Static and Dynamic Analysis of BINARY.EXE & WINHLP.EXE

Andrew McConnell

George Mason University

4400 University Dr, Fairfax, VA 22030, United States

[amcconn@gmu.edu](mailto:amcconn@gmu.edu)

*Abstract*— This paper serves to explore two malware samples down to the barebone assembly. I was given six malware samples and of them, I wrote about our static and dynamic analysis about two of them, as well as a debugging analysis with IDA. Static and dynamic analyses are straight forward, but in possibility analysis, we will be looking at any vulnerabilities the malware might be targeting. The research was conducted on a Windows XP VMware virtual machine.

Keywords—malware, static analysis, dynamic analysis, IDA Pro, Ghidra, ANGR, debugging, CYSE 480, malware analysis, reverse engineering, Windows XP, Linux, Ubuntu

# Introduction

The malware samples that were chosen were binary.exe and winhlp.exe. There will be an introductory section to each malware, and any important structure or flow charts will be listed or referenced. The organization of these two malware samples will be organized differently. For dynamic analysis, the tools used include but aren’t limited to Process Monitor and Wireshark. The tools used in the static analysis of the files were IDA Free 5.0 [1], Ghidra [2], and the Python library ANGR [3]. This section will also contain any information obtained through the debugging capability of IDA and ANGR. The reason that the sections will differ between the malware samples is because there was a major difference in the analysis of each. Binary.exe’s dynamic analysis helped structure the static analysis, but it was vice versa winhlp.exe’s case. The debugging section will follow the static analysis. Finally, the last analysis section will be for any vulnerabilities found in researching any included DLLs or items of that nature. At the end of these three sections, there will be a conclusion and a short section on the severity of the malware being analyzed.

# Explanation of Tools

In addition to the main assembly analysis + debugging tool we learned in class (IDA) I also used Ghidra and ANGR in my static analysis. Ghidra is a tool developed by the NSA and is completely open source and free to use. Up until recently, it was nearly a mirror image of IDA in terms of capability minus the capability to debug, but this was added to Ghidra within the past year. Ghidra also provides a deconstructor window, where it shows the assembly as C pseudocode. I used this mainly to check if my understanding of the assembly output was correct.

ANGR is a python library for analyzing binaries. Going into depth on its capability could warrant a separate five-to-ten-page paper in and of itself, so to keep things short, it is a way to symbolically execute a binary. You can store the separate states as variables and walk down different branches of logic (CMP, JMP statements). It’s a popular tool to use while competing in CTFs, which admittedly, doesn’t have very similar goals to analyzing a malware sample. This tool allows easy access to the register values in any given states without having to enter a debugging mode, which is required in IDA. This tool is mainly a separate way to view the same data in Ghidra & IDA, but I chose it to learn something new. There isn’t much on the data output from ANGR since I didn’t use it that much, but I did spend some time learning about it. The output from Ghidra and IDA were far more useful for this project.

I feel a little more agile on a Linux machine, so I do offload the malware samples onto an Ubuntu VM to use some tools on it there too. The tools I used there were Ghidra and IDA like in Windows; this is where I ran some ANGR analysis, and the tools objdump [4], exiftool [5], and Detect it Easy [6] function similarly to PEiD.

# Binary.Exe

Initial observations of this binary were met with nearly nothing. I chose to use some dynamic analysis methods first, mainly dealing with network traffic analysis. Wireshark and TCP view came up with nothing at first, and the only thing I noticed was that binary.exe would delete itself while running. After this I opened it up in IDA and Ghidra, but first we’ll explore the other dynamic methods I took that found what this malware was trying to accomplish.

## Dynamic Analysis

The tool used for the most successful dynamic analysis of binary.exe was Process Monitor [7]. Upon running, I could see that binary.exe used a lot of CPU power until it promptly deleted itself. Process Monitor revealed that binary.exe tries to access bogus directories and/or files in its own directory. This is done to bury its true actions. The bogus files/directories have names that look like “CNiqn”, “pAJsSnPC” and the like. It loops through around eight to ten of these different names. They can be seen in the output of the Strings tool. Scattered throughout these file accesses, are functions that could be malicious, and some of them give us clues as to where to go in the static analysis. Right before checking these bogus directories, it checks the hex addresses in its own file: 0x23400, 0x2B000, 0x4400, and 0x400. These could be beneficial to explore these in the static analysis of binary.exe. There are three more offsets that are read later, and they are 0xC400, 0x14400, and 0x1C400. The first four that are read before the bogus directory access loop will be referred to as Group A and the second group will be referred to as Group B. The image base of the program is 0x400000.

