SANS

www.sans.org

FORENSICS 526

MEMORY FORENSICS IN-DEPTH

Workbook: Memory Forensics In-Depth Hands-on Exercises

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

For526_Wkbk_A06_01

Exercise 1: Virtual Machine Setup

Objectives

- Install and prepare your Ubutu SIFT lab workstation for digital memory forensic analysis for the week.
- Set up and activate the Windows 8.1 workstation virtual machine
- Gain experience with the FOR526 memory forensics weapons arsenal and platforms.

What is on the USB?

The course USB drive contains several subdirectories. A brief summary of these is below:

/netwars/	Data files for the NetWars Memory Forensics Tournament
/exercises/	Evidence files for each exercise in the class, contained in subdirectories
/utilities/	Just in case your machine didn't come to class fully-loaded, this directory has the basic essentials (VMWare Player, 7zip, etc.)
/FOR526_vms/	Compressed archives containing the VMWare appliances you will need for this course. (1) Customized Ubuntu SIFT VM (1) Win8.1x64 VM
/live analysis/	Triage tools for live audit collection and memory analysis

Part 1: Preparation for the Ubuntu SIFT Setup

- Copy "**`SIFT_FOR526_VM.zip`**" to your local system and unzip using 7zip (located in the utilities directory on the USB) or a comparable archive utility (other than the native Windows archive tool).
- Ensure that you have a working copy of VMWare Workstation, VMWare Fusion or VMWare Player. A copy of VMWare Player is located in the utilities directory on the FOR526 USB.

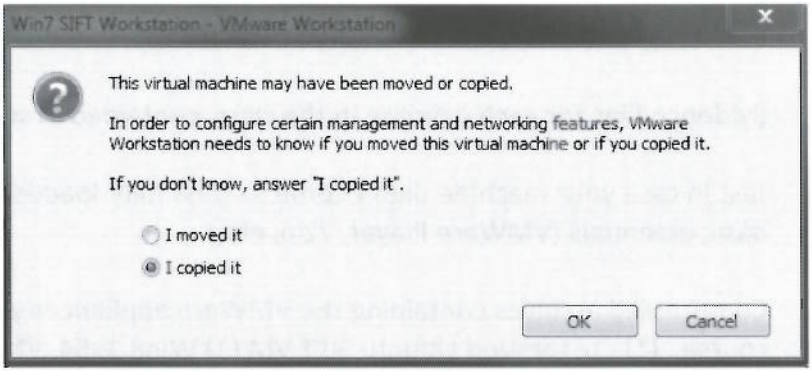
Exercise – Setting up your Ubuntu SIFT Lab Environment

- Start VMWare environment and open the "SIFT_FOR526_VM.vmx". Before powering on the VM, do the following:
  - **Adjust Memory** -> Select "Edit Virtual Machine Settings" -> Select Memory
  - **Adjust Processors** -> Select Processors and Assign more Cores and/or Processors
    NOTE: Do not give your VM more than ½ of your machines memory. If your machine slows down as a result, reduce the amount of memory/processors allocated to the VM.

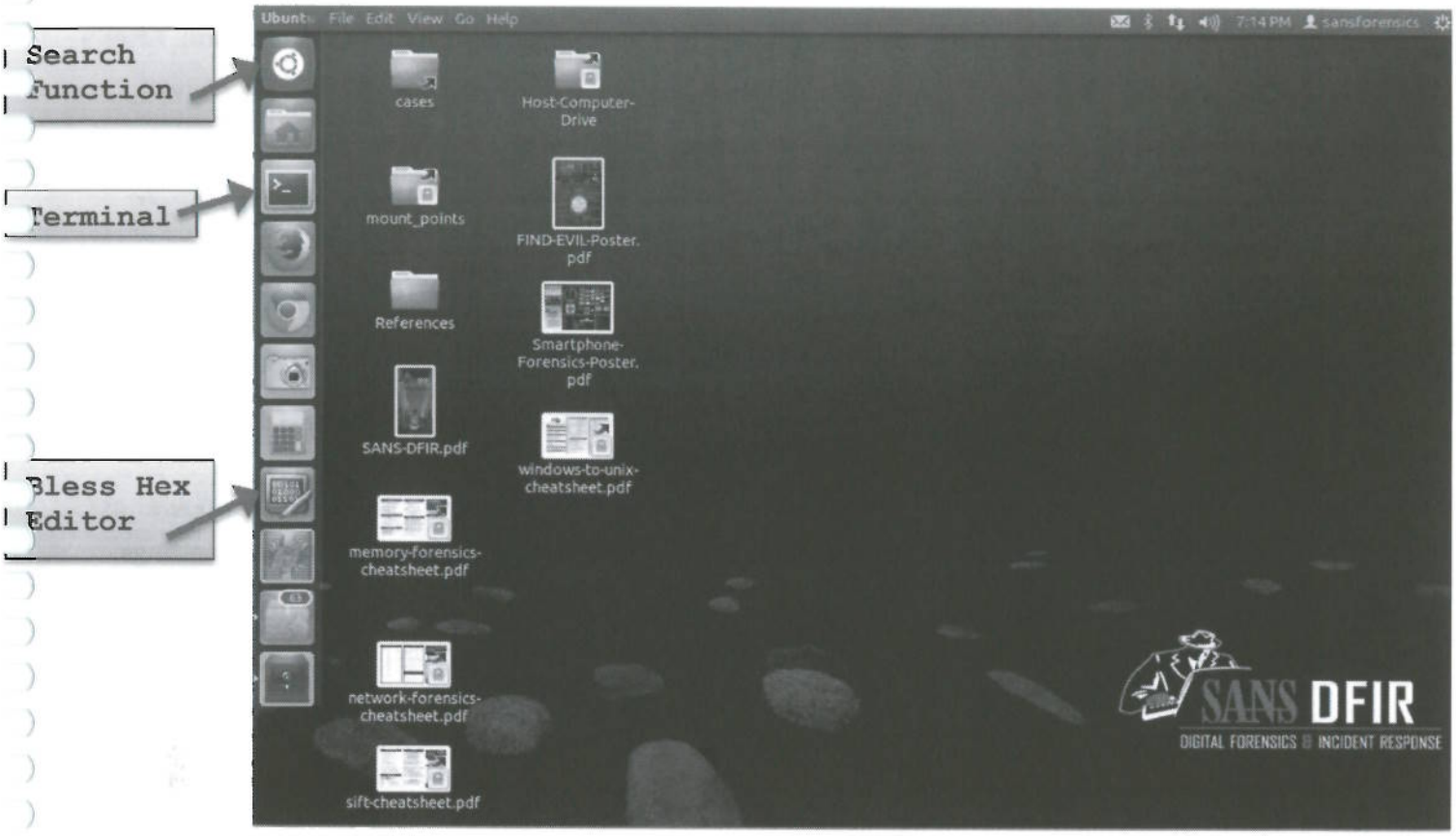
- Click the "Power on this virtual machine" link or "Play" icon to start the virtual machine.

NOTE: If you get an error stating you must enable "Intel VT" technology, boot into your system's BIOS and enable the feature there. If upon opening your SIFT VM, you have a black screen, shut it down and disable 3D Graphics in the VM Settings in VMWare and attempt to boot the VM again.

- Depending on your software version, VMWare may prompt you to "upgrade this virtual machine". Click "Upgrade" if you see this dialog.
- When asked if you "moved or copied" the virtual machine, click "I copied it".



- After the system boots to the login screen, use the following credentials and click "Log In".
  - Username: **`sansforensics`**
  - Password: **`forensics`**



- Click the terminal icon in the toolbar on the right side of the desktop to launch a terminal instance.
  - Change directories by typing the following:

```
cd /cases
```

  - Extract the files in the bootcamp.rar archive located at /cases.

```
rar x bootcamp.rar
```

Part 2: Preparation for the Win8.1 VM Setup

Copy "**`Win8.1_x64_FOR526.zip`**" to your local system and unzip using 7zip (located in the utilities directory on the USB) or a comparable archive utility (other than the native Windows archive tool). After a long extraction process (maybe up to 20-30 min), you should now see a folder in your directory called "Win8.1_x64_FOR526".

- Ensure that you have a working copy of VMWare Workstation, VMWare Fusion or VMWare Player (also located in the utilities directory on the USB).

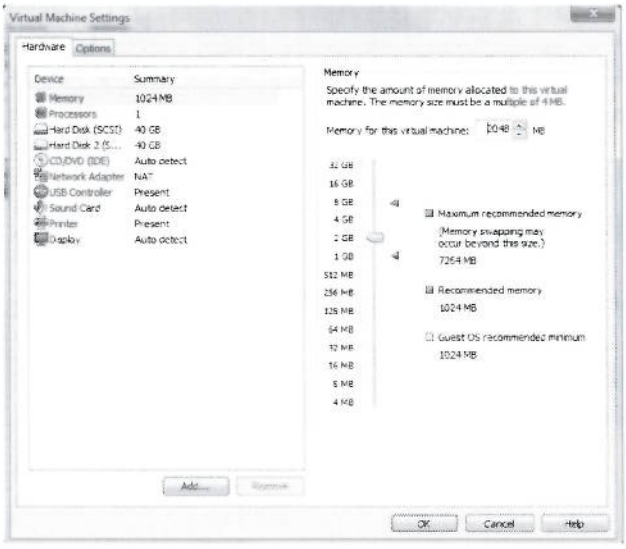
Exercise – Setting up your Windows 8.1 VM Environment

- Start VMWare environment and open the "Win8.1_x64_FOR526.vmx". DO NOT START THE VM YET.

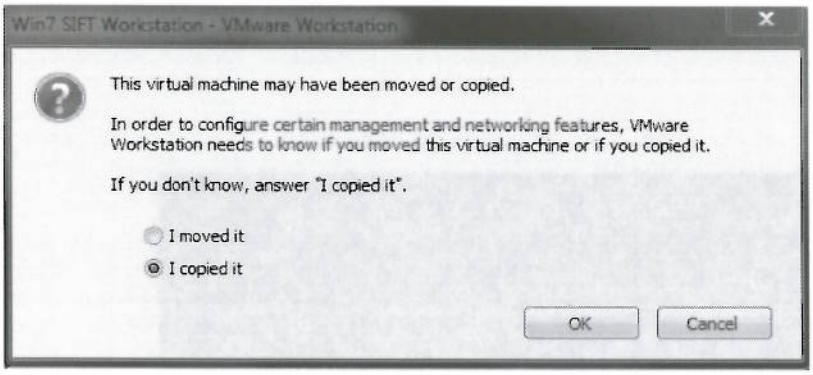
- Tweak the settings of your VMware configuration to allow for more memory and processor power as your system will allow. Note: Do not ever increase it beyond the maximum recommended settings.

- **Adjust Memory** -> Select "Edit Virtual Machine Settings" -> Select Memory

- **Adjust Processors** -> Select Processors and Assign more Cores and/or Processors
  NOTE: Do not give your VM more than ½ of your machines memory. If your machine slows down as a result, reduce the amount of memory/processors allocated to the VM.



- Click the "Power on this virtual machine" link or "Play" icon to start the virtual machine. Press "Power on virtual machine" and select "I copied it". (Note: If upon opening your SIFT VM, you have a black screen, shut it down and disable the "Accelerate 3D Graphics" option in the VM Settings in VMWare and attempt to boot the VM again.)

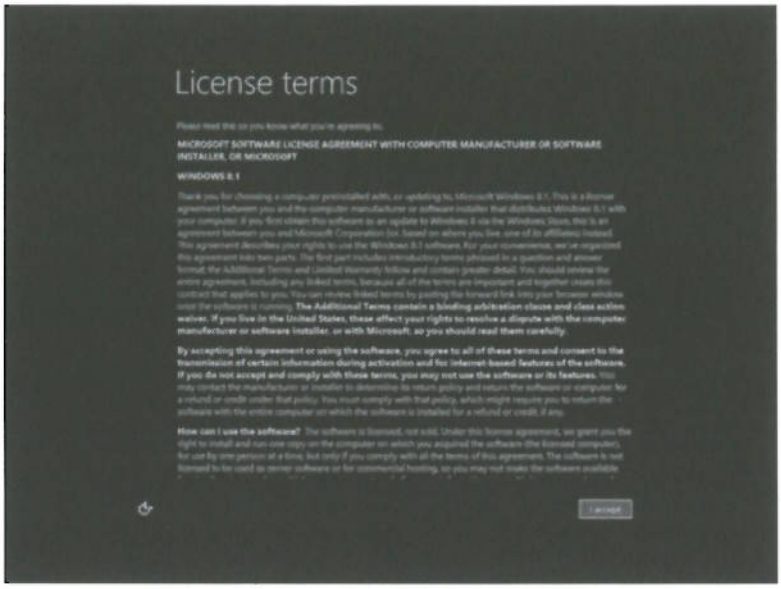


If you get an error on boot please read or skip to 10 -> Some may get this error: "Binary translation is incompatible with long mode on this platform. Disabling long mode. Without long mode support, the virtual machine will not be able to run 64-bit code. For more details seehttp://vmware.com/info?id=152."

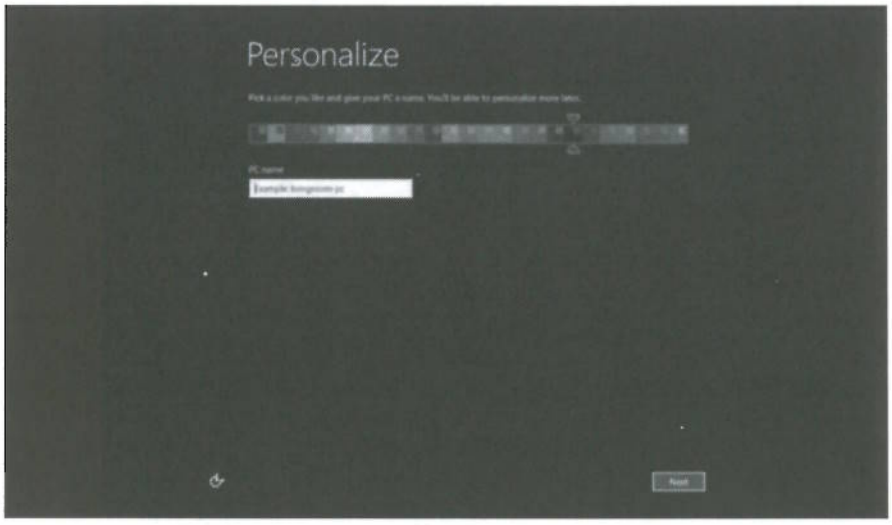
The first thing to try is to verify that VT technology is enabled in the BIOS, as described in this VMware guide on 64-bit (http://www.vmware.com/pdf/processor_check.pdf).

Input your Windows 8 License Key.

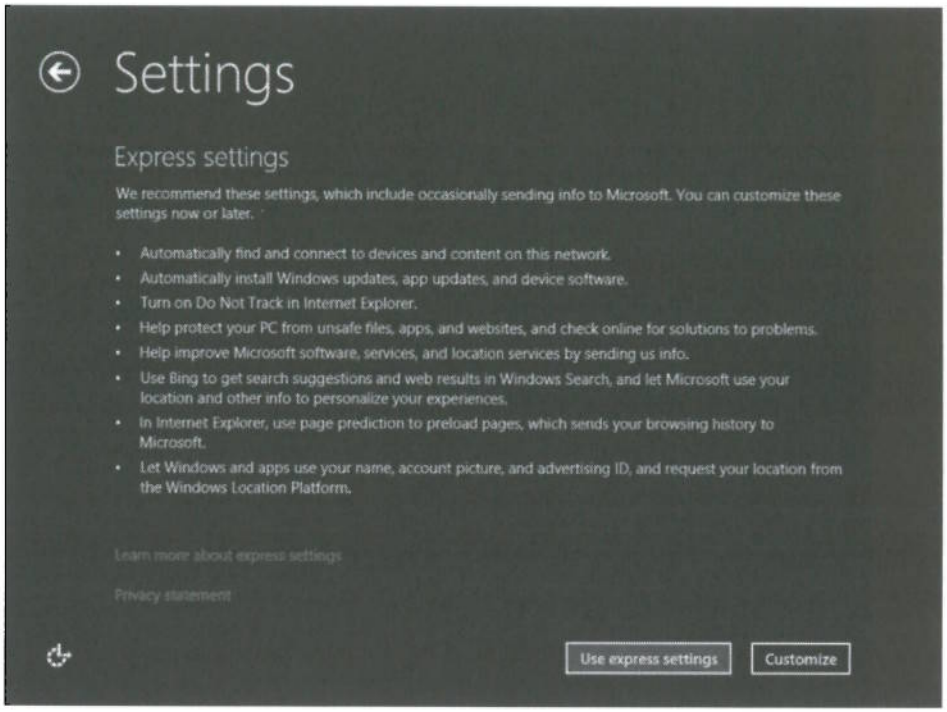
- Accept License Agreement.



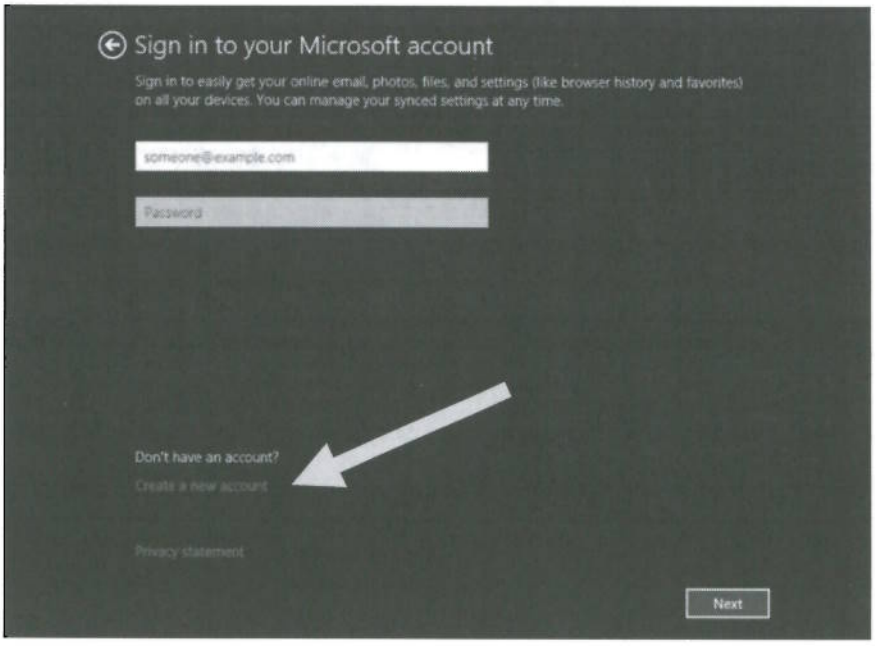
- Personalize your desktop install. Choose any color that you like and choose a proper system name such as Win8.1-SIFT.



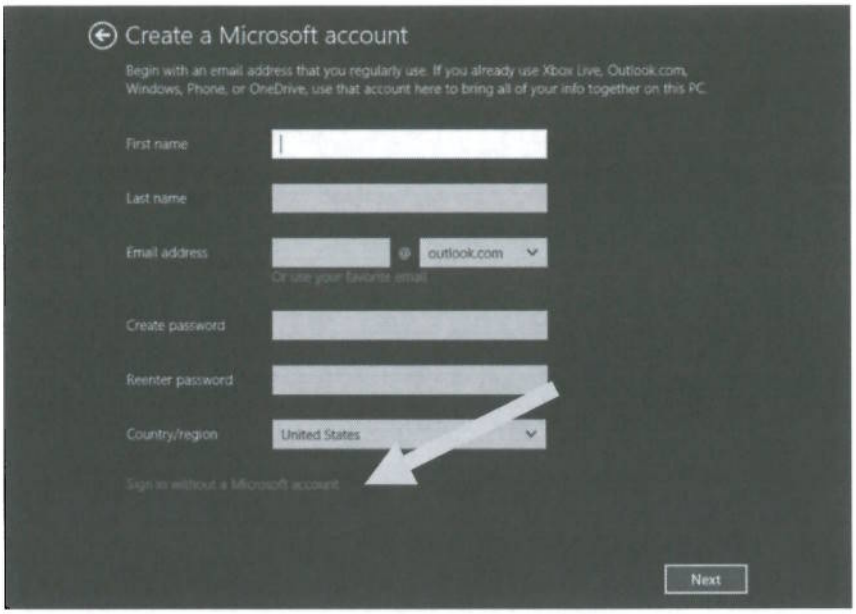
- Select "Use Express Settings" option.



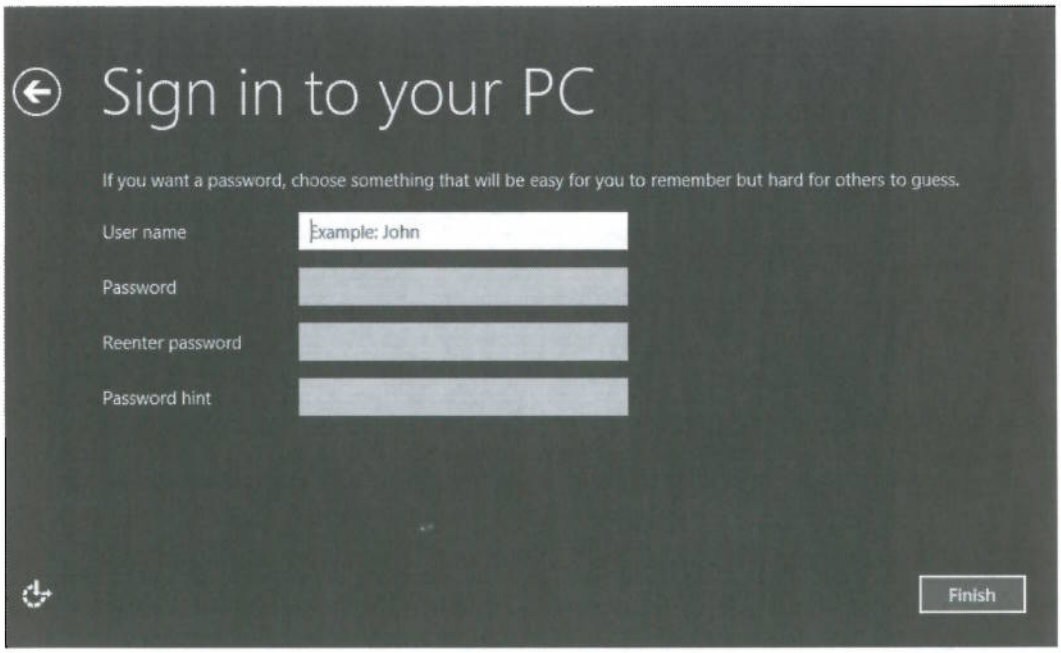
- Select Create a New Account when prompted to login using your Microsoft Live account.



- Then select "Login without Microsoft Account" and create a new user.



- Sign in to your PC. Create a username and a password.



- Your system will now finalize the settings for the install.

- Log out of the current user.

- Log in as sansforensics using the password “forensics”.

- If VMware asks you to upgrade “VMware Tools” feel free to do so.
- When the desktop loads, you should see a similar setup to that shown below.

Exercise 2 – Spotting Hidden Processes

Objectives

- Practice Volatility syntax and create "muscle memory" by running through some basic plugins
- Analyze the output from two different plugins to enumerate processes
- Detect processes that have been obfuscated through direct kernel object manipulation

Exercise Preparation

Start a terminal and change into the **/cases/exercise2** directory in your SIFT 3.0. Unzip **processfu.zip**.

```
$ cd /cases/exercise2
$ unzip processfu.zip
```

Detecting Hidden Processes Exercise – Questions

Overview: Analyze the **processfu.img** memory image file to detect signs of hidden processes.

1. Detecting the Correct System Profile

a. Using the **imageinfo** profile, determine what version of Windows the target system was running.

_____

b. Launch **Bless Hex Editor** in your Ubuntu SIFT and open **processfu.img**. Select "Find" from the **Search** menu and enter the signature for the KDBG, selecting as "hexadecimal" for value type.

```
00 00 00 00 00 00 00 00 4B 44 42 47
```

(8 null bytes followed by KDBG)

What are the two bytes that follow this signature that **imageinfo** uses in its system profile identification process?

_____

2. Enumerating Process by List Walking

a. Using the **pslist** plugin, enumerate the number of actively running **svchost.exe** processes. Who is the parent of these processes and is that normal?

b. There are 3 **cmd.exe** processes seen in the **pslist** output. Do any of them have actively running child processes?

3. Enumerating Process by Scanning

a. Using the **psscan** plugin, how many EPROCESS structures are identified for **cmd.exe** processes? Why are there duplicate entries for the same PID?

b. Of these three unique **cmd.exe** processes, which are shown to have (or had in the past) child processes, based on **psscan** output? Are the child processes actively running? Do they have an exit time? What might explain the difference in your answers for 2b & 3b?

4. Extra Credit.

a. Determine the cause of the anomalous behavior (hidden process). Try using the "vol.py –h" to get some ideas of other plugins to run. Identifying commands that were entered in the command shell is an excellent way to find out what might have caused this anomaly. (Hint: **cmdscan** and **consoles** plugins)

b. What was the IP address of the remote system that connected to our target? What port did the remote system connect to and what process was responsible for binding that port on the local system?

Part I. Detecting Hidden Processes

1. Detecting the Correct System Profile

a. Using the **imageinfo** profile, determine what version of Windows the target system was running.

```
vol.py -f processfu.img imageinfo
```

According to the imageinfo output, the target system was a Windows XP SP2 x86.

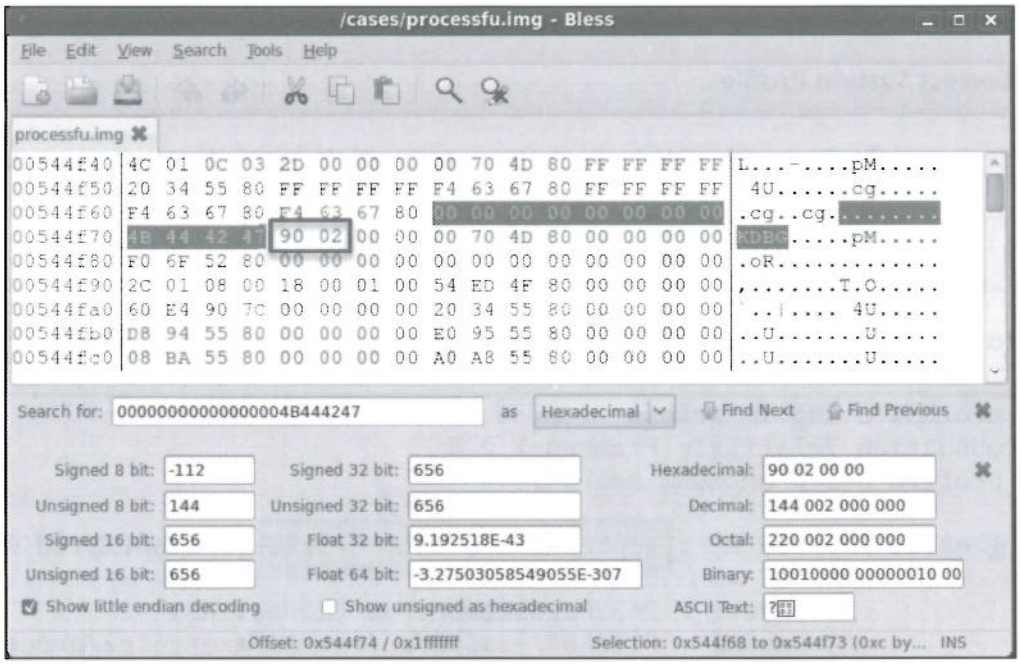
```
$ vol.py -f processfu.img imageinfo
Volatility Foundation Volatility Framework 2.4
Determining profile based on KDBG search...

          Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with WinXPSP
2x86)
                     AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (/cases/exercise2/processfu.img)
                      PAE type : PAE
                           DTB : 0x323000L
                          KDBG : 0x80544f60
          Number of Processors : 1
     Image Type (Service Pack) : 2
                KPCR for CPU 0 : 0xffdff000
             KUSER_SHARED_DATA : 0xffdf0000
           Image date and time : 2014-01-13 03:09:21 UTC+0000
     Image local date and time : 2014-01-12 22:09:21 -0500
```

b. Launch Bless Hex Editor in your Ubuntu SIFT and open **processfu.img**. Select "Find" from the **Search** menu and enter the signature for the KDBG, selecting as "hexadecimal" for value type.

```
00 00 00 00 00 00 00 00 4B 44 42 47
```

(8 null bytes followed by KDBG)

What are the two bytes that follow this signature that **imageinfo** uses in its system profile identification process?



The two byte signature after "KDBG" is 90 02, indicative of Windows XP.

2. Enumerating Process by List Walking

a. Using the **`pslist`** plugin, enumerate the number of actively running **`svchost.exe`** processes. Who is the parent of these processes and is that normal?

```
vol.py -f processfu.img --profile=WinXPSP2x86 pslist| grep -i svchost
```

There are six svchost.exe processes running on this system (that are in the doubly linked list) when the image was acquired. The parent process for all of these instances is the legitimate parent pid, 696, the services.exe process.

```
$ vol.py -f processfu.img --profile=WinXPSP2x86 pslist |grep -i svchost |wc -l
Volatility Foundation Volatility Framework 2.3.1
6
$ vol.py -f processfu.img --profile=WinXPSP2x86 pslist |grep -i svchost
Volatility Foundation Volatility Framework 2.3.1
0x82204c88 svchost.exe            880    696     17      196      0      0 2014-01-11 22:04:21 UTC+0000
0x8236a978 svchost.exe            964    696     11      261      0      0 2014-01-11 22:04:21 UTC+0000
0x8220aa50 svchost.exe           1080    696     61     1260      0      0 2014-01-11 22:04:21 UTC+0000
0x8230eda0 svchost.exe           1148    696      5       76      0      0 2014-01-11 22:04:21 UTC+0000
0x82225328 svchost.exe           1220    696     13      174      0      0 2014-01-11 22:04:21 UTC+0000
0x8240dda0 svchost.exe           1504    696      5       90      0      0 2014-01-11 22:04:29 UTC+0000
```

b. There are 3 **cmd.exe** processes seen in the **pslist** output. Do any of them have actively running child processes?

```
0x8247a978 cmd.exe             3792    840      1       31      0      0 2014-01-13 02:51:03 UTC+0000
0x82210da0 cmd.exe             1944    840      1       32      0      0 2014-01-13 03:02:50 UTC+0000
0x8232a020 cmd.exe             3680    840      1       32      0      0 2014-01-13 03:06:30 UTC+0000
0x8225d930 wmiprvse.exe        3824    880      7      140      0      0 2014-01-13 03:07:25 UTC+0000
0x82361768 winpmem_1.4.exe     4092   3680      1       21      0      0 2014-01-13 03:09:21 UTC+0000
```

Of the three cmd.exe processes, only one, PID 3680, has an actively running child processes, PID 4092, winpmem.exe.

