Ardamax Keylogger Malware Analysis

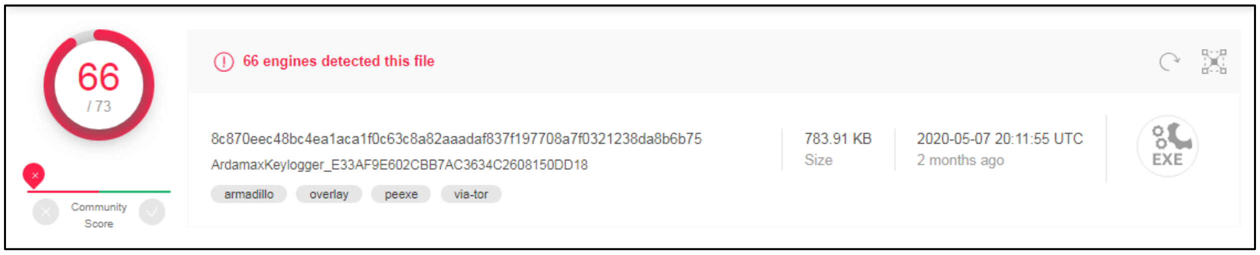
By Ethan Letourneau, Florida State University

Introduction and Setup Details

The program Ardamax is a commercial keylogger sold by Ardamax Software which began selling the program sometime in 2013 (the exact date is unknown and undocumented by the developer site). With the nature of the malware being commercial, it is easily known what the malware does – as advertised, it "captures user activity and saves it to a logfile", and some versions of it captures screenshots of the infected host at certain triggering events. The earlier versions of this keylogger are easily spotted by antivirus software, but more recent versions are able to evade detection.

The analysis environment will be a Windows 7 SP1 virtual machine (ran through Oracle VirtualBox) with no updates and Windows Defender disabled. The machine has 4 GB of virtual memory and a 32 GB dynamically allocated virtual HDD. Though this keylogger is not known to propagate over networks, I will be disabling the network adapter on this VM as a precaution and ensuring that there are no shared folders between the VM and my host machine. I will first perform static analysis to determine exactly how Ardamax performs its advertised function, and later perform dynamic analysis, running the binary to see what the effects are in files, Windows processes, etc. The SHA-256 value for the sample I will be analyzing is pictured at the beginning of the static analysis section.

Static Analysis of Dropper


66 engines detected this file

8c870eec48bc4ea1aca1f0c63c8a82aaadaf837f197708a7f0321238da8b6b75
ArdamaxKeylogger_E33AF9E602CBB7AC3634C2608150DD18

783.91 KB Size

2020-05-07 20:11:55 UTC 2 months ago

Figure 1: VirusTotal Results for Sample

The first step in analyzing the sample statically was taken in uploading the file to VirusTotal (see “Figure 1”) to find if it was a known malware. The answer was overwhelmingly “yes”, as all but a handful of antivirus program recognized it as being malicious.

Next was the analysis of the binary assembly itself. I loaded the file into Ghidra, which was able to correctly find the entry function – this indicated that the file was not packed and no unpacking would be necessary. Said entry function seemed to be pretty noisy, with a lot of setting of variables and running of preliminary functions (i.e., the call of __set_app_type(int), __p_fmode(), and so on). There are some non-DLL function calls happening here before the end of the entry, but they all do something rudimentary and then immediately return – for instance, one sets the control floating point number and returns, another seemingly just returns after doing nothing. At the end of the entry, there is finally a call to GetStartupInfoA and then GetModuleHandleA, and subsequentially to a function that does not return.

