Analysing malware samples
Coursework 2

Pablo Collado Soto
19134031

INDEX

1. INTRODUCTION 4

1.2 Importing the VM 4

1.3 Installing the Guest Additions 4

1.4 Configuring the Network 5

1.5 Showing the hidden files 6

1.6 Stopping automatic updates 7

1.7 Regarding the firewall 7

1.8 Communicating with the outside world: shared folders 8

1.9 Getting the tools 9

1.10 Better safe than sorry 11

1.11 Final notes 12

2. THE FIRST MALWARE SAMPLE 13

2.1 Malware study 13

2.2 Removal process 23

2.3 Summing up 24

3. THE SECOND MALWARE SAMPLE 25

3.1 Malware study 25

3.2 Removal process 33

3.3 Summing up 34

4. THE THIRD MALWARE SAMPLE 35

4.1 Malware study 35

4.2 Removal process 43

4.3 Summing up 44

5. THE FOURTH MALWARE SAMPLE 45

5. 1 Malware study 45

5.2 Removal process 56

5.3 Summing up 56

6. FINAL THOUGHTS **56**

7. ANNEX **57**

7.1 Sample #1 **57**

7.2 Sample #2 **57**

7.3 Sample #3 **57**

7.4 Sample #4 **57**

1. Introduction

In this report we have tried to convey all the findings we have stumbled upon when analysing the Windows XP virtual machine that was handed to us for inspection. We have chosen 4 malware samples and analysed as thoroughly as we could so that we could come to understand how the operated and ultimately how to remove them. We hope to have been clear and detailed enough.

Please note that we'll reference the first edition of Practical Malware Analysis book by Michael Sikorski and Andrew Honig throughout the document with the [PMAL] tag. When writing it down we'll include the pages we are extracting information from. This has been our only paper reference; the rest have been obtained through the Internet and are linked in the document itself where appropriate.

1.1 Setting up

Before setting off on a quest to find malware lurking around, we need to define the scenario we'll be working on. As stated in the coursework specification, our environment will be a virtual machine running Windows XP.

In order to make it feel like at home we have taken a series of steps so that all the necessary components and tools were available. We'll briefly describe the entire process so that what we include in this report is as reproducible as possible.

Please note that in the ensuing discussion we'll refer to the machine running Windows XP as the guest machine and the local machine as the host machine.

1.2 Importing the VM

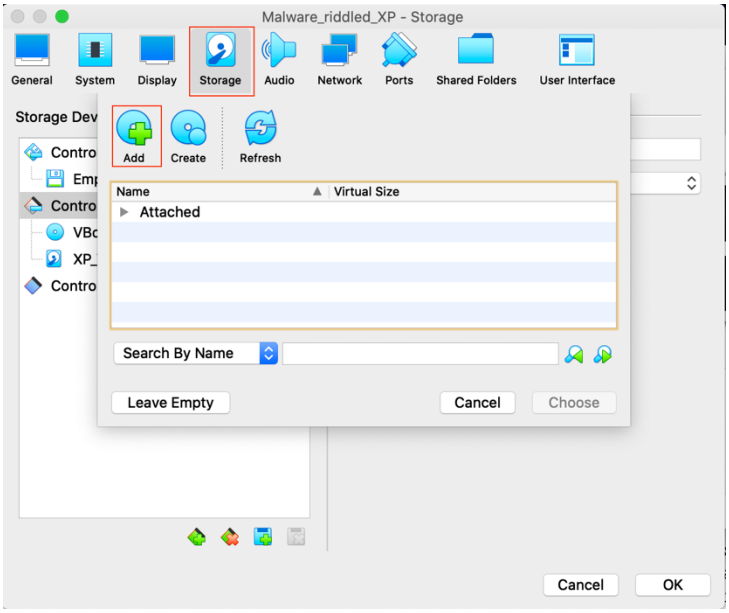
The guest machine we are to work with is tremendously specific and so is very hard to exactly replicate. That's why the way it's been distributed is through a virtual machine image using the **.ova* format. As we are working with Oracle's VirtualBox virtualization software we just need to open it up and import it, just as easy as that!

1.3 Installing the Guest Additions

We shan't forget Windows XP uses a graphical desktop environment. We'll need to make the guest machine's desktop as big as ours so that we can comfortably move around. To this end, we can install the so called *VirtualBox Guest Additions* which just bundle up a set of drivers designed to make using VMs a seamless experience. Installing it is as easy as going into the *Device* menu within the running VM and clicking on *Insert Guest Additions CD Image…* We can then navigate to Windows XP's *My Computer* from the *Start* menu and double click

on the disk unit to bootstrap the installation. After rebooting the VM we should be ready to go.

It's likely that before the guest additions can be inserted into the machine, we need to create an optical disk bay. We can do so from the settings of the VM before booting it up by just adding a new entry to the *IDE Controller*. We are attaching a screenshot showing how to accomplish that.



1.4 Configuring the Network

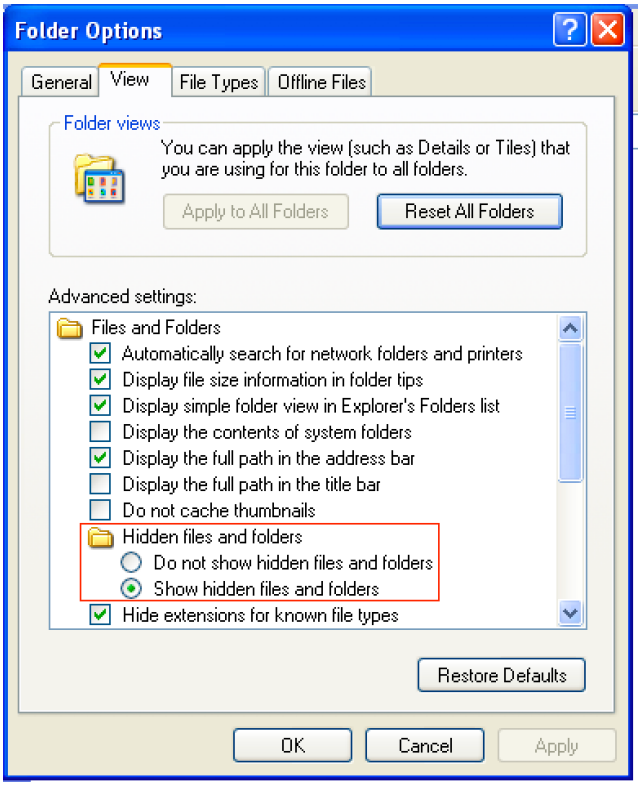
As our intention is to "detonate" malware within the machine we should mind the network configuration so that we don't let any of the samples roam freely on the network. This can be easily accomplished by just clicking on the *Devices > Network > Connect Network Adapter* option within the running VM which will just toggle the NIC (Network Interface Card) as we please. This lets us have an entirely functional network setup that can be enabled at will. As our host machine is a MacBook Pro we'll use a *bridged adapter* configuration which will expose the guest VM as an independent machine to the LAN our host machine is connected to. This implies it'll get its very own IP address within the LAN our host is attached to. We can also point out that the network topology we are logically implementing is a layer 2 bridge connecting the host and guest machines so that both still belong to the same network segment (i.e. LAN). If we were to have a Windows-based host, we would go with a *NAT* configuration option as that's less error prone and given the Windows XP machine is not to behave as a server we won't perceive any functional difference.

Either way, as we intend on leveraging *WireShark's* capabilities for capturing network traffic we need to install the suitable NIC driver. This will also force us to select the correct NIC when configuring the VM, thing we can do under the *Advanced* section within the *Network* section of the configuration. Given the driver we were provided we'll choose the *Intel Pro/1000 MT Desktop (82540EM)* NIC so that everything remains compatible. The driver installer can be found in our course repository as usual. Please refer to the annex on the repository structure. Installation is just a matter of double-clicking the executable and accepting the different steps.

1.5 Showing the hidden files

As more recent Windows releases, we see that Windows XP already made use of the *NTFS* file system for its drives. It, as many others, has support for hidden files. These files are nonetheless not as hidden as we might expect. We can inspect them in a fairly easy way both through the graphical file explorer and from Windows' CLI (*cmd*). In order to show hidden files on the file explorer we just need to go to the *Control Panel* (which we can do from the *Start* menu) and click on the *Folder Options* option. We are attaching a screenshot showing the option we are to tick.

We can also use good ol' *cmd* to navigate the file hierarchy and inspect hidden files. As we know we'll find a gold mine on *C:\WINDOWS\msagent\intl\MS_PMAL_Agent* we might as well show how we can list that hidden directory. We just need to navigate to its parent directory with cd *C:\WINDOWS\msagent\intl* to then run *dir /AH*. As seen in *dir's* help which we can read through with *help dir*, we can control its output through a series of options. Hidden files and folders won't be shown by default, but we can force it to display them with the */AH* which only displays files with the *Hidden* attribute. Then, we can indeed see the *MS_PMAL_Agent* directory if we run *dir /AH* after the previous cd statement.

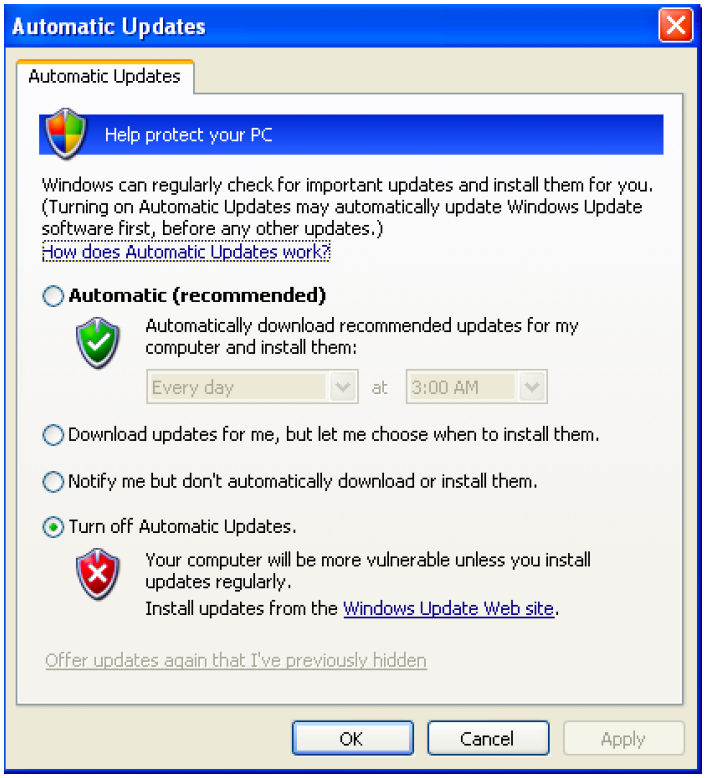


Either way, we need to be aware of the existence of hidden files and take measures so as to have them

present and take them always into account.

1.6 Stopping automatic updates

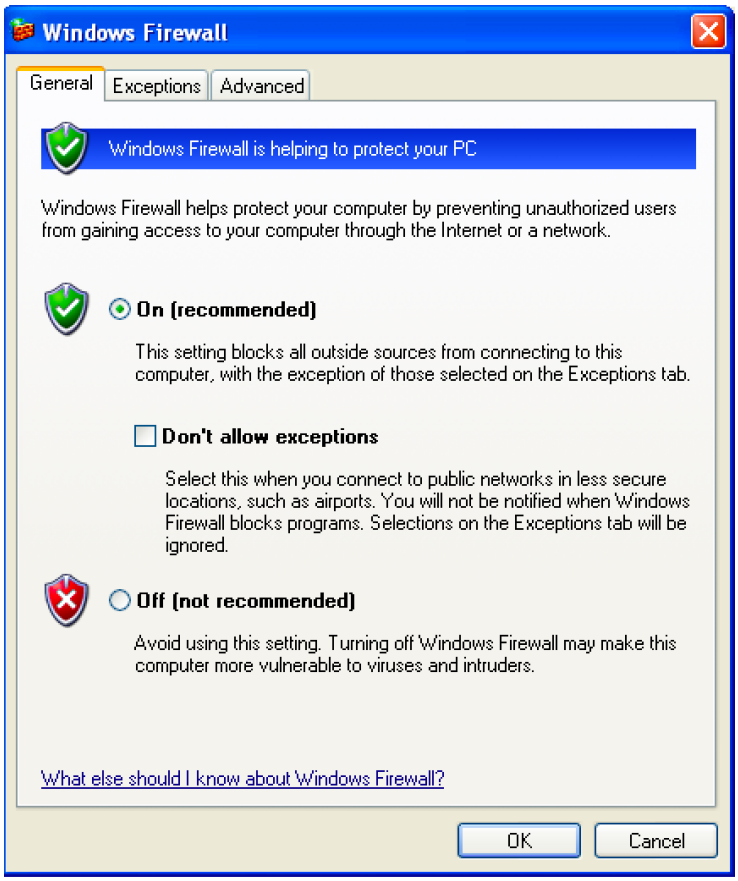
The malware samples we have been provided are guaranteed to behave as expected if and only if we maintain the correct OS version. That's why we need to prevent automatic updates from happening. We can easily do so by going to the *Control Panel* and clicking on *Automatic Updates*. We can them disable them form the next window that'll pop up and that we are including in below.



This will guarantee nothing is meddled with as we carry out our analysis so that we can worry only about crucial and important aspects whilst being sure no "funny" behaviours are taking place behind the scenes.

1.7 Regarding the firewall

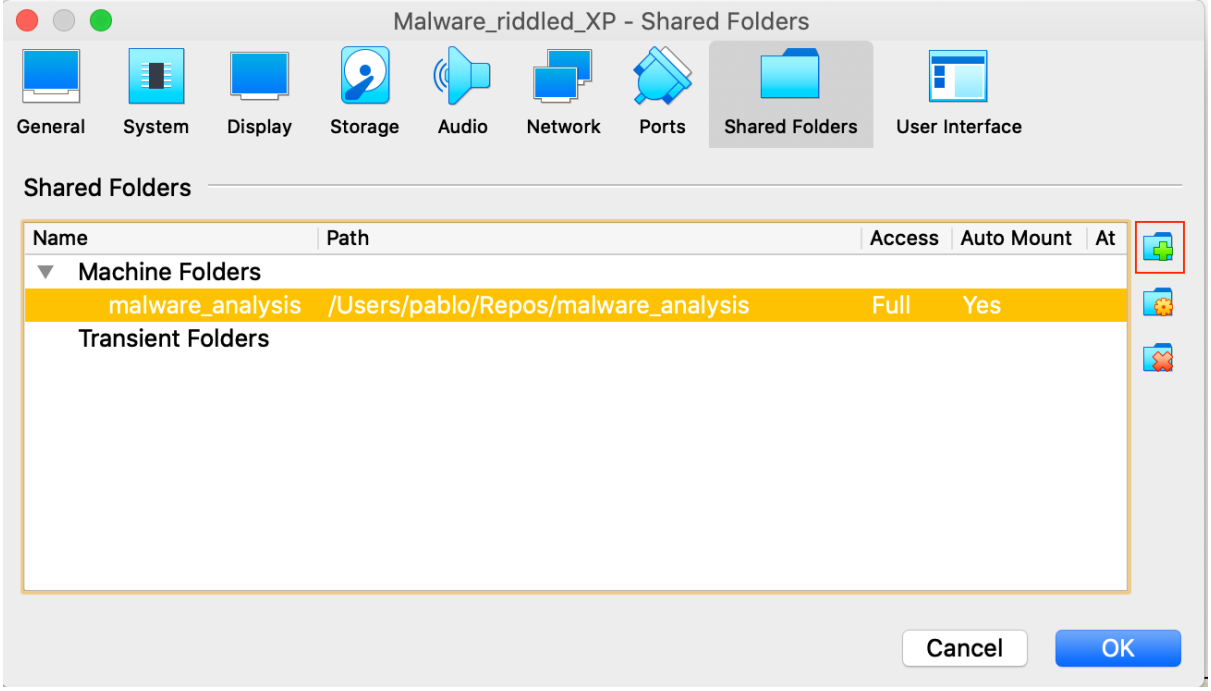
As before proceeding to the analysis we can't be sure of there being any malicious use of the network we decided that shutting the firewall down before having any leads as to what the behaviour of the malware was a little bit of an overkill. We don't depend on the network for using our tools either so there is no immediate need for doing so. We'll keep in mind that it's actually up in case it becomes something to consider at some point. Nonetheless, we can find the configuration under *Control Panel > Windows Firewall*. We are attaching a screenshot showing our current configuration.



Please note that we intend to capture network traffic from within the guest itself so there is no need to have any external machines capturing traffic. Should that need arise we'll shut the firewall down as needed. Again, we don't want to take any actions that might turn up being unnecessary as we strive to carry out an analysis that's as little invasive as we can.

1.8 Communicating with the outside world: shared folders

In order to speed our work process and in an effort to let us work as much as possible from our host machine we have decided to leverage the power of shared folders to share files and folders between the host and guest seamlessly. In order to do so we just need to go to *Devices > Shared Folders > Shared Folder Settings…* on the running VM. We'll then add a new folder by clicking on the marked icon.



We just need to point it to the folder we want to share from our host machine and select the *Auto-mount* and *Make Permanent* options so that it's added to the VM whenever we boot it up. We'll see that the shared folder shows up as a networked drive within *My Computer* in Windows XP and that anything we copy to it on one end will appear on the other and vice versa. This will let us move files from one machine to the other in no time thus speeding up the analysis process by a great amount.

1.9 Getting the tools

A malware analyst is only as good as his/her tools so we need to get our hands on a few handy programs that will aide us in our analysis. We mustn't fall into the trap of thinking that he more tools we employ the better and more thorough our analysis will be. Whilst it is indeed true that different tools may provide different insight, we should always trust our own judgement as the tools are submitted to us and not the opposite. Thus, once we have concluding evidence on a piece of malware, we won't continue running it through several tools for the sake of doing it if we believe this is not necessary. That'll let us employ less tools in a more efficient way so that the overall outcome of our work is as solid as we want it to be. We are including a complete relation of them as well as a short description so that the reader can have an idea of what we are looking for when we use each of them. Note the ones marked with * were installed by us and their setups can be found within the aforementioned repository. The rest were already present in the imported VM.

- **Wireshark:** This program is a network traffic analyser and capturer. It'll capture all the traffic passing through our NIC (Network Interface Card) and then apply a series of filters on them to ease the classification of the flows they belong to. In other words, it'll capture link layer frames and then display the information in a graphical way which shows the protocol it belongs to in each and every network layer of the Internet Protocol Stack (Link/Net/Transport/Application layers) as well as other useful information such as port numbers, IP addresses, application data… We have employed it when analysing the first piece of malware due to the fact that we had leads indicating the use of the network. We could thanks to it discover that the malware under study was effectively trying to access a remote site through the HTTP protocol. In order to work faster we have captured the traffic on the Windows XP machine and then passed it to the host machine to inspect it with wireshark on a faster setup. As there was no need for a live analysis, we preferred this way of working. This tool is used for **dynamic analysis.**
- **Cutter:** This tool is just a graphical frontend to *radare2*, a reverse-engineering framework that's very appreciated by the community. We have used it to disassemble suspicious executable files as well as DLLs. We have leveraged the capability it has of renaming functions and parameters to make the code easier to handle as well as the decompiler functionality that represents the disassembled code resembling a HHL (High Level Language). In order to facilitate working with cutter we have passed the samples to the host machine and we ran it from it. This tool is used for **static analysis.**
- **Process Explorer:** This program runs under the VM and shows information regarding the current running processes as well as the DLLs they include. Thanks to it we were able to spot the suspicious *msgina32.dll* executable in the second malware sample for instance. This tool is used for **dynamic analysis.**
- **Process Monitor:** This program also runs under the VM and lets us inspect the program while it's running. We can see the files it opens; the library calls it makes among other aspects which let us have a deeper understanding of the process operation. We need to remember to apply filters to look only for the program we are interested in as the output becomes overwhelming pretty fast. This program is ran form the VM. This tool is used for **dynamic analysis.**
- **PEview:** This tool lets us inspect PE formatted executables (*.exe) and DLLs (*.dll) so that we can see the sections making them up and spot suspicious file

contents. We have used it extensively. This program is ran from the VM. This tool is used for **static analysis**.