3. Enumerating Process by Scanning

a. Using the **psscan** plugin, how many EPROCESS structures are identified for **cmd.exe** processes? Why are there duplicate entries for the same PID?

```
vol.py -f processfu.img --profile=WinXPSP2x86 psscan| grep -i cmd.exe
```

```
sansforensics@SIFT-Workstation:/cases$ vol.py -f processfu.img --profile=WinXPSP2x86 psscan |grep -i cmd.exe
Volatility Foundation Volatility Framework 2.3.1
0x02210da0 cmd.exe            1944    840 0x08ac02c0 2014-01-13 03:02:50 UTC+0000
0x0232a020 cmd.exe            3680    840 0x08ac02e0 2014-01-13 03:06:30 UTC+0000
0x0247a978 cmd.exe            3792    840 0x08ac0380 2014-01-13 02:51:03 UTC+0000
0x05221978 cmd.exe            3792    840 0x08ac0380 2014-01-13 02:51:03 UTC+0000
0x05686da0 cmd.exe            1944    840 0x08ac02c0 2014-01-13 03:02:50 UTC+0000
0x0f460020 cmd.exe            3680    840 0x08ac02e0 2014-01-13 03:06:30 UTC+0000
0x13be4020 cmd.exe            3680    840 0x08ac02e0 2014-01-13 03:06:30 UTC+0000
0x1997bda0 cmd.exe            1944    840 0x08ac02c0 2014-01-13 03:02:50 UTC+0000
```

Psscan scans for pattern matches based on the presence of process pool tags. The duplicates exist due to the highly fragmented and constantly changing nature of physical memory. Objects are commonly copied and moved, with old structures not being overwritten. Therefore, psscan was able to identify more than one structure for these processes, with one only EPROCESS block actually active per running process.

b. Of these three unique **cmd.exe** processes, which are shown to have (or had in the past) child processes, based on **psscan** output?

```
vol.py -f processfu.img --profile=WinXPSP2x86 psscan| grep
'1944\|3680\|3792'
```

```
sansforensics@SIFT-Workstation:/cases$ vol.py -f processfu.img --profile=WinXPSP2x86 psscan |grep '1944\|3680\|3792'
Volatility Foundation Volatility Framework 2.3.1
0x01f72da0 nc.exe             3284   1944 0x08ac0280 2014-01-13 03:05:08 UTC+0000
0x02210da0 cmd.exe            1944    840 0x08ac02c0 2014-01-13 03:02:50 UTC+0000
0x0232a020 cmd.exe            3680    840 0x08ac02e0 2014-01-13 03:06:30 UTC+0000
0x02361768 winpmem_1.4.exe    4092   3680 0x08ac03a0 2014-01-13 03:09:21 UTC+0000
0x0247a978 cmd.exe            3792    840 0x08ac0380 2014-01-13 02:51:03 UTC+0000
0x05221978 cmd.exe            3792    840 0x08ac0380 2014-01-13 02:51:03 UTC+0000
0x0529b768 winpmem_1.4.exe    4092   3680 0x08ac03a0 2014-01-13 03:09:21 UTC+0000
0x05686da0 cmd.exe            1944    840 0x08ac02c0 2014-01-13 03:02:50 UTC+0000
0x0b903390 winpmem_1.4.exe    2056   3792 0x08ac0280 2014-01-13 02:51:53 UTC+0000
0x0dcb7da0 nc.exe             3284   1944 0x08ac0280 2014-01-13 03:05:08 UTC+0000
0x0f460020 cmd.exe            3680    840 0x08ac02e0 2014-01-13 03:06:30 UTC+0000
0x12fa6390 winpmem_1.4.exe    2056   3792 0x08ac0280 2014-01-13 02:51:53 UTC+0000
0x13be4020 cmd.exe            3680    840 0x08ac02e0 2014-01-13 03:06:30 UTC+0000
0x1997bda0 cmd.exe            1944    840 0x08ac02c0 2014-01-13 03:02:50 UTC+0000
```

Psscan has identified an additional child process, nc.exe PID 3284, that was not seen in pslist output. While it is possible that additional terminated child processes could still have EPROCESS structures in memory that psscan would hit on, we can easily rule this out due to the absence of an exit time (far right column).

Another possibility exists that this nc.exe process existed in a previous boot. Previous boot processes are ended abruptly and no exit time is written to their EPROCESS structure, though the structure may still exist in memory after reboot. We can again rule this out, because the creation time of the nc.exe process is after the current smss.exe creation time.

4. **Extra Credit.**

   a. Determine the cause of the anomalous behavior (hidden process). Try using the "vol.py –h" to get some ideas of other plugins to run. Identifying commands that were entered in the command shell is an excellent way to find out what might have caused this anomaly. (Hint: **cmdscan** and **consoles** plugins)

```
vol.py -f processfu.img --profile=WinXPSP2x86 cmdscan
```

```
sansforensics@SIFT-Workstation:/cases$ vol.py -f processfu.img --profile=WinXPSP2x86 cmdscan
Volatility Foundation Volatility Framework 2.3.1
**************************************************
CommandProcess: csrss.exe Pid: 628
CommandHistory: 0xfc4848 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 21 LastAdded: 20 LastDisplayed: 20
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x160
Cmd #0 @ 0xfb8668: ipconfig
Cmd #1 @ 0xfc4760: tasklist
Cmd #2 @ 0x4edc68: cd Desktop
Cmd #3 @ 0xfc4ad0: dir
Cmd #4 @ 0x4edc88: cd Rootkits
Cmd #5 @ 0xfb8658: dir
Cmd #6 @ 0x4edcf0: cd FU
Cmd #7 @ 0xfd4e50: dir
Cmd #8 @ 0x4edd08: cd FU_Rootkit
Cmd #9 @ 0x4edd30: dir
Cmd #10 @ 0x4edd40: cd EXE
Cmd #11 @ 0x4edd58: dir
Cmd #12 @ 0x4edd68: tasklist
Cmd #13 @ 0x4edd88: fu.exe /?
Cmd #14 @ 0x4edeb8: fu.exe -ph 3284
Cmd #15 @ 0x4edee0: tasklist
```

   b. What was the IP address of the remote system that connected to our target? What port did the remote system connect to and what process was responsible for binding that port on the local system?

```
vol.py -f processfu.img --profile=WinXPSP2x86 connections
```

```
sansforensics@SIFT-Workstation:/cases$ vol.py -f processfu.img --profile=WinXPSP2x86 connections
Volatility Foundation Volatility Framework 2.3.1
Offset(V)  Local Address             Remote Address            Pid
---------- ------------------------ ------------------------ ---
0x82375c38 192.168.89.166:4444      192.168.89.160:58933      3284
sansforensics@SIFT-Workstation:/cases$ vol.py -f processfu.img --profile=WinXPSP2x86 psscan |grep 3284
Volatility Foundation Volatility Framework 2.3.1
0x01f72da0 nc.exe             3284   1944 0x08ac0280 2014-01-13 03:05:08 UTC+0000
0x0dcb7da0 nc.exe             3284   1944 0x08ac0280 2014-01-13 03:05:08 UTC+0000
```

Exercise: Key Takeaways

1. The **`pslist`** plugin walks the doubly-linked list of processes, similar to the volatile data collection tools, SysInternals pslist or the native Windows tasklist. This way of enumerating processes can be evaded through direct kernel object manipulation. Malicious code with access to the kernel can change the values of the EPROCESS structures that are in front and behind of the target process so they no longer point to this process structure – no longer including it in the **`pslist`** process enumeration.
2. **`Psscan`** enumerates processes through brute-force scanning, identifying process pool allocation based on pool tags. In this exercise, we saw that this output contains duplicates. In addition, we should expect to see EPROCESS structures from terminated or previous boot processes that have not yet been overwritten in physical memory.
3. In comparing the outputs of both plugins, we can detect hidden processes that have been unlinked from the active process list but are still running in memory. Later in the class, we will introduce a plugin, **`psxview`**, that will automate this comparison for us, as well as bring in process data from other sources in Windows memory.

This page intentionally left blank.

Exercise 3: Live Memory Analysis with Rekall

Objectives

- Capture a memory image of a target system using an open-source free memory acquisition tool
- Introduce different memory acquisition methods (\\Device\Physical Memory & PTE remapping) and output formats offered by winpmem
- Validate the resultant memory image by scanning for Windows memory structures using the Volatility framework plugins

Live System Triage and Memory Analysis

Mandiant's Redline is an excellent audit and memory capture parsing tool, but the End User License Agreement for Redline prohibits its use for commercial purposes. The accepted interpretation of this EULA is that you may use this product for investigations if you are working on an internal security team. You may not use the software "to provide paid services to third parties without written permission from Mandiant"[1]. Therefore, in this lab, we offer an alternative live system triage and memory acquisition techniques using the free open-source winpmem tool written by Michael Cohen (@scudette) and the Rekall Memory Forensic Framework.

1. Launch your Windows 8.1 VM with VMware Workstation/Player/Fusion. If needed, login to the Windows 8.1 VM with the credentials below:

```
User: sansforensics
Password: forensics
```

2. Open the **`Exercise_3.txt`** file located on the **`sansforensics`** desktop with Notepad.

3. Open a command shell with **`Administrative privileges`** by right-clicking on the **`cmd`** icon and select "**`Run as Administrator`**".

4. Change directories via command line to the **C:\Program Files\Rekall** directory.

```
C:\Windows\System32\> cd C:\Program Files\Rekall
```

Winpmem allows an incident responder to manually load a kernel driver, allowing for live physical memory analysis using the Rekall Memory Forensic Framework.

- Type the following command in the command window to see the available options you can use with winpmem:

```
C:\Program Files\Rekall> winpmem_1.6.2.exe -h
```

```
c:\Program Files\Rekall>winpmem_1.6.2.exe -h
Winpmem - A memory imager for windows.
Copyright Michael Cohen (scudette@gmail.com) 2012-2014.

Version 1.6.2 Nov  1 2014
Usage:
  winpmem_1.6.2.exe [option] [output path]

Option:
  -l    Load the driver and exit.
  -u    Unload the driver and exit.
  -d [filename]
        Extract driver to this file (Default use random name).
  -h    Display this help.
  -w    Turn on write mode.
  -0    Use MmMapIoSpace method.
  -1    Use \\Device\PhysicalMemory method (Default for 32bit OS).
  -2    Use PTE remapping (AMD64 only - Default for 64bit OS).
  -3    Use PTE remapping with PCI instrospection (AMD64 Only).
  -e    Produce an ELF core dump.
  -p    Also acquire the pagefile.
        This flag may be followed by the pagefile path.
```

- We will be manually loading the winpmem driver using the "-l" option and running Rekall for live memory analysis.

```
C:\Program Files\Rekall> winpmem_1.6.2.exe -l
```

```
c:\Program Files\Rekall>winpmem_1.6.2.exe -l
Extracting driver to C:\Users\SANSFO~2\AppData\Local\Temp\pmeDB1F.tmp
Driver Unloaded.
Loaded Driver C:\Users\SANSFO~2\AppData\Local\Temp\pmeDB1F.tmp.
Deleting C:\Users\SANSFO~2\AppData\Local\Temp\pmeDB1F.tmp
CR3: 0x00001AA000
 5 memory ranges:
Start 0x00001000 - Length 0x0009E000
Start 0x00100000 - Length 0x00002000
Start 0x00103000 - Length 0xBFDDD000
Start 0xBFF00000 - Length 0x00100000
Start 0x100000000 - Length 0x40000000
Acquitision mode PTE Remapping
```

1. Now that the winpmem driver is loaded on the target system, we can begin to analyze current system state of the running system using a Rekall interactive session. Type the following command to invoke a Rekall interactive session referencing the loaded winpmem kernel driver and using notepad as a paging tool scroll buffer.

```
C:\Program Files\Rekall> Rekal -f \\.\pmem --pager=notepad
```

2. Begin your triage of the live system with the imageinfo plugin. This plugin is clearly not necessary to find the target system's profile, but provides us with insight into many aspects of the system itself. What is the value for "Sec Since Boot? (Yours will be different from what is shown below.)
In what timezone is your target system operating?

```
[1] pmem 16:52:10> imageinfo
```

---

3. Next, enumerate running processes using **`pslist`**. With Rekall's pslist plugin, you can specify the methods in which processes are enumerated: PsActiveProcessHead, CSRSS, PspCidTable, Session and Handles.

   a. First run **`pslist`** using the **`PsActiveProcessHead`** method, shown below.

```
[1] pmem 16:52:10> pslist method="PsActiveProcessHead"
```

   What is the PID of your system's notepad.exe process? (Note: It will be different from those shown in the Step-by-Step review).

---

   b. Next run **`pslist`** in its default mode, using all five methods of process enumeration. Note the additional processes shown in this output, due to the additional parsing methods invoked by Rekall. Note any false positives that pslist generates below.

---

4. Enumerate the dlls of the "notepad.exe" process using Rekall's **dlllist**.

```
[1] pmem > dlllist proc_regex="Notepad.exe"
```

What is the command line with which notepad.exe was instantiated?

1. Now that the winpmem driver is loaded on the target system, we can begin to analyze current system state of the running system using a Rekall interactive session. Type the following command to invoke a Rekall interactive session referencing the loaded winpmem kernel driver.

```
C:\Program Files\Rekall> Rekal -f \\.\pmem --pager=notepad
```

```
C:\Program Files\Rekall>rekal -f \\.\pmem --pager=notepad

------------------------------------------------------------------------
The Rekall Memory Forensic framework 1.2.1 (Col de la Croix).

"We can remember it for you wholesale!"

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License.

See http://www.rekall-forensic.com/docs/Manual/tutorial.html to get started.
------------------------------------------------------------------------
[1] pmem 17:12:34>
```

2. Begin your triage of the live system with the **imageinfo** plugin. This plugin is clearly not necessary to find the target system's profile, but provides us with insight into many aspects of the system itself.
   What is the value for "Sec Since Boot? (Yours will be different from what is shown below.)
   In what timezone is your target system operating? **UTC.**

```
[1] pmem 21:25:55> imageinfo
----------------> imageinfo()
        Fact                          Value
-------------------- ------------------------------
Kernel DTB           0x1aa000
NT Build             9600.winblue_r3.140827-1500
NT Build Ex          9600.17328.amd64fre.winblue_r3
                     .140827-1500
Signed Drivers       -
Time (UTC)           2015-03-24 19:44:20+0000
Time (Local)         2015-03-24 19:44:20+0000
Sec Since Boot       907.953125
NtSystemRoot         C:\Windows

**************** Physical Layout ****************
  Phys Start      Phys End     Number of Pages
-------------- -------------- ---------------
0x000000001000 0x00000009f000 158
0x000000100000 0x000000102000 2
0x000000103000 0x0000bfee0000 785885
0x0000bff00000 0x0000c0000000 256
0x000100000000 0x0001b8800000 755712
```

3. Next, enumerate running processes using **pslist**. With Rekall's pslist plugin, you can specify the methods in which processes are enumerated: PsActiveProcessHead, CSRSS, PspCidTable, Session and Handles.

a. First run **pslist** using the **PsActiveProcessHead** method, shown below.

```
[1] pmem 16:52:10> pslist method="PsActiveProcessHead"
```

```
[1] pmem 15:19:34> pslist method="PsActiveProcessHead"
-----------------> pslist(method="PsActiveProcessHead")
  _EPROCESS           Name           PID   PPID   Thds     Hnds     Sess  Wow64          Start
-------------- -------------------- ----- ------ ------ -------- ------ ------ -------------------
--------
0xe0003563f900 System                   4      0     94        -      - False  2015-01-03 15:09:52
0xe0003775e040 smss.exe               316      4      2        -      - False  2015-01-03 15:09:52
0xe00037788900 csrss.exe              384    376      8        -      0 False  2015-01-03 15:09:53
0xe00037792380 wininit.exe            448    376      1        -      0 False  2015-01-03 15:09:53
0xe00037750080 csrss.exe              456    440      9        -      0 False  2015-01-03 15:09:53
0xe00037aa0080 svchost.exe            460    532     24        -      0 False  2015-01-03 15:09:55
```

b. Next run **pslist** in its default mode, using all five methods of process enumeration. Note the additional processes shown in this output, due to the additional parsing methods invoked by Rekall..

```
[1] pmem 16:52:10> pslist
```

```
0xe000b26bf900 TabTip.exe            1716    992     11        -      0 False  2015-03-23 23:18:46+0000 -
0xe000b2f56900 rekal.exe             1732    592      2        -      0 False  2015-03-24 03:16:23+0000 -
0xe000b25c8900 notepad.exe           2000   1484      1        -      1 False  2015-03-24 03:16:18+0000 -
0xe000b497c900 svchost.exe           2032    560     18        -      0 False  2015-03-23 23:15:28+0000 -
0xe000b529e080 WUDFHost.exe          2084    992      9        -      0 False  2015-03-23 23:15:28+0000 -
0xe000b3c30900 svchost.exe           2468    560      3        -      0 False  2015-03-23 23:15:28+0000 -
0xe000b3c2c900 msdtc.exe             2484    560     18        -      0 False  2015-03-23 23:15:28+0000 -
0xe000b3c4f900 svchost.exe           2528    560      7        -      0 False  2015-03-23 23:15:28+0000 -
0xe000b267e900 TabTip.exe            2580    992      0        -      0 False  2015-03-23 23:19:22+0000 2015-03-23 23:19
:22+0000
0xe000b26f7900 TabTip32.exe          2948   1716      1        -      0 True   2015-03-23 23:18:46+0000 -
0xe000b53ec900 dllhost.exe           3164    628      3        -      0 False  2015-03-23 23:15:42+0000 -
0xe000b26ad900 vmtoolsd.exe          3176   1484      6        -      0 False  2015-03-23 23:18:56+0000 -
```

4. Enumerate the dlls of the "Notepad.exe" process using Rekall's **dlllist**.

   What is the command line with which notepad.exe was instantiated?

```
[1] pmem > dlllist proc_regex="Notepad.exe"
```

```
[1] pmem 03:23:44> dlllist proc_regex="notepad.exe"
dlllist(proc_regex="notepad.exe")
**************************************************
notepad.exe pid: 2000
Command line : "C:\Windows\system32\NOTEPAD.EXE" C:\Users\sansforensics408\Desktop\Exercise_3.txt

     Base             Size             Load Reason/Count          Path
---------------- ---------------- ------------------------------ ----
0x7ff7eadb0000         0x3a000 LoadReasonStaticDependency     C:\Windows\system32\NOTEPAD.EXE
0x7ff9076b0000         0x1a6000 LoadReasonStaticDependency     C:\Windows\SYSTEM32\ntdll.dll
0x7ff906f00000         0x13a000 LoadReasonDynamicLoad          C:\Windows\system32\KERNEL32.DLL
0x7ff904e20000         0x10f000 LoadReasonStaticDependency     C:\Windows\system32\KERNELBASE.dll
0x7ff906a00000          0xa5000 LoadReasonStaticDependency     C:\Windows\system32\ADVAPI32.dll
0x7ff907080000         0x144000 LoadReasonStaticDependency     C:\Windows\system32\GDI32.dll
0x7ff907270000         0x171000 LoadReasonStaticDependency     C:\Windows\system32\USER32.dll
0x7ff906dd0000          0xa7000 LoadReasonStaticDependency     C:\Windows\system32\msvcrt.dll
0x7ff9071d0000          0x9e000 LoadReasonStaticDependency     C:\Windows\system32\COMDLG32.dll
0x7ff9053f0000         0x140f000 LoadReasonStaticDependency     C:\Windows\system32\SHELL32.dll
0x7ff8f8750000          0x7b000 LoadReasonStaticDependency     C:\Windows\system32\WINSPOOL.DRV
0x7ff904f30000         0x178000 LoadReasonStaticDependency     C:\Windows\system32\ole32.dll
0x7ff906ea0000         0x51000 LoadReasonStaticDependency     C:\Windows\system32\SHLWAPI.dll
0x7ff902850000         0x25a000 LoadReasonStaticDependency     C:\Windows\WinSxS\amd64_microsoft.windows.common-co
595b64144ccf1df_6.0.9600.17031_none_6242a4b3ecbb55a1\COMCTL32.dll
0x7ff9052b0000          0xc1000 LoadReasonStaticDependency     C:\Windows\system32\OLEAUT32.dll
0x7ff906800000          0x57000 LoadReasonStaticDependency     C:\Windows\SYSTEM32\sechost.dll
0x7ff906c90000         0x137000 LoadReasonStaticDependency     C:\Windows\system32\RPCRT4.dll
0x7ff906ab0000         0x1d6000 LoadReasonStaticDependency     C:\Windows\SYSTEM32\combase.dll
0x7ff903040000          0x9f000 LoadReasonStaticForwarderDepen C:\Windows\system32\SHCORE.DLL
```

Exercise: Key Takeaways

- Live memory analysis is a bleeding-edge technique that offers the examiner workarounds for situations were full physical memory acquisition is not possible or practical. Rekall Memory Forensic Framework offers this functionality by making use of the winpmem kernel driver.
- Rekall Memory Forensic Framework shares many of the same plugin names as the Volatility framework, but it is important to recognize that the methods employed by Rekall are often very different in nature in parsing of these memory structures and artifacts.

References:

[1] Rekall Memory Forensic Framework Documentation. http://www.rekall-forensic.com/docs.html

This page intentionally left blank.

Exercise 4 – Memory Acquisition with winpmem

Objectives

- Capture a memory image of a target system using an open-source free memory acquisition tool
- Introduce different memory acquisition methods (\\Device\Physical Memory & PTE remapping) and output formats and methods offered by winpmem
- Validate the resultant memory image by scanning for Windows memory structures using the Rekall Memory Forensic Framework plugins

Capture memory image

The Rekall development team continues to push the capabilities of their memory acquisition tool, winpmem, that we just used to perform live system memory analysis. In addition to being able to send the captured image outbound to a network share, winpmem can acquire the pagefile during memory acquisition. If run on a host system running virtualization software, winpmem acquires the memory in such a way that allows Rekall to analyze the virtual machines in their own virtualized memory ranges.

1. Launch your Windows 8.1 VM with VMware Workstation/Player/Fusion.

2. If needed, login to the Windows 8.1 VM with the credentials below:

```
User: sansforensics
Password: forensics
```

3. Launch your Windows 8.1 VM with VMware Workstation/Player/Fusion.

4. Open a command shell with **Administrative privileges** by right-clicking on the **cmd** icon and select "**Run as Administrator**".

5. Change directories via command line to the **C:\Program Files\Rekall** directory.

```
C:\Windows\System32\> cd C:\Program Files\Rekall
```

- Type the following command in the command window to see the available options you can use with winpmem:

```
C:\Program Files\Rekall> winpmem_1.6.2.exe -h
```

```
c:\Program Files\Rekall>winpmem_1.6.2.exe -h
Winpmem - A memory imager for windows.
Copyright Michael Cohen (scudette@gmail.com) 2012-2014.

Version 1.6.2 Nov  1 2014
Usage:
  winpmem_1.6.2.exe [option] [output path]

Option:
  -l    Load the driver and exit.
  -u    Unload the driver and exit.
  -d [filename]
        Extract driver to this file (Default use random name).
  -h    Display this help.
  -w    Turn on write mode.
  -0    Use MmMapIoSpace method.
  -1    Use \\Device\PhysicalMemory method (Default for 32bit OS).
  -2    Use PTE remapping (AMD64 only - Default for 64bit OS).
  -3    Use PTE remapping with PCI instrospection (AMD64 Only).
  -e    Produce an ELF core dump.
  -p    Also acquire the pagefile.
        This flag may be followed by the pagefile path.
```

6. We will be making use of the "-e" and "-p" options. Type the following command to create an elf memory dump and send it to your SIFT 3.0 workstation, seen on the network as "SIFTWORKSTATION":

```
C:\Program Files\Rekall> winpmem_1.6.2.exe -p -e
\\SIFTWORKSTATION\cases\image.elf
```

```
C:\Program Files\Rekall>winpmem_1.6.2.exe -p -e \\SIFTWORKSTATION\cases\image.elf
Will write an elf core dump.
Extracting driver to C:\Users\SANSFO~2\AppData\Local\Temp\pme1885.tmp
Driver Unloaded.
Loaded Driver C:\Users\SANSFO~2\AppData\Local\Temp\pme1885.tmp.
Deleting C:\Users\SANSFO~2\AppData\Local\Temp\pme1885.tmp
Will write an elf coredump.
CR3: 0x00001AA000
 5 memory ranges:
Start 0x00001000 - Length 0x0009E000
Start 0x00100000 - Length 0x00002000
Start 0x00103000 - Length 0xBFDDD000
Start 0xBFF00000 - Length 0x00100000
Start 0x100000000 - Length 0xB8800000
Acquitision mode PTE Remapping

00% 0x00001000 .

00% 0x00100000 .

00% 0x00103000 ..............................................
```

Analyze the ELF Captured file with Rekall in the SIFT 3.0 VM

For analysis of our newly created memory dump, we will use the Rekall Memory Forensic Framework v.1.2.1 on the SIFT 3.0 virtual machine.

1. **Verify the Captured Elf file exists in your SIFT 3.0 VM /cases directory**
   - In a terminal window, navigate to the /cases directory and list contents.

```
$ cd /cases
$ ls -al *.elf
```

```
sansforensics@siftworkstation:/cases$ ls -al *.elf
-rwxrwxr-x 1 sansforensics sansforensics 14381191876 Dec 25 15:25 demo.elf
-rwxr--r-- 1 sansforensics sansforensics  4294431333 Jan  3 13:58 image1.elf
-rwxrwxr-x 1 sansforensics sansforensics 12884366020 Jan  3 13:27 image.elf
```

2. **Launch an interactive Rekall analysis session referencing the captured .elf file, invoking `less` as a "pager" to handle output that overruns the length of a page of output.**

```
$ rekal -f image.elf --pager=less
```

```
sansforensics@siftworkstation:/cases$ rekall -f image.elf --pager=less

----------------------------------------------------------------------------
The Rekall Memory Forensic framework 1.2.0 (Col de la Croix).

"We can remember it for you wholesale!"

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License.

See http://www.rekall-forensic.com/docs/Manual/tutorial.html to get started.
----------------------------------------------------------------------------
[1] image.elf 14:55:02>
```

If the above command is run with the "-v" option, you are able to see the profile selection and the identification of both the pagefile that is a part of the image file and the elf image format itself.

```
INFO:root:Loading pagefile into physical offset 0x140010000
DEBUG:root:Succeeded instantiating Elf64CoreDump
DEBUG:root:Voting round with base: Elf64CoreDump
```

3. **Enumerate the Page Files being referenced by the target system using Rekall**

```
[1] image.elf > pagefiles
```

```
[1] image.elf 15:02:49> pagefiles
----------------------> pagefiles()
_MMPAGING_FILE Number  Size (b)        Filename
-------------- ---   ---------- --------------------
0xe001d1aae130      0 8589934592 \??\C:\pagefile.sys
0xe001d2456150      1  268435456 \??\C:\swapfile.sys
[1] image.elf 15:03:21>
```

4. Run the following commands and observe how the plugins with large output launch scite for easier analysis: **`pslist`**, **`handles`**, **`filescan`**
5. Since we have run all of these plugins in the same interactive shell using Rekall, we can now print out the saved session information that Rekall uses to speed subsequent plugin runs. Type the following and note the specific details that are retained specific to this memory image:

```
[1] image.elf > print session
```

```
[1] image.elf 15:18:04> print session
Rekall Memory Forensics session Started on Sat Jan  3 15:00:17 2015.

Config:
{
  __dummy = False
  autodetect = <rekall.utils.JITIterator object at 0x2ae4c90>
  autodetect_threshold = 1.0
  base_filename = image.elf
  buffer_size = 20971520
  cache = {
    ObjectTypeMap = <Array 0 x Pointer @ 0xF80374747740>
    ObpInfoMaskToOffset = [0, 32, 32, 64, 16, 48, 48, 80, 32, 64, 64, 96, 48, 80, 80, 112, 16, 48, 48, 80, 32, 64, 64, 96, 48, 80, 80,
8, 80, 32, 64, 64, 96, 48, 80, 80, 112, 64, 96, 96, 128, 32, 64, 64, 96, 48, 80, 80, 112, 64, 96, 96, 128, 80, 112, 112, 144, 4, 36, 3
8, 100, 52, 84, 84, 116, 20, 52, 52, 84, 36, 68, 68, 100, 52, 84, 84, 116, 68, 100, 100, 132, 20, 52, 52, 84, 36, 68, 68, 100, 52, 84,
 68, 68, 100, 52, 84, 84, 116, 68, 100, 100, 132, 84, 116, 116, 148, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
, 0, 0. 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ...
  debug = False
  dtb = 1744896
  ept = None
  filename = image.elf
  help = False
  kernel_base = 18446735292452339712
  logging = warn
  max_collector_cost = 4
  name_resolution_strategies = ['Module', 'Symbol', 'Export']
  nocolors = False
  notebook_dir = /home/sansforensics
  pagefile = []
  plugin = []
  profile_path = ['https://raw.githubusercontent.com/google/rekall-profiles/master', 'http://profiles.rekall-forensic.com']
  quiet = False
  renderer = text
  session_id = 1
  session_list = [<rekall.session.InteractiveSession object at 0x2b59950>]
  session_name = Default session
  timezone = UTC
  verbose = False
}

[1] image.elf 15:18:08>
```

Exercise: Key Takeaways

- Great work acquiring memory from a target Windows system. Rekall had no problem parsing the Windows 8.1 x64 system memory image. Gaining familiarity with winpmem for acquisition as you have adds to your tools arsenal - those that you KNOW will work.
- We will continue to discover the differences that currently exist between the Volatility Framework and Rekall as the course goes on. We, as examiners, win when we have more than one tool that gets the job done.

This page intentionally left blank.

Exercise 5 – Unstructured Memory Analysis

Objectives

- Conduct a forensic memory analysis using both unstructured and structured parsing techniques
- Investigate extracted network packets for relevant forensic artifacts
- Apply file system forensics to artifacts recovered from system memory
- Identify key aspects of malware through the use of Bulk Extractor and Volatility
- Practice Registry Analysis to identify forms of persistence used by the malware

Exercise Part 1: Preparation

1. Prior to starting your SIFT VM, consider adding additional memory and CPU cores via the VMWare Settings to enhance tool performance during this exercise.

2. Start a terminal and change into the /cases directory and create the **output** directory.

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

3. Using Bulk Extractor, run the default scanners with the additional **wordlist** scanner against the first memory image we will be analyzing **win7crypto.vmem**.

```
$ bulk_extractor -e wordlist -o /cases/output/ben /cases/win7crypto.vmem
```

Exercise Part 1: Applications of Memory Analysis to a Criminal Investigation– Questions

Part I. Overview: Ben Bitdiddle's wife has disappeared and he is a suspect. We suspect Ben was having an affair with "Jess". We have obtained a memory image of his computer, **win7crypto.vmem**. Run bulk extractor on this memory image and use the **BEViewer** to view the results and answer the following questions.

```
$ BEViewer &
```

1. **Bulk_Extractor Output Analysis**
   Using the Bulk Extractor output, answer the following questions. Remember your weapons arsenal includes BEViewer and Wireshark (and whatever command-line kung fu you choose to wield).

a. Viewing the **email_histogram.txt** report via BE Viewer, determine Jess' email address, based on frequency of occurrence.

_____

b. Viewing the **url_facebook-id.txt** report via BE Viewer, determine Jess' Facebook page, based on frequency of occurrence. (This may require that you visit the urls to resolve id to username.)

_____

c. The password to Ben's account was "ke$ha". Search through the **wordlist** via BE Viewer, find any related candidate passwords in the memory image?

_____

d. Did Ben perform any web searches pertinent to this investigation? If so, what did he search for?

_____

_____

e. What was most likely Ben's IP address? (Verify this by viewing the **packets.pcap** in Wireshark.)

_____

f. Extract a jpg image from the tcp stream referenced in or around packet 283 from host 23.15.7.10 with a frame length of 380 bytes in the **packets.pcap** file using Wireshark's export function, saving as a jpg file to your local directory. What is the size of this image file?