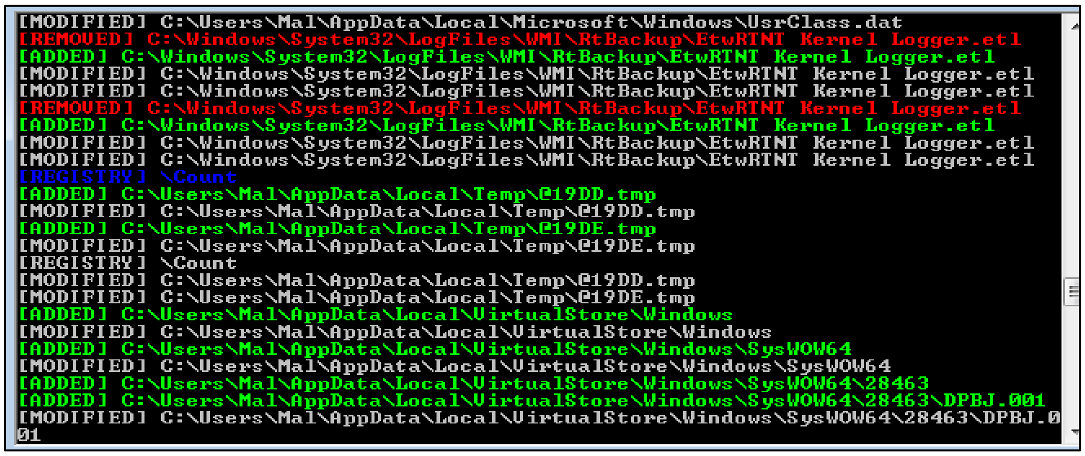
Going into this function that does not return (FUN_0040141e) it immediately calls another (FUN_00401230) with a pointer to empty data passed in (we’ll call it p1). In this function, almost immediately a new buffer object is put into p1, which is then passed as the second parameter into a call of GetTempPathW – this puts the path to the machines temporary folder into p1’s buffer. Then assigned to a local variable (we’ll call filename) is a call to GetModuleFileNameA with the buffer of p1 passed into it, which gives filename the fully qualified path to the file that contains the module of p1. Reading between the lines, this function continues on to call CreateFileA several times using variations on a single stack data point to generate the file names (we’ll see more about these in the dynamic analysis). We will name this “create_files_in_temp”. Upon returning to the original function (0040141e), if the return value of the create_files_in_temp function was 0, another function is called with the same parameter of the filename being passed in (if it failed, a call to GetMessageBoxW displays an error message). This one calls LoadLibraryW with the parameter+8 passed in as the “lpLibFileName” parameter. We can assume then that this file being dropped by the program is going to be either a DLL or another executable. LoadLibraryW’s call returns to a variable hModule, which will either be a handle to the library/executable or null. If it is not null, there is a call of GetProcAddress on the handle to get a function from the file called “sfx_main”, the address of which will be returned. I renamed this to “load_lib_get_sfx_main”.

This seemed to be all there was to the initial file, leading me to believe all it is designed to do was to drop the actual keylogger and additional files that are needed by it to run. Looking through the rest of the function and back to the entry, there was no evidence of indirect calling of another function.

Dynamic Analysis

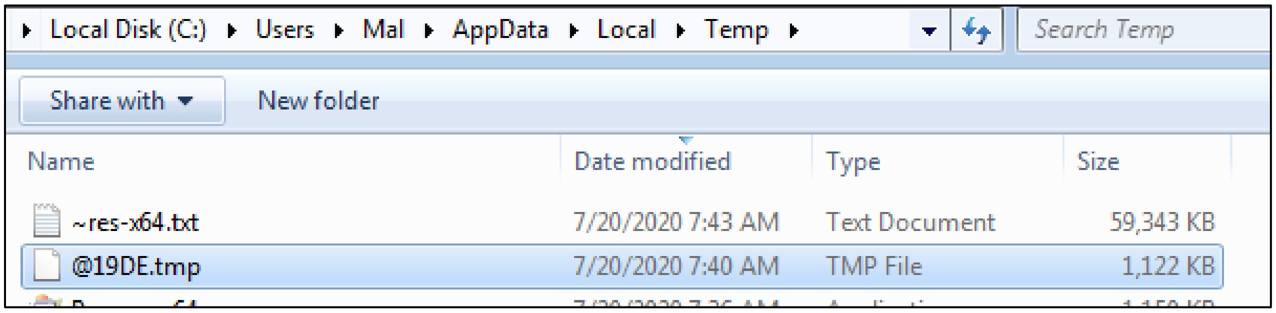
Since the dropper puts the malware of interest on the machine, I decided to go ahead with the dynamic analysis to trigger the placing of the actual keylogger on my virtual machine so I could analyze it statically afterwards. I used four tools in particular for this: Process Explorer, Process Monitor, RegFsNotify, and Regshot.I found that Process Explorer and Procmon crashed when attempting to run them, and found through Microsoft support that they no longer run on fresh installations of VM's and require Windows update KB2533623. After reverting to the clean snapshot of the VM and running Windows Update until it was up-to-date, the tools worked as expected.

