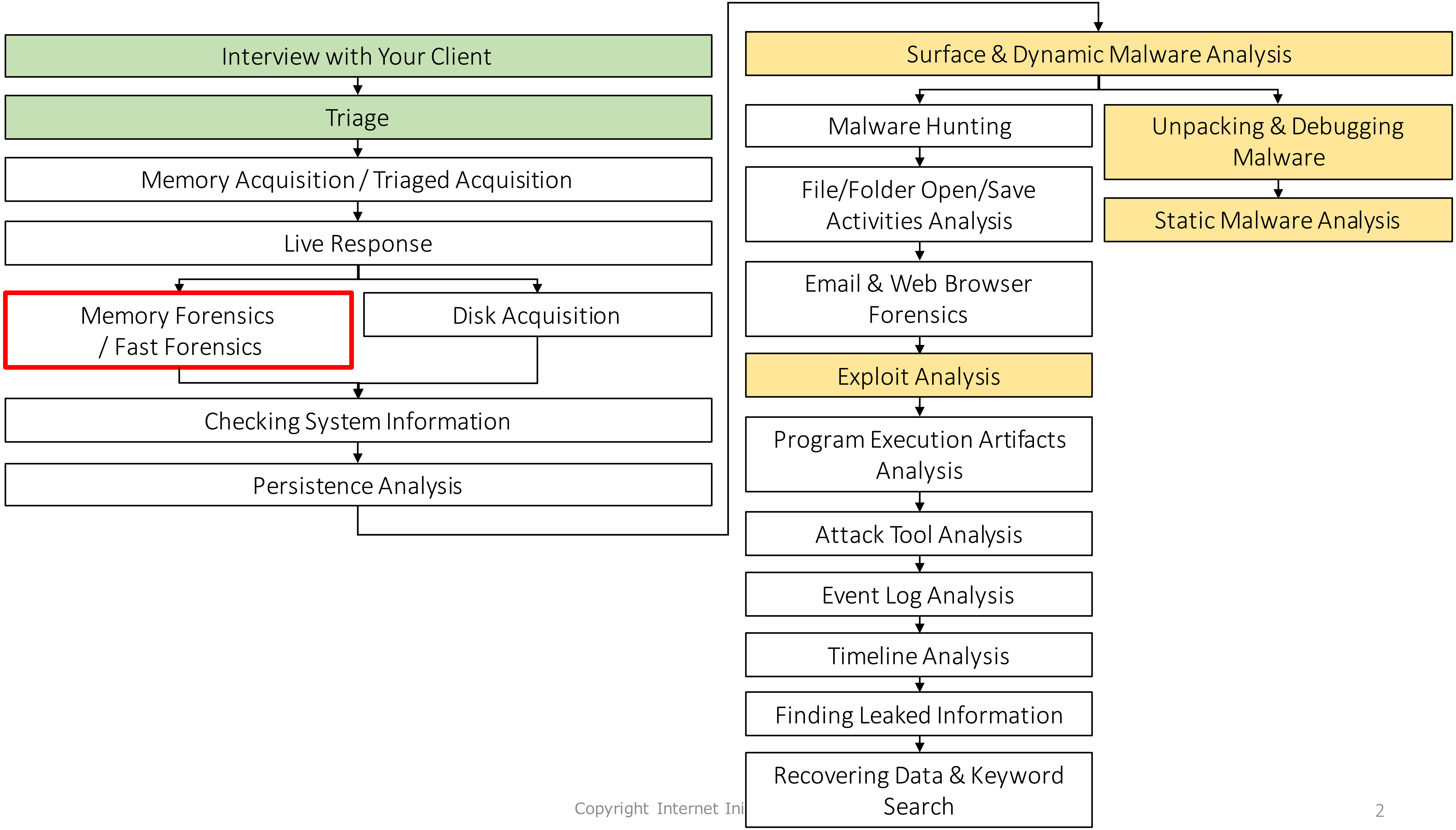
Memory Forensics


Interview with Your Client
Triage
Memory Acquisition / Triaged Acquisition
Live Response
Memory Forensics / Fast Forensics
Disk Acquisition
Checking System Information
Persistence Analysis
Surface & Dynamic Malware Analysis
Malware Hunting
Unpacking & Debugging Malware
Static Malware Analysis
File/Folder Open/Save Activities Analysis
Email & Web Browser Forensics
Exploit Analysis
Program Execution Artifacts Analysis
Attack Tool Analysis
Event Log Analysis
Timeline Analysis
Finding Leaked Information
Recovering Data & Keyword Search

Memory Forensics 101 (1)

- What is Memory Forensics?
  - It is a method to analyze memory images of infected computers.

- Why Memory Forensics?
  - It is important for analysts to investigate rapidly which process communicated with malicious hosts, and what kind of programs were executed on the infected computers.
  - The amount of memory image size is smaller than the disk. Therefore, it takes less time to acquire and analyze images. We can quickly start to investigate it while acquiring disk images.

Memory Forensics 101 (2)

- What is the difference between memory forensics and live response?
  - Initial investigation of memory forensics is similar to live response.
  - In addition, we can access code and data without running OSes. Therefore, we are unlikely to be affected by rootkits and anti-analysis techniques.
- What is the difference between this analysis and disk forensics?
  - Since malware decrypts or de-obfuscates its data and its configuration while running on memory, unlike packed files on a disk, we may be able to access the plain data. This is one of the advantages over disk forensics.

Memory Forensics Tools (1)

- Memory Image Acquisition Tools
  - WinPmem
  - DumpIt
- Triaged Acquisition (Fast Forensic) Tool
  - KAPE
  - CDIR Collector
- Memory Image Analysis Tools
  - Volatility Framework
  - Rekall
  - Redline
  - WinDbg
- We are going to focus on Volatility Framework.

Memory Forensics Tools (2)

- Volatility Framework
  - It is one of the open source memory forensics tools.
  - It can run on multiple platforms and it can analyze these OSes.
    - Windows
    - Linux
    - macOS
  - It has a plugin architecture.
    - You can extend its feature easily.
    - Many security researchers and security companies published various plugins.

Memory Forensics Tools (3)

- WinDbg
  - It is a debugger for Windows OSes and Windows applications.
  - It is maintained by Microsoft.
  - It can handle Microsoft crash dump format memory images.
    - On the other hand, it cannot manage raw and aff4.
  - You can use an extension called "pykd“ to manage WinDbg from python.
    - It is a third party tool.

Volatility Plugins (1) - Processes

Plugin name	Short Description	URL
pslist	This is a module for listing processes by referring the links of EPROCESS structures.	(If this column is blank, the plugin is included by default.)
pstree	This is similar to pslist. In addition, it creates a process tree.	
psscan	This is similar to pslist. It can also list terminated processes because it searches EPROCESS structures in memory image.	
psxview	This creates a set of lists of processes with modules such as pslist and psscan at the same time to find hidden processes by rootkits.	
ldrmodules	This is a module for detecting unlinked DLLs on PEB.	

Volatility Plugins (2) - Privileges

Plugin name	Short Description	URL
privs	This is a module for displaying process privileges	
getsids	This is a module for enumerating SIDs on each process	

Volatility Plugins (3) - Network Activities

Plugin name	Short Description	URL
netscan	This is a module for scanning network connections and sockets for Vista or later	
sockscan	This is a pool chunks scanner for tcp socket objects	
sockets	This is a module to create a list of opened sockets like “netstat -na” command	

Volatility Plugins (4) - Malware Scanning

Plugin name	Short Description	URL
malfind	This is a module for finding hidden and injected code	
hollowfind	This is a module to find a code injection technique called process hollowing	https://github.com/monnappa22/HollowFind
yarascan	This is a module for scanning user mode processes or kernel space with Yara signatures	
autoruns	This is a module for scanning and parsing auto-start locations	https://github.com/tomchop/volatility-autoruns

Volatility Plugins (5) - OS Artifacts

Plugin name	Short Description	URL
mftparser	This is a module for scanning and parsing potential MFT records	
prefetchparser	This is a module for scanning and parsing Windows prefetch files	https://github.com/superponible/volatility-plugins
shimcachemem	This is a module for scanning and parsing shimcache entries	https://github.com/fireeye/Volatility-Plugins/tree/master/shimcachemem
autoruns	This is a module for scanning and parsing auto-start locations	https://github.com/tomchop/volatility-autoruns
handles	This is a module for dumping handles such as files and registries used by processes and the kernel	

Volatility Plugins (6) - Kernel

Plugin name	Short Description	URL
ssdt	This is a module for displaying SSDT entries.	
driverirp	This is a module for detecting IRP hook.	
idt	This is a module for dumping IDT entries.	

Volatility Plugins (7) - Detecting and Finding Hooks

Plugin name	Short Description	URL
messagehooks	This is a module for listing desktop and thread window message hooks.	
apihooks	This is a module for detecting API hooks in process and kernel memories.	
ssdt	This is a module for displaying SSDT entries.	
driverirp	This is a module for detecting IRP hook.	
idt	This is a module for dumping IDT entries.	

Volatility Plugins (8) - Command line info.

Plugin name	Short Description	URL
consoles	This is a module for extracting command history by scanning for _CONSOLE_INFORMATION.	
cmdline	This is a module for displaying process command-line with its arguments.	

Volatility Plugins (9) - Registry

Plugin name	Short Description	URL
hivelist	This is a module for displaying a list of registry hives.	
hivedump	This is a module for printing out a hive.	
dumpregistry	This is a module for dumping registry files out to disk.	

Volatility Plugins (10) - Web Browser History

Plugin name	Short Description	URL
iehistory	This is a module for reconstructing Internet Explorer cache / history.	
firefoxhistory	This is a module for analyzing Firefox history.	https://github.com/superponible/volatility-plugins
chromehistory	This is a module for investigating Chrome history.	https://github.com/superponible/volatility-plugins

Volatility Plugins (11) - Memory Format Conversion

Plugin name	Short Description	URL
raw2dmp	This is a module for converting a raw image to a Microsoft crash dump format.	
imagecopy	This is a module for converting a hibernation file to a raw image.	

Volatility Plugins (12) - Timeline

Plugin name	Short Description	URL
timeliner	This is a module to create a timeline from processes, threads, TCP/IP socket information, and other data, on memory.	
AutoTimeliner	This is a wrapper script to Execute timeliner, mftparser and shellbags plugins and merge the outputs.	https://github.com/andreafortuna/autotimeliner

The First Step of Volatility Framework

Volatility Framework: The First Step (1)

- To use volatility, we need to know some things.
- Run Volatility Framework to check some information.
  1. Launch command prompt and move to the volatility directory

```
cd volatility
```

  2. Run "vol.py --info"

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

Volatility Framework: The First Step (2)

- When you run Volatility, you need to specify a profile that corresponds to the OS version of the infected PC.
- On Windows 10, you should specify a profile corresponding to the OS build number.
  - For example, if the user used Windows 10 64 bit 1607, you would need to choose specify "Win10x64_14393".
  - You can refer to the URL below to check the recent OS versions and the OS build numbers.
    - https://technet.microsoft.com/en-us/windows/release-info.aspx

Version	OS build
1709	16299.251
1703	15063.936
1607	14393.2097
1511	10586.1417

Volatility Framework: The First Step (3)

C:\Windows\system32\cmd.exe

```
VistaSP0x86           - A Profile for Windows Vista SP0 x86
VistaSP1x64           - A Profile for Windows Vista SP1 x64
VistaSP1x86           - A Profile for Windows Vista SP1 x86
VistaSP2x64           - A Profile for Windows Vista SP2 x64
VistaSP2x86           - A Profile for Windows Vista SP2 x86
Win10x64              - A Profile for Windows 10 x64
Win10x64_10586        - A Profile for Windows 10 x64 (10.0.10586.306 / 2016-04-23)
Win10x64_14393        - A Profile for Windows 10 x64 (10.0.14393.0 / 2016-07-16)
Win10x64_15063        - A Profile for Windows 10 x64 (10.0.15063.0 / 2017-04-04)
Win10x64_16299        - A Profile for Windows 10 x64 (10.0.16299.0 / 2017-09-22)
Win10x86              - A Profile for Windows 10 x86
Win10x86_10586        - A Profile for Windows 10 x86 (10.0.10586.420 / 2016-05-28)
Win10x86_14393        - A Profile for Windows 10 x86 (10.0.14393.0 / 2016-07-16)
Win10x86_15063        - A Profile for Windows 10 x86 (10.0.15063.0 / 2017-04-04)
Win10x86_16299        - A Profile for Windows 10 x86 (10.0.16299.15 / 2017-09-29)
Win2003SP0x86         - A Profile for Windows 2003 SP0 x86
Win2003SP1x64         - A Profile for Windows 2003 SP1 x64
Win2003SP1x86         - A Profile for Windows 2003 SP1 x86
Win2003SP2x64         - A Profile for Windows 2003 SP2 x64
Win2003SP2x86         - A Profile for Windows 2003 SP2 x86
Win2008R2SP0x64       - A Profile for Windows 2008 R2 SP0 x64
Win2008R2SP1x64       - A Profile for Windows 2008 R2 SP1 x64
Win2008R2SP1x64_23418 - A Profile for Windows 2008 R2 SP1 x64 (6.1.7601.23418 / 2016-04-09)
Win2008SP1x64         - A Profile for Windows 2008 SP1 x64
Win2008SP1x86         - A Profile for Windows 2008 SP1 x86
Win2008SP2x64         - A Profile for Windows 2008 SP2 x64
Win2008SP2x86         - A Profile for Windows 2008 SP2 x86
Win2012R2x64          - A Profile for Windows Server 2012 R2 x64
Win2012R2x64_18340    - A Profile for Windows Server 2012 R2 x64 (6.3.9600.18340 / 2016-05-13)
Win2012x64            - A Profile for Windows Server 2012 x64
```

Specify this profile for example

Volatility Framework: The First Step (4)

This is just an example. You don't need to execute it.