_____

g. What were the MAC addresses of Ben's computer and his router? (Extra Credit. There is a MAC address present in the memory image for a non-VMware device. What's the MAC address and who made the device?)

_____

h. Using the **windirs.txt** file, find the creation time for **mysecretdata.tc**, a TrueCrypt volume being actively accessed at the time this memory image was created.

_____

Exercise Part 2: Preparation

Part II. Overview: Cridex is a financial banking Trojan capable of credential stealing, remote access and screen captures. It typically infects its hosts via malicious attachments and embedded URL links in spam emails. In March 2012, over one hundred financial institutions were hit by this Trojan, which has capabilities similar to Zeus and SpyEye.

1. Start a terminal and change into the /cases directory and ensure the **`output`** directory in this directory.

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

2. Using Bulk Extractor, run the default scanners against the first memory image we will be analyzing **`cridex.vmem`**. For faster processing of the image, we will **NOT** be invoking the wordlist scanner.

```
$ bulk_extractor -o /cases/output/cridex  /cases/cridex.vmem
```

Exercise Part 2: Applications of Memory Analysis in Malware Investigations: Questions

1. **Unstructured Analysis**
   Using the Bulk Extractor output, answer the following questions. Remember your weapons arsenal includes BEViewer and Wireshark (and whatever command-line kung fu you choose to wield).

   a. Viewing the **`url_histogram.txt`** report via BE Viewer, list the banking urls that appear in the top 20 most frequently occurring hits.

   _____

   b. Use Virus Total to check the reputation of some of the urls that include IP addresses with the port 8080. What are these IPs associated with?

   _____

   c. Open the **`packets.pcap`** file from the Bulk Extractor output with Wireshark. Which remote system is shown sending data to the local host via port 8080?

   _____

2. **Image Identification**

- Using Volatility, run the **imageinfo** plugin against the **cridex.vmem**.
  a. When was the memory image created?

  _____

  b. What profile should be specified when analyzing this image?

  _____

3. **Check for Persistence via the Registry**
   - Though this persistence technique has been around for many years, modern malware makes use of the "Run" key in the Software and User hives frequently. Dump the "Run" key values by using the **printkey** plugin from Volatility using the below command:

```
$ vol.py -f cridex.vmem --profile=WinXPSP3x86 printkey
-K "Software\Microsoft\Windows\CurrentVersion\Run"
```

(Although we haven't covered the printkey plugin in class yet, we will be covering this plugin and more in greater detail in upcoming modules.)

  a. When was this key last updated?

  _____

  b. Based on the values of this key, what program will be executed when the user account "Robert" logs in?

  _____

4. **Tying Network Activity to a Process**

- Based on the **packets.pcap** file carved by Bulk Extractor, this system was actively communicating with a malicious remote system. Let's attempt to tie that activity to a specific process.

  a. Enumerate network connections (walking the singly linked list using **connections** and scanning for specific TCP connection pool structure using **connscan**). What process (name and PID) is associated with the remote connection?

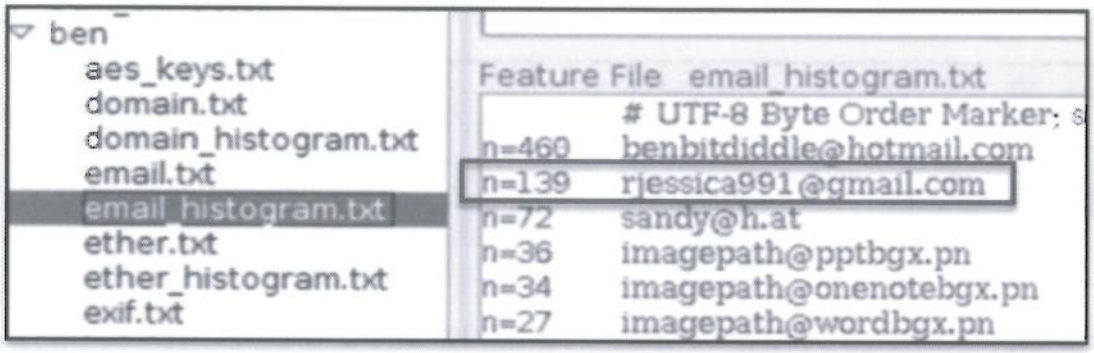
  _____

  b. By running **malfind** against this process, are there any clear indications of code injection? Dump the identified memory sections and upload to Virustotal for further analysis.

  _____

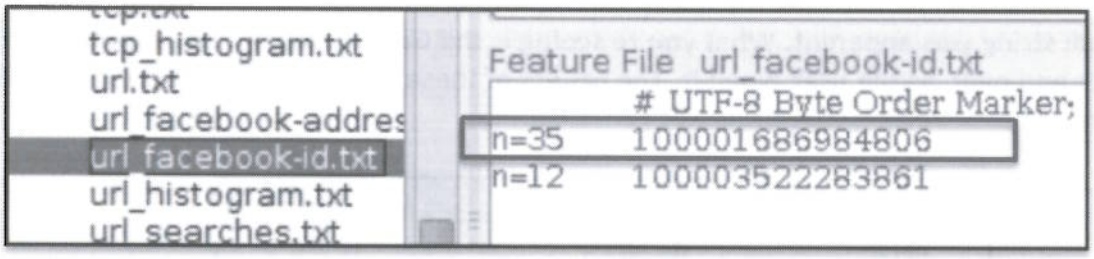
1. Bulk_Extractor Output Analysis

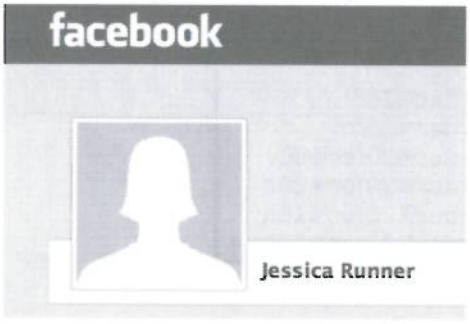
a. Viewing the **`email_histogram.txt`** report via BE Viewer, determine Jess' email address, based on frequency of occurrence.



Ben's address, benbitdiddle, is at the top of the list with 460 references. Just below that is the second most common e-mail address, rjessica991@gmail.com, with 139 references. Given that we're looking for "Jess", it's quite probable that Jess' e-mail address is `rjessica991@gmail.com`.

b. Viewing the **`url_facebook-id.txt`** report via BE Viewer, determine Jess' Facebook page, based on frequency of occurrence. (This may require that you visit the urls to resolve id to username.)





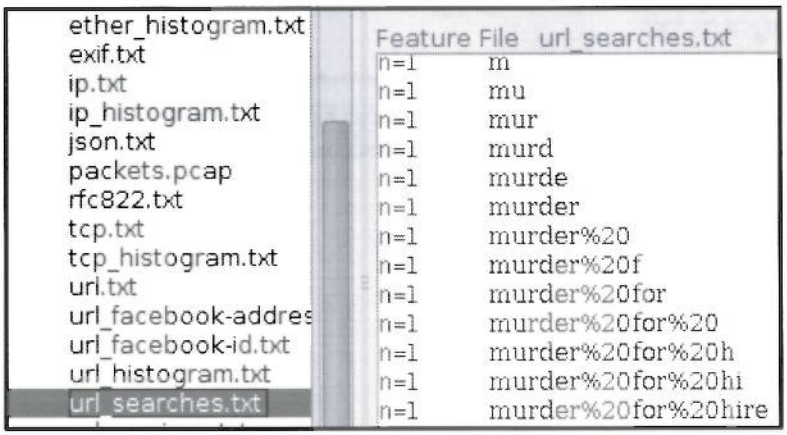
Although we could look through the complete list of URLs, Bulk Extractor helpfully pulls them out into a separate file, `url_facebook-id.txt`.

We find two id numbers in that file. Appending each to a Facebook URL, https://www.facebook.com/, we are able to find the two Facebook pages. Ben is 100003522283861 and Jessica is 100001686984806. As an aside, we note that Ben remains a fan of Ke$ha, even on Facebook.

c. The password to Ben's account was "ke$ha". Search through the `wordlist` via BE Viewer or CLI, find any related candidate passwords in the memory image?

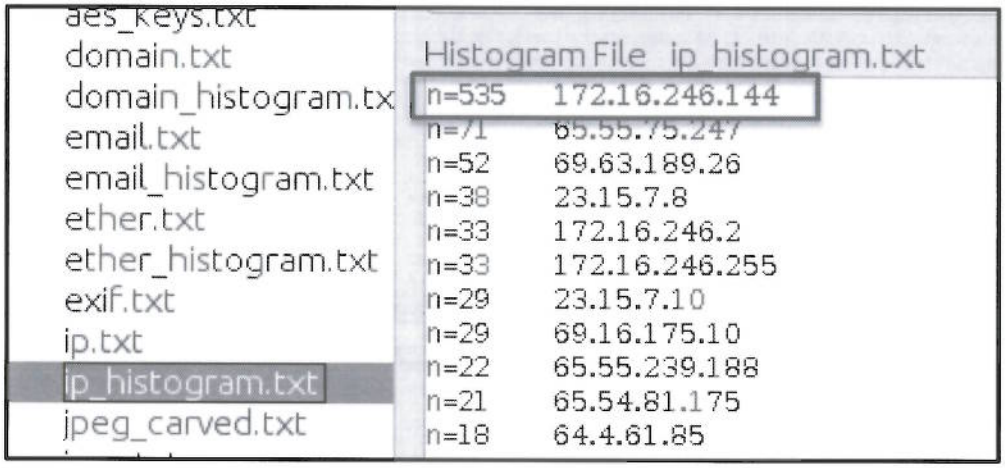
d. Did Ben perform any web searches pertinent to this investigation? If so, what did he search for?

The file `url_searches.txt` lists the URLs that appear to be search queries found in the memory image.
Keep in mind these weren't necessarily searched for. They could have just appeared on a web page as links to a search query. (For example, the instructors never searched for anything related to Lady Gaga.) But the repeated queries which end up being "murder for hire" are rather suspicious in a missing person's case. Notice how the query was repeated, with one letter being added at a time, until the full string was apparent. What you're seeing is the Google search client sending the query string, over and over, as the user typed it. The results of these repeated sends were the search suggestions.

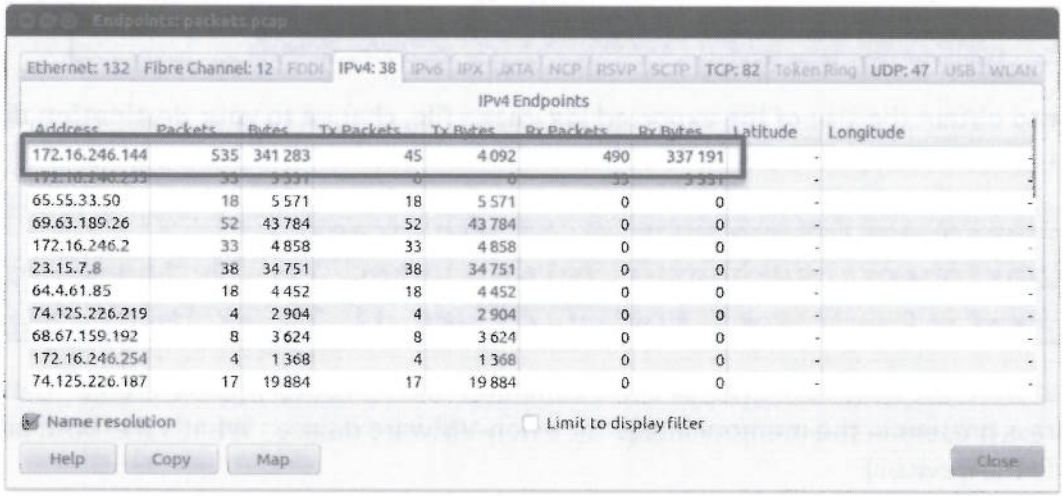


e. What was most likely Ben's IP address? (Verify this by viewing the **`packets.pcap`** in Wireshark.)

The file ip_histogram.txt shows that far and away the most common IP address carved from the win7crypto.vmem file was 172.16.246.144. This is a private IP address and is often used by VMware machines.

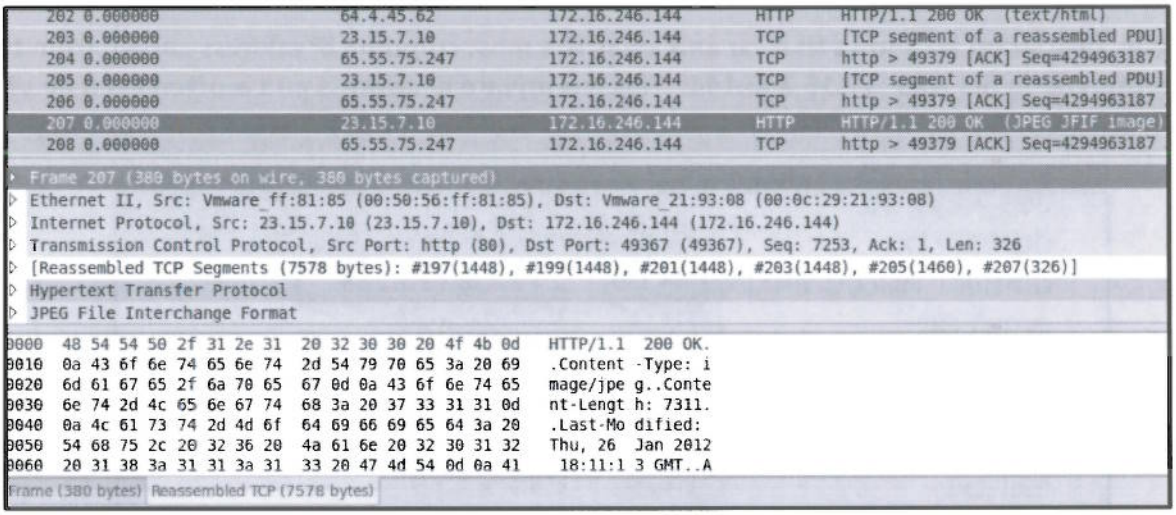


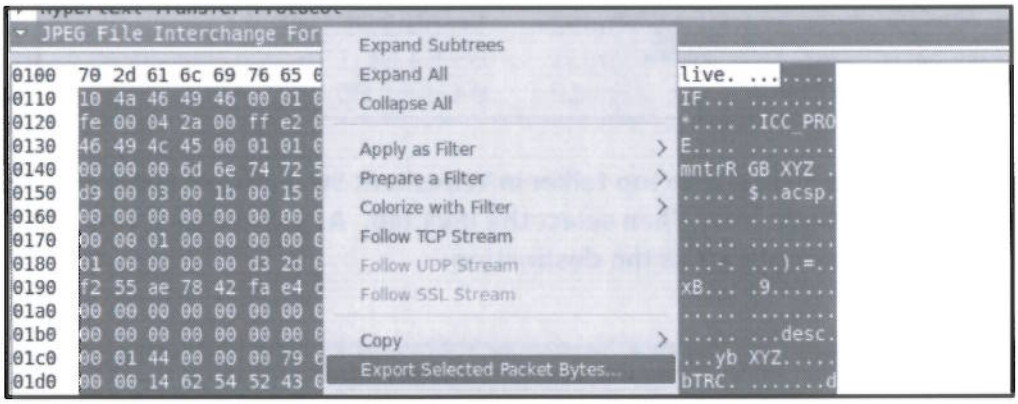
This IP address can also be seen as the top talker in Wireshark Statistics on endpoints. From the toolbar, select Statistics>Endpoints. Then select the IPv4 tab. As you can see, the 172.16.246.144 has 535 packets, 490 of them with it as the destination.



f. Extract the jpg image from the tcp stream referenced in or around packet 283 from host 23.15.7.10 with a frame length of 380 bytes in the **`packets.pcap`** file using Wireshark's export function, saving as a jpg file to your local directory. What is the size of this image file?

Select packet 283 and navigate within the details pane to the `JPEG File Interchange Format`**. Right click on this header and select** `Export Selected Packet Bytes` **and save as a jpg file.**



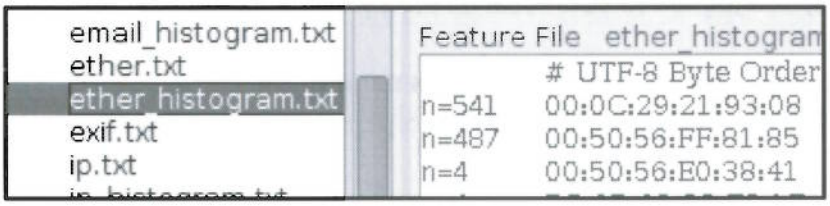


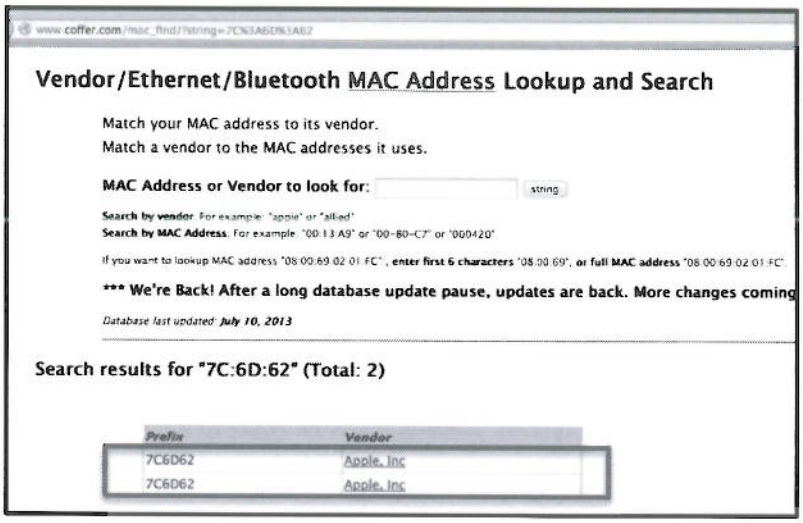
In order to obtain the size of the exported jpg image file, change to your destination directory and type `ls -lh <imagefile>`**.**

```
sansforensics@ubuntu:~$ cd win7crypto/
sansforensics@ubuntu:~/win7crypto$ ls -lh image.jpg
-rw-r--r-- 1 root root 7.2K Jan 11 10:54 image.jpg
```

g. What were the MAC addresses of Ben's computer and his router? (Extra Credit. There is a MAC address present in the memory image for a non-VMware device. What's the MAC address and who made the device?)

Looking at the histogram of MAC addresses found, we see that far and away the two most common were 00:0C:29:21:93:08 and 00:50:56:FF:81:85. Both of those are Vmware devices. There are lots of web-based lookups for MAC addresses, such as http://www.coffer.com/mac_find/**.**





Extra Credit. The MAC address that is not related to VMWare is the 7C:6D:62, seen above as belonging to Apple.

h. Using the `windirs.txt` file, find the creation time for `mysecretdata.tc`, a TrueCrypt volume being actively accessed at the time this memory image was created.

```
$ cd /cases/output/ben
$ cat windirs.txt |grep -i mysecretdata
```

```
18170880        mysecretdata.tc <fileobject src='mft'><atime>2012-02-16T12:06:00Z</atime><attr_flags>8224</attr_flag
s><crtime>2012-02-16T12:06:00Z</crtime><ctime>2012-02-10T16:56:24Z</ctime><filename>mysecretdata.tc</filename><files
ize>1000000000000</filesize><filesize_alloc>1048576</filesize_alloc><lsn>168615081</lsn><mtime>2012-02-10T16:56:24Z<
/mtime><nlink>4</nlink><par_ref>376</par_ref><par_seq>2</par_seq><seq>2</seq></fileobject>
```

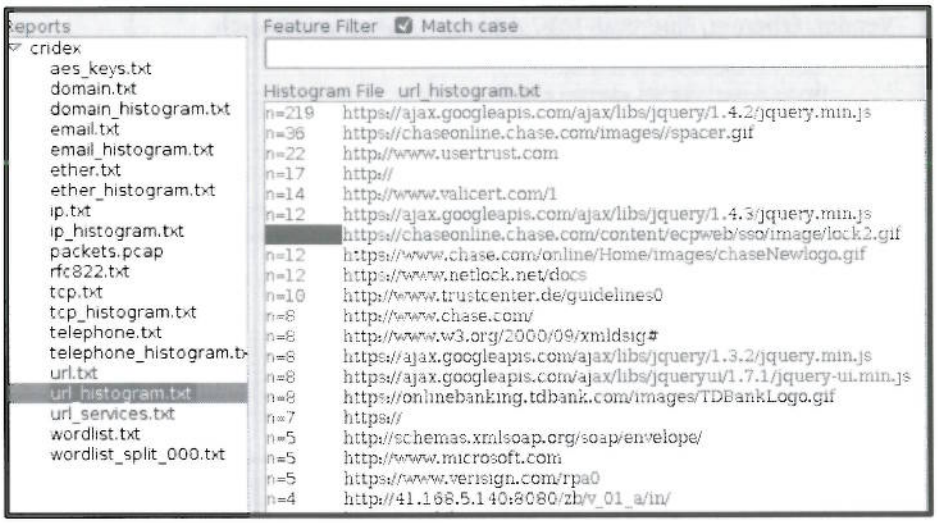
Exercise Part 2: Applications of Memory Analysis in Malware Investigations: Questions with Step-by-Step

1. Unstructured Analysis

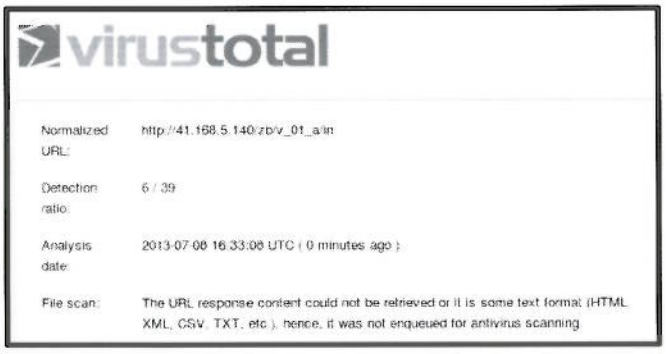
Using Bulk Extractor, run the default scanners against the first memory image we will be analyzing `cridex.vmem`. For faster processing of the image, we will **NOT** be invoking the wordlist scanner.

a. Viewing the `url_histogram.txt` report via BE Viewer, list the banking urls that appear in the top 20 most frequently occurring hits.

The Cridex Trojan contains the structure of the banking organizations websites, as seen in the data content pane of the Bulk Extractor, if you select one of these banking site url hits. This "website templating" allows for easier identification of valuable fields to send back to the C&C server.



b. Use VirusTotal to check the reputation of some of the urls that include IP addresses with the port 8080. What are these IPs associated with?

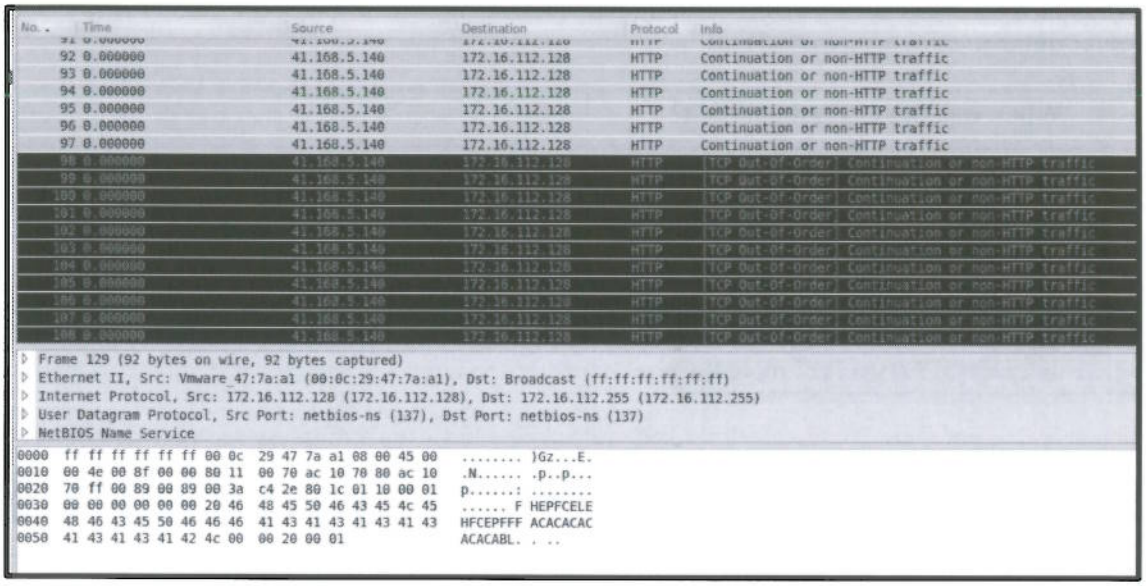


Virustotal allows you to scan a url using the multiple AV software products, in addition to allowing you to gain a historical view on whether the urls have been seen associated with past malware. In this case, only 6 of 39 associate this url with malware at the time of publishing.

c. Open the packets.pcap file from the Bulk Extractor output with Wireshark. Which remote system is shown sending data to the local host via port 8080?

Remote IP - 41.168.5.140

Note that the packets shown in Wireshark from the pcap file do not have time/datestamps. Recreation of a "conversation" would need to be based on sequence number. One tool that does this is TCPflow, also written by Simson Garfinkel.



2. Image Identification

- Using Volatility, run the **`imageinfo`** plugin against the **`cridex.vmem`**.
  a. When was the memory image created?

07/22/2012 02:45:08 UTC

  b. What profile should be specified when analyzing this image?

WinXPSP3x86

```
Volatile Systems Volatility Framework 2.2
Determining profile based on KDBG search...

          Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with Win
XPSP2x86)
                     AS Layer1 : JKIA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (/mnt/hgfs/Desktop/FOR526_0520
13/Additional_Exercises/cridex.vmem)
                      PAE type : PAE
                           DTB : 0x2fe000L
                          KDBG : 0x80545ae0L
          Number of Processors : 1
     Image Type (Service Pack) : 3
                KPCR for CPU 0 : 0xffdff000L
             KUSER_SHARED_DATA : 0xffdf0000L
           Image date and time : 2012-07-22 02:45:08 UTC+0000
     Image local date and time : 2012-07-21 22:45:08 -0400
```

3. Check for Persistence via the Registry

- Though this persistence technique has been around for many years, modern malware makes use of the "Run" key in the Software and User hives frequently.

Dump the "Run" key values by using the **`printkey`** plugin from Volatility using the below command:

```
$ vol.py -f cridex.vmem --profile=WinXPSP3x86 printkey
-K "Software\Microsoft\Windows\CurrentVersion\Run"
```

(Although we haven't covered the printkey plugin in class yet, we will be covering this plugin and more in greater detail in upcoming modules.)

a. When was this key last updated?

07/22/2012 02:31:51 UTC

b. Based on the values of this key, what program will be executed when the user account "Robert" logs in?

```
----------------------------
Registry: \Device\HarddiskVolume1\Documents and Settings\Robert\NTUSER.DAT
Key name: Run (S)
Last updated: 2012-07-22 02:31:51 UTC+0000

Subkeys:

Values:
REG_SZ        KB00207877.exe  : (S) "C:\Documents and Settings\Robert\Application Data\KB00207877.exe"
```

C:\Documents and Settings\Robert\Application Data\KB00207877.exe

4. Tying Network Activity to a Process

Based on the **packets.pcap** file carved by Bulk Extractor, this system was actively communicating with a malicious remote system. Let's attempt to tie that activity to a specific process.

a. Enumerate network connections (walking the singly linked list using **connections** and scanning for specific TCP connection pool structure using **connscan**). What process (name and PID) is associated with the remote connection?

```
$ vol.py -f cridex.vmem --profile=WinXPSP3x86 connscan
```

```
Offset(P)  Local Address             Remote Address            Pid
---------- ------------------------ ------------------------- ---
0x02087620 172.16.112.128:1038      41.168.5.140:8080         1484
0x023a8008 172.16.112.128:1037      125.19.103.198:8080       1484
```

```
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                 Exit
---------- ------------------ ------ ------ ------ -------- ------ ------ ------------------- ----
0x823c89c8 System                    4      0     53      240 ------      0
0x822f1020 smss.exe                368      4      3       19 ------      0 2012-07-22 02:42:31
0x822a0598 csrss.exe               584    368      9      326      0      0 2012-07-22 02:42:32
0x82298700 winlogon.exe            608    368     23      519      0      0 2012-07-22 02:42:32
0x81e2ab28 services.exe            652    608     16      243      0      0 2012-07-22 02:42:32
0x81e2a3b8 lsass.exe               664    608     24      330      0      0 2012-07-22 02:42:32
0x82311360 svchost.exe             824    652     20      194      0      0 2012-07-22 02:42:33
0x81e29ab8 svchost.exe             908    652      9      226      0      0 2012-07-22 02:42:33
0x823001d0 svchost.exe            1004    652     64     1118      0      0 2012-07-22 02:42:33
0x821dfda0 svchost.exe            1056    652      5       60      0      0 2012-07-22 02:42:33
0x82295650 svchost.exe            1220    652     15      197      0      0 2012-07-22 02:42:35
0x821dea70 explorer.exe           1484   1464     17      415      0      0 2012-07-22 02:42:36
0x81eb17b8 spoolsv.exe            1512    652     14      113      0      0 2012-07-22 02:42:36
```

b. By running **malfind** against this process, are there any clear indications of code injection? Dump the identified memory sections and upload to Virustotal for further analysis.

```
$ vol.py -f cridex.vmem --profile=WinXPSP3x86 malfind -p 1484
```

```
Volatile Systems Volatility Framework 2.2
Process: explorer.exe Pid: 1484 Address: 0x1460000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 33, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x01460000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x01460010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x01460020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x01460030  00 00 00 00 00 00 00 00 00 00 00 00 e0 00 00 00   ................

0x1460000 4d               DEC EBP
0x1460001 5a               POP EDX
0x1460002 90               NOP
0x1460003 0003             ADD [EBX], AL
0x1460005 0000             ADD [EAX], AL
0x1460007 000400           ADD [EAX+EAX], AL
0x146000a 0000             ADD [EAX], AL
0x146000c ff               DB 0xff
0x146000d ff00             INC DWORD [EAX]
0x146000f 00b800000000     ADD [EAX+0x0], BH
0x1460015 0000             ADD [EAX], AL
0x1460017 004000           ADD [EAX+0x0], AL
```

Exercise: Key Takeaways

1. Through the use of unstructured memory analysis, we were able to isolate specific remote IP addresses previous associated with other banking Trojans. In viewing carved packets from the memory image, we show active communication between this local host and a malicious site.
2. By taking the information gained in unstructured memory analysis, we parsed process structures and network connections to determine the presence of code injection in the Explorer.exe process. If we examine this injected code further, we can see based on its function calls that is has Internet communication and credential stealing capabilities. More on this from a write up can be found here at M86 Security Labs blog: http://labs.m86security.com/2012/03/the-cridex-trojan-targets-137-financial-organizations-in-one-go/

This page intentionally left blank.

