1 Analysis Conducted

1.1 Static Analysis

1.1.1 010 Hex Editor

The samples were first analysed with the use of the 010 hex editor. In the hex editor, the "**MZ**" header was observed which indicates it is a PE file. The string "**This program cannot be run on DOS mode.**" was also observed at the PE header of the file (see Figure 1a). Analysis was conducted further to look for possible embedded PE files. However, no visible embedded duplicated PE header was discovered at the PE header of the file.

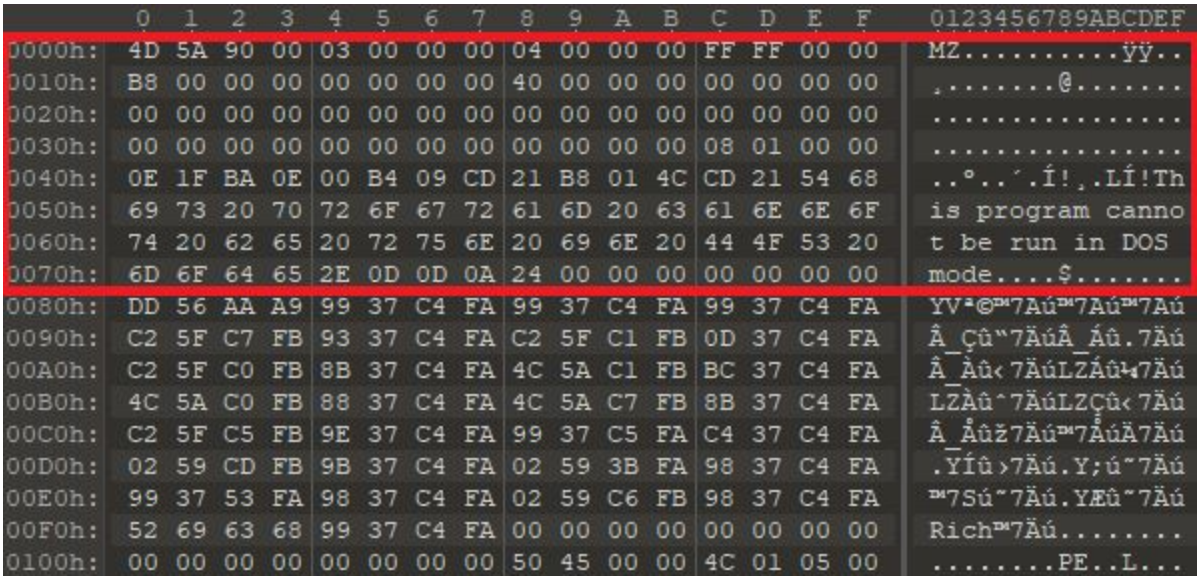

Figure 1a: PE header of the sample file viewed in hex editor

1.1.2 CFF Explorer

CFF Explorer was used to further analyse the files as they are all PE files. When analysing the PE section headers, no abnormalities were found. This suggests that the files given may not have been packed by a packer. We then proceeded to analyse the DLL characteristics. The flag "**DLL can move**" was initially checked before we unchecked it.

Next, we look at the Import Directories which shows the imported DLLs and functions used. **Kernel32.dll**, **User32.dll** and **Shell32.dll** are used which can be identified as host-based indicators. Lastly, under the Resource Directory, there is a resource named RSRC. However, we are unsure what it is or does at this static analysis stage.

1.1.3 Strings

The Strings command-line tool was used to analyze the strings in the PE files given. To output the result into a text file, the following command in the console was used:

```
strings 3F9C0FDA8529BA1CCEA96B4D330D9684 > 3F9C0FDA8529BA1CCEA96B4D330D9684.txt
```

Figure 1b: Piping output of strings to a text file

Once the result was written to a text file, further analysis was conducted. Firstly, the string "**This program cannot be run in DOS mode.**" was searched to see the number of matches. If there

are more than one, it would mean that there is another embedded PE file in the given PE file. However, there is only one matched.
Next, strings such as “**droppedTemp.jpg**”, “**part21.jpg**”, and “**part21.jpg - Windows Photo Viewer**” was observed. These strings might be the possible flags produced from the given files. Lastly, the imported functions available were also analysed from the result produced from Strings. Based on the functions, there are no network-based indicators and they are host-based indicators.

