New DarkGate Variant Uses a New Loading Approach

netskope.com/jp/blog/new-darkgate-variant-uses-a-new-loading-approach

2023年11月1日

Nov 01 2023

Summary

In the past month, the Netskope Threat Labs team observed a considerable increase of SharePoint usage to deliver malware caused by an attack campaign abusing Microsoft Teams and SharePoint to deliver a malware named DarkGate.

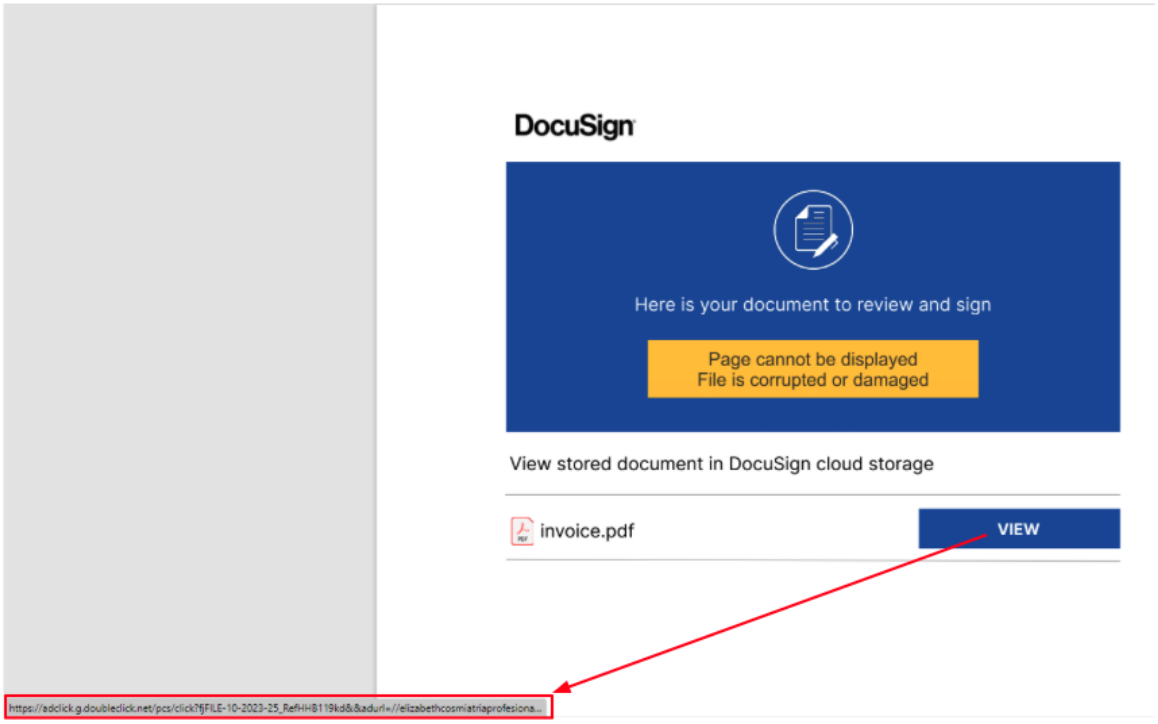
DarkGate (also known as MehCrypter) is a malware that was first reported by enSilo (now Fortinet) in 2018 and has been used in multiple campaigns in the past months. Since its recent update announcement in an underground forum, several campaigns have been conducted to deliver the malware using different methods, such as phishing and SEO poisoning.

DarkGate appeals to many attackers because of its broad feature set, which includes HVNC, keylogging, information stealing, and downloading and executing other payloads. DarkGate can be used as a starting point for bigger attacks, including Ransomware infections.

Netskope Threat Labs recently identified a new DarkGate variant delivered via MSI using a loading approach based on Cobalt Strike Beacon's default shellcode stub. Correlating the analyzed samples with findings from other researchers, we could determine that this is part of a new version of the DarkGate malware. Let's take a closer look:

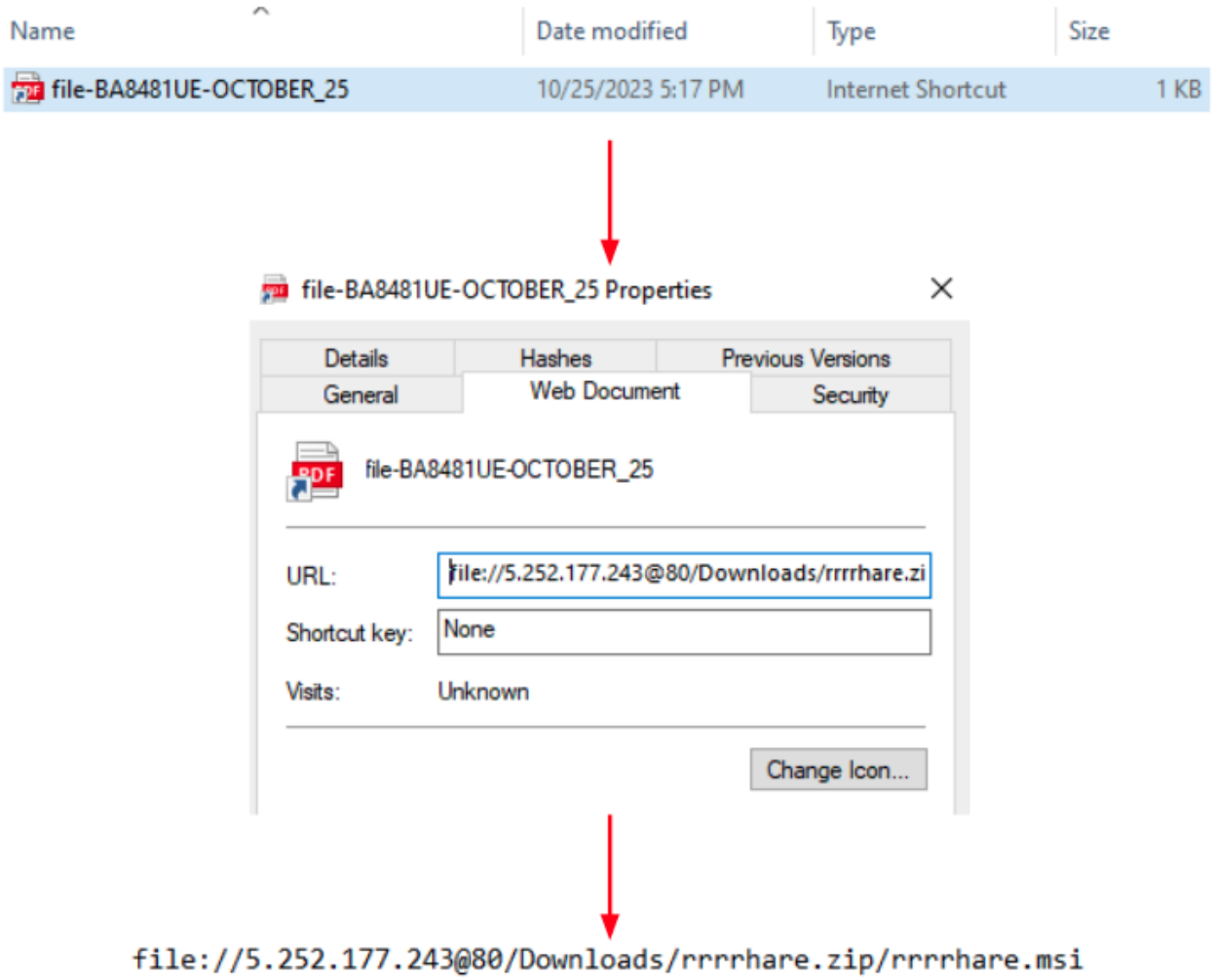
Infection analysis

The infection starts via a fake invoice email delivering a PDF document to the victim. The PDF file contains a DocuSign template that is used as an attempt to lure the user to open a document to be reviewed:



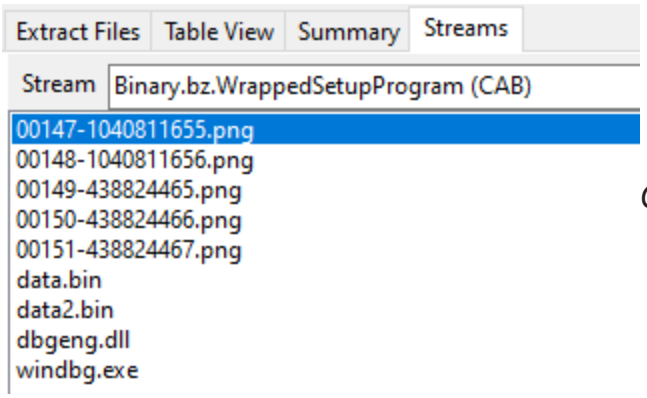
Example of the malicious document sent to the victim

Once the user clicks on the fake document a CAB file is downloaded. The CAB file contains an internet shortcut that once executed downloads an MSI file to the infected machine:



Internet shortcut leading to the MSI download

Once the user executes the MSI file a whole chain of loading mechanisms starts using the files presented in another CAB file inside the MSI:

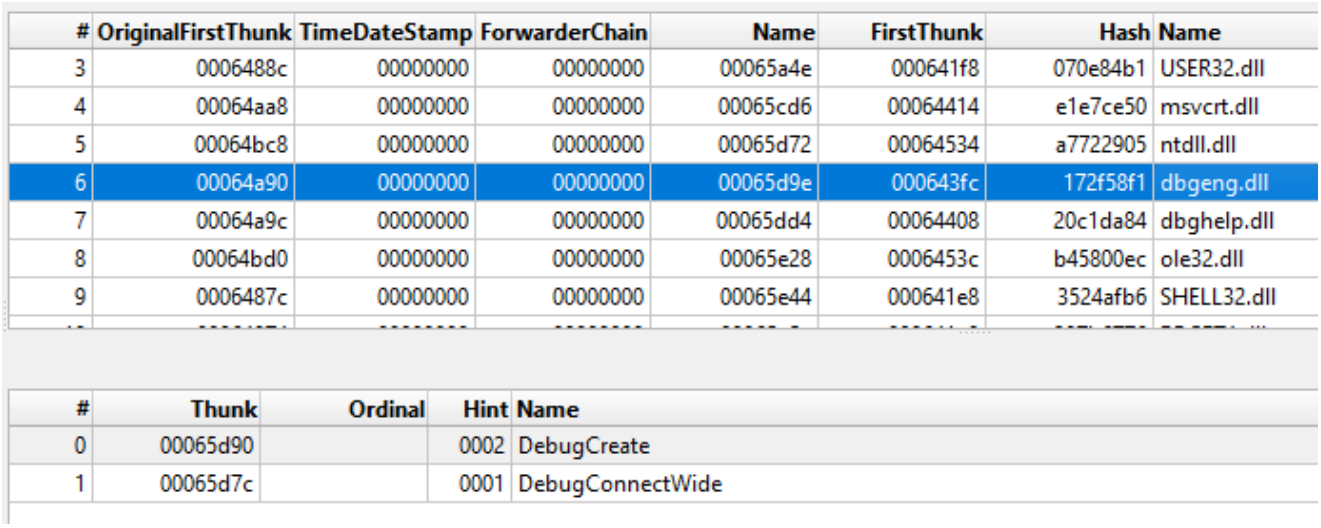


CAB file content

Stage 1 – DLL Side-Loading

The chain starts via the execution of the windbg.exe binary present in the CAB file. The DLL side-loading technique is used here in order to execute a fake version of the dbgeng.dll DLL file. Since windbg.exe imports functions from dbgeng.dll, this DLL will be included in its

import table, causing the Windows loader to map the DLL into windbg.exe's address space and then execute the DllMain function:



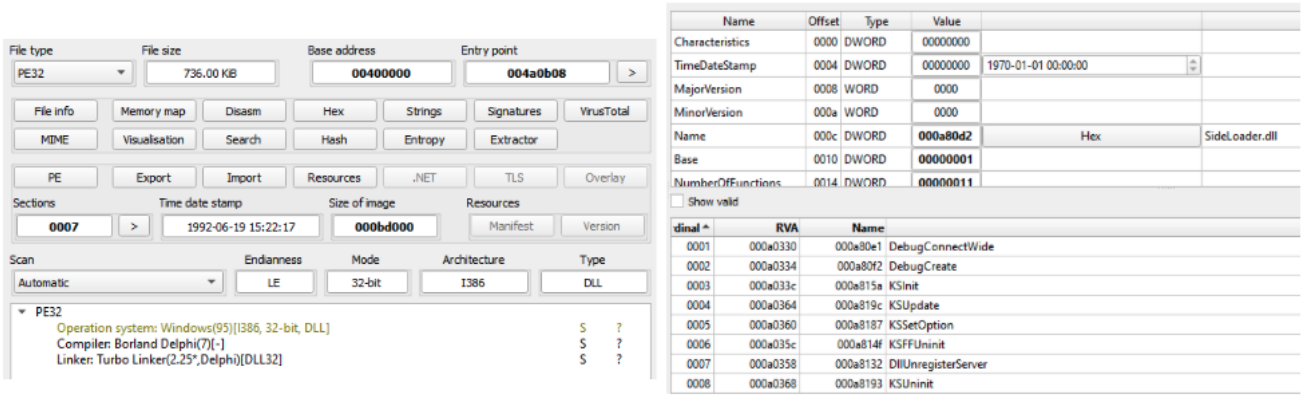
#	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk	Hash	Name
3	0006488c	00000000	00000000	00065a4e	000641f8	070e84b1	USER32.dll
4	00064aa8	00000000	00000000	00065cd6	00064414	e1e7ce50	msvcrt.dll
5	00064bc8	00000000	00000000	00065d72	00064534	a7722905	ntdll.dll
6	00064a90	00000000	00000000	00065d9e	000643fc	172f58f1	dbgeng.dll
7	00064a9c	00000000	00000000	00065dd4	00064408	20c1da84	dbghelp.dll
8	00064bd0	00000000	00000000	00065e28	0006453c	b45800ec	ole32.dll
9	0006487c	00000000	00000000	00065e44	000641e8	3524afb6	SHELL32.dll

#	Thunk	Ordinal	Hint	Name
0	00065d90		0002	DebugCreate
1	00065d7c		0001	DebugConnectWide

View of Windbg.exe dependencies in the Import Table using DIE

The dbgeng.dll is written in the Delphi programming language and has the internal name of SideLoader.dll, a common name observed in several DarkGate DLLs. It also contains export functions required for different binaries, such as windbg.exe and KeyScramblerLogon.exe, which was also observed being abused to side-load malicious DLLs.

In the KeyScramblerLogon.exe case, the side-loaded DLL is named KeyScramblerIE.dll and that is also written in Delphi. The loading methods and decoding algorithm are slightly different from the version presented in this blog, which abuses the WinDbg binary.