Exercise 6 – Page File Analysis and YARA Rules

Objectives

- Learn to use page_brute to analyze the paging file in Windows
- Gain experience using YARA
- Learn to create YARA signatures to perform custom carving of memory

Exercise Preparation

- Start a terminal and change into the **/cases/exercise6** directory and unzip **insider_pagefile.zip**.

```
$ cd /cases/exercise6
$ unzip insider-pagefile.zip
```

Exercise Part I: Analyze Page File

1. **Page File Identification**
   - Open a terminal and change directory to ~/page_brute. The page_brute tool is still in beta. As such, it does not operate well from outside of its own directory. Technically, this would just require you to specify the location of the default YARA rules file, but we're anti-typo and this is a much easier option.
   - Ensure that the sample page file in /cases/exercise6 has been unzipped. The target file is **insider-pagefile.sys**.
   - Run the Linux **file** command against insider-pagefile.sys. What is the type of the file?

     _____

   - Based on this identification, do you think Linux will be any help in parsing this file directly? Does it appear to have any underlying file structure?

     _____

2. **Page File Examination**
   - Try dumping out a portion of the page file using the hexdump command. Using the hex dump is a good way to examine unstructured data. Run the following command to dump the first 4096 bytes (4k) of the page file. You want to examine the page file 4k at a time since this is the default page size in Windows.

```
hexdump -C -n 4096 /cases/exercise6/insider-pagefile.sys
```

- Does the hex dump contain any notable data?

_____

- You can examine additional portions of the page file from the command line 4k at a time by adding the –s switch to skip some number of bytes (the number to skip should be a multiple of 4 bytes).

```
hexdump -C -n 4096 -s 4096 /cases/exercise6/insider-pagefile.sys
```

Exercise Part II: Analysis Using page_brute

1. **Run page_brute with default yara signatures**
   - Use page_brute with the following options to parse the insider's paging file.

```
./page_brute-BETA.py -f /cases/exercise6/insider-pagefile.sys
-o ~/insider-case
```

The –f option specifies the input file while the –o option specifies the output directory.

NOTE: This command can take considerable time to complete. If you are working on this live in class, you'll want to use the pre-cooked data provided in /cases/insider-pageBrute-precooked.zip. Change directory back to your home directory and unzip this file to analyze the output data. You can do this with the following two commands:

```
cd ~
unzip /cases/exercise6/insider-pageBrute-precooked.zip
```

   - One annoyance with page_brute is that is creates all directories with 777 permissions. This causes the Ubuntu terminal to display the directory names in an odd fashion. Use the following command to fix that. In the example command below, you are running this from the directory containing the insider-case directory (your home directory in this case). Every directory found in that directory will now be 755 permissions (yielding better security and a normal display).

```
find insider-case -type d -exec chmod 755 {} \;
```

2. **Review Output/Precooked Data**
   - Change directory into ~/insider-case/. What types of data did the default parsers for page_brute discover?

_____

_____

- Change directory to the ~/insider-case/webartifact_javascript directory and run the ll command. What size are the files?

_____

- Examine the file 274286.block. You could open it in the Bless hex editor or simply use the hexdump command to dump the file. How would you characterize the file contents?

```
hexdump -C 274286.block
```

_____

- Could exploited web pages or other malicious file types be located in the page file?

_____

- Change directory to ~/insider-case/webartifact_html and run the ll command. Note that the files are all again 4096 bytes.

- Examine the data in the file 70488.block using the hexdump command. How would you characterize this data? How might this be useful in an investigation?

```
hexdump -C 70488.block
```

_____

Exercise Part III: Examine and Create YARA rules

In this section, we'll learn more about the structure of YARA rules by dissecting the rules written by another author. YARA is a complex and powerful language. This lab will only scratch the surface of its capabilities. You are highly encouraged to continue exploring YARA on your own outside of class.

1. **Examine default_signatures.yar**

- Change directory back to the ~/page_brute directory. Examine the YARA rules defined by default for page_brute using the grep command (shown below). The default rules are stored in default_signatures.yar. This should give you some idea of the types of artifacts that page_brute can detect.

```
grep rule default_signatures.yar
```

- Examine the structure of a rule by opening the file in scite. In this exercise, we'll only consider the first rule, administrative_share_abuse.

```
scite default_signatures.yar &
```

```
rule administrative_share_abuse
{
        meta:
                author="@matonis"
                description="syntax for accessing adminstrative shares"
        strings:
                $s0 = /(copy|del|psexec|net)/ nocase
                $s1 = "\\c$\\windows\\system32\\" nocase
                $s2 = "\\c$\\system32\\" nocase
                $s3 = "\\admin$\\" nocase
        condition:
                $s0 and (any of ($s1,$s2,$s3))
}
```

- The meta portion of the rule should be fairly self-explanatory. This section is completely optional and contains fields for the rule author as well as a short description of the rule.

- Examine the $s0 portion of the rule. Which commands does the rule author look for?
  Note: the nocase modifier is used to indicate that the string should be parsed case insensitive.
  Note: the | symbol is a symbolic OR. It allows many strings to be expressed in a single variable.

  _____

- Examine strings $s1, $s2, and $s3. Which directories does the author look for the commands identified above to be run in? Note: the double backslash is required because the backslash is an escape sequence in the YARA rule language.

  _____

- Finally, examine the condition statement. Under which conditions will this rule be matched?

  _____

  _____

2. Building a new YARA rule.

- In this case, you'll build a YARA rule to detect executable files in the page file. This is sometimes indicative of a downloaded or copied executable. When executables are run from a local drive, they are not paged to disk (since they are already on the disk). Interesting indeed!

- We could signature on something interesting like the string "This program cannot be run in DOS mode". However, this string will not provide additional information about the file, such as the filename. Instead, we'll focus on two pieces of data that should be part of the file VERSION_INFO. The two strings you want to match are 'FileVersion' and 'InternalName'. Both of these strings are Unicode, so we'll have to use the 'wide' modifier to the YARA rule.

- In the page_brute directory, use scite to create a new rule file called paged_exes.yar using the following command:

```
scite paged_exes.yar &
```

- In the new file, enter this text to create the shell for the new rule:

```
rule paged_exes {
        meta:
                author="FOR526 student"
                description="find paged exes using data from the VERSION_INFO structure"
        strings:
                $s0 = ""
                $s1 = ""
        condition:
                # Enter your own condition here
}
```

- Populate $s0 and $s1 with the appropriate strings to search for. Then, write a condition string that can be used to detect the VERSION_INFO structure in executable files paged to disk. Since the strings are stored as Unicode, don't forget to apply the 'wide' modified after each string. If you are finding this too difficult, check out the solutions for the lab to get the exact syntax.

- When you are done, save the file and exit scite.

Exercise Part IV: Run page_brute with the New Rule

1. In the page_brute directory, run the following command to parse the page file, this time looking for paged executable files:

```
./page_brute-BETA.py -r paged_exes.yar -o ~/insider-exes    \
-f /cases/exercise6/insider-pagefile.sys
```

Note: If page_brute tells you that the rule did not compile, carefully check the syntax of the rule in paged_exes.yar before continuing. Ask an instructor for help if needed.

2. **Examine results obtained with new YARA rules**
   - Change directory to the output directory, ~/insider-exes. Inside this directory there should be a single directory (because you only had one rule matched). Change directory to paged_exes. Run

the following command to determine the number of files present. How many results were obtained?

```
ls -1| wc -l
```

_____

- Examine one of the output files to ensure that it contains what you expected. Since this page file fragment contains binary data, a hex viewer is preferred over a text editor. Run the following command to examine the file '454286.block'.

```
hexdump -C 454286.block
```

- The file stored in this page fragment had the internal name Classpnp.sys. That doesn't sound too bad. But rootkit.sys sure would. Or maybe a randomly named exe file... Unfortunately, this process seems way too manual to perform by hand. Let's use some shell-fu to analyze this better. Type the following command into the command prompt:

  Note: The switch following the letter e in each strings command is the letter L, not the number 1. The character following the capital A in the grep commands is the number 1.

```
for i in `ls`; do echo; echo $i; strings -e l $i | grep -A1
InternalName; strings -e l $i | grep -A1 OriginalFilename;
done
```

- Note that the **OriginalFilename** and **InternalName** attributes don't need to match, but they should be very similar. Another addition might be to add the strings for **ProductVersion** or **Description**. Most commercial software populates these fields in the VERSION_INFO structure, but malware tends not to.

Optional Homework: If you have additional time, consider adding additional fields to the output of the shell script above. This would require examining the fields available in the hexdump output and adding grep lines to the for loop. This type of information is useful in real world investigations and should be part of your arsenal going forward. After adding fields to output, consider creating a shell script that you can run so you don't have to copy (or remember) the long for loop.

Exercise Part I: Analyze Page File - Step-by-Step

1. **Page File Identification**
   - Open a terminal and change directory to ~/page_brute. The page_brute tool is still in beta. As such, it does not operate so as well from outside of it's own directory. Technically, this would just require you to specify the location of the default YARA rules file, but we're anti-typo and this is a much easier option.
   - Ensure that the sample page file in /cases/exercise6 has been imported and unzipped. The target file is insider-pagefile.sys.
   - Run the Linux file command against insider-pagefile.sys. What is the type of the file?
     - **data**

```
sansforensics@SIFT-Workstation$ file /cases/insider-pagefile.sys
/cases/insider-pagefile.sys: data
```

   - Based on this identification, do you think Linux will be any help in parsing this file directly? Does it appear to have any underlying file structure?
     - **This is unstructured data. Linux will be of no help in parsing the data.**

2. **Page File Examination**
   - Try dumping out a portion of the page file using the hexdump command. Using the hex dump is a good way to examine unstructured data. Run the following command to dump the first 4096 bytes (4k) of the page file. You want to examine the page file 4k at a time since this is the default page size in Windows.
     - `hexdump -C -n 4096 /cases/exercise6/insider-pagefile.sys`
     - Does the hex dump contain any notable data?
     - **No, the data is all zeros.**

```
sansforensics@SIFT-Workstation$ hexdump -C -n 4096 /cases/insider-pagefile.sys
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000
```

   - You can examine additional portions of the page file from the command line 4k at a time by adding the –s switch to skip some number of bytes (the number to skip should be a multiple of 4 bytes).

```
hexdump -C -n 4096 -s 4096 /cases/exercise6/insider-pagefile.sys
```

Exercise Part II: Analysis using page_brute - Step-by-Step

1. **Run page_brute with Default_signatures.yar.**
   - Use page_brute with the following options to parse the insider's paging file.

     ```
     ./page_brute-BETA.py -f /cases/exercise6/insider-pagefile.sys -
     o ~/insider-case
     ```

     The –f option specifies the input file while the –o option specifies the output directory.

```
sansforensics@SIFT-Workstation$ ./page_brute-BETA.py -f /cases/insider-pagefile.sys
 -o ~/insider-case
[+] - PAGE_BRUTE processing file: /cases/insider-pagefile.sys
[+] - Ruleset Compilation Successful.
[+] - PAGE_BRUTE running with the following options:
        [-] - FILE: /cases/insider-pagefile.sys
        [-] - PAGE_SIZE: 4096
        [-] - RULES TYPE: DEFAULT
        [-] - RULE LOCATION: default_signatures.yar
        [-] - INVERSION SCAN: False
        [-] - WORKING DIR: /home/sansforensics/insider-case
        ================

        [!] FLAGGED BLOCK 70488: webartifact_html
        [!] FLAGGED BLOCK 71556: webartifact_html
        [!] FLAGGED BLOCK 72452: webartifact_javascript
        [!] FLAGGED BLOCK 72455: webartifact_javascript
```

     NOTE: This command can take considerable time to complete. If you are working on this live in class, you'll want to use the pre-cooked data provided in /cases/insider-pageBrute-precooked.zip. Change directory back to your home directory and unzip this file to analyze the output data. You can do this with the following two commands:

```
sansforensics@SIFT-Workstation$ cd ~
sansforensics@SIFT-Workstation$ unzip /cases/insider-pageBrute-precooked.zip
Archive:  /cases/insider-pageBrute-precooked.zip
   creating: insider-case/
   creating: insider-case/webartifact_javascript/
  inflating: insider-case/webartifact_javascript/72457.block
  inflating: insider-case/webartifact_javascript/274284.block
```

     ```
     cd ~
     unzip /cases/exercise6/insider-pageBrute-precooked.zip
     ```

   - One annoyance with page_brute is that is creates all directories with 777 permissions. This causes the Ubuntu terminal to display the directory names in an odd fashion. Use the following command to fix that. In the example command below, you are running this from the directory containing the insider-case directory (your home directory in this case). Every directory found in that directory will now be 755 permissions (yielding better security and a normal display).

```
find insider-case -type d -exec chmod 755 {} \;
```

2. **Review Output/Precooked Data**

- Change directory into ~/insider-case/. What types of data did the default parsers for page_brute discover?

  webartifact_html
  webartifact_javascript

- Change directory to the ~/insider-case/webartifact_javascript directory and run the ll command. What size are the files?

  4096 bytes

```
sansforensics@SIFT-Workstation$ cd ~/insider-case/webartifact_javascript
sansforensics@SIFT-Workstation$ ll
total 60
1073893 -rw-rw-rw- 1 sansforensics sansforensics 4096 2014-01-11 19:44 168494.block
1073900 -rw-rw-rw- 1 sansforensics sansforensics 4096 2014-01-11 19:44 168497.block
1073899 -rw-rw-rw- 1 sansforensics sansforensics 4096 2014-01-11 19:44 168498.block
1073901 -rw-rw-rw- 1 sansforensics sansforensics 4096 2014-01-11 19:44 168499.block
1073897 -rw-rw-rw- 1 sansforensics sansforensics 4096 2014-01-11 19:44 168500.block
1073898 -rw-rw-rw- 1 sansforensics sansforensics 4096 2014-01-11 19:45 274281.block
1073891 -rw-rw-rw- 1 sansforensics sansforensics 4096 2014-01-11 19:45 274284.block
```

- Examine the file 274286.block. You could open it in the Bless hex editor or simply use the hexdump command to dump the file. How would you characterize the file contents?

```
hexdump -C 274286.block
```

  This appears to be part of an antivirus definition file. This appears to be the case because of the number of small script elements and the names of various viruses and malicious software in the file as well.

```
sansforensics@SIFT-Workstation$ hexdump -C 274286.block
00000000  10 03 00 10 6e 8c d7 c5  29 79 00 00 00 00 00 00  |....n...)y......|
00000010  00 00 00 00 00 00 00 00  34 02 00 00 00 00 00 00  |........4.......|
00000020  3c 73 63 72 69 70 74 73  72 63 3d 22 68 74 74 70  |<scriptsrc="http|
00000030  3a 2f 2f 25 37 31 25 37  61 25 36 38 25 32 65 25  |://%71%7a%68%2e%|
00000040  37 34 25 37 38 25 37 33  25 36 38 25 36 39 25 32  |74%78%73%68%69%2|
00000050  65 25 36 33 25 36 66 25  36 64 2f 25 36 32 25 33  |e%63%6f%6d/%62%3|
00000060  32 25 32 65 25 36 31 25  37 33 25 37 30 90 00 21  |2%2e%61%73%70..!|
00000070  23 53 43 50 54 3a 45 78  70 6c 6f 69 74 3a 57 69  |#SCPT:Exploit:Wi|
00000080  6e 33 32 2f 50 64 66 66  69 72 2e 41 5f 4e 65 77  |n32/Pdffir.A_New|
```

- Could exploited web pages or other malicious file types be located in the page file?

Yes, exploits or other malicious file artifacts could be located in the page file. This is a great use for unstructured analysis of the page file.

- Change directory to ~/insider-case/webartifact_html and run the ll command. Note that the files are all again 4096 bytes.

- Examine the data in the file 70488.block using the hexdump command. How would you characterize this data? How might this be useful in an investigation?

```
hexdump -C 70488.block
```

This data can best be characterized as the manifest for .Net assembly. This means that there is likely some executable code or resource data stored in the remainder of the page file fragment.

```
sansforensics@SIFT-Workstation$ hexdump -C 70488.block
00000000  0d 60 0c 50 0b 30 00 00  11 11 06 00 11 74 11 00  |.`.P.0.......t..|
00000010  11 64 10 00 11 d2 0d c0  4c 42 00 00 01 00 00 00  |.d......LB......|
00000020  ec 1f 00 00 3f 20 00 00  dc 47 00 00 00 00 00 00  |....? ...G......|
00000030  19 1b 03 00 09 01 5e 00  02 30 00 00 40 44 00 00  |......^..0..@D..|
00000040  01 00 00 00 8e 22 00 00  c7 22 00 00 c0 47 00 00  |....."..."...G..|
00000050  00 00 00 00 e2 02 00 00  11 15 08 00 15 74 0b 00  |.............t..|
00000060  15 64 0a 00 15 34 08 00  15 52 11 c0 4c 42 00 00  |.d...4...R..LB..|
00000070  01 00 00 00 f3 20 00 00  04 21 00 00 20 48 00 00  |..... ...!.. H..|
```

```
00000e30  20 20 3c 64 65 70 65 6e  64 65 6e 74 41 73 73 65  |  <dependentAsse|
00000e40  6d 62 6c 79 3e 0d 0a 20  20 20 20 20 20 20 3c  |mbly>..        <|
00000e50  61 73 73 65 6d 62 6c 79  49 64 65 6e 74 69 74 79  |assemblyIdentity|
00000e60  0d 0a 20 20 20 20 20 20  20 20 20 20 20 20 74 79  |..            ty|
00000e70  70 65 3d 22 77 69 6e 33  32 22 0d 0a 20 20 20 20  |pe="win32"..    |
00000e80  20 20 20 20 20 20 20 20  6e 61 6d 65 3d 22 4d 69  |        name="Mi|
00000e90  63 72 6f 73 6f 66 74 2e  57 69 6e 64 6f 77 73 2e  |crosoft.Windows.|
00000ea0  43 6f 6d 6d 6f 6e 2d 43  6f 6e 74 72 6f 6c 73 22  |Common-Controls"|
00000eb0  0d 0a 20 20 20 20 20 20  20 20 20 20 20 20 76 65  |..            ve|
00000ec0  72 73 69 6f 6e 3d 22 36  2e 30 2e 30 2e 30 22 0d  |rsion="6.0.0.0".|
00000ed0  0a 20 20 20 20 20 20 20  20 20 20 20 20 70 72 6f  |.           pro|
00000ee0  63 65 73 73 6f 72 41 72  63 68 69 74 65 63 74 75  |cessorArchitectu|
00000ef0  72 65 3d 22 61 6d 64 36  34 22 0d 0a 20 20 20 20  |re="amd64"..    |
00000f00  20 20 20 20 20 20 20 20  70 75 62 6c 69 63 4b 65  |        publicKe|
00000f10  79 54 6f 6b 65 6e 3d 22  36 35 39 35 62 36 34 31  |yToken="6595b641|
00000f20  34 34 63 63 66 31 64 66  22 0d 0a 20 20 20 20 20  |44ccf1df"..     |
```

Exercise Part III: Examine and Create YARA rules - Step-by-Step

- In this section, we'll learn more about the structure of YARA rules by dissecting the rules written by another author. YARA is a complex and powerful language. This lab will only scratch the surface of its capabilities. You are highly encouraged to continue exploring YARA on your own outside of class.

- Change directory back to the ~/page_brute directory. Examine the YARA rules defined by default for page_brute using the grep command (shown below). The default rules are stored in default_signatures.yar. This should give you some idea of the types of artifacts that page_brute can detect.

```
grep rule default_signatures.yar
```

```
sansforensics@SIFT-Workstation$ grep rule default_signatures.yar
rule administrative_share_abuse
rule remote_system_syntax
rule http_request_header
rule http_response_header
rule webartifact_html
rule webartifact_javascript
rule cmdshell
rule webartifact_gmail
rule social_security_syntax
rule smtp_fragments
rule irc
rule ftp
```

- Examine the structure of a rule by opening the file in scite. In this exercise, we'll only consider the first rule, administrative_share_abuse.

```
scite default_signatures.yar &
```

```
rule administrative_share_abuse
{
    meta:
        author="@matonis"
        description="syntax for accessing adminstrative shares"
    strings:
        $s0 = /(copy|del|psexec|net)/ nocase
        $s1 = "\\c$\\windows\\system32\\" nocase
        $s2 = "\\c$\\system32\\" nocase
        $s3 = "\\admin$\\" nocase
    condition:
        $s0 and (any of ($s1,$s2,$s3))
}
```

- The meta portion of the rule should be fairly self-explanatory. This section is completely optional and contains fields for the rule author as well as a short description of the rule.

- Examine the $s0 portion of the rule. Which commands does the rule author look for?
  Note: the nocase modifier is used to indicate that the string should be parsed case insensitive.
  Note: the | symbol is a symbolic OR. It allows many strings to be expressed in a single variable.

  The author looks for copy, del, psexec, and net.

- Examine strings $s1, $s2, and $s3. Which directories does the author look for the commands identified above to be run in? Note: the double backslash is required because the backslash is an escape sequence in the YARA rule language.

  The author looks for commands to be run in these directories:
- **\c$\windows\system32**
- **\c$\system32**
- **\admin$**

- Finally, examine the condition statement. Under which conditions will this rule be matched?

  The rule will be matched when any of the commands specified in $s0 are found in the same page fragment as any of the directories specified in S1, S2, or $s3.

3. **Building a new YARA rule.**
   - In this case, you'll build a YARA rule to detect executable files in the page file. This is sometimes indicative of a downloaded or copied executable. When executables are run from a local drive, they are not paged to disk (since they are already on the disk). Interesting indeed!

   - We could signature on something interesting like the string "This program cannot be run in DOS mode". However, this string will not provide additional information about the file, such as the filename. Getting results like that can be interesting, but sometimes feels like a tease. Instead, we'll focus on two pieces of data that should be part of the file VERSION_INFO. The two strings you want to match are 'FileVersion' and 'InternalName'. Both of these strings are Unicode, so we'll have to use the 'wide' modifier to the YARA rule.

   - In the page_brute directory, use scite to create a new rule file called paged_exes.yar using the following command:

     - `scite paged_exes.yar &`

   - In the new file, enter this text to create the shell for the new rule:

```
rule paged_exes {
        meta:
                author="FOR526 student"
                description="find paged exes using data from the VERSION_INFO structure"
        strings:
                $s0 = ""
                $s1 = ""
        condition:
                # Enter your own condition here
}
```

- Populate $s0 and $s1 with the appropriate strings to search for. Then, write a condition string that can be used to detect the VERSION_INFO structure in executable files paged to disk. Since the strings are stored as Unicode, don't forget to apply the 'wide' modified after each string. If you are finding this too difficult, check out the solutions for the lab to get the exact syntax.

```
rule paged_exes {
        meta:
                author="FOR526 student"
                description="find paged exes using data from the VERSION_INFO structure"
        strings:
                $s0 = "FileVersion" wide
                $s1 = "InternalName" wide
        condition:
                $s0 and $s1
}
```

- When you are done, save the file and exit scite.

Exercise Part iV: Run page_brute with the new rule - Step-by-Step

1. In the page_brute directory, run the following command to parse the page file, this time looking for paged executable files:

```
./page_brute-BETA.py -r paged_exes.yar -o ~/insider-exes    \
-f /cases/exercise6/insider-pagefile.sys
```

Note: If page_brute tells you that the rule did not compile, carefully check the syntax of the rule in paged_exes.yar before continuing. Ask an instructor for help if needed.

```
sansforensics@SIFT-Workstation$ ./page_brute-BETA.py -r paged_exes.yar
   -o ~/insider-exes   -f /cases/insider-pagefile.sys
[+] - PAGE_BRUTE processing file: /cases/insider-pagefile.sys
[+] - YARA rule of File type provided for compilation: paged_exes.yar
..... Ruleset Compilation Successful.
[+] - PAGE_BRUTE running with the following options:
        [-] - FILE: /cases/insider-pagefile.sys
        [-] - PAGE_SIZE: 4096
        [-] - RULES TYPE: FILE
        [-] - RULE LOCATION: paged_exes.yar
        [-] - INVERSION SCAN: False
        [-] - WORKING DIR: /home/sansforensics/insider-exes
        ================

        [!] FLAGGED BLOCK 61747: paged_exes
        [!] FLAGGED BLOCK 61851: paged_exes
        [!] FLAGGED BLOCK 61879: paged_exes
        [!] FLAGGED BLOCK 61890: paged_exes
```

2. **Examine results obtained with new YARA rules**

- Change directory to the output directory, ~/insider-exes. Inside this directory there should be a single directory (because you only had one rule matched). Change directory to paged_exes. Run the following command to determine the number of files present. How many results were obtained?

```
ls -1| wc -l
```

628 pages were discovered containing the pattern specified in the YARA rule.

```
sansforensics@SIFT-Workstation$ ls -1|wc -l
628
```

- Examine one of the output files to ensure that it contains what you expected. Since this page file fragment contains binary data, a hex viewer is preferred over a text editor. Run the following command to examine the file '454286.block'.

```
hexdump -C 454286.block
```

```
sansforensics@SIFT-Workstation$ hexdump -C 454286.block
00000000  77 55 6e 72 65 67 69 73  74 65 72 00 9b 02 49 6f  |wUnregister...Io|
00000010  57 4d 49 57 72 69 74 65  45 76 65 6e 74 00 41 02  |WMIWriteEvent.A.|
00000020  49 6f 52 65 67 69 73 74  65 72 44 65 76 69 63 65  |IoRegisterDevice|
00000030  49 6e 74 65 72 66 61 63  65 00 48 02 49 6f 52 65  |Interface.H.IoRe|
00000040  67 69 73 74 65 72 50 72  69 6f 72 69 74 79 43 61  |gisterPriorityCa|
```

```
00000b10  01 00 49 00 6e 00 74 00  65 00 72 00 6e 00 61 00  |..I.n.t.e.r.n.a.|
00000b20  6c 00 4e 00 61 00 6d 00  65 00 00 00 43 00 6c 00  |l.N.a.m.e...C.l.|
00000b30  61 00 73 00 73 00 70 00  6e 00 70 00 2e 00 73 00  |a.s.s.p.n.p...s.|
00000b40  79 00 73 00 00 00 00 00  80 00 2e 00 01 00 4c 00  |y.s...........L.|
```

- The file stored in this page fragment had the internal name Classpnp.sys. That doesn't sound too bad. But rootkit.sys sure would. Or maybe a randomly named exe file… Unfortunately, this process seems way too manual to perform by hand. Let's use some shell-fu to analyze this better. Type the following command into the command prompt:

  Note: The switch following the letter e in each strings command is the letter L, not the number 1. The character following the capital A in the grep commands is the number 1.

  ```
  for i in `ls`; do echo; echo $i; strings -e l $i | grep -A1
  InternalName; strings -e l $i | grep -A1 OriginalFilename; done
  ```

```
sansforensics@SIFT-Workstation$ for i in `ls`; do echo;echo $i; strings -e l $i |
 grep -A1 InternalName; strings -e l $i |grep -A1 OriginalFilename; done

101058.block
InternalName
FileVersion

114944.block
InternalName
mouclass.sys
OriginalFilename
mouclass.sys

117077.block
InternalName
swenum.sys
OriginalFilename
swenum.sys
```

- Note that the **OriginalFilename** and **InternalName** attributes don't need to match, but they should be very similar. Another addition might be to add the strings for **ProductVersion** or Description. Most commercial software populates these fields in the VERSION_INFO structure, but malware tends not to.

Optional Homework: If you have additional time, consider adding additional fields to the output of the shell script above. This would require examining the fields available in the hexdump output and adding grep lines to the for loop. This type of information is useful in real world investigations and should be part of your arsenal going forward. After adding fields to output, consider creating a shell script that you can run so you don't have to copy (or remember) the long for loop.

Exercise – Key Takeaways

- Since the much of the contents of a system's physical memory can and in many cases, will be, sent to the hard disk's page file for temporary storage, the page file contains a wealth of data that holds investigative value for forensic analysis.
- The page_brute tool contains default YARA rules that are useful for parsing some types of data.
- YARA rules are easy to create and highly customizable.

Exercise 7 – Dumping Memory with volshell

Objectives

- Gain experience enumerating modules using the Volatility framework
- Learn to examine OS structures in memory using volshell
- Learn to use volshell to extract arbitrary contents of a memory image and write them to disk

Exercise Preparation

Start a terminal and change into the **/cases/exercise7** directory. Unzip **evil_driver.zip**.

```
$ cd /cases/exercise7
$ unzip evil_driver.zip
```

Verify that your evil_driver.img file has the appropriate md5sum. We've seen some cases where copying and extracting large files (such as these memory images) can cause errors.

```
# md5sum evil_driver.img
14ac99e2e6980d6a962f0ac2a1cb65c8  evil_driver.img
```

Exercise – Questions

1. **Image Identification**
   - Run the **imageinfo** plugin against the **evil_driver.img** image.
     i. When was the memory image created?

     _____

     ii. What profile should be specified when analyzing this image?

     _____

2. Driver Enumeration

In the first Volatility exercise, we started our analysis by enumerating processes via two different methods- by walking linked lists and scanning for pool allocations marked with the process pool tags. In this exercise, we will focus on enumerating drivers. When drivers are loaded into memory, a LDR_DATA_TABLE_ENTRY structure is created. We can walk the linked list of LDR_DATA_TABLE_ENTRY structures to enumerate drivers using the **modules** plugin.

Using the `modules` plugin on the evil_driver.img, enumerate loaded drivers.

- What is the **full path** of the **irykmmww.sys** binary?

  _____

- What is the **base address** of the irykmmww.sys binary?

  _____

- What is the **size** of the **irykmmww.sys** binary?

  _____

- Identify the **offset** of the irykmmww.sys binary's _LDR_DATA_TABLE_ENTRY structure.

  _____

3. Extract Rogue Driver

- Run the `moddump` plugin to dump the **irykmmww.sys** driver from memory. What is the size of the resulting file?

  _____

- Analyze strings for the use of the **IoCreateSymbolicLink** API. This API is normally used for to create a device file that is accessible from user space. Once the device file can be accessed from user space, easy coordination between user and kernel space code can be accomplished. This API is therefore very often used in rootkit drivers. Note that it is also often used in legitimate drivers as well, so this by itself cannot be used to identify rootkit related drivers. This check will also verify that the file was successfully extracted. What is the offset of the IoCreateSymbolicLink string?

  _____

4. **Use volshell to Extract the Driver**

- Run the **volshell** plugin to enter the volatility shell.
  Note: For this portion of the exercise, you will be provided step by step in this portion of the guide. You are not expected to already know this (that's what the lab is about).

```
vol.py -f evil_driver.img --profile=WinXPSP3x86 volshell
```

- In **volshell**, run the command **hh()** to see help output. This shows some (but not all) functions available in volshell.

- In **volshell**, run the command **modules()** to list the modules. Record both the offset and the base of the **irykmmww.sys** module.

  Offset: _____

  Base: _____

- In **volshell**, run the command **db(0xf836f000)** to display a hex dump of memory at the virtual address listed as the base of irykmmww.sys. This step is a sanity check to verify that the familiar 'MZ' header (0x45da) is present as expected at the base address in memory.

