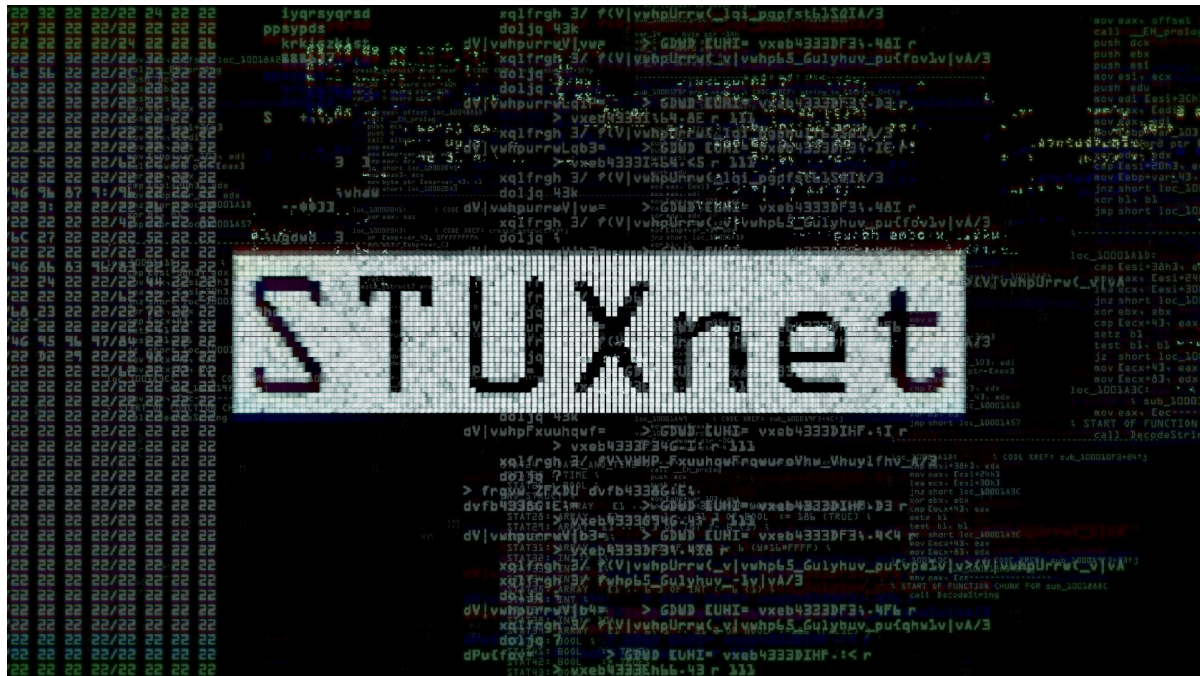


Lab4: Stuxnet (16 + 6 Points)



Objectives and Targets

Stuxnet is a malicious computer worm, first uncovered in 2010. Thought to have been in development since at least 2005, Stuxnet targets SCADA systems and is believed to be responsible for causing substantial damage to Iran's nuclear program.

In this lab, we'll examine Stuxnet's footprint in memory using Volatility 2.0 ([Link](#)). Please follow the instructions and answer all questions.

Remote Login:

Please use secure shell (SSH) software to login my remote linux server (archlinux).

```
IP: 35.231.146.25
UserName: student
Password: cscwcupa
```

```
quake0day@quakes-iMac ~$ ssh student@35.231.146.25
student@35.231.146.25's password:
Last login: Tue Apr 23 16:47:11 2019 from 100.14.108.88
[student@archlinux ~]$
```

Target 1: Hollow Process Injection:

Step 1:

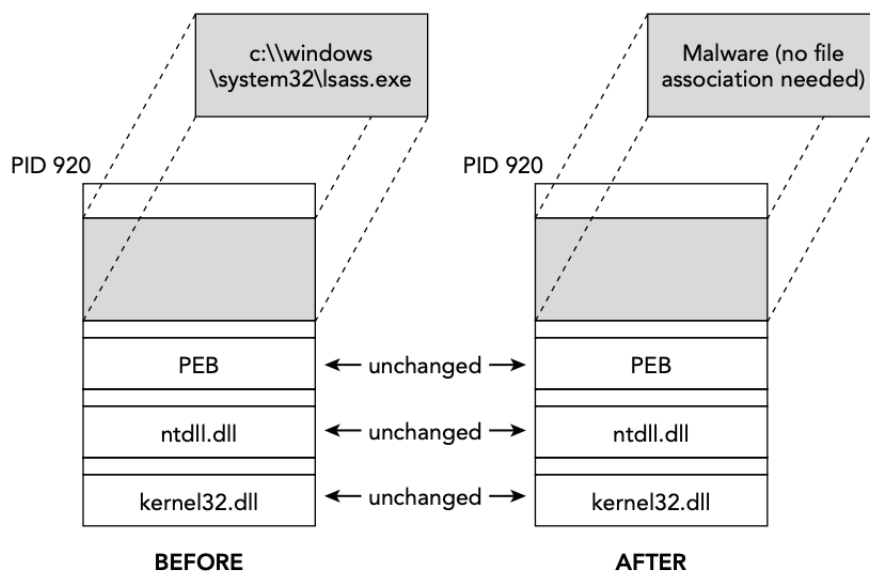
Type `vol.py -f stuxnet.vmem pstree`, it'll show all running processes.

```
[student@archlinux ~]$ vol.py -f stuxnet.vmem pstree
Volatility Foundation Volatility Framework 2.6.1
Name                               Pid  PPid  Thds  Hnds Time
-----
0x823c8830:System                   4    0    59   403 1970-01-01 00:00:00 UTC+0000
. 0x820df020:smss.exe               376   4     3    19 2010-10-29 17:08:53 UTC+0000
.. 0x821a2da0:csrss.exe             600  376    11   395 2010-10-29 17:08:54 UTC+0000
.. 0x81da5650:winlogon.exe          624  376    19   570 2010-10-29 17:08:54 UTC+0000
... 0x82073020:services.exe         668  624    21   431 2010-10-29 17:08:54 UTC+0000
.... 0x81fe52d0:vmtoolsd.exe        1664  668     5   284 2010-10-29 17:09:05 UTC+0000
..... 0x81c0cda0:cmd.exe             968  1664    0  ----- 2011-06-03 04:31:35 UTC+0000
..... 0x81f14938:ipconfig.exe        304   968    0  ----- 2011-06-03 04:31:35 UTC+0000
.... 0x822843e8:svchost.exe         1032  668    61  1169 2010-10-29 17:08:55 UTC+0000
..... 0x822b9a10:wuaclt.exe           976  1032     3   133 2010-10-29 17:12:03 UTC+0000
..... 0x820ecc10:wscntfy.exe         2040  1032     1    28 2010-10-29 17:11:49 UTC+0000
.... 0x81e61da0:svchost.exe          940   668    13   312 2010-10-29 17:08:55 UTC+0000
.... 0x81db8da0:svchost.exe          856   668    17   193 2010-10-29 17:08:55 UTC+0000
..... 0x81fa5390:wmiprvse.exe         1872  856     5   134 2011-06-03 04:25:58 UTC+0000
.... 0x821a0568:VMUpgradeHelper     1816  668     3    96 2010-10-29 17:09:08 UTC+0000
.... 0x81fee8b0:spoolsv.exe          1412  668    10   118 2010-10-29 17:08:56 UTC+0000
.... 0x81ff7020:svchost.exe          1200  668    14   197 2010-10-29 17:08:55 UTC+0000
.... 0x81c47c00:lsass.exe            1928  668     4    65 2011-06-03 04:26:55 UTC+0000
.... 0x81e18b28:svchost.exe          1080  668     5    80 2010-10-29 17:08:55 UTC+0000
.... 0x8205ada0:alg.exe               188   668     6   107 2010-10-29 17:09:09 UTC+0000
.... 0x823315d8:vmacthlp.exe          844   668     1    25 2010-10-29 17:08:55 UTC+0000
.... 0x81e0eda0:jqs.exe              1580  668     5   148 2010-10-29 17:09:05 UTC+0000
.... 0x81c498c8:lsass.exe            868   668     2    23 2011-06-03 04:26:55 UTC+0000
.... 0x82279998:imapi.exe            756   668     4   116 2010-10-29 17:11:54 UTC+0000
... 0x81e70020:lsass.exe             680   624    19   342 2010-10-29 17:08:54 UTC+0000
0x820ec7e8:explorer.exe            1196  1728    16   582 2010-10-29 17:11:49 UTC+0000
. 0x81c543a0:Procmon.exe             660  1196    13   189 2011-06-03 04:25:56 UTC+0000
. 0x81e86978:TSVNCache.exe           324  1196     7    54 2010-10-29 17:11:49 UTC+0000
. 0x81e6b660:VMwareUser.exe         1356  1196     9   251 2010-10-29 17:11:50 UTC+0000
. 0x8210d478:jusched.exe             1712  1196     1    26 2010-10-29 17:11:50 UTC+0000
. 0x81fc5da0:VMwareTray.exe         1912  1196     1    50 2010-10-29 17:11:50 UTC+0000
```

A normal Windows XP installation has **just one instance of lsass.exe** that the Winlogon process creates when the system boots. However, the process tree reveals that the two new lsass.exe instances were both created by Services.exe.

