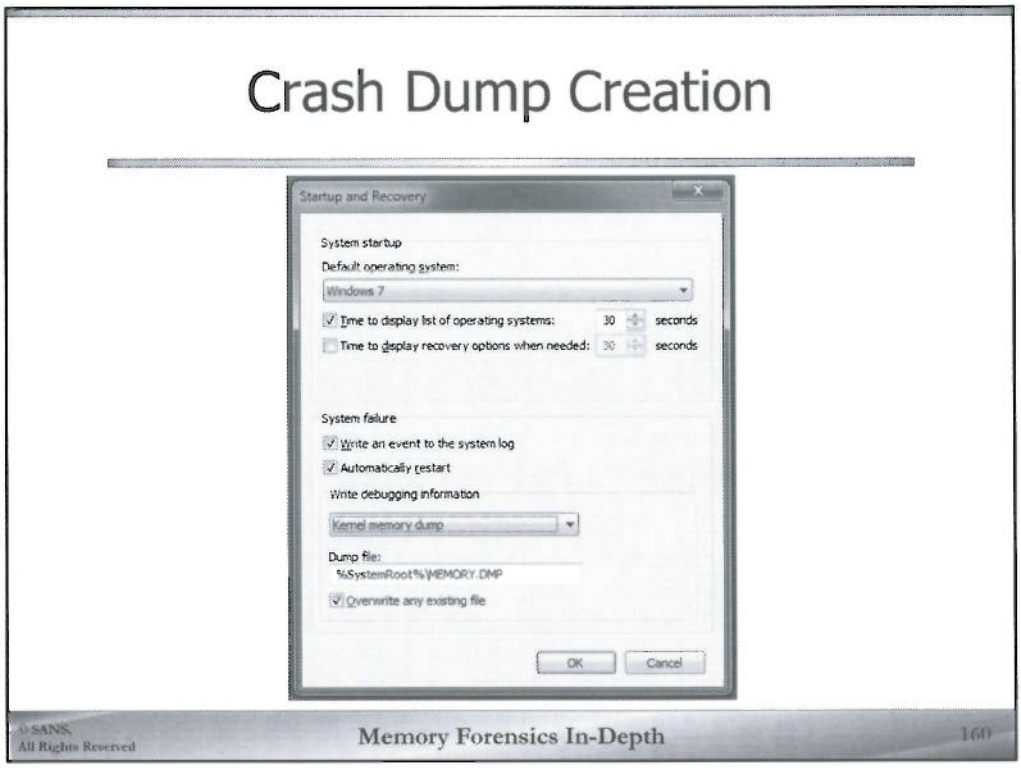
Crash dumps can be valuable, especially when looking at the systems belonging to malware authors. Malware usually requires a driver, and drivers need to be debugged. During that debugging process, it's quite likely the developer crashed their own system. If they did so, the code of their driver, perhaps in an earlier, less hardened state, should be in the crash dump file!

We are now going to discuss what is a crash dump file, what information can be gleaned from it, and how you can use them in your investigations.

Windows crashes when it encounters a condition which it can't handle. These exceptional situations may come from the processor or software running in kernel mode. When this happens, Windows doesn't know what to do and so punts. The operating system displays a screen affectionately known as the Blue Screen of Death (BSOD). The BSOD includes a stop code or bug check code, and sometimes a description of what the stop code means. In the example shown above, the code 0x8e refers to an unhandled exception. There are several references on the web with lists of stop codes. For example, here's one from Microsoft: http://msdn.microsoft.com/en-us/library/aa126132.aspx.

Along with the stop code, Windows provides some data related to the crash, which can be useful in fixing it. In the example above you can see four values after the stop code. These are parameters of the crash.

You can also see, at the bottom of the image, some messages about how Windows has written out physical memory. This is the creation of the crash dump file.

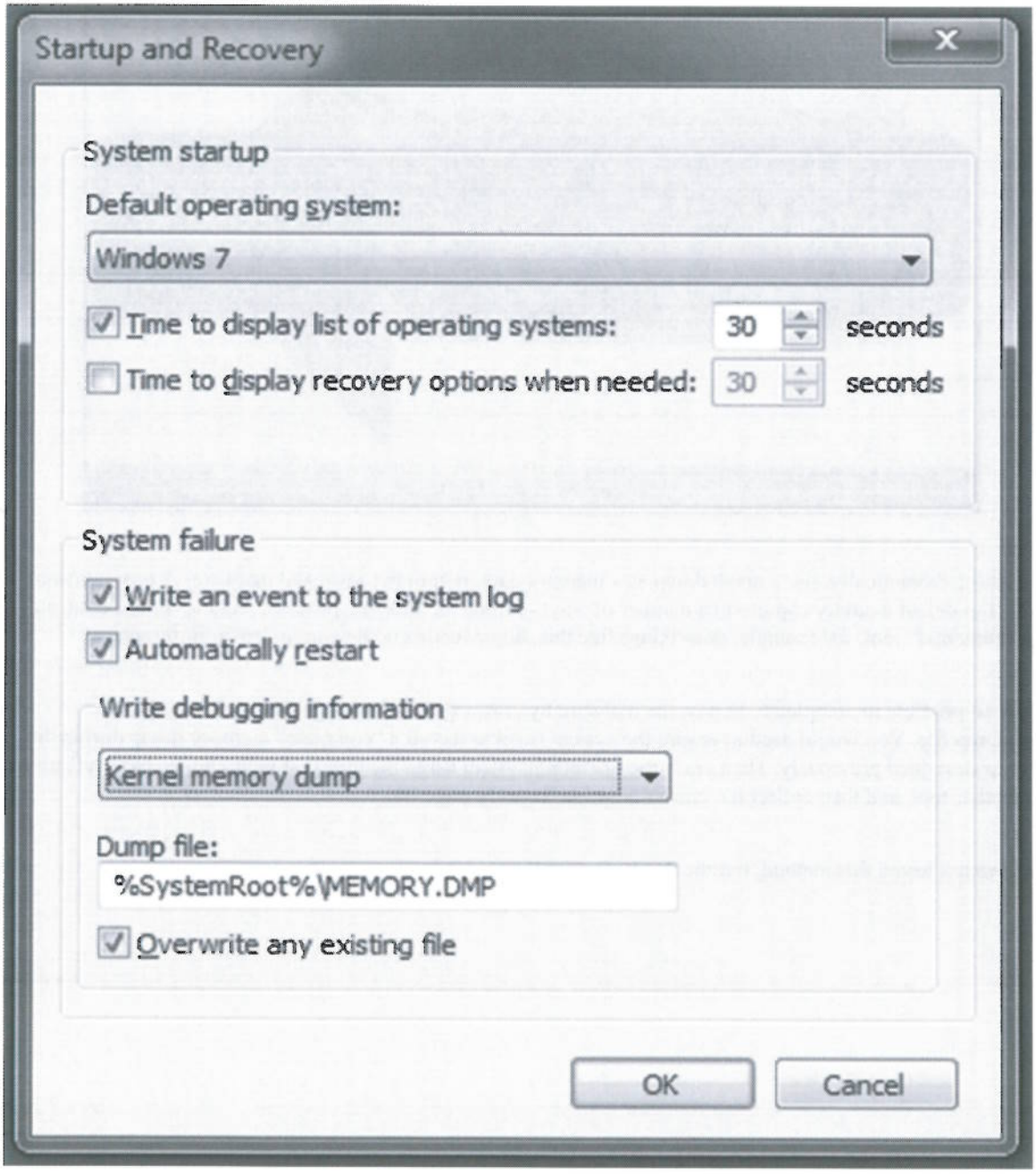


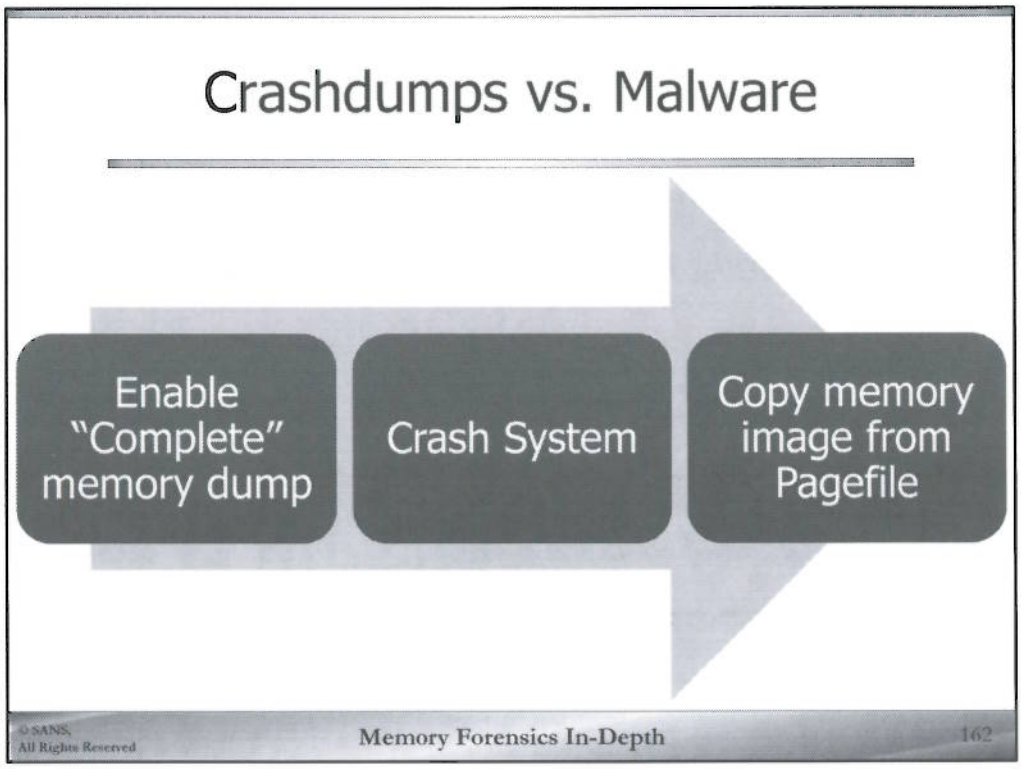
Crash dumps can be small, kernel sized, or complete. That is, they can hold the details of the crash and the memory space of the module which crashed, the memory of the entire kernel, or the complete memory of the computer.

By default, Windows has crash dumps enabled. This is good. By default, however, Windows is configured to create kernel crash dumps. This is not the ideal situation. If you can, please encourage the subjects of your investigation to enable full crash dumps. This is done from My Computer->System and Security->System->Advanced Settings->Startup and Recovery. The dialog box you should see is shown above. Users can also get to this screen from the Control Panel.

The actual crash dump file is not created at the time of the crash. When the operating system crashes, it is in an unstable state. It would be unwise to attempt to open a new file handle in such a state, and so the OS overwrites a file to which it already has an open file handle. Specifically, the OS overwrites the paging file with the data which will become the crash dump. Generally in a crash, whatever caused the crash should be in main memory, not the paging file. So for debugging it makes sense to overwrite the paging file. (Yes, the crash could have been caused because the operating system needed something which was in the paging file, but couldn't access it. In such a case it doesn't so much matter what was needed, as that something was needed.)

The next time the machine is booted, the crash dump information is copied from the paging file to a configurable location. The default location is %SystemRoot%\MEMORY.DMP, or C:\Windows\MEMORY.DMP. This next boot may appear to take a while. Be patient! There may be a lot of data to copy.

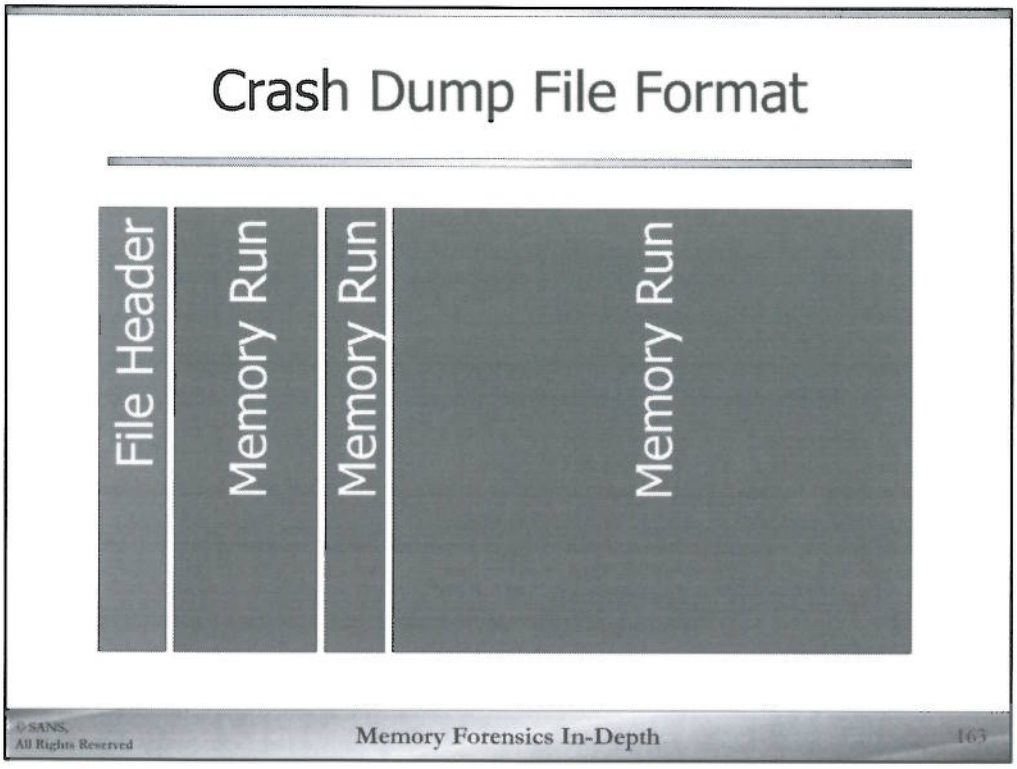

Startup and Recovery
System startup
Default operating system:
Windows 7
Time to display list of operating systems:
30
seconds
Time to display recovery options when needed:
30
seconds
System failure
Write an event to the system log
Automatically restart
Write debugging information
Kernel memory dump
Dump file:
%SystemRoot%\MEMORY.DMP
Overwrite any existing file
OK
Cancel



You could, theoretically, use a crash dump as a memory capture tool for advanced malware. A program could attempt to defeat memory capture in a number of ways--killing the imaging process, feeding it false data, etc. The "Dimentia" tool, for example, does things like this, https://code.google.com/p/dementia-forensics/.

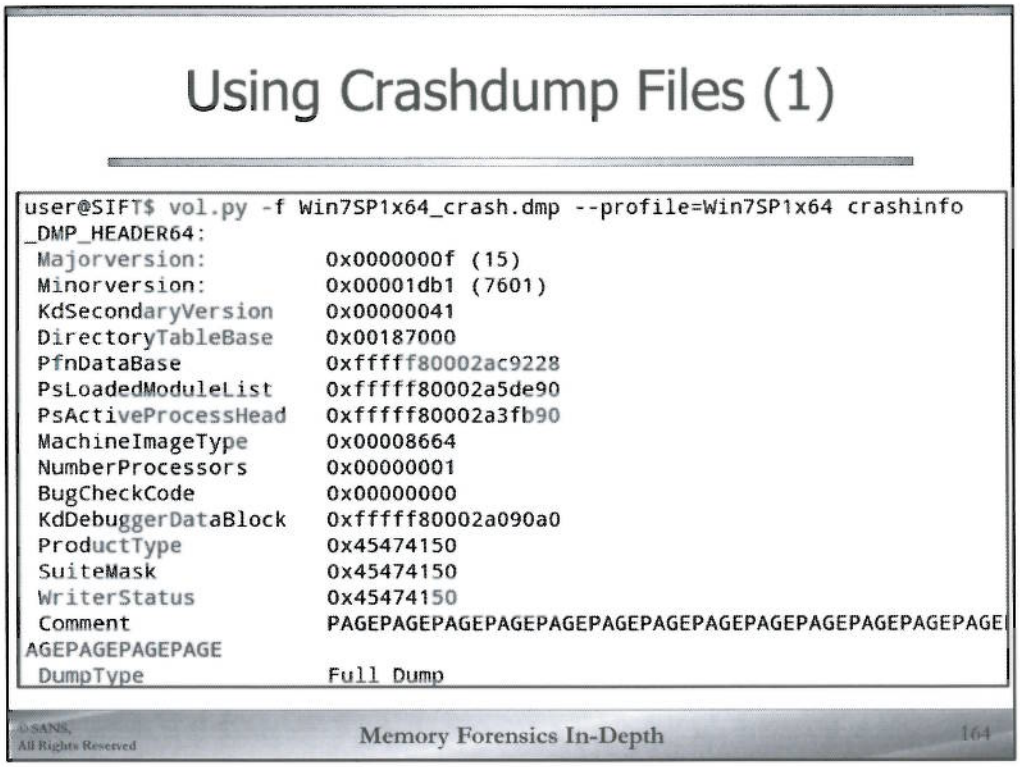
It may be possible to completely bypass the malware by crashing the operating system and using the crashdump file. You would need to ensure the system is set to record a "complete" memory dump during the crash as described previously. Then crash the operating system using the utility of your choice, NotMyFault or some other tool, and then collect the crash dump file from the page file.

We have not tested this method, but the theory is sound!



Crash dump files, like hibernation files, have a header. The header points to a series of memory "runs", much like an NTFS filestream. The header details how much data is in each run, where in the crash dump file it lives, and where it existed in memory. In the example shown above, there were three memory runs which follow the header.

The header begins with a signature. On 32-bit systems the signature is PAGEDUMP, and on 64-bit systems it's PAGEDU64. The signature is followed by fields which describe the operating systems' major and minor versions, the virtual addresses of some kernel variables (e.g., DirectoryTableBase, MmPfnDatabase, PsActiveProcessHead, etc.), and the bug check code and parameters.

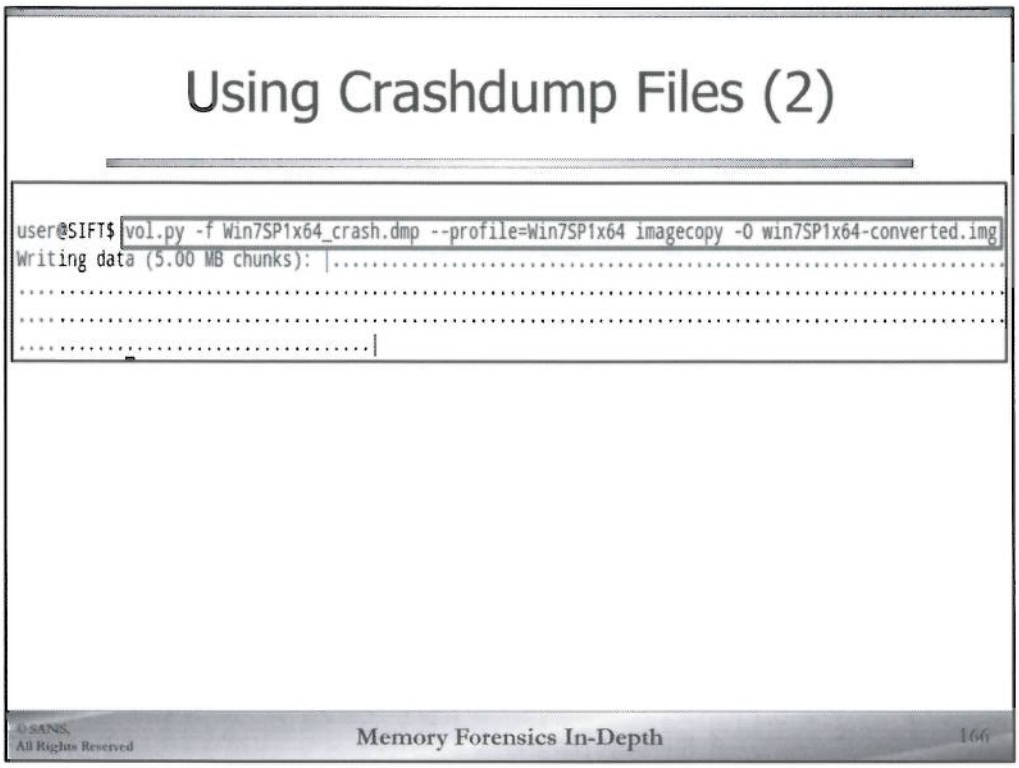


Like hibernation files, the Volatility framework has a plugin for gleaning information from crashdump files. The crashinfo plugin displays information gleaned from the crash dump header. Unfortunately the crashinfo plugin in the SANS version of Volatility has some bugs and doesn't always work correctly. Here's a sample of crashinfo running, using the trunk version of the code. You don't need to execute this line, it's just an example.

```
user@SIFT$ vol.py -f Win7SP1x64_crash.dmp --profile=Win7SP1x64 crashinfo
_DMP_HEADER64:
 Majorversion:          0x0000000f (15)
 Minorversion:          0x00001db1 (7601)
 KdSecondaryVersion     0x00000041
 DirectoryTableBase     0x00187000
 PfnDataBase            0xfffff80002ac9228
 PsLoadedModuleList     0xfffff80002a5de90
 PsActiveProcessHead    0xfffff80002a3fb90
 MachineImageType       0x00008664
 NumberProcessors       0x00000001
 BugCheckCode           0x00000000
 KdDebuggerDataBlock    0xfffff80002a090a0
 ProductType            0x45474150
 SuiteMask              0x45474150
 WriterStatus           0x45474150
 Comment
```

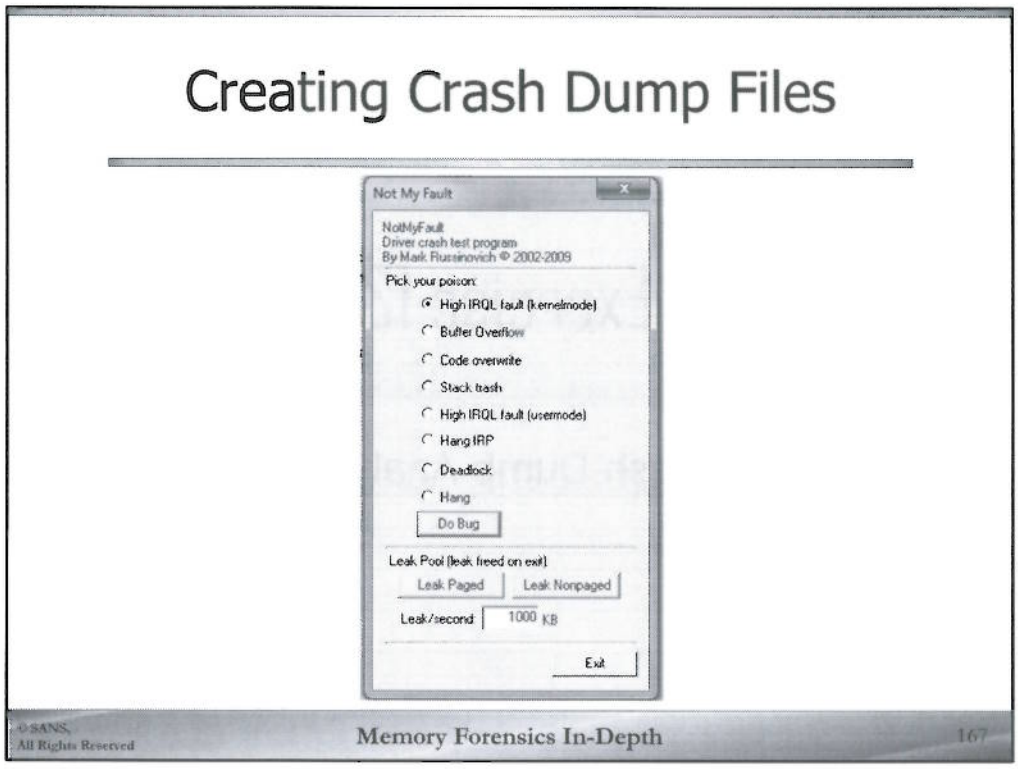
```
PAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPA
GEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGEPAGE
 DumpType              Full Dump
 SystemTime            -
 SystemUpTime

Physical Memory Description:
Number of runs: 3
FileOffset    Start Address    Length
00002000      00001000         0009e000
000a0000      00100000         7fde0000
7fe80000      7ff00000         00100000
7ff7f000      7ffff000
```

Here we can see the values from the header, including some kernel variables. At the bottom of the file we see three memory runs. For each we get the offset in the file where the run begins, the offset in physical memory where the run begins, and the size of the run.

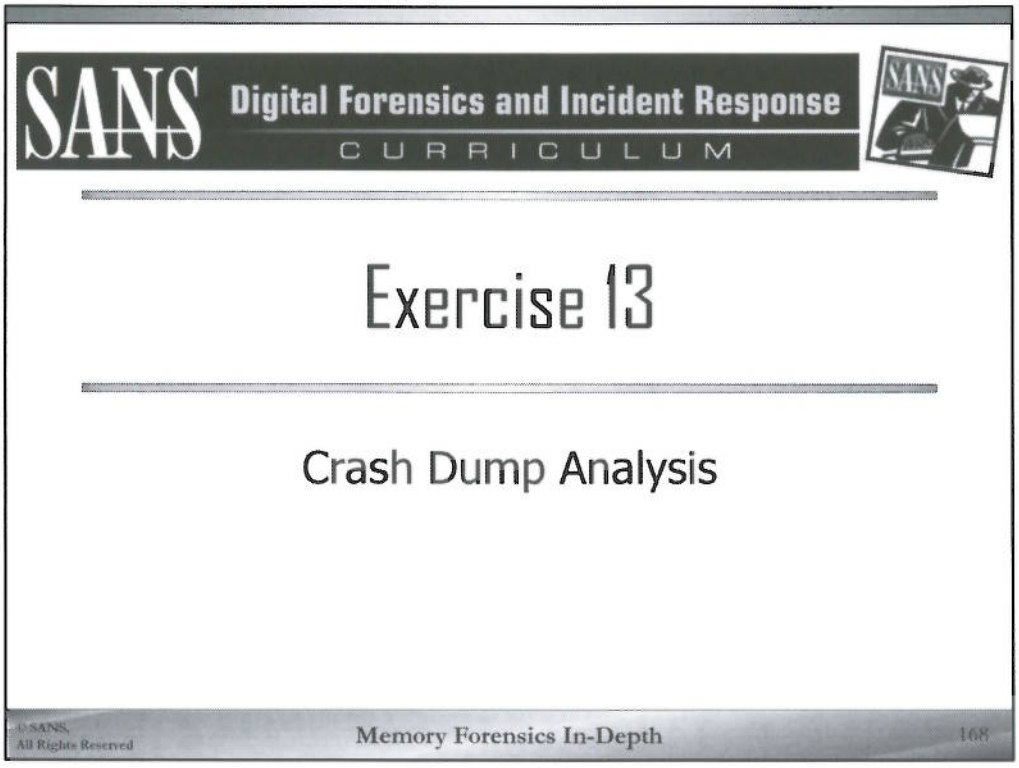


In addition to extracting metadata from the crashdump files, the Volatility framework can process crash dump files just like any other kind of memory image. As with hibernation files, such processing operations are slow. You will probably be better off converting a crash dump file to a flat memory image first. Again, you can use the imagecopy plugin to covert a crash dump file to a flat memory image. Here's an example of how you would make that conversion. You do not have to execute this line.

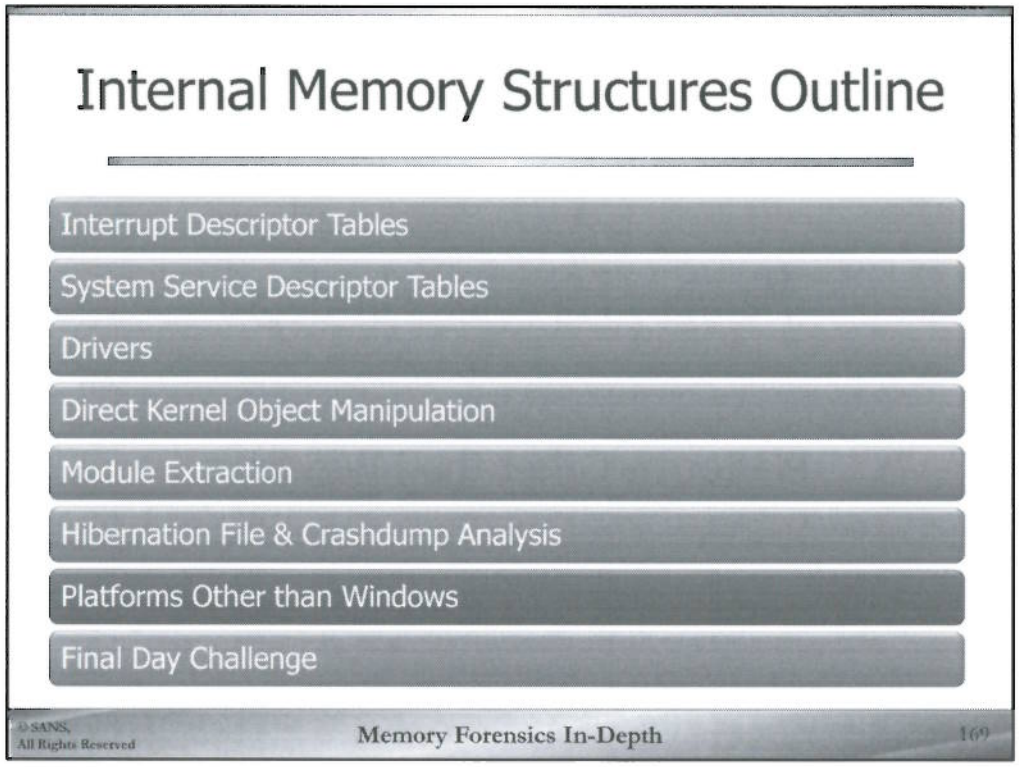
user@SIFT$ vol.py -f Win7SP1x64_crash.dmp --profile=Win7SP1x64 imagecopy -O win7SP1x64-converted.img



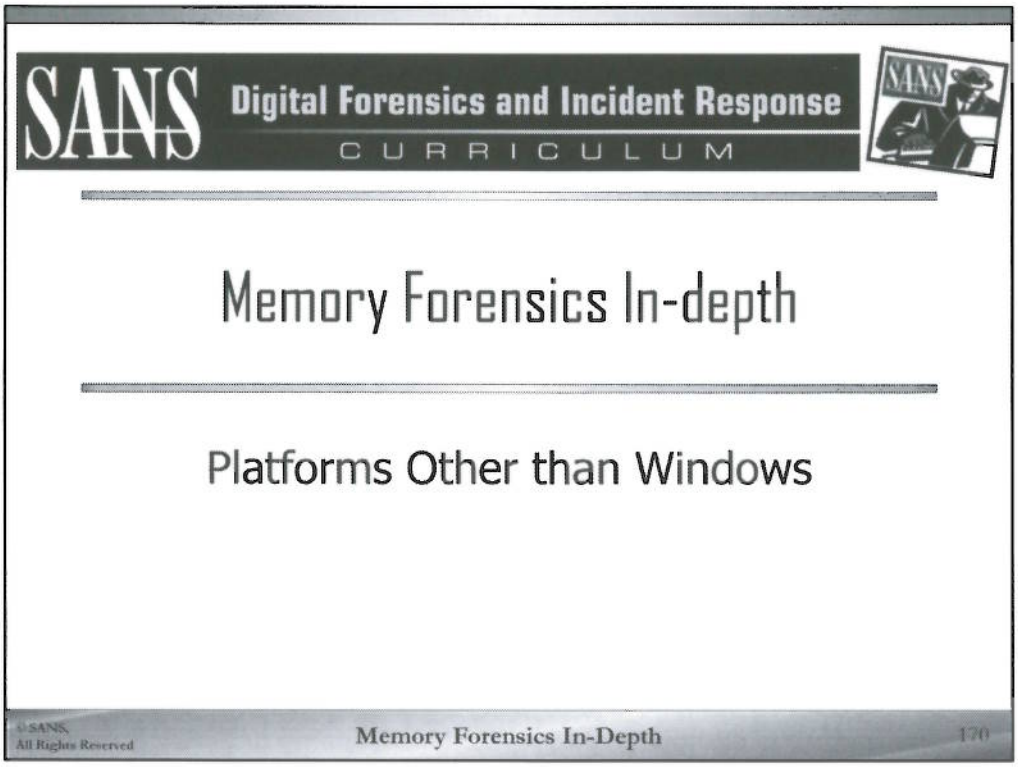
If you'd like to experiment with creating and analyzing crash dumps, you will need a driver which can reliably crash the system. Although you could certainly write your own, Mark Russinovich of Microsoft has written one for you. His tool is called "Not My Fault", and you can get a copy at http://download.sysinternals.com/files/notmyfault.zip. The program installs a driver which you can use to create a variety of bug check codes. Working with the created files is a good way to practice dealing with crash dump files. Remember, you must let the system reboot after crashing in order to copy the crash dump data from the paging file to the MEMORY.DMP file. If you don't, try looking at the paging file for the PAGEDUMP or PAGEDU64 header!