- The 'MZ' header is present as expected. Before we can manually dump the driver to a file, we need to determine how much memory is allocated for it. To do this, you'll need to examine the _LDR_DATA_TABLE_ENTRY structure for the driver. In **volshell**, run the following commands to dump the structure. Note that the address used for this command is the offset field we found earlier in the output of the **modules()** command. Record the memory allocated for the module, as located in the SizeOfImage field.

```
dt('_LDR_DATA_TABLE_ENTRY', 0x81d1c7d0)
```

  Allocated Memory Size : _____

- In **volshell**, run the following commands to instantiate a new object representing the address space of the memory file.

```
addrspace = self._proc.get_process_address_space()
```

- In **volshell**, run the following commands to read the contents of the loaded module into memory and store them in a variable we've named 'data'. Note that the size of the image is 16384. We will use the hex representation of the number as this is probably more familiar when dealing with the size of sections in memory.

```
data = addrspace.read(0xf836f000, 0x4000)
```

- In **volshell**, run the following commands to create a handle to the output file. In this case the output file is '/tmp/f836f000.dmp' and the file mode is set to write (w) and binary (b). On Linux platforms, the 'b' option is superfluous, however excluding it will cause problems on Windows systems.

```
outfile = open('/tmp/f836f000.dmp', 'wb')
```

- In **volshell**, run the following command to write the data read from the virtual memory space to the output file opened in the last step.

```
outfile.write(data)
```

- In **volshell**, run the following command to close the file. Depending on a number of factors, the file will still have a zero length on the file system until the file is closed (or the data is flushed). Since we'll be closing the file immediately anyway,

```
outfile.close()
```

5. **Examine the output file from volshell**
   - Use the `ls -l` command to examine the file created by volshell. What is the file size?

     _____

   - Why is the file size different between the volshell and moddump techniques?

     _____

     _____

6. **Compare the output files**
   - Use the `ssdeep` command to compare the output files. The ssdeep command uses a fuzzy (or piecewise) hashing algorithm to identify how much content multiple files have in common.
   - Change directory to /tmp

     ```
     cd /tmp
     ```

   - Create a signature file from the dump file created by the `moddump` plugin.

     ```
     ssdeep /tmp/driver.f836f000.sys > sigs
     ```

   - Now compare the `volshell` output to the file created with `moddump`.

     ```
     ssdeep -m sigs /tmp/f836f000.dmp
     ```

   - Even though the size is different, are the files similar according to the fuzzy hashing algorithm? What is the matching score? _____

7. **Optional Homework #1:** Use the `volshell` plugin to dump other areas of memory. Using the techniques described above, it is possible to dump arbitrary sections of memory. This is sometimes preferable to using the `moddump` and `dlldump` plugins. That is because these plugins try to validate the executable headers, which may be corrupted. Further these plugins do not offer the ability to dump arbitrary sections of allocated memory, which may not belong to a module (heap memory for example).
   - One potential pitfall with the steps shown above is that volshell's `read` function may fail if the memory being read is paged to disk (or otherwise not in the physical memory image). For this reason, it may be preferable to use the `zread` function in place of `read`. The `zread` function operates identically except that it fills in inaccessible areas with zeroes. The reason this is not the first choice function is that using it does not make it obvious when memory has been paged out.

```
data = addrspace.zread(0xf836f000, 0x4000)
```

Note that if you are looking for modules to dump for the optional homework, try using the dlllist command to find a suspicious DLL with a name very similar to the suspicious driver.

1. **Image Identification**
   - Run the **imageinfo** plugin against the evil_driver.img image
     i. When was the memory image created?

     2009-05-05 19:28:57 UTC

     ii. What profile should be specified when analyzing this image?

     WinXPSP3x86

```
user@SIFT$ vol.py -f evil_driver.img imageinfo
Determining profile based on KDBG search...

          Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with WinXPSP2x86)
                     AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (/cases/evil_driver.img)
                      PAE type : PAE
                           DTB : 0x319000L
                          KDBG : 0x80545b60
          Number of Processors : 1
     Image Type (Service Pack) : 3
                KPCR for CPU 0 : 0xffdff000
             KUSER_SHARED_DATA : 0xffdf0000
           Image date and time : 2009-05-05 19:28:57 UTC+0000
     Image local date and time : 2009-05-05 15:28:57 -0400
```

2. **Driver Enumeration**
   - What is the **full path** of the **irykmmww.sys** binary?

     C:\WINDOWS\system32\drivers\irykmmww.sys

```
user@SIFT$ vol.py -f evil_driver.img --profile=WinXPSP3x86 modules
Offset(V)  Name                 Base             Size File
---------- -------------------- ---------- ---------- ----
0x81e09670 mrxsmb.sys           0xf698e000    0x70000 \SystemRoot\system32\DRIVERS\mrxsmb.sys
0x8219ae70 hidusb.sys           0xf82ba000     0x3000 \SystemRoot\system32\DRIVERS\hidusb.sys
0x81ed1bd0 HIDCLASS.SYS         0xf877a000     0x9000 \SystemRoot\system32\DRIVERS\HIDCLASS.SYS
0x82198038 HIDPARSE.SYS         0xf89fa000     0x7000 \SystemRoot\system32\DRIVERS\HIDPARSE.SYS
0x82197c18 mouhid.sys           0xf8167000     0x3000 \SystemRoot\system32\DRIVERS\mouhid.sys
0x81dbd970 mrxdav.sys           0xf649b000    0x2d000 \SystemRoot\system32\DRIVERS\mrxdav.sys
0x81d1c7d0 irykmmww.sys         0xf836f000     0x4000 \??\C:\WINDOWS\system32\drivers\irykmmww.sys
```

   - What is the **base address** of the irykmmww.sys binary?

     0xf836f000

   - What is the **size** of the irykmmww.sys binary?

0x4000

- Identify the **offset** of the irykmmww.sys binary's _LDR_DATA_TABLE_ENTRY structure.

0x81d1c7d0

3. **Extract Rogue Driver**
   - Run the **moddump** plugin to dump the irykmmww.sys driver from memory. What is the size of the resulting file?

14592

```
# vol.py -f evil_driver.img --profile=WinXPSP3x86 moddump
    -b 0xf836f000 -D /tmp

# ls -l /tmp/driver.f836f000.sys
```

```
user@SIFT$ vol.py -f evil_driver.img --profile=WinXPSP3x86 moddump -b 0xf836f000 -D /tmp
Module Base Module Name          Result
----------- -------------------- ------
0x0f836f000 irykmmww.sys         OK: driver.f836f000.sys
user@SIFT$ ls -l /tmp/driver.f836f000.sys
-rw-r--r-- 1 sansforensics dfircon 14592 Dec 23 10:31 /tmp/driver.f836f000.sys
```

   - Analyze strings for the use of the **IoCreateSymbolicLink** API. This API is normally used to create a device file that is accessible from user space. Once the device file can be accessed from user space, easy coordination between user and kernel space code can be accomplished. This API is therefore very often used in rootkit drivers. Note that it is also often used in legitimate drivers as well, so this by itself cannot be used to identify rootkit related drivers. This check will also verify that the file was successfully extracted. What is the offset of the IoCreateSymbolicLink string?

0x3586

```
# strings -atx /tmp/driver.f836f000.sys

# strings -atx /tmp/driver.f836f000.sys | grep
IoCreateSymbolicLink
```

```
sansforensics@SIFT-Workstation:/cases/bootcamp$ strings -atx /tmp/driver.f836f000.sys
     4d !This program cannot be run in DOS mode.
     b8 Rich
    1c8 .text
    1ef h.data
    218 INIT
```

TRUNCATED OUTPUT

```
   353c IoDeleteSymbolicLink
   3554 IofCompleteRequest
   356a KeServiceDescriptorTable
   3586 IoCreateSymbolicLink
   359e IoCreateDevice
```

4. **Use volshell to Extract the Driver**
   - Run the **volshell** plugin to enter the volatility shell.
     Note: For this portion of the exercise, you will be provided step by step in this portion of the guide. You are not expected to already know this (that's what the lab is about).

     ```
     vol.py -f evil_driver.img --profile=WinXPSP3x86 volshell
     ```

```
user@SIFT$ vol.py -f evil_driver.img --profile=WinXPSP3x86 volshell
Current context: System @ 0x823c8830, pid=4, ppid=0 DTB=0x319000
Python 2.7.3 (default, Feb 27 2014, 19:58:35)
Type "copyright", "credits" or "license" for more information.

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

In [1]:
```

   - In **volshell**, run the command **hh()** to see help output. This shows some (but not all) functions available in volshell.

   - In **volshell**, run the command **modules()** to list the modules. Record both the offset and the base of the **irykmmww.sys** module.

     Offset: 0x81d1c7d0

     Base: 0xf836f000

```
0x81dbd970 0xf649b000 \SystemRoot\system32\DRIVERS\mrxdav.sys
0x81e901c0 0xf8c4e000 \SystemRoot\System32\Drivers\ParVdm.SYS
0x821665b8 0xf8c50000 \??\C:\Program Files\VMware\VMware Tools\Driv
0x8219f328 0xf6359000 \SystemRoot\system32\DRIVERS\srv.sys
0x81dfb8c8 0xf6110000 \SystemRoot\System32\Drivers\HTTP.sys
0x81d1c7d0 0xf836f000 \??\C:\WINDOWS\system32\drivers\irykmmww.sys
0x822e9c88 0xf8a7a000 \??\C:\Program Files\Mandiant\Mandiant Intell
```

- In **volshell**, run the command **db(0xf836f000)** to display a hex dump of memory at the virtual address listed as the base of irykmmww.sys. This step is a sanity check to verify that the familiar 'MZ' header (0x45da) is present as expected at the base address in memory.

```
>>> db(0xf836f000)
0xf836f000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00  MZ..............
0xf836f010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
0xf836f020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xf836f030  00 00 00 00 00 00 00 00 00 00 00 00 d0 00 00 00  ................
0xf836f040  0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68  ........!..L.!Th
0xf836f050  69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f  is.program.canno
0xf836f060  74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20  t.be.run.in.DOS.
0xf836f070  6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00  mode....$.......
```

- The 'MZ' header is present as expected. Before we can manually dump the driver to a file, we need to determine how much memory is allocated for it. To do this, you'll need to examine the _LDR_DATA_TABLE_ENTRY structure for the driver. In **volshell**, run the following commands to dump the structure. Note that the address used for this command is the offset field we found earlier in the output of the **modules()** command. Record the memory allocated for the module, as located in the SizeOfImage field.

```
dt('_LDR_DATA_TABLE_ENTRY', 0x81d1c7d0)
```

Allocated Memory Size : 16384

```
>>> dt('_LDR_DATA_TABLE_ENTRY', 0x81d1c7d0)
[_LDR_DATA_TABLE_ENTRY _LDR_DATA_TABLE_ENTRY] @ 0x81D1C7D0
0x0   : InLoadOrderLinks              2178009040
0x8   : InMemoryOrderLinks            2178009048
0x10  : InInitializationOrderLinks    2178009056
0x18  : DllBase                       4164349952
0x1c  : EntryPoint                    4164358350
0x20  : SizeOfImage                   16384
0x24  : FullDllName                   \??\C:\WINDOWS\system32\drivers\irykmmww.sys
0x2c  : BaseDllName                   irykmmww.sys
0x34  : Flags                         152059904
0x38  : LoadCount                     1
0x3a  : TlsIndex                      73
0x3c  : HashLinks                     2178009100
0x3c  : SectionPointer                4294967295
0x40  : CheckSum                      18033
0x44  : LoadedImports                 4294967294
0x44  : TimeDateStamp                 -
0x48  : EntryPointActivationContext   0
0x4c  : PatchInformation              7471209
```

- In **volshell**, run the following commands to instantiate a new object representing the address space of the memory file.

```
addrspace = self._proc.get_process_address_space()
```

- In **volshell**, run the following commands to read the contents of the loaded module into memory and store them in a variable we've named 'data'. Note that the size of the image is 16384. We will use the hex representation of the number as this is probably more familiar when dealing with the size of sections in memory.

```
data = addrspace.read(0xf836f000, 0x4000)
```

- In `volshell`, run the following commands to create a handle to the output file. In this case the output file is '/tmp/f836f000.dmp' and the file mode is set to write (w) and binary (b). On Linux platforms, the 'b' option is superfluous, however excluding it will cause problems on Windows systems.

```
outfile = open('/tmp/f836f000.dmp', 'wb')
```

- In `volshell`, run the following command to write the data read from the virtual memory space to the output file opened in the last step.

```
outfile.write(data)
```

- In `volshell`, run the following command to close the file. Depending on a number of factors, the file will still have a zero length on the file system until the file is closed (or the data is flushed). Since we'll be closing the file immediately anyway,

```
outfile.close()
```

5. **Examine the output file from volshell**
   - Use the `ls -l` command to examine the file created by `volshell`. What is the file size? **16384**

```
user@SIFT$ ls -l /tmp/f836f000.dmp
-rw-r--r-- 1 sansforensics dfircon 16384 Dec 23 10:43 /tmp/f836f000.dmp
```

   - Why is the file size different between the `volshell` and `moddump` techniques?

     Because the moddump command parses the PE header, it recreates the file as it was found on disk. The windows loader allocates memory sections on page boundaries, while the compiler strives to create a file as small as possible. This causes the "in memory" representation of an executable to almost always be larger than the on disk version.

6. **Compare the output files**
   - Use the `ssdeep` command to compare the output files. The ssdeep command uses a fuzzy (or piecewise) hashing algorithm to identify how much content multiple files have in common.
   - Change directory to /tmp

     ```
     cd /tmp
     ```

   - Create a signature file from the dump file created by the `moddump` plugin.

     ```
     ssdeep /tmp/driver.f836f000.sys > sigs
     ```

   - Now compare the `volshell` output to the file created with `moddump`.

     ```
     ssdeep -m sigs /tmp/f836f000.dmp
     ```

   - Even though the size is different, are the files similar according to the fuzzy hashing algorithm? What is the matching score? **100**

Optional Homework #1:

Use the `volshell` plugin to dump other areas of memory. Using the techniques described above, it is possible to dump arbitrary sections of memory. This is sometimes preferable to using the `moddump` and `dlldump` plugins. That is because these plugins try to validate the executable headers, which may be corrupted. Further these plugins do not offer the ability to dump arbitrary sections of allocated memory, which may not belong to a module (heap memory for example).

- One potential pitfall with the steps shown above is that volshell's `read` function may fail if the memory being read is paged to disk (or otherwise not in the physical memory image). For this reason, it may be preferable to use the `zread` function in place of `read`. The `zread` function operates identically except that it fills in inaccessible areas with zeroes. The reason this is not

the first choice function is that using it does not make it obvious when memory has been paged out.

```
data = addrspace.zread(0xf836f000, 0x4000)
```

Note that if you are looking for modules to dump for the optional homework, try using the dlllist command to find a suspicious DLL with a name very similar to the suspicious driver.

Exercise – Key Takeaways

- Using volshell isn't scary at all.
  - It can be used to examine and extract values from memory structures at a granular level.
  - It can be used to dump arbitrary sections of memory – a useful skill when memory parsing tools fail or return errors.
- Volshell can be used to dump memory sections with corrupt PE headers or sections that never had PE headers. Corrupt PE headers can foil other plugins in like procexedump and procmemdump, so having a backup technique can save the day when your standards fail.

Exercise 8 – Find All the Malware!

Objectives

- Conduct a forensic memory analysis of a compromised system utilizing the Rekall framework
- Recognize the similarities between Rekall and Volatility™
- Identify malicious DLLs and injected processes in a Windows memory image
- Complete malicious DLL extraction in order to conduct further binary analysis
- Recover evidence of batch file execution and the file itself from a memory image

Exercise Preparation

1. Start a terminal and change into the **`/cases/exercise8`** directory. Unzip the archived memory image **`fariet1.zip`**.

```
$ cd /cases/exercise8
$ unzip fariet1.zip
```

Exercise – Questions

You receive report of a machine functioning strangely. The admin believes that it is infected with malware. The admin obtained a memory image as part of the newly implemented triage procedures. You have been asked to identify what, if anything, is on the machine. The helpdesk unfortunately reloaded the machine (contrary to SOP) so this memory capture is your only hope for analyzing the malware that targeted your organization.

1. Image Identification

- Run the `imageinfo` plugin against the **fariet1.vmem**.
  a. When was the memory image created?

  _____

  b. What profile should be specified when analyzing this image?

  _____

2. Rogue Process Analysis

- Run the `pslist` & `pstree` commands and check to see if there are any suspiciously named processes.

  a. Are there any system processes that are spawning user processes?

  _____

  b. What other processes are running on the system that are commonly used by attackers to execute tools?

  _____

3. Analyze Suspicious Process Objects

a. List the DLLs for the iexplore process PID 3340.

  i. What is the command line with which this process was run?

  _____

  ii. List any suspicious DLLs shown based on file path analysis.

  _____

b. What DLL and export are being executed by the rundll32.exe process with a PID 3276?

DLL: _____

EXPORT: _____

c. What DLL and export are being executed by the rundll32.exe process with a PID 3416?

DLL: _____

EXPORT: _____

4. **Extract suspicious DLL for further analysis**

   a. Extract the suspicious DLL seen in PID 3416 `dlllist` output: "colcs.DLL". What are some notable API function calls, identified by using **strings**?

   _____

   b. Upload this PE file to Virustotal, a third-party antivirus scanning service, for further analysis. How many AV software vendors identify this DLL?

   _____

5. **Analyze CMD.exe Process for Attacker Activity**

   a. With what command line was the cmd.exe process with PID 2088 started?

   _____

   b. Based on the command line alone, what is the probable purpose of the cmd.exe command with process ID 2088? Is there anything odd about the cmd.exe arguments?

   _____

   c. Determine whether the cmd.exe process has a handle to the batch script it is executing. Was this expected or not? Why or why not?

   _____

   d. It is still possible that the batch script is resident in a memory section of the cmd.exe process. Follow the steps below to view its contents. Record your findings.

      - Dump the cmd.exe process to a local directory using the `memdump` plugin.
      - Use the strings command to extract the strings (batch files are strings after all) from the process' memory space.
      - Finally, search through the strings for the contents of the batch file using **grep** for the del command or **less** to view output page by page.

   _____

6. **Extra Credit (if you have time): Repeat the exercise using Volatility™ and compare the results.**

1. **Image Identification**
   - Run the `imageinfo` plugin against the **fariet1.vmem**.
     a. When was the memory image created?

        06/21/2013 20:54:43 UTC

     b. What profile should be specified when analyzing this image?

        Win7SP0x86

```
$ rekall -f /cases/fariet1.vmem imageinfo
```

```
user@SIFT$ rekall -f fariet1.vmem imageinfo
        Fact                          Value
-------------------- ------------------------------
Kernel DTB           0x185000
NT Build             7600.win7_rtm.090713-1255
NT Build Ex          7600.16385.x86fre.win7_rtm.090
                     713-1255
Signed Drivers       True
Time (UTC)           2013-06-21 20:54:43+0000
Time (Local)         2013-06-22 00:54:43+0000
Sec Since Boot       6278.515625
NtSystemRoot         C:\Windows

**************** Physical Layout ****************
Phys Start  Phys End  Number of Pages
---------- ---------- ---------------
0x00001000 0x0009f000 158
0x00100000 0x3fee0000 261600
0x3ff00000 0x40000000 256
```

2. **Analysis for Rogue Processes**

   - Run the `pslist` & `pstree` commands and check to see if there are any suspiciously named processes.

     a. Are there any system processes that are spawning user processes?

Most notably, we see 2 **iexplore.exe** processes being spawned from svchost.exe, a system process. Though this is seen in some normal instances, it almost always warrants further investigation when a system process spawns a user process.

```
user@SIFT$ rekall -f fariet1.vmem pstree
_EPROCESS                                  PPid   Thds   Hnds            Time
------------------------------------------ ------ ------ ------ ----------------------
  0x86847530 csrss.exe (360)                352      9    489 2013-06-20 19:01:14+0000
  0x86853530 wininit.exe (400)              352      3     78 2013-06-20 19:01:15+0000
. 0x868fa030 services.exe (504)             400      7    200 2013-06-20 19:01:16+0000
.. 0x86ccf8e0 svchost.exe (620)             504     11    356 2013-06-20 19:01:30+0000
... 0x85619030 iexplore.exe (3336)          620     19    418 2013-06-21 20:54:01+0000
.... 0x853d8850 iexplore.exe (3244)        3336     11    219 2013-06-21 20:54:04+0000
... 0x856184e8 iexplore.exe (3340)          620     21    460 2013-06-21 20:54:12+0000
.... 0x8561e9a8 iexplore.exe (1892)        3340     18    352 2013-06-21 20:54:15+0000
... 0x85611d40 WmiPrvSE.exe (4008)          620      8    122 2013-06-21 20:52:50+0000
```

The **dllhost.exe** and **rundll32.exe** processes may or may not be suspicious, but should always be examined. Additionally, the **taskhost.exe** command may be of interest.

b. What other processes are running on the system that are commonly used by attackers to execute tools?

```
0x85638d40 cmd.exe                2456   2188      1       22      1 False  2013-06-21 20:53:13+0000 -
0x8537a9b8 conhost.exe            2528    408      2       31      1 False  2013-06-21 20:53:06+0000 -
0x85302030 VMwareTray.exe         2560   2420      4       70      1 False  2013-06-20 19:37:07+0000 -
0x85302770 vmtoolsd.exe           2572   2420      6      175      1 False  2013-06-20 19:37:07+0000 -
0x85647158 SearchProtocol         2940   1832      7      316      0 False  2013-06-21 20:54:03+0000 -
0x853d8850 iexplore.exe           3244   3336     11      219      1 False  2013-06-21 20:54:04+0000 -
0x85212030 rundll32.exe           3276   2960      5       89      1 False  2013-06-21 20:53:59+0000 -
0x85619030 iexplore.exe           3336    620     19      418      1 False  2013-06-21 20:54:01+0000 -
0x856184e8 iexplore.exe           3340    620     21      460      1 False  2013-06-21 20:54:12+0000 -
0x856203e0 rundll32.exe           3416   2960      6      110      1 False  2013-06-21 20:54:05+0000 -
0x85304030 audiodg.exe            3668    732      4      116      0 False  2013-06-21 20:51:59+0000 -
0x8567fa08 SearchProtocol         3688   1832      4      206      1 False  2013-06-21 20:54:37+0000 -
0x8686db20 conhost.exe            3932    408      2       34      1 False  2013-06-21 20:52:36+0000 -
0x85196778 cmd.exe                3976   2420      1       22      1 False  2013-06-21 20:52:36+0000 -
```

3. Analyze Suspicious Processes for Illegitimate Dlls

a. List the DLLs for the iexplore.exe process with **PID 3340**.

i. What is the command line with which this process was run?

```
$ rekall -f fariet1.vmem dlllist --pid=3340
```

```
user@SIFT$ rekall -f fariet1.vmem dlllist --pid=3340
************************************************
iexplore.exe pid: 3340
Command line : "C:\Program Files\Internet Explorer\iexplore.exe" -Embedding

   Base       Size          Load Reason/Count        Path
---------- ---------- ------------------------------ ----
0x013d0000    0xa6000 65535                          C:\Program Files\Internet Explorer\iexplore.exe
0x76e10000   0x13c000 65535                          C:\Windows\SYSTEM32\ntdll.dll
0x76a40000    0xd4000 65535                          C:\Windows\system32\kernel32.dll
0x75100000    0x4a000 65535                          C:\Windows\system32\KERNELBASE.dll
0x75a50000    0xa0000 65535                          C:\Windows\system32\ADVAPI32.dll
```

ii. List any suspicious DLLs shown based on file path analysis.

```
0x73000000    0x38000 1                              C:\Windows\System32\fwpuclnt.dll
0x72860000    0x2d000 2                              C:\Windows\system32\IEUI.dll
0x74120000     0x5000 2                              C:\Windows\system32\MSIMG32.dll
0x70f10000    0x5a000 1                              C:\Windows\System32\netprofm.dll
0x10000000    0x5c000 1                              C:\Users\user\AppData\Roaming\colcs.dll
0x728a0000    0x2b000 1                              C:\Program Files\Internet Explorer\ieproxy.dll
0x706c0000     0x8000 1                              C:\Windows\System32\npmproxy.dll
0x734a0000    0x32000 1                              C:\Windows\system32\winmm.dll
0x73c40000    0x40000 3                              C:\Windows\system32\UxTheme.dll
```

b. What DLL and export are being executed by the rundll32.exe process with **PID 3276**?

DLL: C:\Users\user\AppData\Roaming\tsxfas.DLL

EXPORT: DelItemString

```
rekall -f fariet1.vmem dlllist --pid=3276
```

```
user@SIFT$ rekall -f fariet1.vmem dlllist --pid=3276
************************************************
rundll32.exe pid: 3276
Command line : rundll32.exe "C:\Users\user\AppData\Roaming\tsxfas.dll",DelItemString

   Base       Size          Load Reason/Count        Path
---------- ---------- ------------------------------ ----
0x00ea0000     0xe000 65535                          C:\Windows\system32\rundll32.exe
0x76e10000   0x13c000 65535                          C:\Windows\SYSTEM32\ntdll.dll
0x76a40000    0xd4000 65535                          C:\Windows\system32\kernel32.dll
0x75100000    0x4a000 65535                          C:\Windows\system32\KERNELBASE.dll
```

c. What DLL and export are being executed by the rundll32.exe process with **PID 3416**?

DLL: C:\Users\user\AppData\Roaming\colcs.DLL

EXPORT: get_user_height_max

```
rekall -f fariet1.vmem dlllist --pid=3416
```

```
user@SIFT$ rekall -f fariet1.vmem dlllist --pid=3416
*************************************************
rundll32.exe pid: 3416
Command line : rundll32.exe "C:\Users\user\AppData\Roaming\colcs.dll",get_user_height_max


   Base       Size          Load Reason/Count          Path
---------- ---------- ------------------------------ ----
0x00ea0000     0xe000 65535                          C:\Windows\system32\rundll32.exe
0x76e10000   0x13c000 65535                          C:\Windows\SYSTEM32\ntdll.dll
0x76a40000    0xd4000 65535                          C:\Windows\system32\kernel32.dll
```

4. Extract suspicious DLL for further analysis

a. Extract the suspicious DLL seen in PID 3416 `dlllist` output: "colcs.DLL". What are some notable API function calls, identified by using **strings**?

```
$ rekall -f fariet1.vmem dlldump --pid=3416 --regex=colcs
--dump_dir=/cases
```

```
user@SIFT$ rekall -f fariet1.vmem dlldump --pid=3416 --regex=colcs --dump_dir=/cases
_EPROCESS        Name          Base          Module          Dump File
---------- --------------- ---------- ------------------- ---------
0x856203e0 rundll32.exe  3416 rundll32.exe     0x10000000 colcs.dll           module.3416.3f6203e0.10000000.colcs.dll
user@SIFT$ ls -l /cases/module.3416.3f6203e0.10000000.colcs.dll
-rw-r--r-- 1 sansforensics dfircon 376832 Dec 23 12:07 /cases/module.3416.3f6203e0.10000000.colcs.dll
```

```
$ strings /cases/module.3416.3f6203e0.10000000.colcs.dll

$ strings -e l /cases/module.3416.3f6203e0.10000000.colcs.dll
```

b. Upload this PE file to Virustotal for further analysis. What is identified as by the majority of AV software vendors?

5. Analyze CMD.exe Process for Attacker Activity

a. With what command line was the cmd.exe process with PID 2088 started?

```
$ rekall -f fariet1.vmem dlllist --pid=2088
```

```
user@SIFT$ rekall -f fariet1.vmem dlllist --pid=2088
*************************************************
cmd.exe pid: 2088
Command line : cmd /c ""C:\Users\user\AppData\Local\Temp\6181015.bat"       "C:\Windows\System32\DllCaches\svchost.exe"   "


   Base       Size          Load Reason/Count        Path
---------- ---------- ------------------------------ ----
0x4a9d0000    0x4c000 65535                          C:\Windows\system32\cmd.exe
0x76e10000   0x13c000 65535                          C:\Windows\SYSTEM32\ntdll.dll
```

b. Based on the command line alone, what is the probable purpose of the cmd.exe command with process ID 2088? Is there anything odd about the cmd.exe arguments?

Using batch files for the deletion of malicious executable files is very common by sophisticated adversaries. This is apparent in this instance because the file targeted for deletion, an executable, is the argument to the batch file being executed. Note that the attacker has placed the svchost.exe malware in the C:\Windows\System32\DllCaches directory. This is probably meant to blend in with the legitimate dllcache directory.

c. Determine whether the cmd.exe process has a handle to the batch script it is executing. Was this expected or not? Why or why not?

```
$ rekall -f fariet1.vmem handles --pid=2088 -t File
```

No, there is no longer a handle in the cmd.exe process to the batch file.

A batch file is convenient for deleting an executable piece of malware because batch files can be self-cleaning. Executables in memory are backed by their on-disk counterparts, sort of like a mini pagefile for each executable image loaded. However, a batch file is not considered an executable module. A batch file's contents are read by the command interpreter and the handle to the file is closed.

d. It is still possible that the batch script is resident in a memory section of the cmd.exe process. Follow the steps below to view its contents. Record your findings.

- Dump the cmd.exe process to a local directory using the `memdump` plugin.

```
$ mkdir /cases/cmd_dump
$ rekall -f fariet1.vmem memdump --pid=2088 -D /cases/cmd_dump/
```

- Use the `strings` command to extract the strings (batch files are strings after all) from the process' memory space.

```
$ cd /cases/cmd_dump
$ strings cmd.exe_2088.dmp > strings.txt
$ strings -e l cmd.exe_2088.dmp >> strings.txt
```

- Finally, search through the strings for the contents of the batch file using the search feature in **less** to view output page by page. (Run the command below and press '**/**' to invoke the search feature. Type your keyword '**del**' or '**goto**' and hit enter. Go to the next hit by press '**n**'.

```
$ less strings.txt
```

```
02CV
02CV
:ktk
del %1
if exist %1 goto
ktk
del %0 0 %0 0
```

6. **Extra Credit (if you have time): Repeat the exercise using Volatility™ and compare the results.**

Exercise – Key Takeaways

1. In walking through the analysis of the Fariet memory image, we were able to use the PID relationship analysis to spot rogue iexplore.exe processes spawned by an svchost.exe process.
2. Compare the usage of Rekall and Volatility™ memory forensics platforms.
3. In analyzing this memory image, we examined how rundll32.exe can be used to call DLLs and how that is shown using Rekall and Volatility™.
4. Becoming familiar with such plugins as dlllist, dlldump and handles will speed analysis and allow for further investigation of suspicious DLLs outside of the memory image.

Exercise 9 – Stuxnet Deep Dive

Objectives

- Conduct a forensic memory analysis using the outlined memory analysis steps
- Identify evidence of dll injection in use by Stuxnet by utilizing the Volatility framework
- Upload identified rogue dlls for further analysis by a third-party AV software scanner.

Exercise Preparation

```
$ cd /cases
$ ls -al
```

1. Start a terminal and change into the **`/cases`** directory.
2. Ensure that the **`stuxnet.vmem`** image is located in this directory by performing a directory listing.

Exercise - Questions

At the time of its first discovery in July 2010, Stuxnet was acknowledged as one of the most sophisticated pieces of malware ever created. It is a large, complex piece of code, with many different components of varying capabilities. Some of its most notable features include zero-day exploits and privilege escalations, a Windows rootkit, complex process injection and hooking techniques, network lateral movement mechanisms and a Command and Control interface.

Through this exercise, you will be able to identify the process injection that Stuxnet exhibits through a "Process Hollowing" technique of legitimate instances of lsass.exe processes.

1. Image Identification

- Run the **imageinfo** plugin against the **`stuxnet.vmem`**.
  a. When was the memory image created?

  _____

  b. What profile should be specified when analyzing this image?

  _____

2. Identify Rogue Processes

- Use the **`pslist`** plugin to walk the doubly-linked list of EPROCESS structures.
  a. Identify the two suspicious lsass processes based on *process start time analysis*.

  _____

b. What is the parent process for the rogue LSASS processes? What is the normal parent process of a legitimate instance of LSASS?

_____

c. Note the number of threads and handles identified for each.