- **PEiD:** This tool detects whether a possible malware sample is packed or not; packed meaning it's compressed so as to make our analysis deliberately more difficult. If a sample were to be packed, we would know that static techniques are bound to fail. This program is ran from the VM. This tool is used for **static analysis**.
- **Strings:** This program reads the contents of whatever file we feed it and outputs printable strings. We have ran it on samples we get out of the virtual machine so we have been using it on the host. Its information let' us formulate an initial strategy for tackling samples in a simple way. This tool is for **static analysis**.
- **Resource Hacker:** This tool opens executable files such as *.exe formatted ones and lets us extract contents from it as well as read them. We used it to extract a DLL from an executable we knew was moving it to another location. This tool is run form the VM and is intended for **static analysis**.
- **Dependency Walker:** This program runs from the VM and it lets us inspect the library functions imported and/or exported by different executables and DLLs. It'll aide in determining whether an executable is packed or not. It runs under the VM and is intended for **static analysis**.

We can then see we haven't used that many tools. We have nonetheless employed them in such a way that we extracted all the advantages they could provide.

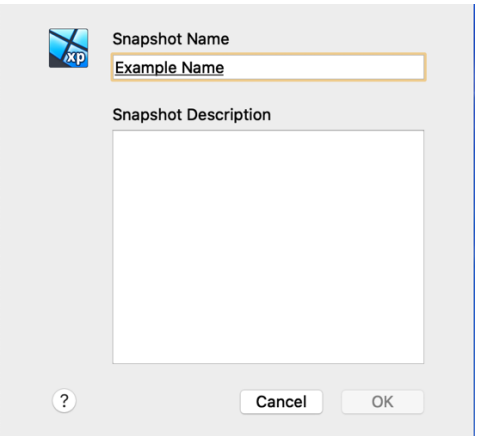
As seen in the tool descriptions we have also been using our host actively to run some of the analysis tools. This has been mainly done because it runs way faster than the VM and we have access to the latest versions of the tools which many times provide additional functionality. Nonetheless, the dynamic traffic capture has been done within the VM and the samples have been exported for an offline analysis where appropriate. We have not used a second VM to aide us in our analysis.

1.10 Better safe than sorry

Playing around with malware can be risky. We shouldn't forget that these pieces of code are designed to, in the least, access sensitive and in the worst to wreck the machine they run on. This implies we might get to a point our VM is "broken" beyond repair so we had better get a *snapshot* of the machine before proceeding so that we have a correctly configured fallback point that we can always fall back to. We can easily get this snapshot by hitting *Cmd + T* on macOS and naming it to something suitable such as *"Backup State"*. We can

also opt to add a description to it, but we believe it's not necessary given the title.

Once saved we can always start the machine from the saved state from the VirtualBox manager as seen in the following picture.



1.11 Final notes

In order to run the VM we just need to launch it from within *VirtualBox's* manager. The user-password pair is *Administrator-AVictim.*

With that we are ready to get our hands dirty. We'll include the analysed malware samples as we found them so that the chronology becomes another valuable asset showing how the analysis progressed with time.

2. The first malware sample

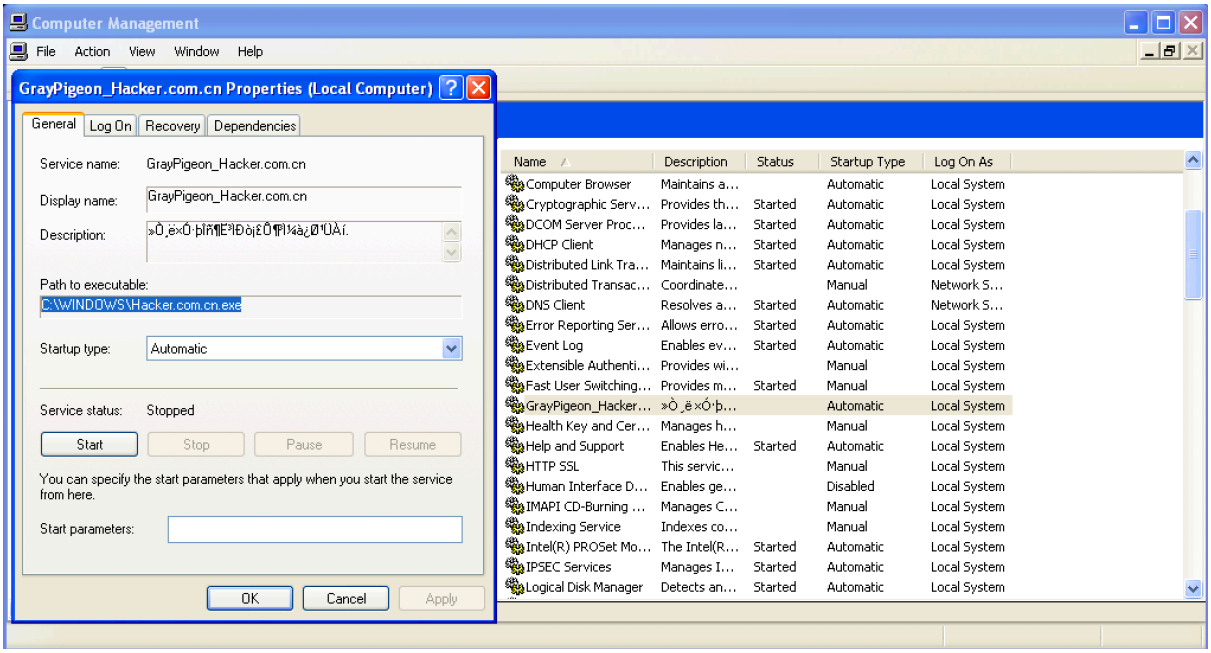
2.1 Malware study

Malware will want to run undetected as long as it possibly can. This should make us consider the different ways programs can be started to try and see whether they are being weaponized to launch malicious software.

Note the following is based on our lecture's contents.

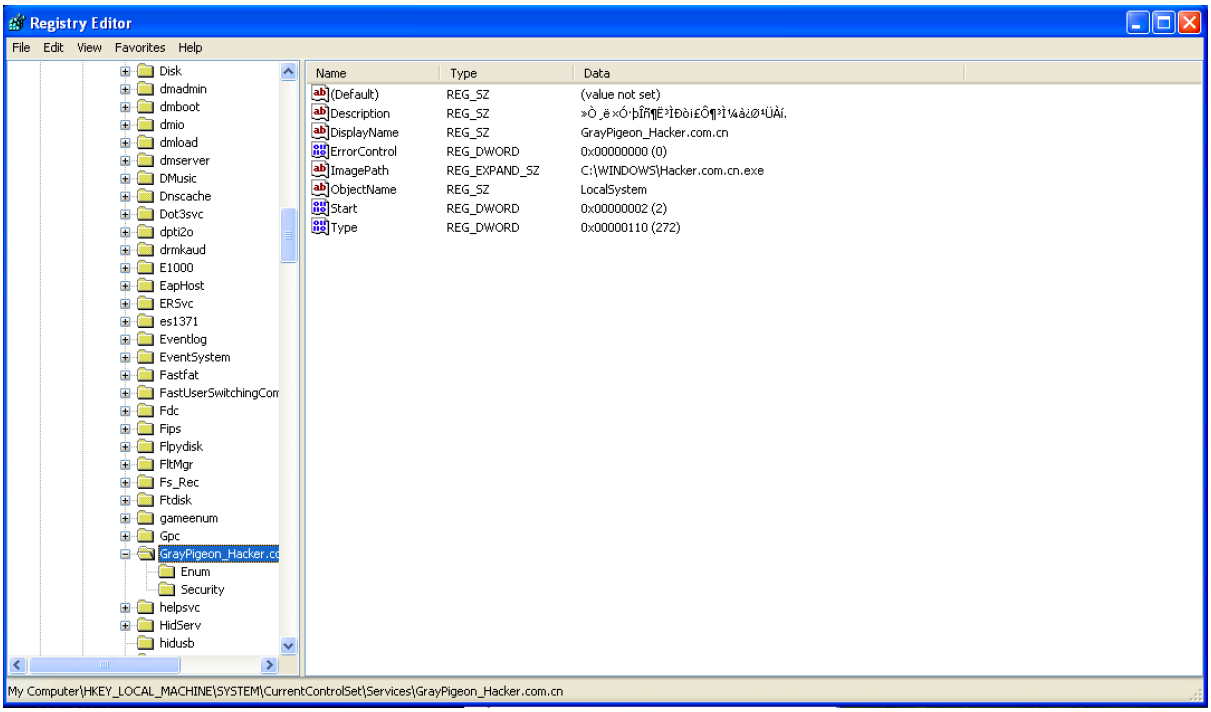
One such example are services. As seen on page 152 of [PMAL] these services are not run as standalone process. Instead, the *Windows Service Manager* will schedule them for execution according to their configuration. The tasks run in this way will remain in the background in the same way *daemons* do. The term *daemon* refers to programs that are not designed for the user to interact with. They carry out a task such as providing an HTTP server accepting external connections and their behaviour can be configured through text-based configuration files in most cases but not by direct interaction.

In order to inspect the existing services on a machine we can just *Right-click* on *My Computer* and select the *Manage* option. We can then navigate through *Services and Applications > Services* to find a list of available services. If we look at the names and descriptions we'll soon stumble upon the *GrayPigeon_Hacker.com.cn* service with description ª"∏Î◊"Σ¸ŒÒ∂ À≥Ã–Ú°£'∂≥Ã°‡øÿπ‹¿Ì.



It certainly doesn't look like a legitimate entry so we can click on it to gather more information about it. In doing so we'll see that it is launching the *C:\WINDOWS\Hacker.com.cn.exe* file when it's invoked so a sensible next step would be to analyze said file.

We'll also try to gather some more information about the service itself to see whether it's being started automatically upon boot, what type of service it is… This type of information can be found on the *Windows Registry* which we can access by going to *Start > Run* and typing *regedit*. We then have to navigate to *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services* and look for our service. Please note we'll contract *HKEY_LOCAL_MACHINE* to *HKLM* from now on. Once we get to *HKLM\SYSTEM\CurrentControlSet\Services\GrayPigeon_Hacker.com.cn* we'll see the different entries configuring this service.



They are quite unreadable, so we believe it's useful to discuss the *sc* command. As seen in its help (which we can access with *sc /?*) it's a program that communicates with the *NT Service Controller and Services* so that it can get information about services. We'll be looking into the service's configuration, so we'll want to be using the *qc* option. Then, by running *sc qc "GrayPigeon_Hacker.com.cn"*

we'll get the same information as before but in a more comfortable format.

```
C:\Documents and Settings\Administrator>sc qc "GrayPigeon_Hacker.com.cn"
[SC] GetServiceConfig SUCCESS

SERVICE_NAME: GrayPigeon_Hacker.com.cn
        TYPE               : 110  WIN32_OWN_PROCESS (interactive)
        START_TYPE         : 2   AUTO_START
        ERROR_CONTROL      : 0   IGNORE
        BINARY_PATH_NAME   : C:\WINDOWS\Hacker.com.cn.exe
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : GrayPigeon_Hacker.com.cn
        DEPENDENCIES       :
        SERVICE_START_NAME : LocalSystem
```

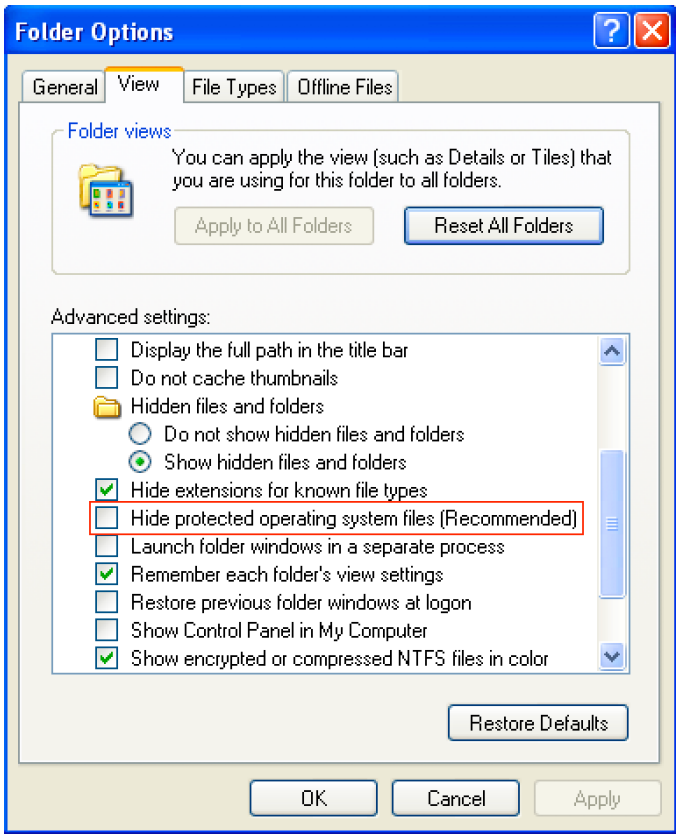
We see how the service is set to be run automatically at boot (its *START_TYPE* is *AUTO_START*. *See this link for more information*) and it's of the *WIN32_OWN_PROCESS* type which means it's code is an *EXE* file as stated in page 153 of [PMAL], something we already knew before!

Either way, we have found that **the malware itself is contained in the aforementioned *EXE* file on *C:\WINDOWS\Hacker.com.cn.exe***. We'll try to copy it to our shared folder so that we can analyze it on our host machine which is way faster than the VM so that scrolling and combing the file is not that cumbersome.

Even though one might think copying the file would have been easy we had to fiddle with the settings a bit. It turns out the *Hacker.com.cn.exe* was marked as a *System File* which made it dodge our current folder options. If we navigate to *C:\WINDOWS* through a command line, we can indeed run *dir /AH* once more and we'll eventually see the *Hacker.com.cn.exe* appear as it should. If we run *attrib* on it to see its attributes, we'll see the following output:

```
C:\WINDOWS>attrib Hacker.com.cn.exe
  SHR      C:\WINDOWS\Hacker.com.cn.exe
```

The *SHR* output indicates it's a *System, Hidden and Read-only* file. This can be seen on attrib's help with *help* attrib. If we visit the folder view options once more, we'll have to uncheck the option hiding system files as seen in the following image.



In doing so the file will show up in the graphical explorer as well. We just want to point out that we could have copied the file to our shared folder with the *copy* command without a problem and we could have skipped tweaking the folder settings again. This shows the power of text-based interfaces.

If we try to copy the file, we'll stumble upon the following error:

Nonetheless, the service that launched the file is already stopped, that is, it's already stopped as we could see in some of the screenshots above. Then we'll just disable it so that it doesn't start at boot and try to copy the file once it's not run in the first place. Once we did that the file copied without a problem. Note that if we are to use the *copy* command to copy the file over, we need to run *attrib -s -h*

C:\WINDOWS\Hacker.com.cn.exe beforehand so that it stops being a system and hidden file. We can then run *copy C:\WINDOWS\Hacker.com.cn.exe Z:\MW_sample_1* to create the *MW_sample_1* in our shared folder.