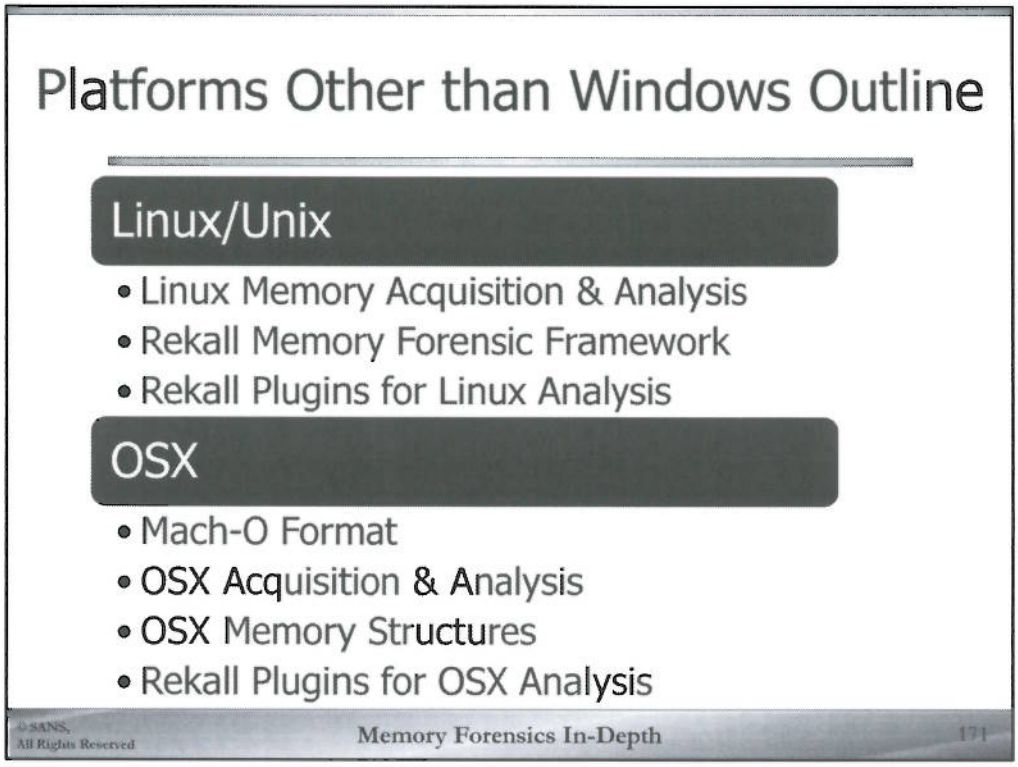
This page intentionally left blank.



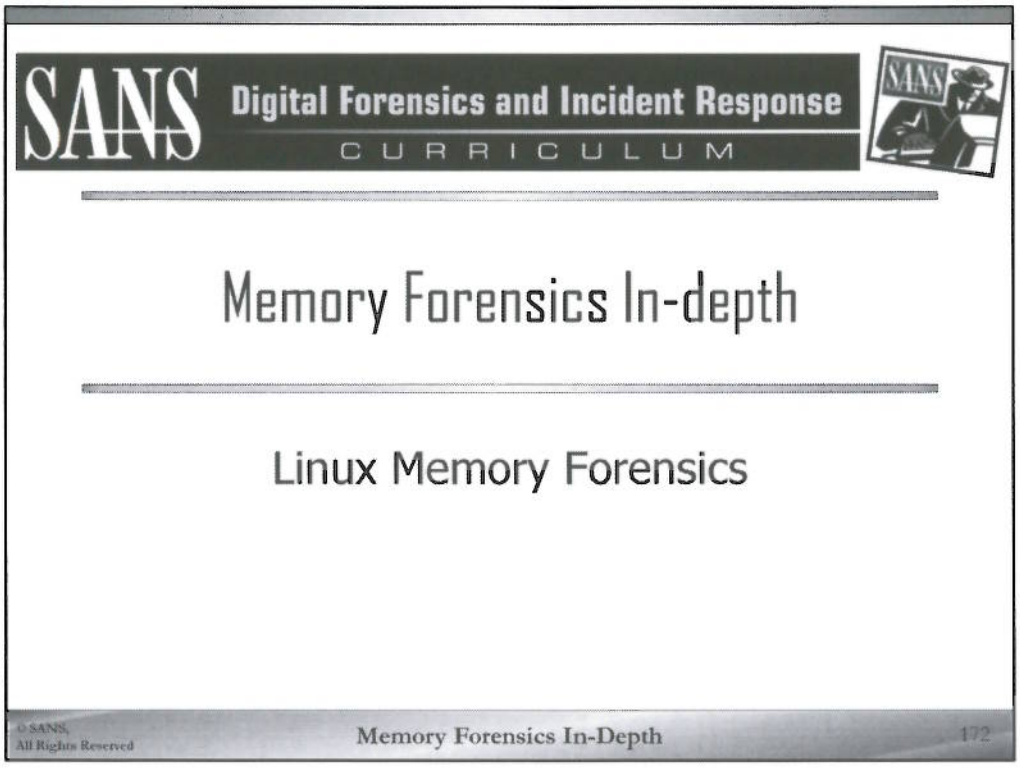
This page intentionally left blank.



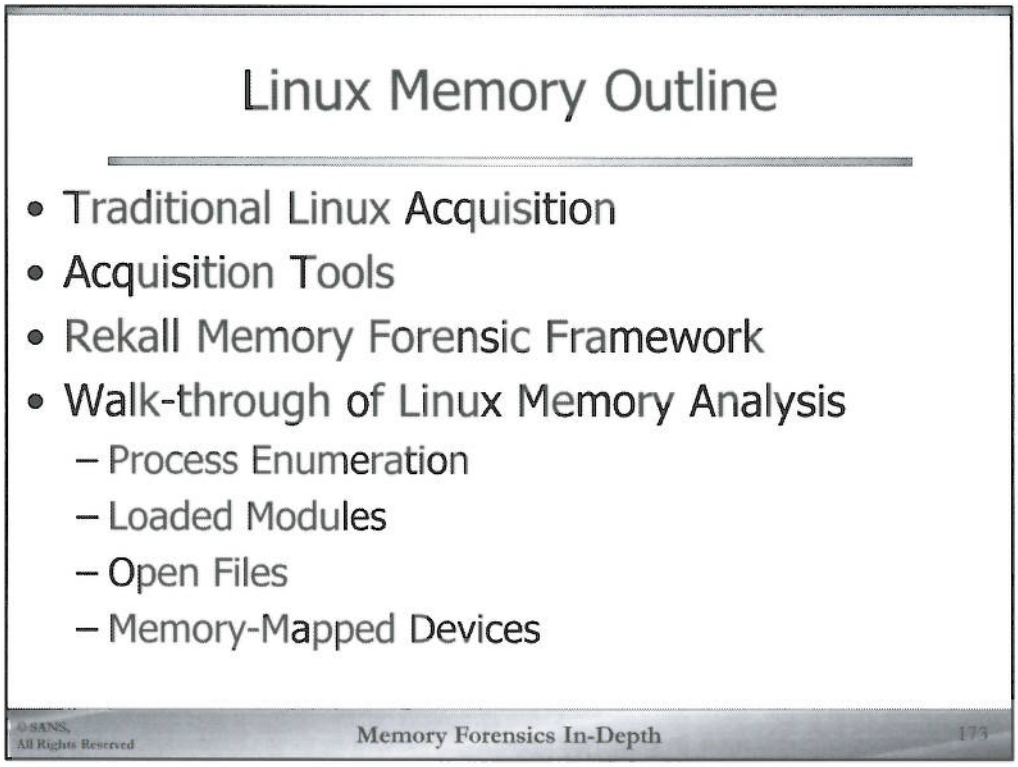
This page intentionally left blank.



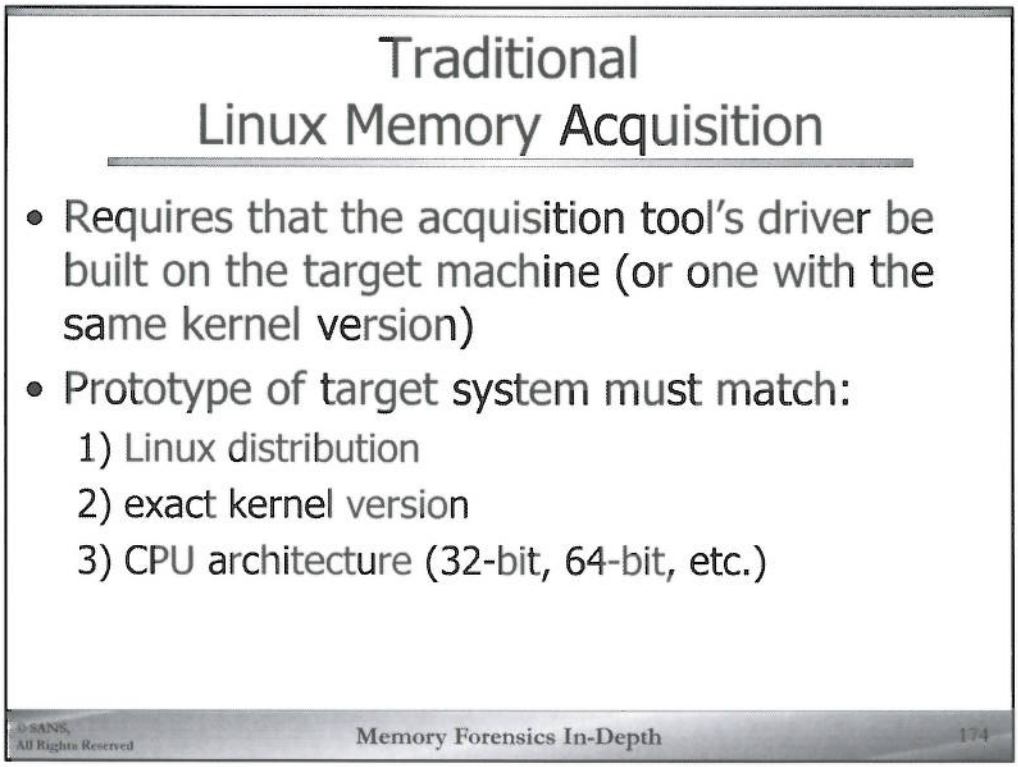
This page intentionally left blank.



This page intentionally left blank.



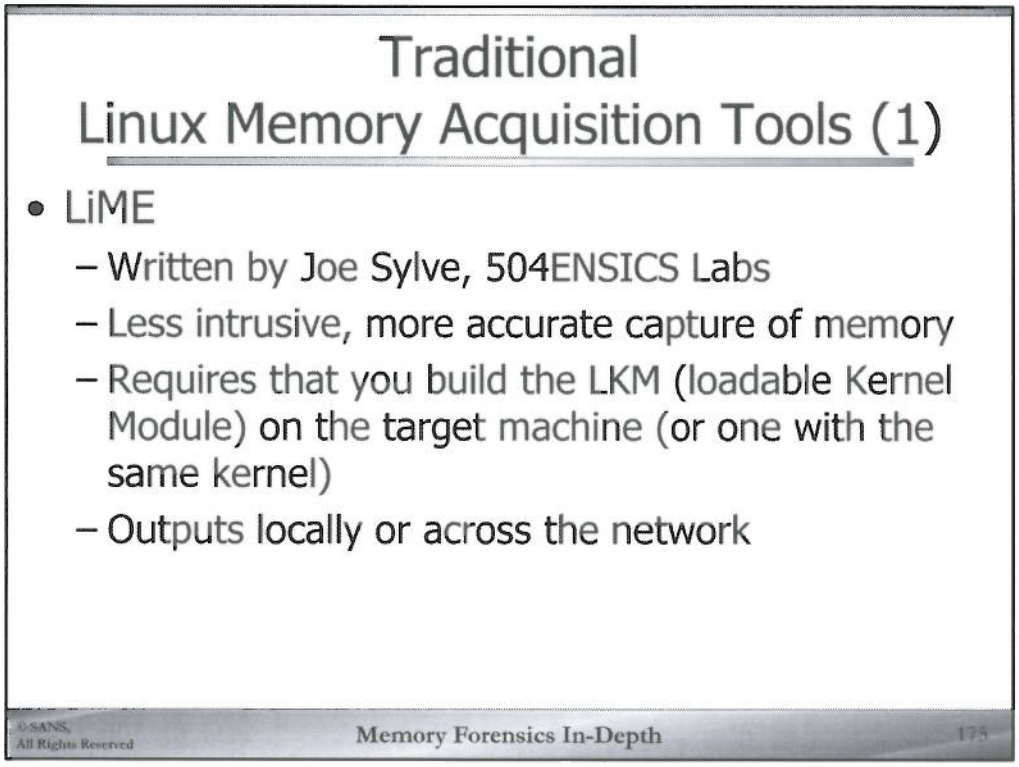
This page intentionally left blank.



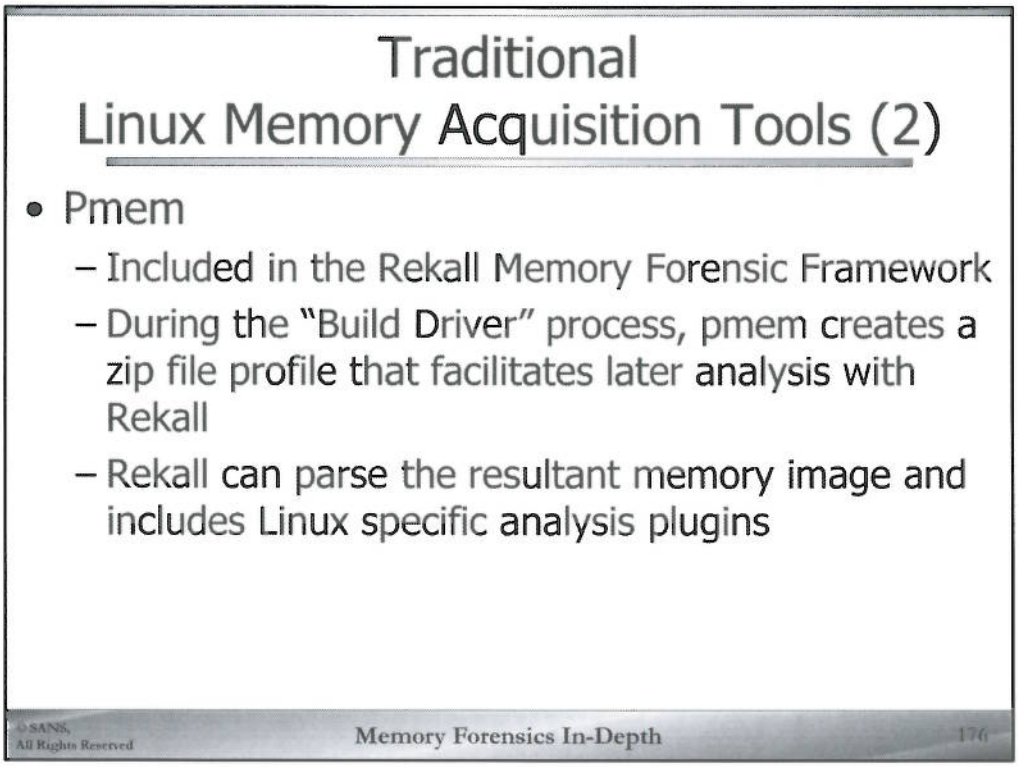
Linux memory acquisition poses additional obstacles for the examiner due to the numerous kernel versions of Linux systems. In order to obtain a memory image from a Linux system, the acquisition tool's driver must be built on the target system or a system that has a matching kernel version. This driver, once loaded, gives the acquisition tool access to physical memory.

The other obstacle in conducting memory analysis of a Linux memory image is the requirement of the target system's kernel profile in order for the parsing tool to correctly interpret memory structures. A profile allows the analysis tool to make sense of the memory image, interpreting the memory structures and offsets specific to the target system's kernel version.

Traditional Linux memory acquisition and analysis tools requires both of these conditions to exist in order for a memory image to be of value to the examiner. Acquisition tools such as LiME or PMem include the initial step of building the loadable driver. Additionally, analysis of Linux memory images with the Volatility framework is possible through the use of dependencies such as dwarfdump which allows for the creation of the target system's profile (a zip file containing information specific to the target system's kernel's data structures (vtypes) and debug symbols) which then enables parsing of memory structures.



In 2012, Joe Sylve from 504ENSICS Labs, developed LiME, originally called DMD. LiME can acquire memory from Linux devices and those that are Linux based, such as Android devices. One of the unique features about LiME is its reduced interaction between user and kernel space processes, decreasing the potential for subversion by malicious software. In order for LiME to run on a target system, extract the software compressed file on the target system and run make to compile the target specific LiME driver. Next, load the driver using the `insmod` command, a program that loads a module into the Linux kernel and in this case, makes physical memory accessible. We are then able to use the tool to acquire an image of target system memory.



The Rekall Memory Forensic Framework includes acquisition tools for Windows, Linux and OSX. There are two acquisition tools provided by Rekall for Linux, pmem and lmap. PMem is a traditional Linux memory imaging tool that follows similar steps to that of LiME, first requiring the build of the target specific driver to be then loaded into the target system's kernel. These steps are shown below:

First Step: Build Driver

```
# tar vxzf linux_pmem_1.0RC1.tgz
linux/
linux/ko_patcher.py
linux/module.c
linux/pmem.c
linux/README
linux/.gitignore
linux/Makefile
# cd rekall/tools/linux
# make
make -C /usr/src/linux-headers-3.11.0-23-generic CONFIG_DEBUG_INFO=y M=`pwd` modules
make[1]: Entering directory `/usr/src/linux-headers-3.11.0-23-generic'
  CC [M]  /cases/pmem/linux/module.o
  CC [M]  /cases/pmem/linux/pmem.o
  Building modules, stage 2.
```

```
MODPOST 2 modules
  CC      /cases/pmem/linux/module.mod.o
  LD [M]  /cases/pmem/linux/module.ko
  CC      /cases/pmem/linux/pmem.mod.o
  LD [M]  /cases/pmem/linux/pmem.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.11.0-23-generic'
cp module.ko module_dwarf.ko
make -C /usr/src/linux-headers-3.11.0-23-generic M=`pwd` modules
make[1]: Entering directory `/usr/src/linux-headers-3.11.0-23-generic'
  CC [M]  /cases/pmem/linux/module.o
  CC [M]  /cases/pmem/linux/pmem.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /cases/pmem/linux/module.mod.o
  LD [M]  /cases/pmem/linux/module.ko
  CC      /cases/pmem/linux/pmem.mod.o
  LD [M]  /cases/pmem/linux/pmem.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.11.0-23-generic'
zip "`uname -r`.zip" module_dwarf.ko /boot/System.map-`uname -r`
  adding: module_dwarf.ko (deflated 66%)
  adding: boot/System.map-3.11.0-23-generic (deflated 79%)

# ls
3.11.0-23-generic.zip  module_dwarf.ko  module.o      pmem.ko    README
ko_patcher.py          module.ko        modules.order  pmem.mod.c
Makefile               module.mod.c     Module.symvers  pmem.mod.o
module.c               module.mod.o     pmem.c         pmem.o
```

Second Step: Load Driver

```
#insmod /cases/pmem/linux/pmem.ko
```

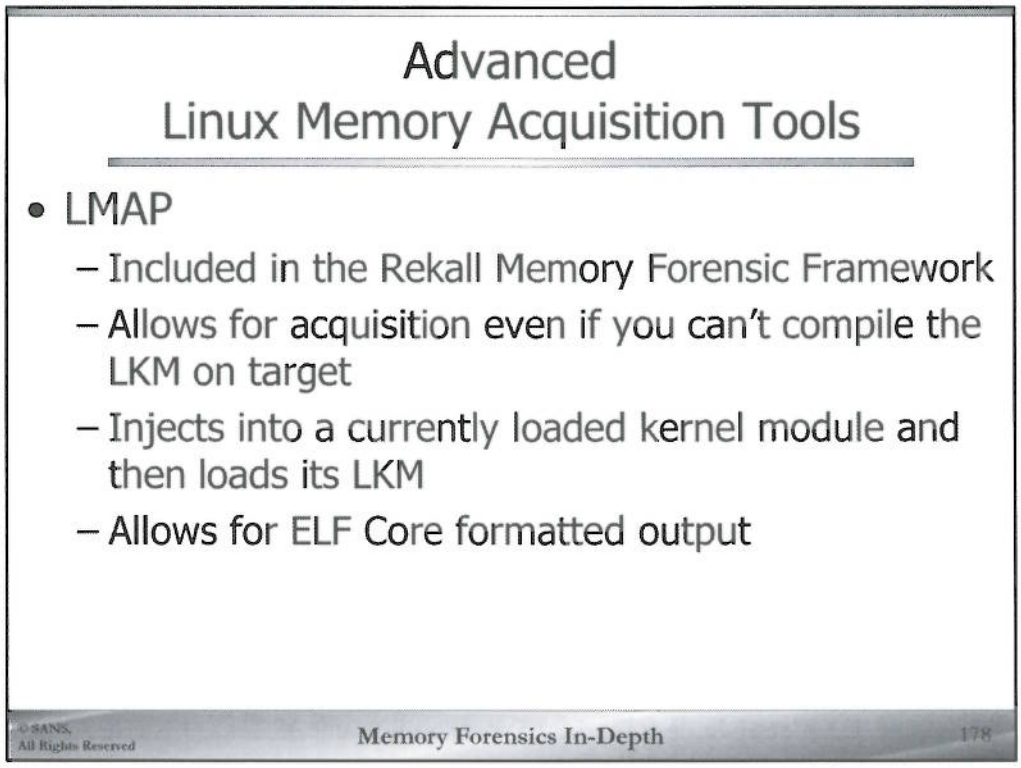
Third Step: Ensure Driver is Running

```
# sudo lsmod |grep pmem
pmem                12680  0
```

Fourth Step: Acquire Memory

```
root@siftworkstation:/cases# dd if=/dev/pmem of=/cases/linux.raw
2097151+1 records in
2097151+1 records out
1073741823 bytes (1.1 GB) copied, 6.49866 s, 165 MB/s
```

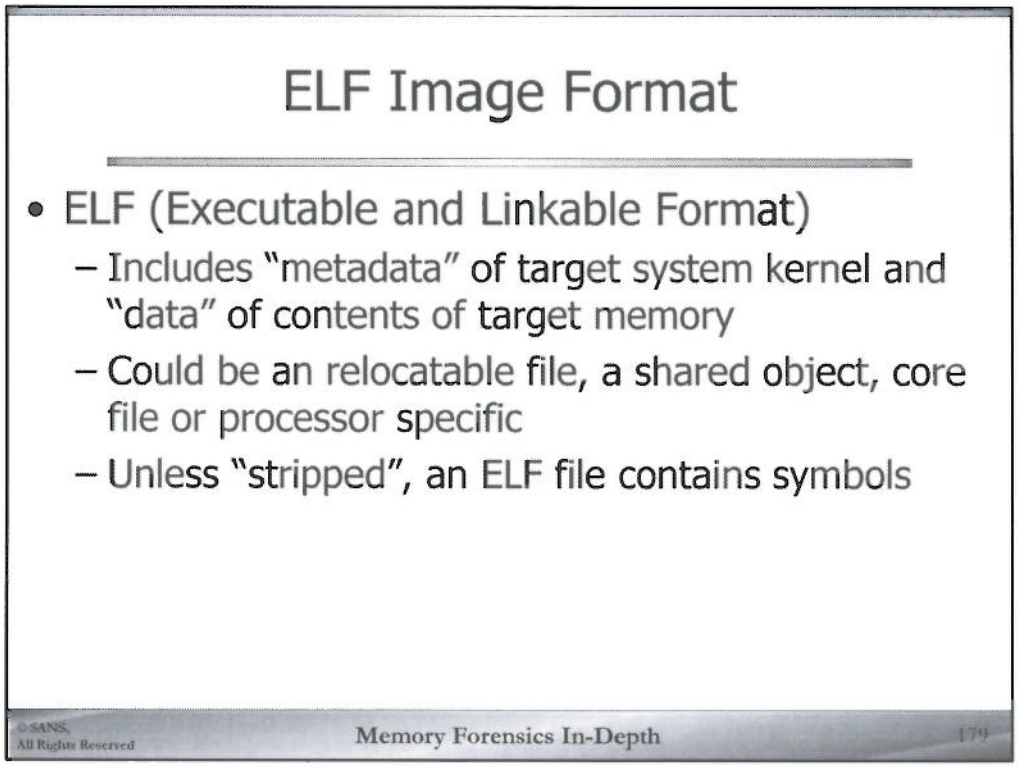
Sources: http://www.forensicswiki.org/wiki/Rekall



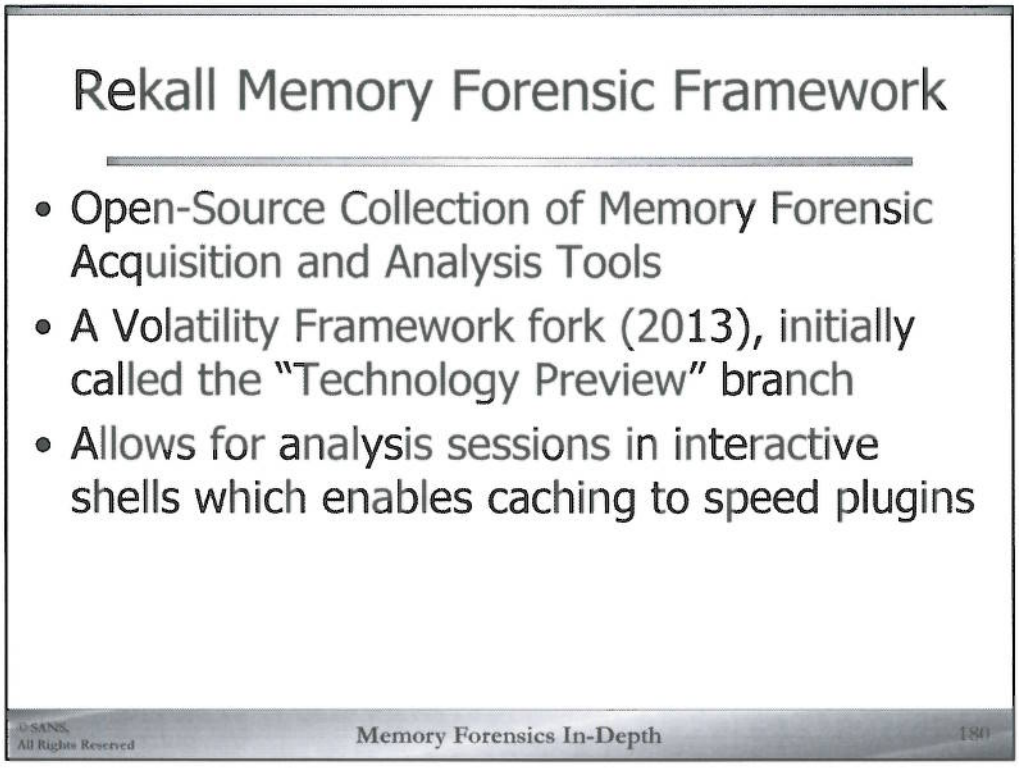
LMAP is included in the Rekall Memory Forensic Framework. It allows for acquisition even if you can't compile the LKM on target. This is significant because traditionally, an LKM must be compiled for the specific kernel on which it was being used. This normally requires the investigator to locate a similar system to compile the LKM on. Compiling on the target system is not usually recommended since it will disturb memory, the very thing you desire to analyze. However, compiling on the target system may be the only option if the target system uses a custom kernel. This may be prohibitively difficult if the target system lacks a compiler and kernel symbols (required to build an LKM).

LMAP avoids this problem by borrowing a technique used in some rootkits. It "infects" a currently loaded kernel module and then loads its LKM. This brings LMAP into memory with the currently loaded legitimate module. LMAP outputs memory in the ELF core format. This is convenient because it allows for kernel symbol information (obtained at runtime) to be stored inside the memory dump file, something that is not possible with a raw memory dump.

Sources: http://www.rekall-forensic.com/docs/References/Presentations/LMAP-DFRWS_EU_2014.html



The LMAP tool (as well as winpmem, which we covered earlier in the course) allows the investigator to create an ELF output file. The advantages of using the ELF file format for memory dumps is clear, since symbols can be stored in the ELF headers that allows for the interpretation of the data by memory analysis tools.