In between the access of Groups A and B, it opens and seems to modify adsldpc.dll, and reads wldap32.dll. Adsldpc.dll, or Active Directory Services Lightweight Directory Access Provider C is a library on windows, which is used for directory management over a network. Examples of this in use would be for distributed computer systems and printer management [8]. I’ve seen also in [9] that adsldpc.dll can create new records in Windows registry, which is why this library might be targeted by the malware. There will be more on my speculation as to what this malware does with adsldpc.dll later on in the analysis.

There are a couple things that binary.exe tries to do at the very end of its execution. Firstly, it looks like it tries and fails to take control of the DeviceIOControl [10] of C:\WINDOWS. This function sends control code directly to device drivers, which could be to obtain camera footage, which is common for hackers to do to start their reconnaissance, or maybe try to listen in from the speakers. It then creates a copy of itself in its directory, and then creates a temporary BAT file in \Local Settings\Temp. Upon inspection, it was not in the file, so a strategy for static analysis could be to find the breakpoint location to place so that we can view this BAT file before it gets deleted. This will be referenced as TBAT file in static analysis. After this file is created, binary.exe accesses a few more DLLs, but the part that stuck out to me was that there was a buffer overflow result on a QueryNameInformationFile on cmd.exe. This function returns various aspects of the file seen in [11]. The buffer overflow might be to access information about the file that only a process that has privilege to open it can view. I’m not sure if this elevates the command line to be run as Administrator or not, but it could be a point to investigate more. The reason I think it might be trying to elevate privileges is because right after it runs the TBAT file from cmd.exe. That is where its execution ends. It doesn’t say when it deletes itself, it only says that it closes itself. I think that the TBAT file is what contains the instructions to delete binary.exe. Like I mentioned before, the dynamic analysis of this file provided a few avenues to take in static analysis.

Also, towards the end of the execution of binary.exe, I saw that there was another buffer overflow result of a QueryNameInformationFile operation of cmd.exe. This could be to escalate to administrator privileges. Or for the same reasons listed for the last buffer overflow.

As for any networking, binary.exe seems to be using IRC to communicate with a “vililam-grea.ru.hsd1.va.comcast.net” and an IP address: “67.218.118.62”. This just seems to be an exfiltration method to gather data and send it to the adversary.

## Static Analysis

I started to go through the assembly line by line until I reached a subroutine call that was nearly 2KB in size. Before I got to that point, I noticed one thing that looked promising. There was a MOV in the assembly that put the KiFastSystemCallRet instruction from ntdll.dll into a local variable. Ntdll.dll is a library that contains kernel level functions. Microsoft has a documented vulnerability with this DLL (ms03-007) which says that it can be used for remote code execution. We can tell that since there’s no network traffic resulting from running binary.exe that perhaps the created script TBAT is the code that it’s trying to execute. This could also be a red herring, but something to keep an eye on since this is one of the most critical forms of exploitation.

Starting with Group A, I went to offset 0x400400 and found nothing, there’s only question marks, which means there is no physical data on the disk at this location. The next address, 0x404400, is at the start of a logic branch called loc\_4043F8, which contains a call to SetCurrentDirectoryA [12]. It seems to use EAX to store the directory it wants to go to, which might be a garbage value (like in the directory loop found in dynamic analysis), but it could also be a legitimate address. We could add a breakpoint around this area of code and find out what values it comes out with. It appears like the hex at the 0x42b000 address wasn’t disassembled or it’s storage for variables, the same applies to 0x423400. The variable at 0x42b000 is 0x57880C78. I can’t find the variable at 0x423400, but the 4 bytes that are at that location are 0x6F00003C.