I started up RegFsNotify (Figure 2) before running the program as well as Procmon, Process Explorer, and Regshot. As soon as I ran the malware, it created the file as expected in the Temp folder called @19DE.tmp (Figure 3), as well as dropping the four files in another directory created in the AppData/Local directory, being DPBJ.001, DPBJ.006, DPBJ.007, and key.bin seen in Figure 4.



```
[MODIFIED] C:\Users\Mal\AppData\Local\Microsoft\Windows\UsrClass.dat
[REMOVED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[ADDED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[MODIFIED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[MODIFIED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[REMOVED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[ADDED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[MODIFIED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[MODIFIED] C:\Windows\System32\LogFiles\WMI\RtBackup\EtwRTNT Kernel Logger.etl
[REGISTRY] \Count
[ADDED] C:\Users\Mal\AppData\Local\Temp\@19DD.tmp
[MODIFIED] C:\Users\Mal\AppData\Local\Temp\@19DD.tmp
[ADDED] C:\Users\Mal\AppData\Local\Temp\@19DE.tmp
[MODIFIED] C:\Users\Mal\AppData\Local\Temp\@19DE.tmp
[REGISTRY] \Count
[MODIFIED] C:\Users\Mal\AppData\Local\Temp\@19DD.tmp
[MODIFIED] C:\Users\Mal\AppData\Local\Temp\@19DE.tmp
[ADDED] C:\Users\Mal\AppData\Local\VirtualStore\Windows
[MODIFIED] C:\Users\Mal\AppData\Local\VirtualStore\Windows
[ADDED] C:\Users\Mal\AppData\Local\VirtualStore\Windows\SysWOW64
[MODIFIED] C:\Users\Mal\AppData\Local\VirtualStore\Windows\SysWOW64
[ADDED] C:\Users\Mal\AppData\Local\VirtualStore\Windows\SysWOW64\28463
[ADDED] C:\Users\Mal\AppData\Local\VirtualStore\Windows\SysWOW64\28463\DPBJ.001
[MODIFIED] C:\Users\Mal\AppData\Local\VirtualStore\Windows\SysWOW64\28463\DPBJ.0
01
```

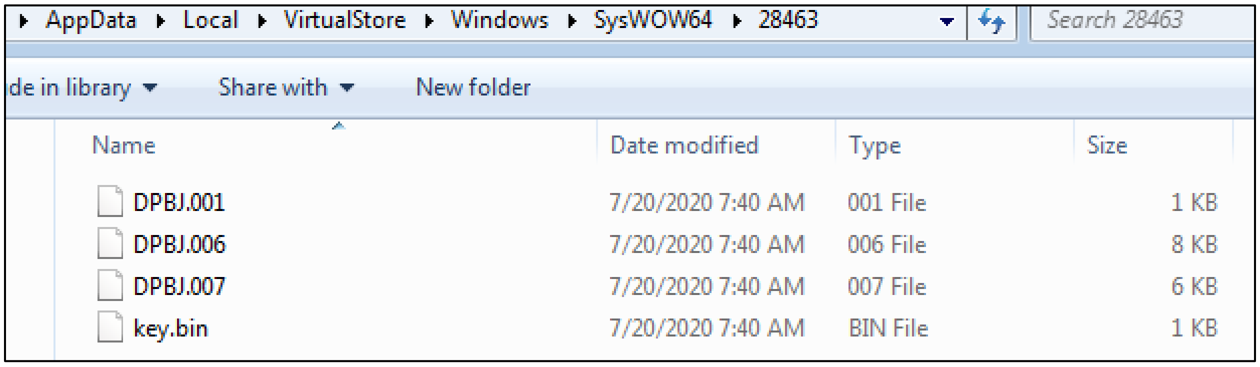
Figure 2: RegFsNotify output upon running dropper