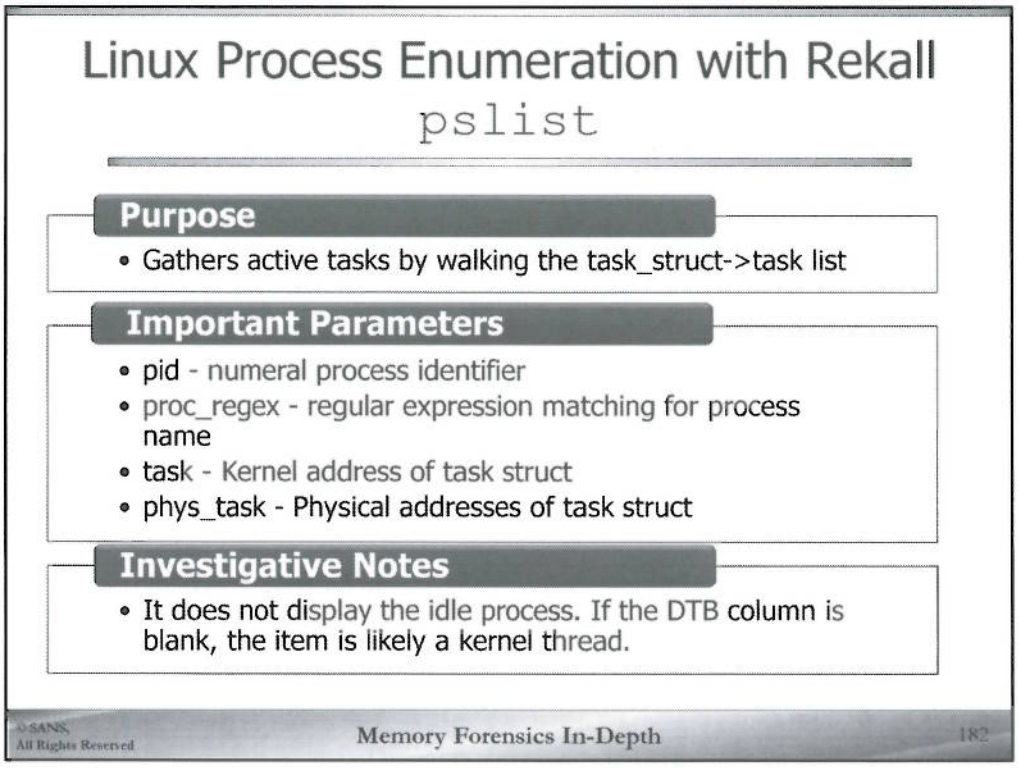


The Rekall Memory Forensic Framework in a collection of memory acquisition and analysis tools implemented in Python under the GNU General Public License. It originated in 2011 as the "Technology Preview" branch of the Volatility Framework, with goals of streamlining code and improving efficiency, performance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

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

```
ubuntu.dd 19:00:46> print session
Rekall Memory Forensics session Started on Thu Jun 26 19:13:07 2014.

Config:
{
 base_filename = ubuntu.dd
```

```
buffer_size = 20971520
 cache = {
  default_address_space = AMD64PagedMemory@0x01C0D000 (Kernel AS@0x1c0d000)
  dtb = 29413376
  kaslr_shift = 0
 }
 debug = False
 ept = None
 filename = /cases/ubuntu.dd
 help = False
 ipython_engine = None
 no_autodetect = False
 nocolors = False
 notebook_dir = /root
 pager = None
 paging_limit = None
 plugin = []
 profile = ./3.11.zip
 profile_path = ['http://profiles.rekall.googlecode.com/git/']
 renderer = TextRenderer
 run = None
 timezone = UTC
 verbose = False
}
```



Of course, the first step in our Six Step Memory Analysis Methodology is Process Enumeration. It is no different when we are analyzing Linux target systems. Rekall includes the pslist plugin to enumerate process, doing so by following the task list, the linked *task_structs* representing active processes on the system. This plugin includes several filtering parameters such as process PID, physical address and kernel address of the process' task struct and regular expression matching.

In order to get detailed information on a specific plugin while in the interactive shell, type "help <plugin>". For example, this is the additional information provided for pslist.

ubuntu.dd 19:38:51> help pslist

----------> help(pslist)

RunPlugin(...)

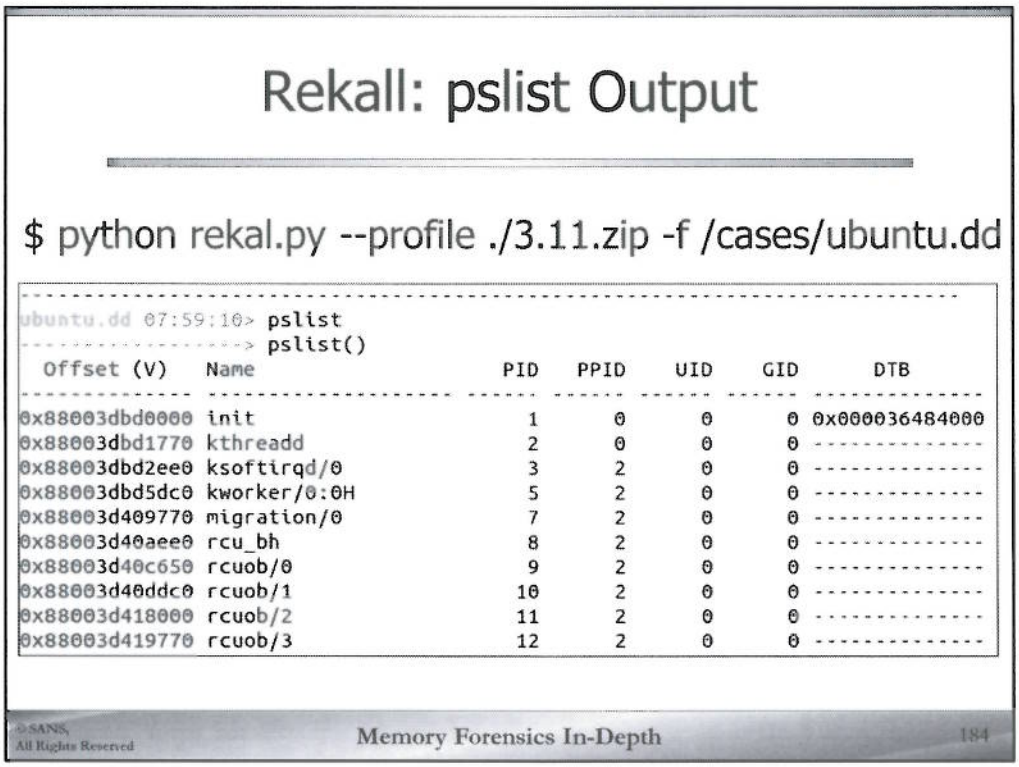
Gathers active tasks by walking the task_struct->task list.

It does not display the swapper process. If the DTB column is blank, the item is likely a kernel thread.Filters processes by parameters.

Link:

http://epydocs.rekall.googlecode.com/git/rekall.plugins.linux.pslist.LinuxPsList-class.html

Parameter	Documentation
proc_regex	A regex to select a process by name.
task	Kernel addresses of task structs.
dtb	The DTB physical address.
pid	One or more pids of processes to select.
task_head	Use this as the process head. If specified we do not use kdbg.
phys_task	Physical addresses of task structs.
output	If specified we write output to this file.
renderer	Use this renderer for the output.

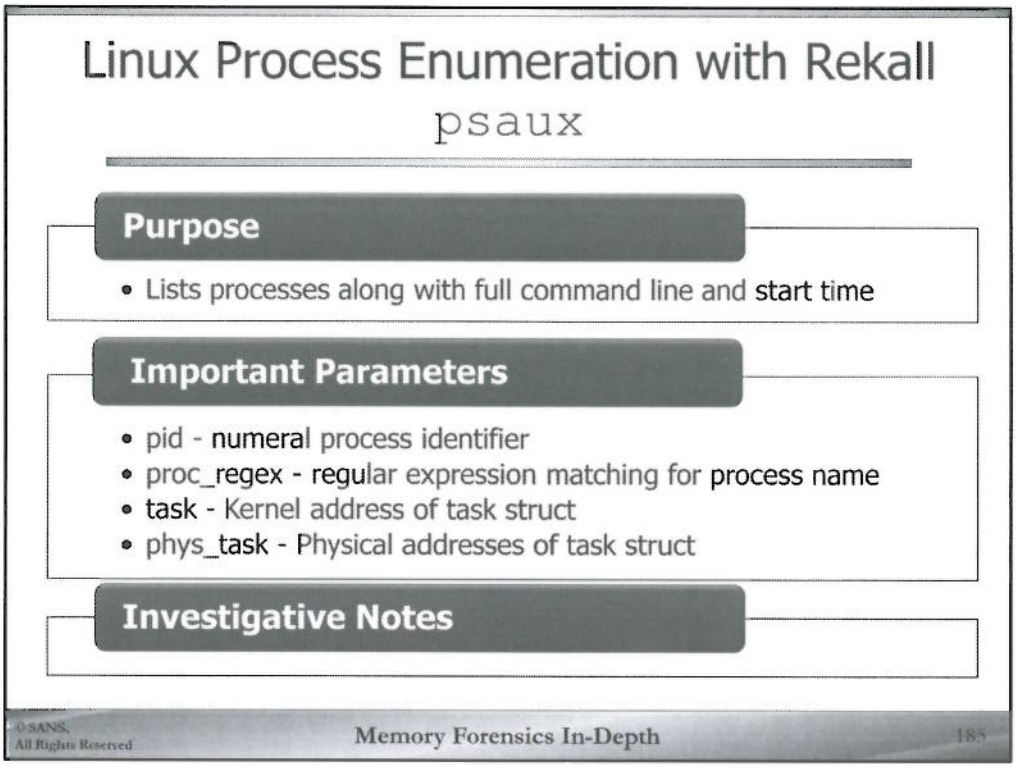


Notice with this first plugin we run in Rekall that we are in an interactive session within the context of the memory image we specified when launching Rekall. We need only enter the command *pslist* and Rekall knows the memory image and profile to use in order to derive the process listing.

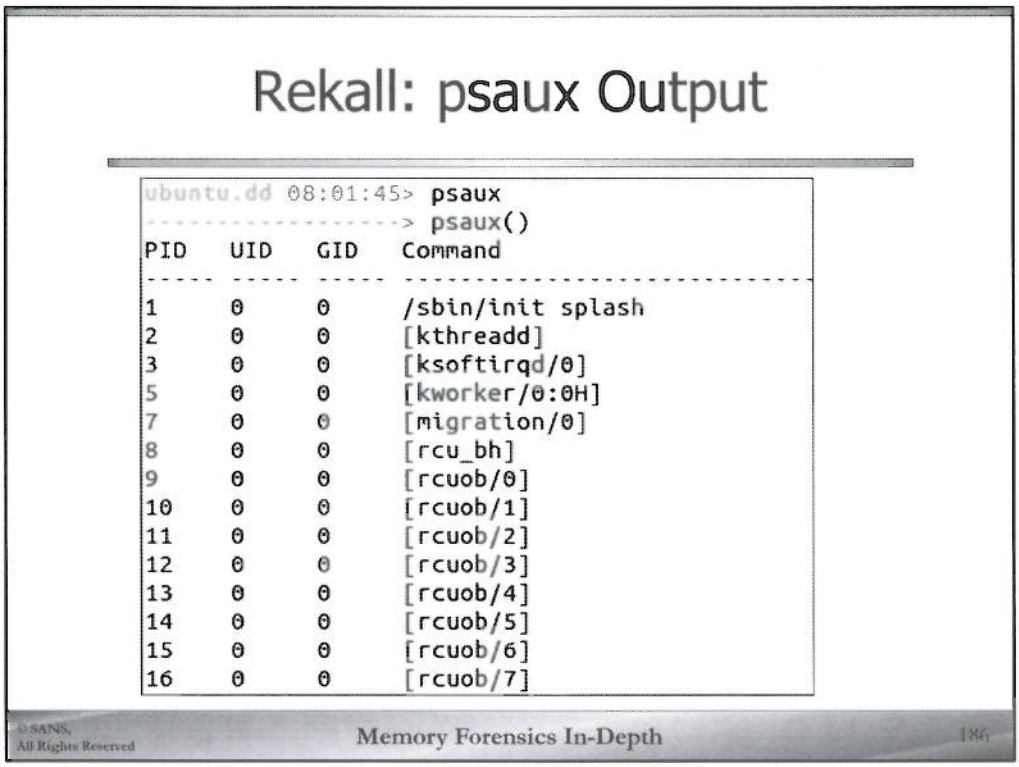
Shown below is a pslist using the regular expression filter feature.

```
ubuntu.dd 19:27:54> pslist proc_regex="gnome"
------------------> pslist(proc_regex="gnome")
 Offset (V)  Name                    PID  PPID  UID   GID     DTB                Start Time
-------------- -------------------- ------ ------ ------ ------ -------------- ------------------------
0x880025d75dc0 gnome-keyring-d      2667     1  1000  1000 0x000025c52000          -
0x8800364eaee0 gnome-session        2678  2423  1000  1000 0x00003b249000          -
0x880039e71770 gnome-settings-      2727  2678  1000  1000 0x000039afc000          -
0x880038b34650 gnome-fallback-      2765  2678  1000  1000 0x000038b60000          -
0x88002516c650 gnome-terminal       2958     1  1000  1000 0x0000252b6000          -
0x880025179770 gnome-pty-helpe      2963  2958  1000  1000 0x00003bb65000          -
0x880025d89770 gnome-screensav      3087  2678  1000  1000 0x000025c06000
```

Sources: http://www.rekall-forensic.com/docs/Manual/tutorial.html#_interactive_plugins



The *ps aux* command in Linux with the 'a' and 'u' options includes a comprehensive listing of all processes for all users. When run on a live system, other output includes process owner, %CPU, %MEM, virtual memory usage and controlling tty. The Rekall *psaux* command shows a simplified output of this command, listing all processes with instantiating command line and start time. Several filter parameters can be used with psaux such as process PID, physical address of task struct and regular expression matching.



Shown in the screenshot above is the *psaux* command being run from inside interactive Rekall framework.

Shown below is psaux with PID filter option:

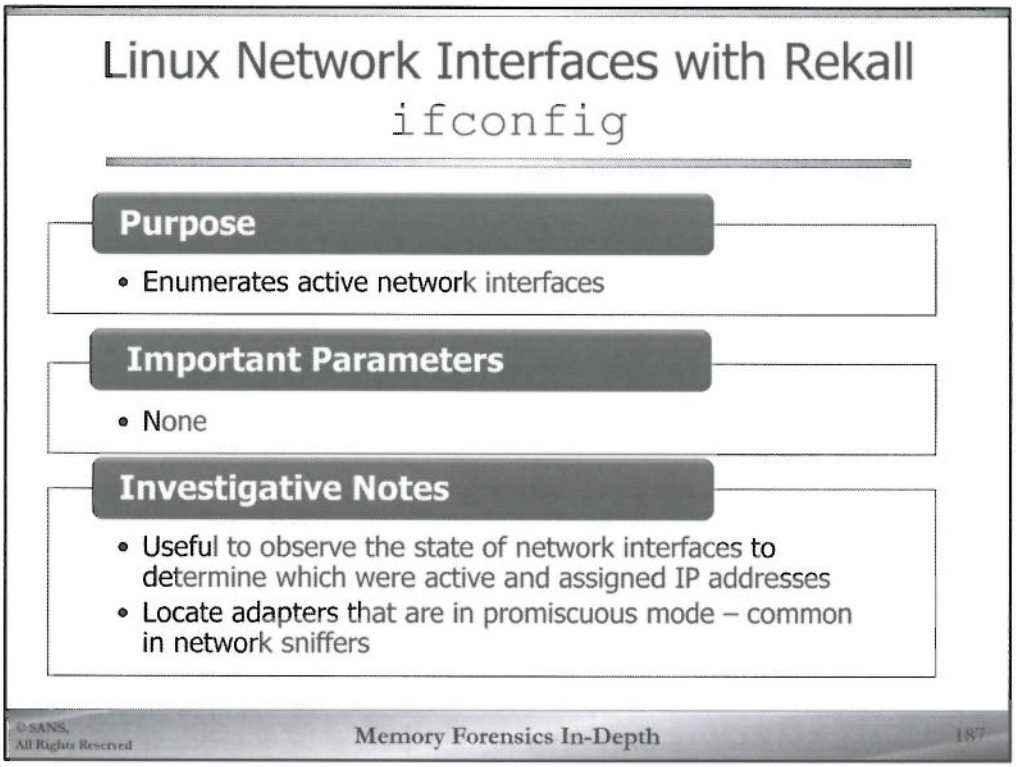
ubuntu.dd 12:01:52> psaux 16275

----------> psaux(16275)

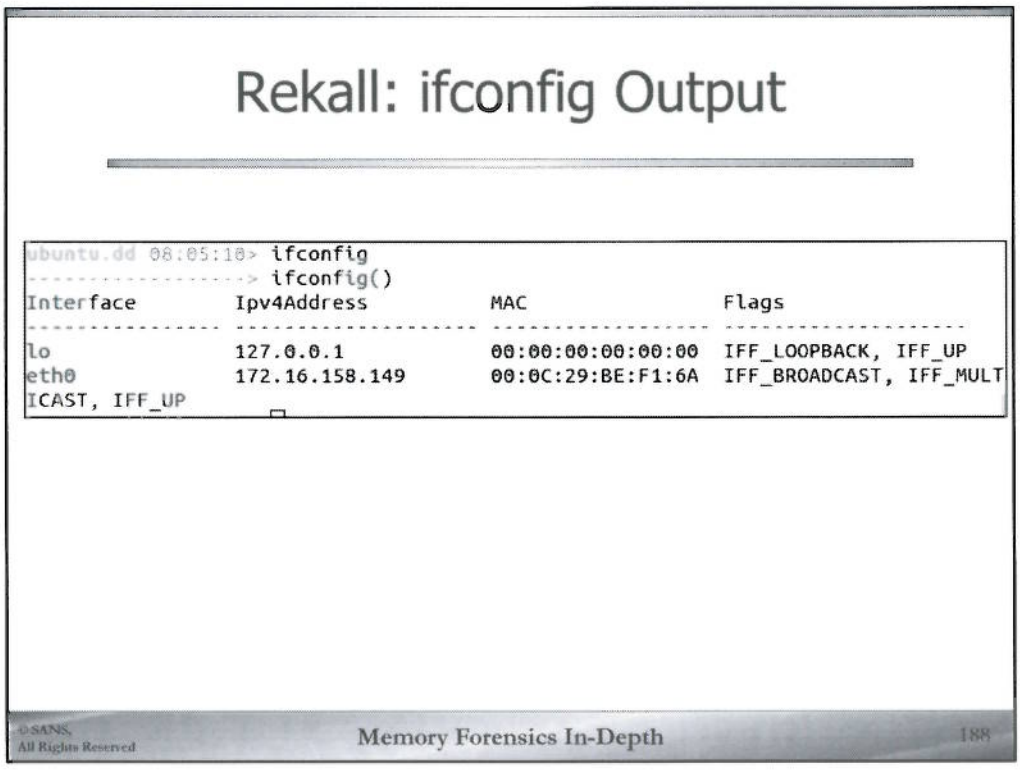
PID UID GID Command

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

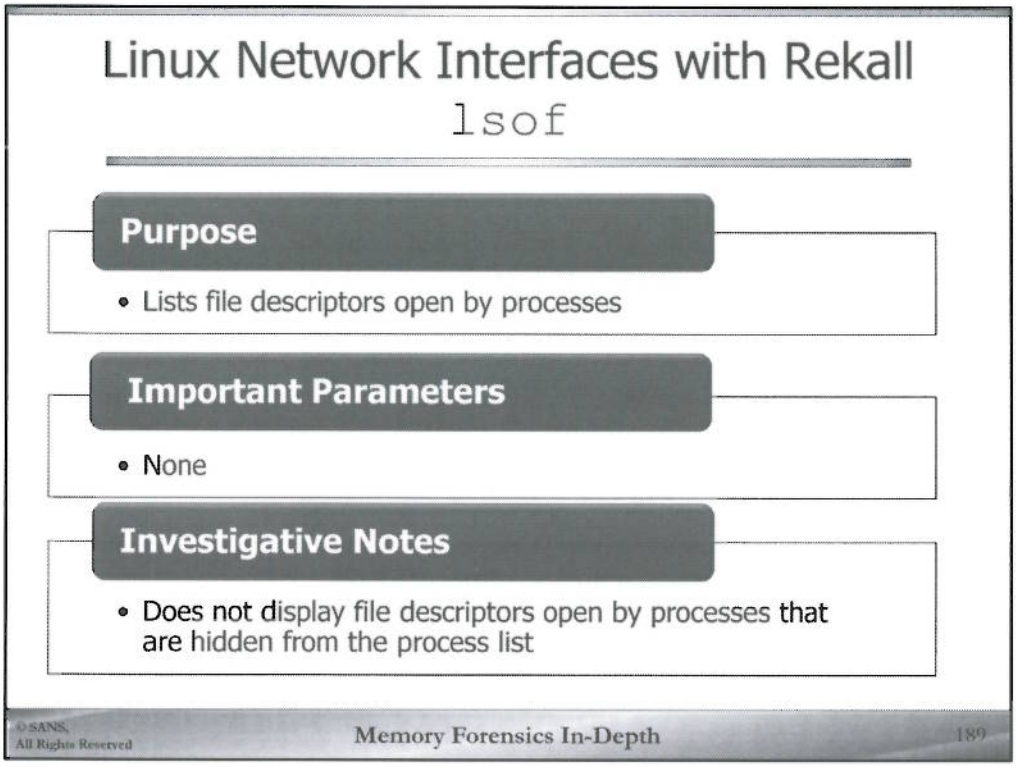
16275 0 0 sleep 946



Rekall's ifconfig plugin enumerates active network interfaces. It is useful to observe the state of network interfaces to determine which were active and as well as determining their configured IP addresses. Perhaps the most significant investigative use of the ifconfig plugin is to locate adapters that are in promiscuous mode. This is very common in network sniffers, such as tcpdump or dsniff. Note that some network sniffers operate without promiscuous mode, simply capturing all unicast traffic destined for the IP addresses configured on the interface (they also get all multicast and broadcast segment on the network). A possible solution then is to look for raw sockets using the lsof, since these are common to sniffers regardless of whether they use promiscuous mode.

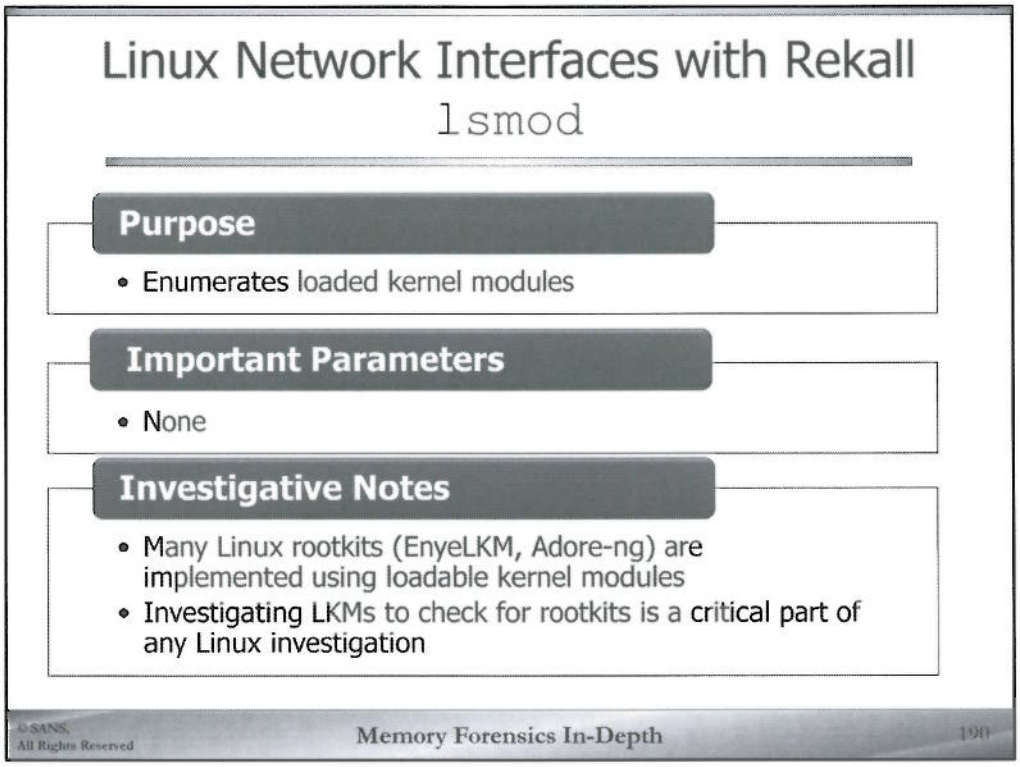


The above slide shows the output of rekall's ifconfig plugin.



The lsof plugin displays open files in the OS, similar to the way the command with the same name works inside the operating system. Just listing files opened by each process would be significant enough. But in Linux it is even more significant since almost everything in Linux is represented as a file. This includes network sockets. Some Linux rootkits may hide listening TCP ports from netstat by intercepting calls to read from /proc/net/tcp. However, the same rootkit might forget to hide the socket from lsof. Because the listening socket is represented as a file, it is found in lsof. The "everything is a file" makes it much more difficult for rootkit authors to completely hide objects in Linux.

One interesting output in lsof is that of a raw socket. Most are familiar with SOCK_STREAM (TCP) and SOCK_DGRAM (UDP). However, a raw socket (SOCK_RAW) can be used to listen for traffic to all ports (and all protocols) on an interface. When combined with promiscuous mode, this is very powerful in creating network sniffers. Some port scanners and fuzzing utilities also use raw sockets for writing since they desire to violate typical protocol behaviors (e.g. sending packets with intentionally incorrect checksums).

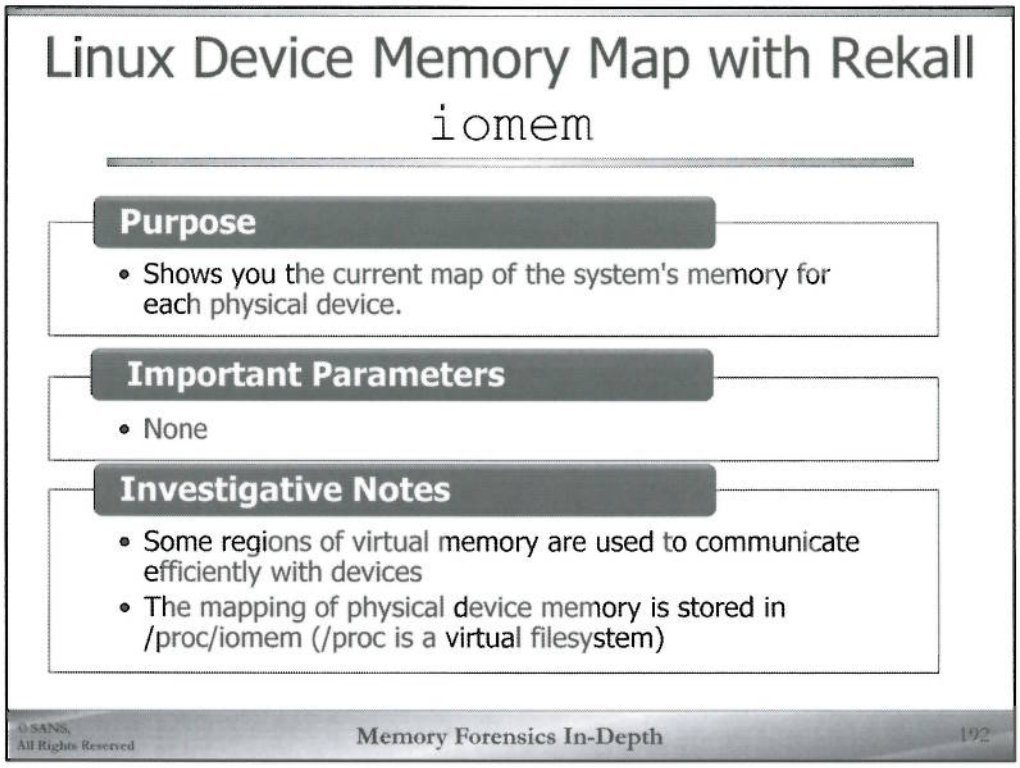


The lsmod plugin enumerates loadable kernel modules. Many Linux rootkits (EnyeLKM, Adore-ng) are implemented using loadable kernel modules. Investigating LKMs to check for rootkits is a critical part of any Linux investigation. Failing to do so may result in missing a rootkit running on the Linux system. EnyeLKM in particular implemented a network accessible backdoor that gave an attacker a backdoor root shell on the system simply by sending a single, specially crafted packet to the compromised host. This is the sort of thing that's easy to find by searching modules in memory, but not the sort of thing you want to miss in an investigation.

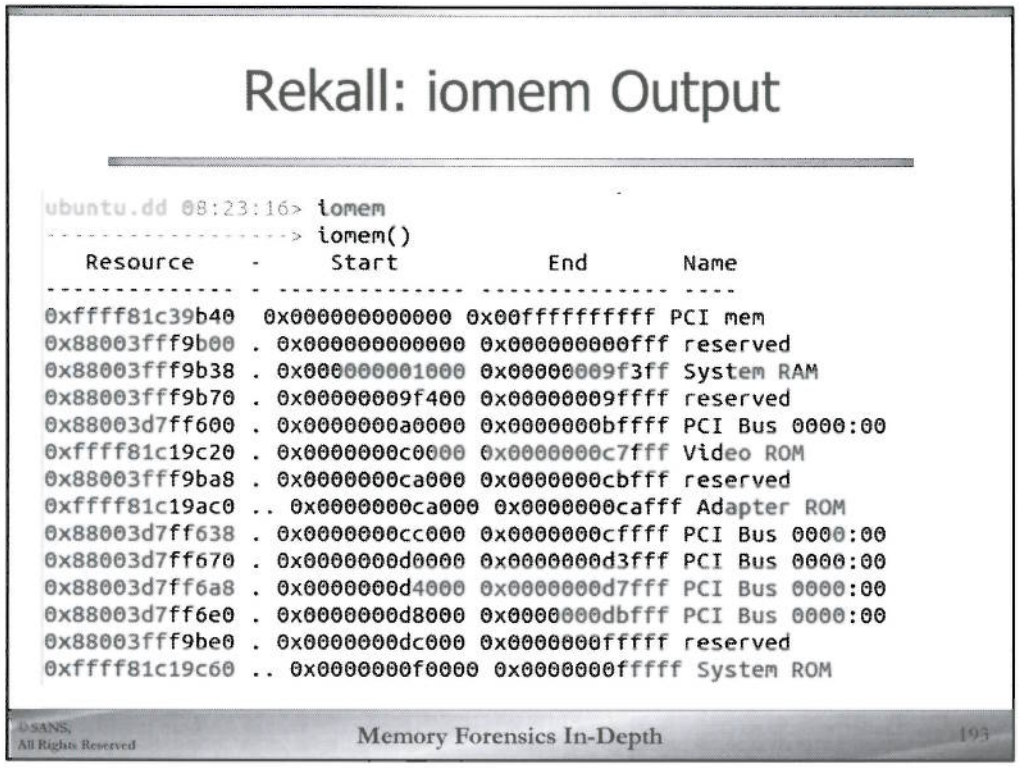
Rekall: lsmod Output

```
ubuntu.dd 08:17:22> lsmod
------------------> lsmod()
******************** Overview ********************
   Virtual       Core Start   Total Size Name                 Section
-------------- -------------- ---------- -------------------- ---------
0xffffa035a160 0xffffa0358000      12680 pmem
0xffffa03aa820 0xffffa03a0000      53928 vmhgfs
0xffffa039d260 0xffffa0399000      23111 dm_crypt
0xffffa0390700 0xffffa038c000      25783 snd_ens1371
0xffffa031a1c0 0xffffa0317000      19693 gameport
0xffffa0384720 0xffffa036a000     134967 snd_ac97_codec
0xffffa033a2e0 0xffffa0333000      38796 ib_iser
0xffffa03140a0 0xffffa0312000      12730 ac97_bus
0xffffa03660c0 0xffffa035d000      48728 rdma_cm
0xffffa0354320 0xffffa034b000      48393 ib_cm
0xffffa02bb000 0xffffa02b8000      18793 iw_cm
```

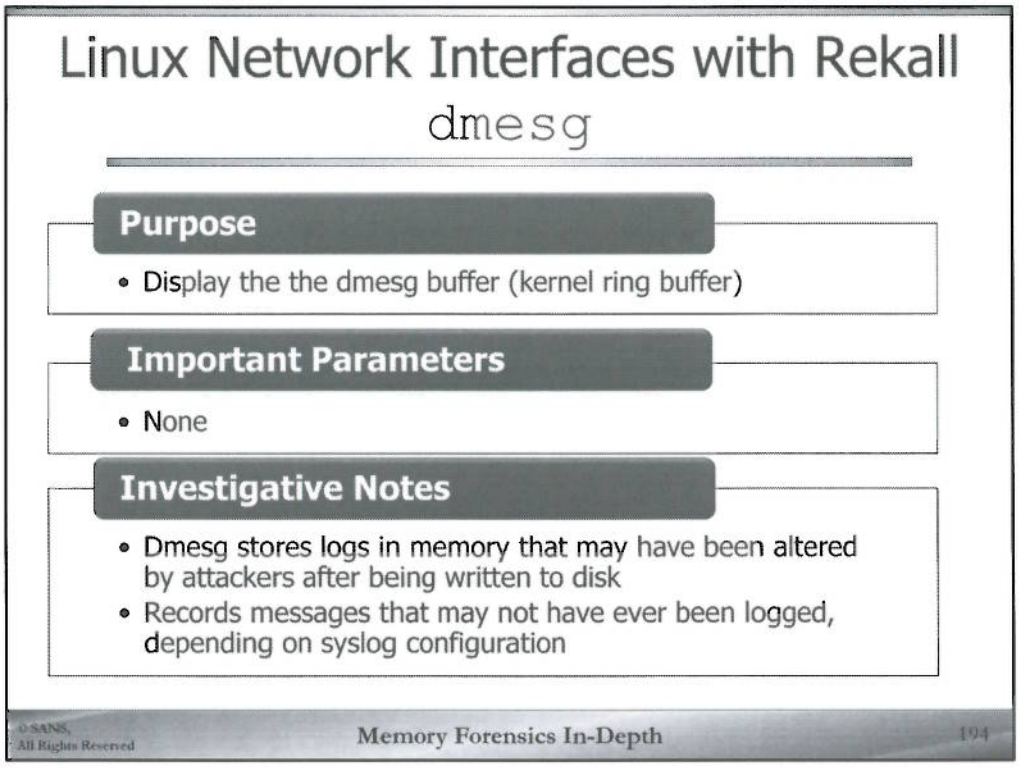

The slide above shows output of the lsmod plugin. Obviously malicious and unknown module names should be investigated. Knowing what normal looks like on Linux is substantially harder than on Windows, owing to the number of variations in Linux kernels. The pmem acquisition tool's lvm named "pmem" is seen here in the list of loaded kernel modules.