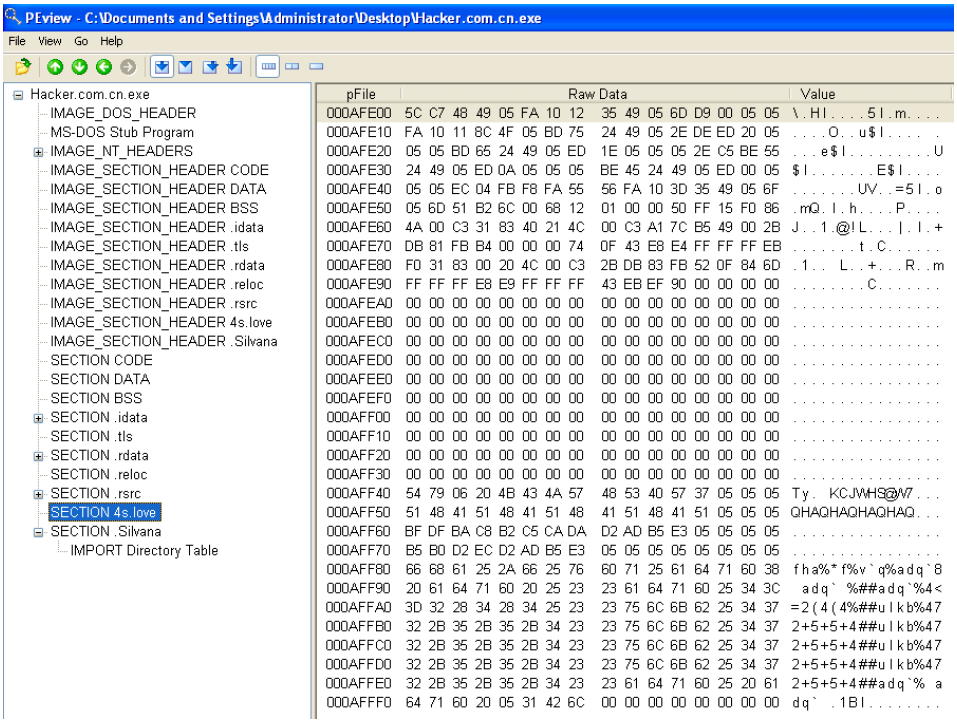
Now we can run the *file* command on the sample from our *macOS* machine to find the following output:

```
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples$ file Hacker.com.cn.exe
Hacker.com.cn.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

Taking this information into account we can be sure the file is indeed a **32-bit Portable Executable file**. Now, this *PE* format is the

As seen in page 14 of [PMAL] this file format is the one employed in Windows executables, *DLLs* (Dynamic Link Libraries) and object code (the one we get when running the compiler on source code before passing it through the linker). It basically tells the Windows program loader all the information it needs for loading and running the program and it also contains the executable compiled program code itself. Information included in this header are the needed library functions, and space/memory requirements among others. As this information is provided in a known format, we can easily spot this type of file.

We can now try to inspect the file using *PEviewer* so that we can clearly inspect the different sections composing the file.

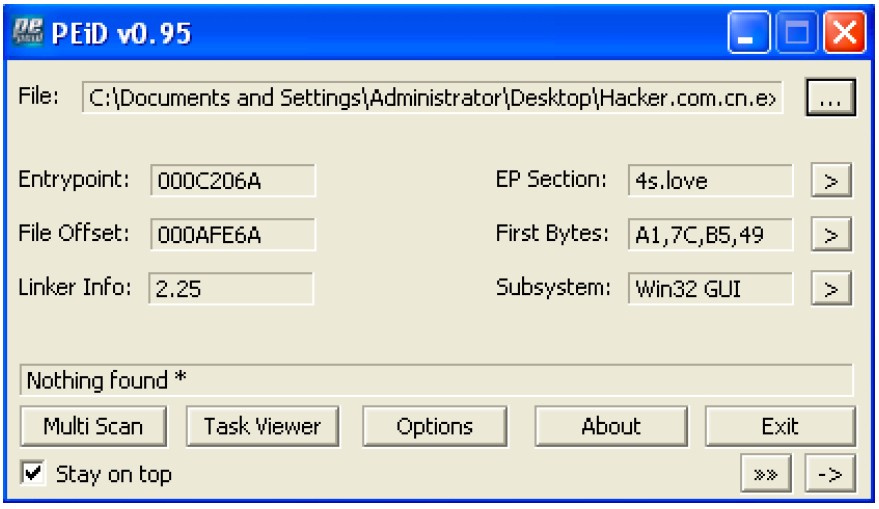


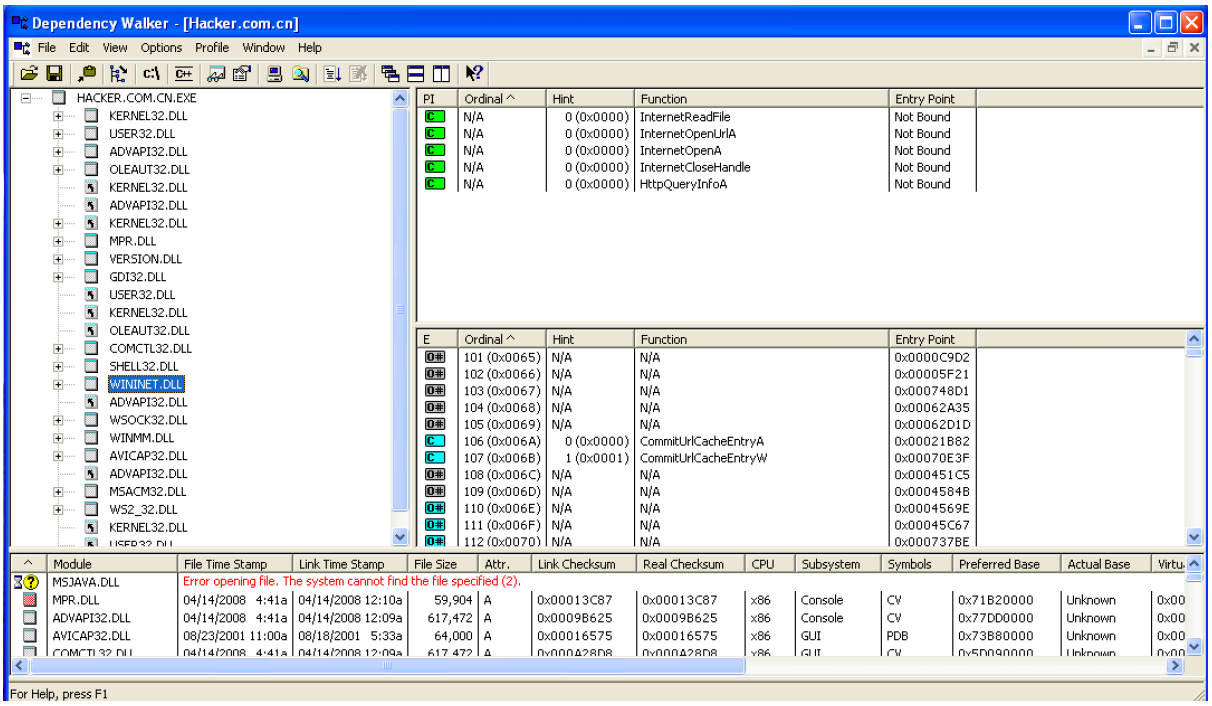
All the sections present in the analysis seem to be normal except *4s.love* and *Silvana*. If we look at the contents shown in the above image, we'll see a bunch of text on the HEX view so we can expect some information to be there. We cannot know much more at the moment, but these 2 sections are worth to keep an eye on for later.

We can continue our static analysis (note we haven't analyzed the running file yet) by looking into whether the file itself is packed. As stated in page 13 of [PMAL] we see that packed programs "pack" or compress the "real" or main program so that it cannot be statically analyzed. Then, an unwrapper is added on top of it so that it can be unpacked at run time and become the program it was supposed to be in memory. In other words, upon execution the unwrapper will kick in, decompress the packed code and run it as the OS itself would have done if it hadn't been packed.

The above implies we can only really statically analyze the unpacker which isn't of much use to us. This approach severely limits the information we can gather through a static analysis and almost always obliges us to perform a dynamic analysis of some sort.

Detecting this type of program can be done through specialized programs such as *PEiD* or by inspecting the imported functions, which can be easily done with programs such as *Dependency Walker*. A very large red flag signaling packed programs is the brevity of the imported function list. If it's very concise we are almost certainly dealing with a packed sample as stated in page 21 of [PMAL]. After running the aforementioned tools on our sample we can conclude it's **not packed** as it has a great deal of imported functions and it's not deemed as packed by *PEiD* itself. We could run it through more "pack detectors" but as we are already aware of 2 factors telling us it's **not** packed we won't go down that rabbit hole: we need to be efficient when analyzing samples and we can't devote time to hypothesis not backed by any evidence. We are attaching the output of both programs below.





We should end by pointing out that the approach to take for analyzing these programs, should we encounter them, is to dynamically analyze them as the program's memory image will be the same no matter if it's packed or not. At the end of the day, the malware sample needs to execute as a normal program so by inspecting its memory usage and attaching debuggers to it we can get a clearer idea so as to its purpose. This in fact constitutes another indicator of a program being packed. We will see that comparing the sizes of the executable and the memory image (program in execution) are uneven with the image being larger. We mustn't forget the executable code was packed!

If we inspect the sample's dependencies again as seen in the above screenshot, we'll soon discover that among the imports we find:

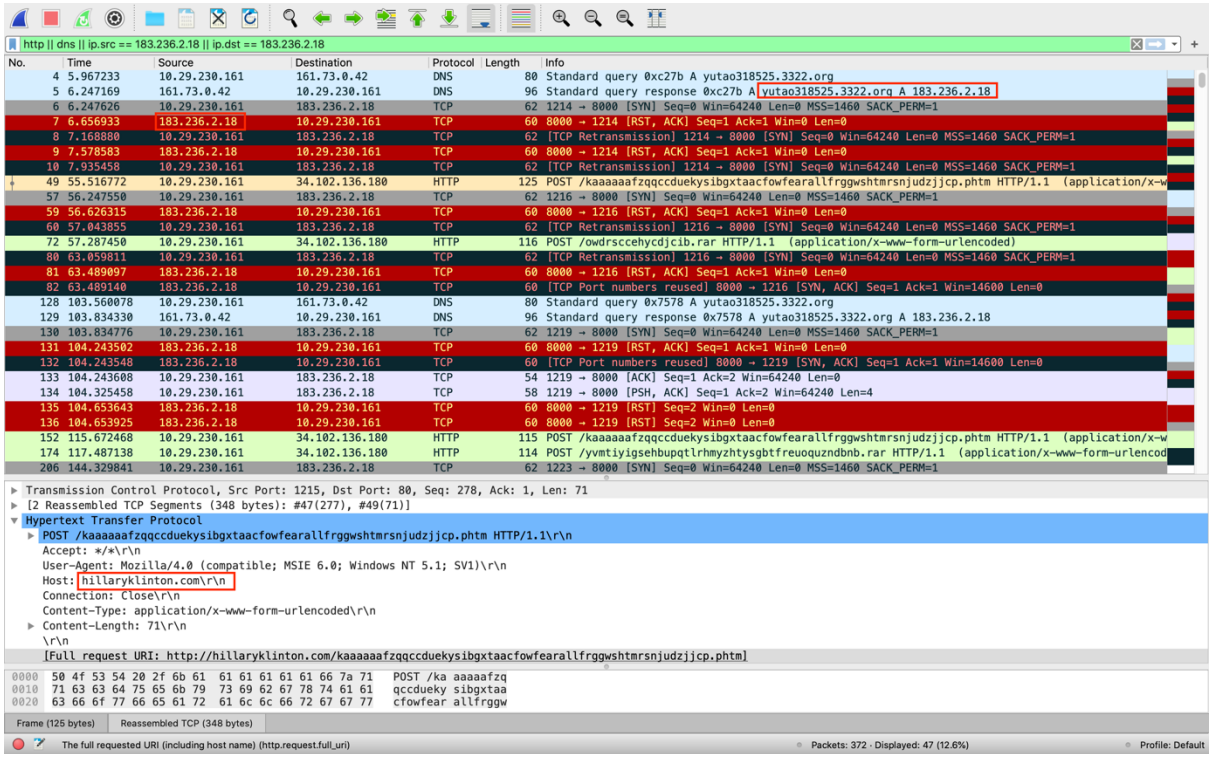
- ***Wininet.dll:*** This DLL contains high-level network functionality as it implements application layer protocols such as FTP, HTTP or DNS among others. We can expect our program to make use of the network to some extent. This implies we'll analyze the traffic it produces through *wireshark*.
- ***Advapi32.dll*:** This DLL let's the process access core Windows functionality such as the registry. As we found this piece of malware being launched from a service, we can expect the use of this library for providing that means of persistence.
- ***Kernel32.dll*:** This DLL is very commonly imported by process. We'll see nonetheless that it might be being

used to inject code into another process by means of the *WriteProcessMemory* function as the *IEXPLORER.EXE* string is present in the program itself. We'll discuss this later.

- ***User32*.dll & Gdi32.dll:** Having these 2 DLLs present indicates this program makes use of some sort of GUI (Graphical User Interface). It can exist even if the current user can't see it, we shouldn't forget Windows is a multi-user system!

Taking this functionality into account we need to try and analyze the sample in a dynamic way to try and capture some network traffic hoping to find new evidence such as the URLs it's trying to connect to for instance. Note the above information belongs to Table 1-1 from page 17 on [PMAL].

Analyzing the network traffic whilst the malware is running, we can see something like the following:



Seeing the different DNS queries as well as the HTTP traffic we find the malware is talking to domains *yutao318525.3322.org* and *hillaryklinton.com*. This implies the malware is either receiving instructions or uploading data to these sites which makes us consider a model where the malware is a client and the server is a C&C center at one of these IP addresses (183.236.2.18 & 34.102.136.180 respectively). Note the communication with *yutao318525.3322.org* is accepted on the server's port 8000 and the one with *hillaryclinton.com* is carried out on port 80, the standard for HTTP.

We should also note that the HTTP method we are using is POST, that is, we are uploading something to the *hillaryklinton.com* site.

Now that we are concerned with a dynamic analysis, we should mention that this process doesn't show up on the *ProcessExplorer* program. If you recall our discussion about the imported libraries you'll see how the *kernel32.dll* provides access to the *WriteProcessMemory* function. This can be used to inject code directly (page 257 on [PMAL]) into another process so that it runs the malicious code without the malware itself being a process. That way, the malware covertly executes on behalf of another task. This is exactly what's happening here. The sample under study is injecting malicious code onto the *IEXPLORER.EXE* process as we can deduce by inspecting the *strings* section on the *cutter* disassembler when analyzing the program:

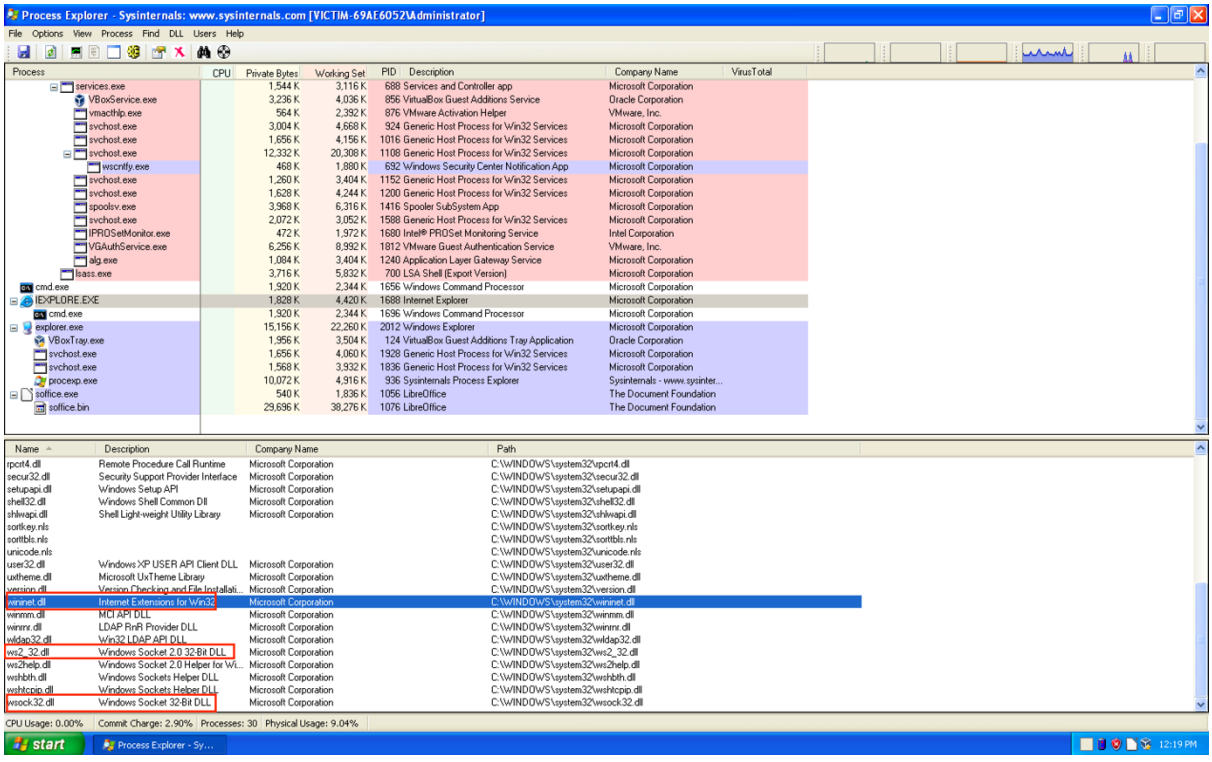
We effectively find the *IEXPLORER.EXE* string on it.

Having taken a look at the *strings* of the sample motivates a new question: is the malware obfuscated? In order to answer we must first define what the term stands for.
In an effort to hamper the malware analyst's efforts a malware author can bloat the code with unnecessary functions, twisted logic, arbitrary jumps and the like so that our task becomes harder and we feel "lost" within the disassembled code. This also applies to the constant strings that are common to every program. Instead of including them in plain ASCII they can be encoded by means of schemes such as *base64* or any other the malware writer devises. This will make our life more difficult and we might even find ourselves forced to debug the program and inspect memory and register contents at runtime as these strings will be eventually decoded at some point so that can be used normally. All in all, the malware programmer is trying to hide how the malware operates through obfuscation by throwing us off the correct track. We mustn't be fooled by this technique and always be mindful of what code we analyze so as to know what is going on and how to effectively counteract it. This discussion is based on page 13 of [PMAL].

Now, this particular sample sits somewhere in the middle of the definition. It's true that some strings are readable like *IEXPLORER.EXE* but it's also true that the URLs we know the program is connecting to **cannot** be found within its strings. We also find a bunch of rubbish (unreadable strings) in this section as well so our answer has to be that this program is

obfuscated or hidden to a certain degree, that is, some parts are readable whilst others aren't. We are attaching a screenshot showing some unreadable strings to complement the previous one.

As we discussed before, this piece of malware is indeed leveraging network capabilities. We can only infer that from a static analysis through the import of *wininet.dll* but once we perform a dynamic analysis with wireshark we find that connections are being made to sites that, unless being made by the malware itself, cannot be explained. What's more we are supposing the sample is injecting its code to *IEXPLORER.EXE* and this process also has access to networking DLLs such as *wininet.dll* as we can confirm with *ProcessExplorer*.



We see that some of the networking DLLs are attached in the image above.

We would also like to delve a little bit deeper into the persistence mechanism used by the malware. As we saw before it's starting itself through a service. Then, as seen in page

140 of [PMAL] this can be achieved by writing entries to the *run* subkey in the registry. We mustn't forget the malware imported *advapi32.dll* which lets it modify the registry. What's more, if we look for the word *run* within the sample's strings, we'll see the following:

Knowing the malware is using this mechanism for persistence we can then take actions to counteract it when deleting the malware.

2.2 Removal process

As we did when we wanted to copy the original executable to another place for analysis, we can just disable a service so that the malware itself doesn't start at boot. We then only have to delete the original executable to get rid of the entire thing. Nonetheless, we prefer to delete the service rather than just disabling it so we can just run *sc*, the same tool we used to query the service's information, to delete it.

We can do so by just hitting *Run > cmd* and executing the following:

```
C:\Documents and Settings\Administrator>sc delete "GrayPigeon_Hacker.com.cn"
[SC] DeleteService SUCCESS
```

We can just then run the following to remove the executable and finish the sample off. Note that now that we know the malware is injecting code into *IEXPLORER.EXE* we know what was causing the issue that wouldn't let us copy the file at the beginning of the analysis. We can just kill the Internet Explorer process from *ProcessExplorer* or the Task Manager and delete the function without a problem. The fact the killing the browser lets us delete the file is a clear indicator that the malicious code is being injected into the otherwise legitimate *IEXPLORER.EXE.*

Either way, after killing the IEXPLORER.EXE process we can just run the following to finish the removal, and therefor analysis, of this first malware sample. Please note we need to mark the file as not *hidden, read-only* or belonging to the *system* before deletion, which we do with the *attrib* command.

```
C:\WINDOWS>attrib -s -h -r Hacker.com.cn.exe

C:\WINDOWS>del Hacker.com.cn.exe
```

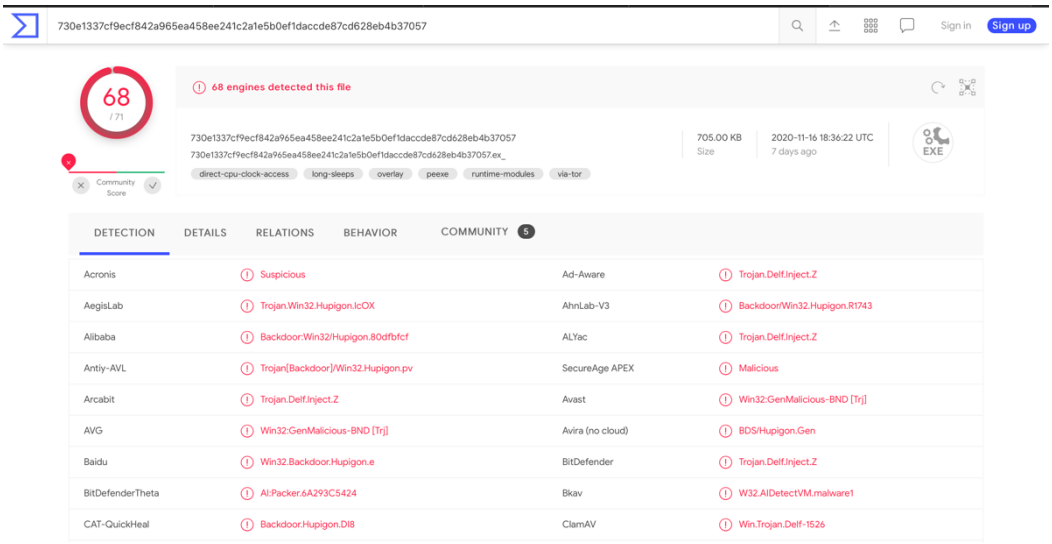
We should also point out that the IEXPLORER.exe process doesn't automatically start at boot once the malware is removed. This implies this piece of malware is also starting it upon execution so that it can inject itself into it.

2.3 Summing up

After all the above we can then summarize the samples behavior as follows since it's executed:

1. The malicious executable is moved to *C:\WINDOWS\Hacker.com.cn.exe* and it's marked as hidden and a system file so as to dodge detection.
2. It sets up a service that runs automatically on boot so that it's persistent.
3. When running it injects malicious code into the *IEXPLORER.EXE* process which will serve as the host.
4. Then it tries to make POST requests to *hillaryklinton.com* and it also establishes a cover channel over a TCP connection with *yutao318525.3322.org* as detected by our *WireShark* analysis.

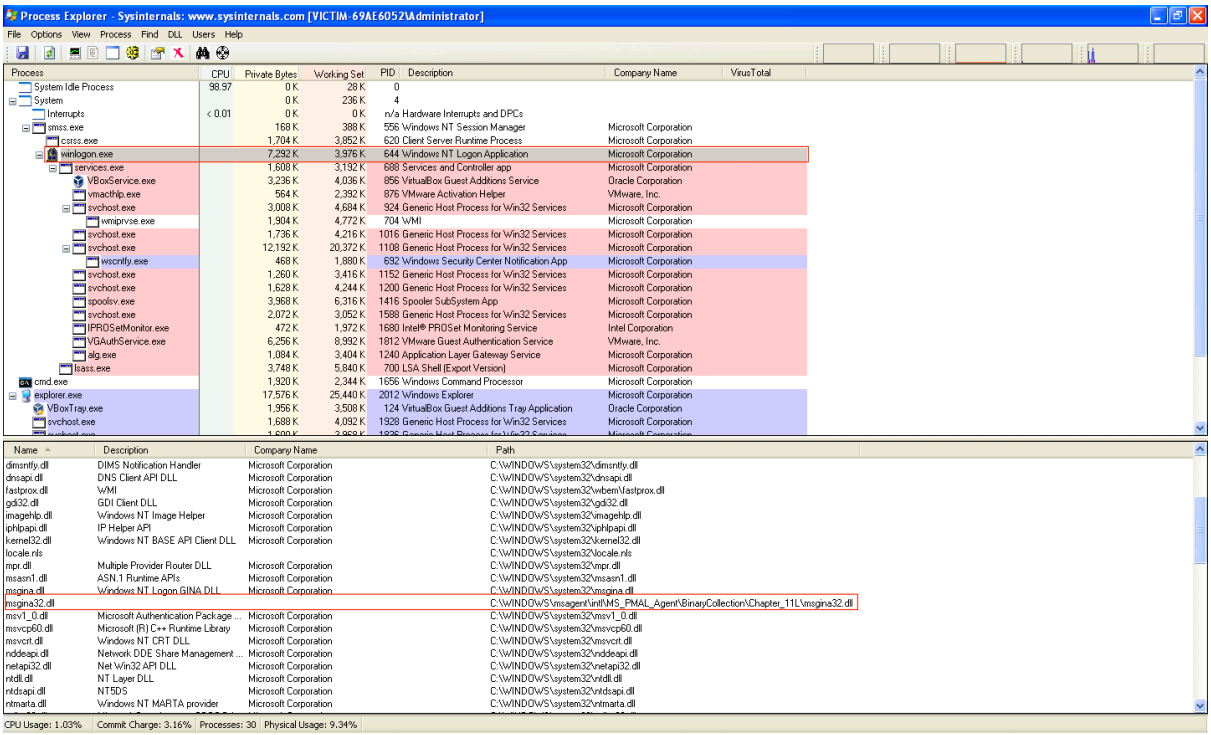
We might be leaving something behind but as the disassembled code was not easy to analyze we cannot state any more than this with confidence. It's true that some DLLs might be getting replaced and that we are missing out on some other functionality, but this is the overall working of the sample. Given the program injection it carries out we can classify this malware as a trojan. If we run it through *VirusTotal* we'll see it's detected as such by 68 antivirus:



3. The second malware sample

3.1 Malware study

Now we'll take a look at the different processes displayed in ProcessExplorer to look for anything unusual that could point us into the right direction to find another piece of malware. When inspecting the DLLs associated with all the processes, we'll see that one of winlogon.exe's DLLs does not live under C:\WINDOWS\system32 or C:\WINDOWS\WinSxS: C:\WINDOWS\msagent\intl\MS_PMAL_Agent\BinaryCollection\Chapter_11L\msgina32.dll.



This DLL doesn't look like something that's legit. We'll just go to the *Chapter_11L* folder and try to see what its contents can tell us.

As we could expect, all the contents under *MS_PMAL_Agent* are hidden. As we had already configured our system so as to show these files, we could nonetheless traverse the directories without any major problems.

Upon reaching the folder we find two executable files: *Lab11-01.exe* and *Lab11-03.exe*. After getting both of them out of the VM for an easier handling, we'll perform a basic *strings* on them to see if there is anything of use.

Given the DLL that brought us to this folder is *msgina32.dll* we'll filter the output with *grep* to look for it. We can

easily do so with a pipe redirection (|) to chain both commands. With that in mind we can see the following:

```
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/GINA_interceptor$ strings Lab11-01.exe | grep msgina32.dll
msgina32.dll
\msgina32.dll
```

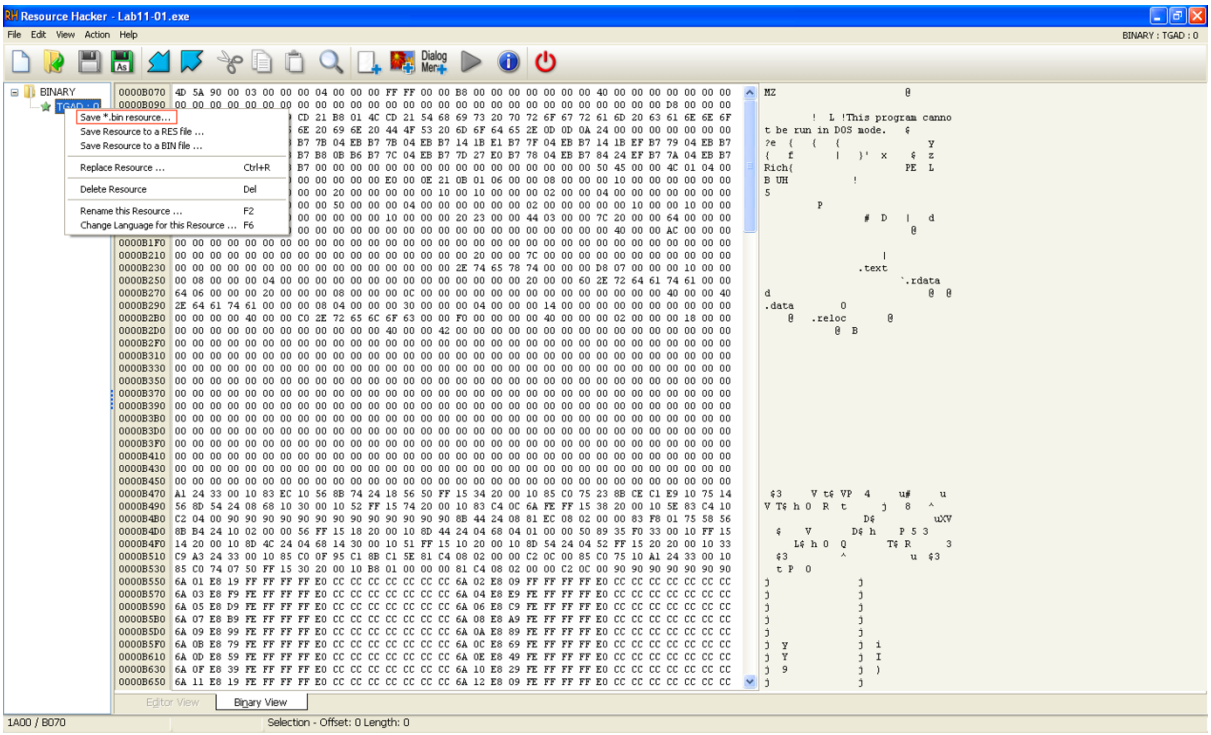
As the *msgina32.dll* itself is contained in the *Lab11-01.exe* executable we can be pretty safe that that's the one we are to delve deeper into to find more useful information.

Taking into account page 235 on [PMAL] we begin to see a lot of elements fall into place. We found this page thanks to the book's index and we were utterly surprised when we discovered that *msgina.dll* is the name of a legit DLL loaded by the *winlogon* process. This set us on the track of a GINA hijacker. As explained in the book, this type of malware leverages the *Graphical Identification and Authentication* system built into Windows XP to recover user credentials. What we'll see is that the original *msgina.dll* is replaced with a modified version that will in most cases just forward *winlogon's* calls to the real functions also exports (*winlogon* needs to be able to call them). In other words, the malicious DLL positions itself in between *winlogon* and *msgnina.dll* so that its operation is mostly transparent. The catch is that it'll run some malicious code before calling the real *msgina.dll* code in some cases. As the GINA system manages user authentication this DLL will have access to the user credentials at some point. Don't forget they're a parameter that's to be passed to the real *msgina.dll* and, as all the information from *winlogon* to *msgina.dll* flows through *msgina32.dll* it will at some point know the user's login credentials. What it'll then do is log them out to a file that we can later retrieve. Note that this file tries to conceal itself by having a name resembling that of a driver (*.sys) so that we don't expect it to contain stolen credentials in the first place. This strategy is just exploiting a system Microfost put in place to provide support third party GINA modules!

If the above were to be true we can dig a little deeper to try and find new indicators of this type of malware. As we explained before, the malicious DLL needs to eventually call the real *msgina.dll* functions so it needs to include them. These are all prepended by *Wlx* so we can try to look at the strings output again but filtering for this type of function names. Doing so revealed the following:

```
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/GINA_interceptor$ strings Lab11-01.exe | grep Wlx
WlxActivateUserShell
WlxDisconnectNotify
WlxDisplayLockedNotice
WlxDisplaySASNotice
WlxDisplayStatusMessage
WlxGetConsoleSwitchCredentials
WlxGetStatusMessage
WlxInitialize
WlxIsLockOk
WlxIsLogoffOk
WlxLoggedOnSAS
WlxLoggedOutSAS
WlxLogoff
WlxNegotiate
WlxNetworkProviderLoad
WlxReconnectNotify
WlxRemoveStatusMessage
WlxScreenSaverNotify
WlxShutdown
WlxStartApplication
WlxWkstaLockedSAS
WlxActivateUserShell
WlxDisconnectNotify
WlxDisplayLockedNotice
WlxDisplaySASNotice
WlxDisplayStatusMessage
WlxGetConsoleSwitchCredentials
WlxGetStatusMessage
WlxInitialize
WlxIsLockOk
WlxIsLogoffOk
WlxLoggedOnSAS
WlxLogoff
WlxNegotiate
WlxNetworkProviderLoad
WlxReconnectNotify
WlxRemoveStatusMessage
WlxScreenSaverNotify
WlxShutdown
WlxStartApplication
WlxWkstaLockedSAS
WlxLoggedOutSAS
```

We need not lose sight of the structure of the attack. We expect to find these function calls in *msgina32.dll*, not on the executable itself. The above output then implies the executable must contain the DLL in some sort of way. We'll then try to extract it from the executable itself, something we can easily do thanks to the *ResourceHacker* tool as seen below.



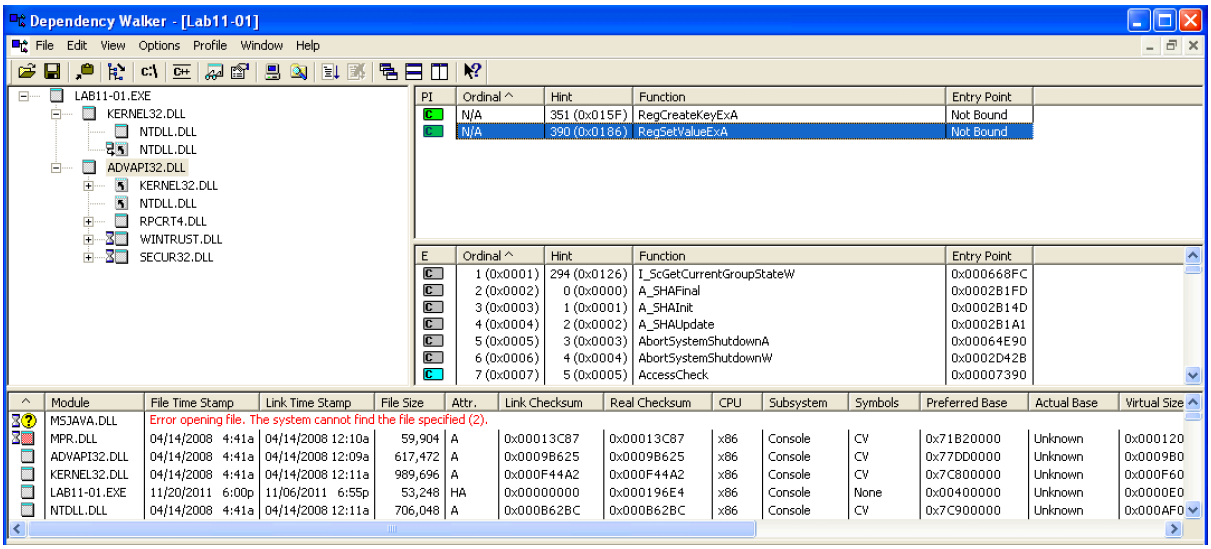
We'll then compare both the extracted file and *msgina32.dll* which is the one that was already loaded by *winlogon*. We can do so thanks to the diff tool and, as expected, the files are identical. Note we saved the resource as *Likely_msgina32.dll*.

```
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/GINA_interceptor$ file msgina32.dll
msgina32.dll: PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/GINA_interceptor$ file Likely_msgina32.dll
Likely_msgina32.dll: PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/GINA_interceptor$ diff -s msgina32.dll Likely_msgina32.dll
Files msgina32.dll and Likely_msgina32.dll are identical
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/GINA_interceptor$
```

As we can be pretty confident the executable is just dropping the DLL to the same folder it lives in we can postpone the dynamic analysis for now. We know what to look for in the disassembly: a call for writing the payload the executable carries and another for altering the registry value configuring the DLL to be loaded by *winlogon*. As shown in page 235 of [PMAL] we expect the key to be changed to be *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL*. We can indeed confirm this string is contained in the executable:

```
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/GINA_interceptor$ strings Lab11-01.exe| grep CurrentVersion
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon
```

We'll now look at the disassembly of the program just to be sure our guess is what's actually going on. It's worth noting this program is **not packed** as we'll see by getting access to the code through disassembly. There's no need to run it through packet detectors like *PEiD* or the like. If we look at the dependencies, we'll nonetheless see how it only imports *kernel32.dll* and *advapi32.dll*. The latter will be used to modify the registry whilst the former will allow the "dropping" of the malicious *msgina32.dll* to the file system.



Even though there are not that many entries we indeed see the ones the program will need so in this case not having a lot of imports is not denoting the program is packed, it's just carrying out its purpose with surgical precision.

We are using *cutter* to analyze the code. In doing so we'll begin reading at *entry0*. After getting the command arguments we'll *call fcn. 0x004011d0* which gets us to the following:

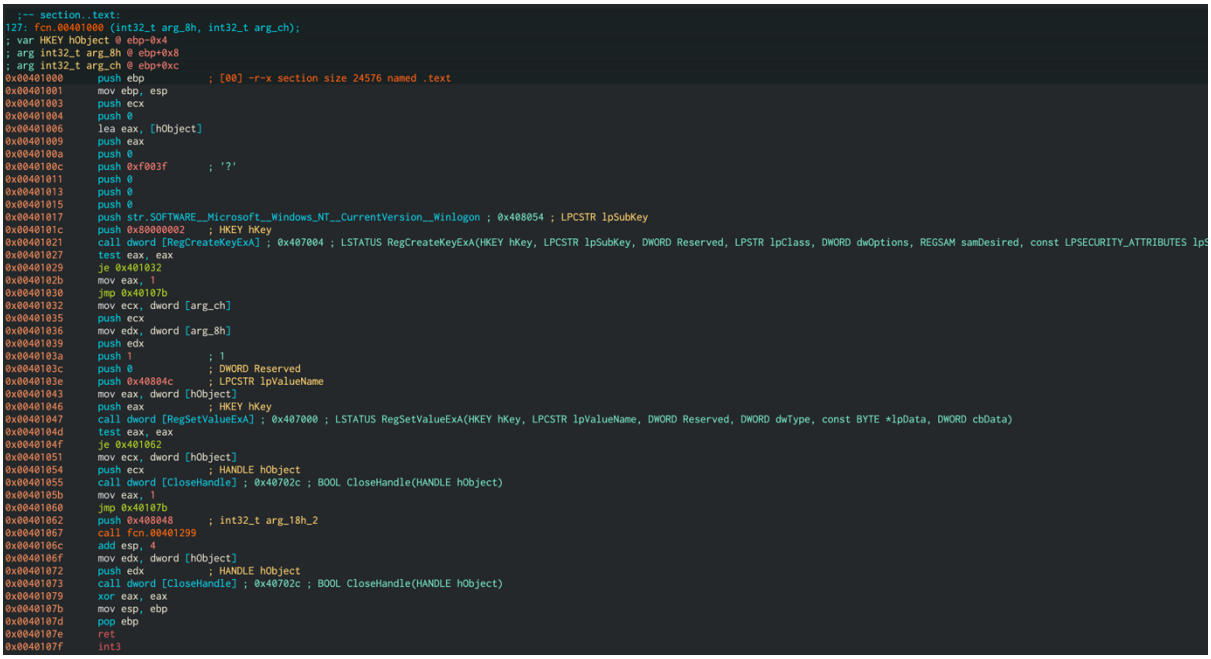
```
201: fcn.004011d0 ();
; var HMODULE var_11ch @ ebp-0x11c
; var int32_t var_118h @ ebp-0x118
; var int32_t var_117h @ ebp-0x117
; var int32_t var_8h @ ebp-0x8
; var int32_t var_4h @ ebp-0x4
0x004011d0      push ebp
0x004011d1      mov ebp, esp
0x004011d3      sub esp, 0x11c
0x004011d9      push ebx
0x004011da      push esi
0x004011db      push edi
0x004011dc      mov dword [var_4h], 0
0x004011e3      push 0              ; LPCSTR lpModuleName
0x004011e5      call dword [GetModuleHandleA] ; 0x407020 ; HMODULE GetModuleHandleA(LPCSTR lpModuleName)
0x004011eb      mov dword [var_11ch], eax
0x004011f1      mov byte [var_118h], 0
0x004011f8      mov ecx, 0x43       ; 'C' ; 67
0x004011fd      xor eax, eax
0x004011ff      lea edi, [var_117h]
0x00401205      rep stosd dword es:[edi], eax
0x00401207      stosb byte es:[edi], al
0x00401208      mov eax, dword [var_11ch]
0x0040120e      push eax            ; HMODULE hModule
0x0040120f      call fcn.00401080
0x00401214      add esp, 4
0x00401217      mov dword [var_4h], eax
0x0040121a      push 0x10e          ; 270
0x0040121f      lea ecx, [var_118h]
0x00401225      push ecx
0x00401226      push 0              ; HMODULE hModule
0x00401228      call dword [GetModuleFileNameA] ; 0x40701c ; DWORD GetModuleFileNameA(HMODULE hModule, LPSTR lpFilename, DWORD nSize)
0x0040122e      push 0x5c           ; '\' ; 92 ; int32_t arg_ch
0x00401230      lea edx, [var_118h]
0x00401236      push edx            ; int32_t arg_8h
0x00401237      call fcn.00401460
0x0040123c      add esp, 8
0x0040123f      mov dword [var_8h], eax
0x00401242      mov eax, dword [var_8h]
0x00401245      mov byte [eax], 0
0x00401248      mov edi, str.msgina32.dll ; 0x4080a4
0x0040124d      lea edx, [var_118h]
0x00401253      or ecx, 0xffffffff ; -1
0x00401256      xor eax, eax
0x00401258      repne scasb al, byte es:[edi]
0x0040125a      not ecx
0x0040125c      sub edi, ecx
0x0040125e      mov esi, edi
0x00401260      mov ebx, ecx
0x00401262      mov edi, edx
0x00401264      or ecx, 0xffffffff ; -1
0x00401267      xor eax, eax
0x00401269      repne scasb al, byte es:[edi]
0x0040126b      add edi, 0xffffffff
0x0040126e      mov ecx, ebx
0x00401270      shr ecx, 2
0x00401273      rep movsd dword es:[edi], dword ptr [esi]
0x00401275      mov ecx, ebx
0x00401277      and ecx, 3
0x0040127a      rep movsb byte es:[edi], byte ptr [esi]
0x0040127c      push 0x104          ; 260 ; int32_t arg_ch
0x00401281      lea eax, [var_118h]
0x00401287      push eax            ; int32_t arg_8h
0x00401288      call section..text  ; fcn.00401000
```

First of all, we'll *call fcn.0x00401080*. Within it we'll call the *FindResource()* and *LoadResource()* functions among others. This subroutine is then in charge of extracting the *msgina32.dll* from the executable itself. After several other calls we'll eventually jump to another subroutine with *call fcn.00401320* which will execute the following:

```
0x004013bb      push edi            ; DWORD nNumberOfBytesToWrite
0x004013bc      push dword [arg_8h] ; LPCVOID arg_ch
0x004013bf      push dword [esi + 0x10] ; DWORD arg_8h
0x004013c2      call fcn.00402302
```

This will eventually call the *WriteFile()* function which makes us be pretty sure this is in fact unpacking the payload DLL.

If we go back to *fcn.0x004011d0* we'll eventually reach a *call section… text* which will take us to *fcn.00401000*, which we include below:



We can easily see the calls to *RegCreateKeyExA()* and *RegSetValueExA()* which give away how the executable is configuring the register to force the loading of the malicious version of *msgina.dll*.

In light of how explicit information was on the executable we must conclude it was **not obfuscated**. Even though it's true that some of the strings displayed by and unfiltered *strings* are not readable the fact that we were able to get all the above with a somewhat simple static analysis prevents us from even considering this to be an **obfuscated** sample.

What's more, this executable did not import any network related libraries so it's **not leveraging** network capabilities at all.

After breaking down this file we can be certain it's an **installer**. It's only concerned with setting up the system in such a way that the malicious DLL it carries can collect the credentials it needs but once it's run it's not necessary.

That is, once the registry is set up correctly and the *msgina32.dll* is installed on the system there is no more need for this executable: it can be deleted. This makes it hard to just state the malware itself is the executable as once it's executed it becomes a 2-file combo. Nonetheless, it's distributed as this standalone executable so if we had to choose one or the other, we would have to say the malware itself is *Lab11-01.exe* itself. Nonetheless we believe it's more accurate the say that once it becomes "alive", the malware is integrated by both the executable and malicious DLL as the one ultimately retrieving the credentials is the DLL itself.

Anyway, we can now analyze the *msgina32.dll* file to take a look at its contents.

Upon opening it up in cutter we find that whenever a legit *msgina.dll* function is called it'll be transparently run. We have structures like the one below which will just call *fcn.00401000* to silently call the original functions. *fcn.00401000* will just call *GetProcAddress()* to get the location for the real function. It'll be returned on *EAX* and used for an unconditional jump as seen. Jumping unconditionally will return control to the original DLL function caller as no return address will be pushed onto the stack: that of the original caller will still be there. In this case the original caller is *winlogon*. This is based off of page 569 of [PMAL].