Now looking into Group B, all the addresses are variables. 0x40C400 contains the variable 0x94A10900; 0x414400 contains 0x6D00D179; and 0x41C400 contains 0x3BB00020. These may or may not be useful, the ascii conversion of all the variables doesn’t reveal anything. The best debugging point to look at is the 0x404400 address.

Digging further through the assembly, something that sticks out is that it grabs the locale (the general area) that is registered to your computer, it actually does this more than once. This could be for some general information for the hacker or hacker group to optimize the malware for the specific area; certain areas of the world tend to be more aware of malware than others. This could also be used somewhere else in the code that didn’t stick out to me. Researching this function, it is called in malware to reveal the language used. I couldn’t make out where it compares so I’m not sure what languages its checking for. Even if I did know, it wouldn’t help too much in understanding the malware’s operation. The locale it checks for is hexadecimal 0x6B7, which doesn’t have a corresponding language tied to it, so it could be another obfuscation technique.

Further into the program it seems like it uses the VirtualUnlock command a few times [13], which is used to unlock pages of virtual memory. This might be to breach kernel memory or the memory of another process. Binary.exe could try to inject itself into another program.

The next imported function used is called CopyFileExA [14]. This copies an existing file to a new file. While I thought this could involve the TBAT file, this function was called many times along with the VirtualUnlock, SetCurrentDirectoryA, and GetUserDefaultLCID (the locale grabber) functions.

Later in my static analysis, I realized that the library adsldpc.dll is loaded in towards the middle of the program through a LoadLibraryA [15] call procedure. This library must be of some importance since it isn’t loaded at the beginning. I also found the list of actual functions which had escaped me before. In this, I found a call to GetProcAddress [16], which probably finds a process to possibly inject itself into (with TBAT).

After using the tools: objdump, exiftool, and ANGR I didn’t find much more useful information to add or pursue in debugging analysis.

## Debug Analysis

I tried a few breakpoint positions, mainly trying to find the right place that would stop the program before running the TBAT program created towards the end of a binary.exe run. I’m pretty sure I found one of the two buffer overflows, but as soon as I started debugging closer to that point, IDA would begin to throw an error. I’ll go over this more in the video, but I was unsuccessful in retrieving the TBAT program. If given more time, I would try a few more approaches to try to capture this file because I think it contains concrete proof as to what the program is doing. I think that this malware is trying to dodge debugging techniques by throwing errors when I get closer to capturing this file, so I believe it is the most important aspect of this malware since they keep it hidden.

To explain my debugging process, I found that the bogus directory names were statically inside the program and viewable in the strings tool. It wasn’t necessarily a loop going through all these directories, there were just a bunch of what I’ll call “obfuscation blocks” in the assembly. The procedure calls were CopyFileExA and SetCurrentDirectoryA which showed up as CreateFile operations on Process Monitor. The sole intention of these processes was to drown the actual operation of the malware in Process Monitor. Some of the strings found looked like they could’ve been scrambled, for example: “\betP\vocAdd\tess” which looks somewhat like “\Local Settings\Temp” but only slightly.

Going back to strings, we see gibberish, but if we look at UTF-16 encoded strings (with the command “strings -n 6 -e l -t x binary.exe) we can see that it references process names and PIDs.

Text

Description automatically generated

Going to these offsets doesn’t reveal much unfortunately. This leads us to our Possibility Analysis and Conclusion.

## Possibility Analysis

I will start off with what I think is the most probable operation of this malware. Confirmed by the screenshot above, we know that binary.exe is heavily concerned with Processes and PIDs. So binary.exe starts off by opening all the libraries that aren’t adsldpc.dll, it opens this one later on. Then it goes into its obfuscation block loop. This clouds Process Monitor with a bunch needless information that somebody just skimming Process Monitor wouldn’t notice, but once you dig and look at what its actual doing, it’s obvious that it’s trying to divert from what it’s actually doing. It does a few small things in the middle of the obfuscation blocks, but the meat of the program takes place at the end. Right at the end the process execution, the main things it does is creates and writes out a temporary .BAT file, which is referred here as TBAT. It then has a BUFFER OVERFLOW result when opening cmd.exe, which research reveals that it is to sneak in extra executions when executing a program. This is probably to hide from antivirus programs, and sneak in the execution of TBAT. After this starts my speculation but I’m fairly certain that TBAT injects either itself or binary.exe into another legitimate process, and my best guess is that it tries to masquerade as adsldpc.dll. I think this is why it opens adsldpc.dll midway through the program and that Process Monitor records its operation as modifying adsldpc.dll. Another thing I noticed in the Process Monitor is that it creates its own registry keys. This is done to maintain persistence and it seems to be in Indonesian, which could lead us to the creators of the malware:

Text

Description automatically generated with low confidence

This is my best guess as to how this program works. In my research I found some other possible vulnerabilities this malware either had as a honeypot/coincidence or it genuinely tried to exploit. The biggest one was in ms03-007 [17], listed earlier. I think that because the VM doesn’t contain the component that is vulnerable in this instance, that it is incapable of being exploited. Ms03-007, if exploited, can allow for remote code execution, one of the most critical vulnerabilities. Windows XP is among the “Affected Software.” For a short window before Microsoft patched it, the vulnerability centered around the attack vector, WebDAV. Windows NT and Windows 2000 had Internet Information Services (IIS) installed by default, but it was not installed in Windows XP. The problem is that ntdll.dll contains an unchecked buffer, which an attacker could send a special HTTP request to a machine running IIS. This could cause the machine to fail or run code of the attacker’s choosing.

This could just be a honeypot or coincidence that the malware uses ntdll.dll for a different purpose, but it also could be trying and failing to execute code remotely on the system.

## Conclusion for binary.exe

Admittedly, it is frustrating that I was unable to capture the TBAT file for further analysis. I think it would’ve contained the evidence needed to classify this program’s intent and operation. But from its other actions we can tell that it is doing something malicious. The fact that I couldn’t dig deeper to find TBAT since the program began to throw errors is damning enough. Nothing about the program was changed except for breakpoints, which are all handled in IDA, not the program itself. The program was dodging reverse engineering and analysis by disabling itself while I was debugging. This is an action taken by malware. We can also view all the entries of directory accesses in Process Monitor which are obviously for the purpose of covering something up, which is suspicious in and of itself.

I would classify the severity of binary.exe, on a machine like the Windows XP virtual machine as very high. Let’s say a normal user runs this executable. They would notice a steep drop in performance, and then the file would then delete itself. If the user didn’t have a second copy of the file, or the means to obtain it again, they wouldn’t be able to send it to a professional to analyze it. Even if they did, the file protects itself from analysis. The computer runs fine after infection, but there’s obviously something going on in the background. Although I didn’t find exactly what it does, we know that it may be looking for vulnerable network components like printers or other computers to spread to, based on adsldpc.dll’s function. Pairing these malicious functions with the fact that binary.exe uses heavy obfuscation on not only its process, but its strings too, these are the reasons for my classification.

For YARA rules involving this malware, I would suggest including adsldpc.dll as one of the Booleans to look out for, as well as the directory names. If the scheme for the naming of the TBAT file is found, or any network function of binary.exe or the TBAT file is discovered, I would include that as well.

# Winhlp.exe

As a reminder, the structure of this malware’s analysis will be opposite binary.exe, the first section will be static analysis and the following section will be dynamic analysis. This file was broken to start off with. I tried to input it into PEiD and Detect it Easy (PEiD clone for linux) and it would say that it couldn’t identify an entrypoint nor could it identify its compiler, which was odd. Ghidra had problems with it too. I opened it up in Hexplorer and, though it took me a second, I realized that the magic number that started off the file was “NZ” instead of “MZ”. Correcting this, PEiD and Detect it Easy began to identify the information needed, like the entrypoint. To start off, let’s observe the output of strings.

## Static Analysis

It was a different scenario when it came to first looking into this malware compared to binary.exe. Right away, I saw strings like “keylogger” and “password”. Though I can’t classify it based on this evidence alone, it clued me into what could’ve been this malware’s function. I then hopped over to Ghidra to start looking at the disassembly and decompilation pseudocode.