_____

3. Enumerate Dlls for Suspicious Processes

- Enumerate the dlls of each suspicious lsass.exe process by walking the linked list.
  a. Compare how many dlls the rogue LSASS processes have to that of the legitimate lsass.
  (Hint: Use wc –l for line count function.)

_____

b. Validate the legitimate number of dlls of the legitimate LSASS process of stuxnet.vmem by comparing it to the LSASS process running in the **`xp-laptop-2005-07-04-1430.vmem`** image.

_____

4. Identify injected Dlls in a Rogue Lsass Process.

a. Run the **`vaddump`** plugin against PID 868, specifying an output directory.
b. By running the **`file`** command against the vaddump output, identify how many PE (portable executable) files were extracted from the VAD of PID 868. (Hint: Use wc –l for line count function.)

_____

c. Identify the number of exe/dlls that are mapped to disk (not injected) in Process 868 through the use of the **`vadinfo`** plugin.

_____

d. Identify the injected dlls that are not mapped to disk through the use of the **`vadinfo`** plugin. Compare the vaddump .dmp files identified by the **`file`** command as PE32 files to output of **`vadinfo`**.

_____

5. Upload Injected Dlls to Virustotal for further analysis.

a. How many antivirus software programs hit on these injected dlls?

_____

b. By evaluating the "Additional Information" provided by VirusTotal for the dll identified as Stuxnet, when was this sample first submitted?

_____

Extra Credit.

Using the Volatility framework's **yarascan**, find the unique pdb path of the mrxnet driver associated with the Stuxnet malware that includes the keyword "myrtus". What is the full path of this pdb file?

_____

Exercise – Questions with Step-by-Step

1. **Image Identification**
   - Run the **imageinfo** plugin against the stuxnet.vmem.
     a. When was the memory image created?

        06/03/2011 04:31:36 UTC

     b. What profile should be specified when analyzing this image?

        WinXPSP3x86

```
$ vol.py -f /cases/stuxnet.vmem  imageinfo
```

```
user@SIFT$ vol.py -f stuxnet.vmem imageinfo
Determining profile based on KDBG search...

          Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86 (Instantiated with WinXPSP2x86)
                     AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (/cases/stuxnet.vmem)
                      PAE type : PAE
                           DTB : 0x319000L
                          KDBG : 0x80545ae0
          Number of Processors : 1
     Image Type (Service Pack) : 3
                KPCR for CPU 0 : 0xffdff000
             KUSER_SHARED_DATA : 0xffdf0000
           Image date and time : 2011-06-03 04:31:36 UTC+0000
     Image local date and time : 2011-06-03 00:31:36 -0400
```

2. **Identify Rogue Processes**
   - Use the **pslist** plugin to walk the doubly-linked list of EPROCESS structures.
     a. Identify the two suspicious **lsass** processes based on process start time analysis.

        PID 868 Start Time 2011-06-03 04:26:55 UTC
        PID 1928 Start Time 2011-06-03 04:26:55 UTC

     b. What is the parent process for the rogue LSASS processes? What is the normal parent process of a legitimate instance of lsass?

        PID 668 Services.exe

        On a normal Windows XP system, the Winlogon process creates lsass when the system boots. On Windows Vista and higher, the Wininit process creates it.

c. Note the number of threads and handles identified for each.

Legitimate LSASS Process
PID 680 Thds 19 Hndls 342

Rogue LSASS Processes
PID 868 Thds 2 Hndls 23
PID 1928 Thds 4 Hndls 65

```
$ vol.py -f /cases/stuxnet.vmem  --profile=WinXPSP3x86 pslist
```

```
0x81fa5390 wmiprvse.exe          1872    856      5      134      0
0x81c498c8 lsass.exe              868    668      2       23      0
0x81c47c00 lsass.exe             1928    668      4       65      0
0x81c0cda0 cmd.exe                968   1664      0 -------      0
0x81f14938 ipconfig.exe           304    968      0 -------      0
```

3. **Enumerate Dlls for Suspicious Processes**
   - Enumerate the dlls of each suspicious **lsass.exe** process by walking the linked list.
     a. Compare how many dlls the rogue LSASS processes have to that of the legitimate lsass.

Legitimate LSASS Process PID 680 – 64 DLLs
Rogue LSASS Process PID 868 – 15 DLLs
Rogue LSASS Process PID 1928 – 35 DLLs

b. Validate the legitimate number of dlls of the legitimate LSASS process of **stuxnet.vmem** by comparing it to the LSASS process running in the **xp-laptop-2005-07-04-1430.vmem** image.

Legitimate Stuxnet Image LSASS Process PID 680 – 64 DLLs
Legitimate xp-laptop-2005-07-04-1430 image LSASS Process PID 536 – 65 DLLs

```
user@SIFT$ vol.py -f stuxnet.vmem --profile=WinXPSP3x86 dlllist -p 680 |wc -l
64
user@SIFT$ vol.py -f stuxnet.vmem --profile=WinXPSP3x86 dlllist -p 868 |wc -l
15
user@SIFT$ vol.py -f stuxnet.vmem --profile=WinXPSP3x86 dlllist -p 1928 |wc -l
35
user@SIFT$ vol.py -f xp-laptop-2005-07-04-1430.vmem pslist |grep lsass
0x821d8248 lsass.exe                536    480     20      369      0      0 2005
user@SIFT$ vol.py -f xp-laptop-2005-07-04-1430.vmem dlllist -p 536 |wc -l
65
```

4. **Identify injected Dlls in a Rogue Lsass Process.**
   a. Run **vaddump** against PID 868, specifying an output directory.
   b. By running the **file** command against the vaddump output, identify how many PE (portable executable) files were extracted from the VAD of PID 868. (Hint: Use wc –l for line count function.)

      Total of 9 PE32 files

```
$ vol.py -f /cases/stuxnet.vmem  --profile=WinXPSP3x86 vaddump -p 868
--dump-dir=/cases/output/lsass_rogue
```

```
sansforensics@SIFT-Workstation:/cases/output/lsass_rogue$ file *.dmp
lsass.exe.1e498c8.0x00010000-0x00010fff.dmp: data
lsass.exe.1e498c8.0x00020000-0x00020fff.dmp: data
lsass.exe.1e498c8.0x00030000-0x0006ffff.dmp: data
lsass.exe.1e498c8.0x00070000-0x00072fff.dmp: data
lsass.exe.1e498c8.0x00080000-0x000f9fff.dmp: PE32 executable for MS Windows (DLL) (GUI) Intel 80386 32-bit
lsass.exe.1e498c8.0x00100000-0x001fffff.dmp: data
lsass.exe.1e498c8.0x00200000-0x0020ffff.dmp: data
lsass.exe.1e498c8.0x00210000-0x0021ffff.dmp: data
lsass.exe.1e498c8.0x00220000-0x00235fff.dmp: data
lsass.exe.1e498c8.0x00240000-0x00280fff.dmp: data
lsass.exe.1e498c8.0x00290000-0x002d0fff.dmp: data
lsass.exe.1e498c8.0x002e0000-0x002e5fff.dmp: DBase 3 index file
lsass.exe.1e498c8.0x002f0000-0x003b7fff.dmp: data
lsass.exe.1e498c8.0x003c0000-0x004c2fff.dmp: data
lsass.exe.1e498c8.0x004d0000-0x004d0fff.dmp: data
lsass.exe.1e498c8.0x004e0000-0x0055ffff.dmp: data
lsass.exe.1e498c8.0x00560000-0x00560fff.dmp: data
lsass.exe.1e498c8.0x00570000-0x0066ffff.dmp: data
lsass.exe.1e498c8.0x01000000-0x01005fff.dmp: PE32 executable for MS Windows (GUI) Intel 80386 32-bit
lsass.exe.1e498c8.0x77dd0000-0x77e6afff.dmp: PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
lsass.exe.1e498c8.0x77e70000-0x77f01fff.dmp: PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
lsass.exe.1e498c8.0x77f10000-0x77f58fff.dmp: PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
lsass.exe.1e498c8.0x77fe0000-0x77ff0fff.dmp: PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
lsass.exe.1e498c8.0x7c800000-0x7c8f5fff.dmp: PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
lsass.exe.1e498c8.0x7c900000-0x7c9aefff.dmp: PE32 executable for MS Windows (DLL) (console) Intel 80386 32-bit
lsass.exe.1e498c8.0x7e410000-0x7e4a0fff.dmp: PE32 executable for MS Windows (DLL) (GUI) Intel 80386 32-bit
lsass.exe.1e498c8.0x7f6f0000-0x7f7effff.dmp: data
lsass.exe.1e498c8.0x7ffb0000-0x7ffd3fff.dmp: data
lsass.exe.1e498c8.0x7ffdc000-0x7ffdcfff.dmp: data
lsass.exe.1e498c8.0x7ffdd000-0x7ffddfff.dmp: DOS executable (COM)
lsass.exe.1e498c8.0x7ffde000-0x7ffdefff.dmp: data
sansforensics@SIFT-Workstation:/cases/output/lsass_rogue$ file *.dmp|grep PE32 |wc -l
9
```

   c. Identify the number of exe/dlls that are mapped to disk (not injected) in Process 868 through the use of the **vadinfo** plugin.

```
$ vol.py -f /cases/stuxnet.vmem  --profile=WinXPSP3x86 -p 868 vadinfo  |
grep -B1 -A9 -i Imagemap | grep -i dll | wc -l
```

```
user@SIFT$ vol.py -f /cases/stuxnet.vmem  --profile=WinXPSP3x86 -p 868 vadinfo
    | grep -B1 -A9 -i Imagemap | grep -i dll | wc -l
7
```

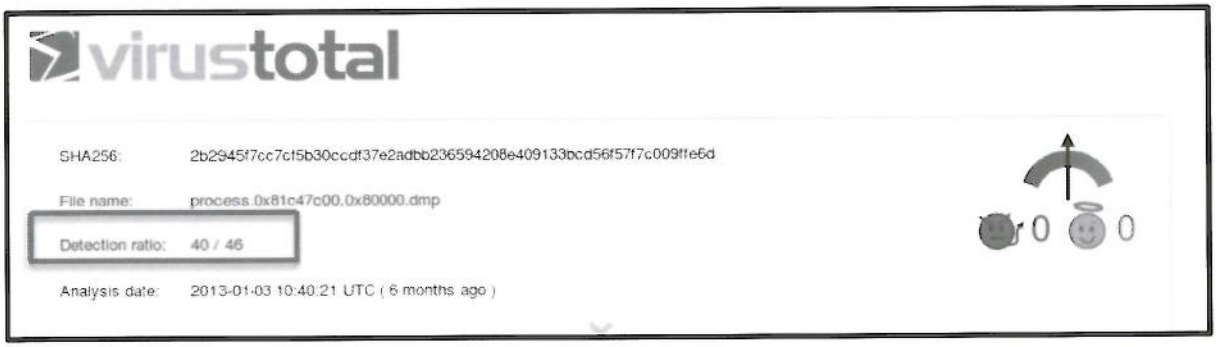
d. Identify the injected dlls that are not mapped to disk through the use of the **vadinfo** plugin. Compare the vaddump .dmp files identified by the **file** command as PE32 files to output of **vadinfo**.

```
$ vol.py -f /cases/stuxnet.vmem  --profile=WinXPSP3x86 vadinfo -p 868
```

```
VAD node @ 0x822e7e70 Start 0x00080000 End 0x000f9fff Tag Vad
Flags: Protection: 6
Protection: PAGE_EXECUTE_READWRITE
ControlArea @81de9890 Segment e2b7dbf0
Dereference list: Flink 00000000, Blink 00000000
NumberOfSectionReferences:          0 NumberOfPfnReferences:          0
NumberOfMappedViews:                1 NumberOfUserReferences:         1
WaitingForDeletion Event:  00000000
Control Flags: Commit: 1, HadUserReference: 1
First prototype PTE: e2b7dc30 Last contiguous PTE: e2b7dff8
Flags2: Inherit: 1
```

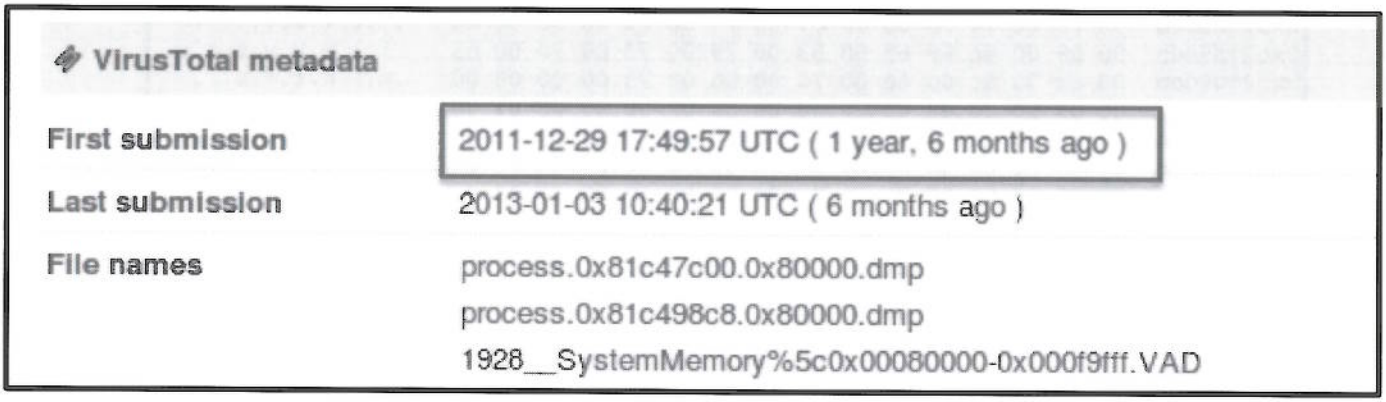
```
VAD node @ 0x81f1ef08 Start 0x01000000 End 0x01005fff Tag Vad
Flags: CommitCharge: 2, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
ControlArea @81fbeee0 Segment e24b4c10
Dereference list: Flink 00000000, Blink 00000000
NumberOfSectionReferences:          1 NumberOfPfnReferences:          0
NumberOfMappedViews:                1 NumberOfUserReferences:         2
WaitingForDeletion Event:  00000000
Control Flags: Commit: 1, HadUserReference: 1
First prototype PTE: e24b4c50 Last contiguous PTE: e24b4c78
Flags2: Inherit: 1
```

5. **Upload the Injected Dlls to Virustotal for further analysis.**
   a. How many antivirus software programs hit on this injected dlls.



b. By evaluating the Additional Information provided by VirusTotal for the dll identified as Stuxnet, when was this sample first submitted?

12/29/2011 17:59:57 UTC



Extra Credit.

Using the Volatility framework's **yarascan**, find the unique pdb path of the mrxnet driver associated with the Stuxnet malware that includes the keyword "myrtus". What is the full path of this pdb file? (Note that **yarascan** will show the 256 bytes from the keyword hit. To view the drive letter of the pdb path, run **yarascan** with the "**-R 256**" option, which will show the 256 bytes prior to the keyword hit.)

```
$ mkdir stuxnet
$ vol.py -f /cases/stuxnet.vmem yarascan -Y "myrtus" -D /cases/stuxnet -K
```

```
sansforensics@siftworkstation:/cases/stuxnet/myrtus$ vol.py -f /cases/stuxnet.vm
em yarascan -Y "myrtus" -D /cases/stuxnet/myrtus -K
Volatility Foundation Volatility Framework 2.4
Rule: r1
Owner: mrxnet.sys
0xb21d9d9b  6d 79 72 74 75 73 5c 73 72 63 5c 6f 62 6a 66 72  myrtus\src\objfr
0xb21d9dab  65 5f 77 32 6b 5f 78 38 36 5c 69 33 38 36 5c 67  e_w2k_x86\i386\g
0xb21d9dbb  75 61 76 61 2e 70 64 62 00 00 00 00 00 00 00 00  uava.pdb........
0xb21d9dcb  00 00 00 00 00 30 18 00 00 1c 1a 00 00 fe ff ff  .....0..........
0xb21d9ddb  ff 00 00 00 00 d8 ff ff ff 00 00 00 00 fe ff ff  ................
0xb21d9deb  ff cb 84 1d b2 cf 84 1d b2 00 00 00 00 fe ff ff  ................
0xb21d9dfb  ff 00 00 00 00 d8 ff ff ff 00 00 00 00 fe ff ff  ................
0xb21d9e0b  ff 21 85 1d b2 25 85 1d b2 00 00 00 00 fe ff ff  .!...%..........
0xb21d9e1b  ff 00 00 00 00 d0 ff ff ff 00 00 00 00 fe ff ff  ................
0xb21d9e2b  ff 9e 86 1d b2 a2 86 1d b2 00 00 00 00 fe ff ff  ................
0xb21d9e3b  ff 00 00 00 00 cc ff ff ff 00 00 00 00 fe ff ff  ................
0xb21d9e4b  ff 0c 87 1d b2 10 87 1d b2 00 00 00 00 fe ff ff  ................
0xb21d9e5b  ff 00 00 00 00 d8 ff ff ff 00 00 00 00 fe ff ff  ................
0xb21d9e6b  ff 95 88 1d b2 99 88 1d b2 00 00 00 00 fe ff ff  ................
0xb21d9e7b  ff 00 00 00 00 d8 ff ff ff 00 00 00 00 fe ff ff  ................
0xb21d9e8b  ff 0f 8c 1d b2 13 8c 1d b2 00 00 00 00 fe ff ff  ................
```

```
$ vol.py -f /cases/stuxnet.vmem yarascan -Y "myrtus" -D /cases/stuxnet -K
-R 256
```

```
sansforensics@siftworkstation:~$ vol.py -f /cases/stuxnet.vmem yarascan -Y "myrt
us" -D /cases/stuxnet -K -s 256 -R 256
Volatility Foundation Volatility Framework 2.4
Rule: r1
Owner: mrxnet.sys
0xb21d9d9b  00 74 00 65 00 6d 00 5c 00 63 00 64 00 66 00 73   .t.e.m.\.c.d.f.s
0xb21d9dab  00 00 00 00 00 5c 00 46 00 69 00 6c 00 65 00 53   .....\.F.i.l.e.S
0xb21d9dbb  00 79 00 73 00 74 00 65 00 6d 00 5c 00 66 00 61   .y.s.t.e.m.\.f.a
0xb21d9dcb  00 73 00 74 00 66 00 61 00 74 00 00 00 5c 00 46   .s.t.f.a.t...\.F
0xb21d9ddb  00 69 00 6c 00 65 00 53 00 79 00 73 00 74 00 65   .i.l.e.S.y.s.t.e
0xb21d9deb  00 6d 00 5c 00 6e 00 74 00 66 00 73 00 00 00 00   .m.\.n.t.f.s....
0xb21d9dfb  00 03 00 00 00 00 00 00 00 00 00 00 00 00 01 00   ................
0xb21d9e0b  00 00 00 00 00 00 00 00 00 00 00 00 00 01 01 01   ................
0xb21d9e1b  00 00 00 00 00 00 00 00 00 00 00 00 00 01 01 01   ................
0xb21d9e2b  00 00 00 00 00 00 01 01 01 00 00 00 00 48 00 00   .............H..
0xb21d9e3b  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0xb21d9e4b  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0xb21d9e5b  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0xb21d9e6b  00 00 00 00 00 00 00 00 00 0c 9f 1d b2 d0 9d 1d   ................
0xb21d9e7b  b2 02 00 00 00 52 53 44 53 87 9a 86 de d8 5f d9   .....RSDS....._.
0xb21d9e8b  4f b7 b7 63 19 d7 47 86 28 01 00 00 00 62 3a 5c   O..c..G.(....b:\
```

Exercise: Key Takeways

1. By using knowledge of Windows Internals and "what normal looks like", we were able to spot the rogue lsass processes instantiated by stuxnet malware components.
2. Through further analysis, we isolated the injected dlls in one of the rogue lsass processes and verified our findings by uploading our samples to Virustotal.
3. This memory image provided an excellent example of "process hollowing", the technique malware uses to create a legitimate process and take it over.

Additional Resources for Stuxnet

- *Mark Russinovich's Analyzing a Stuxnet Infection with the Sysinternals Tools, Part I Part II Part III*
- *MNIN Security Blog: Coding, Reversing, Exploiting: Stuxnet's Footprint in Memory with Volatility 2.0*
- *Symantec's W32.Stuxnet Dossier*

Exercise 10 – Recovering Login Passwords

Objectives

- Through the use of the mimikatz Windows Debugger extension, extract Windows user credentials from the LSASS process
- Apply extracted credentials to gain access to a suspect user's password-protected zip file

Exercise Preparation

1. Open the Windows 8.1 VM provided with the course. Open a command prompt and change directory to C:\Tools where the standalone Volatility framework is located.

```
> cd C:\Tools
```

2. Note that in this lab, there are many command line options that contain zeroes. These are sometimes confused for the capital letter 'O'. As a helpful tip for this lab, the commands entered in Windows Debugger will only make use of the number **zero**.

3. Because this lab introduces several new concepts, there is only a "step by step" portion.

Exercise – Questions with Step-by-Step

1. **Image Identification**
   - On the Win8.1 VM, run the **`imageinfo`** plugin using the standalone Volatility 2.3.1 framework found in the C:\Tools directory against the **encryptedDocs.vmem** memory image in **C:\cases\exercise10**.
     i. When was the memory image created?

     2013-12-31 23:28:08 UTC

     ii. What profile should be specified when analyzing this image?

     Win7SP0x86

```
C:\Tools> volatility-2.3.1.standalone.exe -f
          C:\cases\exercise10\encryptedDocs.vmem imageinfo
```

```
Determining profile based on KDBG search...

          Suggested Profile(s) : Win7SP0x86, Win7SP1x86
                     AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                     AS Layer2 : FileAddressSpace (C:\cases\exercise10\encryptedDocs.vmem)

                      PAE type : PAE
                           DTB : 0x185000L
                          KDBG : 0x82970be8L
        Number of Processors : 1
     Image Type (Service Pack) : 0
                KPCR for CPU 0 : 0x82971c00L
             KUSER_SHARED_DATA : 0xffdf0000L
           Image date and time : 2013-12-31 23:28:08 UTC+0000
     Image local date and time : 2013-12-31 18:28:08 -0500
```

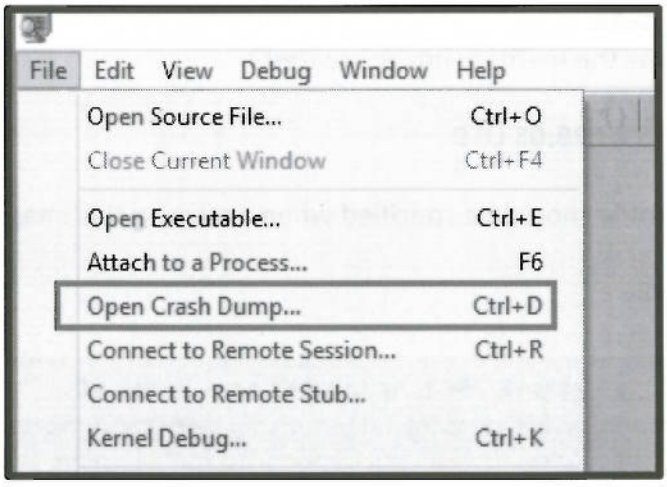
2. **Convert the encryptedDocs.vmem file to a crashdump using Volatility's plugin raw2dmp.**

```
C:\Tools> volatility-2.3.1.standalone.exe -f
          C:\cases\exercise10\encryptedDocs.vmem --profile=Win7SP0x86
          raw2dmp -O C:\cases\exercise10\encryptedDocs.dmp
```

```
c:\Tools>volatility-2.3.1.standalone.exe -f c:\cases\exercise10\encryptedDocs.vmem --profi
le=Win7SP0x86 raw2dmp -O c:\cases\exercise10\encryptedDocs.dmp
Volatility Foundation Volatility Framework 2.3.1
Writing data (5.00 MB chunks): |.....................................................
.....................................................................................
.....................................................|
```

3. **Load the crash dump in Windbg**

- Open the x86 instance Windbg.
- Use the "File -> Open Crash Dump" menu option (or press CTRL-D) to open the crash dump.



You should see the following output:

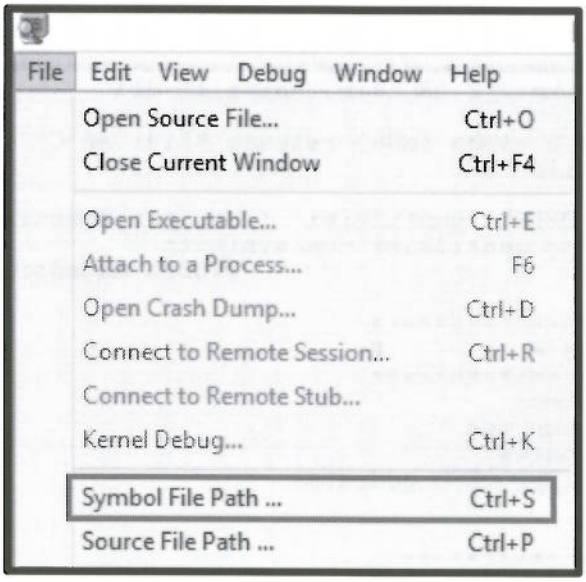
```
***     Type referenced: nt!_KPRCB                                              ***
***                                                                             ***
*******************************************************************************
Probably caused by : ntkrpamp.exe ( ntkrpamp.exe!unknown_error_in_process )

Followup: MachineOwner
---------

kd>
```

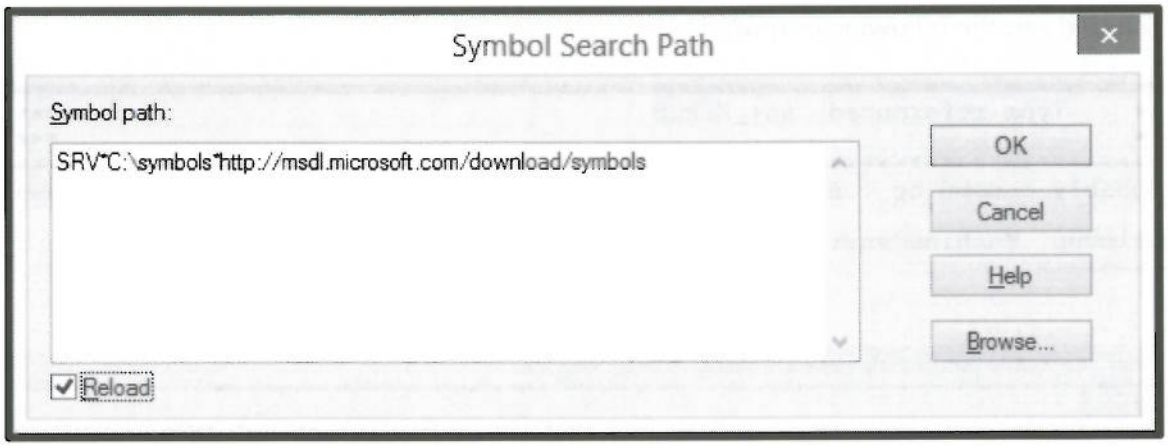
3. **Load Symbols**
   - You will need symbols loaded to get the appropriate output. Run the following two commands in the Windbg prompt.

```
.sympath SRV*C:\symbols*http://msdl.microsoft.com/download/symbols
```

```
.reload
```

   - Alternatively, you can use the menu prompts to add symbols as shown below:





4. **Load the mimikatz DLL**
   - Note that you must load the appropriate mimikatz DLL for the architecture of the memory image you are investigating. This will also be the same as the architecture Windbg you have launched (in this case x86).

```
.load C:\Tools\mimikatz_2.0RC\win32\mimilib.dll
```

**If using the x64 version of WinDbg when analyzing other memory images outside of class, you would type the following to point to the x64 mimikatz extension:

.load C:\Tools\mimikatz_2.0RC\x64\mimilib.dll

```
kd> .load C:\Tools\mimikatz_2.0RC\win32\mimilib.dll

  .#####.   mimikatz 2.0 alpha (x86) release "Kiwi en C" (Dec 30 2013 01:32:59)
 .## ^ ##.  Windows build 7600
 ## / \ ##  /* * *
 ## \ / ##   Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 '## v ##'   http://blog.gentilkiwi.com/mimikatz             (oe.eo)
  '#####'                                    WinDBG extension ! * * */

===================================
#          * Kernel mode *          #
===================================
# Search for LSASS process
0: kd> !process 0 0 lsass.exe
# Then switch to its context
0: kd> .process /r /p <EPROCESS address>
# And finally :
0: kd> !mimikatz
===================================
#          * User mode *            #
===================================
0:000> !mimikatz
===================================
```

5. **Determine the EPROCESS address for lsass.exe**

- To switch into the context of the lsass.exe process you will first need to identify the address of the EPROCESS structure of the lsass.exe process. You are concerned with the address of the lsass.exe process because this process handles authentication (and as such will have the passwords you seek).
- Type the following command into the Windbg prompt:

```
!process 0 0 lsass.exe
```

```
kd> !process 0 0 lsass.exe
PROCESS 868729c8  SessionId: 0  Cid: 01c0    Peb: 7ffdd000  ParentCid: 0150
    DirBase: 3eca20e0  ObjectTable: 88c20d38  HandleCount: 564.
    Image: lsass.exe
```

6. **Switch context to the lsass.exe process**

- Next, using the address of the EPROCESS block identified in the last step, switch context to the lsass.exe process. Switching contexts will load the appropriate DTB and page tables to access the lsass.exe process memory space.

```
.process /r /p 868729c8
```

```
kd> .process /r /p 868729c8
Implicit process is now 868729c8
Loading User Symbols
.....
```

7. **Dump the logged in user passwords from lsass.exe**

- Use the mimikatz plugin to dump passwords for logged on users. These passwords will usually be stored in plaintext due to requirements for backward compatibility and HTTP basic authentication.

```
!mimikatz
```

- Note that there are four users logged into the machine. User 'rob' (a prime suspect in the case) has a password that is fairly long and would be difficult to otherwise brute force. Because we know that users often reuse passwords, this is a golden find for an investigator trying to decipher encrypted files.

```
Authentication Id : 0 ; 351411 (00000000:00055cb3)
Session           : Interactive from 3
User Name         : rob
Domain            : win7_en_x86
SID               : S-1-5-21-1736815449-3557616216-3605033955-1003
        msv :
         [00000003] Primary
         * Username : rob
         * Domain   : win7_en_x86
         * LM       : e16bd2725b58c8ae0b5c593590135a1f
         * NTLM     : 7347077440ac4fbaa310c14c3af6fba6
         * SHA1     : 6dad1c09ed30eb866e83452e42146193c5ef9639
        tspkg :
         * Username : rob
         * Domain   : win7_en_x86
         * Password : exudegreatness
        wdigest :
         * Username : rob
         * Domain   : win7_en_x86
         * Password : exudegreatness
        kerberos :
         * Username : rob
         * Domain   : win7_en_x86
         * Password : exudegreatness
        ssp :
        masterkey :
         [00000000]
         * GUID :   {5f011561-2091-4e06-817e-7f89ab6a6029}
         * Time :   31/12/2013 23:27:09,375
         * Key :    35 1d a1 7c af 2d 86 d9 15 ce 2d 70 9a d1 09 35
```

- User 'jake' apparently uses his Twitter handle as his password. This is obviously bad for security.

```
Authentication Id : 0 ; 241494 (00000000:0003af56)
Session           : Interactive from 2
User Name         : jake
Domain            : win7_en_x86
SID               : S-1-5-21-1736815449-3557616216-3605033955-1001
        msv :
         [00000003] Primary
         * Username : jake
         * Domain   : win7_en_x86
         * LM       : 2ca53154f74775cf3c733079f5d12a29
         * NTLM     : 68021ec6bb26da6671244d0aa2981aaa
         * SHA1     : e7080fc1316e8f7a23e9be358f97454da732550a
        tspkg :
         * Username : jake
         * Domain   : win7_en_x86
         * Password : @malwarejake
        wdigest :
         * Username : jake
         * Domain   : win7_en_x86
         * Password : @malwarejake
        kerberos :
         * Username : jake
         * Domain   : win7_en_x86
         * Password : @malwarejake
        ssp :
        masterkey :
         [00000000]
         * GUID :   {e0f5c44e-42b6-4fd0-baee-999feda29862}
         * Time :   31/12/2013 23:26:44,406
         * Key :    79 02 cd c0 73 78 4f c4 dc 75 85 90 d5 ea ec b6 c2 75
```

- User 'alissa' also uses her Twitter handle as her password. Obviously some users need training to choose better passwords. Of course the key of this lab is not to pick on password selection, but to identify the fact that mimikatz can recover the plaintext passwords of logged in users.

```
Authentication Id : 0 ; 119396 (00000000:0001d264)
Session           : Interactive from 1
User Name         : alissa
Domain            : win7_en_x86
SID               : S-1-5-21-1736815449-3557616216-3605033955-1002
        msv :
         [00000003] Primary
         * Username : alissa
         * Domain   : win7_en_x86
         * LM       : fda6d47d7597e0f69b9a28e82c800b2e
         * NTLM     : e229d54e058d40a6682455c8f5c65148
         * SHA1     : 778cdd95e23b210e91b3a9432fe5a66034f2f804
        tspkg :
         * Username : alissa
         * Domain   : win7_en_x86
         * Password : @sibertor
        wdigest :
         * Username : alissa
         * Domain   : win7_en_x86
         * Password : @sibertor
        kerberos :
         * Username : alissa
         * Domain   : win7_en_x86
         * Password : @sibertor
        ssp :
        masterkey :
         [00000000]
         * GUID :   {ebbe2d24-3b2a-4826-ab9d-5c4bf500da28}
         * Time :   31/12/2013 23:26:14,812
         * Key :    bc ea 8f 75 14 5b 56 56 57 32 20 4c 7f ad 41 4b 15
```

8. **Use the passwords you've recovered to open the zip file C:\cases\exercise10\Secrets.zip.**
   - The passwords recovered with mimikatz can be used to decrypt the encrypted zip file.