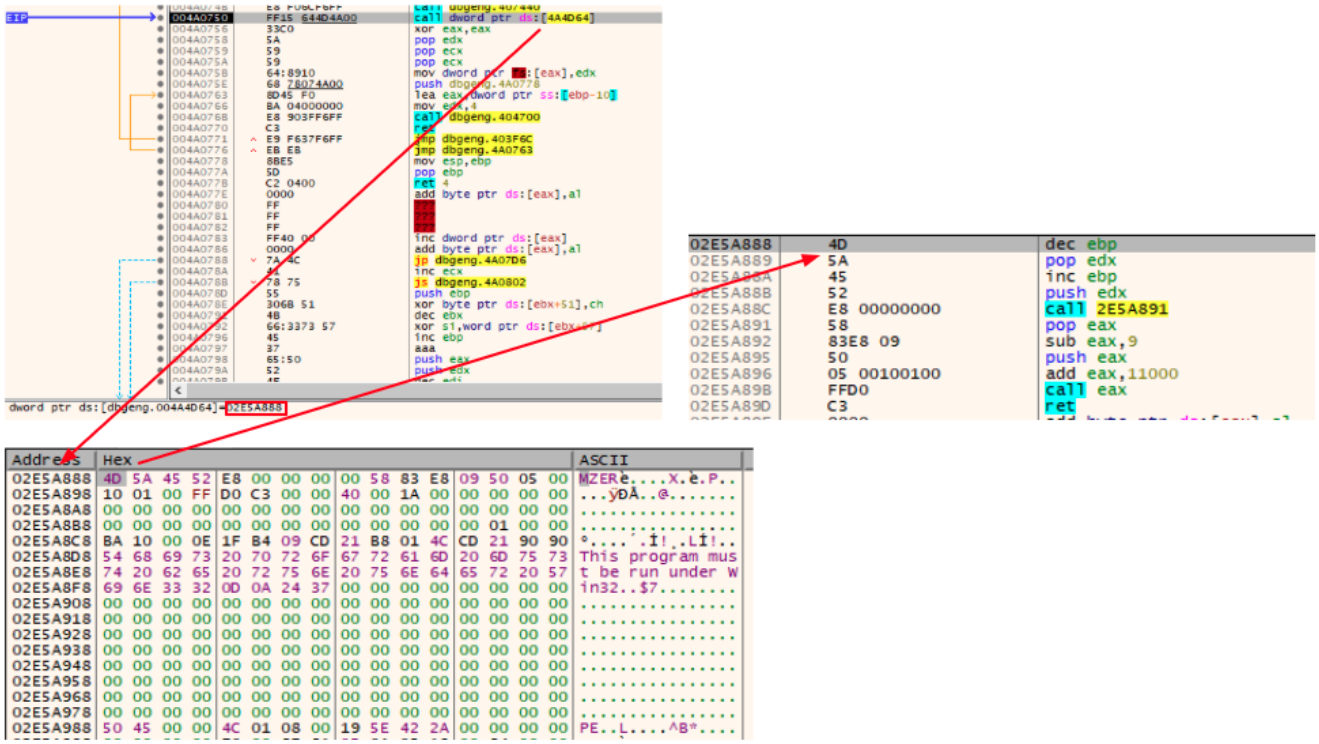


General overview and Export Table view from the fake dbgeng.dll

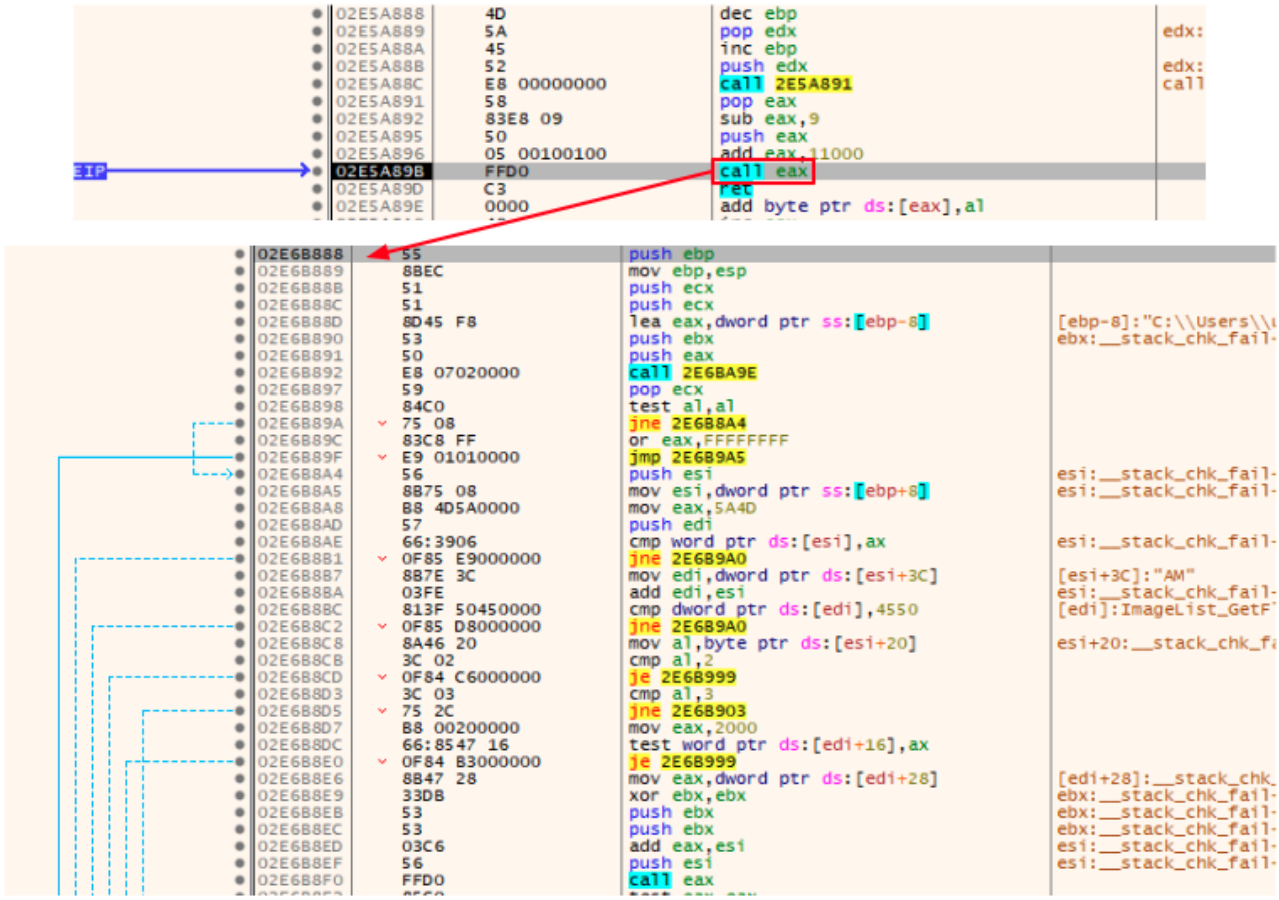
Upon execution of its DllMain function dbgeng.dll reads the content of a file named data.bin, present in the same directory, and decodes it using a custom base64 approach using the "zLAxuU0kQKf3sWE7ePRO2imyg9GSpVoYC6rhlX48ZHnvjJDBNFtMd1I5acwbqT+=" alphabet. This approach is the same used in other variants of DarkGate.