Local Disk (C:) ▸ Users ▸ Mal ▸ AppData ▸ Local ▸ Temp ▸

Name	Date modified	Type	Size
~res-x64.txt	7/20/2020 7:43 AM	Text Document	59,343 KB
@19DE.tmp	7/20/2020 7:40 AM	TMP File	1,122 KB

Figure 3: Temp folder after running dropper



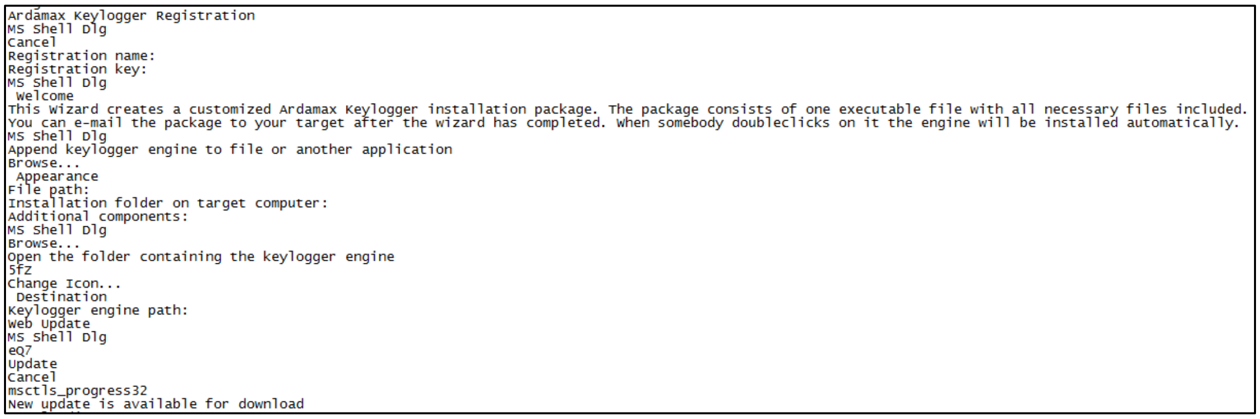
AppData ▸ Local ▸ VirtualStore ▸ Windows ▸ SysWOW64 ▸ 28463

Name	Date modified	Type	Size
DPBJ.001	7/20/2020 7:40 AM	001 File	1 KB
DPBJ.006	7/20/2020 7:40 AM	006 File	8 KB
DPBJ.007	7/20/2020 7:40 AM	007 File	6 KB
key.bin	7/20/2020 7:40 AM	BIN File	1 KB