In the entry/main function it branches off into 2 different functions, a local function and “GetMainArgs” [18]. This function is from crtdll.dll [19]: C RunTime DLL. It contains other functions like printf and memcpy, but what GetMainArgs does is essentially pulls arguments from the command line that are similar to those that are passed to a main function. These are argc (argument count), argv (argument values/list), and the third one listed in the function document was something called env, so I renamed the three right before this function to these three names. My initial thoughts on this function were that it was one of three things: redundant, obfuscation, or threading. It doesn’t look like the main function of this program even takes in these values, so it could just be Ghidra including this to grab the variables. From here I walked into the local function.

To start off, this function declared 2 variables, one of which stood out to me. It was a LPSTR variable [20], and looking at the Microsoft documentation, this is a pointer to an array of 8-bit characters, which MAY be terminated by a null character. At first, I thought that this might be where user input is stored if this was indeed a keylogger. But later on, I saw that this one is to get the current process name since the next function used was GetCommandLineA [21]. This returns a pointer to a command-line string for the current process, and it returns it into the LPSTR variable. It seems to format this string properly before passing it to another function. Before that though, there is a call to GetModuleHandleA [22], which I’m almost certain is an attempt to obfuscate. GetModuleHandleA returns the handle to a module specified by the parameter. The parameter included in this case was 0x0 which is “\0” or NULL. On top of this, the process up to this point hasn’t created a module, and only a parent process can use GetModuleHandleA to retrieve a child process/module. After this came the longer function which might be the keylogging part of the program.

This function contains the strings that stuck out to me like “keylogger”. Digging into the other local function calls here revealed that winhlp.exe creates registry keys and uses wsock32.dll, which deals with networking. This gives me some direction when I move to dynamic analysis. There were a few interesting string variables here that contained the values “wind0ws.exe” (that’s a zero), “password”, and “#help”. Doing some research into the .EXE string revealed that malware will usually disguise itself as some variation of “windows.exe” here’s an example website: [23]. Using this “wind0ws.exe” string, it seems to build a path in a 3-4 function deep call. It does this to avoid analysis, surely. I’m not sure if it’s building a process or if maybe this is simply a storage file named this way so that people won’t mess with it. I think that if this file is used, it will be evident as to what it’s used for in dynamic analysis.

Moving forward in the same local function, I saw that the malware gets the path to the system directory, which I’ve read is usually to just check for compatibility in some cases, but I believe it is for something malicious. After this, there’s a SetFileAttributesA which is passed a local variable, which I think is a file path. There are clues that it is of the form “%s\\%s” so 1-2 directories deep.

A picture containing diagram

Description automatically generated

The next function was alarming, it is the ShellExecuteA command. I read into the documentation and found that it has the following options: edit, explore, find, open, print, runas (administrator), and NULL. These have to do with either a file or directory. My best guess is that the program is most likely going to do the “runas” option, but it was difficult to figure out which one it was using.

This was about midway through the function and there were a few more local function calls so I glanced through them. One of them was 883 lines long of pseudocode so I knew that there was going to need to be some debug analysis to discover what some parts of the code were doing, instead of statically interpreting it.

I found the start of the networking portion of the code where it called WSAStartup, which is when they might start to send data back to the adversaries control server. Directly after this there was a local variable that contained a URL! The URL was irc.efnet.org, IRC being Internet Relay Chat. This protocol is used in other malware cases to send commands to a victim’s computer. It also is described to be very secure and hard to trace. I was wondering if this is how the attackers send commands to the ShellExecuteA function, but looking further in proceeding local functions, I could see there are a few options that are available to the attackers. The only one that is easily decipherable is a command to make the program idle for 5 hours until it inquires for a new command. The other ones involve more local functions and seem to be way more complex. From here, I thought that trying to figure out what the complex functions were trying to do were a waste of time and I moved into dynamic analysis.

## Dynamic Analysis

This malware doesn’t do a good job of hiding its operation. It is more geared to the more oblivious/normal users of computers. The method of hiding is heavily reliant on social engineering, which I will explain more when we get there. I used Process Monitor again mostly to view what winhlp.exe does during a successful run.