```
  ;-- WlxShutdown:
12: gina.dll_WlxShutdown ();
0x100013c0      push str.WlxShutdown ; 0x10003190
0x100013c5      call section..text ; fcn.10001000
0x100013ca      jmp eax
0x100013cc      int3
0x100013cd      int3
0x100013ce      int3
0x100013cf      int3
```

Anyway, we'll find that this is the case for all the common methods **except** for *WlxLoggedOutSAS*(). The body for this one is quite larger as seen in the following screenshot. Please note that we had already found the function logging user credentials and we renamed it to *logCredentials* manually so as to make the screenshot clearer:

```
114: WlxLoggedOutSAS (int32_t arg_20h_4, int32_t arg_20h_3, int32_t arg_20h_2, int32_t arg_1ch, int32_t arg_20h, int32_t arg_24h, int32_t arg_28h, int32_t arg_2ch);
; arg int32_t arg_20h_4 @ esp+0x44
; arg int32_t arg_20h_3 @ esp+0x48
; arg int32_t arg_20h_2 @ esp+0x4c
; arg int32_t arg_1ch @ esp+0x50
; arg int32_t arg_20h @ esp+0x54
; arg int32_t arg_24h @ esp+0x58
; arg int32_t arg_28h @ esp+0x5c
; arg int32_t arg_2ch @ esp+0x60
0x100014a0      push esi
0x100014a1      push edi
0x100014a2      push str.WlxLoggedOutSAS ; 0x1000328c ; LPCSTR lpProcName
0x100014a7      call section..text ; fcn.10001000
0x100014ac      push 0x64           ; 'd' ; 100
0x100014ae      mov edi, eax
0x100014b0      call sub.MSVCRT.dll_void_____cdecl_operator_new_unsigned_int
0x100014b5      mov eax, dword [arg_2ch]
0x100014b9      mov esi, dword [arg_28h]
0x100014bd      mov ecx, dword [arg_24h]
0x100014c1      mov edx, dword [arg_20h]
0x100014c5      add esp, 4
0x100014c8      push eax
0x100014c9      mov eax, dword [arg_1ch]
0x100014cd      push esi
0x100014ce      push ecx
0x100014cf      mov ecx, dword [arg_20h_2]
0x100014d3      push edx
0x100014d4      mov edx, dword [arg_20h_3]
0x100014d8      push eax
0x100014d9      mov eax, dword [arg_20h_4]
0x100014dd      push ecx
0x100014de      push edx
0x100014df      push eax
0x100014e0      call edi
0x100014e2      mov edi, eax
0x100014e4      cmp edi, 1          ; 1
0x100014e7      jne 0x1000150b
0x100014e9      mov eax, dword [esi]
0x100014eb      test eax, eax
0x100014ed      je 0x1000150b
0x100014ef      mov ecx, dword [esi + 0xc]
0x100014f2      mov edx, dword [esi + 8]
0x100014f5      push ecx
0x100014f6      mov ecx, dword [esi + 4]
0x100014f9      push edx
0x100014fa      push ecx
0x100014fb      push eax
0x100014fc      push str.UN__s_DM__s_PW__s_OLD__s ; 0x10003258
0x10001501      push 0              ; int32_t arg_874h
0x10001503      call logCredentials
0x10001508      add esp, 0x18
0x1000150b      mov eax, edi
0x1000150d      pop edi
0x1000150e      pop esi
0x1000150f      ret 0x20
```

After calling the real *WlxLoggedOutSAS()* the user credentials will be logged to a location specified within *logCredentials* itself.

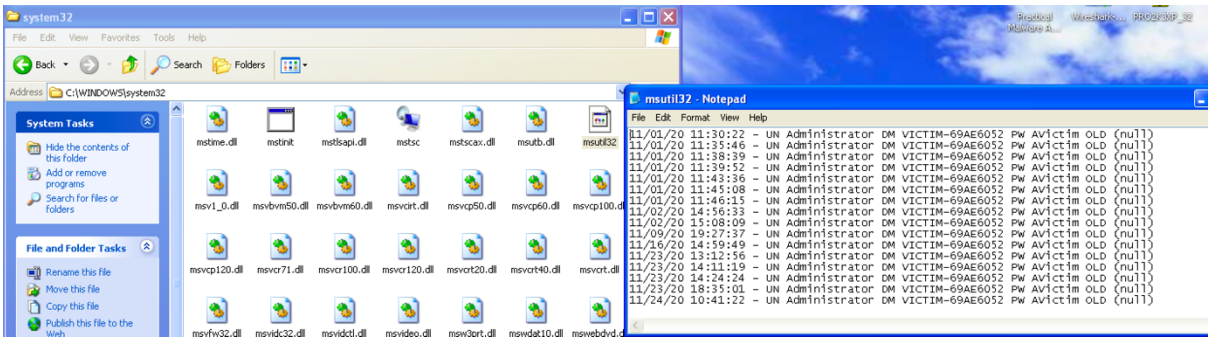
If we now inspect *logCredentials* we'll see the following:

```
231: logCredentials (int32_t arg_874h, int32_t arg_8h, int32_t arg_860h);
; var HLOCAL hMem @ esp+0x18
; var int32_t var_ch_2 @ esp+0x34
; var int32_t var_ch @ esp+0x38
; var int32_t var_38h @ esp+0x60
; var int32_t var_54h @ esp+0x88
; arg int32_t arg_874h @ esp+0x88c
; arg int32_t arg_8h @ esp+0x890
; arg int32_t arg_860h @ esp+0x894
0x10001570      mov ecx, dword [arg_8h]
0x10001574      sub esp, 0x854
0x1000157a      lea eax, [arg_860h]
0x10001581      lea edx, [var_54h]
0x10001585      push esi
0x10001586      push eax
0x10001587      push ecx
0x10001588      push 0x800          ; 2048
0x1000158d      push edx
0x1000158e      call sub.MSVCRT.dll__vsnwprintf
0x10001593      push 0x10003320     ; ' 3'
0x10001598      push str.msutil32.sys ; 0x100032f8
0x1000159d      call sub.MSVCRT.dll__wfopen
0x100015a2      mov esi, eax
0x100015a4      add esp, 0x18
0x100015a7      test esi, esi
0x100015a9      je 0x1000164f
0x100015af      lea eax, [var_54h]
0x100015b3      push edi
0x100015b4      lea ecx, [var_ch]
0x100015b8      push eax
0x100015b9      push ecx
0x100015ba      call sub.MSVCRT.dll__wstrtime
0x100015bf      add esp, 4
0x100015c2      lea edx, [var_38h]
0x100015c6      push eax
0x100015c7      push edx
0x100015c8      call sub.MSVCRT.dll__wstrdate
0x100015cd      add esp, 4
0x100015d0      push eax
0x100015d1      push str.s__s____s ; 0x100032e0 ; const wchar_t *format
0x100015d6      push esi            ; FILE *stream
0x100015d7      call sub.MSVCRT.dll_fwprintf ; int fwprintf(FILE *stream, const wchar_t *format)
0x100015dc      mov edi, dword [arg_874h]
0x100015e3      add esp, 0x14
0x100015e6      test edi, edi
0x100015e8      je 0x10001637
0x100015ea      push 0
0x100015ec      lea eax, [var_ch_2]
0x100015f0      push 0
0x100015f2      push eax
0x100015f3      push 0x409          ; 1033
0x100015f8      push edi
0x100015f9      push 0              ; LPCVOID lpSource
0x100015fb      push 0x1100         ; DWORD dwFlags
0x10001600      call dword [FormatMessageW] ; 0x100020c ; DWORD FormatMessageW(DWORD dwFlags, LPCVOID lpSource, DWORD dwMessageId, DWORD dwLanguageId, LPWSTR lpBuffer, DWORD nSize, va_list *Arguments)
0x10001606      mov ecx, dword [hMem]
0x1000160a      push ecx
0x1000160b      push edi
0x1000160c      push 0x100032a0     ; const wchar_t *format
0x10001611      push esi            ; FILE *stream
0x10001612      call sub.MSVCRT.dll_fwprintf ; int fwprintf(FILE *stream, const wchar_t *format)
```

We have already added a red box around the important calls. The thing to take out is that the filename *msutil32.sys* is provided as an argument before opening a file. Then, this must be the file these credentials are being logged to! As the DLL is imported by *winlogon* which sits on *C:\WINDOWS\system32* we can expect to find it just there. We should point out this string did show up when running *strings* on the sample.

As the *WlxLoggedOutSAS()* method is a *hook* that runs when the user logs out then the credentials will be stolen right there and then for any user.

In any case, if we get the file and take a look at its contents, we'll see the following:



It's a regular text file with our credentials! Each entry corresponds to one logout. We can check that if we delete it it'll be created once more after we log out, provided the malicious malware is still running.

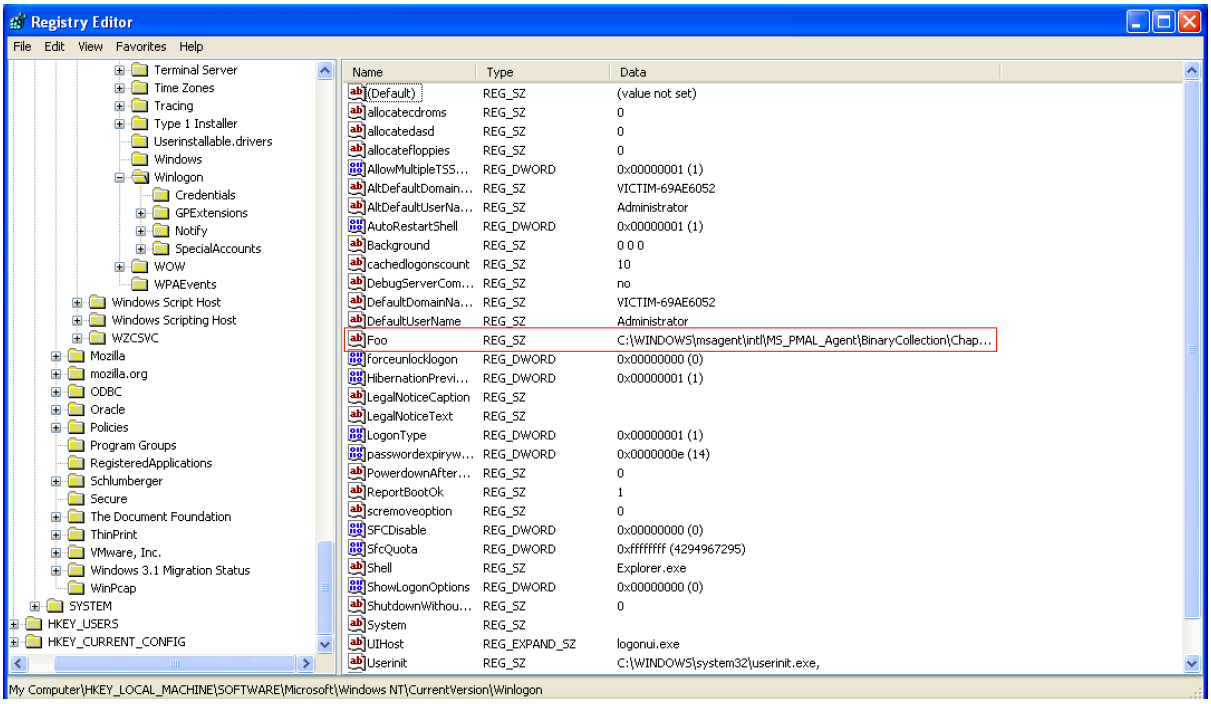
Note the format agrees with the following string included in the DLL and referenced from *logCredentials*. It's printing out 4 strings (%s): our username, our domain name, our current password and the old one. This is the information we indeed see above. The format string is included in the next picture

```
0x10003258          .string "UN %s DM %s PW %s OLD %s" ; len=50
```

3.2 Removal process

The way to get rid of this nasty program is to just delete the associated registry entry we discussed at the beginning of the analysis. It's **not** necessary to download a legitimate copy of *msgina.dll* as it was already being included by *winlogon.exe*. We just need to navigate through the registry and either delete or rename the *GinaDLL* key. We chose to rename it to *Foo* so that we could test things out should something not work as expected

In order to test our hypothesis we just need to delete the *msutil32.sys* file and logout to see if it's created again after deleting the GinaDLL registry entry.
As expected, this wasn't the case: we just need to delete/rename the registry key. What's more, upon deleting or renaming said key we'll see a change to the login screens style: we shouldn't forget we are altering the path the calls to GINA modules take so we can expect small changes such as this to happen.



3.3 Summing up

Summing up, we see how *Lab11-01.exe* is a common **PE executable** (as shown by the file command) installed a malicious DLL that got between *winlogon* and *msgina.dll* in such a way that it retrieved user login credentials upon user logout. It did so by adding the *GinaDLL* entry in the registry and pointing it to the malicious DLL. Then, upon execution of *WlxLoggedOutSAS()* the credentials were being written to *C:\WINDOWS\system32\msutil32.sys* which was nothing more than a text file. Deleting or renaming said registry entry is enough to take care of the entire problem. Note the above is based on pages 566-571 of [PMAL].

4. The third malware sample

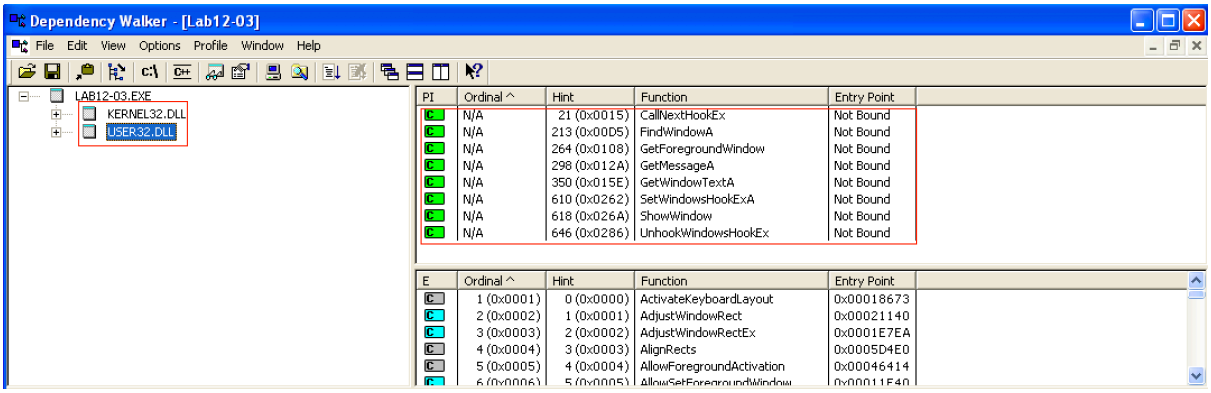
4.1 Malware study

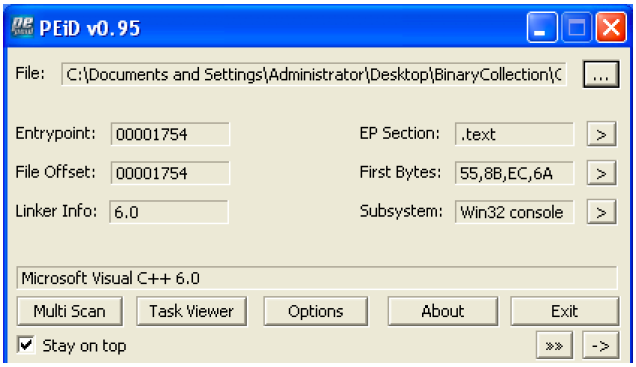
Once we have discovered the *C:\WINDOWS\msagent\intl\MS_PMAL_Agent\BinaryCollection* directory we can just dive deeper into it to find a complete collection of rather suspicious executables.

After browsing around them we settled on analyzing the *Lab12-03.exe* file next living under the *Chapter_12L* directory belonging to the aforementioned one.

We'll begin by running a common static analysis on it. In doing so we'll launch *DependencyWalker* against it to get the library functions it's importing. Even though not a whole bunch of them show up we can conclude several things:

- The sample is **not** packed. Despite the low number of imports, they convey a whole lot of information about the functionality of the sample as we'll see later. Refer to the screenshot below for a highlighted list of crucial imported functions. What's more, we are also including a screenshot of *PEiD's* output showing it's been compiled with Microsoft Visual C++ which is not by any means a packer.
- The process **won't use any network capabilities** as no networking-related DLL is imported. We are also sure it's not hijacking or injecting itself into any other projects as we don't import the *WriteProcessMemory()* function from *kernel32.dll* so we won't have a tangent access to networking recourses either.
- The file itself is a PE executable file as shown by the *file* command. We also ran *PEview* on the sample but as nothing seems to be out of the ordinary we believe it's of no use including any screenshots whatsoever: we are up against a very common executable file with regards to its format.





We would like to draw special attention towards the *SetWindowsHookExA()* function shown in the *Dependency Walker* screenshot. As stated in the Microsoft Developer Network (MSDN) Documentation, the *SetWindowsHooExkA()* function will provide a callback, a function that is to be called upon the occurrence of a given type of event. In other words, our sample will just "subscribe" to a certain occurrence and will be notified by the OS itself when it happens so that it can take the appropriate actions. Upon notification, the callback of *hook* function will be called.

We then need to delve deeper into the code to try and see what the callback function is and what type of event our sample is subscribing to. To do that, we'll employ *cutter*. Inspecting the documentation we linked above we see the first parameter