Figure 4: Directory created by dropper

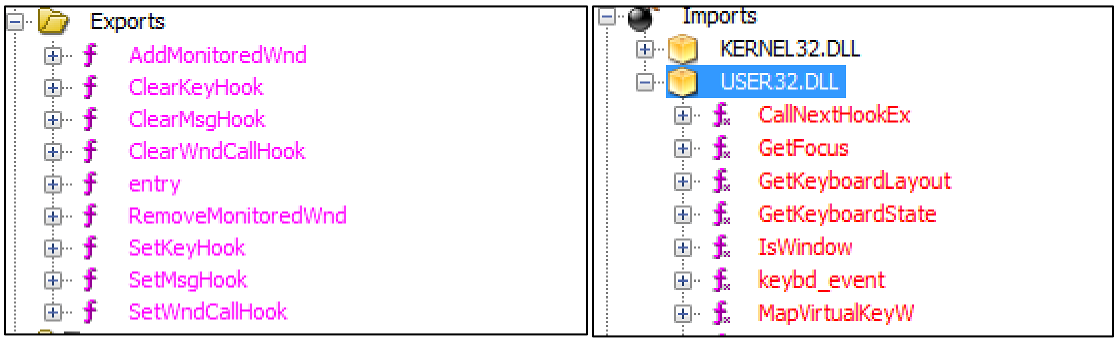
Continued Static Analysis

Using PEDetective, I found that the 006 and 007 files in the newly created directories were portable executables. I ran Strings on the temp file and found a large amount of text suggesting that it is a setup wizard for the keylogger (See Figure 5 below). I loaded the two files found to be PE type into Ghidra – the first, DPBJ.006, seemed to not have much of a discernable control flow, but did have an interesting set of imports and exports. The imports of this file were all from USER32.DLL and KERNEL32.DLL, the functions from the former all pointing to evidence of keylogging, such as GetKeyboardState and CallNextHookEx. Additionally there was the import of GetDesktopWindow, which is probably used by the malware to take screenshots of the infected host. The exports also gave this impression as all of them also had to do with setting hooks and the like, many using the imports from USER32.DLL (a sample of both the imports and exports shown in Figure 6). I believed this file would be used by another to perform the functions of the keylogger, but this was not the keylogging program itself.



```
Ardamax Keylogger Registration
MS Shell Dlg
Cancel
Registration name:
Registration key:
MS Shell Dlg
 Welcome
This wizard creates a customized Ardamax Keylogger installation package. The package consists of one executable file with all necessary files included.
You can e-mail the package to your target after the wizard has completed. When somebody doubleclicks on it the engine will be installed automatically.
MS Shell Dlg
Append keylogger engine to file or another application
Browse...
 Appearance
File path:
Installation folder on target computer:
Additional components:
MS Shell Dlg
Browse...
Open the folder containing the keylogger engine
5fZ
Change Icon...
 Destination
Keylogger engine path:
Web Update
MS Shell Dlg
eQ7
Update
Cancel
msctls_progress32
New update is available for download
```

Figure 5: Snippet from Strings output on @19DE.tmp



```
Exports
  AddMonitoredWnd
  ClearKeyHook
  ClearMsgHook
  ClearWndCallHook
  entry
  RemoveMonitoredWnd
  SetKeyHook
  SetMsgHook
  SetWndCallHook
```

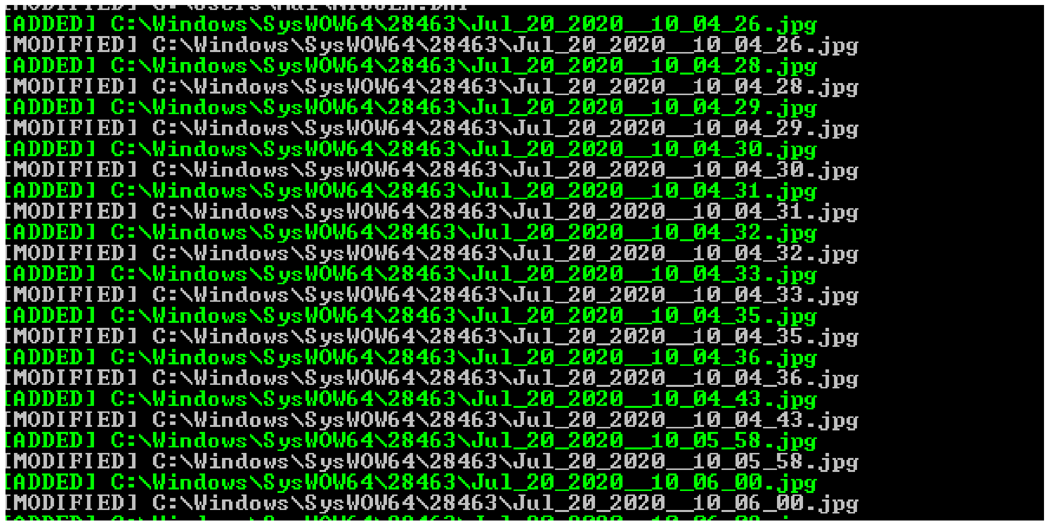
```
Imports
  KERNEL32.DLL
  USER32.DLL
    CallNextHookEx
    GetFocus
    GetKeyboardLayout
    GetKeyboardState
    IsWindow
    keybd_event
    MapVirtualKeyW
```