Upon running winhlp.exe, it creates a thread, so I wonder if the GetMainArgs discussed in the static analysis section does make a thread. I didn’t see anywhere else in the program that a thread is created, but the documentation doesn’t say that it deals with threading. I wonder if this is to capture keystrokes on one thread of the program and send the data out to a server with another. There is a CreateThread [24] function in the main keylogging function of the program, so this must be where it threads a new process. I haven’t heard about malware threading to avoid detection, so this might be for a specific process. Throughout the entire runtime, the program opens a whole bunch of different registry keys. I’m not entirely sure if this is malicious, or to look like it’s a legitimate program since it’s just opening them.

Some curious operations the program does is it sets the random seed of the Cryptography\RNG path. I’m almost certain this is to retrieve files after they’ve been encrypted after they change this. Perhaps the keylogging isn’t the primary method of data extraction. This malware might be outfitted to do some sort of ransomware attack. If they’re setting the seed, they might be preparing to encrypt the drive. This is purely speculation, but I wonder if they are trying to make it so they can use the same key for every victim’s computer’s encrypted contents. This saves on overhead, but if they keep using the same key, it can be broken easy.

Winhlp.exe then accesses the audio drivers of the system. It runs through 3-4 different audio drivers, which is alarming since this could mean that it tries to listen to you through any microphone connected to the system.

Towards the end of its runtime, it bounces around the file system, including my home directory. I ended up finding where it deposited the “wind0ws.exe” file, and I was able to extract it from the system32 folder. I ran a SHA-256 against it and winhlp.exe and it turned out that they were the same file. This is where the social engineering comes into play, and it’s an interesting way to maintain persistence. This malware is named something innocent, and seemingly trustworthy “winhlp.exe”, it sounds like a Windows help file. When it’s run and nothing happens, the user is going to probably get frustrated with the file and won’t run it again. The file is already running in the background, but for whatever reason it shut down, there’s another file that looks harmless called “wind0ws.exe” that might be run by the user.

One last thing that I noticed was that there was another file aside from “wind0ws.exe” that winhlp.exe created. It was created in the C:\WINDOWS\Prefetch\ directory and its name was WINHLP.EXE-13070424.pf. This must be an automatically generated name since it was nowhere found in the strings output or in the Ghidra analysis. Doing some research, I found out that PF files are used by Windows XP for start-up processes. PF files contain the instructions for trace files for start-up. As this isn’t a PE file, we can’t analyze it in IDA or PEiD, so we might be relying on the strings output.

Using the custom strings command (strings -n 6 -e l -t x WINHLP.EXE-13070424.pf), we can see that it lists a lot of directories, including winhlp.exe’s full path as well as wind0ws.exe. It also lists a lot of the same DLLs that I saw in Process Monitor and Ghidra Analysis.





I’m thinking that all these DLLs are loaded along with the malware files, and if the winhlp.exe file is removed, the wind0ws.exe is loaded so that the malware stays persistent.

Moving on to Wireshark, I already knew that it was contacting an irc.efnet.org, so I expected to see network packets involving that. Not only did I see that exact URL, but I also saw it requested to communicate with a “NICK” user. This must be the puppet account for the hackers to send and receive information from. There were two types of command responses from “NICK”, which tells me that the account must be automated to some extent. The first command type that it sends a few times is “NOTICE”. I didn’t see this anywhere in my static analysis, but it might be one of the more complicated commands that had a few local functions. The next type was very interesting, it was “No Takeovers… No Whiners”. When I first saw this, I thought that it may be some sort of “out-of-commission” message the hackers left after they stopped supporting the malware. It is vague though, so I can’t be super sure.

## Debug Analysis

All the data gathered in static and dynamic analysis was pretty much enough to classify this malware, so I did not spend much time on the debugging process for winhlp.exe. As a result, this section will be much shorter than the others. I still wanted to find where the keylogger was outputting to, since I still had the file path indicator (“%s\\%s”).

