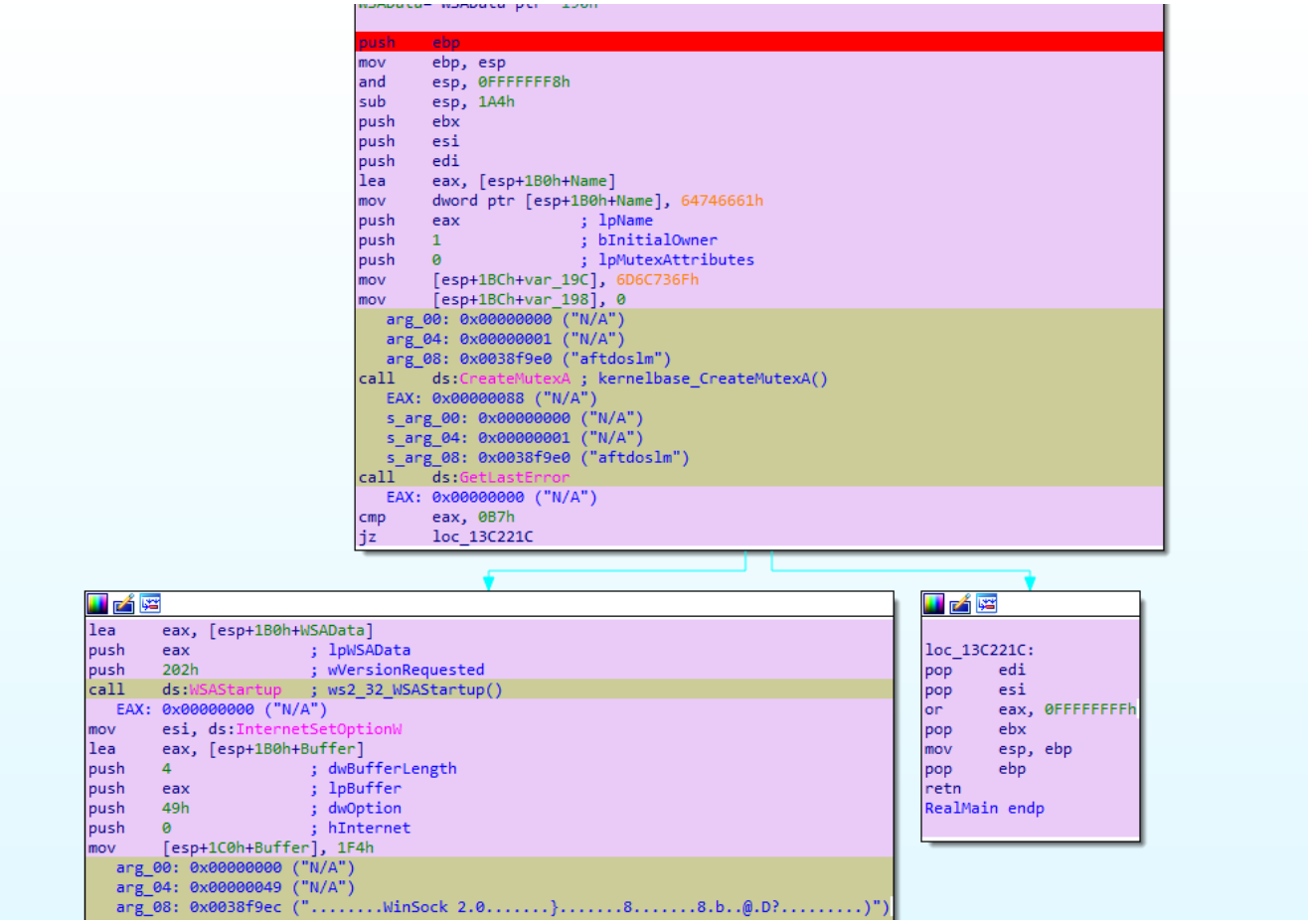
# TokyoX: DLL side-loading an unknown artifact (Part 2)

lab52.io/blog/tokyox-dll-side-loading-an-unknown-artifact-part-2/

As we mentioned in the previous post, we have performed an analysis of the threat which, lacking further information, we have not been able to identify it as a known threat. Thus, for the moment, we will keep referring to it as TokyoX.