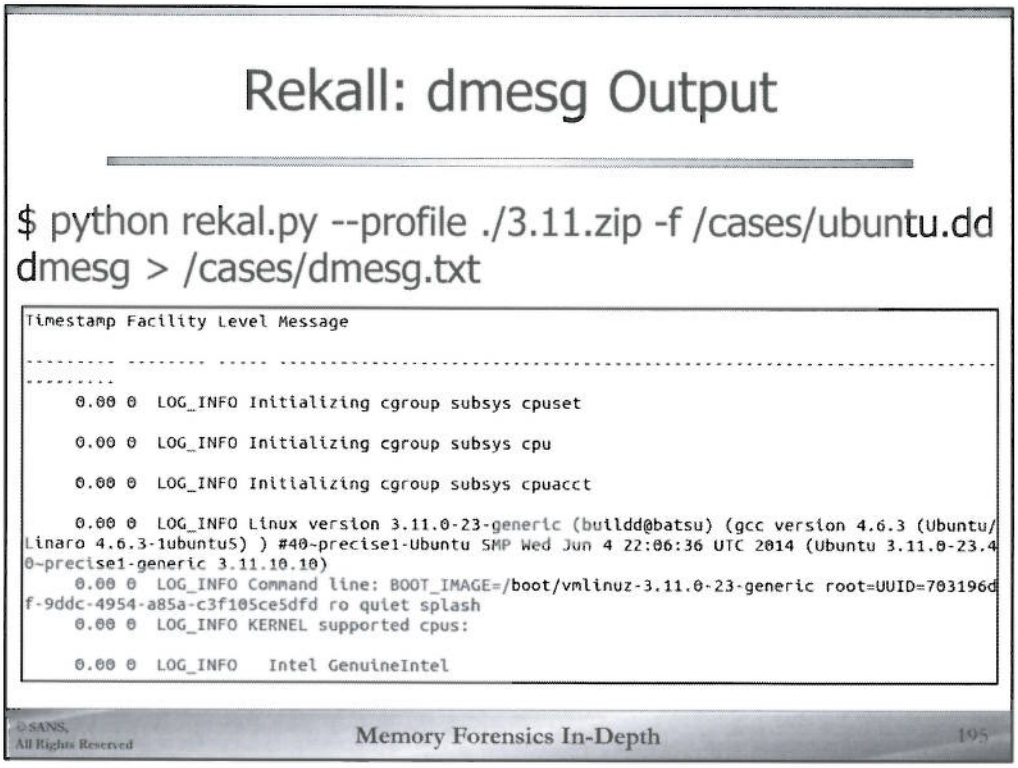
/proc/iomem - This file shows you the current map of the system's memory for each physical device. Certain regions of virtual memory are not backed by physical memory in the page tables. Rather, when these memory regions are accessed, the system dynamically maps memory from devices into these areas. This allows for efficient communication with hardware devices and dramatically reduces the complexity of programming tasks.



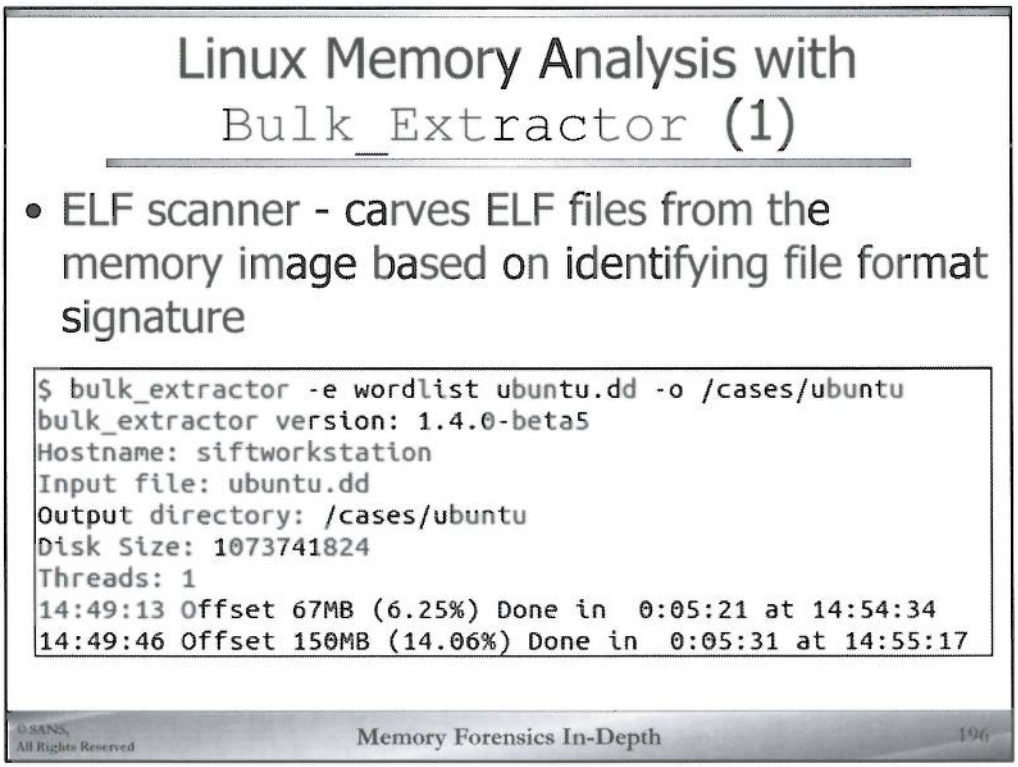
Note the device hierarchy above, represented by dots in the second column. The first line ("PCI mem") shows the entire address range available for PCI devices. However, note that the "System RAM" address range falls within the range for the "PCI mem". This is because "System RAM" is simply a sub-allocation of the larger "PCI mem" address range.



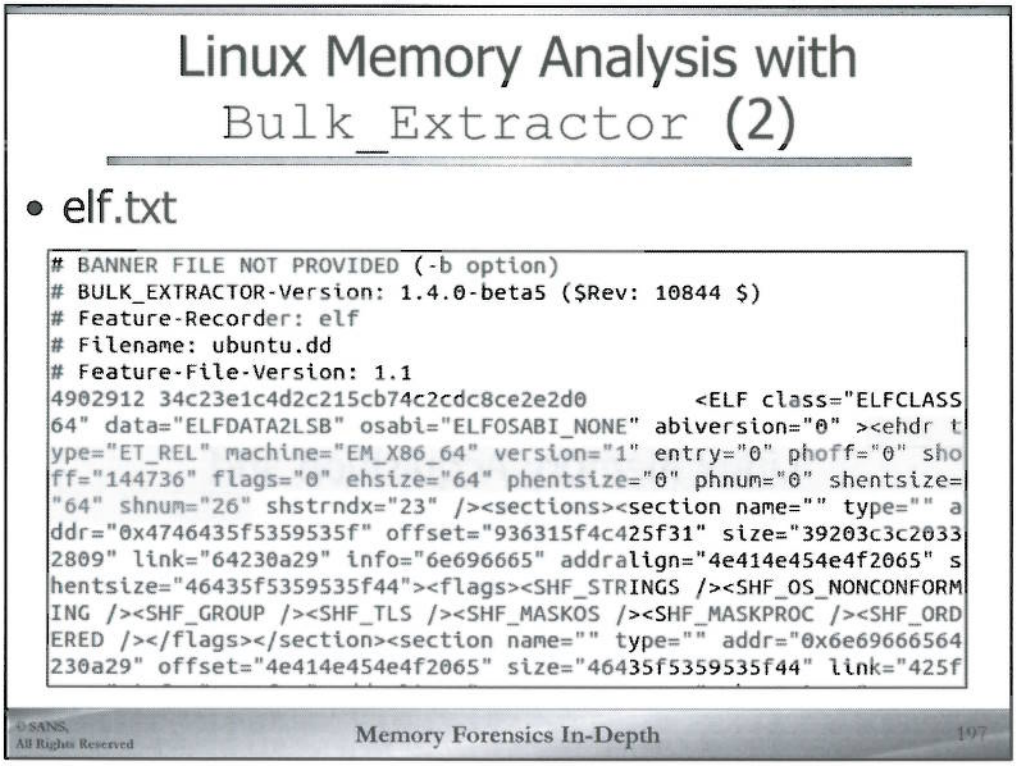
The rekall dmesg plugin is used to examine the kernel ring buffer. The kernel ring buffer contains boot messages that were generated by the kernel. Depending on the configuration of syslog, some of these messages end up in log files. However, analyzing the ring buffer (through dmesg) may provide evidence of log tampering. Further, there may be messages present in the ring buffer that were never written to a log file. This of course depends on the syslog configuration. The default size of the kernel ring buffer is 16392 bytes. However, this value may be changed at compile time by modifying kernel parameters (CONFIG_LOG_BUF_SHIFT).



The dmesg buffer is a great location to check for evidence of devices being inserted (USB drives for instance). Although many entries in the dmesg buffer also find their way into other logs files (via syslog or klog functionality), it is still worth checking dmesg. The reason for this is that attackers may find it easy to edit files on the disk, but hard to change the same data in the dmesg buffer in memory. This may then point to evidence of log tampering. Also note that some messages may be recorded to the dmesg buffer before the logging subsystem has been initialized. This creates another unique source for log entries in dmesg.



The unstructured stream-based analysis tool, Bulk_Extractor, has a scanner that is particularly useful when parsing Linux memory images. The ELF Scanner attempts to carve ELF files based on the file format signature. We know that carving an entire contiguous file from a physical memory image is unlikely, but value may be found in identifying ELF header information. The screenshot above shows that the bulk_extractor tool is run using the same syntax against the Linux memory image, with an output directory of "/cases/ubuntu".



Shown above is a portion of what the ELF scanner carved from an Ubuntu memory image. The ELF header values highlighted in the screenshot above are interpreted in the following chart.

ELF Header Values

ehdr type="ET_REL" - Object File Type (ET_REL = Relocatable File)

machine="EM_X86_64" - Required Architecture for an Individual File

version="1" - Object File Version

entry="0" - Entry Address Point

phoff="0" - Program Header Offset (Bytes into File)

shoff="144736" - Section Header Offset (Bytes into File)

flags="0" - Processor-specific Flags

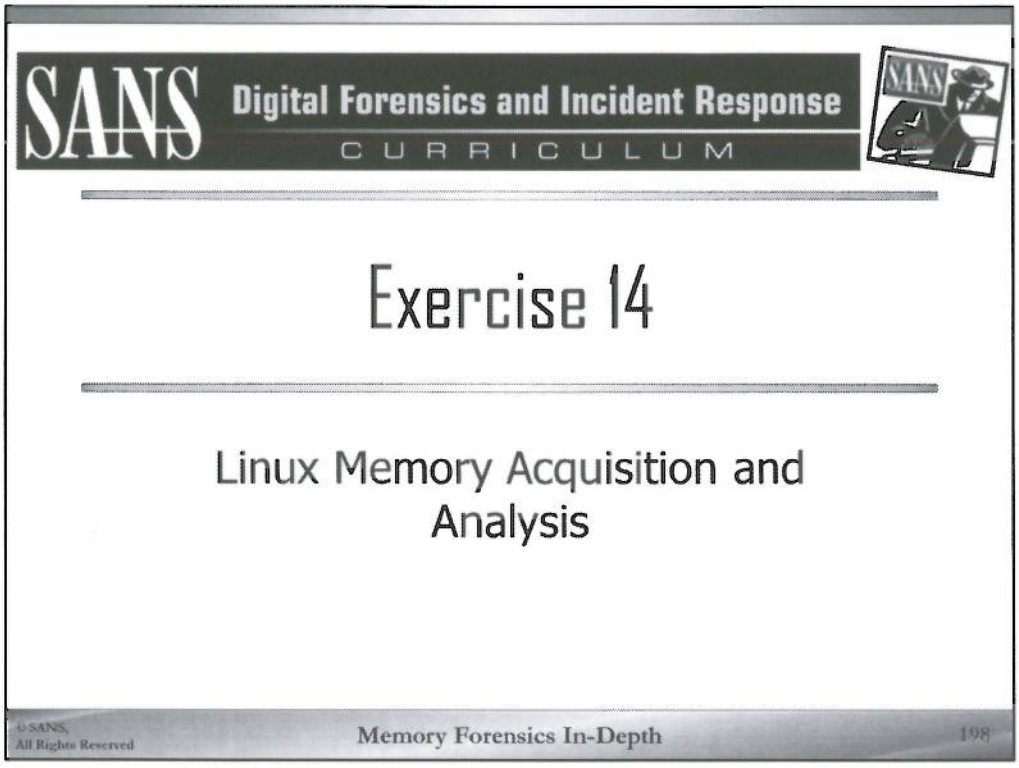
ehsize="64" - ELF header's size in bytes

phentsize="0" - Size in bytes of one entry in the file's program header table

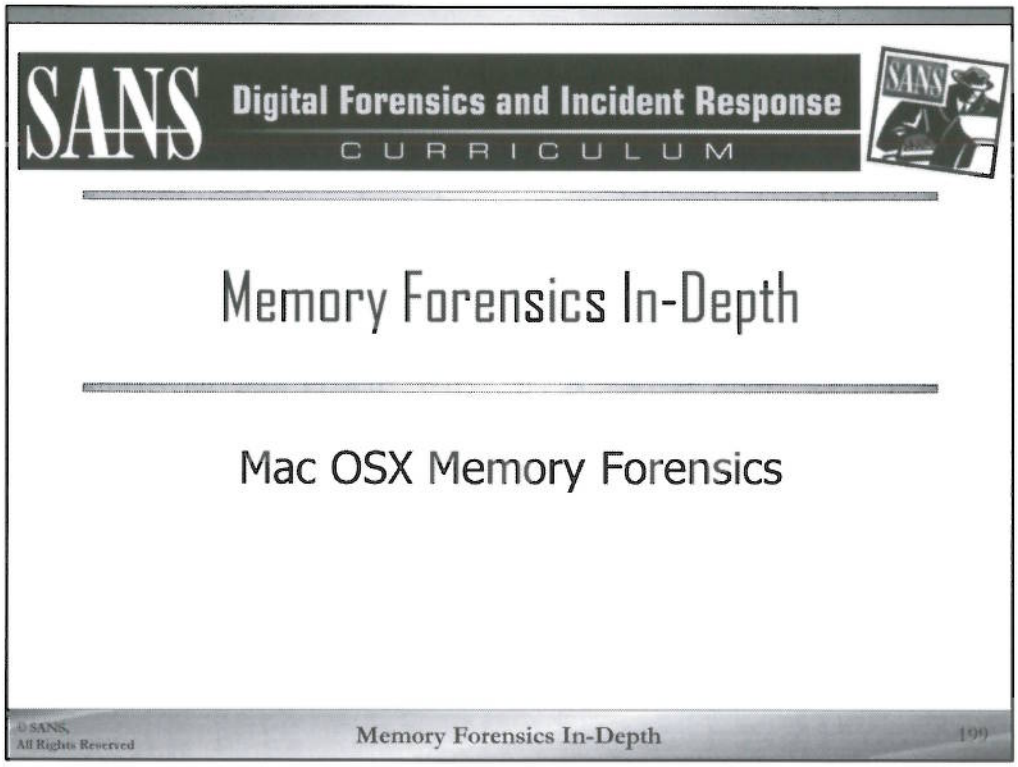
phnum="0" - Number of entries in the program header table

shentsize="64" - Section header's size in bytes

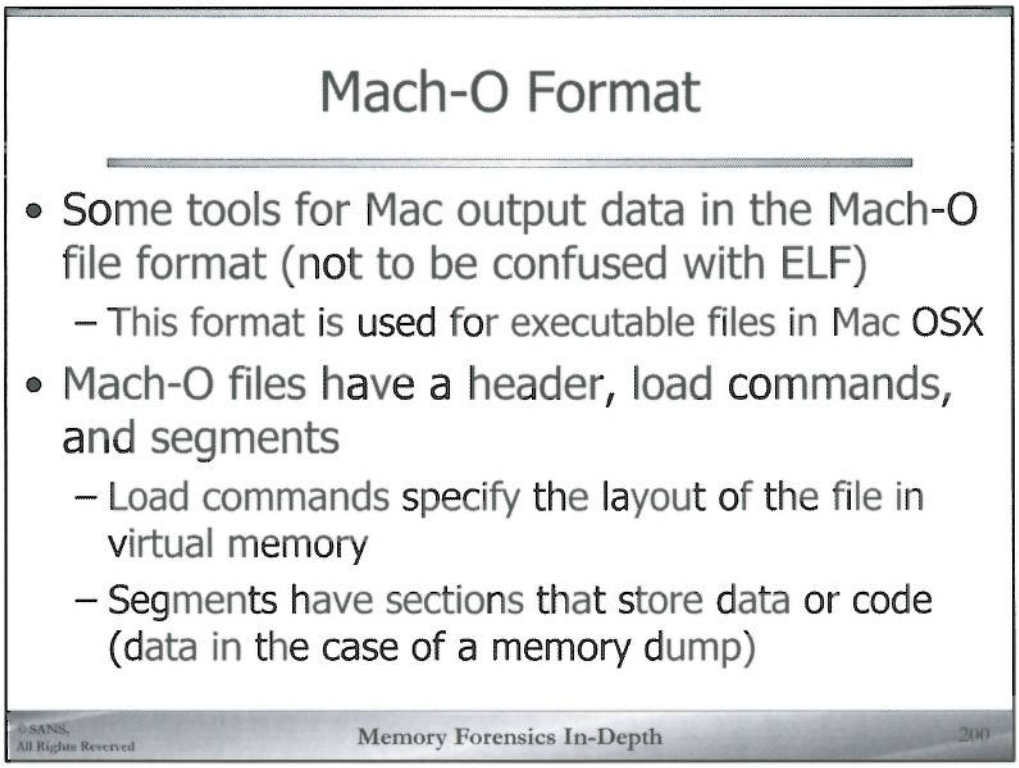
shnum="26" - Number of entries in the section header table



This page intentionally left blank.



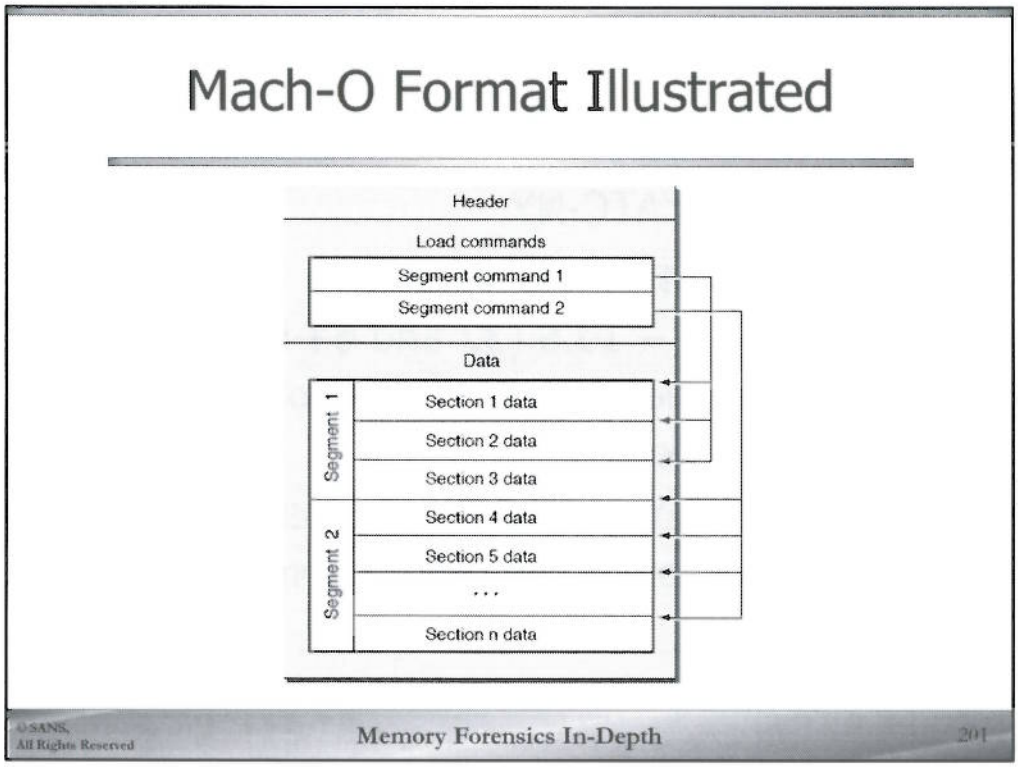
This page intentionally left blank.



Some memory capture tools for Mac output the memory capture data in the Mach-O file format. This format is used for executable files in Mac OSX. This should not be confused with ELF, which is the normal executable file format in Linux.

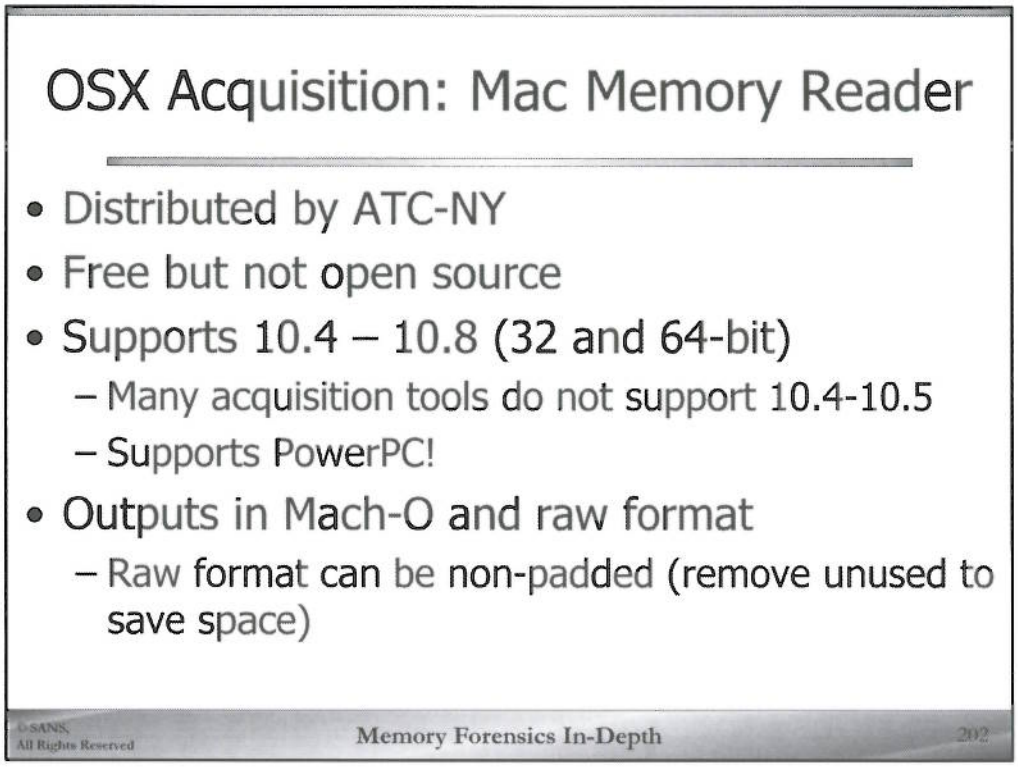
Mach-O files have a header, load commands, and segments. Load commands specify the layout of the file in virtual memory. Segments have sections that store data or code (data in the case of a memory dump). Mach-O files that are used as executables often have a special segment called the link edit segment. This contains the symbol table and string table, which is used by the loader at run time. Mach-O files that contain only memory dump data will not have a link edit segment since they are not executable.

Source:
https://developer.apple.com/library/mac/documentation/developertools/conceptual/MachORuntime/Mach-O_File_Format.pdf



This image, taken from the Mach-O file format documentation shows the layout of a Mach-O format. In the case of a memory image, the segment load commands will be used to describe the relative positions of section data in the full memory dump. This is because physical memory is often sparse and will contain many sections filled entirely with zeroes.

Image credit:
https://developer.apple.com/library/mac/documentation/developertools/conceptual/MachORuntime/Mach-O_File_Format.pdf

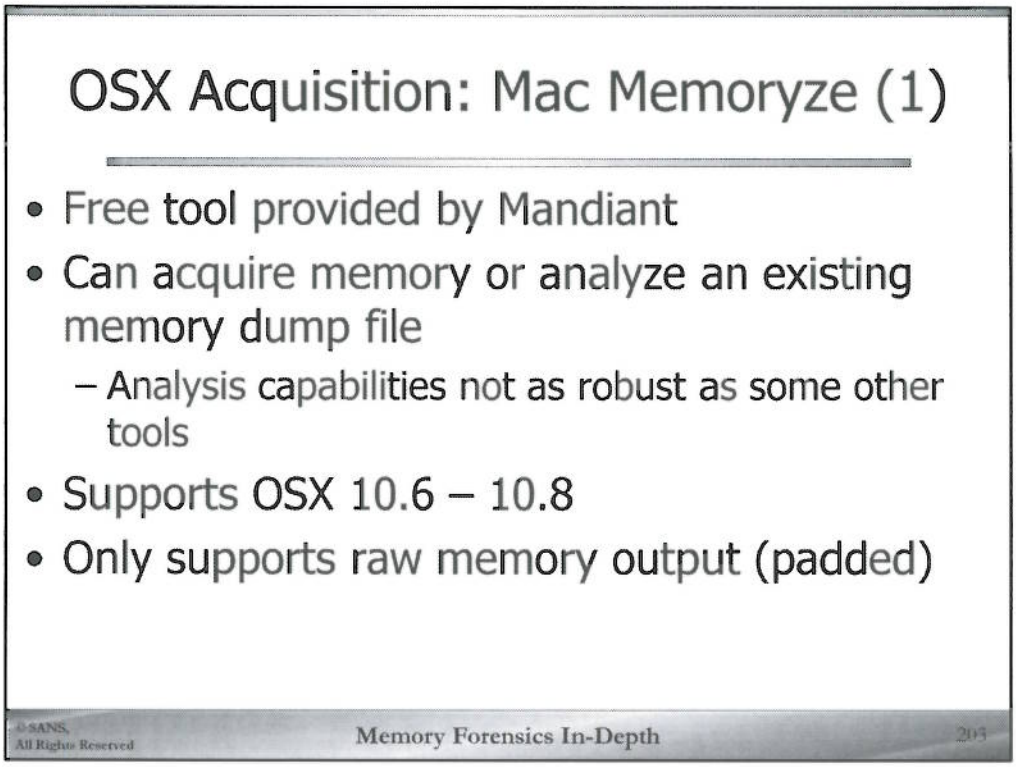


The Mac Memory Reader from CyberMarshal is a free tool to acquire OSX memory. This tool is especially noteworthy because it supports:

- PowerPC (most memory acquisition tools support Intel)
- OSX 10.4 and 10.5 (many tools only support 10.6 and above)

The tool also supports output in both Mach-O and raw format (buffered and unbuffered). Most raw acquisition utilities output only in a buffered format (which is ideal for analysis since physical offsets are preserved), but this creates a much larger file than necessary for simply performing string searches. Unfortunately, this tool does not support OXS 10.9.

Source: http://www.cybermarshal.com/cyber-marshal-utilities/mac-memory-reader

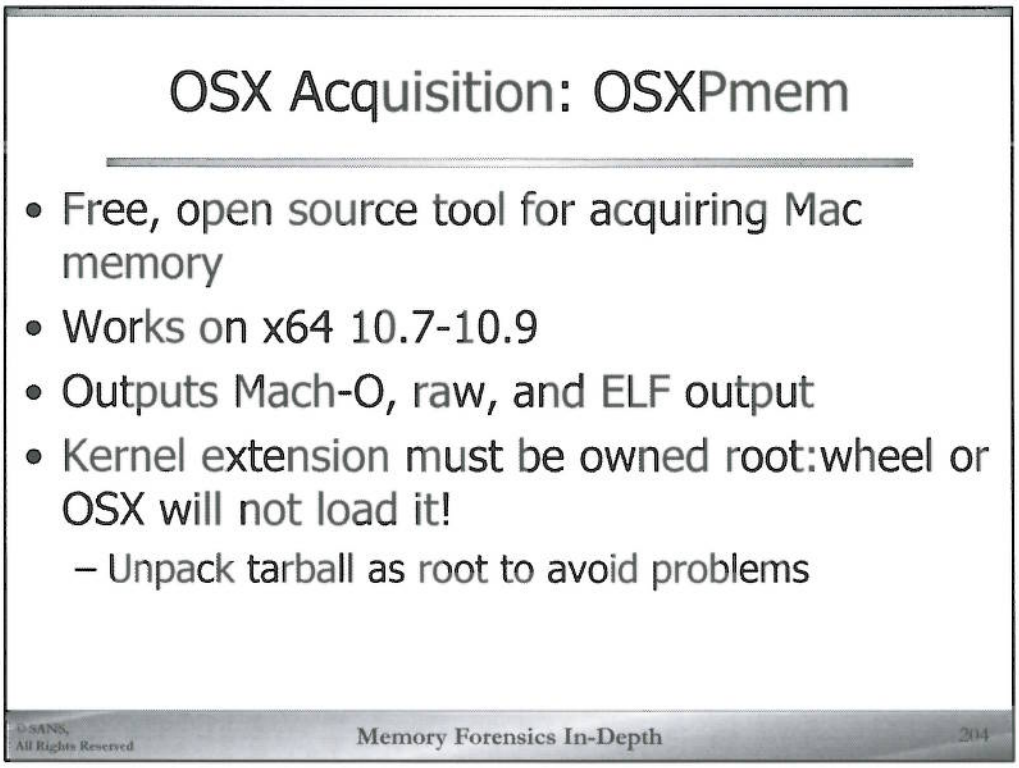


Mac Memoryze is a free tool provided by Mandiant. Some may be familiar with operating Memoryze on Windows (it is also the audit engine used in Redline).