I put some breakpoints on some offsets for an offset “SubKey” and a random text offset with the text: “tsm~äqà{òë¿Ñ++++++¦¦¦+¦+a¯-¦OOfpn”. This might be encrypted or encoded somehow. This file is difficult to debug the same way binary.exe was difficult. It kept returning without going to specified breakpoints and throwing errors. Looking back into Process Monitor, it could be the created file “wind0ws.exe.Manifest”, though I can’t access that file since it must get deleted after a full run through of the program. I don’t think it’s necessary to find this file, though it would’ve been nice.

## Conclusion for winhlp.exe

I’m forgoing a possibility analysis since I’m already certain of this malware’s operation. Winhlp.exe is a keylogger that utilizes IRC to receive commands from and send the logged information to the malware authors/leasers. It utilizes a unique method of social engineering to maintain persistence in plain sight. Its own name is something that could be seen as harmless, and it copies itself into a differently named file “wind0ws.exe” and places it onto the system directory. Even if a normal user becomes suspicious and deletes winhelp.exe, the second program which also looks harmless will persist and probably end up capturing sensitive information from the user.

One other interesting thing to keep an eye on is that it changes the Cryptography seed, which could be a sign of ransomware. I’m not sure why, but they might be trying to limit their overhead by encrypting every victim’s machine with the same key. This is a ‘vulnerability’ against this malware since it doesn’t hide that it does this well. It might be very easy to reverse engineer a key if a cryptanalyst knew the method of encryption (AES, DES, etc.).

This malware was much easier to statically analyze than binary.exe, but it still employed some tactics to avoid analysis. I think some of the functions I saw in IDA were to avoid debugging, some of them being sleep and timeout functions. It had function branches that exited early, which makes it hard to analyze and find the path to the keylogging file.

To score this malware, I’d score it just under binary.exe. While it is still dangerous and can possibly extract sensitive information, this is variable in nature. Some people might not be using their computer for banking, and therefore won’t be typing any of that information in. It can capture passwords and similar items, which is usually shared between websites, at least in the days of Windows XP. For the cryptography seed changing, it looked like it might’ve been in production to make this malware have ransomware capabilities, especially since the “NICK” character was able to transmit controls to the victim computer.

For Yara rules classifying winhlp.exe, I would include the “wind0ws.exe” as one of the strings, as well as “keylogger”, and the website “irc.efnet.org”. Especially with the last one, the hackers would have to change platforms to dodge this classification. If you also could find an IP address for efnet.org, or possibly one for the user “NICK” that could be an easy way to block this malware.

CONCLUSION

The purpose of this project was to use the tool kit we acquired in class to confidently reverse engineer and analyze two malware samples to understand their operation and impact on a target machine. The first sample, named binary.exe in the provided sample set, seemed to be injecting or masquerading itself as adsldpc.dll, which is convenient since that DLL is for communication across a network, allowing it to propagate. There was a possibility that it tried to exploit ms03-007, which allows for remote code execution. The second malware sample, winhlp.exe, is a keylogger that sends and receives messages from an IRC domain. This one was more reliant on social engineering for persistence.

Overall, I wish there was some more time given so that I could’ve extracted the TBAT file from binary.exe’s runtime. Also I mentioned that I would be using ANGR, and while I did use it some, I wasn’t able to use it to its full ability since I still need to learn it. It would’ve helped extract the keylogger file from winhlp.exe since you can feed it a target offset and tell the Simulation Manager to follow the program until it gets to that point, and it can spit out the keylogger file path. I wasn’t sure how to script it at that point, so I was unable to retrieve the file. I would recommend a lab for future classes on ANGR since it is heavily used in CTF challenges.

Though I couldn’t find the TBAT or keylogger files, I do feel like I confidently classified and recognized the full capability of both malware samples, moreso winhlp.exe.

##### Acknowledgment

I feel as if I got more out of this class than most others. I’d been curious about Yara and assembly language for a while, and it was great to finally be able to understand what they were. Thank you Dr. Gebril for the opportunity!