- Volatility command line format

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw pslist --pid=4
```

Profile name

Path to memory image file

Volatility plugin and its option

Extra Exercise 1
Parsing Malware Configuration

The Strategy (1)

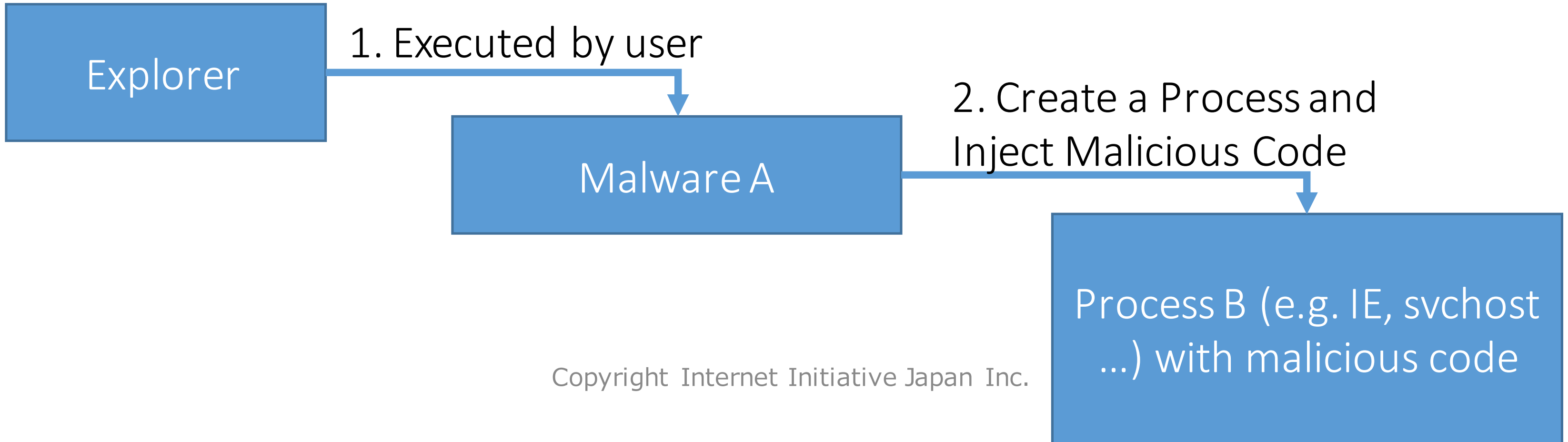
- The information we need to know
  - We need a kernel version and a CPU architecture of the infected machine to use the framework. You need to get them by doing one of the following methods such as:
    - Interviewing with the customer
    - Performing live response
    - Working on offline registry analysis with fast forensics
      - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion
        - ProductName , CSDVersion, CurrentBuildNumber
      - HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment
        - PROCESSOR_ARCHITECTURE
  - In this exercise, let's assume that we already know that Windows 7 SP1 64 bit was running.

The Strategy (2)

- The strategy of memory forensics
  - As we mentioned before, we want to perform persistence analysis with the top priority if we are investigating disk images. However, autoruns module of Volatility Framework takes very long time due to its structure. Therefore, we will check running processes first, like live response. Then, we will check network activities.

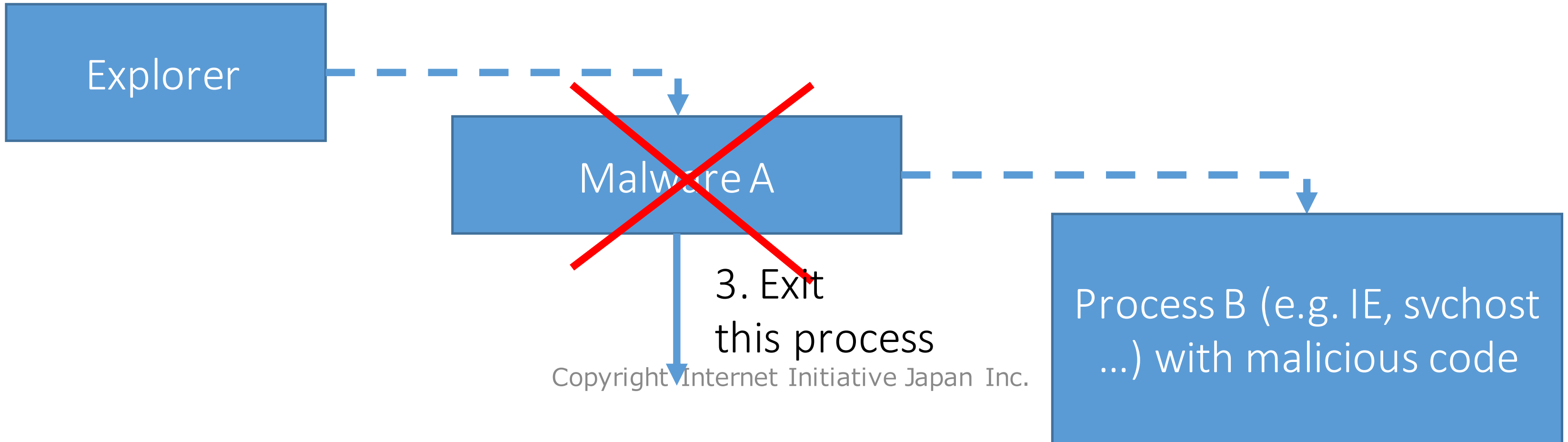
The Strategy (3)

- Finding out orphan processes is important.
- Why?
  1. When a user double-clicks an executable file, the process is created as a child process of explorer.exe.
  2. If the created process (Malware A) uses a remote code injection technique such as Process Hollowing or Reflective PE Injection, the malware creates a child process (Process B) and injects malicious code into the process.



The Strategy (4)

3. The process (Malware A) exits immediately after injecting code into the remote process (Process B),



The Strategy (5)

- Therefore, the link between Explorer and Process B is broken and the process that malicious code has been injected becomes an orphan.

The Strategy (6)

- We can use the following Volatility plugins to list process information (name, process id, parent pid, etc).
  - pslist
    - This plugin lists process information by following a link list in EPROCESS structures. It is similar to "tasklist" command.
  - pstree
    - This plugin displays a process tree with the same method as the pslist module.
  - psscan
    - This plugin is similar to pslist. Furthermore, it can search unlinked EPROCESS structures and shows us the processes information.

Extra Exercise 1: Parsing Malware Configuration (1)

- Memory Image
  - E:\Artifacts\other_memory_image\memory_image1.raw
- Volatility profile
  - Win7SP1x64
- The result files of many plugins
  - E:\Artifacts\other_memory_forensics\extra_exercise1\

- First, let’s investigate orphan processes and network connections.

Extra Exercise 1: Parsing Malware Configuration (2)

- Let’s confirm the image with pstree plugin.

```
cd volatility
vol.py --profile=Win7SP1x64 -f E:\Artifacts\other_memory_image\memory_image1.raw pstree
```

  - E:\Artifacts\other_memory_forensics\extra_exercise1\pstree.txt

- The result of pstree

```
 0xfffffa800d07a540:wmplayer.exe                     2476   2604     27    312 2018-03-04 06:41:30 UTC+0000
. 0xfffffa800d0d21d0:rundll32.exe                    2824   2476      6     88 2018-03-04 06:42:18 UTC+0000
```

- wmplayer.exe is an orphan process. It has a child process, rundll32. And the PIDs are 2476 and 2824.

Extra Exercise 1: Parsing Malware Configuration (3)

- Next, take a look at the command line of "wmplayer.exe".
- Run cmdline plugin.

```
vol.py --profile=Win7SP1x64 -f E:\Artifacts\other_memory_image\memory_image1.raw cmdline --pid=2476,2824
```

- The result of cmdline

```
wmplayer.exe pid:   2476
Command line : "C:\Program Files (x86)\Windows Media Player\wmplayer.exe"
```

  - wmplayer.exe is located in "C:\Program Files (x86)\Windows Media Player".
  - It looks a legitimate program file because it is in the legitimate location.

Extra Exercise 1: Parsing Malware Configuration (4)

- Next, let's check malfind plugins.

```
vol.py --profile=Win7SP1x64 -f E:\Artifacts\other_memory_image\memory_image1.raw malfind --pid=2476,2824
```

  - E:\Artifacts\other_memory_forensics\extra_exercise1\malfind.txt

- This plugin can enumerate "RWX" sections on memory.
- It helps us to find suspicious processes.

Extra Exercise 1: Parsing Malware Configuration (5)

- The Result of malfind
  - You can get two results.
  - The right figure seems to be shellcode because of call instruction (e8).

```
Process: wmplayer.exe Pid: 2476 Address: 0x200000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 47, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x00200000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00200010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00200020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00200030  00 00 00 00 00 00 00 00 00 00 00 00

0x00200000 0000             ADD [EAX], AL
0x00200002 0000             ADD [EAX], AL
0x00200004 0000             ADD [EAX], AL
```

```
Process: wmplayer.exe Pid: 2476 Address: 0xd0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 29, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x000d0000  e8 00 00 00 00 58 83 e8 05 8b 4c 24 04 51 68 00   .....X....L$.Qh.
0x000d0010  08 00 00 8d 88 4c b9 01 00 51 68 2d b4 01 00 8d   .....L...Qh-....
0x000d0020  88 1f 05 00 00 51 68 4c c1 01 00 8d 88 00 00 00   .....QhL........
0x000d0030  00 51 54 e8 06 00 00 00 83 c4 1c c2 04 00 55 8b   .QT...........U.

0x000d0000 e800000000       CALL 0xd0005
0x000d0005 58               POP EAX
0x000d0006 83e805           SUB EAX, 0x5
```

We also tried "hollowfind" plugin, but it could not detect them at this time.

Extra Exercise 1: Parsing Malware Configuration (6)

```
Process: wmplayer.exe Pid: 2476 Address: 0xd0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 29, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x000d0000  e8 00 00 00 00 58 83 e8 05 8b 4c 24 04 51 68 00   .....X....L$.Qh.
0x000d0010  08 00 00 8d 88 4c b9 01 00 51 68 2d b4 01 00 8d   .....L...Qh-....
0x000d0020  88 1f 05 00 00 51 68 4c c1 01 00 8d 88 00 00 00   .....QhL........
0x000d0030  00 51 54 e8 06 00 00 00 83 c4 1c c2 04 00 55 8b   .QT...........U.

0x000d0000 e800000000       CALL 0xd0005
0x000d0005 58               POP EAX
0x000d0006 83e805           SUB EAX, 0x5
0x000d0009 8b4c2404         MOV ECX, [ESP+0x4]
0x000d000d 51               PUSH ECX
0x000d000e 6800080000       PUSH DWORD 0x800
0x000d0013 8d884cb90100     LEA ECX, [EAX+0x1b94c]
0x000d0019 51               PUSH ECX
0x000d001a 682db40100       PUSH DWORD 0x1b42d
0x000d001f 8d881f050000     LEA ECX, [EAX+0x51f]
```

This means that the code gets the memory address where it is executing at the moment. It is called "GetPC (Get Program Counter)" code. It is a typical technique of shellcode.

Extra Exercise 1: Parsing Malware Configuration (7)

- We have confirmed that:
  - wmplayer.exe is an orphan process.
  - Although it seems to be a legitimate file because wmplayer.exe is located in "C:\Program Files (x86)\Windows Media Player\wmplayer.exe“, it seems to have shellcode injected.
- Next, let’s check network activities.

Extra Exercise 1: Parsing Malware Configuration (8)

- Let's run netscan plugin.

```
vol.py --profile=Win7SP1x64 -f E:\Artifacts\other_memory_image\memory_image1.raw netscan
```

  - E:\Artifacts\other_memory_forensics\extra_exercise1\netscan.txt

- The result of netscan

Listening on 1357/tcp and opening 1357/udp.