This threat can only be found in memory, since it is encrypted on disk and its loading is performed as described in the previous post.

After the whole loading process, the first thing the threat does is to check for the existence of a mutex in the system with the name “aftdoslm” and if it finds it, it terminates its execution in order to avoid multiple instances of the threat running in parallel.

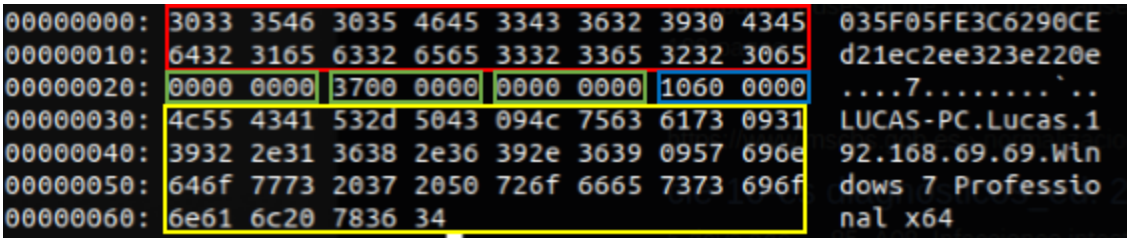



It then generates a 32-character string similar to a md5 hash from the computer’s file system and Volume information, which is used as the victim ID and placed at the beginning of every packet it sends to the command and control server:

```
GetVolumeInformationA(RootPathName, 0, 0, &VolumeSerialNumber, 0, 0, 0, 0);
v8 = 1286355 * VolumeSerialNumber - 1779332088;
v21 = 1286355 * VolumeSerialNumber - 1779332088;
v22 = 59062027 * v8 + 879546856;
v9 = 1286355 * v8 - 1779332088;
v10 = 1146777065 * v8 - 2010843488;
v23 = 1057242641 * v8 + 1661701696;
v24 = 848087043 * v8 + 2083528;
v25 = 2135145081 * v8 - 1682264352;
v26 = 274289083 * v8 - 1732530520;
v27 = 1569996065 * v8 - 1136426880;
v28 = 1356202547 * v8 + 2070181256;
v11 = -867472608 - 1953714167 * v8;
VolumeSerialNumber = -867472608 - 1953714167 * v8;
HeapAlloc_0(hHeap, 0, 255);
wsprintfA(
  hHeap,
  "%08lX%04lX%04lX%02lx%02lx%02lx%02lx%02lx%02lx%02lx%02lx",
  v21,
  (unsigned __int16)v9,
  (unsigned __int16)v10,
  (unsigned __int8)v22,
  (unsigned __int8)v23,
  (unsigned __int8)v24,
  (unsigned __int8)v25,
  (unsigned __int8)v26,
  (unsigned __int8)v27,
  (unsigned __int8)v28,
  (unsigned __int8)v11);
```

It then obtains the user name, computer name, local IP and operating system version, with which it builds a string and generates a first packet for the C2:

At the time of the analysis the command and control server was not available. For that reason, some fields of the packets a priori not used by the threat, could not be identified. However, from the analysis of the threat assembly, it has been possible to observe that the response of the C2 is very similar to those sent by the threat, since it most likely contains the first 44 bytes of the threat request or a very similar content: the victim ID (in red) , the first three dwords (in green). After that part, it will have a fourth dword (in blue) that consists of a command code supported by the threat, after which, depending on the command, it may contain one information or another (in yellow).




The command code in question goes in LittleEndian, so for the command 7010, the field should contain “10 70 00 00”. Inside the threat code, a switch can be found that identifies 6 different commands:

The first one is “6010”, which can be seen in the screenshot along with the contents of the network packet. This command is not among the options accepted by the threat, but it is the one that the threat itself uses when it sends the first packet to the command and control server, and we assume that it is the one used as a beacon when it does not intend to do anything, since a command that is not supported by the switch statement will simply cause a 5-minute sleep and a new request to C2 after this time.