Figure 6: Imports and exports of interest from DPBJ.006

Looking to the second PE dropped, Ghidra also found it importing keylogging functions, as well as exporting two simply named functions, Hook and Unhook. Again, this looks as though it's to be used by a keylogging program, but is not the logger itself. With all this in mind, it seems as though running the dropper normally does not release the keylogger onto the machine.

<u>Dynamic Analysis Part 2</u>

The next step I took was to monitor my virtual machine while running the dropper as an Administrator to see if it would behave differently. I reverted to the clean Snapshot and re-downloaded the dropper once again and readied all the monitoring programs. Running the dropper in this fashion revealed the true nature of the dropper. The .tmp file was still created (under a different name), however this time the 28463 folder was put in the actual Windows directory of SysWOW64, and had more files in it: new DPBJ files ending in .002 and .009, as well as two executables: AKV.exe, and DPBJ.exe. From observations on Process Explorer, DPBJ.exe began running as soon as the dropper executed, and ate a considerable amount of memory, leading me to believe that this was the real keylogger. The DPBJ.002 file is of unknown importance, but likely more functions used by the logger. DPBJ.009 however, after letting the keylogger run for a minute, grew to the considerable size of ~200 MB – my first impression was that this is where the keystrokes were being stored.



```
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_26.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_26.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_28.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_28.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_29.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_29.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_30.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_30.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_31.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_31.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_32.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_32.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_33.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_33.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_35.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_35.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_36.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_36.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_43.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_04_43.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_05_58.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_05_58.jpg
[ADDED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_06_00.jpg
[MODIFIED] C:\Windows\SysWOW64\28463\Jul_20_2020__10_06_00.jpg
```

Figure 7: RegFsNotify sample of screenshots being saved

While DPBJ.exe was running, hundreds of JPEG's were added, modified, and deleted from the 28463 folder (Figure 7 above) that were named after the date and time. As expected, opening one of these files revealed that each was a screenshot (Figure 8 below). Oddly, all the screenshots were deleted at points, leading to the question of what it decides to keep and how. Killing the process tree in Process Explorer seemed to stop the keylogger (RegFsNotify was no longer being spammed with notices of JPEG's being added or modified), so I do not believe that the keylogger, if found and ended, attempts to restart itself immediately nor take precautions against its being killed. To see if perhaps the keylogger attempts to maintain consistency by adding itself to startup programs, I used the Autoruns tool, which revealed that this was indeed the case; as we can see in Figure 9, the DPBJAgent in the Run folder is associated with the dropped DPBJ.exe.