```
0x3dac8010    TCPv4    0.0.0.0:1357            0.0.0.0:0             LISTENING    2476    wmplayer.exe
0x3f73bec0    UDPv4    0.0.0.0:1357            *:*                                2476    wmplayer.exe
 2018-03-04 06:41:43 UTC+0000
0x3d9ea2a0    TCPv4    192.168.153.129:49186   192.168.153.20:1357   CLOSED       2476    wmplayer.exe
0x3f753750    TCPv4    192.168.153.129:49183   192.168.153.18:1357   CLOSED       2476    wmplayer.exe
```

wmplayer.exe tried to connect to neighbor hosts.

Extra Exercise 1: Parsing Malware Configuration (9)

- We were able to find out that wmplayer.exe tried to connect to the following neighbor hosts.
  - Host 1: 192.168.153.20   Port: 1357/tcp
  - Host 2: 192.168.153.18   Port: 1357/tcp
  - wmplayer.exe was also listening on 1357/tcp and opening several udp ports (1357, 54142, 60970, 60974, 65461, 65460).
  - The system administrator told us that these IP addresses does not exist and he did not know about these port numbers.
- We have several questions.
  - Why does wmplayer.exe try to connect to neighbor hosts?
  - What service is provided on 1357/tcp?
  - Why was wmplayer.exe listening on 1357/tcp and opening some udp ports?

Extra Exercise 1: Parsing Malware Configuration (10)

- Let's google about 1357/tcp.
  - Keyword: “1357/tcp malware”
  - You can find JPCERT/CC's blog about PlugX.
    - Analysis of a Recent PlugX Variant - “P2P PlugX”
      - http://blog.jpcert.or.jp/2015/01/analysis-of-a-r-ff05.html
  - According to the article:

> *Note that this P2P communication theoretically can be applied to any other TCP/UDP ports. But in cases which JPCERT/CC has observed, P2P PlugX only uses either TCP/1357 or UDP/1357 for P2P communication.* ***If you see any scanning activity to TCP/1357 or UDP/1357, we highly recommend that you conduct further investigation.***

- It is possible that PlugX had been abusing wmplayer.exe.

Extra Exercise 1: Parsing Malware Configuration (11)

- Let’s confirm whether the process contains code or configurations of PlugX.
- You can use a third party Volatility plugin for scanning and parsing PlugX configuration.
  - You can get the plugin from the following URL:
    - MalConfScan
      - https://github.com/JPCERTCC/MalConfScan
  - In this exercise, the plugin is already installed on your VM.

Extra Exercise 1: Parsi
Configuration (12)

- Let's run malconfscan plugin.

```
vol.py --profile=Win7SP1x64 -f
E:\Artifacts\other_memory_image\memory_im
age1.raw malconfscan --pid=2476
```

This command is in a single line. Don't create a new line.

- E:\Artifacts\other_memory_forensics\extra_exercise1\malconfscan.txt

```
[+] Searching memory by Yara rules.
[+] Detect malware by Yara rules.
[+]   Process Name        : wmplayer.exe
[+]   Process ID          : 2476
[+]   Malware name        : plugx
(snip)
Server  1               : 231p.uk.to:80 (Type 3)
Server  2               : 231p.uk.to:443 (Type 4)
Server  3               : dns34.uk.to:53 (Type 5)
(snip)
Auto Start              : Run Registry or Service setup
Install Folder          : %AUTO%\NVK
Service Name            : NVK
Service Display Name    : NVK
(snip)
Injection               : Enable
(snip)
Injection Process4      : %ProgramFiles%\Windows Media
Player\wmplayer.exe
UACBypass               : Enable
(snip)
UACBypass Process4      : %windir%\system32\rundll32.exe
(snip)
IP Scan                 : Enable
IP Scan value 1         : 1
IP Scan port 1          : 1357
IP Scan value 2         : 1
IP Scan port 2          : 1357
```

C2 Servers

Settings for persistence.
They have strings related to "NVK"

P2P Ports

wmplayer.exe and rundll32.exe were found.

Extra Exercise 1: Parsing Malware Configuration (13)

- Next, let's investigate the persistence.
  - This sample should use Windows service or "Run" key of HKCU, which has strings related to "NVK".
- Let's confirm the image with autoruns plugin. IT TAKES A LONG TIME, so see the result file

```
vol.py --profile=Win7SP1x64 -f E:\Artifacts\other_memory_image\memory_image1.raw autoruns
```

  - E:\Artifacts\other_memory_forensics\extra_exercise1\autoruns.txt
- The result of autoruns

```
Service: NVK (NVK ) - Interactive, Auto Start
    Image path: C:\ProgramData\NVK\dot1x.exe  (Last modified: 2018-03-04 06:41:29 UTC+0000)
    PIDs: -
```

  - Retrieve this file from the disk image and perform static/log analysis later.

Suspicious program file is located in "C:\ProgramData" folder

Extra Exercise 1: Parsing Malware Configuration (14)

- Next, let's check the process rights.
- Let's confirm the image with sids plugin.

```
vol.py --profile=Win7SP1x64 -f E:\Artifacts\other_memory_image\memory_image1.raw getsids --pid=2476
```

  - E:\Artifacts\other_memory_forensics\extra_exercise1\getsids.txt

- The result of getsids

```
wmplayer.exe (2476): S-1-5-18 (Local System)
wmplayer.exe (2476): S-1-5-32-544 (Administrators)
wmplayer.exe (2476): S-1-1-0 (Everyone)
wmplayer.exe (2476): S-1-5-11 (Authenticated Users)
wmplayer.exe (2476): S-1-16-16384 (System Mandatory Level)
```

  - Attacker has already have the SYSTEM privilege.

It ran with the SYSTEM privilege!

Extra Exercise 1: Parsing Malware Configuration (14) - Summary

- The wmplayer.exe process has injected malicious code.
- The malware name is PlugX.
- "C:\ProgramData\NVK\dot1x.exe" is obviously related to the malicious activity. You should analyze all files in this folder later.

Extra Exercise 1: Parsing Malware Configuration (15) - MalConfScan

MalConfScan can parse the following malware configurations.

- Ursnif
- Emotet
- Smoke Loader
- PoisonIvy
- CobaltStrike
- NetWire
- PlugX
- RedLeaves / Himawari / Lavender / Armadill / zark20rk
- TSCookie
- TSC_Loader
- xxmm
- Datper
- Ramnit
- HawkEye
- Lokibot
- Bebloh (Shiotob/URLZone)
- AZORult
- NanoCore RAT
- AgentTesla
- FormBook
- NodeRAT
- Pony
- njRAT

Wrap up
- Memory Forensics

What We Lesson Learned in Memory Forensics (1)

- An initial investigation of memory forensics is similar to live response.
    - Find suspicious processes.
    - Confirm characteristics of processes.
    - Find communications with external hosts.
- We are able to confirm an OS artifact that reside on memory.
    - Registry
- We can check API hooks, possible malicious memory sections and even disassembled code.
- We can dump arbitrary memory sections.

What We Lesson Learned in Memory Forensics (2)

- We used a variety of Volatility plugins in memory forensics exercise.
- Finding a suspicious process
  - pstree
  - cmdline
  - netscan
  - getsids
- Confirming persistence
  - autoruns
- Confirming code injection
  - malfind
- Extracting malware configuration
  - malconfscan

Tips

- Since the execution speed of each plugin is different, you should execute them from the one that is relatively effective and one that ends sooner.
- Memory analysis tools would take a long time if you execute a lot of plugins. We recommend you to prepare a batch file or a shell script in advance to execute them in order that you want to use.
- You should use DumpIt to acquire memory because WinDbg sometimes cannot analyze crash dump images converted from other formats properly.
  - We have confirmed the case that at least a memory image acquired with WinPmem was not analyzed properly.
- We recommend you to prepare plugins for detecting malware that you have analyzed in the past so that you can find it out if a similar incident occurs.

Tips (Cont.)

- You can convert aff4 to raw image with winpmem.

```
winpmem-2.1.post4.exe --export PhysicalMemory -o file:///X:/path_to _image.raw X:/path_to_image.aff4
```

  - Note: The memory image must be acquired by winpmem 2.X.

```
winpmem_v3.3.rc1.exe -e */PhysicalMemory  path_to_image.aff4
```

  - Note: The memory image must be acquired by winpmem 3.X.

- When you analyze Windows 10 memory images, you sometimes lose some results because of memory compression feature. In that case, this improved volatility framework and rekall developed by FireEye may solve the problem.
  - https://www.fireeye.com/blog/threat-research/2019/07/finding-evil-in-windows-ten-compressed-memory-part-one.html

Appendix 15: Memory Forensics

Memory Forensics Exercise - Appendix 15-1

```
import pykd
import re

pattern = re.compile("\w{8}\s+\w+\s+(\w+)\s+(.+)")
for api in pykd.dbgCommand("x nt!*").split("\n"):
    if api != "":
        disasm_lines = ""
        ar_api = api.split()
        addr = ar_api[0]
        name = ar_api[1]
        try:
            disasm_lines = pykd.dbgCommand("u %(addr)s" % {'addr': addr})
        except pykd.DbgException:
            continue
        if disasm_lines:
            disasm_lines = disasm_lines.split('\n')
        else:
            continue
        for disasm_line in disasm_lines:
            r = pattern.search(disasm_line)
            if r:
                opcode = r.group(1)
                operand = r.group(2)
                if opcode == "int" and operand == "0C3h":
                    print("IDT Hook: {0} {1}".format(addr, name))
                    break
```

Enumerating API names and its addresses in the kernel

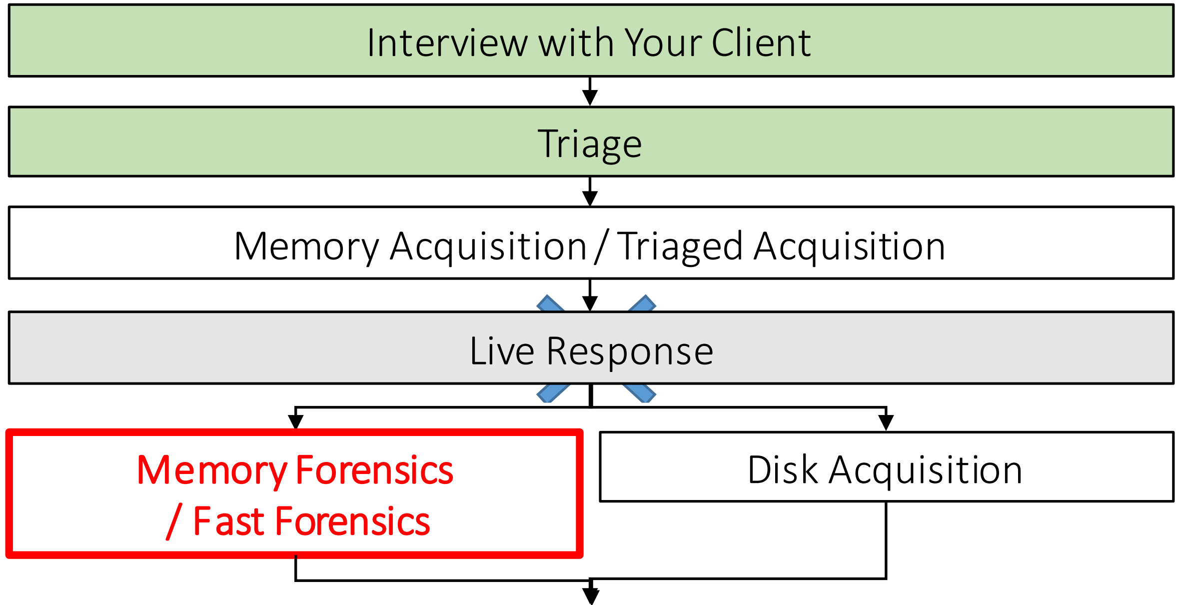
Getting disassembled code on the address

Checking the bytes whether including "INT C3"

Appendix 15-2:
Scenario 1 Labs:
Memory Forensics Lab 1

Scenario 1 Labs: Memory Forensics Lab 1 (1)

- There were no analysis tasks up until this section except for live response. Now, we will start analysis of memory spaces.
- Let’s assume that:
  - We did not perform live response.
  - We do not know anything about this incident yet.



Scenario 1 Labs: Memory Forensics Lab 1 (2)

- The information we need to know
  - We need a kernel version and a CPU architecture of the infected machine to use the framework. You need to get them by doing one of the following methods such as:
    - Interviewing with the customer
    - Performing live response
    - Working on offline registry analysis with fast forensics
  - In this exercise, let's assume that we already know that Windows 10 1607 64 bit was running on Client-Win10-1.
- The strategy for memory forensics
  - As we mentioned in day 1, we want to perform persistence analysis with the top priority on disk forensics. However, autoruns module of Volatility Framework takes very long time due to its structure. Therefore, we will check running processes first, like live response. Then, we will check network activities.

Scenario 1 Labs: Memory Forensics Lab 1 (3)

- Run Volatility Framework
  1. Launch command prompt and move to the volatility directory