##### References

1. Ida Freeware. HexRays Blog. (n.d.). Retrieved December 6, 2021, from https://hex-rays.com/ida-free/.
2. Ghidra. (n.d.). Retrieved December 6, 2021, from https://ghidra-sre.org/.
3. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., &amp; Vigna, G. (2016). Sok: (state of) The art of war: Offensive techniques in binary analysis. 2016 IEEE Symposium on Security and Privacy (SP). https://doi.org/10.1109/sp.2016.17
4. Objdump. The GNU Binary Utilities - objdump. (n.d.). Retrieved December 6, 2021, from https://web.mit.edu/gnu/doc/html/binutils\_5.html.
5. exiftool Application Documentation. (n.d.). Retrieved December 6, 2021, from http://web.mit.edu/Graphics/src/Image-ExifTool-6.99/html/exiftool\_pod.html.
6. Horsicq. (n.d.). Horsicq/detect-it-easy: Program for determining types of files for windows, linux and macos. GitHub. Retrieved December 6, 2021, from https://github.com/horsicq/Detect-It-Easy.
7. Russinovich, M. (2021, October 12). *Process Monitor - Windows Sysinternals*. Process Monitor - Windows Sysinternals | Microsoft Docs. Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon>.
8. Beare, M. (n.d.). *Process search*. ReviverSoft. Retrieved December 6, 2021, from <https://www.reviversoft.com/en/processes/adsldpc.dll?ncr=1>.
9. *Adsldpc.dll - what is adsldpc.dll? - ADS LDAP provider C DLL*. File Inspect Library. (n.d.). Retrieved December 6, 2021, from <https://www.fileinspect.com/fileinfo/adsldpc-dll/>.
10. Karl-Bridge-Microsoft. (2021, October 13). *DeviceIoControl function (ioapiset.h) - win32 apps*. Win32 apps | Microsoft Docs. Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol>.
11. (2021, October 21). *NtQueryInformationFile function (ntifs.h) - windows drivers*. Windows drivers | Microsoft Docs. Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntqueryinformationfile>.
12. *Setcurrentdirectory function (winbase.h) - win32 apps*. Win32 apps | Microsoft Docs. (2021, October 13). Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setcurrentdirectory>.
13. Karl-Bridge-Microsoft. (2021, October 13). *Virtualunlock function (memoryapi.h) - win32 apps*. Win32 apps | Microsoft Docs. Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualunlock>.
14. GrantMeStrength. (2021, October 13). Copyfileexa function (winbase.h) - win32 apps. Win32 apps | Microsoft Docs. Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-copyfileexa>.
15. Karl-Bridge-Microsoft. (2021, October 13). Loadlibrarya function (libloaderapi.h) - win32 apps. Win32 apps | Microsoft Docs. Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>.
16. GetProcAddress function (libloaderapi.h) - win32 apps. Win32 apps | Microsoft Docs. (2021, October 13). Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress>.
17. MS03-007. packet storm. (2003, March 17). Retrieved December 6, 2021, from <https://packetstormsecurity.com/files/30919/ms03-007.html>.
18. TylerMSFT. (2021, August 3). \_\_getmainargs, \_\_wgetmainargs. Microsoft Docs. Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/cpp/c-runtime-library/getmainargs-wgetmainargs?view=msvc-170>.
19. Use the C run-time - visual C++. Use the C Run-time - Visual C++ | Microsoft Docs. (2021, November 16). Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/troubleshoot/cpp/use-c-run-time>.
20. [MS-DTYP]: LPSTR. [MS-DTYP]: LPSTR | Microsoft Docs. (2020, March 30). Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/3f6cc0e2-1303-4088-a26b-fb9582f29197>.
21. *Getcommandlinea function (processenv.h) - win32 apps*. Win32 apps | Microsoft Docs. (2021, June 29). Retrieved December 6, 2021, from https://docs.microsoft.com/en-us/windows/win32/api/processenv/nf-processenv-getcommandlinea.
22. GetModuleHandleA function (libloaderapi.h) - win32 apps. Win32 apps | Microsoft Docs. (2021, October 13). Retrieved December 6, 2021, from https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea.
23. WIND0WS.EXE Information . BleepingComputer. (n.d.). Retrieved December 6, 2021, from <https://www.bleepingcomputer.com/startups/WIND0WS.exe-6368.html>.
24. CreateThread function (processthreadsapi.h) - win32 apps. Win32 apps | Microsoft Docs. (2021, October 13). Retrieved December 6, 2021, from <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread>.