1.2 Reverse Engineering

In the reverse engineering process, IDA Educational disassembler was used. Firstly, the **_main** function was analysed. Based on the 4th and 5th assembly instruction, at least one argument is needed to prevent the program from exiting immediately (see Figure 1c). This indicates that a password is needed to execute the files.

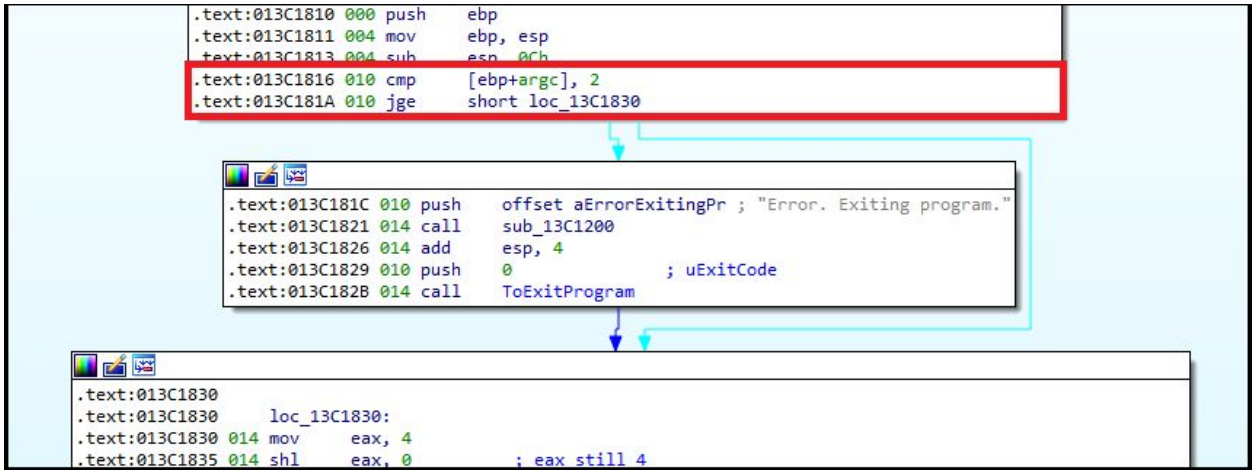

Figure 1c: Assembly instruction in main() showing at least one argument needed

Next, analysis was conducted on the first function called as the first command-line argument is passed in as an argument for the function. In that function, the argument is iterated through to do string comparison with the global variable password (see Figure 1d).