```
cd volatility
```

  2. Run "vol.py --info"

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

Scenario 1 Labs: Memory Forensics Lab 1 (4)

- When you run Volatility, you need to specify a profile that corresponds to the OS version of the infected PC.
- On Windows 10, you should specify a profile corresponding to an OS build number.
  - In this exercise, specify "Win10x64_14393".
  - You can refer to the URL below to check the versions and the OS build numbers.
    - https://technet.microsoft.com/en-us/windows/release-info.aspx

Version	OS build
1709	16299.251
1703	15063.936
1607	14393.2097
1511	10586.1417

Scenario 1 Labs: Memory Forensics Lab 1 (5)

C:\Windows\system32\cmd.exe

```
VistaSP0x86           - A Profile for Windows Vista SP0 x86
VistaSP1x64           - A Profile for Windows Vista SP1 x64
VistaSP1x86           - A Profile for Windows Vista SP1 x86
VistaSP2x64           - A Profile for Windows Vista SP2 x64
VistaSP2x86           - A Profile for Windows Vista SP2 x86
Win10x64              - A Profile for Windows 10 x64
Win10x64_10586        - A Profile for Windows 10 x64 (10.0.10586.306 / 2016-04-23)
Win10x64_14393        - A Profile for Windows 10 x64 (10.0.14393.0 / 2016-07-16)
Win10x64_15063        - A Profile for Windows 10 x64 (10.0.15063.0 / 2017-04-04)
Win10x64_16299        - A Profile for Windows 10 x64 (10.0.16299.0 / 2017-09-22)
Win10x86              - A Profile for Windows 10 x86
Win10x86_10586        - A Profile for Windows 10 x86 (10.0.10586.420 / 2016-05-28)
Win10x86_14393        - A Profile for Windows 10 x86 (10.0.14393.0 / 2016-07-16)
Win10x86_15063        - A Profile for Windows 10 x86 (10.0.15063.0 / 2017-04-04)
Win10x86_16299        - A Profile for Windows 10 x86 (10.0.16299.15 / 2017-09-29)
Win2003SP0x86         - A Profile for Windows 2003 SP0 x86
Win2003SP1x64         - A Profile for Windows 2003 SP1 x64
Win2003SP1x86         - A Profile for Windows 2003 SP1 x86
Win2003SP2x64         - A Profile for Windows 2003 SP2 x64
Win2003SP2x86         - A Profile for Windows 2003 SP2 x86
Win2008R2SP0x64       - A Profile for Windows 2008 R2 SP0 x64
Win2008R2SP1x64       - A Profile for Windows 2008 R2 SP1 x64
Win2008R2SP1x64_23418 - A Profile for Windows 2008 R2 SP1 x64 (6.1.7601.23418 / 2016-04-09)
Win2008SP1x64         - A Profile for Windows 2008 SP1 x64
Win2008SP1x86         - A Profile for Windows 2008 SP1 x86
Win2008SP2x64         - A Profile for Windows 2008 SP2 x64
Win2008SP2x86         - A Profile for Windows 2008 SP2 x86
Win2012R2x64          - A Profile for Windows Server 2012 R2 x64
Win2012R2x64_18340    - A Profile for Windows Server 2012 R2 x64 (6.3.9600.18340 / 2016-05-13)
Win2012x64            - A Profile for Windows Server 2012 x64
```

Specify this profile in this exercise

Scenario 1 Labs: Memory Forensics Lab 1 (6)

This is just an example. Don't execute it yet.

- Volatility command line format

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw pslist --pid=4
```

Profile name

Path to memory image file

Volatility plugin and its option

Scenario 1 Labs: Memory Forensics Lab 1 (7)

- Memory Image
  - E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw
- Volatility profile
  - Win10x64_14393
- The results of several plugins
  - E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\
- First, we should investigate:
  - orphan processes in the process list.
  - network connections.

Scenario 1 Labs: Memory Forensics Lab 1 (8)

- You already know that finding out orphan processes is important.
- We can use the following Volatility plugins to list process information (name, process id, parent pid, etc).
  - pslist
    - This plugin lists process information by following a link list in EPROCESS structures. It is similar to "tasklist" command.
  - pstree
    - This plugin displays a process tree with the same method as the pslist module.
  - psscan
    - This plugin is similar to pslist. Furthermore, it can search unlinked EPROCESS structures and show us the processes information.

Scenario 1 Labs: Memory Forensics Lab 1 (9)

- Let’s run pslist plugin.

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw pslist
```

- Can you find suspicious processes?

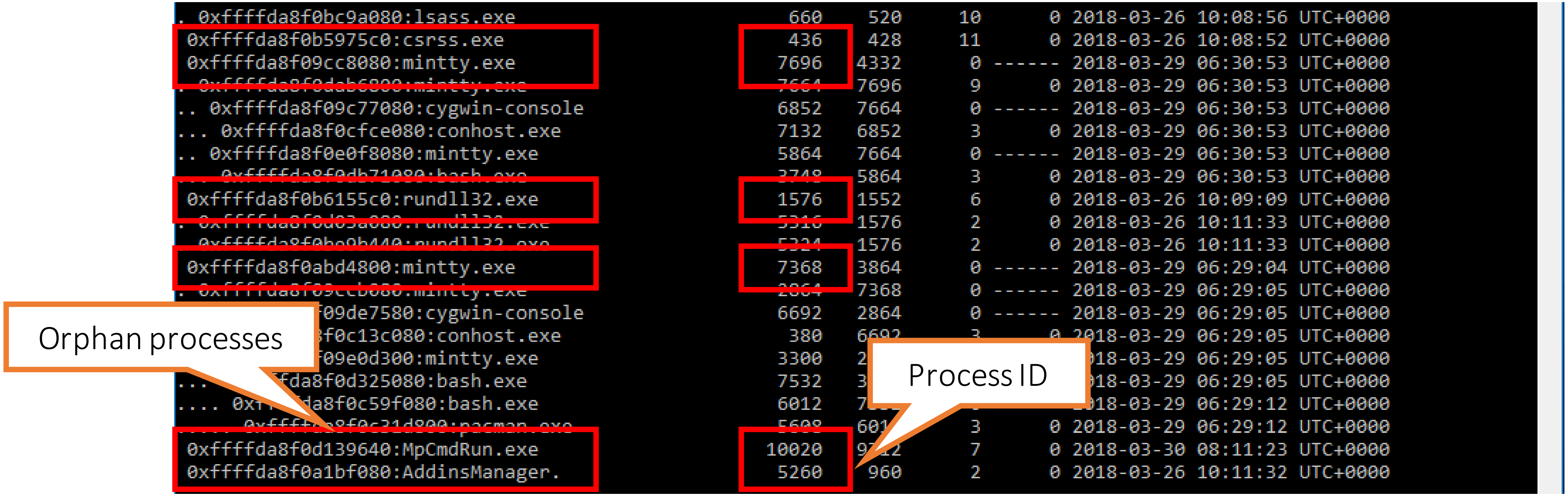
```
cmd.exe
0xffffda8f0cfce080 conhost.exe           7132   6852      3        0      1      0 2018-03-29 06:30:53 UTC+0000

0xffffda8f0e0f8080 mintty.exe            5864   7664      0 --------      1      0 2018-03-29 06:30:53 UTC+0000   2018-
03-29 06:30:53 UTC+0000
0xffffda8f0db71080 bash.exe              3748   5864      3        0      1      0 2018-03-29 06:30:53 UTC+0000

0xffffda8f0a386380 SearchUI.exe          8828    740     32        0      1      0 2018-03-29 06:39:18 UTC+0000

0xffffda8f0d09f800 firefox.exe           7636   4656     22        0      1      0 2018-03-29 08:57:06 UTC+0000

0xffffda8f0d38c800 SystemSettings       11020    740     17        0      1      0 2018-03-30 02:50:43 UTC+0000

0xffffda8f0fac3800 dllhost.exe            196    740      4        0      0      0 2018-03-30 07:38:22 UTC+0000

0xffffda8f0fb18600 svchost.exe           7984    652      4        0      0      0 2018-03-30 07:52:49 UTC+0000
```

Scenario 1 Labs: Memory Forensics Lab 1 (10)

- Then, let's execute pstree plugin.

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw pstree
```

- Can you find orphan processes?



Scenario 1 Labs: Memory Forensics Lab 1 (11)

- Orphan processes are “System”, “WinInit.exe”, "csrss.exe", "mintty.exe", "rundll32.exe", "MpCmdRun.exe", and "AddinsManager.".
- The names imply that:
  - “System” is the kernel process.
  - “Wininit.exe” is one of the core processes. It is in charge of initializing the user mode scheduling infrastructure.
  - "csrss.exe" is a Client/Server Runtime Subsystem.
  - "rundll32.exe" is a program to run an exported API on a DLL.
  - "MpCmdRun.exe" is Windows Defender.
  - "mintty.exe" is a terminal emulator of third party software.
  - "AddinsManager." is an unknown process.
- Let’s confirm the processes in detail. We will use "cmdline" plugin for that.

Scenario 1 Labs: Memory Forensics Lab 1 (12)

- Run cmdline plugin.

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw
cmdline --pid=436,7696,1576,7368,10020,5260
```

  - E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\cmdline_pid.txt

Scenario 1 Labs: Memory Forensics Lab 1 (13)

- The result of cmdline

```
Volatility Foundation Volatility Framework 2.6
************************************************************************
csrss.exe pid:    436
Command line : %SystemRoot%\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,20480,768 Windows=On SubSyste
mType=Windows ServerDll=basesrv,1 ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=sxssrv,4 ProfileControl=Off M
axRequestThreads=16
************************************************************************
rundll32.exe pid:   1576
Command line : RUNDLL32.EXE C:\Windows\SvS.DLL,GnrkQr
************************************************************************
AddinsManager. pid:   5260
Command line : "C:\Windows\addins\AddinsManager.exe"
************************************************************************
mintty.exe pid:   7368
************************************************************************
mintty.exe pid:   7696
************************************************************************
MpCmdRun.exe pid:  10020
Command line : "C:\Program Files\Windows Defender\\MpCmdRun.exe" SpyNetServiceDss -RestrictPrivileges -AccessKey 03D4163
1-64B5-6E44-0A69-3551931CA933 -Reinvoke
```

Scenario 1 Labs: Memory Forensics Lab 1 (14)

- We are going to investigate the details of these processes.
  - csrss.exe
  - MpCmdRun.exe
  - mintty.exe
  - rundll32.exe
  - AddinsManager.exe

Scenario 1 Labs: Memory Forensics Lab 1 (15)

- csrss.exe was launched just after System booted.

```
Name                                                  Pid   PPid   Thds   Hnds Time
-------------------------------------------------- ------ ------ ------ ------ ----
 0xffffda8f098ae040:System                               4      0    103      0 2018-03-26 10:08:27 UTC+0000
        (snip)
 0xffffda8f0b721080:wininit.exe                        520    428      1      0 2018-03-26 10:08:54 UTC+0000
        (snip)
 0xffffda8f0b5975c0:csrss.exe                          436    428     11      0 2018-03-26 10:08:52 UTC+0000
        (snip)
```

- It is an initial process and it is in the legitimate path. In addition, the parent process ID is the same as wininit.
- Next, we also know MpCmdRun.exe tends to be an orphan because it is launched from Task Scheduler.
- As a result, priorities of investigation on these files can be lowered than other candidates.

Scenario 1 Labs: Memory Forensics Lab 1 (16)

- ~~csrss.exe~~ : low priority
- ~~MpCmdRun.exe~~ : low priority
- mintty.exe
- rundll32.exe
- AddinsManager.exe

Memory Forensics Exercise 1 (17)

- There were no remainders of command line arguments for mintty.exe because the processes had already terminated.

When we check the result of pslist with findstr, you can find the processes had already exited.