Q1: Please write down the process IDs (PIDs) for the fake lsass.exe? (1 Point)

Step 2: In fact, Stuxnet uses **Hollow Process Injection** to switch the code of a legit running process with malicious code.



As a result of being hollowed, the virtual address descriptor (VAD) characteristics for the region are drastically different. Only the legitimate one still has a copy of the lsass.exe file mapped into the region.

Please type `vol.py -f stuxnet.vmem vadinfo -p 1928,868,680 -`
`-addr=0x1000000`

```
[student@archlinux ~]$ vol.py -f stuxnet.vmem vadinfo -p 1928,868,680 --addr=0x1000000
Volatility Foundation Volatility Framework 2.6.1
*****
Pid: 680
VAD node @ 0x81db03c0 Start 0x01000000 End 0x01005fff Tag Vad
Flags: CommitCharge: 1, ImageMap: 1, Protection: 7
Protection: PAGE_EXECUTE_WRITECOPY
ControlArea @823e4008 Segment e1735398
NumberOfSectionReferences: 3 NumberOfPfnReferences: 4
NumberOfMappedViews: 1 NumberOfUserReferences: 4
Control Flags: Accessed: 1, File: 1, HadUserReference: 1, Image: 1
FileObject @82230120, Name: \Device\HarddiskVolume1\WINDOWS\system32\lsass.exe
First prototype PTE: e17353d8 Last contiguous PTE: ffffffff
Flags2: Inherit: 1

*****
Pid: 868
VAD node @ 0x81f1ef08 Start 0x01000000 End 0x01005fff Tag Vad
Flags: CommitCharge: 2, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
ControlArea @81fbbee0 Segment e24b4c10
NumberOfSectionReferences: 1 NumberOfPfnReferences: 0
NumberOfMappedViews: 1 NumberOfUserReferences: 2
Control Flags: Commit: 1, HadUserReference: 1
First prototype PTE: e24b4c50 Last contiguous PTE: e24b4c78
Flags2: Inherit: 1

*****
Pid: 1928
VAD node @ 0x82086d40 Start 0x01000000 End 0x01005fff Tag Vad
Flags: CommitCharge: 2, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
ControlArea @81ff33e0 Segment e2343888
NumberOfSectionReferences: 1 NumberOfPfnReferences: 0
NumberOfMappedViews: 1 NumberOfUserReferences: 2
Control Flags: Commit: 1, HadUserReference: 1
First prototype PTE: e23438c8 Last contiguous PTE: e23438f0
Flags2: Inherit: 1
```

Q2: Please find the difference of between the legit lsass.exe VAD and the fake one and write down the differences(1 Point)

Target 2: API Hooking:

Based on Symantec report ([Link](#), page 13), Stuxnet has hooked Ntdll.dll to monitor for requests to load specially crafted file names. These specially crafted filenames are mapped to another location instead — a location specified by Stuxnet.

The functions hooked for this purpose in Ntdll.dll are:

- ZwMapViewOfSection
- ZwCreateSection
- ZwOpenFile
- ZwCloseFile
- ZwQueryAttributesFile
- ZwQuerySection"

Step 1:

Type `vol.py -f stuxnet.vmem apihooks`, it'll shows all hooked APIs. I attached the result for ZWQuerySection Function.

```
*****
Hook mode: Usermode
Hook type: NT Syscall
Process: 940 (svchost.exe)
Victim module: ntdll.dll (0x7c900000 - 0x7c9af000)
Function: ZwOpenFile
Hook address: 0x7c90004c
Hooking module: ntdll.dll

Disassembly(0):
0x7c90d580 b874000000    MOV EAX, 0x74
0x7c90d585 ba4c00907c    MOV EDX, 0x7c90004c
0x7c90d58a ffd2             CALL EDX
0x7c90d58c c21800         RET 0x18
0x7c90d58f 90             NOP
0x7c90d590 b875000000    MOV EAX, 0x75
0x7c90d595 ba             DB 0xba
0x7c90d596 0003          ADD [EBX], AL

Disassembly(1):
0x7c90004c b202          MOV DL, 0x2
0x7c90004e eb0c          JMP 0x7c90005c
0x7c900050 b203          MOV DL, 0x3
0x7c900052 eb08          JMP 0x7c90005c
0x7c900054 b204          MOV DL, 0x4
0x7c900056 eb04          JMP 0x7c90005c
0x7c900058 b205          MOV DL, 0x5
0x7c90005a eb00          JMP 0x7c90005c
0x7c90005c 52           PUSH EDX
0x7c90005d e804000000    CALL 0x7c900066
0x7c900062 f2           DB 0xf2
0x7c900063 00           DB 0x0
```

API Hook Information for ZWQuerySection Function

For `ZwOpenFile` function, the hook address is `0x7c90004c`. Stuxnet uses the "syscall" hooking technique instead of the more common Inline/IAT/EAT hooking. To interactively explore code around the hook address, use the `volshell` command. This time we'll use it to follow the flow of execution when a hooked API is called.