Second comes the “6011” command which is relatively interesting since it first calls a function that will delete all the infection files and terminate the execution of the threat. The fact that makes it interesting to us is that after this call, inside the switch that controls each command, it appears again as a command to perform no action (in this case it should be the 6010 probably) and make another request to C2, although this code will never be executed because the initial function will close the process.

```
DeleteFileW(&FileName);
v26 = 0;
do
{
  v27 = Dst[v26];
  ++v26;
  *(__int16 *)((char *)&v30 + v26
}
while ( v27 );
RemoveDirectoryW(&FileName);
if ( sub_948F82() != 1 && !((*(_WO
{
  v1 = GetCurrentProcess();
  TerminateProcess(v1, uExitCode);
}
sub_9476DC(uExitCode);
ExitProcess(uExitCode);
```

The next command is “7010” which scans the disks installed in the computer using the “GetLogicalDriveStringsW” API and sends the results to the command and control server.

The command “7011” imitates the “Dir” command as it receives after the code a folder path, and allows it to list the content of a folder, showing hidden and system files too.

```
.           0       0       2009-7-14 4:20:8
..          0       0       2009-7-14 4:20:8
All Users       0       0       2009-7-14 6:8:56
Default 0       0       2009-7-14 4:20:8
Default User    0       0       2009-7-14 6:8:56
desktop.ini     1       174     2009-7-14 5:54:24
Lucas   0       0       2018-2-3 11:25:38
Public  0       0       2009-7-14 4:20:8
```

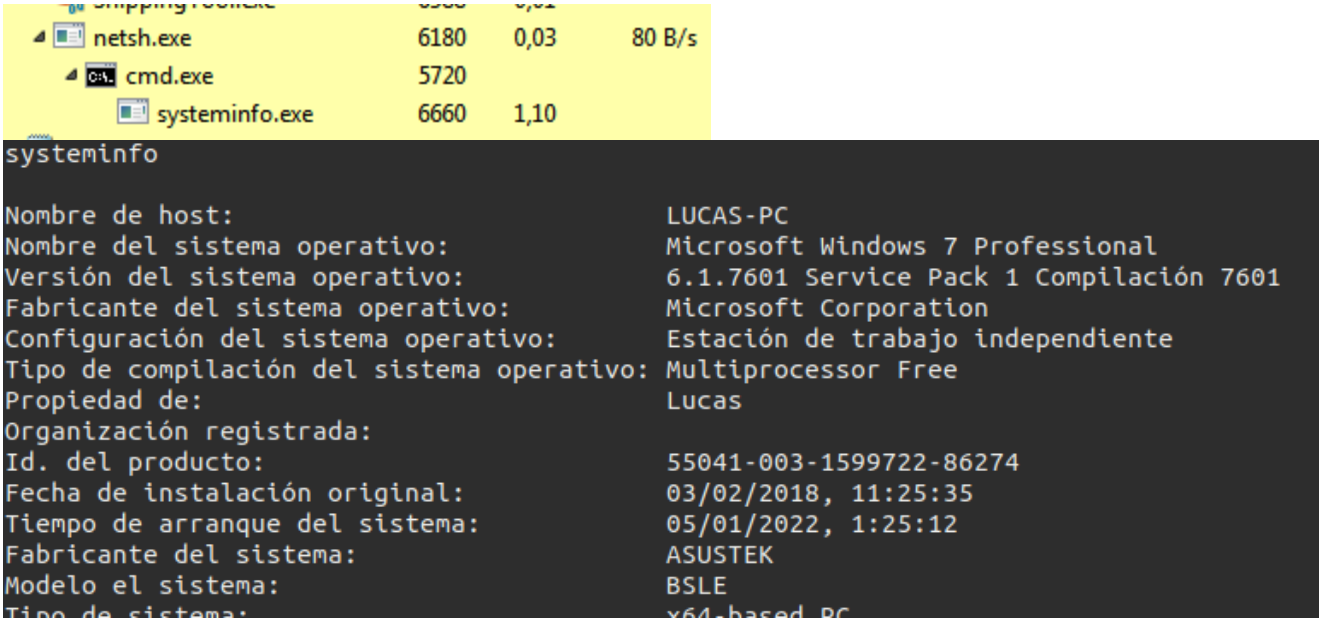
The next command “7017” expects, after the command code, a structure with three dwords followed by a path with a filename, and allows to upload a file from the attacking computer to the victim machine. In order to do so, it first creates an empty file and then generates a new thread that will make a second request to the command and control server. Within the response from the C2, this request expects the contents of the file to be uploaded, which the requesting thread will store in the file generated by the main thread.

```
MultiByteToWideChar(0xFDE9u, 0, v8, strlen(v8), v10, (int)lpMultiByteStrb);
sub_945730(v8);
v11 = CreateFileW(v10, 0x40000000u, 0, 0, 4u, 0x80u, 0);
sub_945730(v10);
if ( v11 == (HANDLE)-1 )
  return GetLastError();
v12 = GetFileSize(v11, 0);
if ( v12 == -1 )
  goto LABEL_9;
v14 = (char *)sub_945700(84);
*((_DWORD *)v14 + 1) = HIDWORD(v17);
*(_DWORD *)v14 = v12;
*((_DWORD *)v14 + 2) = v11;
*((_DWORD *)v14 + 3) = a4;
*((_DWORD *)v14 + 4) = v18[3];
v15 = (_OWORD *)v18[1];
*(_OWORD *)(v14 + 20) = *v15;
*(_OWORD *)(v14 + 36) = v15[1];
v16 = (_OWORD *)v18[2];
*(_OWORD *)(v14 + 52) = *v16;
*(_OWORD *)(v14 + 68) = v16[1];
if ( CreateThread(0, 0, sub_944870, v14, 0, 0) == (HANDLE)-1 )
```