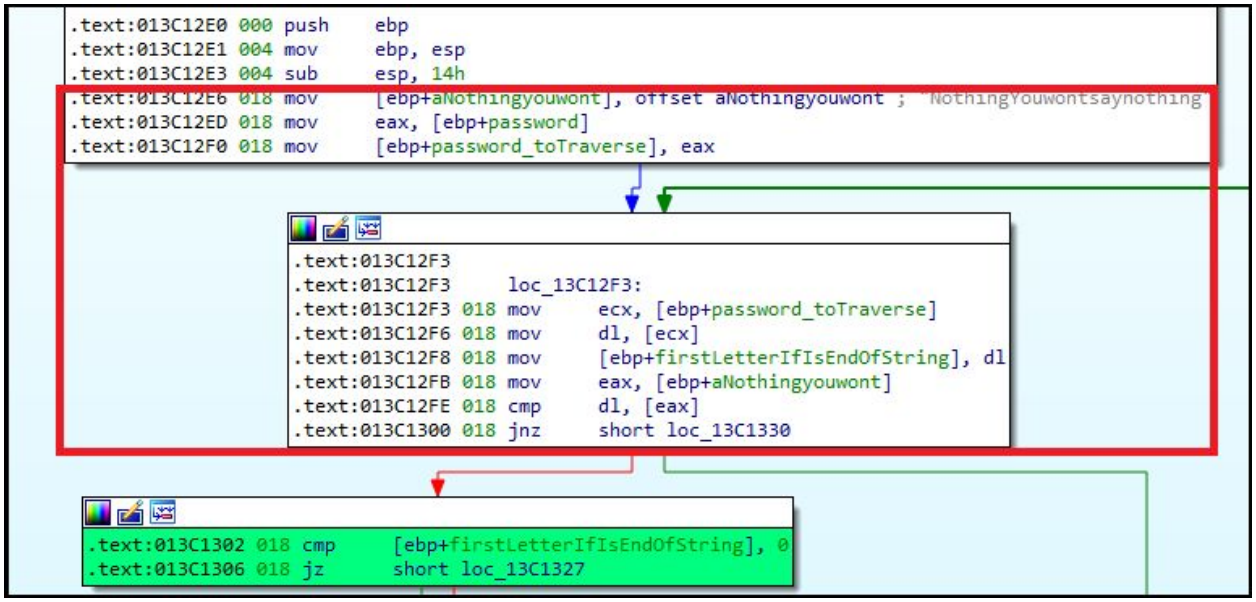

Figure 1d: Command-line argument compared with embedded password

1.3 Dynamic Analysis

After getting the password for the files, they are executed with the password parameter and dynamic analysis is done.

1.3.1 FakeNet-NG

FakeNet-NG was used to simulate network activities to detect if the 22 files contain any network capabilities. However, there is no abnormal traffic to any suspicious domains other than requests to Microsoft for Windows updates. This corresponds with the results from static analysis, where there is no network based Windows API used.

1.3.2 RegShot

RegShot was used to analyze if there are changes to the registry before and after running the files. Despite a number of changes being made to the registry, most of the changes are done by Microsoft Windows and some are by RegShot. Based on the imported functions used by the 22 given files, none of them used registry API. Therefore, we can conclude that no registry is directly accessed or modified by the 22 given files.

1.3.3 AutoRuns

AutoRuns was used to compare the difference before and after running the given files to see if any new files are added for persistence. However, no new files can be seen added into persistence. This shows that the 22 given files do not create persistence or generate another file and create persistence for it.