First you need to break into the shell and switch into the context of a process that has been hooked. Then navigate to the hooked API. I'll start with `ZWQuerySection` which is at `0x7c90004c`.

Step 2:

Type `vol.py -f stuxnet.vmem volshell`

Inside volshell, Type `cc(pid=940)`

Type `dis(0x7c90004c)`

```
[student@archlinux ~]$ vol.py -f stuxnet.vmem volshell
Volatility Foundation Volatility Framework 2.6.1
Current context: System @ 0x823c8830, pid=4, ppid=0 DTB=0x319000
Welcome to volshell! Current memory image is:
file:///home/student/stuxnet.vmem
To get help, type 'hh()'
>>> cc(pid=940)
Current context: svchost.exe @ 0x81e61da0, pid=940, ppid=668 DTB=0xa940100
>>> dis(0x7c90004c)
0x7c90004c b202          MOV DL, 0x2
0x7c90004e eb0c          JMP 0x7c90005c
0x7c900050 b203          MOV DL, 0x3
0x7c900052 eb08          JMP 0x7c90005c
0x7c900054 b204          MOV DL, 0x4
0x7c900056 eb04          JMP 0x7c90005c
0x7c900058 b205          MOV DL, 0x5
0x7c90005a eb00          JMP 0x7c90005c
0x7c90005c 52            PUSH EDX
0x7c90005d e804000000    CALL 0x7c900066
0x7c900062 f200bf005aff22 ADD [EDI+0x22f15a00], BH
0x7c900069 696e20444f5320 IMUL EBP, [ESI+0x20], 0x20534f44
0x7c900070 6d            INS DWORD [ES:EDI], DX
0x7c900071 6f            OUTS DX, DWORD [ESI]
0x7c900072 64652e0d0d0a2400 OR EAX, 0x240a0d
0x7c90007a 0000          ADD [EAX], AL
0x7c90007c 0000          ADD [EAX], AL
0x7c90007e 0000          ADD [EAX], AL
0x7c900080 084063        OR [EAX+0x63], AL
0x7c900083 fe4c210d      DEC BYTE [ECX+0xd]
0x7c900087 ad            LODSD
0x7c900088 4c            DEC ESP
```

```

0x7c900089 210dad4c210d    AND [0xd214cad], ECX
0x7c90008f ad                LODSD
0x7c900090 8f                DB 0x8f
0x7c900091 2e51            PUSH ECX
0x7c900093 ad                LODSD
0x7c900094 4d                DEC EBP
0x7c900095 210dad8f2e53    AND [0x532e8fad], ECX
0x7c90009b ad                LODSD
0x7c90009c 4d                DEC EBP
0x7c90009d 210dad8f2e02    AND [0x22e8fad], ECX
0x7c9000a3 ad                LODSD
0x7c9000a4 0c21            OR AL, 0x21
0x7c9000a6 0dad8f2e6d    OR EAX, 0x6d2e8fad
0x7c9000ab ad                LODSD
0x7c9000ac 4f                DEC EDI
0x7c9000ad 210dad8f2e52    AND [0x522e8fad], ECX
0x7c9000b3 ad                LODSD
0x7c9000b4 9d                POPF
0x7c9000b5 210dad8f2e57    AND [0x572e8fad], ECX
0x7c9000bb ad                LODSD
0x7c9000bc 4d                DEC EBP
0x7c9000bd 210dad526963    AND [0x636952ad], ECX
0x7c9000c3 684c210dad    PUSH DWORD 0xad0d214c
0x7c9000c8 0000            ADD [EAX], AL
0x7c9000ca 0000            ADD [EAX], AL

```

At `0x7c90005d` there is a CALL to `0x7c900066`. But according to the disassembly, `0x7c900066` is in the middle of the instruction that starts at `0x7c900062`. This is an anti-disassembling trick that Stuxnet uses.