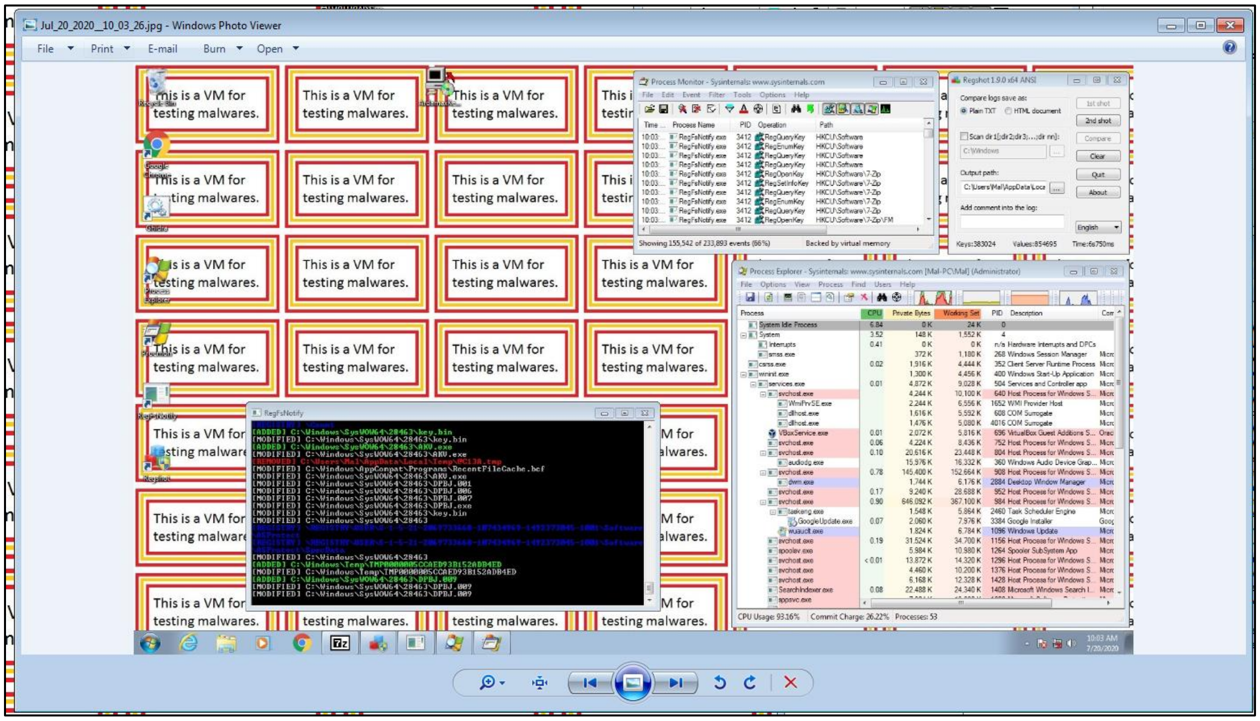
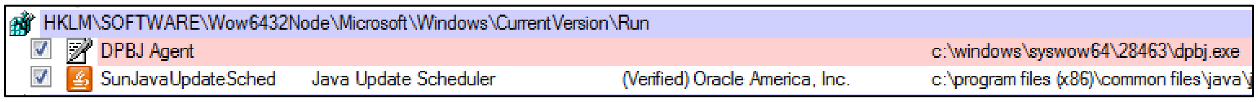

```
[REGISTRY] \Count
[ADDED] C:\Windows\SysWOW64\28463\key.bin
[MODIFIED] C:\Windows\SysWOW64\28463\key.bin
[ADDED] C:\Windows\SysWOW64\28463\AKV.exe
[MODIFIED] C:\Windows\SysWOW64\28463\AKV.exe
[REMOVED] C:\Users\Mal\AppData\Local\Temp\@C13A.tmp
[MODIFIED] C:\Windows\AppCompat\Programs\RecentFileCache.bcf
[MODIFIED] C:\Windows\SysWOW64\28463\AKV.exe
[MODIFIED] C:\Windows\SysWOW64\28463\DPBJ.001
[MODIFIED] C:\Windows\SysWOW64\28463\DPBJ.006
[MODIFIED] C:\Windows\SysWOW64\28463\DPBJ.007
[MODIFIED] C:\Windows\SysWOW64\28463\DPBJ.exe
[MODIFIED] C:\Windows\SysWOW64\28463\key.bin
[MODIFIED] C:\Windows\SysWOW64\28463
[REGISTRY] [illegible]
[REGISTRY] [illegible]
[MODIFIED] C:\Windows\SysWOW64\28463
[ADDED] C:\Windows\Temp\TMP000000005CCAED93B152ADB4ED
[MODIFIED] C:\Windows\Temp\TMP000000005CCAED93B152ADB4ED
[ADDED] C:\Windows\SysWOW64\28463\DPBJ.009
[MODIFIED] C:\Windows\SysWOW64\28463\DPBJ.009
[MODIFIED] C:\Windows\SysWOW64\28463\DPBJ.009
```