```
C:\Tools\volatility>vol.py --profile=Win10x64_14393 -f C:\Users\taro\Desktop\shortcuts\12_MemoryForensics\artifacts\scen
ario1_memory_image\RAM_CLIENT-WIN10-1.raw pslist | findstr mintty
Volatility Foundation Volatility Framework 2.6
0xffffda8f0abd4800 mintty.exe             7368   3864      0 --------      1      0 2018-03-29 06:29:04 UTC+0000   2018-
03-29 06:29:05 UTC+0000
0xffffda8f09ccb080 mintty.exe             2864   7368      0 --------      1      0 2018-03-29 06:29:05 UTC+0000   2018-
03-29 06:30:22 UTC+0000
0xffffda8f09e0d300 mintty.exe             3300   2864      0 --------      1      0 2018-03-29 06:29:05 UTC+0000   2018-
03-29 06:29:05 UTC+0000
0xffffda8f09cc8080 mintty.exe             7696   4332      0 --------      1      0 2018-03-29 06:30:53 UTC+0000   2018-
03-29 06:30:53 UTC+0000
0xffffda8f0dab6080 mintty.exe             7664   7696      9        0      1      0 2018-03-29 06:30:53 UTC+0000

0xffffda8f0e0f8080 mintty.exe             5864   7664      0 --------      1      0 2018-03-29 06:30:53 UTC+0000   2018-
03-29 06:30:53 UTC+0000
```

Scenario 1 Labs: Memory

- Each mintty.exe has child processes such as bash.exe and pacman.exe.
- When we execute the command below, we can find that the full paths of them are:
  - C:\msys64\usr\bin\bash.exe
  - C:\msys64\usr\bin\pacman.exe
- Therefore, it is possible that mintty.exe is also a part of msys64.
- Msys64 is a set of UNIX utilities for Windows.

```
 0xffffda8f09cc8080:mintty.exe           7696
. 0xffffda8f0dab6800:mintty.exe          7664
.. 0xffffda8f09c77080:cygwin-console     6852
... 0xffffda8f0cfce080:conhost.exe       7132
.. 0xffffda8f0e0f8080:mintty.exe         5864
... 0xffffda8f0db71080:bash.exe          3748
 0xffffda8f0b6155c0:rundll32.exe         1576
. 0xffffda8f0d03a080:rundll32.exe        5316
. 0xffffda8f0be9b440:rundll32.exe        5324
 0xffffda8f0abd4800:mintty.exe           7368
. 0xffffda8f09ccb080:mintty.exe          2864
.. 0xffffda8f09de7580:cygwin-console     6692
... 0xffffda8f0c13c080:conhost.exe        380
.. 0xffffda8f09e0d300:mintty.exe         3300
... 0xffffda8f0d325080:bash.exe          7532
.... 0xffffda8f0c59f080:bash.exe         6012
..... 0xffffda8f0c31d800:pacman.exe      5608
```

```
C:\shortcuts\14_MemoryForensics\tools\volatility>vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\
RAM_CLIENT-WIN10-1.raw cmdline --pid=3748,5608
Volatility Foundation Volatility Framework 2.6
************************************************************************
pacman.exe pid:   5608
Command line : "C:\msys64\usr\bin\pacman.exe"
************************************************************************
bash.exe pid:   3748
Command line : "C:\msys64\usr\bin\bash.exe"
```

Scenario 1 Labs: Memory Forensics Lab 1 (19)

- Where is "mintty.exe" located at?
- Let’s confirm it with mftparser plugin.

IT TAKES A LONG TIME. See the result file

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw mftparser
```

- Open "E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\mftparser.txt" and find "mintty.exe".

```
$STANDARD_INFORMATION
Creation                       Modified                       MFT Altered                    Access Date                    Type
------------------------------ ------------------------------ ------------------------------ ------------------------------ ----
2016-10-25 08:32:48 UTC+0000 2018-02-08 12:08:06 UTC+0000   2018-03-29 06:29:59 UTC+0000   2018-03-29 06:29:59 UTC+0000   Archive

$FILE_NAME
Creation                       Modified                       MFT Altered                    Access Date                    Name/Path
------------------------------ ------------------------------ ------------------------------ ------------------------------ ---------
2016-10-25 08:32:48 UTC+0000 2018-03-29 06:29:59 UTC+0000   2018-03-29 06:29:59 UTC+0000   2018-03-29 06:29:59 UTC+0000   msys64\usr\bin\mintty.exe
```

- It only exists as msys64\usr\bin\mintty.exe. Therefore, we can consider it as a part of msys64. Plus, toyoda, the owner of this PC, told us that he installed this software in the interview.

Scenario 1 Labs: Memory Forensics Lab 1 (20)

- ~~csrss.exe~~ : low priority
- ~~MpCmdRun.exe~~ : low priority
- ~~mintty.exe~~ : low priority
- rundll32.exe
- AddinsManager.exe

Scenario 1 Labs: Memory Forensics Lab 1 (21)

- Rundll32.exe, which is used for executing an API on a dynamic link library (DLL), has a suspicious argument.
- The argument is "C:\Windows\SvS.DLL,GnrkQr". It is separated into "C:\Windows\SvS.DLL" and "GnrkQr".
  - The first part is a DLL, which is loaded by rundll32.exe.
  - The second part is an exported API name to be called by rundll32.exe.
    - Most of API names are human readable. But this one is not.

```
Volatility Foundation Volatility Framework 2.6
************************************************************************
csrss.exe pid:    436
Command line : %SystemRoot%\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,20480,768 Windows=On SubSyste
mType=Windows ServerDll=basesrv,1 ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=sxssrv,4 ProfileControl=Off M
axRequestThreads=16
************************************************************************
rundll32.exe pid:   1576
Command line : RUNDLL32.EXE C:\Windows\SvS.DLL,GnrkQr
************************************************************************
AddinsManager. pid:   5260
```

Scenario 1 Labs: Memory Forensics Lab 1 (22)

- When was "SvS.DLL" created?
- Search "SvS.DLL“ in “mftparser.txt“, the result of the mftparser plugin.

```
$STANDARD_INFORMATION
Creation                       Modified                       MFT Altered                    Access Date                    Type
------------------------------ ------------------------------ ------------------------------ ------------------------------ ----
2018-03-14 13:49:28 UTC+0000 2016-01-15 01:21:10 UTC+0000   2018-03-14 13:50:28 UTC+0000   2018-03-14 13:49:28 UTC+0000   Archive

$FILE_NAME
Creation                       Modified                       MFT Altered                    Access Date                    Name/Path
------------------------------ ------------------------------ ------------------------------ ------------------------------ ---------
2018-03-14 13:49:28 UTC+0000 2016-01-15 01:21:10 UTC+0000   2018-03-14 13:50:26 UTC+0000   2018-03-14 13:49:28 UTC+0000   Windows\SvS.DLL
```

Scenario 1 Labs: Memory Forensics Lab 1 (23)

- Next, let’s check network connections because most of the malware connect to C2 servers.
- We can use the "netscan" plugin to enumerate network status (IP address, port number, name, pid, etc).
  - The output is similar to "netstat -na“ command.

Scenario 1 Labs: Memory Forensics Lab 1 (24)

- Let’s confirm the result of netscan plugin.

IT TAKES A LONG TIME, so see the result file

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw netscan
```

  - E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\netscan.txt

- Unfortunately, we can’t confirm any network activities of rundll32.exe in this memory image at this time.

- Next, let’s check the SIDs associated with the process.

Scenario 1 Labs: Memory Forensics Lab 1 (25)

- Let’s confirm the result of getsids plugin.

IT TAKES A LONG TIME, so see the result file

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw getsids
```

- Open "E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\getsids.txt" and search rundll32.exe (PID:1576).

```
rundll32.exe (1576): S-1-5-18 (Local System)
rundll32.exe (1576): S-1-16-16384 (System Mandatory Level)
rundll32.exe (1576): S-1-1-0 (Everyone)
rundll32.exe (1576): S-1-5-32-545 (Users)
rundll32.exe (1576): S-1-5-6 (Service)
rundll32.exe (1576): S-1-2-1 (Console Logon (Users who are logged onto the physical console))
rundll32.exe (1576): S-1-5-11 (Authenticated Users)
rundll32.exe (1576): S-1-5-15 (This Organization)
rundll32.exe (1576): S-1-5-80-2962817144-200689703-2266453665-3849882635-1986547430 (BDESVC)
rundll32.exe (1576): S-1-5-80-864916184-135290571-3087830041-1716922880-4237303741 (BITS)
rundll32.exe (1576): S-1-5-80-3256172449-2363790065-3617575471-4144056108-756904704 (CertPropSvc)
```

Scenario 1 Labs: Memory Forensics Lab 1 (26)

- In this case, we haven't concluded whether "SvS.DLL" is malicious or not yet.
  - But we found that it takes a suspicious argument.
- rundll32.exe has the SYSTEM privilege. If the process was malicious, we can say that the PC is completely compromised at the system level.
  - Attackers had already gained the administrative rights of this machine.

- You need to extract the file after acquiring the disk image, and analyze it.
  - As you already know, it is malware.
- You could also dump the DLL on the memory.
  - Unfortunately, we cannot use this method in this case because it seems that Volatility Framework does not have implementation for dumping 32 bit DLLs on a 64 bit environment yet.

Scenario 1 Labs: Memory Forensics Lab 1 (27)

- ~~csrss.exe~~ : low priority
- ~~MpCmdRun.exe~~ : low priority
- ~~mintty.exe~~ : low priority
- rundll32.exe (SvS.DLL) : suspicious
- AddinsManager.exe

Scenario 1 Labs: Memory Forensics Lab 1 (28)

- AddinsManager.exe is located in “C:\Windows\addins\”. This folder exists on a clean image, but the binary does not exist in it.
- Its name implies a tool to manage addins, but we can’t find any remarkable results on this program when we search it on the Internet.

```
axRequestThreads=16
************************************************************************
rundll32.exe pid:   1576
Command line : RUNDLL32.EXE C:\Windows\SvS.DLL,GnrkQr
************************************************************************
AddinsManager. pid:   5260
Command line : "C:\Windows\addins\AddinsManager.exe"
************************************************************************
mintty.exe pid:   7368
************************************************************************
mintty.exe pid:   7696
************************************************************************
MpCmdRun.exe pid:  10020
Command line : "C:\Program Files\Windows Defender\\MpCmdRun.exe" SpyNetServiceDss -RestrictPrivileges -AccessKey 03D4163
```

Scenario 1 Labs: Memory Forensics Lab 1 (29)

- Next, let’s confirm the result of netscan plugin. IT TAKES A LONG TIME, so see the result file

```
vol.py --profile=Win10x64_14393 -f E:\Artifacts\scenario1_memory_image\RAM_CLIENT-WIN10-1.raw netscan
```

- Open the prepared result file
  - E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\netscan.txt
- Search “AddinsManager” in the file.

```
0xda8f0c61a600      TCPv4    192.168.52.40:57763             192.168.52.35:8080   ESTABLISHED      5260     AddinsManager.
 2018-03-30 08:14:55 UTC+0000
```

"192.168.52.35:8080" seems a HTTP proxy server because of the port number.
AddinsManager.exe must have tried to connect to external hosts.

  - Typical HTTP proxy servers use 3128/tcp or 8080/tcp.

Scenario 1 Labs: Memory Forensics Lab 1 (30)

- Let’s confirm the SIDs associated with this process.
- Open the prepared file.
  - E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\getsids.txt

```
AddinsManager. (5260): S-1-5-18 (Local System)
AddinsManager. (5260): S-1-16-16384 (System Mandatory Level)
AddinsManager. (5260): S-1-1-0 (Everyone)
AddinsManager. (5260): S-1-5-32-545 (Users)
AddinsManager. (5260): S-1-5-6 (Service)
AddinsManager. (5260): S-1-2-1 (Console Logon (Users who are logged onto the physical console))
AddinsManager. (5260): S-1-5-11 (Authenticated Users)
AddinsManager. (5260): S-1-5-15 (This Organization)
AddinsManager. (5260): S-1-5-80-2962817144-200689703-2266453665-3849882635-1986547430 (BDESVC)
AddinsManager. (5260): S-1-5-80-864916184-135290571-3087830041-1716922880-4237303741 (BITS)
AddinsManager. (5260): S-1-5-80-3256172449-2363790065-3617575471-4144056108-756904704 (CertPropSvc)
```

This process had been running with the SYSTEM privilege.

Scenario 1 Labs: Memory Forensics Lab 1 (31)

- When was "AddinsManager.exe" created? Let’s confirm it.
- Open the prepared result file.
    - E:\Artifacts\scenario1_memory_forensics\Client-Win10-1\mftparser.txt

```
$STANDARD_INFORMATION
Creation                       Modified                       MFT Altered                    Access Date                    Type
------------------------------ ------------------------------ ------------------------------ ------------------------------ ----
2016-06-13 05:41:28 UTC+0000 2016-06-13 05:41:28 UTC+0000   2018-03-26 10:11:31 UTC+0000   2016-06-13 05:41:28 UTC+0000   Archive

$FILE_NAME
Creation                       Modified                       MFT Altered                    Access Date                    Name/Path
------------------------------ ------------------------------ ------------------------------ ------------------------------ ---------
2018-03-20 10:00:04 UTC+0000 2018-03-20 10:00:04 UTC+0000   2018-03-20 10:00:04 UTC+0000   2018-03-20 10:00:04 UTC+0000   Windows\addins\ADDINS~1.EXE

$FILE_NAME
Creation                       Modified                       MFT Altered                    Access Date                    Name/Path
------------------------------ ------------------------------ ------------------------------ ------------------------------ ---------
2018-03-20 10:00:04 UTC+0000 2018-03-20 10:00:04 UTC+0000   2018-03-20 10:00:04 UTC+0000   2018-03-20 10:00:04 UTC+0000   Windows\addins\AddinsManager.exe
```

Scenario 1 Labs: Memory Forensics Lab 1 (32)

- It is possible that the creation/modification/access date and time in $SI (STANDARD_INFORMATION) were manipulated because the time of $SI is different from $FN's ($FILE_NAME) one.