The decoded content results in a PE file (also written in Delphi) with a shellcode at the end of the file. The execution flow will then be redirected to the base address (first byte of the DOS header) of the decoded file.

The DOS Header bytes of this file contains a tiny snippet that is responsible for calculating the base address of the current decoded file, adding the RVA of the decoded shellcode to the base address and then calling it via a “call eax” instruction:



Example of the execution being redirected to the decoded file DOS Header using x64dbg



Call to the decoded shellcode entry

The technique employed here is very similar to the Cobalt Strike Beacon's default shellcode stub, which is usually employed to call the Beacon's ReflectiveLoader export function.

The called shellcode then prepares the file to be executed performing actions such as resolving its Import Address Table. The LoadLibraryA and GetProcAddress Windows API functions are resolved by hash using the CRC32 algorithm and then used to resolve the IAT.

The execution flow is then transferred to the stage 2 entry point:

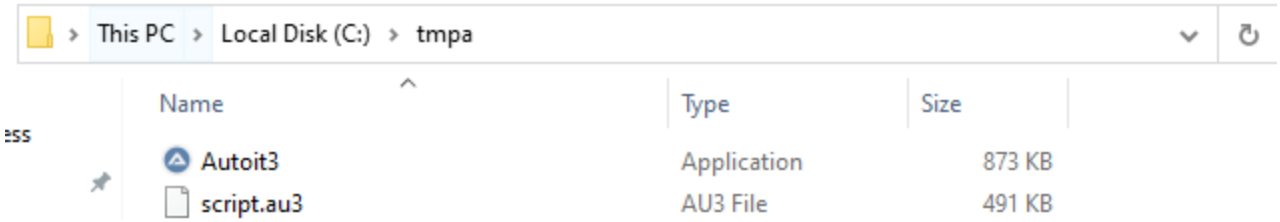
Stage 2 file entrypoint

Stage 2 – Another Delphi loader

The actions performed by this stage is very similar to the first one. The difference here is that the file read and decoded is the data2.bin file. Also, instead of being decoded all at once the malware first tries to find the occurrence of the “splitres” string in the file and then splits it in two parts. After the malware obtains the two parts it decodes both using the same custom base64 approach.

The first decoded part results in the AutoIt.exe binary and the second part is an AutoIt script that will be named script.au3. The use of AutoIt files is a well-known approach used by DarkGate actors.

A directory named “tmpa” is created under “C:\”, both files are written to it, and then the CreateProcessA function is called to execute the AutoIt script using AutoIt.exe:



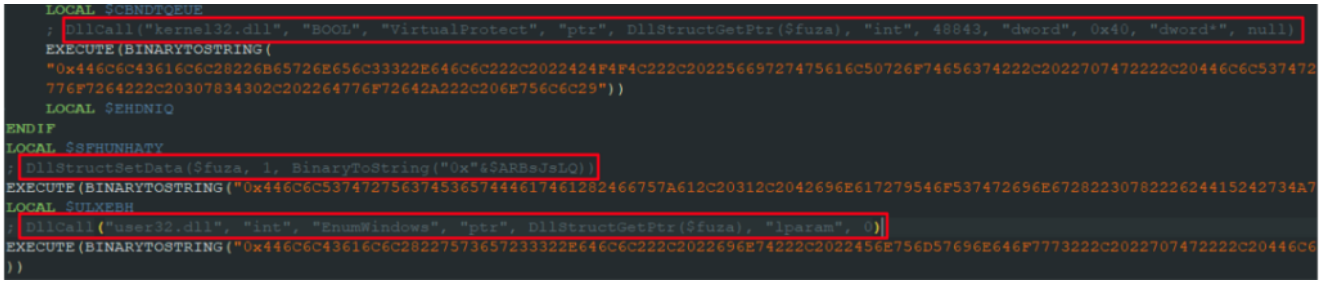
C:\tmpa directory content

AutoIt.exe being used to run the script.au3 script

Stage 3 – The AutoIt script

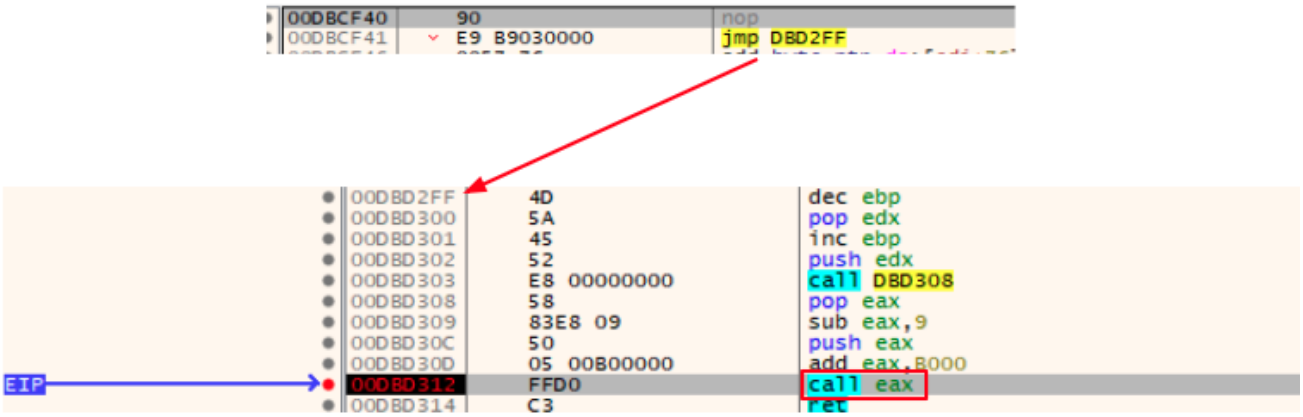
The executed AutoIt script is responsible for constructing a PE file and executing it via the same DOS header approach. The DOS header shellcode is executed by using a callback function passed to the EnumWindows API function.