Figure 8: Screenshot taken by the keylogger

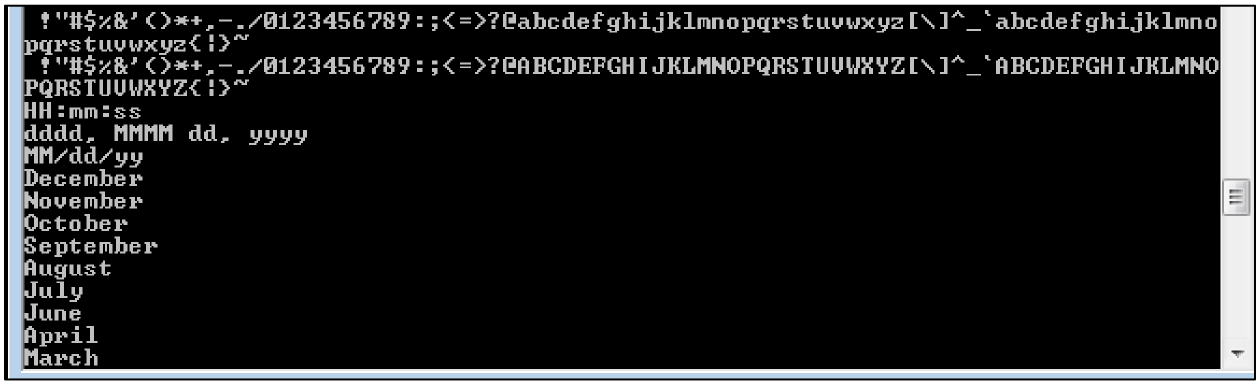


HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run			
☑ DPBJ Agent			c:\windows\syswow64\28463\dpbj.exe
☑ SunJavaUpdateSched	Java Update Scheduler	(Verified) Oracle America, Inc.	c:\program files (x86)\common files\java\j

Figure 9: Autoruns reveals DPBJ Agent

Final Static Analysis

I wanted to finally look at the two executables that were dropped and the very large DPBJ.009 file. I ran Strings on the latter to find that, if this was where the keystrokes were being logged, it was not being stored as plaintext, as Strings did not return any useful information. Perhaps evidence in the DPBJ.exe or AKV.exe disassembly would provide proof of my suspicions. I loaded in the AKV.exe into Ghidra and first inspected the imports. All the hook-related imports were present, as well as other giveaway functions such as MapVirtualKeyW (which is used to translate a key's value into a character) to drive home it's being used to log keys, if that wasn't clear enough. Additionally before looking into the disassembly I ran Strings on the executable to find what looked to be a character map of all possible normal key inputs, as well as strings delineating date information – these are likely used to store keystrokes and the dates they were done (Figure 10).



```
 !"#$%&'()*+,-./0123456789:;<=>?@abcdefghijklmnopqrstuvwxyz[\]^_`abcdefghijklmno
pqrstuvwxyz{|}~
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`ABCDEFGHIJKLMNO
PQRSTUVWXYZ{|}~
HH:mm:ss
dddd, MMMM dd, yyyy
MM/dd/yy
December
November
October
September
August
July
June
April
March
```

Figure 10: Strings output for AKV.exe

Instead of going straight into the entry function on this binary, I went to the calls of the keylogging import function calls to see the meat of the process. The first I looked as was WriteFile, hoping to confirm the suspicions that the data was being stored in the 009 file – the only call of this was in a single function (FUN_00415343) which simply writes the buffer passed in as the second parameter to a file passed in as the first parameter (it was renamed write_p2_to_p1). Write_p2_to_p1, however, was called around 20 times by another function over and over (FUN_004086f8) which starts out by calling another function that simply creates a file named whatever was passed in as the first parameter to the calling function and gives the handle to a local variable ("local_8"). Local_8 is then used by the repetitive calls to write_p2_to_p1. This function is called by a function that is called by a function that is called by the entry function.