- $STANDARD_INFORMATION
  - Creation : 2016-06-13 05:41:28 UTC+0000
- $FILE_NAME
  - Creation : 2018-03-20 10:00:04 UTC+0000

Scenario 1 Labs: Memory Forensics Lab 1 (33) - Summary

- We found two suspicious binaries.
  - SvS.DLL (called by rundll32.exe)
  - AddinsManager.exe

- The suspicious points of "rundll32.exe" and "SvS.DLL".
  - rundll32.exe, which calls SvS.DLL, is an orphan process.
  - It calls a strange API name from SvS.DLL.

- The suspicious points of "AddinsManager.exe".
  - It is an orphan process as well.
  - Its MFT metadata seems to be manipulated.
  - It communicated with external hosts via the victim's proxy server.

- We need to get these binaries after acquiring the disk image, and perform further analysis for both of them such as malware analysis.

Appendix 15-3
Extra Exercise 2
Rootkit Forensics

Extra Exercise 2: Rootkit Forensics (1)

- Memory Image
  - E:\Artifacts\other_memory_image\memory_image2.dmp
- Volatility profile
  - Win7SP1x86
- The result files of several plugins
  - E:\Artifacts\other_memory_forensics\extra_exercise2\

- First, let’s investigate orphan processes and network connections.

Extra Exercise 2: Rootkit Forensics (2)

- Let’s execute pstree plugin.

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp pstree
```

  - E:\Artifacts\other_memory_forensics\extra_exercise2\pstree.txt

- Were you able to find out any suspicious processes?

Extra Exercise 2: Rootkit Forensics (3)

- Let’s check the image with netscan plugin. IT TAKES A LONG TIME, so see the result file

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp netscan
```

  - E:\Artifacts\other_memory_forensics\extra_exercise2\netscan.txt

- Were you able to find out any suspicious processes?

Extra Exercise 2: Rootkit Forensics (4)

- We were not able to find any remarkable processes or any notable network activities.
- Then, we will check auto-start locations of malware.

Extra Exercise 2: Rootkit Forensics (5)

- Let’s check the image with autoruns plugin.

IT TAKES A LONG TIME, so see the result file

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp autoruns
```

  - E:\Artifacts\other_memory_forensics\extra_exercise2\autoruns.txt
- The result of autoruns

```
Service: Ultra3 (None) - Kernel driver, System Start
    Image path: \SystemRoot\$NtUninstallQ923283$\fdisk.sys  (Last modified: 2018-04-19 04:00:22 UTC+0000)
    PIDs: -
```

  - One of the driver files is located in "\SystemRoot\$NtUninstallQ923283$".
    - "\SystemRoot\" = "C:\Windows\".
  - "$NtUninstallQxxxxxx$" folders are supposed to be only on Windows XP/2003 as backup folders of Windows updates. It is very suspicious because this memory image was acquired from Windows 7 SP1. In addition, it is very strange that the running driver was installed there in the first place.
  - You will need to retrieve this file from a disk image and perform malware analysis.

Extra Exercise 2: Rootkit Forensics (6)

- You can find a suspicious driver on this machine located in a strange folder.
- The attackers already had the administrative privilege because they were able to install the driver.
- It was a kernel mode driver so that we need to check any API hooks by overwriting pointers on dispatch routines such as IDT or SSDT, or by code alternation in APIs that is called inline-hooking on the kernel or drivers because rootkits and bootkits hook such tables or code that the tables point to.

Extra Exercise 2: Rootkit Forensics (7)

- Let's confirm the image with apihooks plugin. IT TAKES A LONG TIME, so see the result file.

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp apihooks
```

  - E:\Artifacts\other_memory_forensics\extra_exercise2\apihooks.txt
  - This plugin can detect kernel mode API hooks as well as user mode's one.

There is no suspicious entries in the result of this plugin.

```
*************************************************************
Hook mode: Usermode
Hook type: Import Address
Process: 4052 (DumpIt.exe)
Victim module: DumpIt.exe
Function: netapi32.dll!Net
Hook address: 0x736f2c3f
Hooking module: wkscli.dll

Disassembly(0):
0x736f2c3f 6a1c          PUSH 0x1c
0x736f2c41 68b02c6f73    PUSH DWORD 0x736f2cb0
0x736f2c46 e805e5ffff    CALL 0x736f1150
0x736f2c4b 33f6          XOR ESI, ESI
0x736f2c4d 8975e0        MOV [EBP-0x20], ESI
0x736f2c50 39750c        CMP [EBP+0xc], ESI
0x736f2c53 0f            DB 0xf
0x736f2c54 84            DB 0x84
0x736f2c55 a7            CMPSD
0x736f2c56 15            DB 0x15
```

It is a dumpit process. We can ignore this entry because it is for dumping memory for this exercise.

```
****************************
Hook mode: Kernelmode
Hook type: Unknown Code Page
Victim module: ntoskrnl.exe
Function: <unknown>
Hook address: 0x85745000
Hooking module: <unknown>

Disassembly(0):
0x832e2adb ff1570ca3783    CALL DWORD [0x8337ca70]
0x832e2ae1 5b              POP EBX
0x832e2ae2 5e              POP ESI
0x832e2ae3 5f              POP EDI
0x832e2ae4 c21800          RET 0x18
0x832e2ae7 90              NOP
0x832e2ae8 55              PUSH EBP
0x832e2ae9 53              PUSH EBX
0x832e2aea 56              PUSH ESI
0x832e2aeb 57              PUSH EDI
0x832e2aec 83ec0c          SUB ESP, 0xc
0x832e2aef 8bcc            MOV ECX, ESP
0x832e2af1 81              DB 0x81
0x832e2af2 ec              IN AL, DX

Disassembly(1):
0x85745000 b802000000      MOV EAX, 0x2
0x85745005 c3              RET
0x85745006 0000            ADD [EAX], AL
0x85745008 0000            ADD [EAX], AL
```

This is ntoskrnl.exe. And, there is no suspicious factor in this code. In addition, we confirmed that this code is included in legitimate images. It is a false positive.

Extra Exercise 2: Rootkit Forensics (9)

- Let's confirm the image with ssdt plugin.

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp ssdt
```

  - E:\Artifacts\other_memory_forensics\extra_exercise2\ssdt.txt

  - SSDT stands for "System Service Descriptor Table". It is similar to System call table of the Linux system. It is an interface between kernel land and user land. For example, CreateFile API in kernel32.dll execute ZwCreateFile/NtCreateFile API that is the lower level API of it in ntdll.dll in user land. The lower level API enter the kernel land and it reaches the API with the same name in SSDT.
  - Typical functions in SSDT consists of the APIs only on ntoskrnl.exe and win32k.sys. Therefore, if the table had pointers to any other modules or memory sections, the entries are suspicious.

Extra Exercise 2: Rootkit Forensics (10)

- In this case, there is no suspicious entries in the result of the “ssdt” plugins as well.

```
Select-String "by ntoskrnl.exe$|by win32k.sys$" E:\Artifacts\other_memory_forensics\extra_exercise\ssdt.txt -NotMatch | % { $_.Line } | % {$_.ToString()}
```

```
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 83290d9c with 401 entries
SSDT[1] at 92856000 with 825 entries
```

There is no entries except for ntoskrnl.exe and win32k.sys.

- Next, let’s check IDT hooks.

Extra Exercise 2: Rootkit Forensics (11)

- IDT (Interrupt Descriptor Table)
  - IDT is one of the dispatch routines for processing interrupts on Intel architecture.
  - It can register 256 handlers.

Extra Exercise 2: Rootkit Forensics (12)

- What is interrupt?
  - It is a CPU mechanism to process events or requests that are issued by hardware or software.
  - There are three kinds of interrupts.
    - Exception
      - It is an interrupt caused by a fault in software such as division by 0.
    - Interrupt ReQuest (IRQ)
      - It is an interrupt when a request is made from hardware such as “a key pressed on a keyboard" or “completion of reading HDD data".
    - Software Interrupt
      - It is an interrupt that software intentionally generates with an INT instruction, which is in machine language to issue interrupt .

Extra Exercise 2: Rootkit Forensics (13)

- Let’s take a look at the result of idt plugin.

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp idt
```

  - E:\Artifacts\other_memory_forensics\extra_exercise2\idt.txt
- The result of idt plugin
  - There are some "UNKNOWN" modules.

Extra Exercise 2: Rootkit Forensics (14)

0	B0	0x8	0x8616da58	UNKNOWN	
0	B1	0x8	0x8616dcd8	UNKNOWN	
0	B2	0x8	0x861b37d8	UNKNOWN	
0	B3	0x8	0x861be558	UNKNOWN	
0	B4	0x8	0x861c82d8	UNKNOWN	
0	B5	0x8	0x861d2058	UNKNOWN	
0	B6	0x8	0x8324fbec	ntoskrnl.exe	.text
0	B7	0x8	0x8324fbf6	ntoskrnl.exe	.text
0	B8	0x8	0x8324fc00	ntoskrnl.exe	.text
0	B9	0x8	0x8324fc0a	ntoskrnl.exe	.text
0	BA	0x8	0x8324fc14	ntoskrnl.exe	.text
0	BB	0x8	0x8324fc1e	ntoskrnl.exe	.text
0	BC	0x8	0x8324fc28	ntoskrnl.exe	.text
0	BD	0x8	0x8324fc32	ntoskrnl.exe	.text
0	BE	0x8	0x8324fc3c	ntoskrnl.exe	.text
0	BF	0x8	0x8324fc46	ntoskrnl.exe	.text
0	C0	0x8	0x8324fc50	ntoskrnl.exe	.text
0	C1	0x8	0x8363e3f4	hal.dll	_PAGELK
0	C2	0x8	0x8324fc64	ntoskrnl.exe	.text
0	C3	0x8	0x86f7a670	UNKNOWN	

There are many "UNKNOWN" modules. We have to focus on them. You can use "--verbose" option to confirm details of them.

Extra Exercise 2: Rootkit Forensics (15)

- Let's see the result of idt plugin with verbose mode.

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp idt --verbose
```

  - E:\Artifacts\other_memory_forensics\extra_exercise2\idt_verbose.txt

Extra Exercise 2: Rootkit Forensics (16)

- The result of "idt“ with verbose mode
  - Most of the codes have the same procedures.

powershell.exe

```
PS C:\shortcuts\15_MemoryForensics\tools> select-string -Context 0,10 Unknown E:\Artifacts\other_memory
_forensics\extra_exercise2\idt_verbose.txt | % {$_.context.precontext;$_.line;$_.context.postcontext}
     0      51         0x8 0x861b3a58 UNKNOWN
0x861b3a58 54                 PUSH ESP
0x861b3a59 55                 PUSH EBP
0x861b3a5a 53                 PUSH EBX
0x861b3a5b 56                 PUSH ESI
0x861b3a5c 57                 PUSH EDI
0x861b3a5d 83ec54             SUB ESP, 0x54
0x861b3a60 8bec               MOV EBP, ESP
0x861b3a62 894544             MOV [EBP+0x44], EAX
0x861b3a65 894d40             MOV [EBP+0x40], ECX
0x861b3a68 89553c             MOV [EBP+0x3c], EDX
     0      52         0x8 0x861be7d8 UNKNOWN
0x861be7d8 54                 PUSH ESP
0x861be7d9 55                 PUSH EBP
0x861be7da 53                 PUSH EBX
0x861be7db 56                 PUSH ESI
0x861be7dc 57                 PUSH EDI
0x861be7dd 83ec54             SUB ESP, 0x54
0x861be7e0 8bec               MOV EBP, ESP
0x861be7e2 894544             MOV [EBP+0x44], EAX
0x861be7e5 894d40             MOV [EBP+0x40], ECX
0x861be7e8 89553c             MOV [EBP+0x3c], EDX
     0      53         0x8 0x861c8558 UNKNOWN
```

Extra Exercise 2: Rootkit Forensics (17)

- The result of "idt“ with verbose mode (cont.)
  - But, the “C3” handler was a different story. We can see the consecutive NOPs!
  - We need to investigate around this code deeply.