```
  ;-- section..text:
134: int main (int argc, char **argv, char **envp);
; var HWND hWnd @ ebp-0x8
; var HHOOK hhk @ ebp-0x4
0x00401000      push ebp                ; [00] -r-x section size 12288 named .text
0x00401001      mov ebp, esp
0x00401003      sub esp, 8
0x00401006      mov dword [hhk], 0
0x0040100d      call dword [AllocConsole] ; 0x404004 ; BOOL AllocConsole(void)
0x00401013      push 0
0x00401015      push str.ConsoleWindowClass ; 0x405040 ; LPCSTR lpClassName
0x0040101a      call dword [FindWindowA] ; 0x4040b0 ; HWND FindWindowA(LPCSTR lpClassName, LPCSTR lpWindowName)
0x00401020      mov dword [hWnd], eax
0x00401023      cmp dword [hWnd], 0
0x00401027      je 0x401035
0x00401029      push 0                  ; int nCmdShow
0x0040102b      mov eax, dword [hWnd]
0x0040102e      push eax                ; HWND hWnd
0x0040102f      call dword [ShowWindow] ; 0x4040b4 ; BOOL ShowWindow(HWND hWnd, int nCmdShow)
0x00401035      push 0x400              ; 1024 ; int32_t arg_ch
0x0040103a      push 1                  ; 1 ; int32_t arg_8h
0x0040103c      push 0x405350           ; 'PS@' ; int32_t arg_4h
0x00401041      call fcn.004014f0
0x00401046      add esp, 0xc
0x00401049      push 0
0x0040104b      push 0                  ; DWORD dwThreadId
0x0040104d      call dword [GetModuleHandleA] ; 0x404000 ; HMODULE GetModuleHandleA(LPCSTR lpModuleName)
0x00401053      push eax                ; HINSTANCE hmod
0x00401054      push keyboardHook       ; 0x401086 ; HOOKPROC lpfn
0x00401059      push 0xd                ; 13 ; int idHook
0x0040105b      call dword [SetWindowsHookExA] ; 0x4040b8 ; HHOOK SetWindowsHookExA(int idHook, HOOKPROC lpfn, HINSTANCE hmod, DWORD dwThreadId)
0x00401061      mov dword [hhk], eax
0x00401064      push 0
0x00401066      push 0
0x00401068      push 0
0x0040106a      push 0                  ; LPMSG lpMsg
0x0040106c      call dword [GetMessageA] ; 0x4040bc ; BOOL GetMessageA(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax)
0x00401072      test eax, eax
0x00401074      je 0x401078
0x00401076      jmp 0x401064
0x00401078      mov ecx, dword [hhk]
0x0040107b      push ecx                ; HHOOK hhk
0x0040107c      call dword [UnhookWindowsHookEx] ; 0x4040c0 ; BOOL UnhookWindowsHookEx(HHOOK hhk)
0x00401082      mov esp, ebp
0x00401084      pop ebp
0x00401085      ret
```

to the *SetWindowsHookExA()* function determines the type of event it'll listen for. As the parameters are passed in a reverse order (the last one that's referenced is the first positional argument) due to the stacks LIFO (Last In First Out) nature we can look up the value of the *idHook* parameter (13 == 0xD) shown in the next instruction on the linked documentation. In doing so we'll see that the value stands for *WH_KEYBOARD_LL* which "Installs a hook procedure that monitors low-level keyboard input events" according to the site.

We then see how this program is concerned with keyboard input which makes us believe it's some form of a keylogger right from the beginning. Having this lead early on will let us carry on a much more precise analysis because we now know what we are looking for. We can also see in the screenshot above that the callback is provided as the second parameter. In our case it'll be the *keyboardHook* subroutine. Please note we have renamed the subroutines involved in the malware's functionality to facilitate our work. This is the case for *main*, *keboardHook* and *writeLog*. The former two can be seen in the screenshot above.

We would also like to draw your attention to the while loop we have shown in the image that will call *GetMessageA()* continuously. As seen in the MSDN documentation, this function will block the caller (i.e. the malware) until a message related to one of the events it's subscribed to is received. Upon said reception the hook function we have defined will be called to handle it. If we were not to call this function we would effectively subscribe to an event to then not ask for any updates and thus, the malware's operation wouldn't be the expected one.

The above is all there is to the *main()* function it will just wait in that loop indefinitely. We should then proceed to analyze the *keyboardHook* subroutine to see what it's executing.

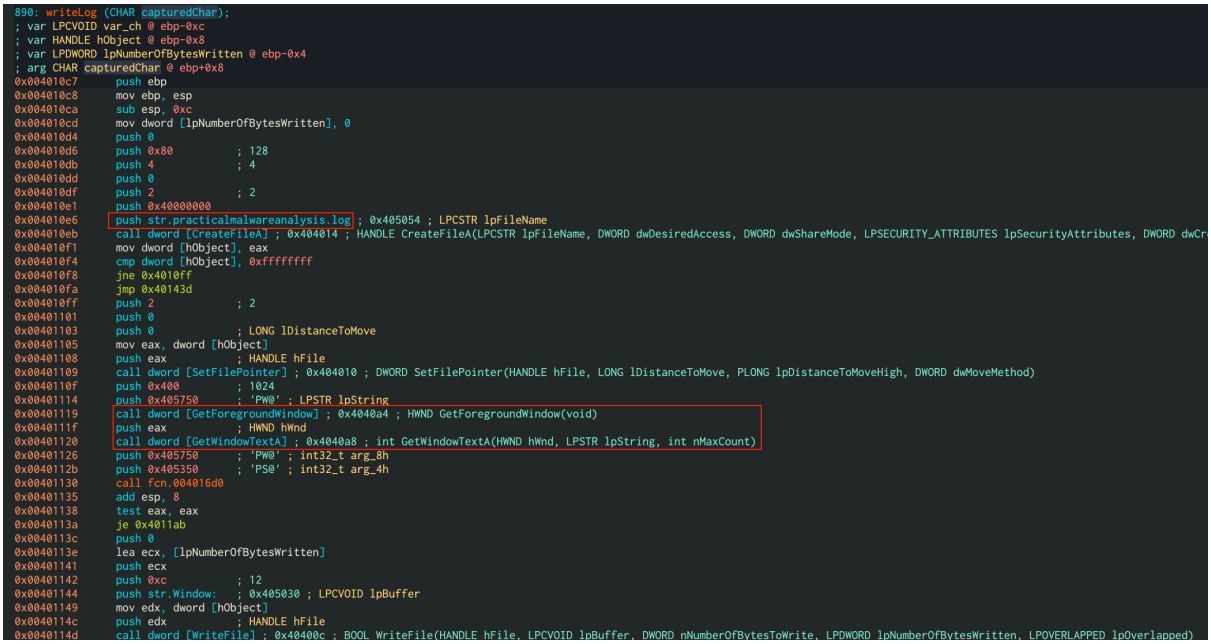
We'll begin by inspecting the disassembled code attached in the image below. We can see that the method is pretty straightforward. As we stated before it'll be called on keyboard events and it'll receive 3 parameters as seen in the MSDN. The first one is of little concern to us and we only need to know that, as seen in the documentation, if the valid is not 0 we should abort immediately and return. Whatever *CallNextHookEx()* returns. The second parameter one is telling the hook what happened with the key and we'll only delve deeper into the function if we have a key that's down no matter if we were pressing ALT at the same time or not. We find the values shown in the disassembly in these two sites of the MSDN. The third parameter is a *KBDLLHOOKSTRUCT* containing

information about the pressed key. The filed we are the most interested in is the first one which contains the *Virtual Key Code* associated with the event. Note that the parameter is passed by referenced to the hook, that is, we are passed a memory address. Also note the *vkCode* struct member is the first one, that is, it's at offset 0 from the passed pointer. That's why on the marked lines we see that this value is read into ECX and then pushed so that the next function in line, *writeLog* can use it. We have renamed the interesting parts so that they are easier to spot. We should also mention that the call to *CallNextHookEx()* is "not" mandatory but tremendously encouraged as seen here.

```
65: keyboardHook (int32_t nCode, int32_t wParam, KBDLLHOOKSTRUCT*lParam);
; arg int32_t nCode @ ebp+0x8
; arg int32_t wParam @ ebp+0xc
; arg KBDLLHOOKSTRUCT*lParam @ ebp+0x10
0x00401086      push ebp
0x00401087      mov ebp, esp
0x00401089      cmp dword [nCode], 0
0x0040108d      jne 0x4010af
0x0040108f      cmp dword [wParam], 0x104
0x00401096      je 0x4010a1                    We apply offset 0 as that's
0x00401098      cmp dword [wParam], 0x100      where the vkCode memebr is!
0x0040109f      jne 0x4010af
0x004010a1      mov eax, dword [lParam]
0x004010a4      mov ecx, dword [eax]
0x004010a6      push ecx
0x004010a7      call fcn.004010c7  ; writeLog
0x004010ac      add esp, 4
0x004010af      mov edx, dword [lParam]
0x004010b2      push edx
0x004010b3      mov eax, dword [wParam]
0x004010b6      push eax
0x004010b7      mov ecx, dword [nCode]
0x004010ba      push ecx
0x004010bb      push 0
0x004010bd      call dword [CallNextHookEx] ; 0x4040ac ; LRESULT CallNextHookEx(HHOOK hhk, int nCode, WPARAM wParam, LPARAM lParam)
0x004010c3      pop ebp
0x004010c4      ret 0xc
```

As discussed, the *writeLog* function will receive a single parameter (*capturedChar*, we renamed it) that's the key associated with the keyboard event. We could just try to write it to a log file, but we would then incur into some problems.

First of all, we wouldn't know the program the key was pressed on. We could try to get the program's name to get a bit more context so as to where the key was pressed. Then, if the key we pressed is NOT a printable character we need to find a way of representing it with a string. That is, if we press BACKSPACE, we should write BACKSPACE to the log file, not the associated code as that's not meaningful.



Taking the above into account we can take a look at the *writeLog* function we are attaching below and, as we read through it, we'll notice several aspects:

- At the very beginning there is a call to *CreateFileA()* which is passed the "practical*malwareanalysis.log*" parameter. This already appeared in the *strings* output and hints that the file everything is being logged to is indeed "*practicalmalwareanalysis.org*". The file descriptor for that file is stored into *iVar1* in the screenshot below.
- We then see a couple of calls to *GetForegroundWindow()* and *GetWindowTextA()* respectively. This will first get the window the key was pressed on and then it's title so that we can add it to the log. Note the later call to *WriteFile()*.
- We've got several other calls to *WriteFile()* and a lot of jumps. Let's make use of *cutter's* decompiler to try and get a clearer view of the functions structure.

The above is pointed out in the following screenshot:

```
void __cdecl fcn.004010c7(uint32_t capturedChar)
{
    int32_t iVar1;
    undefined4 uVar2;
    int32_t iVar3;
    undefined4 var_ch;
    undefined4 hObject;
    undefined4 lpNumberOfBytesWritten;

    lpNumberOfBytesWritten = 0;
    iVar1 = (*_CreateFileA)("practicalmalwareanalysis.log", 0x40000000, 2, 0, 4, 0x80, 0);
    if (iVar1 != -1) {
        (*_SetFilePointer)(iVar1, 0, 0, 2);
        uVar2 = (*_GetForegroundWindow)(0x405750, 0x400);
        (*_GetWindowTextA)(uVar2);
        iVar3 = fcn.004016d0(0x405350, 0x405750);
        if (iVar3 != 0) {
            (*_WriteFile)(iVar1, "\r\n[Window: ", 0xc, &lpNumberOfBytesWritten, 0);
            uVar2 = fcn.00401650(0x405750, &lpNumberOfBytesWritten, 0);
            (*_WriteFile)(iVar1, 0x405750, uVar2);
            (*_WriteFile)(iVar1, 0x40503c, 4, &lpNumberOfBytesWritten, 0);
            fcn.00401550(0x405350, 0x405750, 0x3ff);
            *(undefined *)0x40574f = 0;
        }
        if ((capturedChar < 0x27) || (0x40 < capturedChar)) {
            if ((capturedChar < 0x41) || (0x5a < capturedChar)) {
    // switch table (19 cases) at 0x401441
                switch(capturedChar) {
                case 8:
                    uVar2 = fcn.00401650(0x40508c, &lpNumberOfBytesWritten, 0);
                    (*_WriteFile)(iVar1, "BACKSPACE", uVar2);
                    break;
                case 9:
                    (*_WriteFile)(iVar1, "[TAB]", 5, &lpNumberOfBytesWritten, 0);
                    break;
                case 0xd:
                    (*_WriteFile)(iVar1, "\n[ENTER]", 8, &lpNumberOfBytesWritten, 0);
                    break;
                case 0x10:
                    (*_WriteFile)(iVar1, "[SHIFT]", 7, &lpNumberOfBytesWritten, 0);
                    break;
                case 0x11:
                    (*_WriteFile)(iVar1, "[CTRL]", 6, &lpNumberOfBytesWritten, 0);
                    break;
```

If we pay closer attention to the switch statement that appears, we'll see how it handles the "string conversion" of the non-printable keys. In order to do so we'll try to walk through what happens if the backspace key is pressed on the disassembled code.

According to the site we linked before, the virtual key code for BACKSPACE is 0x08. Now, according to the following screenshot we'll subtract 8 from it and use it as an offset to index address 0x00401220. As the MOV instruction uses the CL operand as the destination we'll only move the byte addressed by the base + offset combination, nothing more. Our offset is then 0x8 - 0x8 = 0x0; the first byte in the table.

```
mov edx, dword [capturedChar]
mov dword [jumpTableIndex], edx
mov eax, dword [jumpTableIndex]
sub eax, 8
mov dword [jumpTableIndex], eax
cmp dword [jumpTableIndex], 0x61
ja case.0x401226.18
mov edx, dword [jumpTableIndex]
xor ecx, ecx    Table with jump addresses
mov cl, byte [edx + 0x40148d]
```

```
0x0040148d      .byte 0x00
0x0040148e      .byte 0x01
0x0040148f      .byte 0x12
0x00401490      .byte 0x12
0x00401491      .byte 0x12
0x00401492      .byte 0x02    Jump address table!
0x00401493      .byte 0x12
0x00401494      .byte 0x12
0x00401495      .byte 0x03
0x00401496      .byte 0x04
0x00401497      .byte 0x12
```

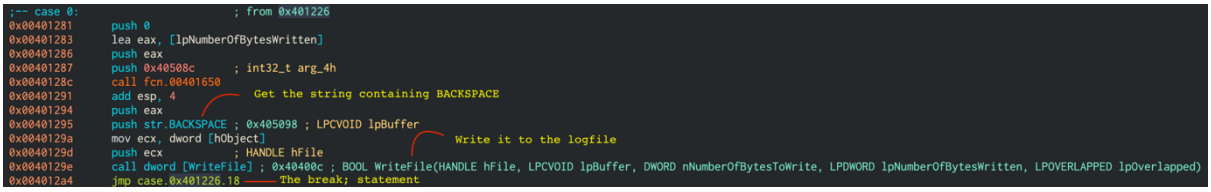
The table image we are showing above has been obtained by showing the contents of that region as data and not instructions. We can then see that the first few elements are: 0x0, 0x1, 0x12, 0x12, 0x12, 0x2, 0x12, 0x12, 0x3, 0x4, 0x12… As we said before our offset is 0 so that value 0x0, the first element in the table, will be loaded into CL. We'll use that value, multiplied by 4, as an offset into the *switch's* table that's located at address 0x401441 as seen in the following screenshot.

```
mov cl, byte [edx + 0x40148d]
jmp dword [ecx*4 + switchAddresses] ; 0x401441 ; switch table (19 cases) at 0x401441
```

If we take a look at the switch table attached in the image below, we'll see that the first entry corresponds to entry 0. We would like to point out that we have discovered exactly how a *switch* is implemented in assembly: a table with the addresses to the different cases where we use the input parameter (multiplied by 4 as we are dealing with 32-bit (4 byte) addresses) as an offset within it to jump to the appropriate case.

```
;-- switchAddresses:
0x00401441      .int32 4199041 ; case.0x401226.0
0x00401445      .int32 4199081 ; case.0x401226.1
0x00401449      .int32 4199013 ; case.0x401226.2
0x0040144d      .int32 4198985 ; case.0x401226.3
0x00401451      .int32 4199109 ; case.0x401226.4
0x00401455      .int32 4199433 ; case.0x401226.5
0x00401459      .int32 4198957 ; case.0x401226.6
0x0040145d      .int32 4199137 ; case.0x401226.7
0x00401461      .int32 4199165 ; case.0x401226.8
0x00401465      .int32 4199193 ; case.0x401226.9
0x00401469      .int32 4199221 ; case.0x401226.10
0x0040146d      .int32 4199249 ; case.0x401226.11
0x00401471      .int32 4199277 ; case.0x401226.12
0x00401475      .int32 4199305 ; case.0x401226.13
0x00401479      .int32 4199333 ; case.0x401226.14
0x0040147d      .int32 4199358 ; case.0x401226.15
0x00401481      .int32 4199383 ; case.0x401226.16
0x00401485      .int32 4199408 ; case.0x401226.17
0x00401489      .int32 4199468 ; case.0x401226.18 ; case.default.0x401226
```

Either way, we see how the first entry corresponds to *case 0* and if we take a look at it, we'll find the following:



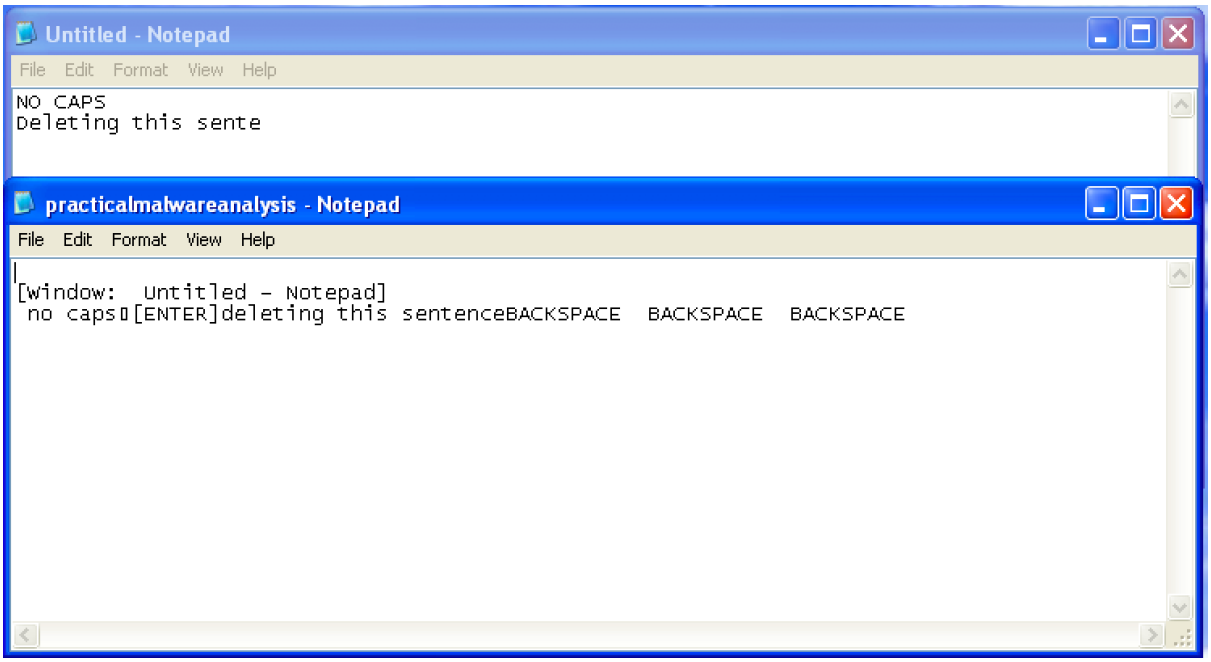
We should mention that the cases are not in order in the disassembled code. This is most likely due to alignment requirements so that the program runs faster by making memory accesses to it more efficient, but we can't be sure.

The way keys are recorder into the log is a little bit different. Instead of going through the *switch* we'll instead write the passed parameter directly after adding 0x20 to it. We'll see the reason for that value and how it makes the keylogger only record keystrokes as lowercase keys rather than upper case ones. We shouldn't forget we also know if the SHIFT key is being pressed so we can infer when the user is typing with caps. To check the above we can refer to the decompiled version of the function again as it'll make our life way easier:

```
} else {
    capturedChar = capturedChar + 0x20;
    (*_WriteFile)(iVar1, &capturedChar, 1, &lpNumberOfBytesWritten, 0);
```

We find that if the virtual key code is between 0x41 and 0x5A included (the ASCII and virtual key codes for 'A' and 'Z', respectively) we'll just increment the number by 0x20 and write it to the file. Then, if we capture the 'A' key we'll get 0x41 + 0x20 = 0x61 = 'a', the ASCII code for lower case A. This is what we'll ultimately see written to the file! We'll check this is true when attaching some screenshots showing the malware's generated logs.

Either way, we have broken down the code and we know that the program will start, set a hook for keyboard events and log the pressed keys to the "practicalmalwareanalysis.log" file living in the same directory as the executable. In order to test this, we can just run the program. And open up a notepad to write stuff into. Notice that the log file will be appended to if it exists or created if it doesn't. We started with a blank one to generate the following screenshot:



Now that we know the malware is running, we can just open up *ProcessExplorer* and we'll effectively see it appear as a process. To stop the keylogger we just need to kill that process. As it's taking no measures to ensure it's persistent, such as dropping keys into the registry, it will only start if clicked on by the user. Then, killing the process and removing the executable from the system ensures the malware's complete removal. Even though it may not be such a serious thread with regards to how it won't start automatically it's carrying out its keylogging functionality perfectly fine.

We would like to mention that if you open up several keylogger processes they'll begin fighting among themselves and the log will be somewhat incomplete as there is no synchronization mechanism controlling that the file access is done in an orderly manner.

You'll probably be wondering why we haven't carried out a dynamic analysis per-se. As we were able to uncover most, if not all, of the malware's functionality there was no need for this kind of analysis, we just ran the program without attaching any tools to confirm what we already suspected. As we stated during the introduction it's of no use to make an analysis deeper than it needs to be. We already knew what to expect, we checked that was actually the case too, so we don't have to continue digging at all.

4.2 Removal process

As the malware is not trying to start automatically the removal process will be easier than before. We can check that by looking for the "practicalmalwareanalysis.log" file across

all the filesystem. As we only found the one we knew existed we can be sure no other instances were running camouflaged within other processes. We can just kill all its process instances and delete the executable to completely remove it. We also saw that it wasn't packed or obfuscated in any way despite finding some unreadable strings as all the crucial information was free for us to read and interpret. No networking capabilities are employed either as no networking libraries are included. Through the study of this piece of malware we have become familiar with Windows hook functions as well as with the assembly language constructs commonly used within this operating system. We are now ready to analyse our last malware sample!

4.3 Summing up

Summing up, we see how the malware is contained in a PE executable that implements a keylogger leveraging Windows hook functions. Once the executable is run all the keyboard input will be logged out to a text file in the same folder as the executable.

Please note this discussion is based on pages 597-599 from [PMAL].

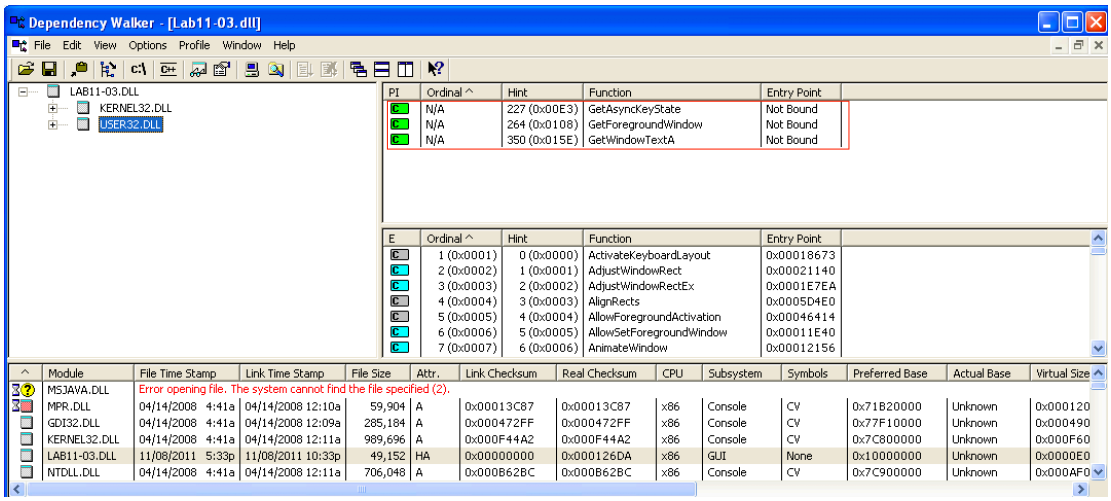
5. The fourth malware sample

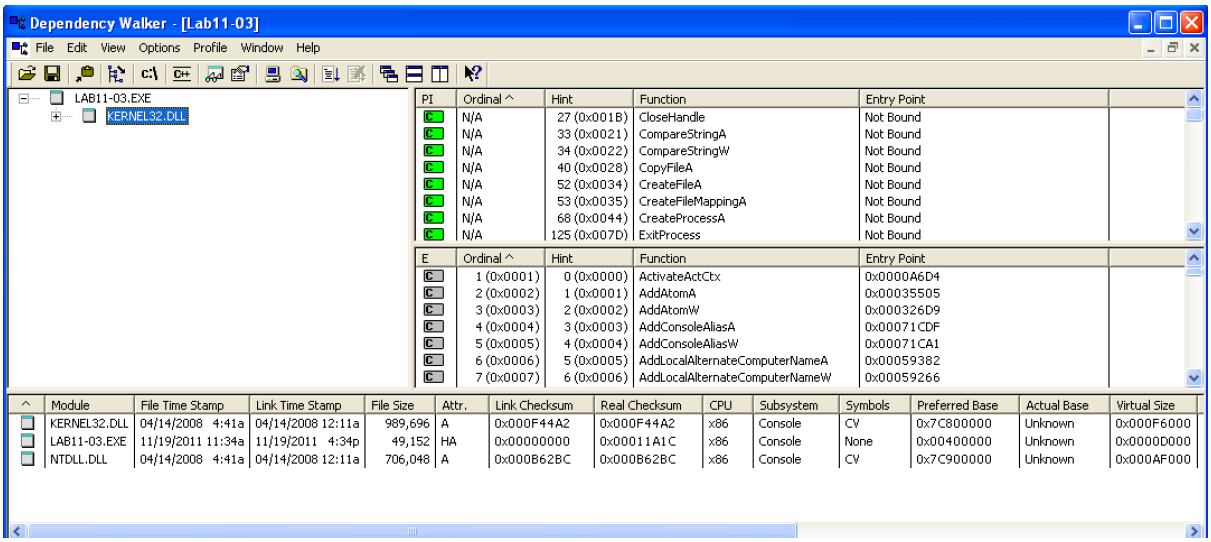
5. 1 Malware study

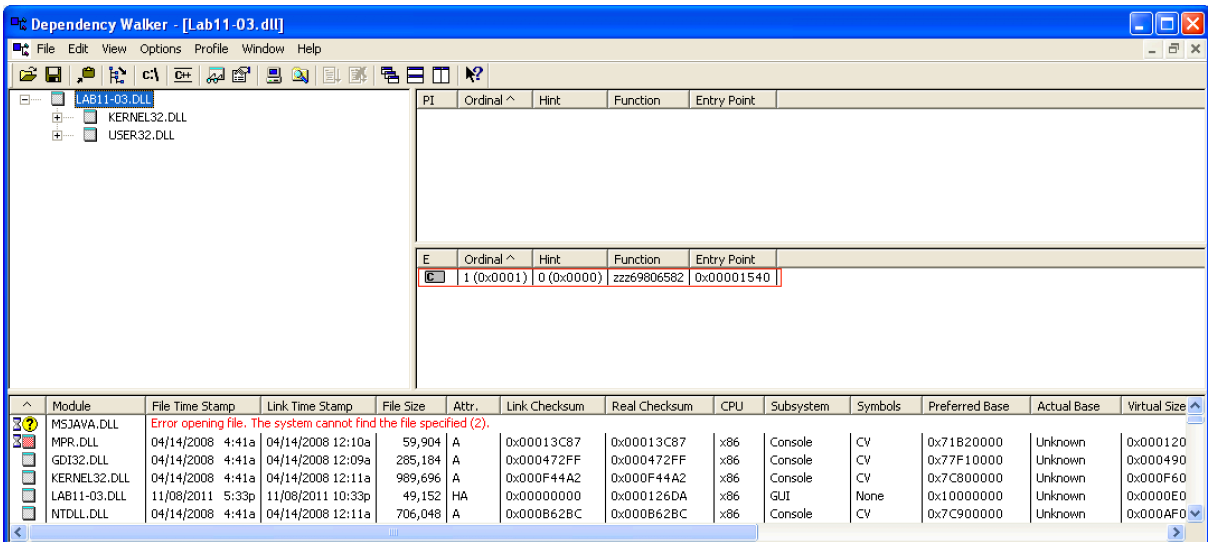
We'll again browse around the *MS_PMAL_Agent* directory to try and analyse some other of the samples with the hope of finding some malware to look at. The ensuing discussion is based on pages 239-241 and 581-586 of [PMAL].

In doing so we stumble upon the *Lab11-03.exe* and *Lab11-03.ddl* files. We'll begin, as always, by checking the dependencies both import and export as well as the strings they contain. In doing so we get the following results:

- Taking a look at the DLL's imports raises a huge red flag as we find the *GetAsyncKeyState()*, *GetForegroundWindow()* and *GetWindowTextA()*. We can recall the latter 2 from the previous sample and we already know they want to try and find the active window and get its title, respectively. After taking a look at the first function's documentation on the MSDN we see that, when called, it'll get the state of the key we pass as a parameter. You'll also note that we need to pass the key's virtual key code, just what we read in the previous sample. These imports put us on the track of a keylogger. We should nonetheless point out that it's very different to the previous one we analysed with respect to its implementation. This sample uses a technique known as *polling* whilst the previous one based its operation on the use of asynchronous events handled by a hook function. After analysing this program in its entirety we'll compare both approaches in more depth.
- The executable's imports are rather scarce, so we expect it to be carrying out some very specific function after not being needed anymore. Note how the keylogging functionality is most likely implemented in the DLL.
- The DLL exports a strangely named function: *zzz69806582*. We'll need to keep an eye out for it in the future.





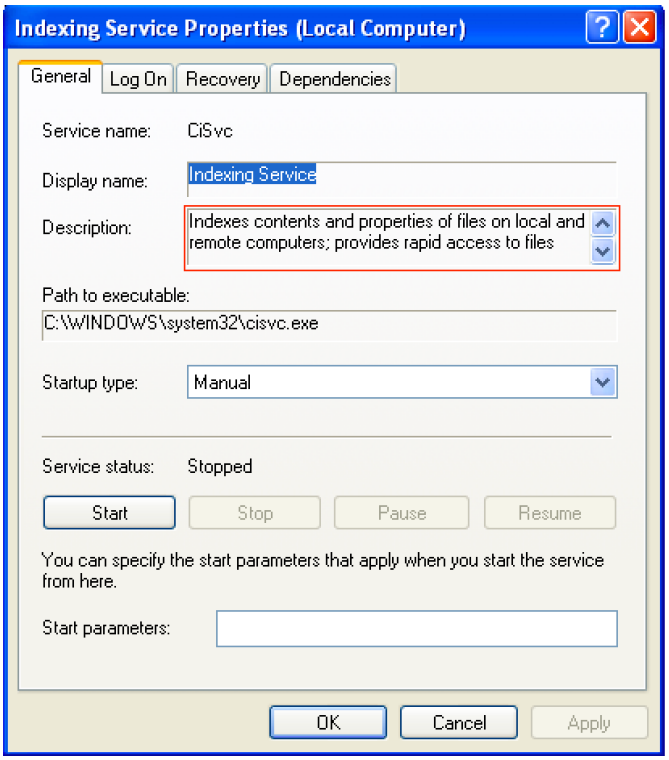


We also ran *strings* on both samples to find most of them were perfectly visible. We found the following suggestive entries in the executable file. Please note we have issued that complex command to combine several differentiated parts of the output so that they would fit in a single screenshot.

```
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/Trojan_keylogger$ strings Lab11-03.exe | tail -n 5; strings Lab11-03.exe| grep --color=never cmd
net start cisvc
C:\WINDOWS\System32\%s
cisvc.exe
Lab11-03.dll
C:\WINDOWS\System32\inet_epar32.dll
cmd.exe
.cmd
```

This output suggests that the executable is starting the *Cisvc* indexing service (hence the call to *net start cisvc*). This is also backed by the presence of the *cmd.exe* string which implies that this command will be issued through a command prompt. The fact that the *Lab11-03.dll* and *inet_epar32.dll* names appear suggests that we may be copying the *Lab11-03.dll* to some location on the victim's drive. Seeing *C:\WINDOWS\System32\%s* also makes us believe the final destination of the copy can be the *system32* directory. We'll

confirm or refute our suspicions when looking into the disassembled executable later on. We would like to point out that the *Cisvc* service is in charge of indexing files in the system so that searches and the file manager can run faster as seen in its service entry:



Inspecting the strings within the DLL only made us reaffirm ourselves in the thought of it implementing a keylogger. We found strings with textual representations of keys such as SHIFT, the path to a file called *kernel64x.dll* and references to *WriteFile()*, needed for updating the file containing the pressed keys, and *Sleep()*, which can be called for freeing system resources so that the keylogger doesn't bog down the computer. Remember that as this keylogger implements its functionality through polling it will run an infinite while loop that can be very heavy on resources…

```
0x%x
<SHIFT>
%s: %s
C:\WINDOWS\System32\kernel64x.dll
```

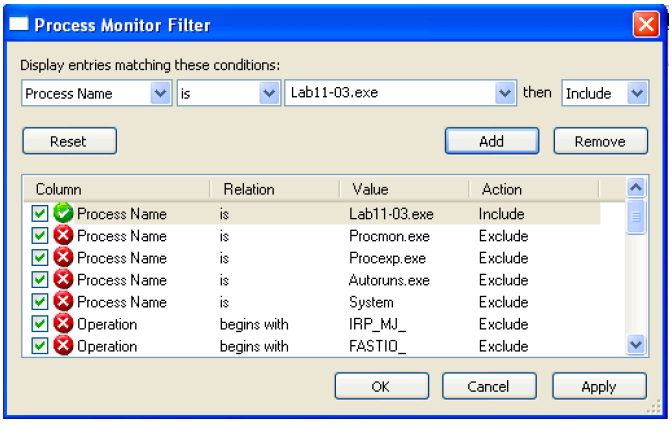
```
H:mm:ss
dddd, MMMM dd, yyyy
M/d/yy
December
November
October
September
August
July
June
April
March
February
January
Saturday
Friday
Thursday
Wednesday
Tuesday
Monday
Sunday
SunMonTueWedThuFriSat
JanFebMarAprMayJunJulAugSepOctNovDec
Sleep
WriteFile
```

```
=!=l=
?#?A?^?v?
1+1>1E1W1_1o1
1!232S2X2
2?3E3S3
5;5U5\5`5d5h5l5p5t5x5
5:6E6`6g6l6p6t6
7Z7`7d7h7l7
8h;n;u;
>.>3>B>H>X>c>u>
2!2(2.282>2C2I2Y2b2|2
4$4,424:4C4L4
555;5C5R5
6h7u7
:$:5:Q:l:y:
;:;g;{;
<5<=<f<s<x<
<K=R=k=
-0H0P0V0\0
081w1
20272?2D2H2L2u2
2"3(3,30343
4M4T4X4\4`4d4h4l4p4
5A7T7
:':_:q:
0t0x0|0
0t2|2
5@8D8H8L8
; ;$;(;,;0;4;8;<;@;D;H;L;P;T;X;\;`;d;h;l;p;t;x;|;
```

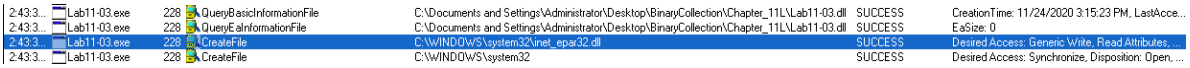
It's also true that we found some unreadable strings within the DLL. This could imply it's somewhat obfuscated but again, given how much information we have been able to extract this might as well not be the case. What we can be sure of however is that none of the files is packed. Both are detected by PEiD as being compiled by Visual Studio which is by no means an obfuscation technique. Taking a look at them with PEview showed no strange sections either and given none import any network related libraries we can be sure neither of them will leverage the network for anything. As one could guess, both files stick to the PE file type as well. As the above is very common we have decided not to bloat the report with yet more images, so we are not including them. They are pretty similar to the ones we have already provided previously.

When studying the previous samples, we have taken more of a static approach towards the analysis. We'll try to employ more dynamic-style techniques in this case to show how they can also be extremely powerful.

In doing so we'll run the sample with *ProcessMonitor* active so that we can take a look at what it's doing. We'll need to set up a filter for that just as we show in the next screenshot. We are also



including an image showing *ProcessMonitor's* output when the sample is running.



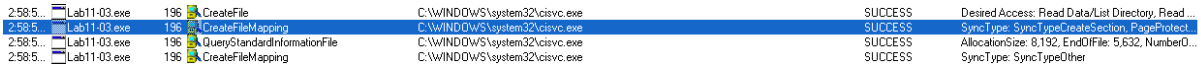
We find that the process is creating file *C:\WINDOWS\system32\inet_eapar32.dll*. We'll compare this one with the *Lab11-03.dll* file we already have to see if there is any difference. As shown in the screenshot below, there is none. We would also like to point out that the *inet_eapar32.dll* file was hidden (H attribute) within the *system32* directory, something that arises our suspicions as no other DLLs are hidden in that directory.

```
pablo@hoth:0:~/Repos/malware_analysis/Cw_2_malware_hunting/Malware_samples/Trojan_keylogger$ diff -s inet_epar32.dll Lab11-03.dll
Files inet_epar32.dll and Lab11-03.dll are identical
```

```
C:\>attrib WINDOWS\system32\inet_epar32.dll
A   H       C:\WINDOWS\system32\inet_epar32.dll
```

Either way, we now know the executable is copying *Lab11-03.dll* to *C:\WINDOWS\system32\inet_eapar32.dll*.

If we continue dissecting *ProcessMonitor*'s output, we'll see that there are indeed references to *cisvc.exe* as seen below.



We'll now begin looking at the disassembled code for the executable to try and see how it's managing to make the keylogger work in the first place.

We begin by seeing that, as we expected, the *Lab11-03.dll* file is copied into *C:\WINDOWS\system32\inet_eapar32.dll* right off the bat. We'll then eventually call the function we have labeled as *modifyCisv* to inject some malicious code into the *cisv.exe* executable and we'll finally start the *cisv* service by executing a command with a CLI by executing *cmd.exe*. This is all shown in the following screenshot where we have pointed out the crucial instructions. As before, we are using *cutter* on our local machine for the disassembly.

```
86: int main (int argc, char **argv, char **envp);
; var int32_t var_104h @ ebp-0x104
0x004012d0      push ebp
0x004012d1      mov ebp, esp
0x004012d3      sub esp, 0x104
0x004012d9      push 0
0x004012db      push str.C:__WINDOWS__System32__inet_epar32.dll ; 0x4091b8
0x004012e0      push str.Lab11_03.dll ; 0x4091a8 ; LPCSTR lpExistingFileName
0x004012e5      call dword [CopyFileA] ; 0x40801c ; BOOL CopyFileA(LPCSTR lpExistingFileName, LPCSTR lpNewFileName, BOOL bFailIfExists)
0x004012eb      push str.cisvc.exe ; 0x40919c
0x004012f0      push str.C:__WINDOWS__System32___s ; 0x409184 ; int32_t arg_ch
0x004012f5      lea eax, [var_104h]
0x004012fb      push eax            ; int32_t arg_8h
0x004012fc      call fcn.00401452
0x00401301      add esp, 0xc
0x00401304      lea ecx, [var_104h]
0x0040130a      push ecx            ; LPCSTR lpFileName
0x0040130b      call modifyCisv
0x00401310      add esp, 4
0x00401313      push str.net_start_cisvc ; 0x409174 ; int32_t arg_8h
0x00401318      call fcn.004013bc  ; runCommand
0x0040131d      add esp, 4
0x00401320      xor eax, eax
0x00401322      mov esp, ebp
0x00401324      pop ebp
0x00401325      ret
```

Copying Lab11-03.dll to C:\WINDOWS\system32\inet_epar32.dll by calling CopyFileA().

Calling a function similar to sprintf() to generate the "C:\WINDOWS\system32\cisvc.exe" string.

The first parameter (passed in reverse order) contains the %s placeholder!

The string generated with sprinf() is passed as a parameter to modifyCisv. Note we tell sprintf() to read onto the buffer pointed to by var_104h!

Calling runCommand to start the service once the executable has been patched!

We have broken down the *main()*'s functionality in the screenshot so we are ready to take a look at what *modifyCisv* does. Again, this function is quite complex so instead of the pure assembly code we'll be looking at the decompiled version. We already skimmed through it and changed a couple of variable names to try and make it as understandable as possible.