9. **If you have extra time, try to recover the passwords used on other memory images provided for the course.**
   - Try an x64 memory image as well.

Exercise – Key Takeaways

- **Recovery of plaintext passwords is possible for logged in users.**
- **Four users were logged in, with at least one using a fairly strong password.**
- **Due to password reuse by the user "Rob", you were able his Windows password to decrypt the zip file and get the secret data. In this instance, brute forcing the zip file password would have been prohibitively difficult.**

This page intentionally left blank.

Exercise 11–Memory Forensics and Insider Cases

Scenario

You have been retained by Better Widgets Inc. (BW) to investigate a potential insider case. A rival company, Even Better Widgets Inc. (EBW), is suspected of planting an insider to smuggle information from the company. The insider is believed to have stolen archives containing customer data from the company's database. These archives may contain credit card numbers. BW estimates that thousands of credit cards were stolen. However, disk-based forensics have not turned up any good leads. Because BW produces e-voting machines, they have personal information for many millions of voters. If this information were released, the damage would be huge.

The suspect was terminated and removed from the building where he was working. HR says that there is no reason for this user to have accessed any customer records. Customer records at BW may contain social security numbers and credit card numbers (complete with CV2 values). If these are found on the suspect's machine, it would be a clear indication of wrongdoing.

Lucky for you, shortly after the suspect was removed from the premises, a memory capture of his machine was obtained. However, because the company's internal forensics analyst hasn't been through FOR526 yet, he didn't know what to do with the memory. That's where you come in. Use the skills you've learned up to this point to process this memory image.

Objectives

- Conduct a forensic memory analysis of a case involving a potential insider
- Gain experience using Bulk Extractor
- Use unstructured memory analysis to provide investigative findings

Exercise Preparation

1. Start a terminal and change into the **`/cases/exercise11`** directory. Unzip the archived system memory image **`insider-case.zip`**.

```
$ cd /cases/exercise11
$ unzip insider-case.zip
```

Exercise – Questions

1. **Image Identification**
   - Although this exercise focuses largely on unstructured analysis, structured analysis tools are still useful to confirm the date and time that the image was taken. Run the **`imageinfo`** plugin against the insider-case.vmem image to determine the OS, architecture, and when the image was obtained.

     ```
     cd /cases/exercise11
     vol.py -f insider-case.vmem imageinfo
     ```

     i. When was the memory image created?

     _____

     ii. What profile should be specified when analyzing this image?

     _____

2. **Process Image with Volatility**
   - Run the `pslist` plugin. Are there any exited processes listed? Are there any web browser processes listed?

     ```
     vol.py -f insider-case.vmem --profile=Win7SP1x64 pslist
     ```

     _____

     _____

   - Run the `netscan` plugin against the image. Does the netscan plugin show any evidence that the user was running a web browser? If so, what?

     ```
     vol.py -f insider-case.vmem --profile=Win7SP1x64 netscan
     ```

     _____

     _____

   - Run the `shellbags` plugin and examine the output. Is there evidence that the suspect may have accessed a network share?

     ```
     vol.py -f insider-case.vmem --profile=Win7SP1x64 shellbags
     ```

     _____

- Run the `userassist` plugin. Is there any evidence that the suspect used an anti-forensics tool (or evidence eliminator)?

```
vol.py -f insider-case.vmem --profile=Win7SP1x64 userassist
```

---

3. Run Bulk Extractor

- Based on the information you have been provided, the suspect may have sold customer's credit card numbers and used anti-forensic tools on his workstation. Run bulk extractor to determine what evidence can be extracted from the memory image.
- Create a new directory for the output named /cases/insider.

```
mkdir /cases/exercise11/insider
```

- Run bulk extractor using the default options. Because this may take a long time to perform, simply start the run and then terminate it. Instead of waiting for bulk extractor to finish, unzip the data in the file bulkExtractor-insider-precooked.zip.

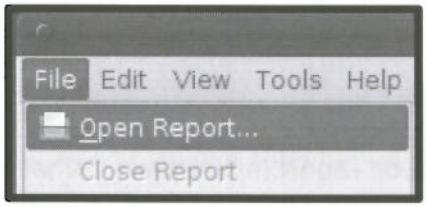
```
bulk_extractor -o insider insider-case.vmem

unzip bulkExtractor-insider-precooked.zip
```

4. Review Bulk Extractor Output

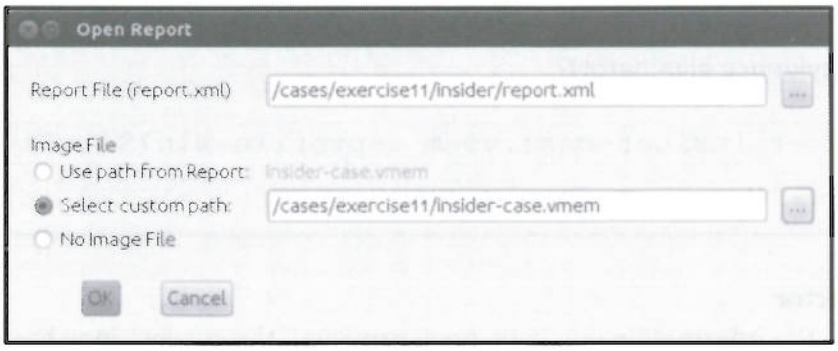
- Open a terminal window in the SIFT VM and type:

```
BEViewer &
```

- From BEViewer' toolbar, select File -> Open Report. These actions are shown in the screenshots below.



- In the Open Report dialog, select the report file /cases/exercise11/insider/report.xml and press Ok. If Because Bulk Extractor was run with a relative path for the image file, you will have to manually specify the memory image location. Note that the 'Select Image File' dialog has a display filter that will normally exclude .vmem files.



- Highlight the ccn.txt field in BEViewer. In the Feature File tab, click on the first result. Examine the Image pane. Does this appear to be a false positive or did the suspect have a list of credit card numbers on his machine?

_____

- Click on the ip_histogram.txt report in BEViewer. What two IP addresses have the most entries?

_____

_____

- Assuming that 192.168.221.131 is the IP of the suspect machine, the destination IP being communicated with most is 173.194.46.47. Run nslookup to determine what domain the IP belongs to.

```
nslookup 173.194.46.47
```

_____

- What type of activity might you expect to be attributed to this domain? Use the search engine of your choice to research the answer.

_____

_____

- Examine the packets carved from memory by opening the file /cases/insider/packets.pcap in wireshark. After a quick examination of the packets, what is dominant protocol of the packets carved? What implications does this have for forensics?

```
wireshark /cases/exercise11/insider/packets.pcap &
```

_____

_____

- To make the packet capture easier to analyze, remove all HTTPS traffic using a display filter. In the Filter tab enter the following and click Apply:

```
!(tcp.port == 443)
```

- Examine the remaining packets. Based on the packet capture, what was the hostname of the machine that this memory image was collected from?

_____

5. **Optional Homework #1:**
   - There are many additional artifacts of CCleaner that can be recovered through the use of the mftparser. Try using this plugin to recover additional artifacts.

6. **Optional Homework #2:**
   - Although page_brute was originally written to parse page files, it can also be useful for parsing a memory image. Try running page_brute against this suspect image and see what forensics artifacts you can recover. Don't forget to also try running page_brute with the new YARA signature you created in the earlier exercise.

1. **Image Identification**
   - Although this exercise focuses largely on unstructured analysis, structured analysis tools are still useful to confirm the date and time that the image was taken. Run the **`imageinfo`** plugin against the insider-case.vmem image to determine the OS, architecture, and when the image was obtained.
     i. When was the memory image created?

     2013-11-23 22:08:57 UTC

     ii. What profile should be specified when analyzing this image?

     Win7SP1x64

```
user@SIFT$ vol.py -f insider-case.vmem imageinfo
Determining profile based on KDBG search...

          Suggested Profile(s) : Win2008R2SP0x64, Win7SP1x64, Win7SP0x64, Win2008R2SP1x64
                     AS Layer1 : AMD64PagedMemory (Kernel AS)
                     AS Layer2 : FileAddressSpace (/cases/insider-case.vmem)
                      PAE type : No PAE
                           DTB : 0x187000L
                          KDBG : 0xf800029fa0a0
        Number of Processors : 2
    Image Type (Service Pack) : 1
               KPCR for CPU 0 : 0xfffff800029fbd00L
               KPCR for CPU 1 : 0xfffff880009e9000L
            KUSER_SHARED_DATA : 0xfffff78000000000L
          Image date and time : 2013-11-23 22:08:57 UTC+0000
    Image local date and time : 2013-11-23 17:08:57 -0500
```

2. **Process Image with Volatility**
   - Run the `pslist` plugin. Are there any exited processes listed? Are there any web browser processes listed?

     `vol.py -f insider-case.vmem --profile=Win7SP1x64 pslist`

     There is one exited process (soffice.bin). There are no web browser processes running.

```
user@SIFT$ vol.py -f insider-case.vmem --profile=Win7SP1x64 pslist
Offset(V)          Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start
------------------ -------------------- ------ ------ ------ -------- ------ ------ ------------------------------
0xfffffa80018ae040 System                    4      0     93      548 ------      0 2013-11-23 21:30:56 UTC+0000
0xfffffa8001eb6670 smss.exe                252      4      2       30 ------      0 2013-11-23 21:30:56 UTC+0000
0xfffffa8002048060 csrss.exe               344    328      9      433      0      0 2013-11-23 21:30:59 UTC+0000
0xfffffa8002d8bb30 wininit.exe             400    328      3       77      0      0 2013-11-23 21:31:03 UTC+0000
0xfffffa8002d8eb30 csrss.exe               408    392     10      431      1      0 2013-11-23 21:31:03 UTC+0000
0xfffffa8002df1b30 services.exe            452    400      7      207      0      0 2013-11-23 21:31:03 UTC+0000
0xfffffa8002e03b30 lsass.exe               460    400      6      594      0      0 2013-11-23 21:31:03 UTC+0000
0xfffffa8002e0ab30 lsm.exe                 468    400     11      146      0      0 2013-11-23 21:31:03 UTC+0000
0xfffffa8002e4ab30 winlogon.exe            516    392      5      128      1      0 2013-11-23 21:31:03 UTC+0000
0xfffffa80031b7060 svchost.exe             616    452     13      362      0      0 2013-11-23 21:31:04 UTC+0000
0xfffffa8003200b30 svchost.exe             700    452      9      304      0      0 2013-11-23 21:31:04 UTC+0000
0xfffffa8003231b30 svchost.exe             780    452     22      538      0      0 2013-11-23 21:31:04 UTC+0000
0xfffffa800327b890 svchost.exe             828    452     16      390      0      0 2013-11-23 21:31:05 UTC+0000
0xfffffa80032aab30 svchost.exe             880    452     31      907      0      0 2013-11-23 21:31:05 UTC+0000
0xfffffa8003325b30 svchost.exe             112    452     18      707      0      0 2013-11-23 21:31:05 UTC+0000
```

- Run the `netscan` plugin against the image. Does the netscan plugin show any evidence that the user was running a web browser? If so, what?

```
vol.py -f insider-case.vmem --profile=Win7SP1x64 netscan
```

Multiple closed network connections show the process iexplore.exe. This offers evidence of browser use. If no browser history is present, then this may indicate that the suspect has intentionally cleared it or used private browsing.

```
0x7e473be0     TCPv4    -:49213          173.194.115.16:80    CLOSED     1768    iexplore.exe
0x7e4d5010     TCPv4    -:49221          173.194.115.17:80    CLOSED     1768    iexplore.exe
0x7e4fa890     TCPv4    -:50763          63.63.63.63:80       CLOSED     5       LX????ZJ?
0x7e51d010     TCPv4    -:49222          173.194.115.19:80    CLOSED     63966
0x7e51ecf0     TCPv4    -:49230          23.63.226.106:80     CLOSED     1768    iexplore.exe
0x7e52e770     TCPv4    -:49232          23.63.226.106:80     CLOSED     1768    iexplore.exe
```

- Run the `shellbags` plugin and examine the output. Is there evidence that the suspect may have accessed a network share?

```
vol.py -f insider-case.vmem --profile=Win7SP1x64 shellbags
```

Yes, it appears that the suspect may have accessed a network share based on the Z:\fileshare entry in the shellbags output. Z: is not a standard drive letter for a local drive and other examination can confirm that the system drive was c:. Further examination of the registry (left as an exercise for the student) can confirm whether this was actually a network share.

```
      File Attr              Path
------ ----
2013-08-15 22:45:44 UTC+0000   DIR                 Z:\fileshare
```

- Run the `userassist` plugin. Is there any evidence that the suspect used an anti-forensics tool (or evidence eliminator)?

```
vol.py -f insider-case.vmem --profile=Win7SP1x64 userassist
```

Yes, the userassist output clearly shows that the CCleaner was installed on the machine. If this is not used by systems admins, this would point to the use of anti-forensics tool use.

```
REG_BINARY   {6D809377-6AF0-444B-8957-A3773F02200E}\CCleaner\uninst.exe :
Count:        1
Focus Count:  0
Time Focused:  0:00:00.500000
Last updated:  2013-11-23 22:08:25 UTC+0000
0x00000000  00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
0x00000010  00 00 80 bf 00 00 80 bf 00 00 80 bf 00 00 80 bf  ................
0x00000020  00 00 80 bf 00 00 80 bf 00 00 80 bf 00 00 80 bf  ................
0x00000030  00 00 80 bf 00 00 80 bf ff ff ff ff 30 88 cd 87  ............0...
0x00000040  98 e8 ce 01 00 00 00 00                          ........
---------------------------
```

3. **Run Bulk Extractor**
   - Based on the information you have been provided, the suspect may have sold customer's credit card numbers and used anti-forensic tools on his workstation. Run bulk extractor to determine what evidence can be extracted from the memory image.
   - Run bulk extractor using the default options. Because this may take a long time to perform, simply start the run and then terminate it. Instead of waiting for bulk extractor to finish, unzip the data in the file bulkExtractor-insider-precooked.zip.

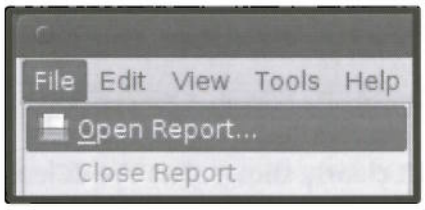
```
sansforensics@SIFT-Workstation:/cases$ bulk_extractor -o insider insider-case.vmem
bulk_extractor version: 1.4.1
Hostname: SIFT-Workstation
Input file: insider-case.vmem
Output directory: insider
Disk Size: 2147483648
Threads: 4
 7:31:14 Offset 67MB (3.12%) Done in  0:02:57 at 07:34:11
 7:31:20 Offset 150MB (7.03%) Done in  0:02:32 at 07:33:52
 7:31:26 Offset 234MB (10.94%) Done in  0:02:24 at 07:33:50
 7:31:31 Offset 318MB (14.84%) Done in  0:02:10 at 07:33:41
 7:31:39 Offset 402MB (18.75%) Done in  0:02:14 at 07:33:53
```

```
bulk_extractor -o insider insider-case.vmem

unzip bulkExtractor-insider-precooked.zip
```

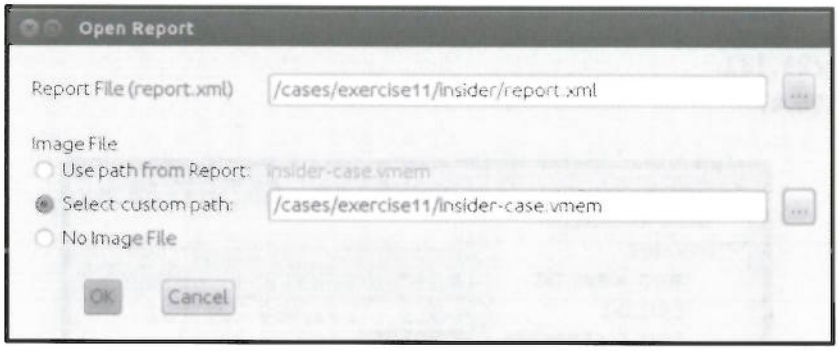
4. **Review Bulk Extractor Output**
   - Open a terminal window in the SIFT VM and type:

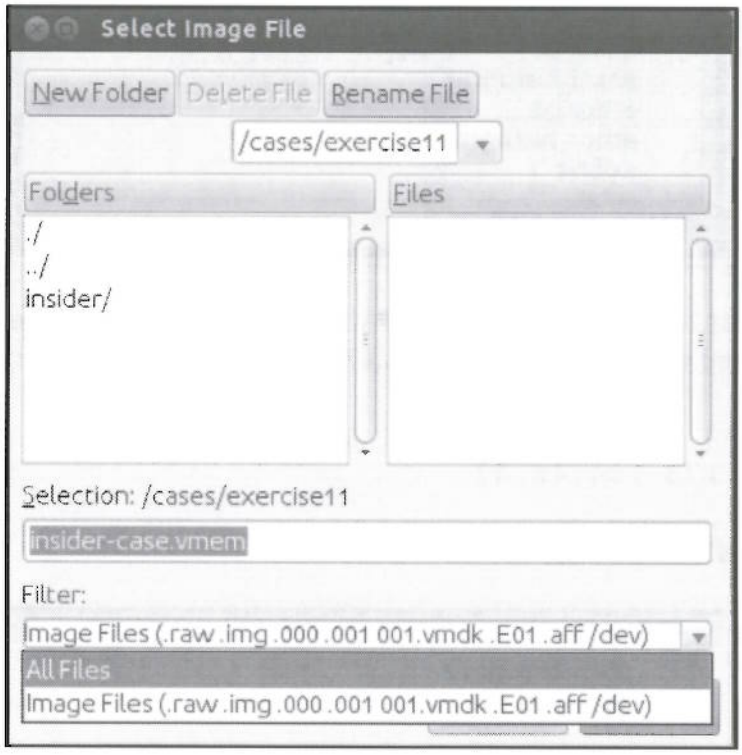
     ```
     BEViewer &
     ```

   - From BEViewer' toolbar, select File -> Open Report. These actions are shown in the screenshots below.



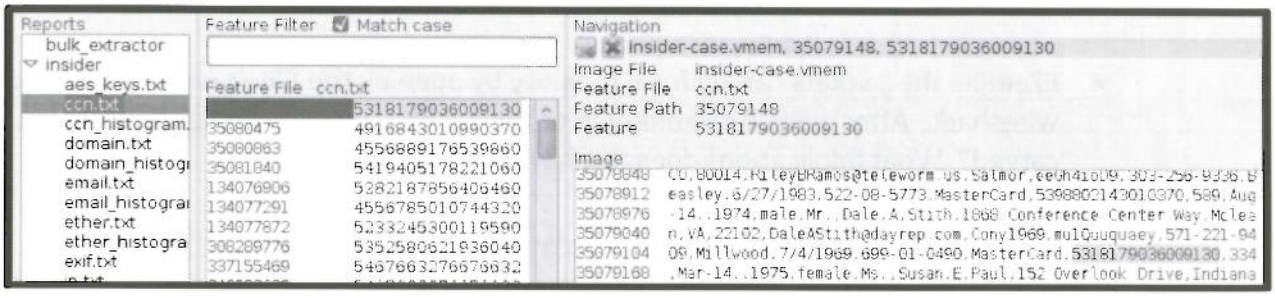
   - In the Open Report dialog, select the report file /cases/exercise11/insider/report.xml and press Ok. If Because Bulk Extractor was run with a relative path for the image file, you will have to manually specify the memory image location. Note that the 'Select Image File' dialog has a display filter that will normally exclude .vmem files.





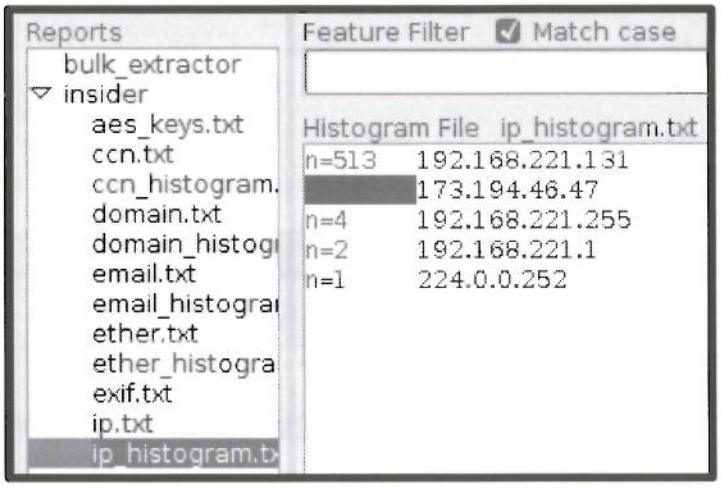
- Highlight the ccn.txt field in BEViewer. In the Feature File tab, click on the first result. Examine the Image pane. Does this appear to be a false positive or did the suspect have a list of credit card numbers on his machine?

  Based on the output, it appears that the suspect had a list of credit card numbers and other personally identifiable information open in memory at some point.



- Click on the ip_histogram.txt report in BEViewer. What two IP addresses have the most entries?

192.168.221.131
173.194.46.47



- Assuming that 192.168.221.131 is the IP of the suspect machine, the destination IP being communicated with most is 173.194.46.47. Run nslookup to determine what domain the IP belongs to.

```
nslookup 173.194.46.47
```

ord08s10-in-f15.1e100.net

```
sansforensics@SIFT-Workstation:/cases$ nslookup 173.194.46.47
Server:         192.168.25.2
Address:        192.168.25.2#53

Non-authoritative answer:
47.46.194.173.in-addr.arpa      name = ord08s10-in-f15.1e100.net.
```

- What type of activity might you expect to be attributed to this domain? Use the search engine of your choice to research the answer.

This domain is associated with Google application services. This may indicate that the suspect used some Google application (gmail, chat, etc.).

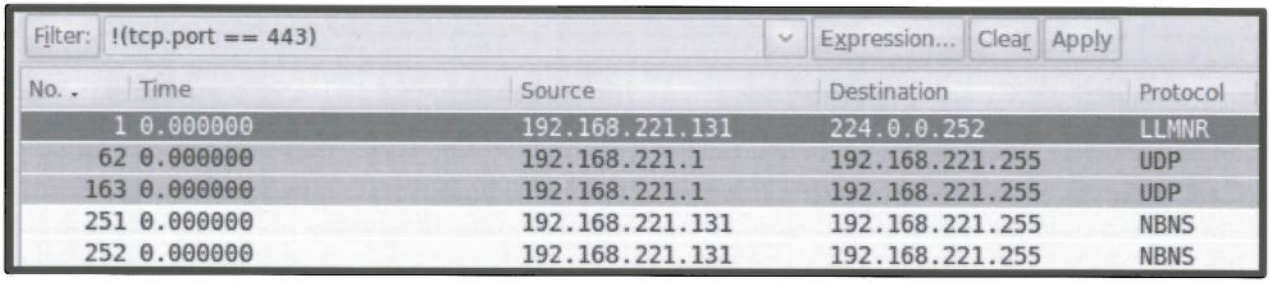
- Examine the packets carved from memory by opening the file /cases/insider/packets.pcap in wireshark. After a quick examination of the packets, what is dominant protocol of the packets carved? What implications does this have for forensics?

```
wireshark /cases/exercise11/insider/packets.pcap &
```

The vast majority of packets are HTTPS (TCP port 443). Unless the suspect's traffic passed through an SSL stripping proxy, these communications will be unrecoverable.

- To make the packet capture easier to analyze, remove all HTTPS traffic using a display filter. In the Filter tab enter the following and click Apply:

```
!(tcp.port == 443)
```



- Examine the remaining packets. Based on the packet capture, what was the hostname of the machine that this memory image was collected from?

  WIN-8TVP0GNM4H3

5. **Optional Homework #1:**
   - There are many additional artifacts of CCleaner that can be recovered through the use of the mftparser. Try using this plugin to recover additional artifacts.

6. **Optional Homework #2:**
   - Although page_brute was originally written to parse page files, it can also be useful for parsing a memory image. Try running page_brute against this suspect image and see what forensics artifacts you can recover. Don't forget to also try running page_brute with the new YARA signature you created in the earlier exercise.

Exercise – Key Takeaways

1. This poor suspect is guilty!
   a. Identified credit card numbers and PII in the suspect's memory image. There was no legitimate business use for this data.
   b. The suspect installed CCleaner, an anti-forensics tool. At the very least this is an acceptable use policy violation.
   c. Based on network data, it is possible that the suspect used some type of service with Google.
   d. The suspect used Internet Explorer previously (based on volatility netscan output).
2. Both structured and unstructured memory analysis were useful in this case. Unique artifacts were found using each technique.
3. Although disk and network forensics might add to the investigation, you were able to uncover many findings using only memory forensics.

This page intentionally left blank.

EXERCISE 12 - BLACK ENERGY

Objectives

- Conduct a forensic memory analysis using the six memory analysis steps
- Gain experience using Volatility and Rekall
- Examine the advanced variant **malfind** plugin
- Practice extracting a suspicious driver from a memory image for further analysis

Exercise Preparation

1. Start a terminal and change into the **/cases/exercise12** directory. Unzip **be2.zip**.

```
$ cd /cases/exercise12
$ unzip be2.zip
```

Exercise - Questions

Overview: The Black Energy rootkit is one of the earliest known examples of cyber-warfare. Its use for DDOS dates to late 2008 when it was employed by Russia during the Russian-Georgia war. It provides a good test piece to analyze output from some of the more complicated Volatility plugins. Dell published an excellent malware analysis report on the Black Energy malware which can found here:

http://www.secureworks.com/cyber-threat-intelligence/threats/blackenergy2/

The walkthrough for this lab utilizes both rekall and Volatility™. Try using both frameworks throughout the lab to perform your analysis. This is one lab where there are definitely some differences that can be observed.

1. Enumerate processes of the **be2.vmem**. Identify "zombies" that are terminated but have not yet been reaped from the doubly-linked list of processes.

_____

2. Display the command line of the "**1e0f1b9b697ab49(...).exe**" process using one of the techniques taught in the class. Why is the command line that was used to instantiate the process not parsable?

_____

_____

3. Utilizing any of the other plugins or methods discussed in this class, determine the full path of the "**1e0f1b9b697ab49(...).exe**" process.

_____

4. Run the **malfind** plugin (specifying a dump directory) to search for injected memory pages. Which process shows the highest likelihood of code injection (include name, PID, and Address)?

_____

5. Review your **malfind** dump directory and find the output file that relates to the injected memory section. (Hint: The filename contains the section memory address.)

_____

6. Start your rootkit analysis by running the **ssdt** plugin -- remember to filter out legitimate SSDT entries by piping your results to **egrep -v '(ntoskrnl|win32k)'**. How many instructions were hooked and by what module?

_____

_____

7. Use the **modules** or **modscan** plugin to better identify the hooking driver found in the **ssdt** output (search for the module name). Write down the base address.

_____

8. Use **moddump** to extract the suspicious driver. Submit the driver and the memory section identified by **malfind** to VirusTotal to confirm your findings (Hint: anti-virus has a harder time identifying static malware samples and thus often will use a generic hit descriptor).