Similarly, the command “7018” also expects after the command code a structure with three dwords, identical to the “7017” command, followed by a path with a filename. In this case, it generates a new thread that tries to read that file within the victim computer and makes a second request to the C2 containing the raw content, allowing the attacker to exfiltrate files from the infected computer.

With respect to the three dwords received before the file path, it has only been possible to identify the third one for the moment, which consists of the total size of the file to be sent or received and is used before obtaining the file to reserve a buffer of that size, plus one in the Heap of the application in order to recover the content there before dumping it to disk or reading it from disk.

Finally, there is the “8001” command with a command string to be run in a cmd.exe shell, since it calls CreateProcessW for “cmd.exe” with the input and output redirected by anonymous pipes to the executable itself. Then, it executes the command received in the request and replies to the C2 with the output of that command, thus forming a remote Shell of the victim machine.




Although **we do not have enough evidence to make any kind of attribution**, there are some characteristic TTPs in the infection chain: the attacker using a Dll-sideload with a legitimate binary and a library, the fact that the library is responsible for decrypting and executing a third encrypted file in memory, and the fact of removing the MZ characters from the beginning of the binary and replacing them with another string. This last characteristic is very common in APT groups that use PlugX as the main threat, where instead of using “tokyo” to replace MZ, they usually add “plug” at the beginning, which has motivated the name of the threat.

All of this explains the attribution of this hash to PlugX by some antivirus engines, although as it has been observed in the post, it is a somewhat smaller threat, which is not developed in Delphi as plugx and that keeps a greater resemblance to HttpBrowser in terms of capabilities. However, HttpBrowser does not share code or style of command codes with this sample, so it is most likely a different tool for the first infection stage.

| | |
|---|---|
| aftdoslma | Mutex |
| C:\Users\Public\Z46F4501-F44G-4DLS-AV8E-4E4DFDY918FG-DL5F102TVSFG7 | Path |
| 382b3d3bb1be4f14dbc1e82a34946a52795288867ed86c6c43e4f981729be4fc | Dll |
| 3493331e8f6151a37a48d11243e0fa32e756e8ca78a454912630865b48a43693 | TokyoX ciphered |

| | |
|---|---|
| 6370aa86e4bd2079913553bf34a5fe983b54d4907d168b278d6fe3caaf278d13 | Zip File |
| 31.192.107[.]187:443 | C2 |

Customers with Lab52's APT intelligence private feed service already have more tools and means of detection for this campaign.
In case of having threat hunting service or being client of S2Grupo CERT, this intelligence has already been applied.

If you need more information about Lab52's private APT intelligence feed service, you can contact us through the following link