Once we decode the AutoIt script, we can see the commands responsible for the loading process are encoded in hexadecimal. The decoded commands were added as comments in the screenshot below to demonstrate the mentioned actions:



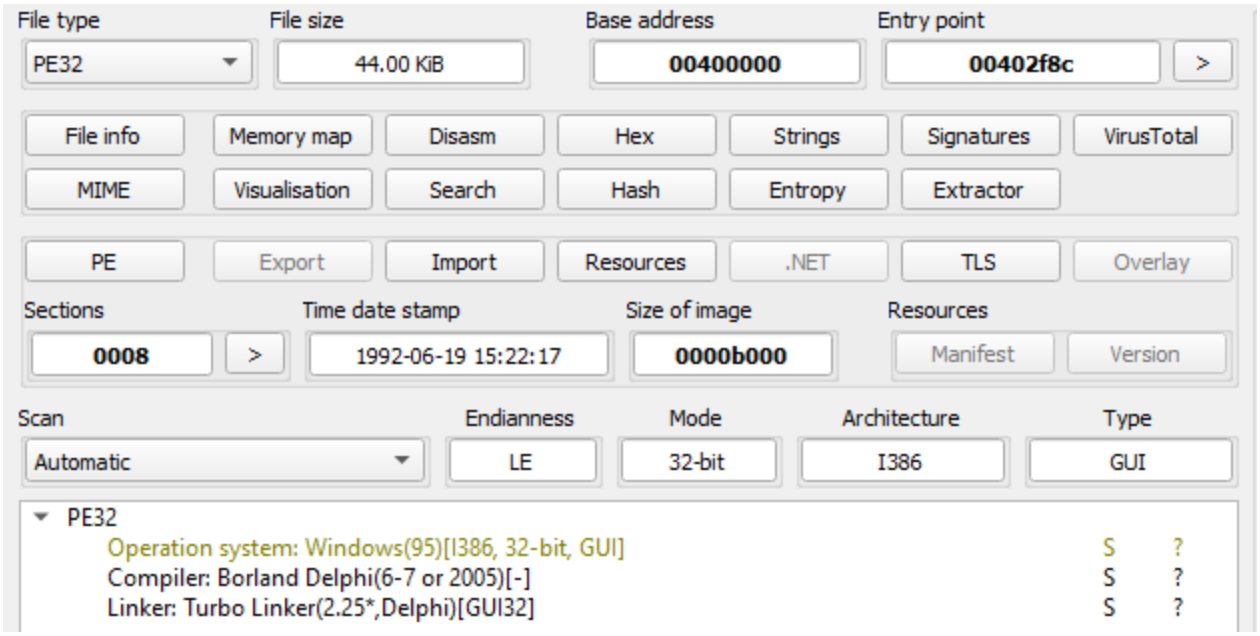
AutoIt script content with the important commands commented

Once the callback function is called, the same loading process occurs and the loader shellcode transfers the execution to another Delphi binary. The small difference in this case is that instead of going directly to the DOS header snippet, the callback function first goes to a kind of gate that would jump to the DOS header:

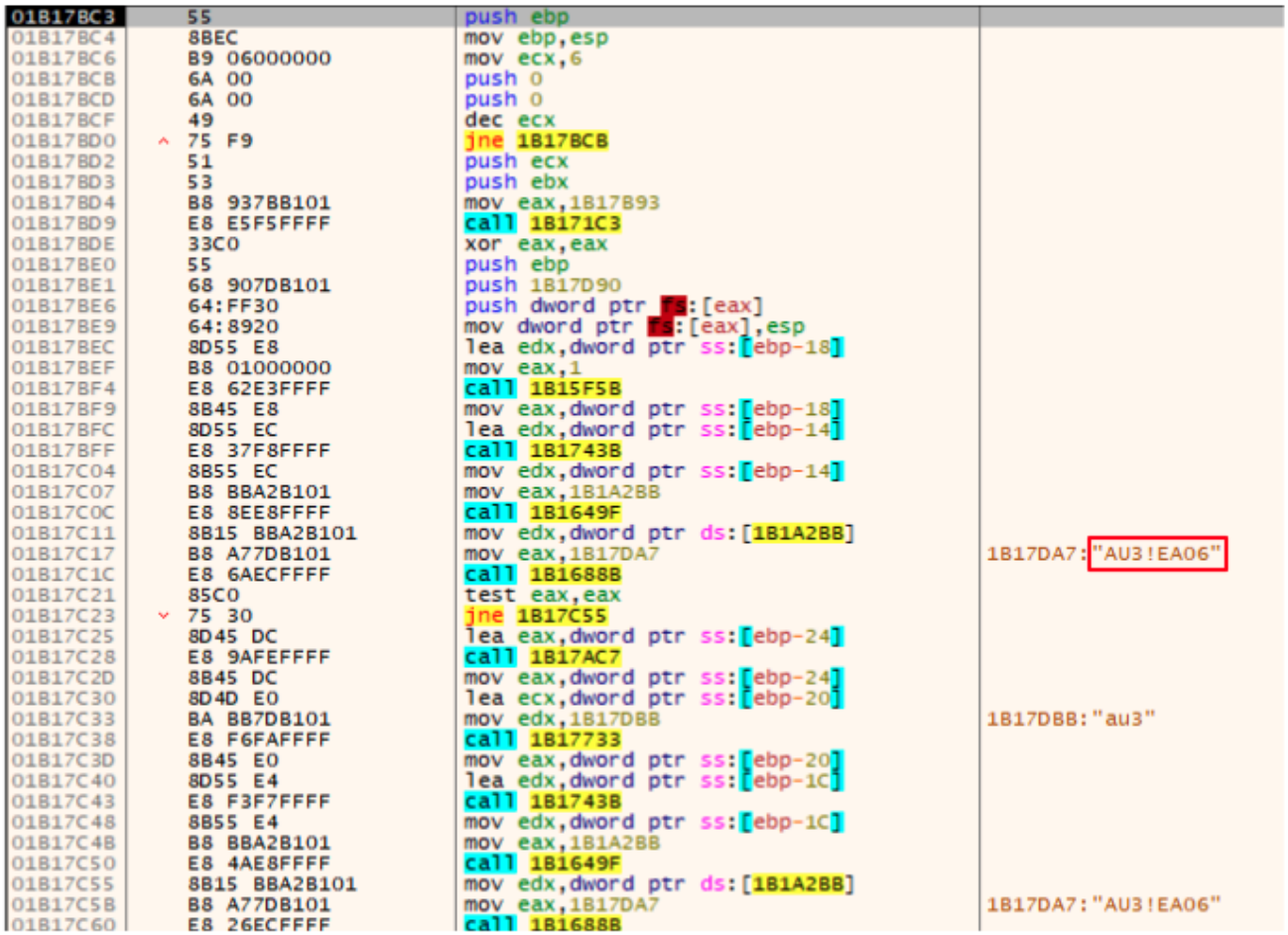


DOS Header snippet transferring the execution to the loader shellcode

Stage 4 – Again a Delphi loader



Decoded Delphi file overview



Stage 4 entrypoint

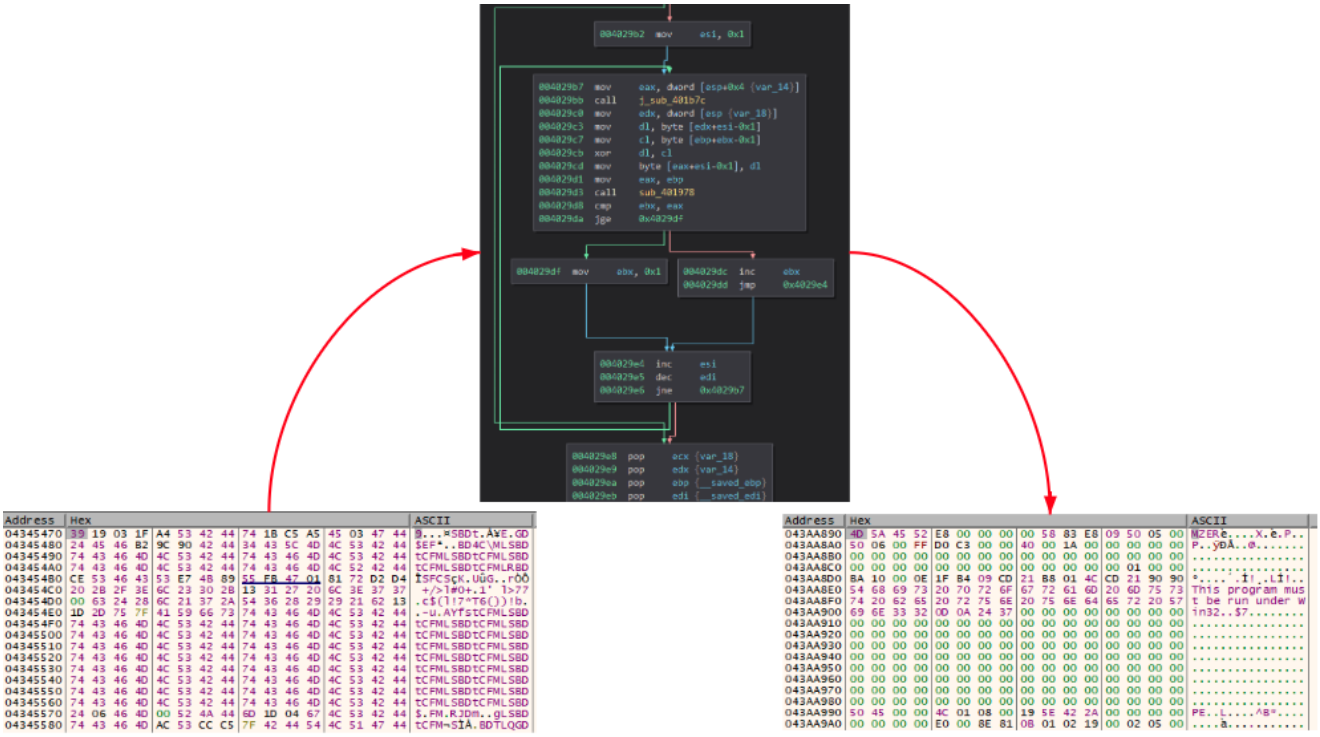
Like the stage 2 payload, this payload will also look for a specific pattern in a file, but instead of an external file it searches in the script.au3 script content. It looks for the “AU3!EA06” string (a known AutoIt script signature).