Can acquire memory or analyze an existing memory dump file, however the memory dump file must be in the raw format (this is also the only format it supports for output). The tool supports OSX 10.6 – 10.8. Unfortunately, the tool's analysis capabilities not as robust as some other tools.

Source:

https://www.mandiant.com/resources/download/mac-memoryze

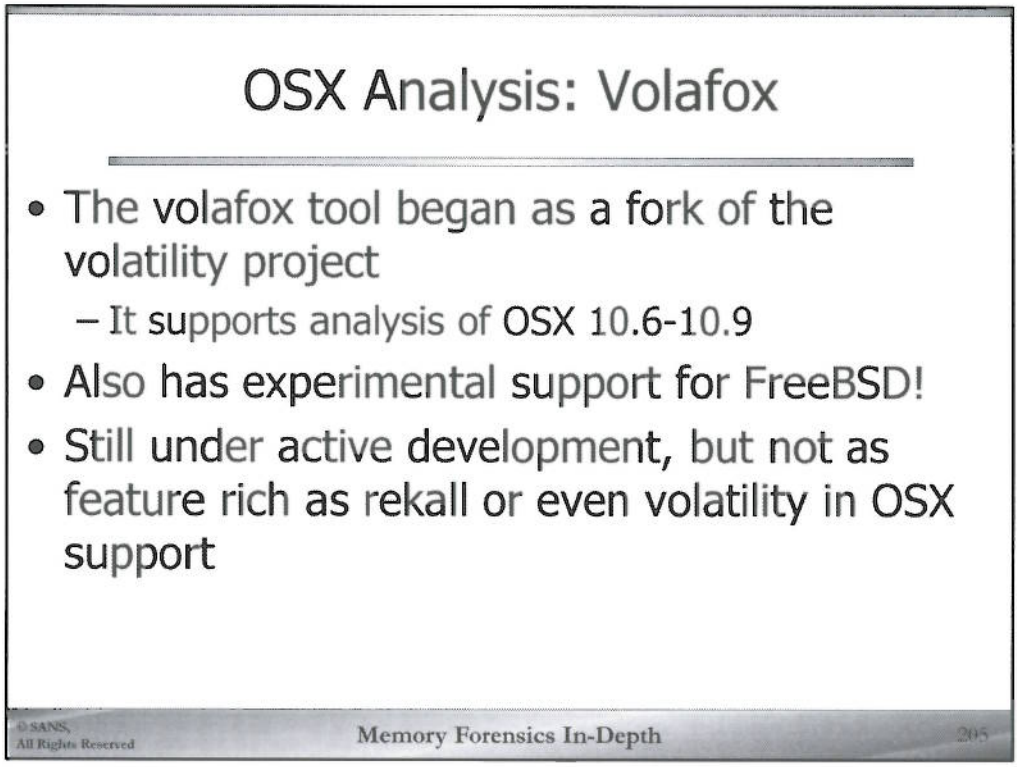


OSXPmem is a free and open source tool for acquiring Mac memory. It works on 10.7-10.9 (x64 only). The authors report that it may work on 10.6, but may suffer some stability issues. However, because it must load a kernel module, you probably don't want to risk it. Use a tool that officially supports OSX memory dumps on 10.6 if acquiring from that platform. OSXPmem supports Mach-O, raw, and ELF output.

Since the software is distributed as a tarball, note that it may be unpacked as a non-root user. However, the kernel extension will only be loaded if it is owned by the root user and the wheel group. Otherwise, it will display an error. Switch to the root user before unpacking the tarball and all the permissions will be correct.

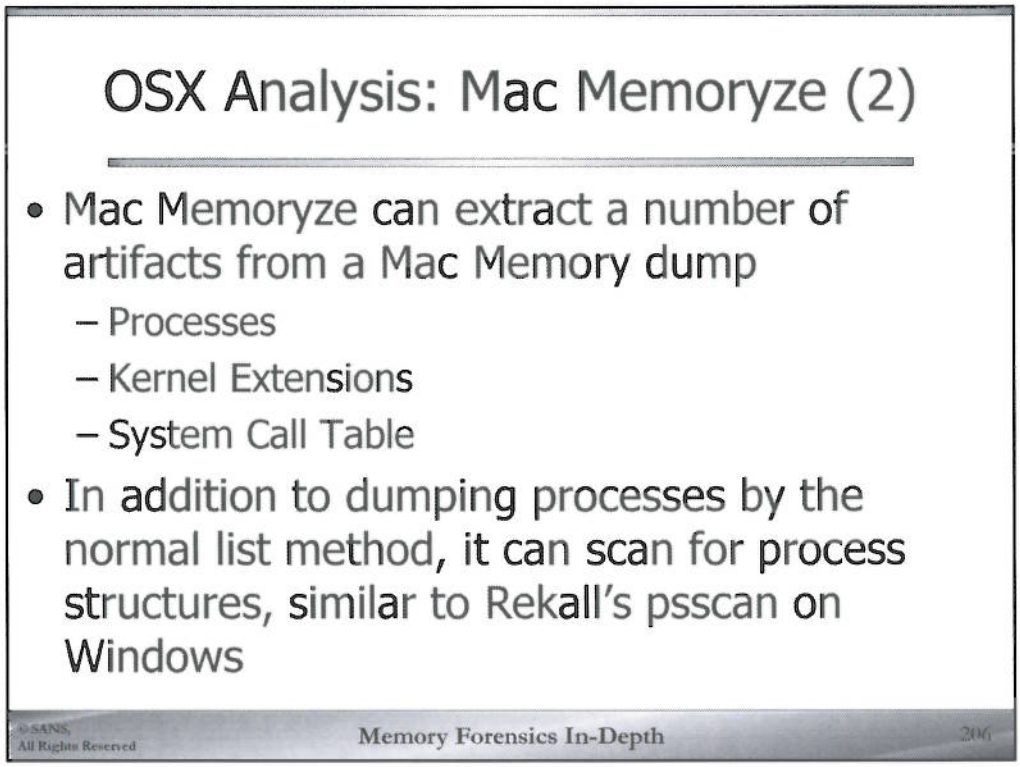
Source:

http://www.rekall-forensic.com



The volafox tool began as a fork of the volatility project. It supports analysis of OSX 10.6-10.9, but requires that files be raw memory formats. Volafox is the only memory analysis tool that supports FreeBSD. Although volafox is still under active development, but not as feature rich as rekall or even volatility in OSX support.

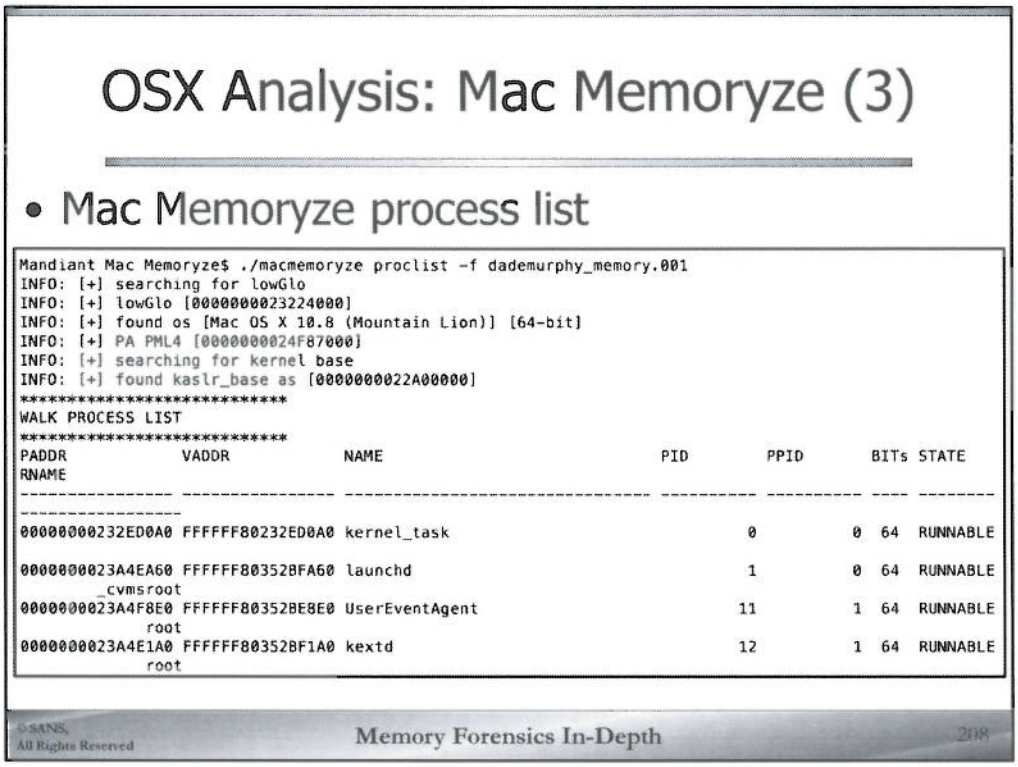
Source: https://code.google.com/p/volafox/



Mac Memoryze can extract a number of artifacts from a Mac Memory dump. These include processes, kernel extensions, and the system call table. In addition to dumping processes by the normal list method, it can scan for process structures, similar to Rekall's psscan on Windows. The command line usage for the tool is shown below. Unfortunately, the tool can only be run on OSX.

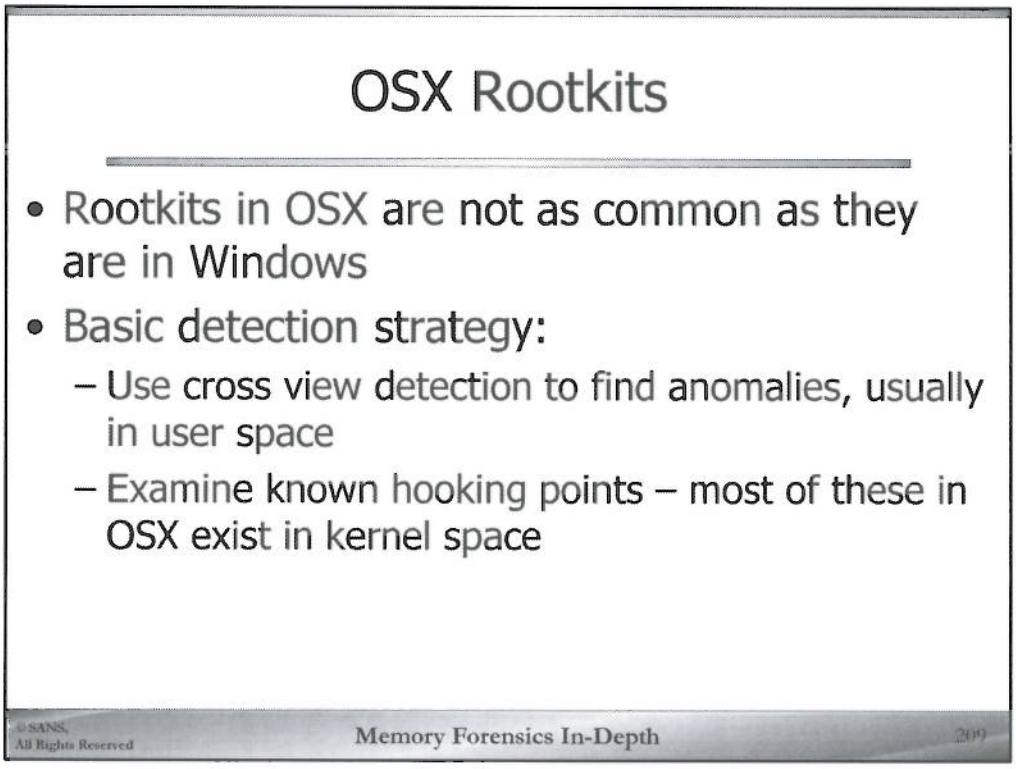
```
Mandiant Mac Memoryze$ ./macmemoryze -h
Mandiant Mac Memoryze v1.0
Copyright Mandaint 2012

usage: macmemoryze [dump|proclist|kextlist|syscalllist] options
        -h              help screen
        -f filename     previously dumpped memory (otherwise uses physical memory and driver)
        -x              xml output dump
        -q              quite (dont display % complete)
        -f              name of file to dump to proclist
        -w              parse process file handles with process
        -s              parse process section info with process
        -t              dump process sections [requires -s option]
        -c              carve processes (dont walk list)
        -r              walk mach task list
        -p pid          pid to process
```

-n name	name of process to process kextlist
-c	carve kexts from memory syscalllist
-s	syscall table
-m	mach_trap table



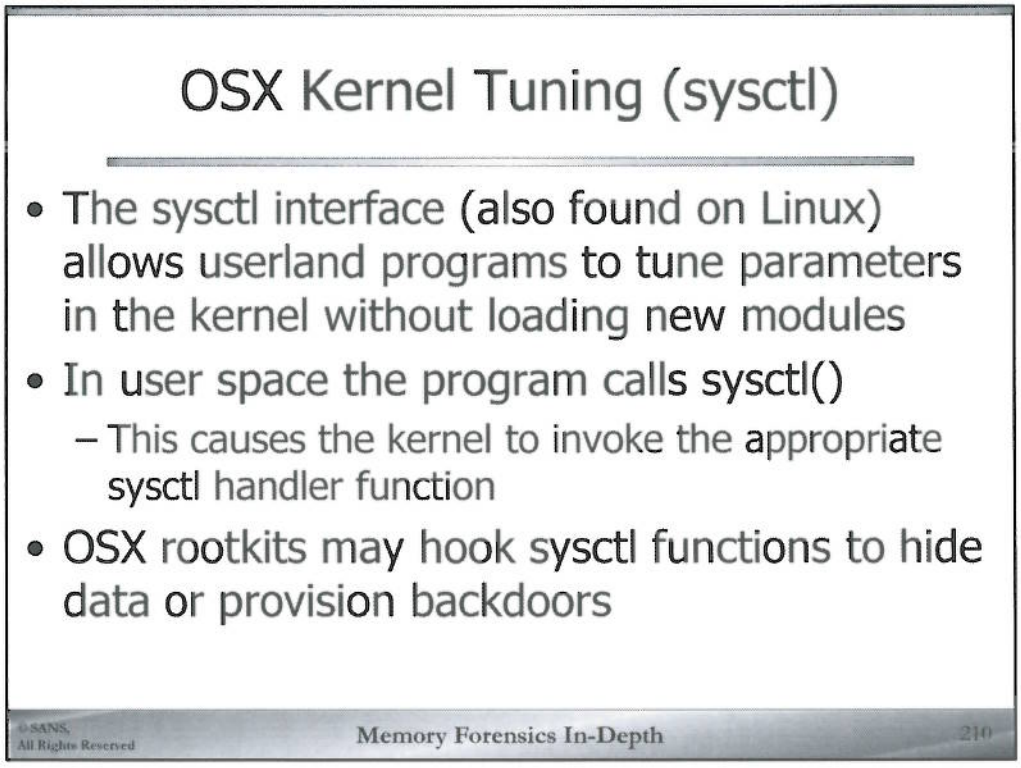
The output below shows a partial process listing obtained with Mac Memoryze.

```
Mandiant Mac Memoryze$ ./macmemoryze proclist -f dademurphy_memory.001
INFO: [+] searching for lowGlo
INFO: [+] lowGlo [0000000023224000]
INFO: [+] found os [Mac OS X 10.8 (Mountain Lion)] [64-bit]
INFO: [+] PA PML4 [0000000024F87000]
INFO: [+] searching for kernel base
INFO: [+] found kaslr_base as [0000000022A00000]
*****************************
WALK PROCESS LIST
*****************************
PADDR       VADDR       NAME                 PID   PPID  BITs STATE    STARTED              EUID  RUID  USERNAME
----------- ----------- -------------------- ----- ----- ---- -------- -------------------- ----- ----- --------------------
00000000232ED0A0 FFFFFF80232ED0A0 kernel_task      0     0   64 RUNNABLE 2013-12-22 21:36:27     0     0
0000000023A4EA60 FFFFFF80352BFA60 launchd          1     0   64 RUNNABLE 2013-12-22 21:36:27     0     0 _cvmsroot
0000000023A4F8E0 FFFFFF80352BE8E0 UserEventAgent  11     1   64 RUNNABLE 2013-12-22 21:36:28     0     0 root
0000000023A4E1A0 FFFFFF80352BF1A0 kextd           12     1   64 RUNNABLE 2013-12-22 21:36:28     0     0 root
0000000023A4F480 FFFFFF80352BE480 notifyd         14     1   64 RUNNABLE 2013-12-22 21:36:28     0     0 root
0000000023A4F020 FFFFFF80352BE020 securityd       15     1   64 RUNNABLE 2013-12-22 21:36:28     0     0 root
0000000023A50BC0 FFFFFF80352BDBC0 diskarbitrationd 16    1   64 RUNNABLE 2013-12-22 21:36:28     0     0 root
... truncated ...
```



Rootkits in OSX are not as common as they are in Windows. The basic strategy for detecting rootkits in OSX is similar to that in Windows.

First, use cross view detection to find anomalies. These anomalies are usually most readily apparent in user space. Second, investigators should use tools to examine known hooking points. In OSX, these are most often found in kernel space.

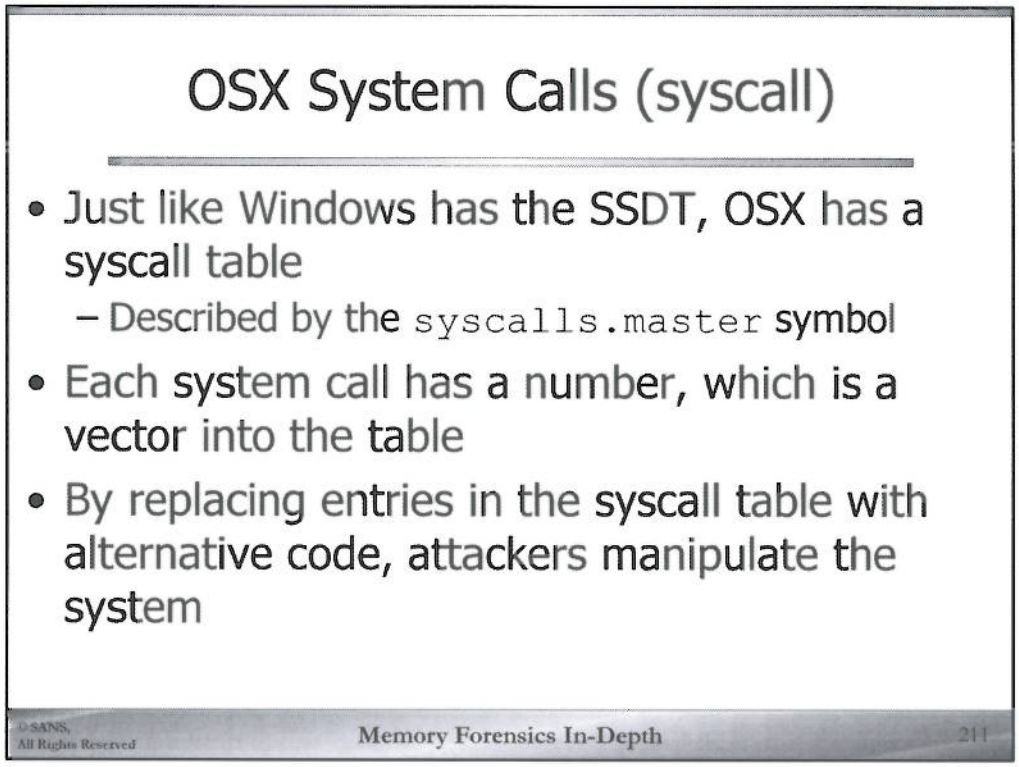


The sysctl interface (also found on Linux) allows userland programs to tune parameters in the kernel without loading new modules. Because kernel modules are specific to the running kernel, this greatly simplifies the operation of the system. It also increases stability and simplifies debugging since kernel tuning uses an existing, well tested interface.

In user space the program calls sysctl(). This causes the kernel to invoke the appropriate sysctl handler function from a table stored in the kernel. The sysctl handler may return data to the user.

OSX rootkits may hook sysctl functions to hide data or provision backdoors. Since unprivileged users may be able to read sysctl values, a read request also invokes the handler. The handler then executes code in the kernel, which could be used to elevate an unprivileged process to root. Remember: the kernel operates in "God mode", so once an attacker has code running in kernel space, the sky is the limit.

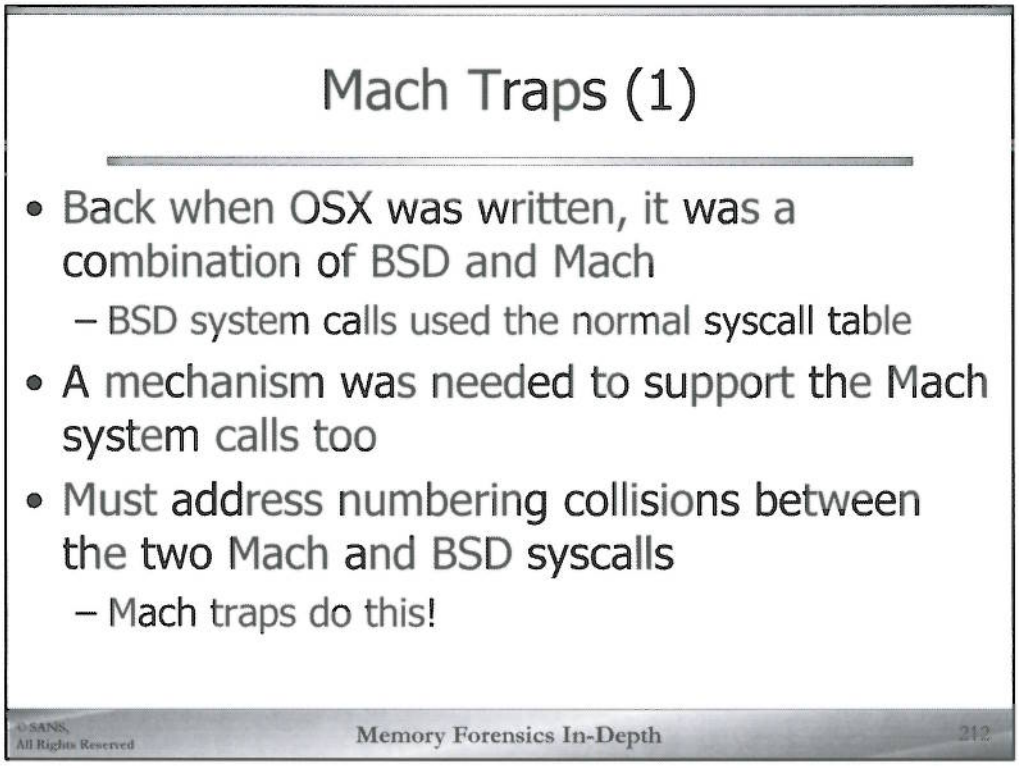
Source:
https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/boundaries/boundaries.html



Just like Windows has the SSDT, OSX has a syscall table. This table is described by the `syscalls.master` symbol in the kernel. This table contains the functions called when kernel functionality (e.g. system calls) are requested from user space.

Each system call has a number, which is a vector into the table. By replacing entries in the syscall table with alternative code, attackers manipulate the system. Some example system calls that an attacker might hook would be AUE_VFORK and AUE_KILL (creation and termination of processes respectively).

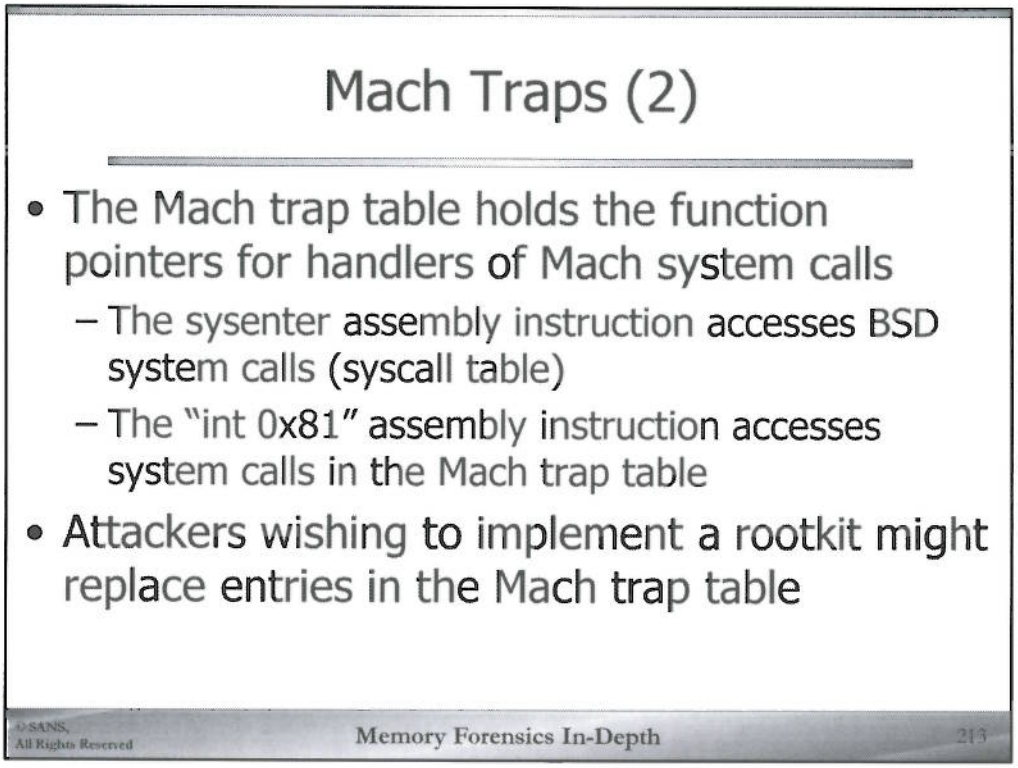
Sources: www.opensource.apple.com/source/xnu/xnu-1504.3.12/bsd/kern/syscalls.master



Back when OSX was written, it was a combination of BSD and Mach OS. BSD system calls used the normal syscall table, but a mechanism was needed to support the Mach system calls too. Because each OS was developed separately, there were significant numbering collisions between the each OS's system calls.

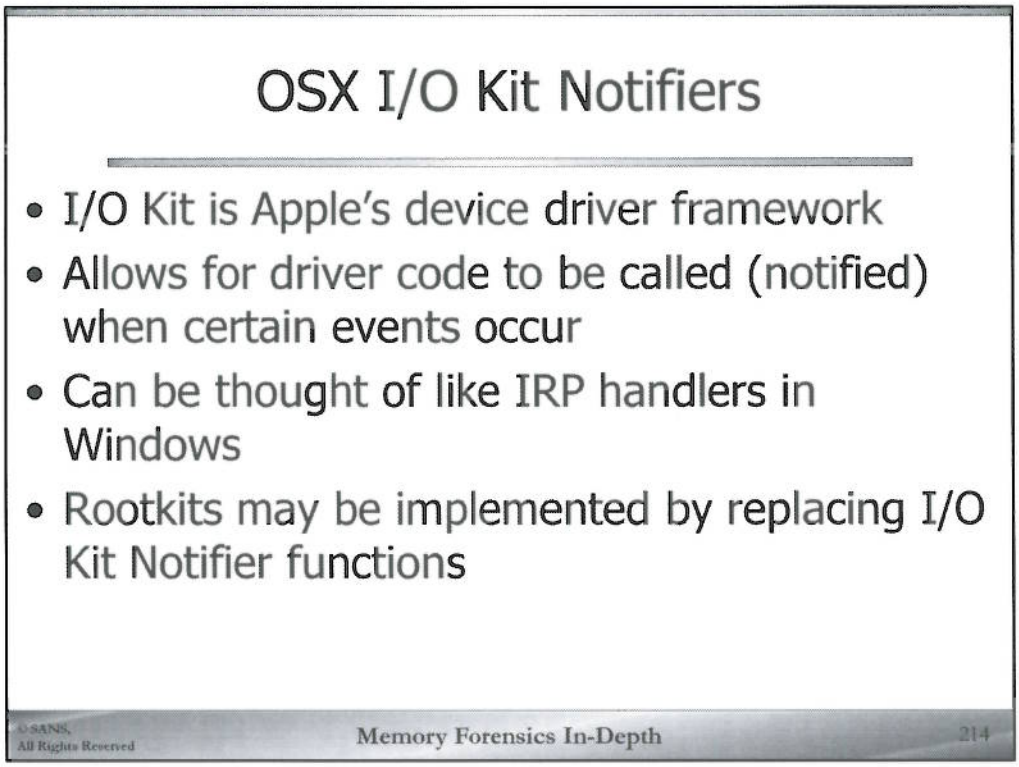
A scheme was needed to address numbering collisions between the two Mach and BSD syscalls. The scheme developed was Mach traps.

Source: Abusing Mach on OSX



The Mach trap table holds the function pointers for handlers of Mach system calls. The sysenter assembly instruction accesses BSD system calls (these are stored in the syscall table). The "int 0x81" assembly instruction accesses system calls in the Mach trap table. It is unclear what percentage of modern code actually uses the Mach traps as opposed to the BSD system calls. However, attackers wishing to implement a rootkit might replace entries in the Mach trap table. In fact, failing to do so might lead to avenues for cross view detection if the syscall table were already hooked.

Source: Abusing Mach on OSX

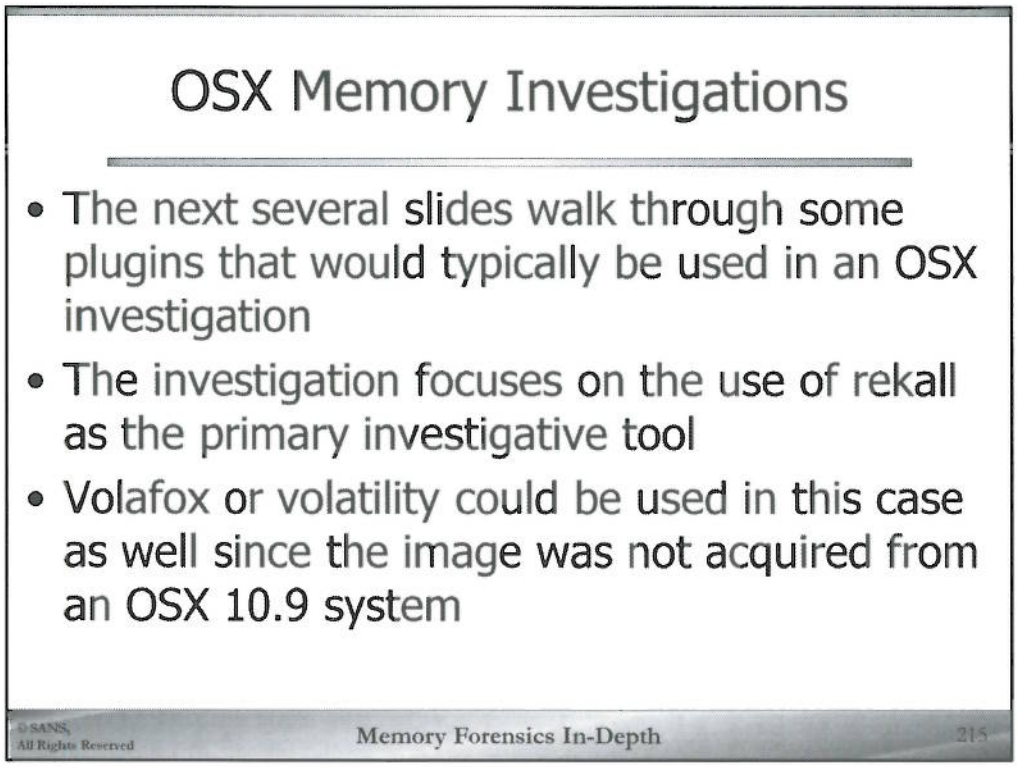


I/O Kit is Apple's device driver framework. It allows for driver code to be called (notified) when certain events occur. For instance, it might notify the USB device driver when new hardware is inserted in a USB port.