2 Approach to solve the Flag

After reverse engineering was conducted on all 22 files, the same trend was noticed. All the 22 files require a password and the passwords are all different. However, the structure of the files are the same. See Figure 2 for the list of renamed functions called from main() based on its purpose.

```
.text:013C1830 014 mov     eax, 4
.text:013C1835 014 shl     eax, 0          ; eax still 4
.text:013C1838 014 mov     ecx, [ebp+argv]
.text:013C183B 014 mov     edx, [ecx+eax]  ; get 1st arg cause each array inc 4 bytes
.text:013C183E 014 mov     [ebp+password], edx
.text:013C1841 014 mov     eax, [ebp+password]
.text:013C1844 014 push    eax
.text:013C1845 018 call    CheckForMalwarePassword
.text:013C184A 018 add     esp, 4          ; clear password that was pushed to stack
.text:013C184D 014 call    CreateDroppedTempJPG_ReturnPathOfIt
.text:013C1852 014 mov     [ebp+pathOf_dropTemp_jpg], eax ; C:\Users\hades\AppData\Local\Temp\droppedTemp.jpg
.text:013C1855 014 call    GetTempPathOf_part21_jpg
.text:013C185A 014 mov     [ebp+pathOf_part21_jpg], eax ; C:\Users\hades\AppData\Local\Temp\part21.jpg
.text:013C185D 014 mov     ecx, [ebp+password]
.text:013C1860 014 push    ecx
.text:013C1861 018 mov     edx, [ebp+pathOf_part21_jpg]
.text:013C1864 018 push    edx
.text:013C1865 01C mov     eax, [ebp+pathOf_dropTemp_jpg]
.text:013C1868 01C push    eax
.text:013C1869 020 call    GeneratePart21jpg
.text:013C186E 020 add     esp, 0Ch
.text:013C1871 014 mov     ecx, [ebp+pathOf_part21_jpg]
.text:013C1874 014 push    ecx
.text:013C1875 018 call    Open_Part21jpg_2Sec_Close
.text:013C187A 018 add     esp, 4
.text:013C187D 014 mov     edx, [ebp+pathOf_part21_jpg]
.text:013C1880 014 push    edx             ; pathOf_part21_jpg
.text:013C1881 018 mov     eax, [ebp+pathOf_dropTemp_jpg]
.text:013C1884 018 push    eax             ; pathOf_dropTemp_jpg
.text:013C1885 01C call    DeleteEveryFileCreated
.text:013C188A 01C add     esp, 8
.text:013C188D 014 push    1               ; uExitCode
.text:013C188F 018 call    ToExitProgram
.text:013C1894 018 mov     esp, ebp
.text:013C1896 004 pop     ebp
.text:013C1897 000 retn
```

Figure 2: Renamed functions called from main()

2.1 Further Reverse Engineering and Debugging

After running the file with the password as the parameter, we continued with reverse engineering and debugging to understand the functionalities of the executable file. The **CreateDroppedTempJpg_ReturnPathOfIt** function loads the resource RSRC and creates a **droppedTemp.jpg** file in the TEMP folder and returns its path. The **GetTempPathOf_part21_jpg** function then obtains the path to create the part jpeg file (e.g. part21.jpg) in the TEMP folder. However, the part jpeg file is not created yet. The part jpeg file was observed to be created after execution of the **GeneratePart21jpg** function which takes in the droppedTemp.jpg path, the password, and part jpeg paths as arguments. The **Open_Part21jpg_2Sec_Close** function then opens the part jpeg file in Windows Photo Viewer for 2 seconds before closing. Lastly, the **DeleteEveryFileCreated** function deletes all files that were created, including droppedTemp.jpg and the part jpeg file.

2.2 Automated Scripting

We coded a script that will automatically extract the flags for us. Firstly, the script will traverse the assembly code in the file and look for the password in IDA Educational tool. Next, it will run the debugger with the password as the argument and set breakpoints after all function calls in main(). When each breakpoint is triggered, droppedTemp.jpg and the part jpeg file are checked in the TEMP folder. If they are created, they will be moved to another storage folder and the debugger will be stopped. Lastly, paint was used to combine all the part jpeg files to form the picture flag as seen in section 4.

To run the script, Python 3 and IDA Educational/Pro from Hex Rays is required. Enter “python GetAllFlags.py <path of IDA>” into CMD to run or “python GetAllFlags.py -h” for usage examples. More instructions are included in README.txt.

3 Special Observation

droppedTemp.jpg: As droppedTemp.jpg cannot be opened by a photo viewer, we viewed its contents in a hex editor. We noticed that it does not contain a jpeg file header and the contents are the same as the RSRC resource found during static analysis. Furthermore, the file size of droppedTemp.jpg and the part jpeg are the same. However, the content of part jpeg file cannot be found in the 22 files given. Therefore, droppedTemp.jpg might be used to decode and form the part jpeg file.

Anti-bugging: In the 22 files given, the **IsDebuggerPresent()** function is found. However, breakpoints are set on the function before running the IDA Educational debugger. It is observed that IsDebuggerPresent() was not executed. Next, ScyllaHide plugin was also used in x32dbg to observe if there is any difference in the files produced. However, the content of the files remained the same.

4 Flag