Exercise – Questions with Step-by-Step

1. Enumerate processes of the **be2.vmem**. Identify "zombies" that are terminated but have not yet been reaped from the doubly-linked list of processes.

```
$ rekall -f be2.vmem pslist
```

```
user@SIFT$ rekall -f be2.vmem pslist
_EPROCESS          Name              PID   PPID   Thds     Hnds     Sess  Wow64          Start                        Exit
---------- -------------------- ----- ------ ------ -------- ------ ------ ------------------------ ------------------------
0x810b1660 System                    4      0     59      179      - False  -                        -
0xff3802c8 VMip.exe                180    212      0        -      0 False  2010-08-15 19:22:11+0000 2010-08-15 19:22:11+0000
0xff3b1d78 cmd.exe                 212   1668      0        -      0 False  2010-08-15 19:22:11+0000 2010-08-15 19:22:11+0000
0xff25a7e0 alg.exe                 216    676      7      110      0 False  2010-08-11 06:06:39+0000 -
0xff3667e8 VMwareTray.exe          432   1724      1       49      0 False  2010-08-11 06:09:31+0000 -
0xff374980 VMwareUser.exe          452   1724      8      203      0 False  2010-08-11 06:09:32+0000 -
0x80f94588 wuauclt.exe             468   1028      5      134      0 False  2010-08-11 06:09:37+0000 -
0xff1f6da0 1e0f1b9b697ab49         476   1724      0        -      0 False  2010-08-15 19:21:25+0000 2010-08-15 19:21:27+0000
0xff2ab020 smss.exe                544      4      3       21      - False  2010-08-11 06:06:21+0000 -
0xff1ecda0 csrss.exe               608    544     11      400      0 False  2010-08-11 06:06:23+0000 -
0xff1ec978 winlogon.exe            632    544     22      519      0 False  2010-08-11 06:06:23+0000 -
0xff247020 services.exe            676    632     16      268      0 False  2010-08-11 06:06:24+0000 -
0xff255020 lsass.exe               688    632     22      348      0 False  2010-08-11 06:06:24+0000 -
0xff218230 vmacthlp.exe            844    676      1       24      0 False  2010-08-11 06:06:24+0000 -
0x80ff88d8 svchost.exe             856    676     19      321      0 False  2010-08-11 06:06:24+0000 -
0xff364310 wscntfy.exe             888   1028      1       27      0 False  2010-08-11 06:06:49+0000 -
0xff217560 svchost.exe             936    676      9      261      0 False  2010-08-11 06:06:24+0000 -
0x80fbf910 svchost.exe            1028    676     87     1394      0 False  2010-08-11 06:06:24+0000 -
0xff38b5f8 TPAutoConnect.e        1084   1968      1       61      0 False  2010-08-11 06:06:52+0000 -
0xff22d558 svchost.exe            1088    676      7       81      0 False  2010-08-11 06:06:25+0000 -
0xff203b80 svchost.exe            1148    676     15      212      0 False  2010-08-11 06:06:26+0000 -
0xff1d7da0 spoolsv.exe            1432    676     15      137      0 False  2010-08-11 06:06:26+0000 -
0x80f1b020 cmd.exe                1572    476      0        -      0 False  2010-08-15 19:21:27+0000 2010-08-15 19:21:27+0000
0xff1b8b28 vmtoolsd.exe           1668    676      6      222      0 False  2010-08-11 06:06:35+0000 -
0xff3865d0 explorer.exe           1724   1708     13      309      0 False  2010-08-11 06:09:29+0000 -
0x80f60da0 wuauclt.exe            1732   1028      7      178      0 False  2010-08-11 06:07:44+0000 -
0xff1fdc88 VMUpgradeHelper        1788    676      5      100      0 False  2010-08-11 06:06:38+0000 -
0xff143b28 TPAutoConnSvc.e        1968    676      5      100      0 False  2010-08-11 06:06:39+0000 -
```

Processes 476, 1572, 212, 180 have no running threads or handles, yet are still included in the doubly-linked list. This occurs in some cases, due to another process having a handle open to these processes, disallowing the kernel from removing and destroying them.
The process "1e0f1b9b697ab49(...).exe"PID 476 that was spawned from a command shell, PID 1572, should raise suspicions due to its odd name and suspicious parent.

2. Display the command line of the "**1e0f1b9b697ab49(...).exe**" process using one of the techniques taught in the class. Why is the command line that was used to instantiate the process not parsable?

```
user@SIFT$ vol.py -f be2.vmem --profile=WinXPSP2x86 dlllist -p 476
**************************************************************************
1e0f1b9b697ab49 pid:    476
Unable to read PEB for task.
user@SIFT$
```

Note that rekall does not show the same level of detail in the error message:

```
user@SIFT$ rekall -f be2.vmem dlllist --pid=476
************************************************
1e0f1b9b697ab49 pid: 476
Command line : -


   Base       Size          Load Reason/Count        Path
---------- ---------- ------------------------------ ----
user@SIFT$
```

Although the EPROCESS block still exists and is linked to the active process list via flinks and blinks, the process no longer has a Process Environment Block. Without access to the PEB, we cannot determine the command line of the process.

3. Utilizing any of the other plugins or methods discussed in this class, determine the full path of the "**1e0f1b9b697ab49(…).exe"** process.

```
user@SIFT$ rekall -f be2.vmem userassist
---------------------------
Registry: \Device\HarddiskVolume1\Documents and Settings\Administrator\NTUSER.DAT @ 0xe1da4008
Key path: $$$PROTO.HIV/Software/Microsoft/Windows/CurrentVersion/Explorer/UserAssist/{5E6AB780-7743-11CF-A12B-
ount
Last updated: 2010-06-10 16:11:44+0000

Subkeys:

Values:
```

…. Snip ….

```
REG_BINARY    UEME_RUNPATH:C:\Documents and Settings\Administrator\Desktop\1e0f1b9b697ab498833f5951b2a01b2f.exe :
ID:             2
Count:          1
Last updated:   2010-08-15 19:21:25+0000
  Offset                         Hex                                Data
---------- ----------------------------------------------- ----------------
       0x0 02 00 00 00 06 00 00 00 70 aa d3 0d af 3c cb 01  ........p....<..
```

With our knowledge of that fact that this process has executed in the past, we can anticipate that there will be evidence of execution in the NTUser.dat of the user currently logged in. By running the `userassist` plugin, we can extract the full path of this process from the `userassist` key and determine how many times this user has run "1e0f1b9b697ab49(…).exe".

4. Run the **`malfind`** plugin to search for injected memory pages. Which process shows the highest likelihood of code injection (include name, PID, and Address)?

```
user@SIFT$ rekall -f be2.vmem malfind -D output
**************************************************
Process: svchost.exe Pid: 856 Address: 0xc30000
Vad Tag: VadS Protection: EXECUTE_READWRITE
Flags: CommitCharge: 9, MemCommit: 1, PrivateMemory: 1, Protection: 6

0xc30000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00  MZ..............
0xc30010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
0xc30020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xc30030 00 00 00 00 00 00 00 00 00 00 00 00 f8 00 00 00  ................

  0xc30000    - 4d                    DEC EBP
  0xc30001    - 5a                    POP EDX
  0xc30002    - 90                    NOP
  0xc30003    - 0003                  ADD [EBX], AL
  0xc30005    - 0000                  ADD [EAX], AL
  0xc30007    - 000400                ADD [EAX+EAX], AL
  0xc3000a    - 0000                  ADD [EAX], AL
```

The svchost.exe process with PID 856 and section address 0xc30000 is the most likely candidate for injection due to it containing a legitimate PE32 portable executable header ("MZx90x00"). The disassembly snippets from other processes do not appear to represent legitimate code.

5. Review your **malfind** dump directory and find the output file that relates to the injected memory section. (Hint: The filename contains the section memory address.)

   The name of the output file is: process.0x80ff88d8.0xc30000.dmp

6. Start your rootkit analysis by running the **ssdt** plugin -- remember to filter out legitimate SSDT entries by piping your results to **egrep -v '(ntoskrnl|win32k)'**. How many instructions were hooked and by what module?

```
$ vol.py -f be2.vmem --profile=WinXPSP2x86 ssdt | egrep -v
'ntoskrnl|win32k'
```

```
user@SIFT$ vol.py -f be2.vmem --profile=WinXPSP2x86 ssdt |egrep -v 'ntoskrnl|win32k'
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at ff3aab90 with 284 entries
  Entry 0x0041: 0xff0d2487 (NtDeleteValueKey) owned by 00004A2A
  Entry 0x0047: 0xff0d216b (NtEnumerateKey) owned by 00004A2A
  Entry 0x0049: 0xff0d2267 (NtEnumerateValueKey) owned by 00004A2A
  Entry 0x0077: 0xff0d20c3 (NtOpenKey) owned by 00004A2A
  Entry 0x007a: 0xff0d1e93 (NtOpenProcess) owned by 00004A2A
  Entry 0x0080: 0xff0d1f0b (NtOpenThread) owned by 00004A2A
  Entry 0x0089: 0xff0d2617 (NtProtectVirtualMemory) owned by 00004A2A
  Entry 0x00ad: 0xff0d1da0 (NtQuerySystemInformation) owned by 00004A2A
  Entry 0x00ba: 0xff0d256b (NtReadVirtualMemory) owned by 00004A2A
  Entry 0x00d5: 0xff0d2070 (NtSetContextThread) owned by 00004A2A
  Entry 0x00f7: 0xff0d2397 (NtSetValueKey) owned by 00004A2A
```

Note that in this case, rekall does not detect the SSDT hooks appropriately.

```
user@SIFT$ rekall -f be2.vmem ssdt |egrep -v 'nt!|win32k!'

************** Table 0 @ 0x80501030 **************
  Entry      Target    Symbol
---------- ---------- ------

************** Table 1 @ 0xbf997600 **************
  Entry      Target    Symbol
---------- ---------- ------
```

7. Use the **modules** or **modscan** plugin to better identify the hooking driver found in the **ssdt** output (search for the module name). Write down the base address.

```
# vol.py -f be2.vmem --profile=WinXPSP2x86 modules
```

```
user@SIFT$ rekall -f be2.vmem modules
---------- -------------------- ---------- ---------- ----
0x80f0c050 rasacd.sys           0xfc174000     0x3000 \SystemRoot\system32\DRIVERS\rasacd.sys
0x80fb6c88 hidusb.sys           0xfbf42000     0x3000 \SystemRoot\system32\DRIVERS\hidusb.sys
0x810d6a08 vmscsi.sys           0xfc8b7000     0x3000 vmscsi.sys
0x80fa4618 usbuhci.sys          0xfc77b000     0x5000 \SystemRoot\system32\DRIVERS\usbuhci.sys
0x80fba140 usbhub.sys           0xfc64b000     0xf000 \SystemRoot\system32\DRIVERS\usbhub.sys
0x80f724f8 usbccgp.sys          0xfc7e3000     0x8000 \SystemRoot\system32\DRIVERS\usbccgp.sys
0x80fa67f0 cdrom.sys            0xfc56b000     0xd000 \SystemRoot\system32\DRIVERS\cdrom.sys
0x80fedbb8 mssmbios.sys         0xfc967000     0x4000 \SystemRoot\system32\DRIVERS\mssmbios.sys
0x80f04700 mrxdav.sys           0xf35d8000    0x2d000 \SystemRoot\system32\DRIVERS\mrxdav.sys
0xff375b08 00004A2A             0xff0d1000     0x8361 00004A2A
0xff3d87e0 msgpc.sys            0xfc5fb000     0x9000 \SystemRoot\system32\DRIVERS\msgpc.sys
0x81003510 mrxsmb.sys           0xf3aa6000    0x6f000 \SystemRoot\system32\DRIVERS\mrxsmb.sys
```

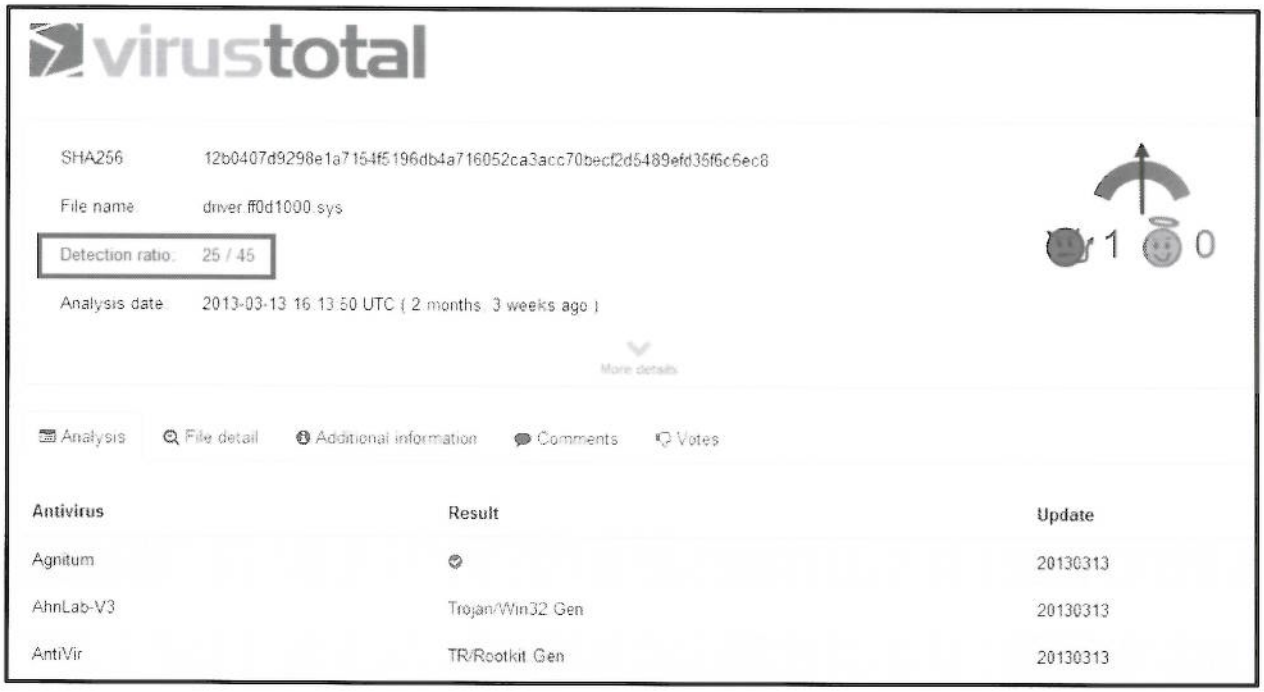
```
user@SIFT$ vol.py -f be2.vmem --profile=WinXPSP2x86 modules
---------- -------------------- ---------- ---------- ----
0x810d6a08 vmscsi.sys           0xfc8b7000     0x3000 vmscsi.sys
0x80fa67f0 cdrom.sys            0xfc56b000     0xd000 \SystemRoot\system32\DRIVERS\cdrom.sys
0xff3d87e0 msgpc.sys            0xfc5fb000     0x9000 \SystemRoot\system32\DRIVERS\msgpc.sys
0x80fedbb8 mssmbios.sys         0xfc967000     0x4000 \SystemRoot\system32\DRIVERS\mssmbios.sys
0x80f0c050 rasacd.sys           0xfc174000     0x3000 \SystemRoot\system32\DRIVERS\rasacd.sys
0x80efc0c8 dump_vmscsi.sys      0xfbf36000     0x3000 \SystemRoot\System32\Drivers\dump_vmscsi.sys
0xff375b08 00004A2A             0xff0d1000     0x8361 00004A2A
```

The base address of module 00004A2A is 0xff0d1000

8. Use **moddump** to extract the suspicious driver. Submit the driver and the memory section identified by **malfind** to VirusTotal to confirm your findings. (Hint: anti-virus has a harder time identifying static malware samples and thus often will use a generic hit descriptor)

```
# mkdir moddump-output
# vol.py -f be2.vmem --profile=WinXPSP2x86
  moddump -b 0xff0d1000 --dump-dir=./moddump-output
```

```
user@SIFT$ mkdir moddump-output
user@SIFT$ vol.py -f be2.vmem --profile=WinXPSP2x86 moddump -D moddump-output/ -b 0xff0d1000
Module Base Module Name          Result
----------- -------------------- ------
0x0ff0d1000 00004A2A             OK: driver.ff0d1000.sys
```



Exercise: Key Takeaways

1. As we might expect, the more memory images we look through, the more anomalies we will encounter. In this image for example, we see evidence of "zombie" processes, terminated processes that linger in the doubly-linked list of active processes. Their presence is not indicative of malicious activity but occurs naturally in the wild.
2. Through the use of advanced analysis Volatility plugins, such as malfind and ssdt, we were able to isolate an injected dll in a svcshost.exe process and dump it for further analysis.
3. In addition, we identified a driver responsible for hooking 14 different functions in the System Service Descriptor Table (SSDT). Using moddump, we isolate this driver and uploaded it to Virustotal for further analysis and identification.

This page intentionally left blank.

Exercise 13 – Linux Memory Acquisition

Objectives

- Capture a memory image from a Linux system
- Analyze capture memory image with Rekall Memory Forensics Framework to assess system state
- Identify key aspects of Linux process enumeration and system triage through the use of Rekall Memory Forensics Framework

Exercise Preparation

- In the Ubuntu SIFT VM, start a terminal and change to root by running "**sudo su -**".

Change to the **/cases/exercise14** directory and unarchive the **linux_pmem_1.0RC1.tgz** using the command below, creating the **linux** subdirectory.

```
$ sudo su -
```

```
# cd /cases/exercise14
# tar vxzf linux_pmem_1.0RC1.tgz
```

Exercise Part 1: Linux Kernel Profile Creation

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

1. Change into the **/cases/exercise14/linux** directory and compile the kernel specific driver to gain access to your target Linux system.

```
# cd /cases/exercise14/linux
# make
```

```
root@siftworkstation:/cases/exercise14/linux# make
make -C /usr/src/linux-headers-3.11.0-26-generic CONFIG_DEBUG_INFO=y M=`pwd` modules
make[1]: Entering directory `/usr/src/linux-headers-3.11.0-26-generic'
  CC [M]  /cases/exercise14/linux/module.o
  CC [M]  /cases/exercise14/linux/pmem.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /cases/exercise14/linux/module.mod.o
  LD [M]  /cases/exercise14/linux/module.ko
  CC      /cases/exercise14/linux/pmem.mod.o
  LD [M]  /cases/exercise14/linux/pmem.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.11.0-26-generic'
cp module.ko module_dwarf.ko
make -C /usr/src/linux-headers-3.11.0-26-generic M=`pwd` modules
make[1]: Entering directory `/usr/src/linux-headers-3.11.0-26-generic'
  CC [M]  /cases/exercise14/linux/module.o
  CC [M]  /cases/exercise14/linux/pmem.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /cases/exercise14/linux/module.mod.o
  LD [M]  /cases/exercise14/linux/module.ko
  CC      /cases/exercise14/linux/pmem.mod.o
  LD [M]  /cases/exercise14/linux/pmem.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.11.0-26-generic'
zip "`uname -r`.zip" module_dwarf.ko /boot/System.map-`uname -r`
  adding: module_dwarf.ko (deflated 66%)
  adding: boot/System.map-3.11.0-26-generic (deflated 79%)
```

Second Step: Load Driver

1. In order to gain access to physical memory of the target Linux system, the examiner must load the newly created kernel driver "**pmem.ko**" using the **insmod** command. "**Insmod**" installs a loadable module in the running kernel.
   Note: Be sure that you are loading the driver as root.

```
# insmod /cases/exercise14/linux/pmem.ko
```

```
root@siftworkstation:/cases/exercise14/linux# ls
3.11.0-26-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
```

Third Step: Ensure Driver is Running

- List loaded modules and search for the pmem.ko module to verify it has successfully loaded.

```
# lsmod |grep pmem
```

```
root@siftworkstation:/cases/exercise14/linux# lsmod |grep pmem
pmem                   12680  0
```

Fourth Step: Acquire Memory

1. Acquire physical memory of the target Linux system using the dd tool. (This should take a few minutes and depending on how much RAM you have assigned your Linux VM.)

```
# dd if=/dev/pmem of=/cases/linux.raw conv=noerror,sync
```

```
root@siftworkstation:/cases/exercise14/linux# dd if=/dev/pmem of=/cases/exercise14
/linux.raw conv=noerror,sync
4194303+1 records in
4194304+0 records out
2147483648 bytes (2.1 GB) copied, 13.5586 s, 158 MB/s
```

Fifth Step: Apply profile to Captured Memory Image

- Convert the profile created by pmem located in the /cases/exercise14/linux directory to a usable profile that Rekall Memory Forensic Framework can use to parse the Linux memory image. (Note: Your profile name could be different from that shown below due to your version of Ubuntu.)

```
# cd /cases/exercise14/linux
# rekal convert_profile 3.11.0-26-generic.zip Ubuntu3.11.0.26-generic.zip
```

```
root@siftworkstation:/cases/exercise14/linux# ls
3.11.0-26-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
root@siftworkstation:/cases/exercise14/linux# rekall convert_profile 3.11.0-26-gener
ic.zip Ubuntu_3.11.0-26-generic.zip
root@siftworkstation:/cases/exercise14/linux# ls
3.11.0-26-generic.zip  module.ko       Module.symvers  pmem.o
ko_patcher.py          module.mod.c    pmem.c          README
Makefile               module.mod.o    pmem.ko         Ubuntu_3.11.0-26-generic.zip
module.c               module.o        pmem.mod.c
module_dwarf.ko        modules.order   pmem.mod.o
```

Sixth Step: Unload pmem Module

```
# rmmod pmem.ko
```

Analyzing a Linux Memory Image with Rekall Memory Forensics Framework

The Rekall Memory Forensics Framework has plugins specifically for Linux systems that target process enumeration, listing open files and network connectivity. In order to get a list of analysis modules that you can use in Rekall, type `rekal -h`. See the expected output below and note the version of Rekall we will be using.

```
# rekal -h
```

```
root@siftworkstation:/usr/local/src# rekal -h
usage: rekal [--autodetect_threshold AUTODETECT_THRESHOLD]
             [--buffer_size BUFFER_SIZE] [--debug]
             [--ipython_engine IPYTHON_ENGINE]
             [--logging {debug,info,warning,critical,error}] [--no_autodetect]
             [--output OUTPUT] [--plugin PLUGIN [PLUGIN ...]]
             [--profile_path PROFILE_PATH] [-f FILENAME] [-h] [-r RUN] [-v]
             [--dtb DTB] [-p PROFILE] [--nocolors] [--pager PAGER]
             [--paging_limit PAGING_LIMIT] [--renderer RENDERER]
             [--timezone TIMEZONE] [--ept EPT]
             Plugin ...

--------------------------------------------------------------------------------
The Rekall Memory Forensic framework 1.0rc11.

"We can remember it for you wholesale!"
```

Type the following to enter an interactive framework within Rekall using your target Linux memory image as the current context:

```
# rekal --profile Ubuntu3.11.0-26-generic.zip -f /cases/linux.raw
```

A. Process Enumeration (pslist)

```
--------------------------------------------------------------------------------
ubuntu.dd 07:59:10> pslist
------------------> pslist()
  Offset (V)   Name                    PID   PPID    UID     GID      DTB
-------------- -------------------- ------ ------ ------ ------ --------------
0x88003dbd0000 init                      1      0      0      0 0x000036484000
0x88003dbd1770 kthreadd                  2      0      0      0 --------------
0x88003dbd2ee0 ksoftirqd/0               3      2      0      0 --------------
0x88003dbd5dc0 kworker/0:0H              5      2      0      0 --------------
0x88003d409770 migration/0               7      2      0      0 --------------
0x88003d40aee0 rcu_bh                    8      2      0      0 --------------
0x88003d40c650 rcuob/0                   9      2      0      0 --------------
0x88003d40ddc0 rcuob/1                  10      2      0      0 --------------
0x88003d418000 rcuob/2                  11      2      0      0 --------------
0x88003d419770 rcuob/3                  12      2      0      0 --------------
```

B. Open Files List (lsof)

```
- 06:52:52> lsof
----------> lsof()
       Name             Pid      User      FD        Size       Offset      Node   Path
------------------- -------- -------- -------- ------------ ------------ -------- ----
init                1               0        0            0            0     5862 /dev/null
init                1               0        1            0            0     5862 /dev/null
init                1               0        2            0            0     5862 /dev/null
init                1               0        3            0            0     8516 pipe:[8516]
init                1               0        4            0            0     8516 pipe:[8516]
init                1               0        5            0            0     5825 anon_inode:in
otify
init                1               0        6            0            0     5825 anon_inode:in
otify
```

C. Network Configuration (ifconfig)

```
ubuntu.dd 08:05:10> ifconfig
-------------------> ifconfig()
Interface         Ipv4Address           MAC                 Flags
---------------- -------------------- ------------------ --------------------
lo                127.0.0.1             00:00:00:00:00:00  IFF_LOOPBACK, IFF_UP
eth0              172.16.158.149        00:0C:29:BE:F1:6A  IFF_BROADCAST, IFF_MULT
ICAST, IFF_UP
```

Exercise: Key Takeaways

1. With the Rekall Memory Forensic Framework's Pmem Memory Acquisition tool and the imaging tool "dd", we can create a Linux kernel-specific loadable module and create a memory image of a target Linux system.
2. Rekall offers many of the Linux memory parsing plugins that have the same functionality as those for Windows systems, and in some cases, uses the same syntax, as well as platform-specific plugins unique to Linux.

This page intentionally left blank.

FOR526 Memory Forensics Tournament

Objectives

- Provide participants hands-on unguided challenges in analyzing sample memory images.
- Unlike the labs throughout the course, these do not have explicit instructions and require participants to apply analysis skills you have learned throughout the course.

Exercise Preparation

For the tournament, you will need both the SIFT 3.0 Ubuntu and Windows 8.1 VMs we have used throughout the course. **You will be provided an IP address for your host machine, but the IP configurations of your VMs do not need to be changed for the tournament.**

```
$ cd /cases
$ cp -r /media/FOR526/netwars  /cases
$ cd netwars
```

Tournament Rules

The Memory Forensics Netwars Tournament is three levels of multiple "missions" per level. Each mission focuses on specific memory images provided to you on the FOR526 USB in the "netwars" directory.

New levels will be unlocked when sufficient points have been acquired. Completion of all questions on a level is **NOT** required in order to move to the next. Participants will continue solving missions until time expires or they have exhausted the supply of missions.

Incorrect Answers
One free wrong answer is allowed per question. After that, you lose one point per wrong answer attempted.

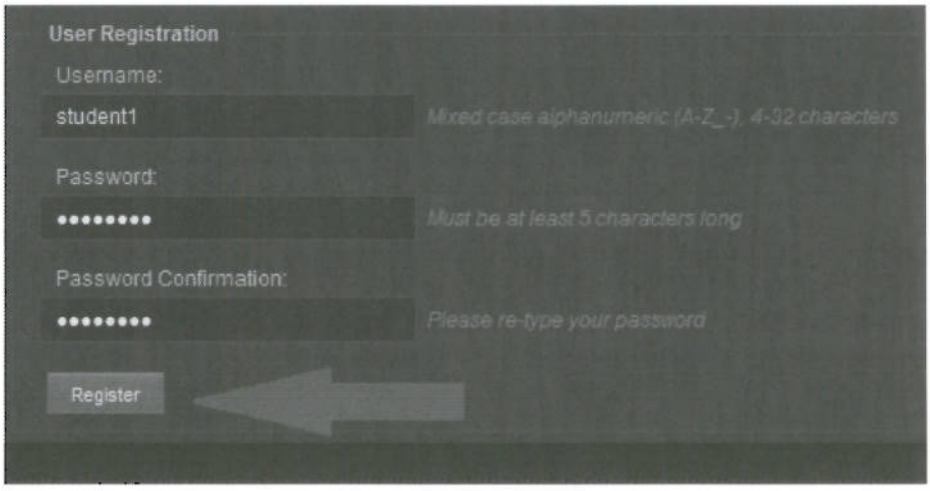
Hints
Hints can be used to solve missions. If you use one of the hints, 50% of the value for that question will be deducted from your score. The second hint, if available, 50% of the remaining points for that question will be deducted from your score.

Feedback
If you feel that a question is incorrect, consult the instructor. We welcome feedback on the questions, hints and answers. Although we will not make changes to the game in progress, we will incorporate feedback into future iterations of the game at other events.

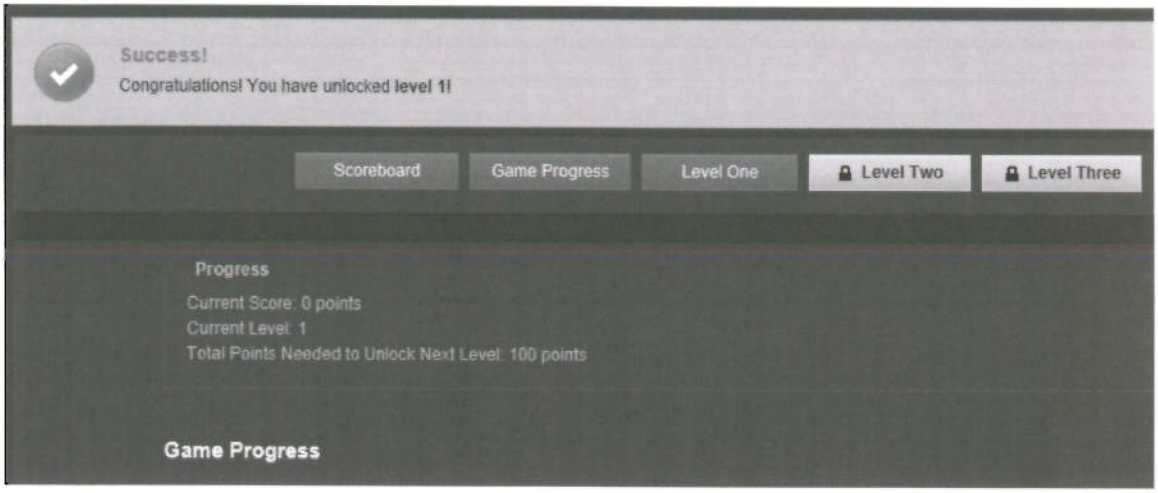
Ground Rules
Don't attack student machines. Don't attack the scoring server. DoS attacks are absolutely off-limits during this tournament. Failure to comply will result in dismissal from the class.

Account Registration

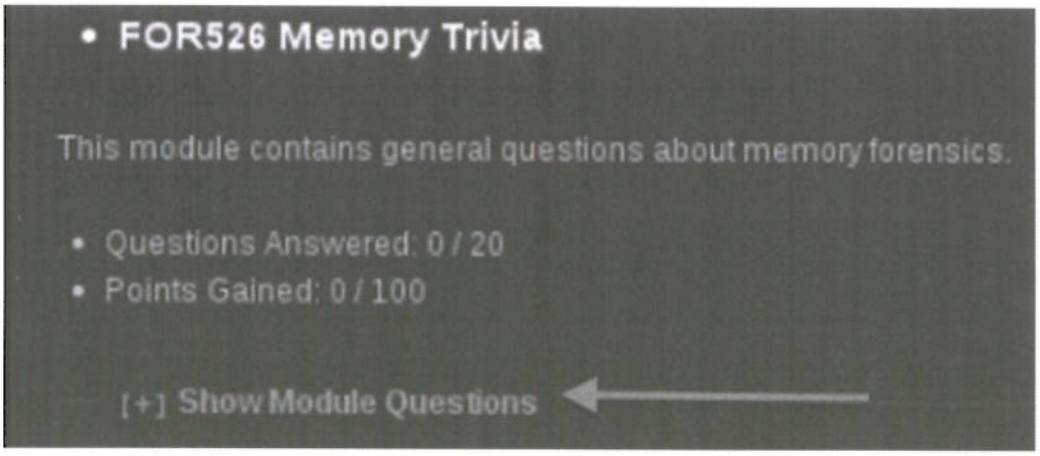
- Open a browser and navigate to the scoring server at the address below:
  - **https://10.10.10.20**
- Create an account
  - No inappropriate usernames please
- Please register for an account, but note that you will not be able to log in until the instructor starts the tournament.



- 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.
- The 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.



- Any Final Questions?
  - Barring any questions, the instructor will now start the tournament
  - The tournament server will run for at least six hours
  - Lunch and breaks on your own
  - Winners get lethal forensicator coins

- **Game On! Enjoy the Tournament!**

This page intentionally left blank.

Challenge Exercise: APT Compromised Host

Objectives

- Conduct a forensic memory analysis applying concepts acquired throughout the FOR526 course
- Examine a system, applying the 6 step investigative methodology without the aid of guided questions
- Gain experience using Volatility, Bulk_Extractor, windbg, Redline and all tools contained in the memory forensics weapons arsenal.

Exercise Preparation

Start a terminal and change into the /cases directory. Unzip the **APT.zip** file.

```
$ cd /cases
$ unzip APT.zip
```

APT Exercise – Questions

Not much about this memory image is known at this point. However, this image is contains malware from an actual APT case found in the wild. Your goal is to investigate the memory image and detect the malicious code and some of its unique behaviors. This is a very realistic scenario, as when you are working a real case, you will not have questions guiding your analysis back in your work environment.

Accept the challenge by utilizing the step by step investigative methodology taught to you in FOR526 to detect the malware and signature some of its behaviors for further detection through your enterprise.

Your instructor will hand out and review the solution once you discover the malware found in this image. There are various techniques that can used to find the malware. Record your methods as you work through this analysis and we will discuss during the review.