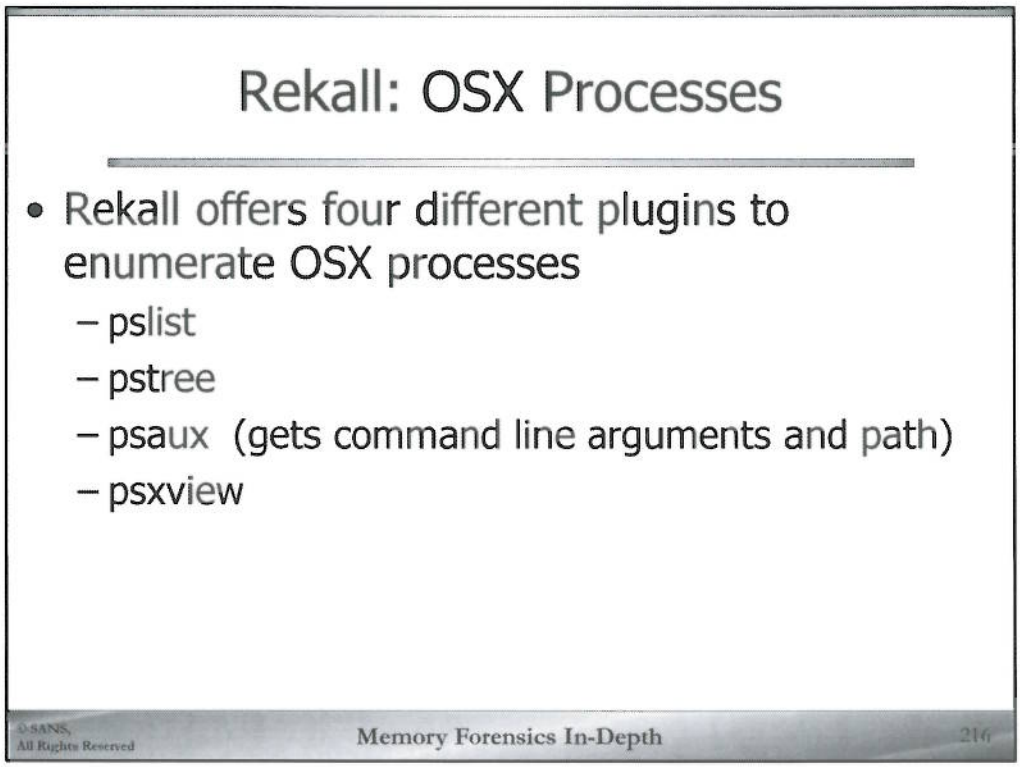
You can think of I/O Kit notifiers like IRP handlers in Windows. Both allow for asynchronous operation. Note that rootkits may be implemented by replacing I/O Kit Notifier functions. One such use might be to register for notifications when the system is going to sleep (suspend to disk). This would allow the malware to remove portions of its code before the system suspends. The rootkit could then reinsert itself by having registered for a notifier for the system resume event.

Source:

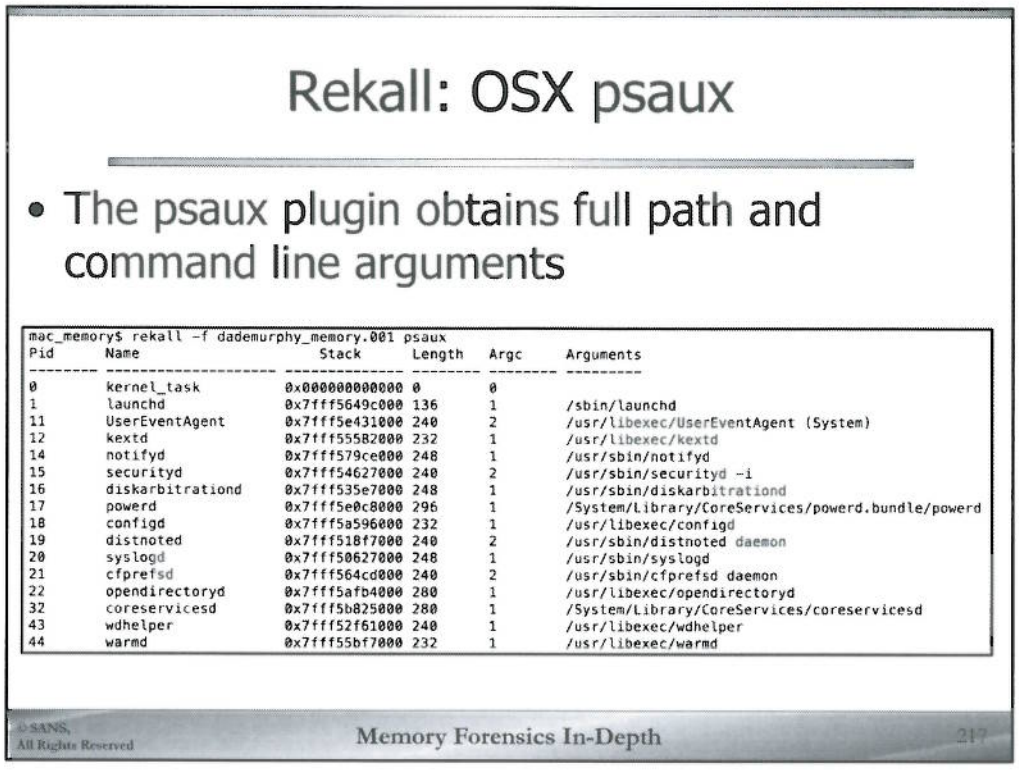
https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/services/services.html



The next several slides walk through some plugins that would typically be used in an OSX investigation. The investigation focuses on the use of rekall as the primary investigative tool. Volafox or volatility could be used in this case as well since the image was not acquired from an OSX 10.9 system.

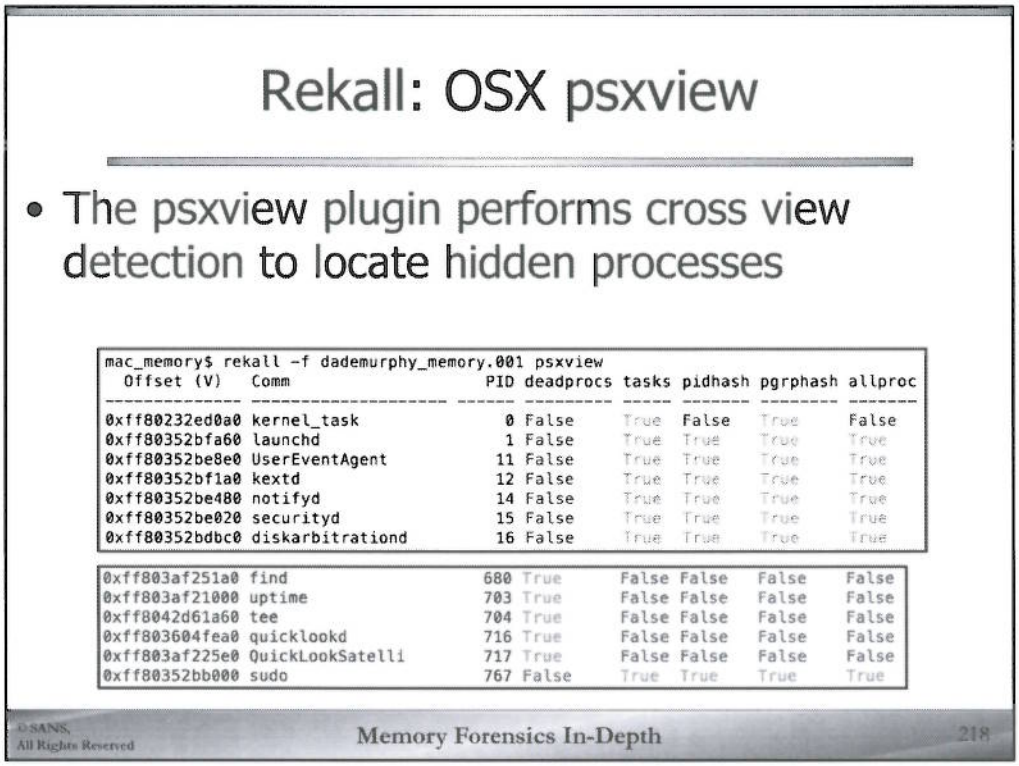


Rekall offers four different plugins for enumerating OSX processes. Most should be fairly obvious, but psaux obtains the process list and also gets the command line arguments and full path to the process. This is particularly useful for verifying that malware is not camouflaged using a familiar process name. The pslist, pstree, and psxview plugins operate as they do on Windows.



The display above shows the partial output of examining OSX memory with rekall using the psaux command. Output below is truncated:

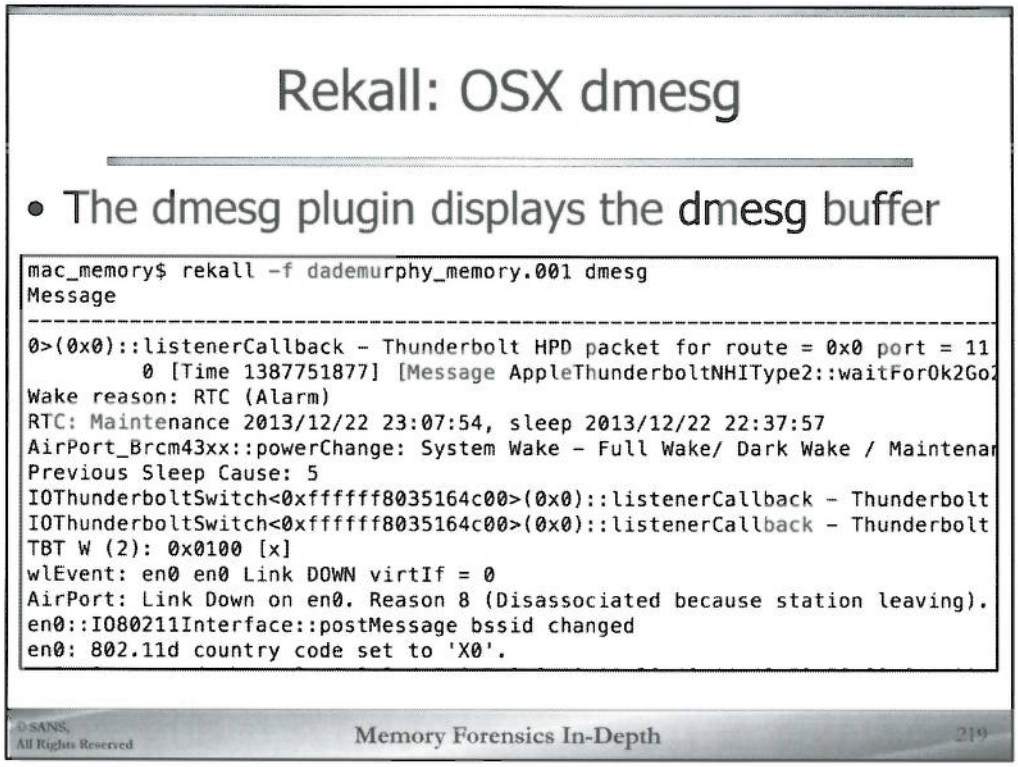
```
mac_memory$ rekall -f dademurphy_memory.001 psaux

Pid   Name              Stack           Length  Argc  Arguments
---   ----              -----           ------  ----  ---------
0     kernel_task       0x000000000000  0       0
1     launchd           0x7fff5649c000  136     1     /sbin/launchd
11    UserEventAgent    0x7fff5e431000  240     2     /usr/libexec/UserEventAgent (System)
12    kextd             0x7fff55582000  232     1     /usr/libexec/kextd
14    notifyd           0x7fff579ce000  248     1     /usr/sbin/notifyd
15    securityd         0x7fff54627000  240     2     /usr/sbin/securityd -i
16    diskarbitrationd  0x7fff535e7000  248     1     /usr/sbin/diskarbitrationd
17    powerd            0x7fff5e0c8000  296     1     /System/Library/CoreServices/powerd.bundle/powerd
18    configd           0x7fff5a596000  232     1     /usr/libexec/configd
19    distnoted         0x7fff518f7000  240     2     /usr/sbin/distnoted daemon
20    syslogd           0x7fff50627000  248     1     /usr/sbin/syslogd
21    cfprefsd          0x7fff564cd000  240     2     /usr/sbin/cfprefsd daemon
22    opendirectoryd    0x7fff5afb4000  280     1     /usr/libexec/opendirectoryd
32    coreservicesd     0x7fff5b825000  280     1     /System/Library/CoreServices/coreservicesd
43    wdhelper          0x7fff52f61000  240     1     /usr/libexec/wdhelper
44    warmd             0x7fff55bf7000  232     1     /usr/libexec/warmd
45    usbmuxd           0x7fff5fc00000  504     2     /System/Library/PrivateFrameworks/MobileDevice.framework/Versions/A/Resources/usbmuxd -launchd
48    stackshot         0x7fff55d0e000  224     2     /usr/libexec/stackshot -t
49    SleepServicesD    0x7fff5f41e000  280     1     /System/Library/CoreServices/SleepServicesD
51    revisiond         0x7fff59ba1000  376     1     /System/Library/PrivateFrameworks/GenerationalStorage.framework/Versions/A/Support/revisiond
... truncated output ...
```



The above output shows rekall's psxview plugin run on OSX memory. Notice that those items found in deadprocs should not be expected to be detected by the other methods.

Raw output from the psxview command below (truncated for space):

```
mac_memory$ rekall -f dademurphy_memory.001 psxview
 Offset (V)  Comm                  PID   deadprocs tasks   pidhash  pgrphash allproc
-------------- -------------------- ------ --------- -----   -------   --------  -------
0xff80232ed0a0 kernel_task           0     False     True    False     True      False
0xff80352bfa60 launchd               1     False     True    True      True      True
0xff80352bc8c0 UserEventAgent        11    False     True    True      True      True
0xff80352bf1a0 kextd                 12    False     True    True      True      True
0xff80352be480 notifyd               14    False     True    True      True      True
0xff80352be020 securityd             15    False     True    True      True      True
0xff80352bdbc0 diskarbitrationd      16    False     True    True      True      True
0xff80352bd760 powerd                17    False     True    True      True      True
... truncated ...
0xff803af21000 uptime                703   True      False   False     False     False
0xff8042d61a60 tee                   704   True      False   False     False     False
0xff803604fea0 quicklookd            716   True      False   False     False     False
0xff803af225e0 QuickLookSatelli      717   True      False   False     False     False
0xff80352bb000 sudo                  767   False     True    True      True      True
... truncated ...
```



The slide above shows the (truncated) output of Rekall's dmesg plugin. This plugin obtains the contents of the dmesg buffer, which stores boot logging data. This can be particularly useful if an attacker has modified data in the logs on disk to remove evidence that a kernel extension (device driver) was loaded.

Truncated output of the dmesg plugin shown below:

mac_memory$ rekall -f dademurphy_memory.001 dmesg

Message

0>(0x0)::listenerCallback - Thunderbolt HPD packet for route = 0x0 port = 11 unplug = 0

0 [Time 1387751877] [Message AppleThunderboltNHIType2::waitForOk2Go2Sx - retries = 2

Wake reason: RTC (Alarm)

RTC: Maintenance 2013/12/22 23:07:54, sleep 2013/12/22 22:37:57

AirPort_Brcm43xx::powerChange: System Wake - Full Wake/ Dark Wake / Maintenance wake

Previous Sleep Cause: 5

IOThunderboltSwitch<0xffffff8035164c00>(0x0)::listenerCallback - Thunderbolt HPD packet for route = 0x0 port = 11 unplug = 0

IOThunderboltSwitch<0xffffff8035164c00>(0x0)::listenerCallback - Thunderbolt HPD packet for route = 0x0 port = 12 unplug = 0

TBT W (2): 0x0100 [x]

wlEvent: en0 en0 Link DOWN virtIf = 0

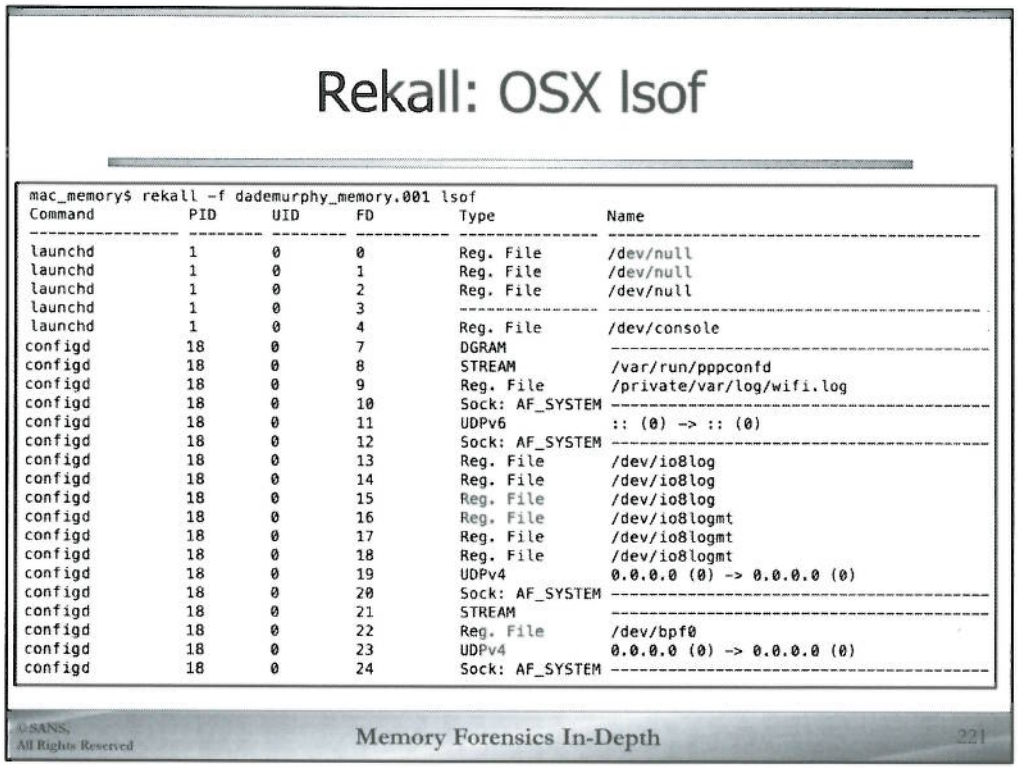
AirPort: Link Down on en0. Reason 8 (Disassociated because station leaving).

en0::IO80211Interface::postMessage bssid changed

en0: 802.11d country code set to 'X0'.

en0: Supported channels 1 2 3 4 5 6 7 8 9 10 11 36 40 44 48 52 56 60 64 100 104 108 112 116 120 124 128 132 136 140 149 153 157 161 165

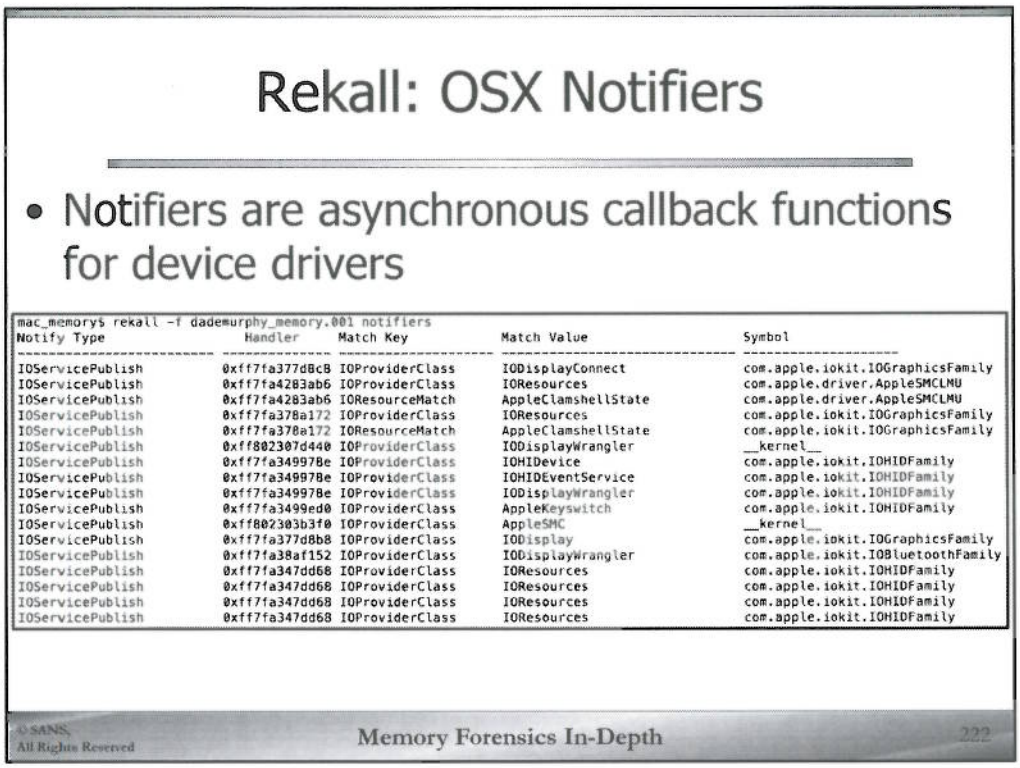
Graphics suppressed 181 ms
AppleUSBMultitouchDriver::checkStatus - received Status Packet, Payload 2: device was reinitialized
en0: 802.11d country code set to 'US'.
en0: Supported channels 1 2 3 4 5 6 7 8 9 10 11 36 40 44 48 52 56 60 64 100 104 108 112 116 120 124 128 132 136 140 149 153 157 161 165
MacAuthEvent en0 Auth result for: c8:3a:35:f3:06:98 MAC AUTH succeeded
wlEvent: en0 en0 Link UP virtIf = 0
AirPort: Link Up on en0
… truncated …



Rekall's lsof plugin can be used to enumerate file handles that are currently open in the operating system. This is especially useful if a suspicious process is found and the investigator wants to know what files it is using. For instance, a malicious process might have a copy of its configuration file open. Alternatively, it might have exfiltration files open. Also note that network sockets will also be visible in the lsof plugin output.

Truncated output from the lsof plugin is shown below:

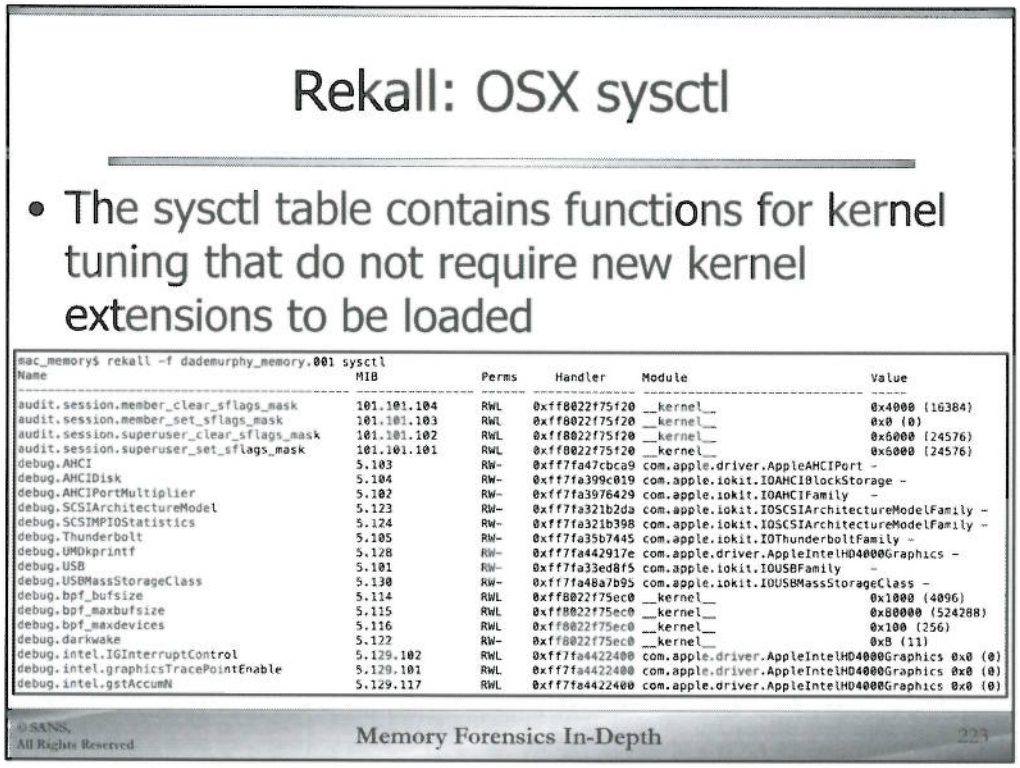
```
mac_memory$ rekall -f dademurphy_memory.001 lsof
Command        PID    UID    FD      Type          Name
-------------- -------- -------- ---------- --------------- ----------------------------------------
launchd        1      0      0       Reg. File     /dev/null
launchd        1      0      1       Reg. File     /dev/null
launchd        1      0      2       Reg. File     /dev/null
launchd        1      0      3       --------------- ----------------------------------------
... truncated ...
```



The notifiers plugin identifies the I/O Kit notification functions that are registered in the kernel. Note that these are analogous to IRP handlers in Windows. Rekall identifies the type when known. Those with unknown types may warrant further investigation.

Truncated output shown below:

```
mac_memory$ rekall -f dademurphy_memory.001 notifiers
Notify Type            Handler    Match Key          Match Value              Symbol
---------------------- ---------- ------------------ ------------------------ --------------------
IOServicePublish       0xff7fa377d8c8 IOProviderClass    IODisplayConnect      com.apple.iokit.IOGraphicsFamily
IOServicePublish       0xff7fa4283ab6 IOProviderClass    IOResources           com.apple.driver.AppleSMCLMU
IOServicePublish       0xff7fa4283ab6 IOResourceMatch    AppleClamshellState   com.apple.driver.AppleSMCLMU
IOServicePublish       0xff7fa378a172 IOProviderClass    IOResources           com.apple.iokit.IOGraphicsFamily
IOServicePublish       0xff7fa378a172 IOResourceMatch    AppleClamshellState   com.apple.iokit.IOGraphicsFamily
IOServicePublish       0xff802307d440 IOProviderClass    IODisplayWrangler     __kernel__
IOServicePublish       0xff7fa349978c IOProviderClass    IOHIDevice            com.apple.iokit.IOHIDFamily
IOServicePublish       0xff7fa349978c IOProviderClass    IOHIDEventService     com.apple.iokit.IOHIDFamily
IOServicePublish       0xff7fa349978e IOProviderClass    IODisplayWrangler     com.apple.iokit.IOHIDFamily
IOServicePublish       0xff7fa3499ed0 IOProviderClass    AppleKeyswitch        com.apple.iokit.IOHIDFamily
... truncated ...
```

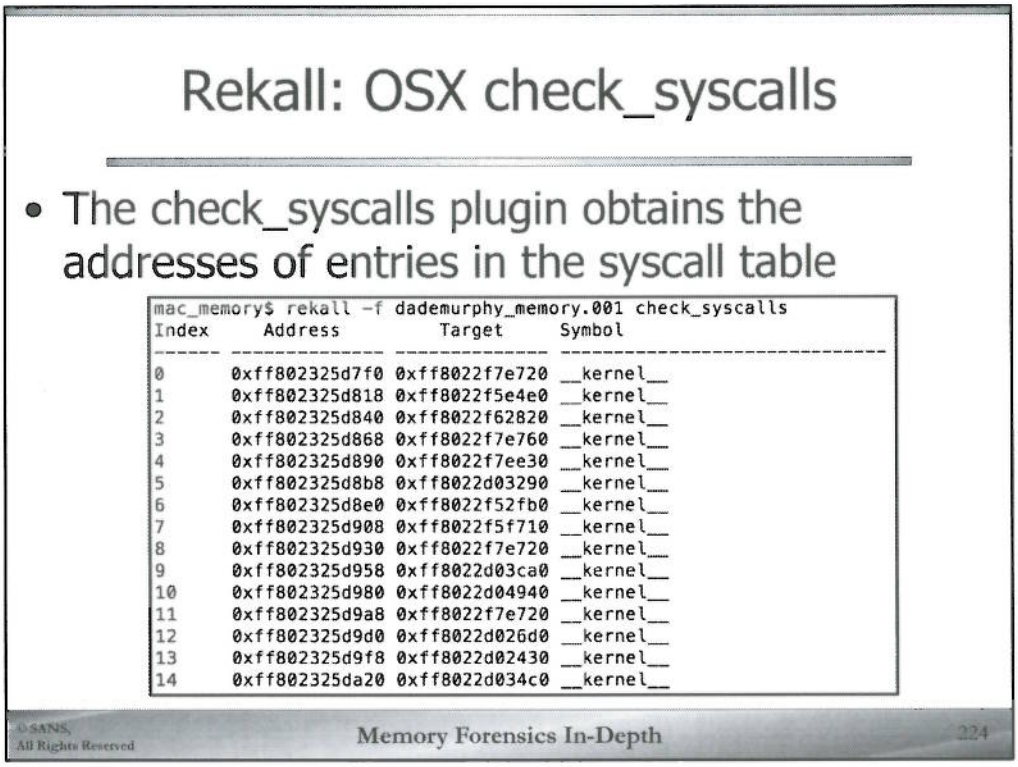


The sysctl table contains functions for kernel tuning that do not require new kernel extensions to be loaded. Attackers might overwrite functions in this table to provide backdoor capability.