Let's disassemble `0x7c900066`

Type `dis(0x7c900066)`

```

>>> dis(0x7c900066)
0x7c900066 5a                POP EDX
0x7c900067 ff22            JMP DWORD [EDX]

```

The first two lines shows `POP EDX; JMP DWORD [EDX]`. Stuxnet plays a little tricks here:

1. When the CALL at `0x7c90005d` is executed, its return address (`0x7c900062`) is pushed onto the stack.
2. The POP EDX instruction at `0x7c900066` then removes that value from the stack and places it in EDX.
3. At `0x7c900067`, EDX is dereferenced and called. So the pointer being

dereferenced is stored in `0x7c900062`.

Q3: Please draw three figures with stack and EDX register to show the little tricks that Stuxnet did. Label it with 1,2,3 (6 Points)

At address `0x7c900062`, the first 4 byte of data is being used as the memory address of the next hop (`JMP DWORD [EDX]`) which I highlighted in the following picture (`f200bf00`)

```
0x7c900062 f200bf00 5aff22 ADD [EDI+0x22ff5a00], BH
```

Q4: This value is stored using the little endian format, please convert this value back and write down the actual memory address. (1 point)

Step 3:

Let's disassemble the memory address that contains the rootkit (the answer for Q4, you need to convert `f200bf00` to get the address).

Type `dis(MEMORY_ADDRESS_FROM_Q4)`

It shows the assembly code of the rootkit.

```

>>> dis(1-11-11)
5a POP EDX
84d2 TEST DL, DL
7425 JZ 0xbf011c
feca DEC DL
0f8482000000 JZ 0xbf0181
feca DEC DL
0f84bb000000 JZ 0xbf01c2
feca DEC DL
0f84fe000000 JZ 0xbf020d
feca DEC DL
0f8440010000 JZ 0xbf0257
e98c010000 JMP 0xbf02a8
e8f9010000 CALL 0xbf031a
85d2 TEST EDX, EDX
7413 JZ 0xbf0138
52 PUSH EDX
8b5208 MOV EDX, [EDX+0x8]
3b54240c CMP EDX, [ESP+0xc]
7508 JNZ 0xbf0137
c744243040000000 MOV DWORD [ESP+0x30], 0x40
5a POP EDX
52 PUSH EDX
e81e020000 CALL 0xbf035c
837a0400 CMP DWORD [EDX+0x4], 0x0
7509 JNZ 0xbf014d
5a POP EDX
8d542408 LEA EDX, [ESP+0x8]
cd2e INT 0x2e
eb0c JMP 0xbf0159
5a POP EDX
8d542408 LEA EDX, [ESP+0x8]
64ff15c000000000 CALL DWORD [FS:0xc0]
85c0 TEST EAX, EAX
7523 JNZ 0xbf0180
e8b8010000 CALL 0xbf031a
85d2 TEST EDX, EDX
7418 JZ 0xbf017e
8b5208 MOV EDX, [EDX+0x8]
3b542408 CMP EDX, [ESP+0x8]
750f JNZ 0xbf017e
8b DB 0x8b
54 PUSH ESP
24 DB 0x24

```

The instructions highlighted show how the malware eventually calls the requested system service. It uses the IDT instead of the SSDT. Although Windows itself doesn't use the IDT for system service dispatching anymore

(that stopped with Windows 2000), the IDT was still kept around for backward capability.

Check slides (ch14.pptx, ch15.pptx) answer the following questions:

Q5: What's the meaning of `INT 0x2e`? (1 point)

Q6: What's IDT? What's SSDT? (2 points)

Step 4:

Q7: Find the real memory address of the rest of system API

Repeat Step 1 and 2, and reveal the real memory address for

* ZwCreateSection

* ZwOpenFile

* ZwCloseFile

* ZwQueryAttributesFile

(4 points)

Target 3: Kernel Forensics (Bonus 6 points):

Stuxnet loads two modules: maxnet.sys and mrxcls.sys. The first one installs a file system registration change callback to receive notification when new files systems become available (so it can immediately spread or hide files). The second one installs an image load callback, which is used to inject code into processes when they try to load other DLL.

Please use Volatility, read the command reference ([Link](#)). Find a way to find the malicious kernel drivers, kernel callbacks and point out the malicious devices inside memory.

Deliverables:

- A detailed project report (**lab4_report.pdf**) in **PDF format** to describe what you have done, including screenshots and code snippets.

Submission

- Check lab due date on the course website. Late submission will not be accepted.
- The assignment should be submitted to D2L directly.
- No copy or cheating is tolerated. If your work is based on others', please

give clear attribution. Otherwise, you **WILL FAIL** this course.