```
0x861d2068 89553c              MOV [EBP+0x3c], EDX
     0     C3        0x8 0x86f7a670 UNKNOWN
0x86f7a670 90                  NOP
0x86f7a671 90                  NOP
0x86f7a672 90                  NOP
0x86f7a673 90                  NOP
0x86f7a674 90                  NOP
0x86f7a675 90                  NOP
0x86f7a676 90                  NOP
0x86f7a677 90                  NOP
0x86f7a678 90                  NOP
0x86f7a679 90                  NOP
```

Extra Exercise 2: Rootkit Forensics (18)

- Let's run volshell plugin.

```
vol.py --profile=Win7SP1x86 -f E:\Artifacts\other_memory_image\memory_image2.dmp volshell
```

- We can execute Volatility Framework interactively, such as disassemble or dump hex data on arbitrary addresses, with this plugin.
    - ps() : Print active processes in a table view.
    - cc(pid=*PID*) : Change current shell context.
    - sc() : Show the current context.
    - db(address, length=128) : Print bytes as canonical hexdump.
    - dd(address, length=128) : Print dwords at address.
    - dis(address, length=128) : Disassemble code at a given address.

Extra Exercise 2: Rootkit Forensics (19)

- Let’s disassemble the address at the “C3” handler.

```
>>> dis(0x86f7a670)
0x86f7a670 90                   NOP
0x86f7a671 90                   NOP
0x86f7a672 90                   NOP
0x86f7a673 90                   NOP
0x86f7a674 90                   NOP
0x86f7a675 90                   NOP
0x86f7a676 90                   NOP
0x86f7a677 90                   NOP
0x86f7a678 90                   NOP
0x86f7a679 90                   NOP
0x86f7a67a 90                   NOP
0x86f7a67b 90                   NOP
0x86f7a67c 90                   NOP
0x86f7a67d 90                   NOP
0x86f7a67e 90                   NOP
0x86f7a67f 90                   NOP
0x86f7a680 6a08                 PUSH 0x8
0x86f7a682 6888a6f786           PUSH DWORD 0x86f7a688
0x86f7a687 cb                   RETF
0x86f7a688 fb                   STI
0x86f7a689 50                   PUSH EAX
0x86f7a68a 51                   PUSH ECX
0x86f7a68b 0fb6442414           MOVZX EAX, BYTE [ESP+0x14]
0x86f7a690 8b4c2418             MOV ECX, [ESP+0x18]
0x86f7a694 894c2414             MOV [ESP+0x14], ECX
0x86f7a698 8b0d50dcf986         MOV ECX, [0x86f9dc50]
```

Jump to 0x86f7a688 with PUSH + RETF instruction. It is a suspicious code because compilers never output such a code.

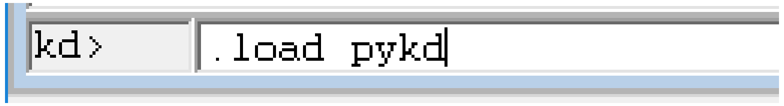
Extra Exercise 2: Rootkit Forensics (20)

- How can we find APIs that call this code? Unfortunately, apihooks plugin could not detect them as you saw in this case.

- Therefore, we will find them using WinDbg with pykd and our simple python script.
- The script flow is:
  1. Enumerating API names and the addresses in the kernel
  2. Acquiring the disassembled code on the address
  3. Displaying an API name if the "INT C3" is included in the bytes
  4. Repeating steps 2 to 3 on every API
  - See the source code on “Appendix 15-1” in this chapter if you are interested.

Extra Exercise 2: Rootkit Forensics (21)

1. Start “WinDbg (X64) (windbg.exe)” in the shortcuts folder.
2. Click “File”, choose “Open Crash Dump”, and open the file below.
   - E:\Artifacts\other_memory_image\memory_image2.dmp
3. Input ".load pykd“ at the bottom of the window.



Extra Exercise 2: Rootkit Forensics (22)

4. We can see fifteen APIs displayed when we execute our script by inputting the command below at the bottom of the window.

```
!py -3 C:\Tools\pykd_scripts\find_idt_hook.py
```

```
IDT Hook: 83414e2a nt!NtCreateKey
IDT Hook: 8344a644 nt!NtQueryInformationProcess
IDT Hook: 83443cd4 nt!NtQuerySystemInformation
IDT Hook: 83448d71 nt!ObOpenObjectByName
IDT Hook: 8345837a nt!NtClose
IDT Hook: 833d531c nt!IoCreateDevice
IDT Hook: 8347aa59 nt!NtEnumerateKey
IDT Hook: 835165ad nt!NtShutdownSystem
IDT Hook: 8346e9bf nt!NtTerminateProcess
IDT Hook: 8324952f nt!IofCallDriver
IDT Hook: 8345fcae nt!NtQueryKey
IDT Hook: 83482056 nt!NtCreateUserProcess
IDT Hook: 834efc02 nt!NtCreateThread
IDT Hook: 834a7176 nt!NtSaveKey
IDT Hook: 83475c8c nt!NtReadFile
```

Extra Exercise 2: Rootkit Forensics (23)

- How do they call the “C3” handler? Let’s confirm it.
- When we input "u nt!NtCreateKey" at the bottom of the window, we can disassemble hooked “NtCreateKey” API.

```
83414e2a 6a04           push    4
83414e2c cdc3           int     0C3h
83414e2e 90             nop
83414e2f 51             push    ecx
83414e30 6a00           push    0
83414e32 ff7520         push    dword ptr [ebp+20h]
83414e35 ff751c         push    dword ptr [ebp+1Ch]
83414e38 ff7518         push    dword ptr [ebp+18h]
```

- The code pushes a number four to the stack, and then issues the “int 0xC3” instruction to call the handler.

Extra Exercise 2: Rootkit Forensics (24)

- We found several inline hooks on the kernel APIs.
- It is possibly a malicious activity because legitimate drivers, except for several special drivers such as anti-virus software, never do it.

- Now we need to dump the memory region and analyze it in detail.

NtCreateKey API (Hooked by the rootkit)

IDT C3 Handler calls with an argument (4)

C3 Handler

```
86f7a678 90              nop
86f7a679 90              nop
86f7a67a 90              nop
86f7a67b 90              nop
86f7a67c 90              nop
86f7a67d 90              nop
86f7a67e 90              nop
86f7a67f 90              nop
86f7a680 6a08            push    8
86f7a682 6888a6f786      push    86F7A688h
86f7a687 cb              retf
86f7a688 fb              sti
86f7a689 50              push    eax
86f7a68a 51              push    ecx
86f7a68b 0fb6442414      movzx   eax,byte ptr [esp+14h]        (1)
86f7a690 8b4c2418        mov     ecx,dword ptr [esp+18h]
86f7a694 894c2414        mov     dword ptr [esp+14h],ecx
86f7a698 8b0d50dcf986    mov     ecx,dword ptr ds:[86F9DC50h]  (2)
86f7a69e 8d04c1          lea (3) eax,[ecx+eax*8]
86f7a6a1 8b4804          mov     ecx,dword ptr [eax+4]
86f7a6a4 894c2418        mov     dword ptr [esp+18h],ecx
86f7a6a8 59              pop     ecx
86f7a6a9 8b00            mov (4) eax,dword ptr [eax]
86f7a6ab 870424          xchg    eax,dword ptr [esp]           (5)
86f7a6ae c20c00          ret     0Ch
86f7a6b1 cc              int     3
86f7a6b2 ff2508c3f086    jmp     dword ptr ds:[86F0C308h]
86f7a6b8 ff2504c3f086    jmp     dword ptr ds:[86F0C304h]
86f7a6be ff2500c3f086    jmp     dword ptr ds:[86F0C300h]
86f7a6c4 ff25fcc2f086    jmp     dword ptr ds:[86F0C2FCh]
86f7a6ca ff25f8c2f086    jmp     dword ptr ds:[86F0C2F8h]
86f7a6d0 ff25f4c2f086    jmp     dword ptr ds:[86F0C2F4h]
```

Function Table

```
kd> dd poi(86F9DC50)
87293770  86f11790 872d0980 86f11020 85881eb8
87293780  86f16a60 86c3ea78 86f16cc0 85cab630
87293790  86f16ea0 8591bdf0 86f16f90 858fb990
872937a0  86f19f00 859e3ad8 86f1a040 87d2aad8
872937b0  86f19fe0 87d2aa40 86f19ce0 858ba008
872937c0  86f1aa60 858ba0a8 86f19d90 87b80988
872937d0  86f1a880 87b808f0 86f1a930 87cac660
872937e0  86f19c30 87cac5c8 86f3c570 859e0518
```

(1) EAX = 4 (Get INT C3's argument)

(2) ECX = 0x87293770 = ptr [0x86f9dc50]

(3) EAX = 0x87293790 = 0x87293770 + 4 * 8

(4) EAX = 0x86f163a0 = ptr [0x87293790]

(5) Exchange values: [ESP] = 0x86f163a0

Go to this function (ret instruction pops the top value on stack)

```
86f16ea0 55              push    ebp
86f16ea1 8bec            mov     ebp,esp
86f16ea3 6aff            push    0FFFFFFFFh
86f16ea5 6828aaf786      push    86F7AA28h
86f16eaa 6810a9f786      push    86F7A910h
86f16eaf 64a100000000    mov     eax,dword ptr fs:[00000000h]
86f16eb5 50              push    eax
86f16eb6 648925000000000 mov     dword ptr fs:[0],esp
```

Extra Exercise 2: Rootkit Forensics (25)

- What memory section includes the function (0x86f16ea0) ? Let's confirm it.

```
kd> !address 0x86f16ea0


Usage:
Base Address:           86000000
End Address:            88000000
Region Size:            02000000
VA Type:                NonPagedPool
```

We found that it is included in a NonPagedPool memory.
Roughly speaking, pool means kernel version of heap memory.
NonPaged means the pool memory will never swap out.

Extra Exercise 2: Rootkit Forensics (26)

- We can get the head of the pool chunk address and its size with "!pool" command.

```
kd> !pool 86f16ea0
Pool page 86f16ea0 region is Nonpaged pool
 86f16000 size:  8d8 previous size:    0  (Free )  .... (Protected)

86f168d8 doesn't look like a valid small pool allocation, checking to see
if the entire page is actually part of a large page allocation...

*86f0b000 : large page allocation, tag is NtFs, size is 0x9b000 bytes
            Pooltag NtFs : StrucSup.c, Binary : ntfs.sys
```

Head of the pool chunk address is this value.

Extra Exercise 2: Rootkit Forensics (27)

- Let's dump it and see the bytes around the address!

```
kd> db 86f0b000
86f0b000  00 00 00 00 03 00 00 00-04 00 00 00 ff ff 00 00  ................
86f0b010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
86f0b020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
86f0b030  00 00 00 00 00 00 00 00-00 00 00 00 d0 00 00 00  ................
86f0b040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
86f0b050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
86f0b060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS 
86f0b070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

- It seems to have a PE file header, but it looks strange because the two bytes from the head is not "4D 5A (MZ)". It's really suspicious.

Extra Exercise 2: Rootkit Forensics (28)

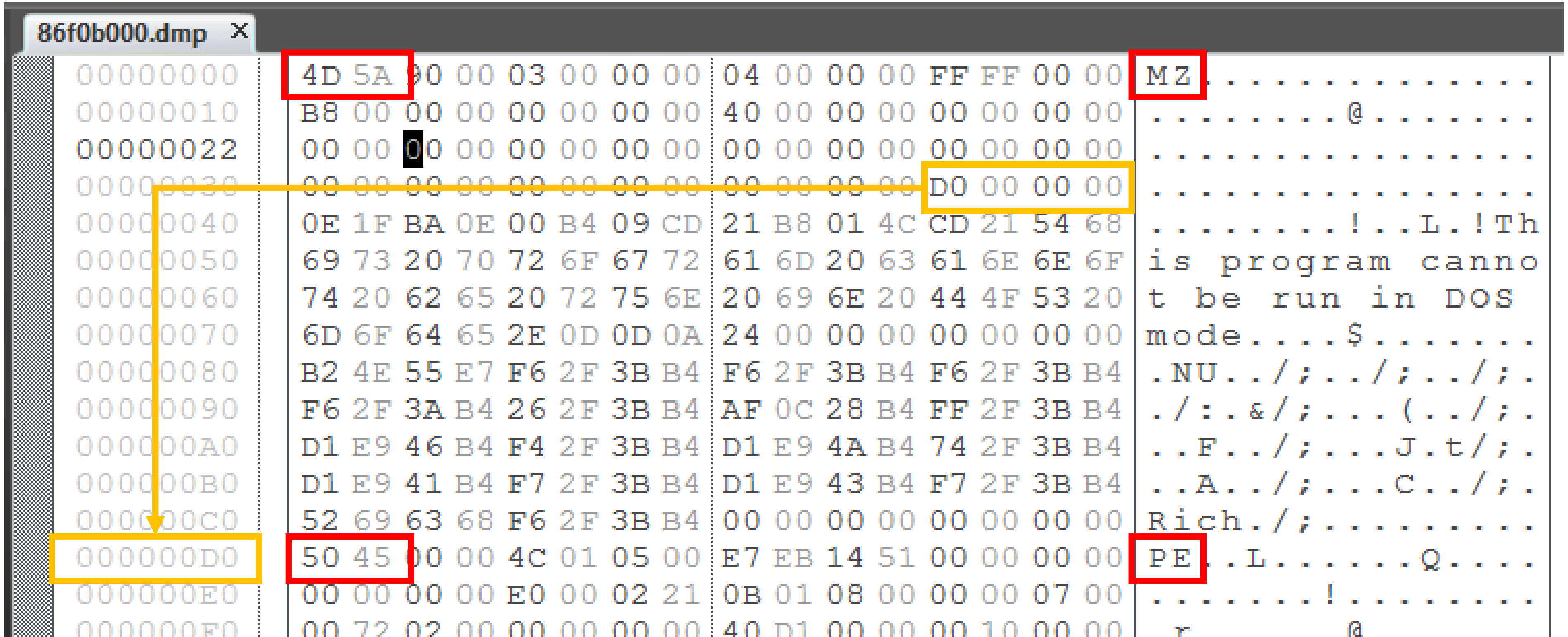
- Dump the pool chunk on the address 0x86f0b000 and its size 0x9b000 to your desktop with “.writemem” command on WinDbg.

```
.writemem C:\Users\taro\Desktop\86f0b000.dmp 86f0b000 L9b000
```

- Now, we need to fix several values on the PE header of the dumped file. Copy this file with an arbitrary name for backup.