Truncated output shown below:

```
mac_memory$ rekall -f dademurphy_memory.001 sysctl
Name                                        MIB             Perms    Handler      Module                                  Value
------------------------------------------- --------------- ------ -------------- --------------------------------------- -----
audit.session.member_clear_sflags_mask      101.101.104     RWL    0xff8022f75f20 __kernel__                              0x4000 (16384)
audit.session.member_set_sflags_mask        101.101.103     RWL    0xff8022f75f20 __kernel__                              0x0 (0)
audit.session.superuser_clear_sflags_mask   101.101.102     RWL    0xff8022f75f20 __kernel__                              0x6000 (24576)
audit.session.superuser_set_sflags_mask     101.101.101     RWL    0xff8022f75f20 __kernel__                              0x6000 (24576)
debug.AHCI                                  5.103           RW-    0xff7fa47cbca9 com.apple.driver.AppleAHCIPort -
debug.AHCIDisk                              5.104           RW-    0xff7fa399c019 com.apple.iokit.IOAHCIBlockStorage -
debug.AHCIPortMultiplier                    5.102           RW-    0xff7fa3976429 com.apple.iokit.IOAHCIFamily    -
debug.SCSIArchitectureModel                 5.123           RW-    0xff7fa321b2da com.apple.iokit.IOSCSIArchitectureModelFamily -
debug.SCSIMPIOStatistics                    5.124           RW-    0xff7fa321b398 com.apple.iokit.IOSCSIArchitectureModelFamily -
debug.Thunderbolt                           5.105           RW-    0xff7fa35b7445 com.apple.iokit.IOThunderboltFamily -
debug.UMDkprintf                            5.128           RW-    0xff7fa442917e com.apple.driver.AppleIntelHD4000Graphics -
debug.USB                                   5.101           RW-    0xff7fa33ed8f5 com.apple.iokit.IOUSBFamily     -
debug.USBMassStorageClass                   5.130           RW-    0xff7fa48a7b95 com.apple.iokit.IOUSBMassStorageClass -
debug.bpf_bufsize                           5.114           RWL    0xff8022f75ec0 __kernel__                              0x1000 (4096)
debug.bpf_maxbufsize                        5.115           RWL    0xff8022f75ec0 __kernel__                              0x80000 (524288)
debug.bpf_maxdevices                        5.116           RWL    0xff8022f75ec0 __kernel__                              0x100 (256)
```

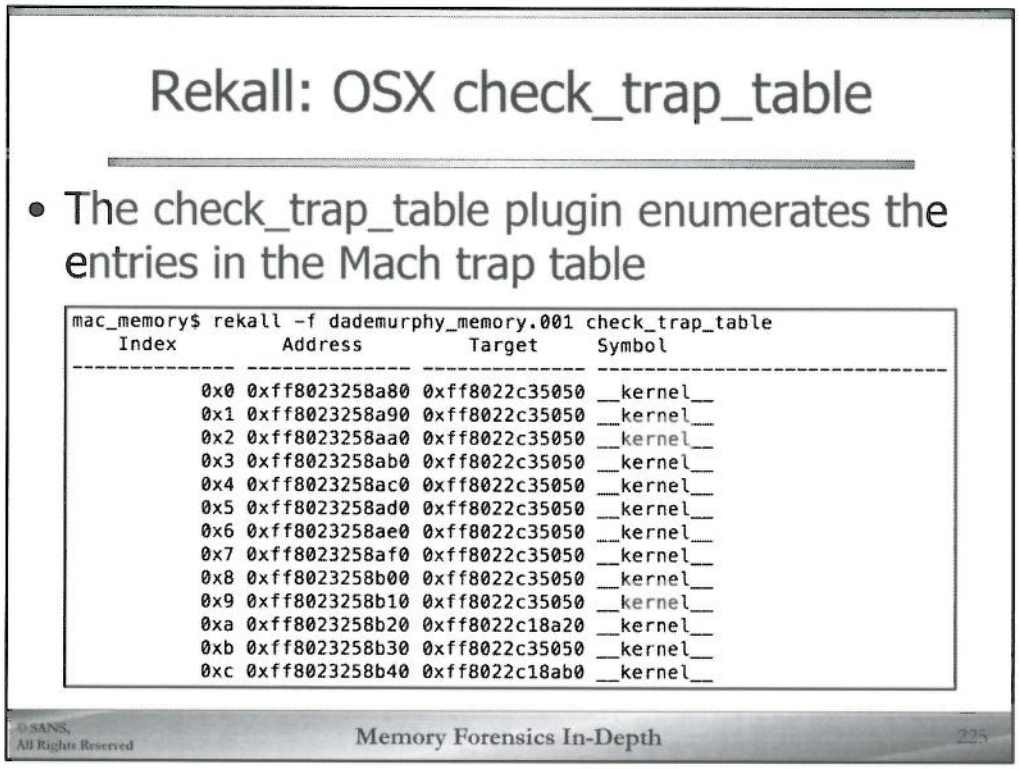
… truncated …



The check_syscalls plugin obtains the addresses of entries in the syscall table. This can be used to determine if any addresses are anomalous.

Truncated output shown below:

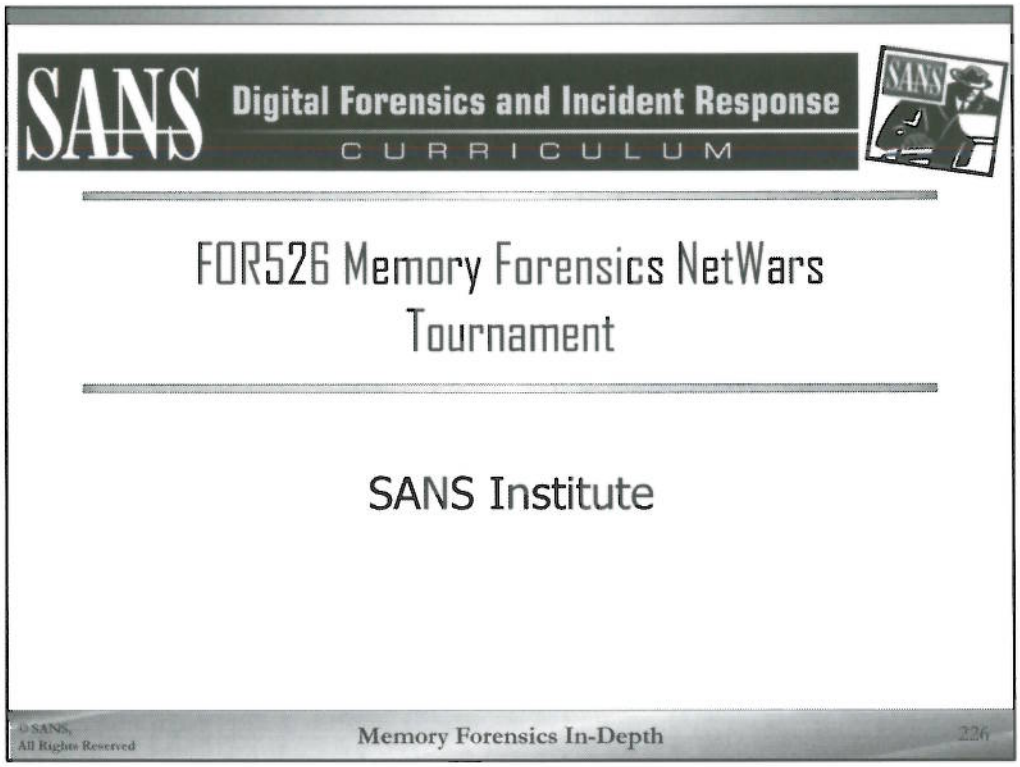
```
mac_memory$ rekall -f dademurphy_memory.001 check_syscalls
Index    Address        Target     Symbol
------ -------------- -------------- ------------------------------
0      0xff802325d7f0 0xff8022f7e720 __kernel__
1      0xff802325d818 0xff8022f5e4e0 __kernel__
2      0xff802325d840 0xff8022f62820 __kernel__
3      0xff802325d868 0xff8022f7e760 __kernel__
4      0xff802325d890 0xff8022f7ee30 __kernel__
5      0xff802325d8b8 0xff8022d03290 __kernel__
6      0xff802325d8e0 0xff8022f52fb0 __kernel__
7      0xff802325d908 0xff8022f5f710 __kernel__
8      0xff802325d930 0xff8022f7e720 __kernel__
9      0xff802325d958 0xff8022d03ca0 __kernel__
10     0xff802325d980 0xff8022d04940 __kernel__
11     0xff802325d9a8 0xff8022f7e720 __kernel__
... truncated ...
```



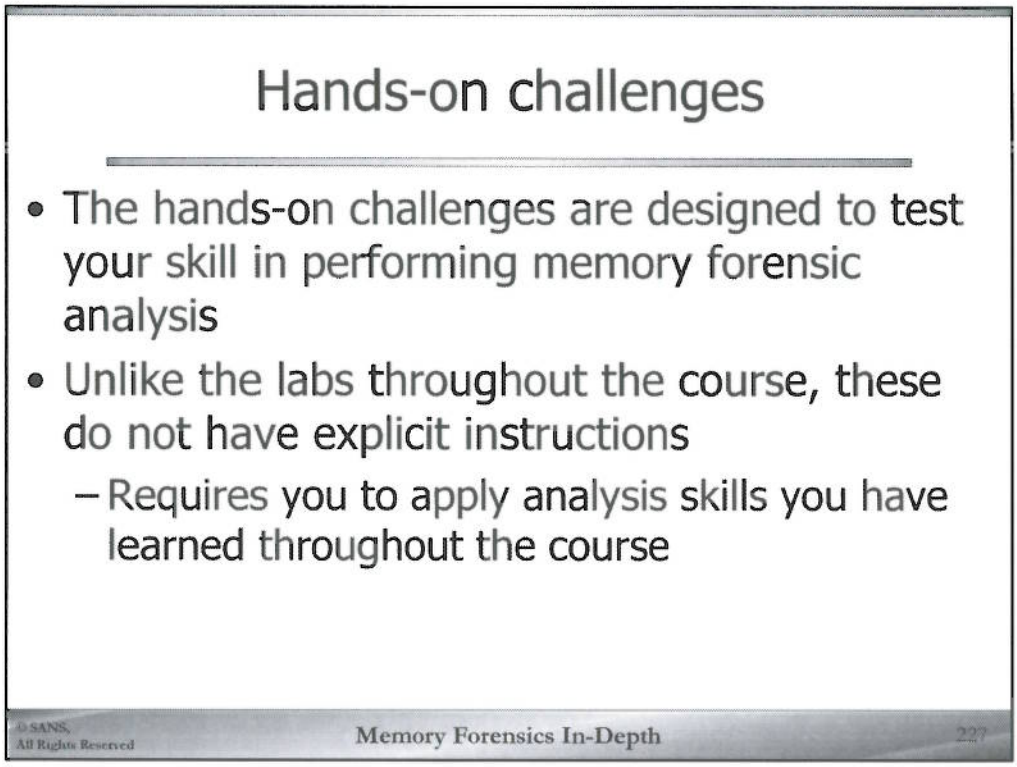
The check_trap_table plugin enumerates the entries in the Mach trap table. An attacker might overwrite entries in the Mach trap table to implement a rootkit. The Mach trap table is an alternate system call table used in OSX.

Truncated output shown below:

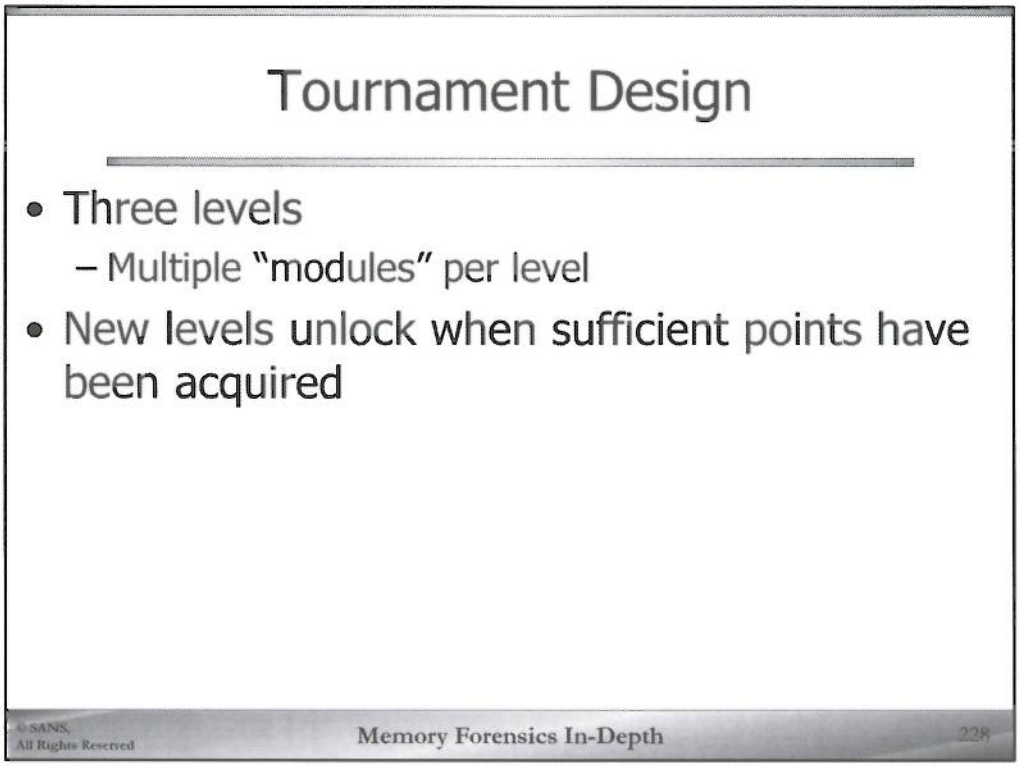
```
mac_memory$ rekall -f dademurphy_memory.001 check_trap_table
   Index        Address        Target     Symbol
-------------- -------------- -------------- ------------------------------
         0x0 0xff8023258a80 0xff8022c35050 __kernel__
         0x1 0xff8023258a90 0xff8022c35050 __kernel__
         0x2 0xff8023258aa0 0xff8022c35050 __kernel__
         0x3 0xff8023258ab0 0xff8022c35050 __kernel__
         0x4 0xff8023258ac0 0xff8022c35050 __kernel__
         0x5 0xff8023258ad0 0xff8022c35050 __kernel__
         0x6 0xff8023258ae0 0xff8022c35050 __kernel__
         0x7 0xff8023258af0 0xff8022c35050 __kernel__
         0x8 0xff8023258b00 0xff8022c35050 __kernel__
         0x9 0xff8023258b10 0xff8022c35050 __kernel__
         0xa 0xff8023258b20 0xff8022c18a20 __kernel__
         0xb 0xff8023258b30 0xff8022c35050 __kernel__
... truncated ...
```



This section assigns students to the role of a memory forensics analyst. You will be presented with a variety of hands-on questions involving real-world memory captures in the context of a challenging and fun tournament. These questions further your ability to respond to typical memory analysis tasks in an instructor-led lab environment and offer additional learning opportunities. Moreover, the questions are designed to reinforce skills covered earlier in the course. By applying the techniques learned earlier in the course, you can solidify your knowledge and shore up skill areas where you might need additional practice.

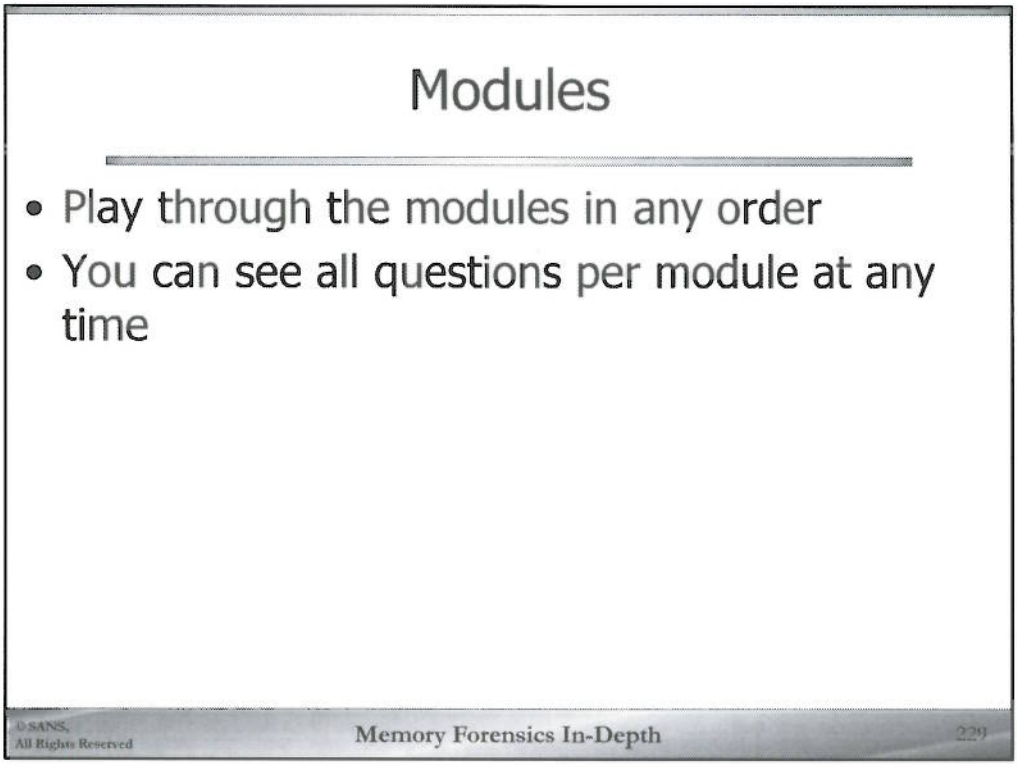


For the tournament, you will need to reference the files located on the course USB in the "netwars" directory. This folder is located off of the root of the USB drive.



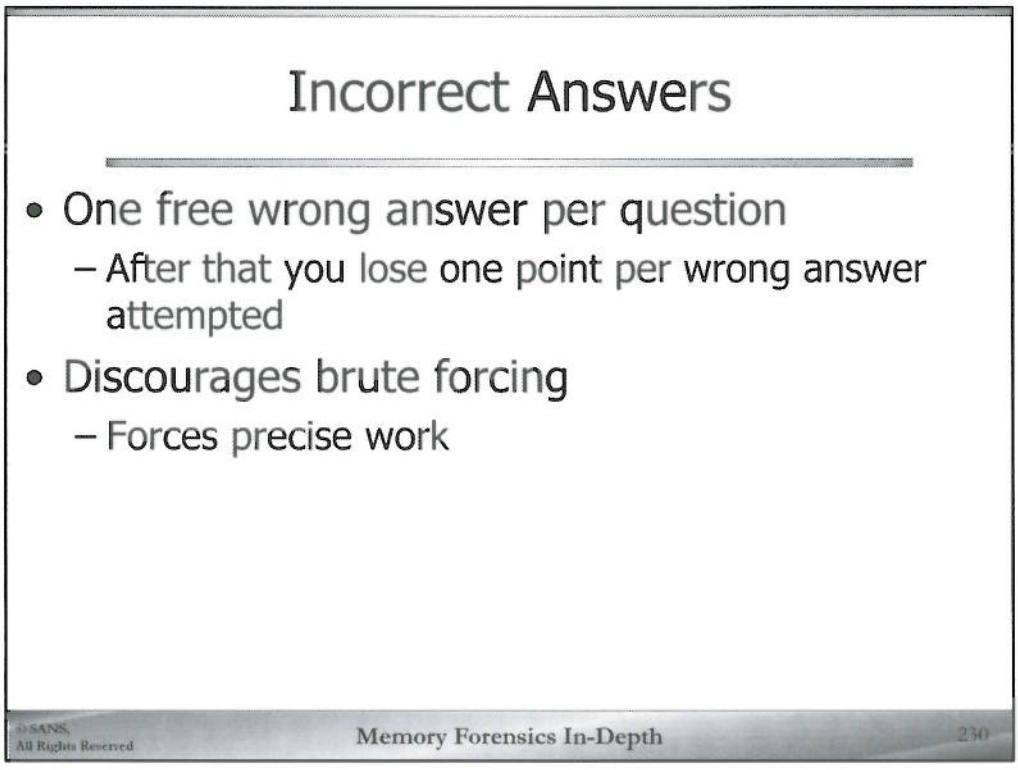
Once a sufficient number of points have been accumulated at level 1, new questions are unlocked at level 2. Participants may continue to answer questions at level 1, but have the option of answering level 2 questions as well. Level 3 questions will be unlocked when a sufficient point value has been acquired at level 2. Currently the tournament only has 3 levels.

Participants will continue solving questions until time expires or they have exhausted the supply of questions.



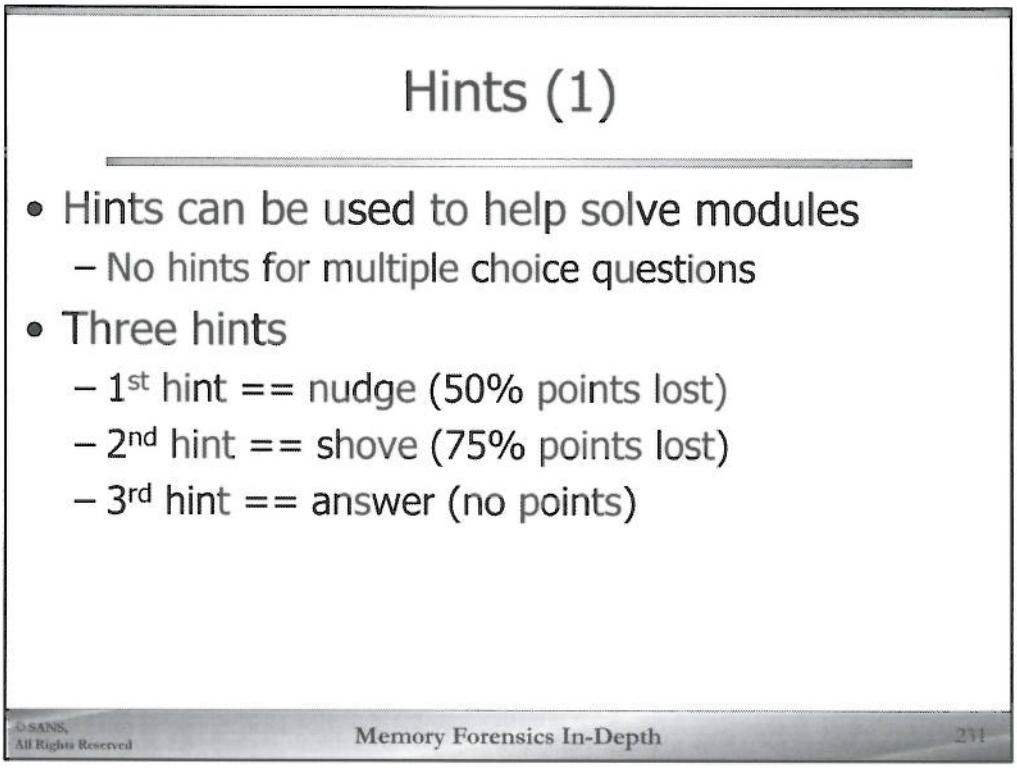
The tournament is comprised of a series of modules across several levels. When the tournament begins, you may access any of the modules on level 1. Each module is comprised of one or more questions.

If you get stuck, you can always change to another module and come back to the original module.



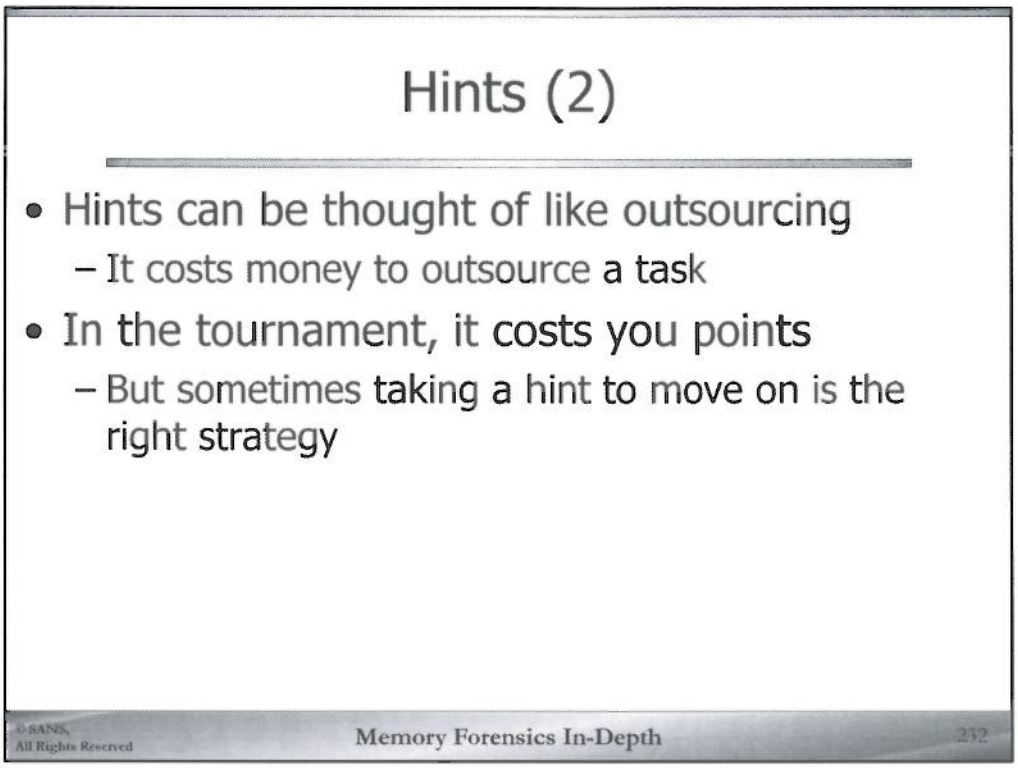
For each question, a player may answer incorrectly once without being assessed a penalty (a free guess). To discourage rampant guessing in the tournament, penalty points will be assessed for each subsequent incorrect guess on a per-question basis.

Besides discouraging brute force attempts to answer a question, these penalties serve another real world purpose. In the real world, it is unlikely to be discovered if you guess and put incorrect information in a report. If the error is discovered, there is a very real cost of discovery (QA). Also, incident response actions may be taken based on the findings of the malware analysis report. If these findings are incorrect, then there is a very real cost associated with the actions taken.



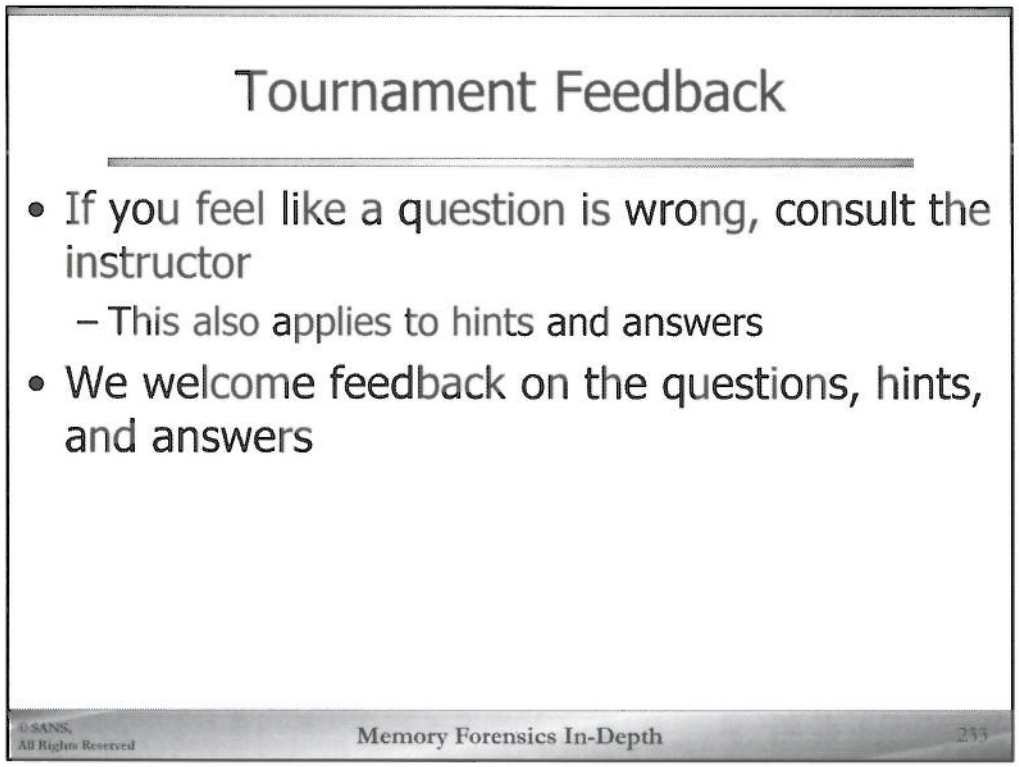
Most questions have two hints. The first hint is thought of as a nudge. It might point you to a tool, or generic class of tools, that can solve the problem. The second hint is more of a shove. These hints direct you to the answer without giving it to you. An example might be the exact syntax to solve the module, only requiring that you analyze the result.

If you use one of these hints, you can count on loss of 50% of the available points for the question. The second hint, if available, will take an additional 50% of the available points. Essentially, if you take both hints you can count on getting 25% of the total points for a given question. While this sounds very punitive, it was designed to be. When you perform memory analysis in an investigation, you don't receive any hints. If you need hints in the real world you pay a consultant for them. In this case, you are paying in the form of points.



In the real world, if a particular task is beyond your ability level (or that of your team) you would normally outsource the task to a contractor or consultant. This obviously costs money for your organization. Sometimes you outsource work not because you or your team can't handle the work, but because the task is not a good use of your time (in other words, the opportunity cost of performing the work outweighs the cost of outsourcing).

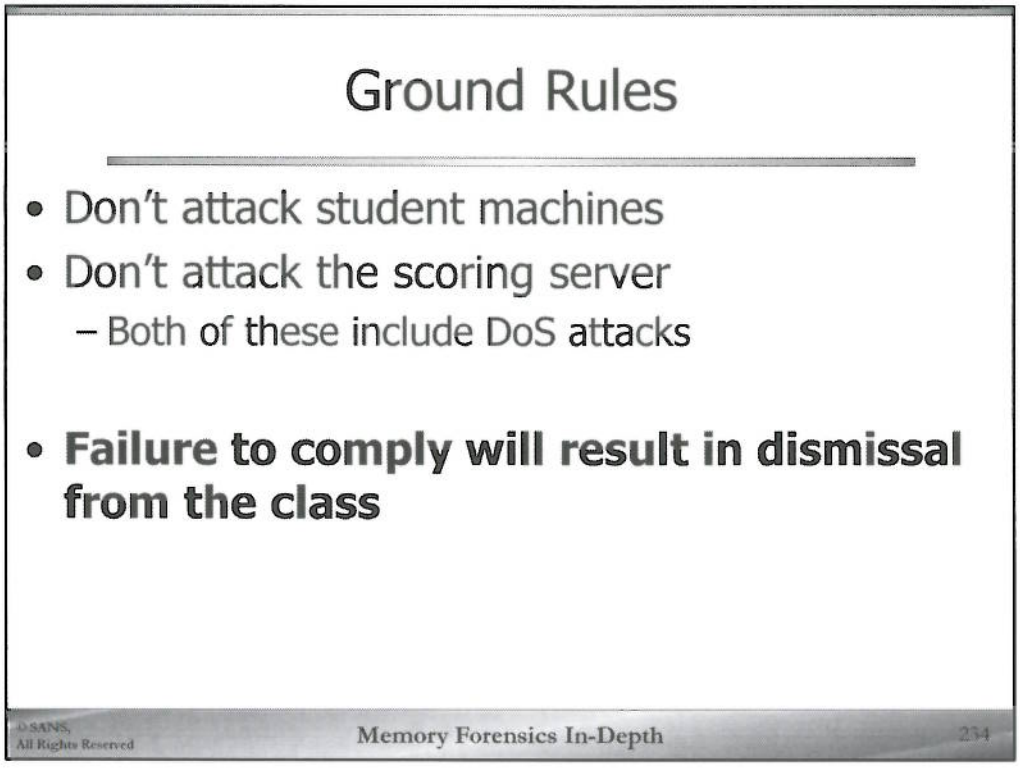
Hints cost a number of points in the tournament, the number of points is relative to the number of hints that are taken for a given question. After taking three hints, you will lose 100% of the points for the question.