```
undefined4 __cdecl modifyCisv(undefined4 fnameToPatch)
{
    int32_t iVar1;
    undefined4 uVar2;
    int32_t iVar3;
    int32_t iVar4;
    int32_t iVar5;
    int32_t iVar6;
    int32_t iVar7;
    int32_t iVar8;
    undefined4 *puVar9;
    undefined4 *puVar10;
    uint32_t var_30h;
    undefined4 hObject;
    uint32_t var_28h;
    int32_t var_24h;
    uint32_t var_20h;
    uint32_t var_1ch;
    undefined4 var_18h;
    uint32_t var_14h;
    undefined4 var_10h;
    undefined4 hFileMappingObject;
    int32_t var_8h;
    undefined4 lpBaseAddress;

    iVar1 = (*_CreateFileA)(fnameToPatch, 0xc0000000, 1, 0, 4, 0x80, 0);
    if (iVar1 == -1) {
        uVar2 = 0xffffffff;
    } else {
        uVar2 = (*_GetFileSize)(iVar1, 0);
        iVar3 = (*_CreateFileMappingA)(iVar1, 0, 4, 0, uVar2, 0);
        if (iVar3 == -1) {
            (*_CloseHandle)(iVar1);
            uVar2 = 0xffffffff;
        } else {
            iVar4 = (*_MapViewOfFile)(iVar3, 6, 0, 0, uVar2);
            if (iVar4 == 0) {
                (*_CloseHandle)(iVar1);
                (*_CloseHandle)(iVar3);
                (*_UnmapViewOfFile)(0);
                uVar2 = 0xffffffff;
```

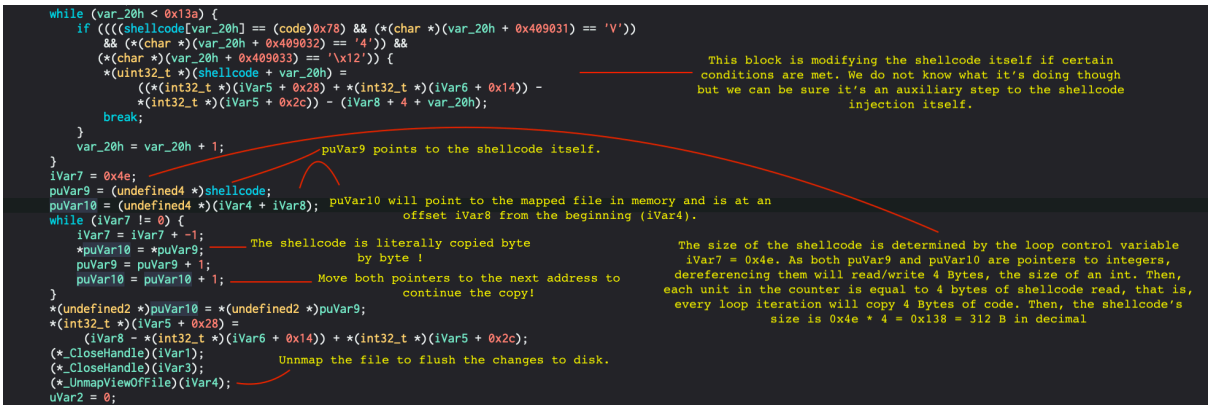
Get a handle/file descriptor to the file passed as a parameter (cisvc.exe)

Map the file into memory, note we pass the file handle as a parameter (iVar1)

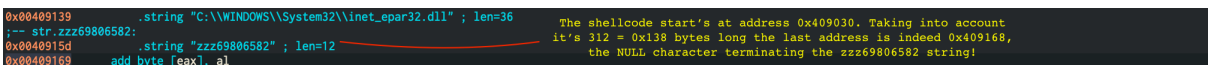
Get a reference to the file mapping in memory. If something goes wring unmap() the file and return, that is, bail.

After we get the file mapped into memory, we can modify the associated memory locations. Instead of then calling *WriteFile()* explicitly we move the changes to the file itself implicitly by *unmapping* it from memory with *UnmapViewOfFile()*. This mechanism is implemented thanks to the OS's virtual memory implementation where we'll project a file into memory and modify it as we would any normal variable. Upon unmapping the changes will be written back to disk. This approach is both more optimal as we don't have to wait for the HDD that often and "quieter" in the sense that we manage to modify a file without explicitly calling *WriteFile()*.

Assuming the file is mapped into memory correctly we'll then bomb a part of it with a *shellcode* that we want to be executed before the "real" executable itself. In order to do so we'll copy information from a buffer we have already initialized with the static *shellcode* into the mapped file in memory. We can see that process being carried out in the next image.



Again, the shellcode injection is thoroughly explained in the screenshot through annotations. We want to stress that the *shellcode* label was given by us. If we inspect the memory contents at that address, we'll find some instructions and a couple of strings at the end. As we know the shellcode's size is 312 bytes, we know they fall into the shellcode without any uncertainty.

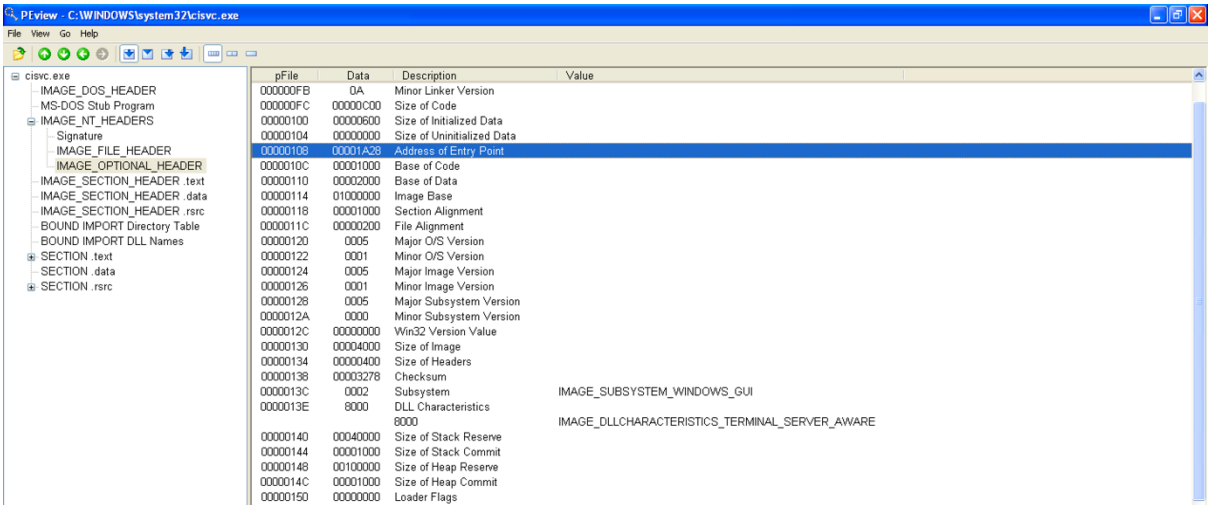


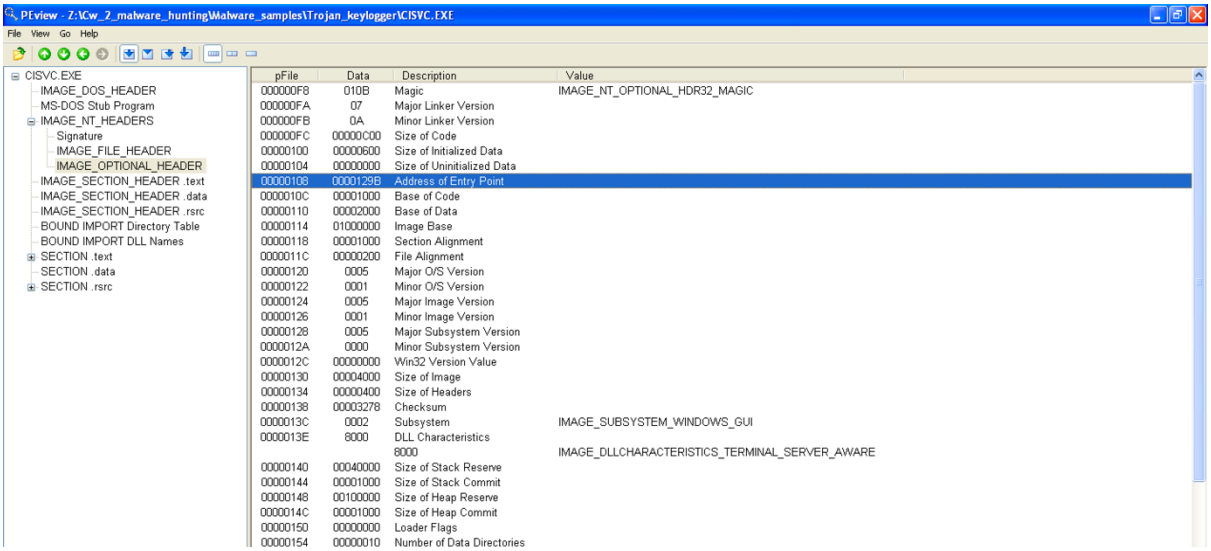
```
;-- shellcode:
0x00409030      push ebp
0x00409031      mov ebp, esp
0x00409033      sub esp, 0x40
0x00409039      jmp 0x409134
0x0040903e      push esi
0x0040903f      push edi
0x00409040      mov esi, dword [esp + 0xc]
```

Shellcode's beginning. We find instructions that have been crafted for a very particular purpose that are to run before the original executable which this code is injected into.

We find a couple of strings at the end that suggest that *inet_epar32.dll* is imported by the shellcode and that it somehow uses it's *zzz69806582* export, the one we already said looked suspicious. After comparing an injected and clean *cisvc.exe* executable we'll delve deeper into that function within the *Lab1-03.dll*.

We are including two screenshots showing the injected and original executables respectively. As the malware had already executed in our VM we had to look for the original executable on the Internet and found it here. The subtle difference between them besides the fact the injected one has the shell code embedded into it is that the *Address of the Entry Point* is also altered. As we want the shellcode to execute before the real executable, we have to mark it as the first instruction to be executed. This implies the shellcode must somehow return execution to the normal *cisvc.exe* so that it just doesn't crash out!



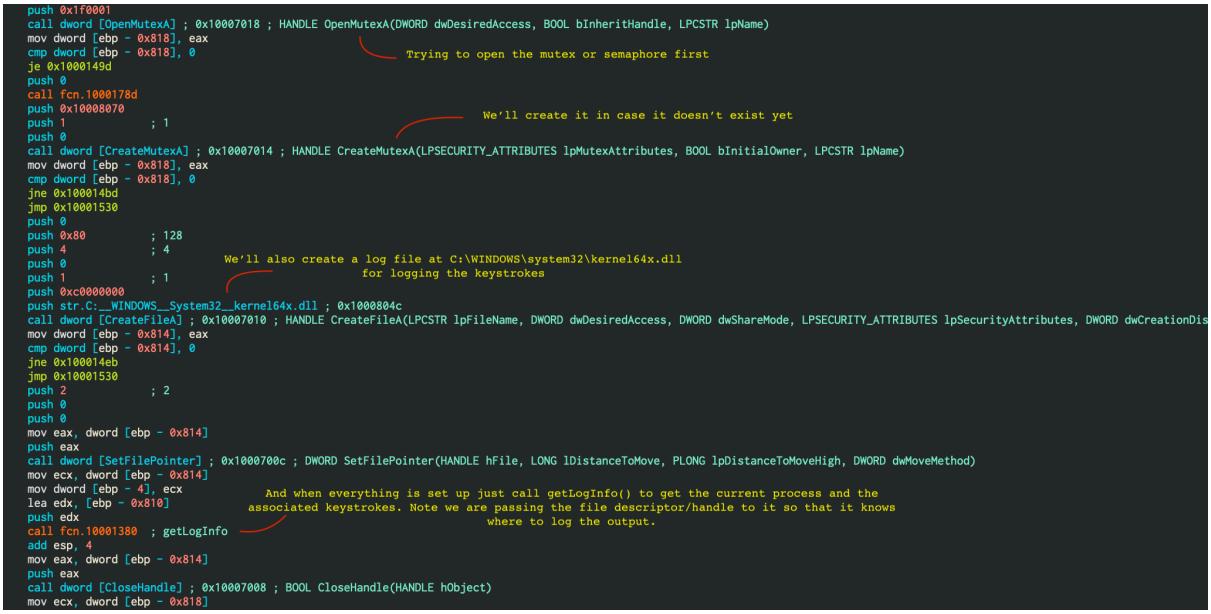


We should also point out that the shellcode is written *onto* the executable itself, it's not appended. That's why we'll see both files have the exact same size, 5632 bytes as seen with the *ls -l* command.

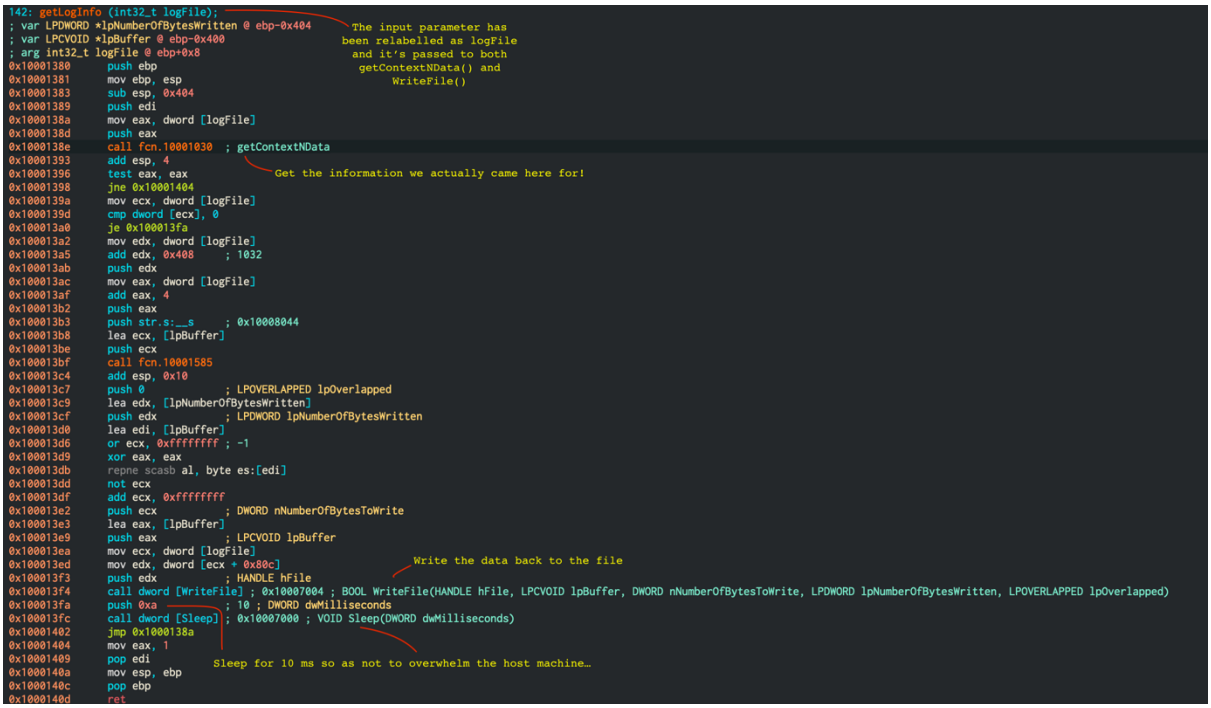
Knowing how things are going to develop we will now analyze the *zzz69806582* subroutine to learn how the keylogger is operating. We'll, as always, perform the disassembly through *cutter*.

As the function we suspect the modified *cisvc.exe* is exporting is *zzz69806582* we'll look for it in the source file. We'll soon find out that it's just creating a thread with a call to *CreateThread()* so we'll look for the thread's entry point which we have labelled *threadEntryAddress*. We are showing this first function in the screenshot below.

The above subroutine will first try to open a mutex with *OpenMutexA()*. In case it already exists the function will return an error as seen in the MSDN documentation; otherwise it'll just create said mutex with a call to *CreateMutexA()* so that the next execution of the process would just quit with an error should the original instance be still alive. With this mechanism we are in the end guaranteeing only one thread will be logging user input at any given time. After that we'll just create the *C:\WINDOWS\system32\kernel64x.dll* file to write the logs to. With everything set we'll then call the function we have labelled *getLogInfo()* that will get the current program's name as well as the user's input. The core of this method is included below.



If we take a closer look at *getLogInfo()* we'll just see it will pass the file handle it gets first of all to the function we have tagged as *getContextNData()* that's concerned with getting the current process window and its associated title so that we can provide more information to the gathered data as well as that data itself. After that it'll write the contents back to the passed log file and *Sleep()* for 10 milliseconds so that it doesn't exhaust system resources. It'll nonetheless run often enough so as to get the users input. Let's not forget we are way slower than computers…



Then we only need to find out what is going on in the *getContextNData()* function we are attaching below.

```
push eax
call section..text ; getWindowInfo
add esp, 8
cmp eax, 1          ; 1
je 0x10001127
mov ecx, dword [arg_8h]
cmp dword [ecx + 0x404], 0x400
jae 0x100010c7
mov edx, dword [arg_8h]
mov eax, dword [edx + 0x404]
mov dword [var_81ch], eax
jmp 0x100010d1
mov dword [var_81ch], 0x400 ; 1024
mov ecx, dword [var_81ch]
push ecx
lea edx, [var_400h]
push edx
mov eax, dword [arg_8h]
add eax, 4
push eax
call fcn.10001671
add esp, 0xc
test eax, eax
je 0x10001127
mov ecx, dword [arg_8h]
mov edx, dword [ecx + 0x404]
mov dword [var_820h], edx
mov eax, dword [var_820h]
push eax
lea ecx, [var_400h]
push ecx
mov edx, dword [arg_8h]
add edx, 4
push edx
call fcn.100015d7
add esp, 0xc
mov eax, dword [arg_8h]
mov dword [eax], 1
push 0x10           ; 16 ; int vKey
call dword [GetAsyncKeyState] ; 0x100070f8 ; SHORT GetAsyncKeyState(int vKey)
movsx ecx, ax
and ecx, 0x8000
```

We are also calling the function we have labelled as getWindowInfo() which will in turn call both GetForegroundWindow() and GetWindowTextA() so that we can find what is the program receiving the user input. We are excluding it from the report as it's not adding enough so as to justify another screenshot.

We'll then call GetAsyncKeyState() several times as we had already discussed which will in turn let us know the keys that have been pressed.

```
vKey = 0;
while( true ) {
    if (0xfe < vKey) goto code_r0x100012fb;
    uVar3 = (*_GetAsyncKeyState)(vKey);
    if ((((int32_t)(int16_t)uVar3 & 0x8000U) != 0) && ((uVar3 & 1) != 0)) break;
    vKey = vKey + 1;
}
```

Loop reading possible keys

We also added a small screenshot showing a while loop iterating over the possible keys that could be easily spotted thanks to the decompiling function bundled with *cutter*.

5.2 Removal process

In order to remove it we just need to delete the injected *cisvc.exe* executable and replace it with a clean one we can get form the link we already attached, an installation CD or by restoring the system itself. After that we only need to remove the malware itself and the associated executable. We might also want to remove the logging file itself even though it's not mandatory to halt the malware's operation.

5.3 Summing up

After the analysis we can then conclude that the malware contained in the *Lab11-03.exe* file will trojanize the Cisvc service by injecting some shellcode in it that will eventually load a DLL that it copied to the *C:\WINDOWS\system32* directory and that will record the keypresses to later write them to the *kernel64x.dll* file under the *system32* directory.

When compared to the previous sample we see how this keylogger uses a polling mechanism that makes it constantly ask for updates on the keyboard state. This is easily spotted by loops and manual program stops through functions like *Sleep()*. This is opposite to the event-based functionality we saw before where a function will be called upon keyboard change. This make the former approach much more resource intensive; that's why we need to manually yield resources by sleeping in the first place… Anyway, we can see how the same functionality can be achieved through different implementations.

6. Final thoughts

After having analyzed 4 different pieces of malware in such a thorough way we have become used to the different static and dynamic skills we can employ to face any potential threats we might stumble across in the future. We are also confident in our ability to identify key aspects of malware functionality in assembly as well as in our certainty about what signs to look for that could give malware away.

With these tools under the belt we are poised to have a better chance against the threats we might encounter in the world of cyber security.

7. Annex

We have decided to add a small annex where the defining characteristics for each piece of malware are noted in a succinct way:

7.1 Sample #1

- Location – *C:\WINDOWS\Hacker.com.cn.exe*
- File format – PE.
- Packed? – No.
- Obfuscated? – Partially.
- Network activity? – Yes.
- Malware type – Trojan.
- Removal steps – Delete services and executable.

7.2 Sample #2

- Location – *C:\WINDOWS\msagent\intl\MS_PMAL_Agent\BinaryCollection\Chpter_11L\Lab11-01.exe*
- File format – PE.
- Packed? – No.
- Obfuscated? – No.
- Network activity? – No.
- Main functionality – Keylogger.
- Removal steps – Remove registry entry.

7.3 Sample #3

- Location – *C:\WINDOWS\msagent\intl\MS_PMAL_Agent\BinaryCollection\Chpter_12L\Lab12-03.exe*
- File format – PE.
- Packed? – No.
- Obfuscated? – No.
- Network activity? – No.
- Main functionality – Keylogger.
- Removal steps – Kill processes and delete executables.

7.4 Sample #4

- Location – *C:\WINDOWS\msagent\intl\MS_PMAL_Agent\BinaryCollection\Chpter_11L\Lab11-03.exe*
- File format – PE.
- Packed? – No.
- Obfuscated? – No.
- Network activity? – No.
- Main functionality – Keylogger.
- Removal steps – Restore original file and delete sample.