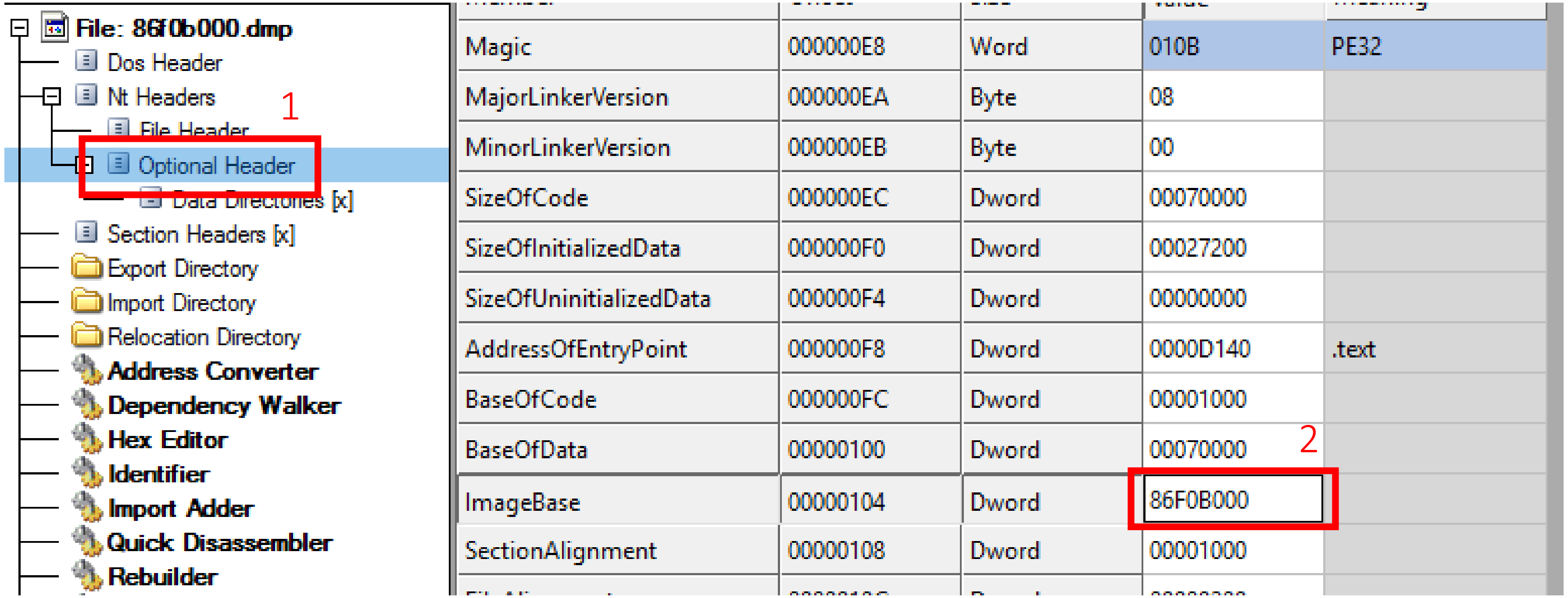
Extra Exercise 2: Rootkit Forensics (29)

- Next, open "FileInsight" and drag and drop the dumped file into it. Then fix MZ and PE header.
  - Offset 0 : "4D 5A" (MZ)
  - Offset 0xD0 : "50 45" (PE)



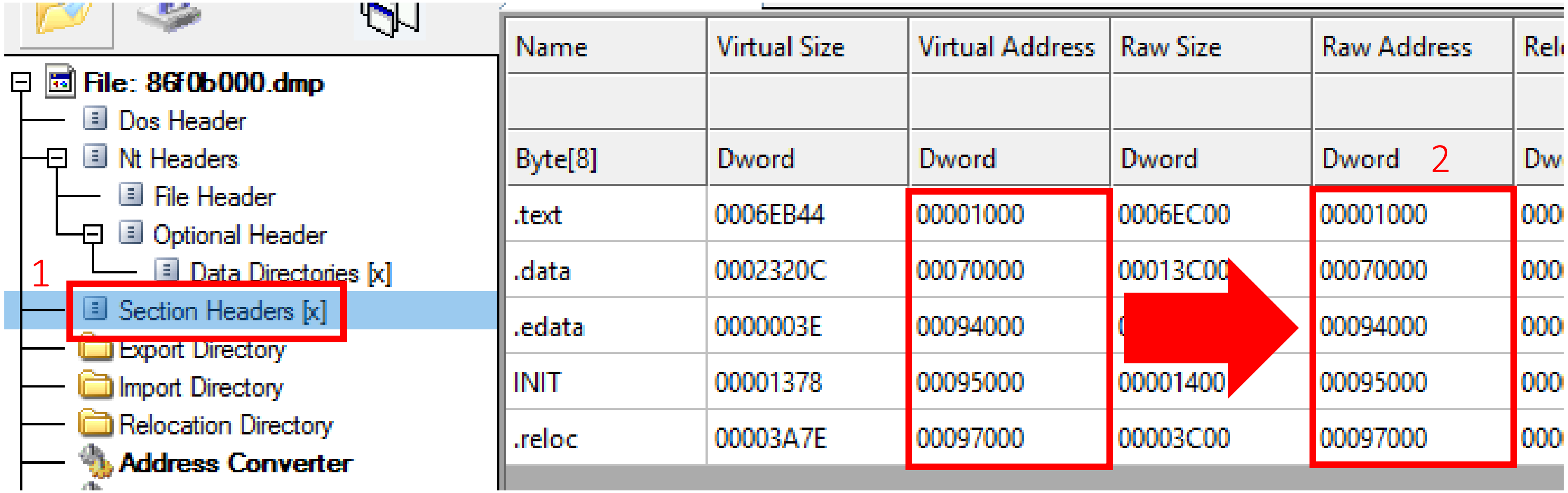
Extra Exercise 2: Rootkit Forensics (30)

- Next, open “CFF Explorer”. And fix the value of “Image Base” to “86F0B000”.
  - The value is the head of the pool chunk address on memory.



Extra Exercise 2: Rootkit Forensics (31)

- Finally, overwrite the values of “Raw Address” in “Section Headers” with the values of “Virtual Address”.

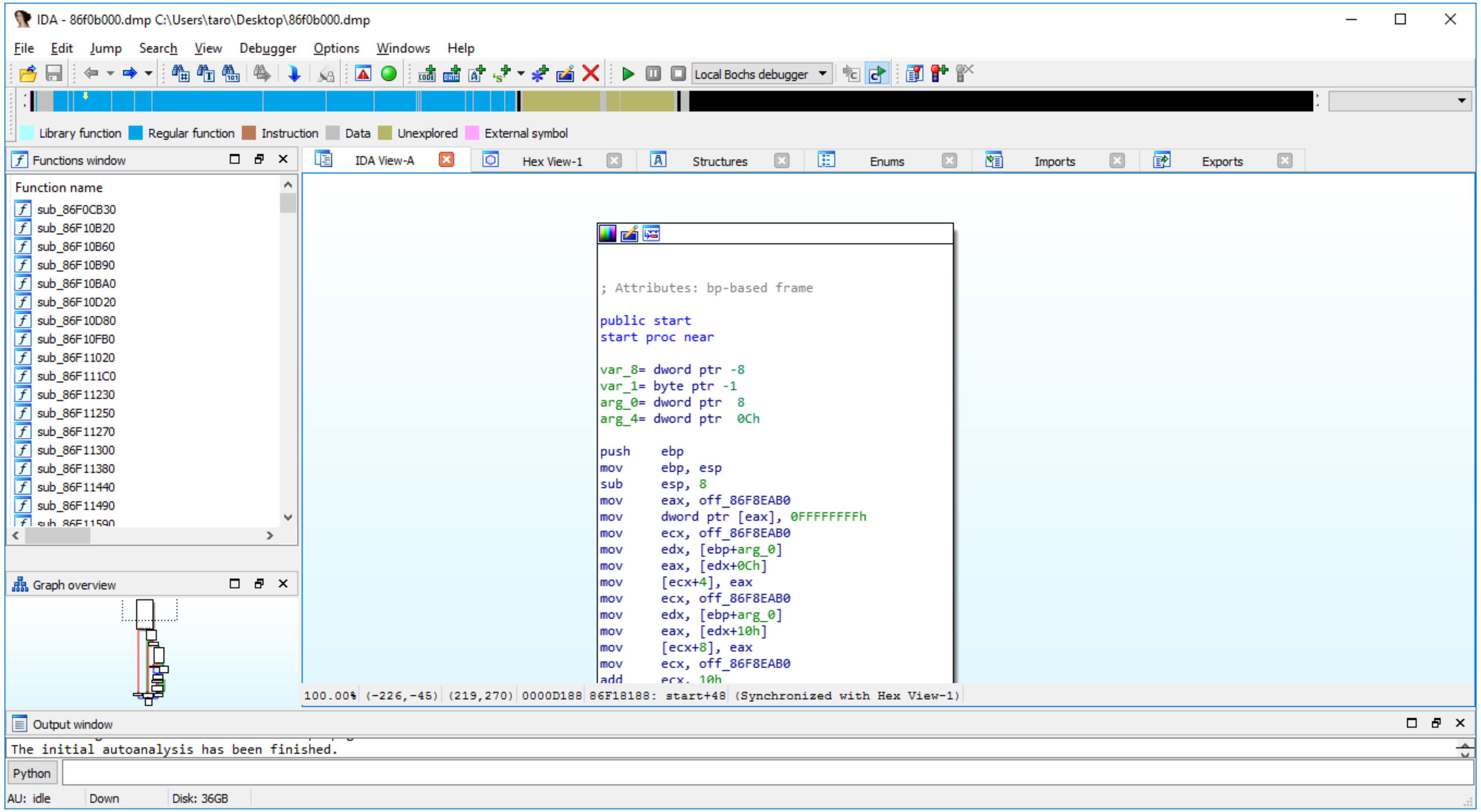


Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Rel
Byte[8]	Dword	Dword	Dword	Dword	Dw
.text	0006EB44	00001000	0006EC00	00001000	000
.data	0002320C	00070000	00013C00	00070000	000
.edata	0000003E	00094000	[illegible]	00094000	000
INIT	00001378	00095000	00001400	00095000	000
.reloc	00003A7E	00097000	00003C00	00097000	000

When Windows loader loads each section to memory, it aligns the values on “Virtual Address”.

Extra Exercise 2: Rootkit Forensics (32)

- Load it into IDA. Then you can analyze the code.



Extra Exercise 2: Rootkit Forensics (33) - Summary

- We were able to get a suspicious entry from the result of autoruns plugin.
  - It was a kernel mode driver and was installed in the “$NtUninstallQ923283$” folder, which does not exist on Windows 7 by default.
- One of IDT handlers was overwritten.
- We were able to dump a suspicious PE file on memory.
  - MZ and PE headers were manipulated.

- We are not sure whether the code on memory and the driver on disk are related to each other or not. We need to perform malware analysis for further investigation.
- See the URL if you are interested about this malware in detail.
  - https://www.gdatasoftware.com/blog/2014/06/23953-analysis-of-uroburos-using-windbg