Even if you don't agree with an answer, everyone else is playing the same tournament and we want to keep it fair. If you sincerely believe that a question is ambiguous, let the instructor know and they'll provide feedback to the course author. Specific feedback with wording suggestions is always most appreciated.

Although we will not make changes to the game in progress based on feedback, we will incorporate feedback into future iterations of the game at other events. This ensures everyone is playing the same tournament.

Note: If you're taking this course Online, consult the SANS Online SME for assistance <online-sme@sans.org>.

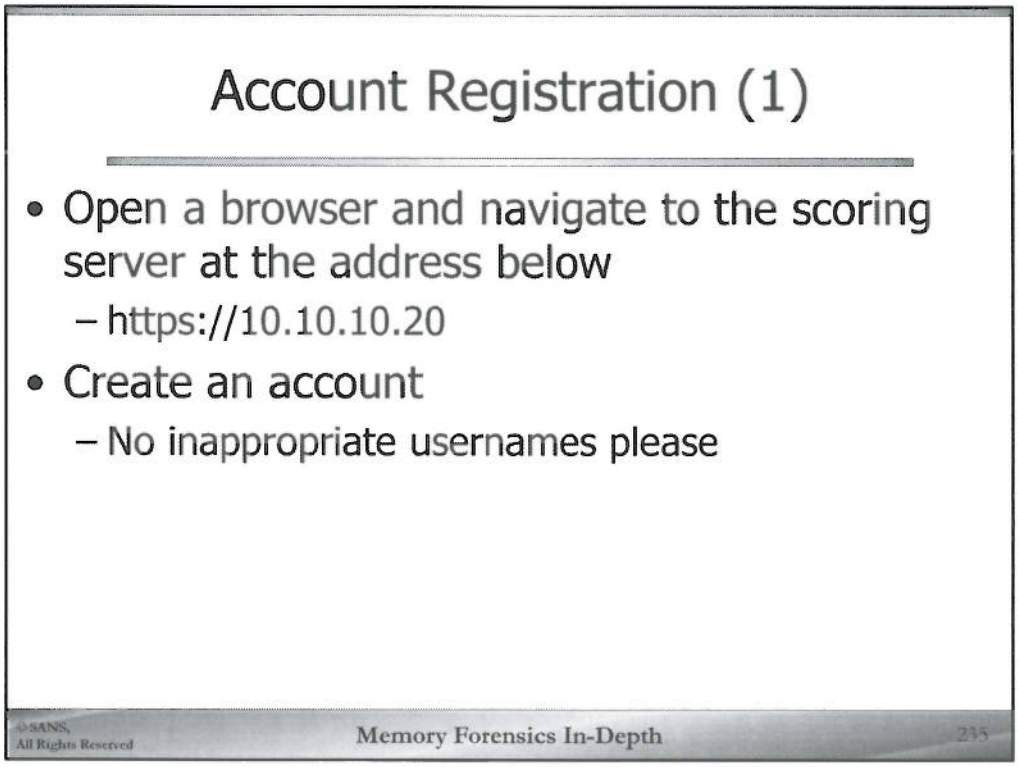


Short version: No shenanigans.

Long version: Just a couple of ground rules here. Since we're all adults, we almost feel silly having to clarify this; however, let's just get it out of the way.

Be courteous to your neighbor. Don't DoS the scoring server or other students. Don't attack the scoring server or other students either. If we find you doing either of these, you will be dismissed from the class immediately. We will report you to GIAC as an honor code violator and they may revoke your certifications. Bottom line, don't be that person!

We're all networked in the room, or over the Internet for Online students. This means that you could hack someone if they had an unpatched vulnerability. Be respectful. Don't hack someone else's box. It isn't a joke, it's a crime. If you get caught doing it, you'll be kicked out of class. Seriously. Again, if we catch any such shenanigans, you're out. We're here to learn and anything that disrupts the learning process will not be tolerated.

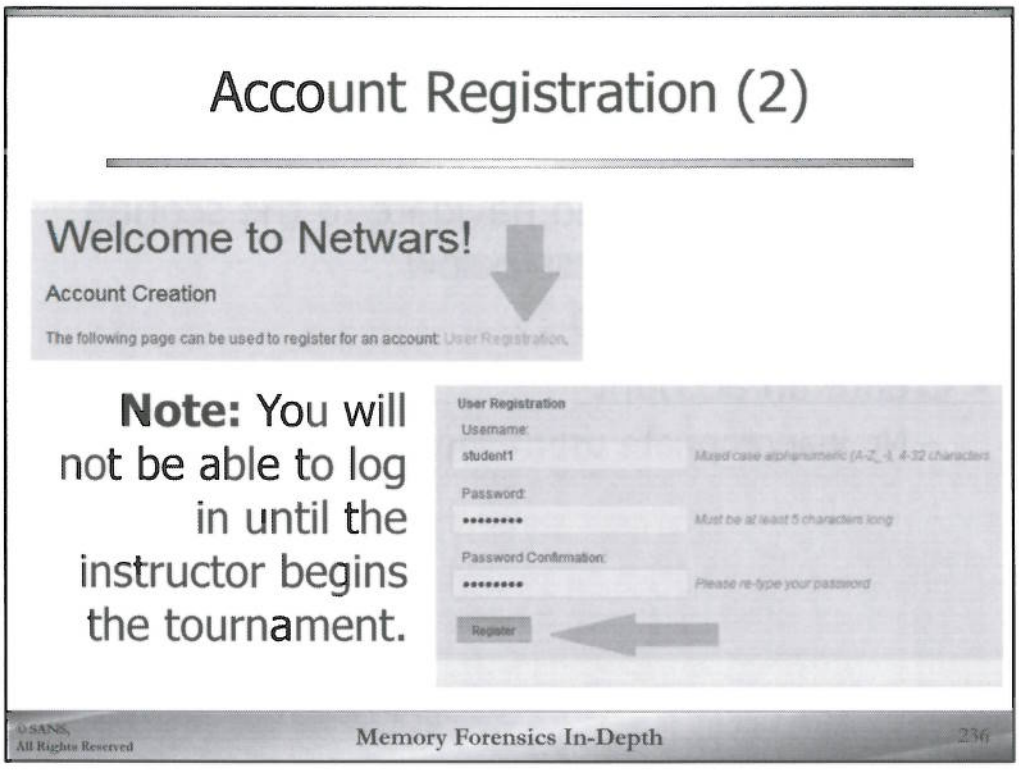


By default, the scoring server is located at 10.10.10.20. However, depending on the conference setup this address may be changed (you will be notified of any changes, if so).

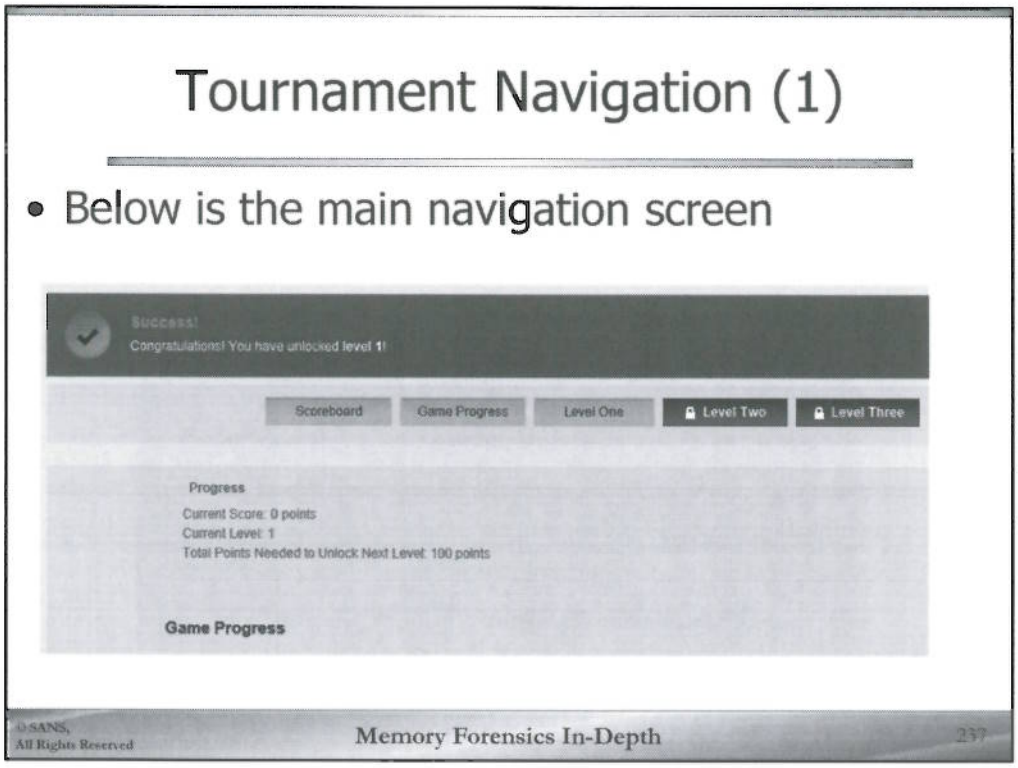
Once your IP information is set appropriately, you should be able to ping the scoring server. Each person should click the link to register for the tournament. However, you will not be able to log in until the instructor authorizes all accounts and grants access to the tournament. Point your web browser to https://10.10.10.20, create an account, and wait for the instructor to start the tournament. Once the instructor starts the tournament, you may log in. The tournament will end after a minimum of 6 hours of gameplay, the winner will be awarded, and the instructor will answer any questions you may have about the malware sample.

Online students: If you are playing via SANS Online (OnDemand or vLive), you should connect to the scoring server at the URL sent to you in a separate e-mail entitled FOR526 Virtual Lab Access. If you did not receive this e-mail, please e-mail virtual-labs-support@sans.org to address this. There is no need to reconfigure your network, you will use the OpenVPN client to to connect to the Virtual Lab. OnDemand students may play day 6 modules throughout their access to the course material. OnDemand does not use teams for the FOR526.6 module. In some instances the server may be reset and you will have to create a new account for access.

Simulcast students: Simulcast students should obtain the URL from for the scoring server from the instructor. There is no need to change your network settings. Simulcast students will compete against the in-class students for lethal forensicator coins.

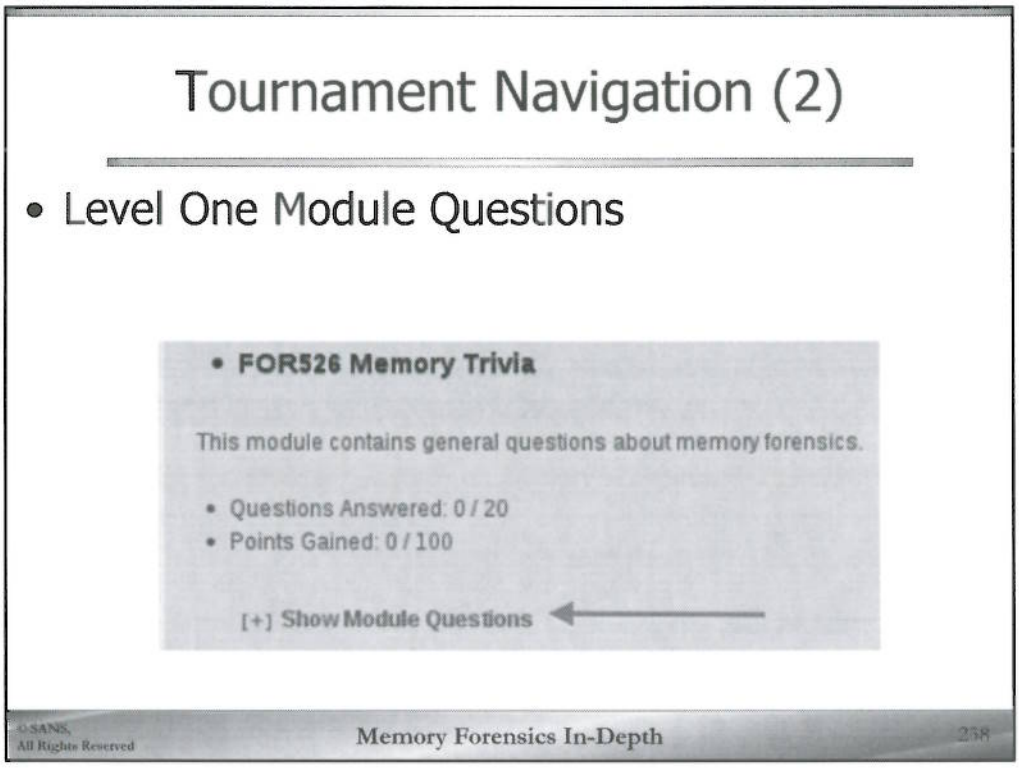


Please register for an account, but note that you will not be able to log in until the instructor starts the tournament.



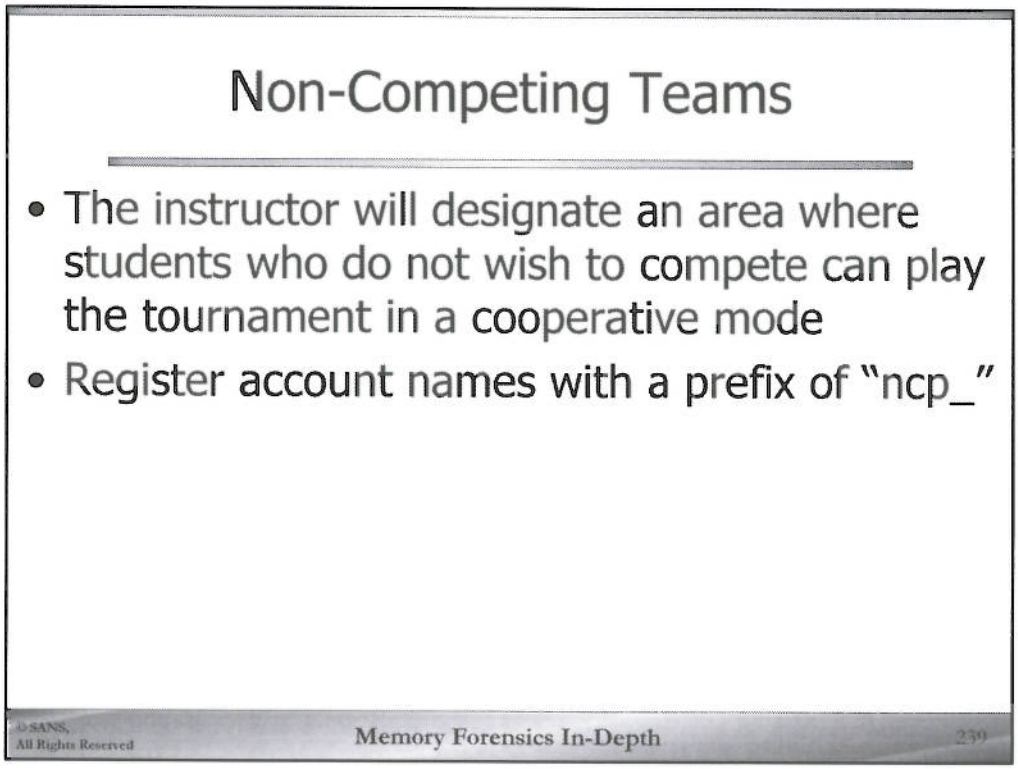
This is the main navigation screen. When you first log in, level one has been unlocked. Notice that level two and level three show a lock button meaning that they are currently not accessible. When you gain enough points, levels two and three will be unlocked.

This screen also displays links to the scoreboard and a link to your customized tournament progress. The tournament progress link allows you to examine which questions you've already answered and your progress through the tournament.



After clicking on the level one button, all level one questions are accessible. A description of the module is provided above the module questions, as well as the total number of questions answered and points gained from the module.

To access the module questions, click the "Show Module Questions" link as indicated by the arrow above.

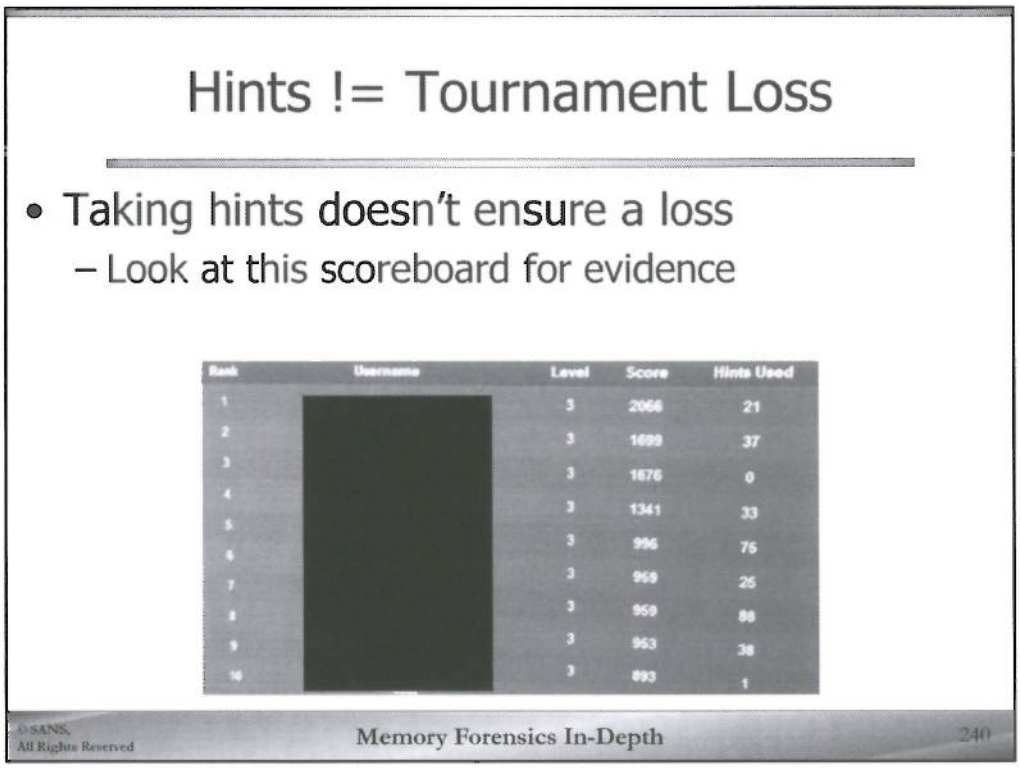


While most people prefer to compete for a lethal forensicator coin, we recognize that some people simply want to solidify their skills. To facilitate this, the instructor will designate an area where non-competing students can congregate and collaborate on the tournament questions. Because others are working on the modules, we ask that you keep the noise to a minimum while working on the questions. Also, because a team can establish a much higher point total than an individual, we ask that you register account names starting with the prefix "ncp_" (which stands for "not competing").

The instructor will still answer questions for non-competing students, but please try to leverage others in the area first. If you are not competing, please try to help your fellow students. You'll find that explaining a concept often helps reinforce it for you (so you get a benefit too).

You may also begin playing the tournament as a competitor and then transition to a non-competitor if you find the modules too difficult.

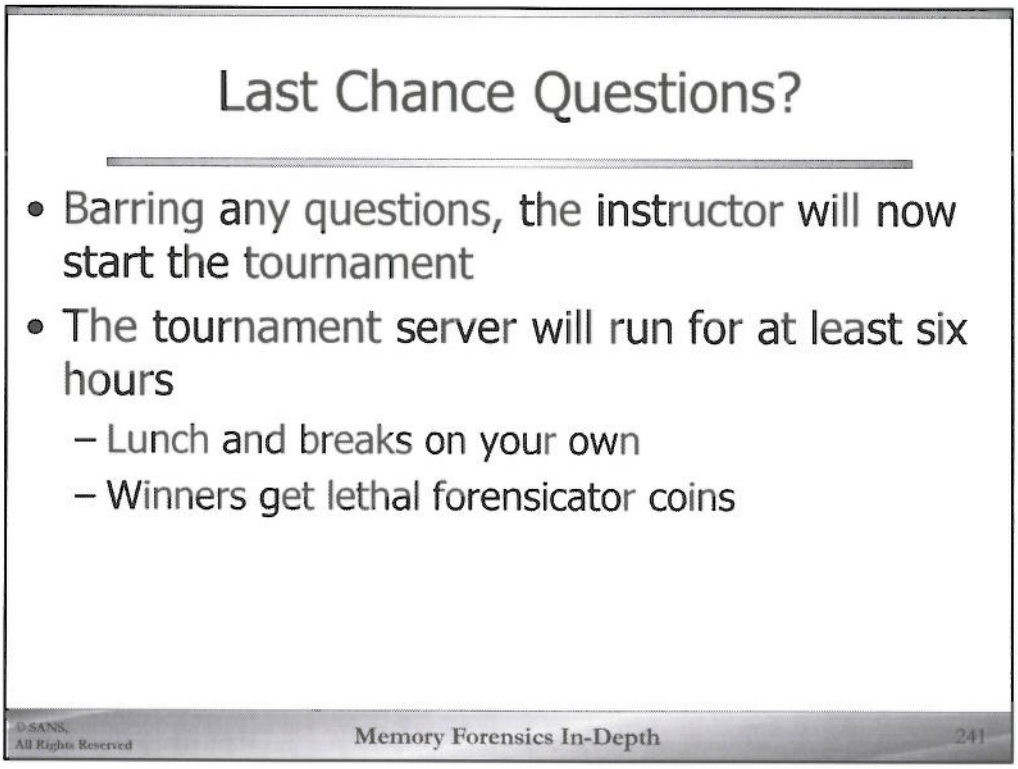
Note: This does not apply to SANS OnDemand students. Since SANS OnDemand students are provided access to the tournament indefinitely, the unfair advantage precludes them from winning a coin.



We've noted that some people feel like taking hints will result in failure. On many of the previous games, taking hints has actually been a winning strategy. There are a LOT of points available in the tournament (yes, people have actually finished everything, but we admit that it is challenging). Sometimes a hint helps you move ahead quickly and work most effectively.

Note the range of hints taken in this actual scoreboard from a previous tournament. The first and second place competitors took many hints to obtain great scores. The third place competitor was able to obtain an impressive points tally while taking no hints (quite an accomplishment!). There was a real race for fifth place in this tournament (bragging rights were at stake) and two people used more than 75 hints to try to improve their scores! In the end, one of these individuals was able to obtain the 5th place score (and secure the bragging rights).

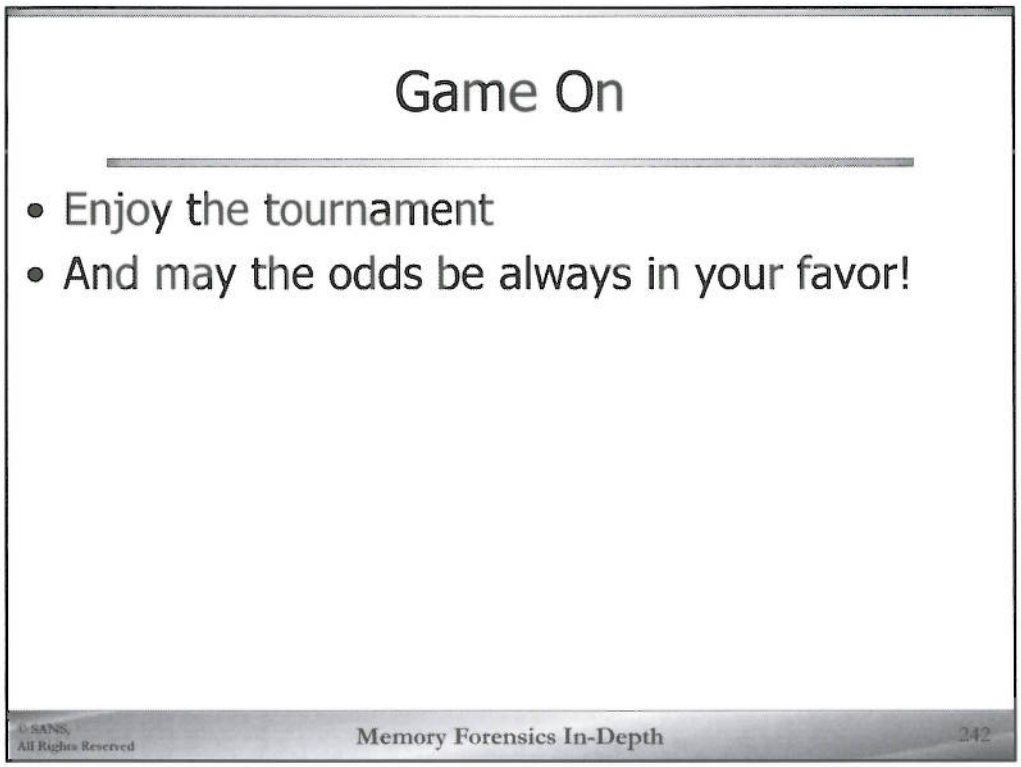
We show this to illustrate that taking a hint isn't a bad thing and it also isn't an admodule of failure. In fact, in every tournament we've run the winner has taken a fair number of hints to get the winning score in the time allotted for tournament play.



The tournament will be suspended after at least six hours of play has occurred. In conferences, the tournament will usually be suspended at 3:30 PM (this assumes a start of no later than 9:30 AM). You may continue to work on the module throughout lunch. Breaks and lunch are on your own. When the instructor terminates the tournament, winners will be awarded lethal forensicator coins and the class will be released. Time permitting, some questions may be reviewed by the instructor if anyone wants to stay and "get their learn on". In any case, the class will be dismissed no later than 5PM (or the normal stop time for days 1-5).

In vLive sessions, the server will be left on between the two vLive sessions to facilitate answering as many questions as possible.

OnDemand students will have an account on the server for their entire access period. OnDemand students do not compete for lethal forensicator coins, but can still play for maximum points and bragging rights. OnDemand students have much more time to complete modules than in-class students, meaning that taking hints is to complete the maximum number of challenges in the time allotted will be less of an issue.



Enjoy the tournament. We think it is both fun and a true learning experience that helps reinforce skills taught throughout the course. May the odds be always in your favor.

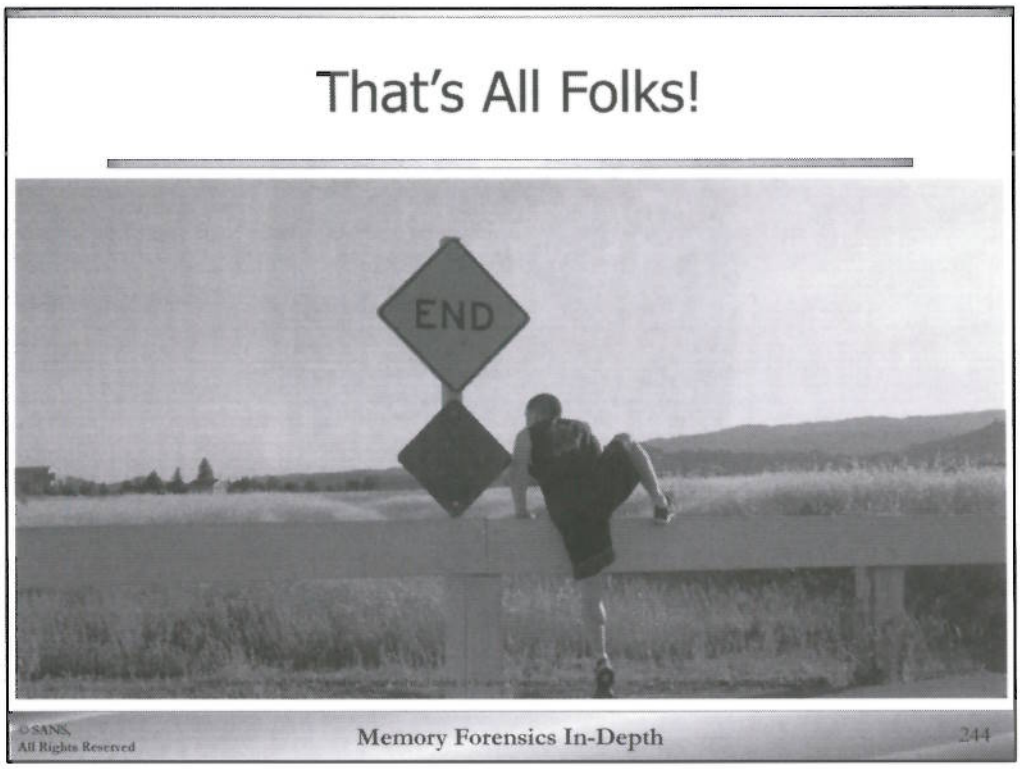
This Course is Part of the SANS Technology Institute (STI) Master's Degree Curriculum.

If you're brain is hurting from all you've learned in this class, but you still want more, consider applying for a Master's Degree from STI. We offer two hands-on, intensive Master's Degree programs:

- Master of Science in Information Security Engineering
- Master of Science in Information Security Management

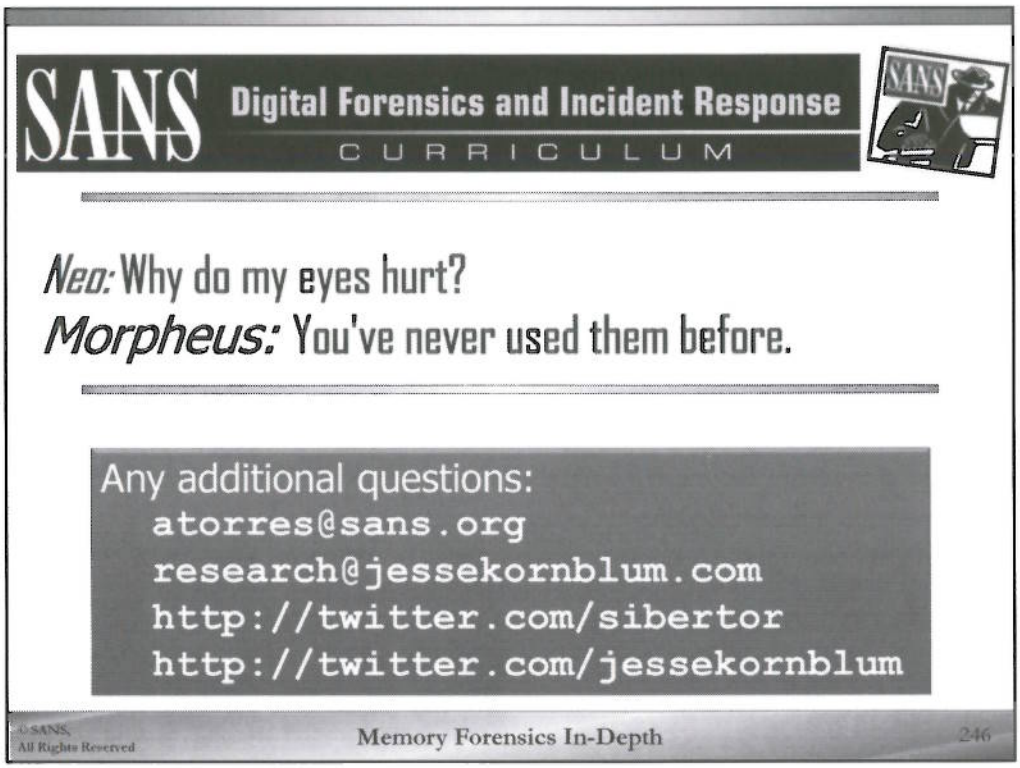
If you have a bachelor's degree and are ready to pursue a graduate degree in information security, please visit www.sans.edu for more information.

www.sans.edu **720-941-4932** **info@sans.edu**



This the end of the course. But don't stop here! Have fun exploring memory forensics!

This page intentionally left blank.



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