Usually this signature would be in the beginning of the file but in this case there’s another occurrence in the file. Once this string is found, the first 8 bytes next to the signature will be collected and saved for usage later:

```
00015620  01 A0 CE 40 6C C0 2C 3C 60 87 84 BF C9 97 3B FF  . I@lA,<`‡„¿E—;ÿ
00015630  D1 F5 93 73 A8 D8 C1 24 C6 E9 04 71 42 8C 92 3E  Ñõ"s¨ØÁ$Æé.qBŒ'>
00015640  A4 43 EF A7 87 74 DD 03 A8 CF 44 00 3E E7 61 E6  ¤Cï§‡tÝ.¨ÏD.>çaæ
00015650  23 47 B1 48 81 50 41 55 33 21 45 41 30 36 74 43  #G±H.PAU3!EA06tC
00015660  46 4D 4C 53 42 44 39 19 03 1F A4 53 42 44 74 1B  FMLSBD9...¤SBDt.
00015670  C5 A5 45 03 47 44 24 45 46 B2 9C 90 42 44 34 43  Å¥E.GD$EF²œ.BD4C
00015680  5C 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  \MLSBDtCFMLSBDtC
00015690  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
000156A0  46 4D 4C 52 42 44 CE 53 46 43 53 E7 4B 89 55 FB  FMLRBDÎSFCSçK‰Uû
000156B0  47 01 81 72 D2 D4 20 2B 2F 3E 6C 23 30 2B 13 31  G..rÒÔ +/>l#0+.1
000156C0  27 20 6C 3E 37 37 00 63 24 28 6C 21 37 2A 54 36  ' l>77.c$(l!7*T6
000156D0  28 29 29 21 62 13 1D 2D 75 7F 41 59 66 73 74 43  ())!b..-u.AYfstC
000156E0  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
000156F0  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
00015700  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
00015710  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
00015720  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
00015730  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
00015740  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
00015750  46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43  FMLSBDtCFMLSBDtC
00015760  46 4D 4C 53 42 44 24 06 46 4D 00 52 4A 44 6D 1D  FMLSBD$.FM.RJDm.
```

Occurrence of the AutoIt signature followed by a 8 bytes key value

The content next to the saved 8 bytes buffer is read and a multi-byte XOR operation is performed against it using the buffer as a XOR key. The result of this operation is the DarkGate final payload:



Multi-byte operation resulting in the final Darkgate payload

During the investigation we observed different XOR keys used for different payloads. The following is a list of some of the obtained keys:

SHA256	XOR key

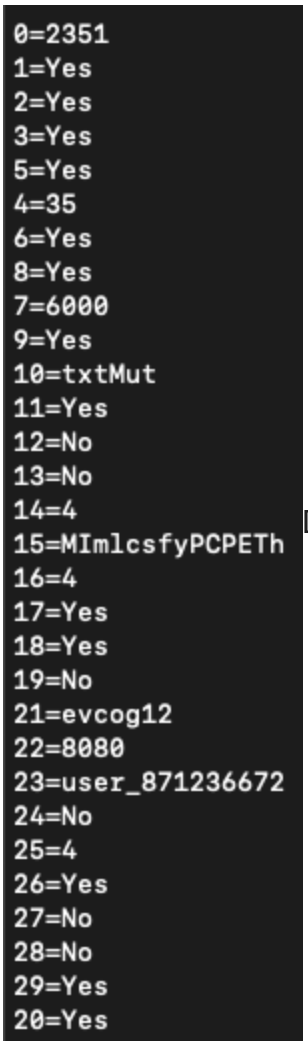
SHA256	XOR key
1fb6b8bed3a67ee4225f852c3d90fd2b629f2541ab431b4bd4d9d9f5bbd2c4b7	vJDAbKIz
567d828dab1022eda84f90592d6d95e331e0f2696e79ed7d86ddc095bb2efdc8 99f25de5cc5614f4efd967db0dae50f20e2acbae9e98920aff3d98638b9ca1f1 de3f49e68c45db2f31d1cc1d10ff09f8cfce302b92a1f5361c8f34c3d78544e5	ELkMtLfA
68952e8c311d1573b62d02c60a189e8c248530d4584eef1c7f0ff5ee20d730ab	RmDbBDsf
d4e766f81e567039c44ccca90ef192a7f063c1783224ee4be3e3d7786980e236	xfNwSUCl
5e94aa172460e74293db106a98327778ae2d32c6ce6592857a1ec0c581543572	tCFMLSBD

Exactly like the other stages, the execution flow will be transferred to the decoded file DOS header which will call the loader shellcode entry, and then the shellcode will call the DarkGate payload entry point:

```
04530B38    55              push ebp
04530B39    8BEC            mov ebp,esp
04530B3B    B9 0D000000     mov ecx,D                       D:'\r'
04530B40    6A 00           push 0
04530B42    6A 00           push 0
04530B44    49              dec ecx
04530B45    75 F9           jne 4530B40
04530B47    B8 50075304     mov eax,4530750
04530B4C    E8 D35CFBFF     call 44E6824
04530B51    33C0            xor eax,eax
04530B53    55              push ebp
04530B54    68 0A0F5304     push 4530F0A
04530B59    64:FF30         push dword ptr fs:[eax]
04530B5C    64:8920         mov dword ptr fs:[eax],esp
04530B5F    E8 7C31FFFF     call 4523CE0
04530B64    E8 9720FFFF     call 4522C00
04530B69    B8 200F5304     mov eax,4530F20                 4530F20:"c:\\darkgatedebugg"
04530B6E    E8 3D40FFFF     call 4524BB0
04530B73    84C0            test al,al
04530B75    74 0D           je 4530B84
04530B77    83CA FF         or edx,FFFFFFFF
04530B7A    B8 200F5304     mov eax,4530F20                 4530F20:"c:\\darkgatedebugg"
04530B7F    E8 E479FFFF     call 4528568
04530B84    8D45 EC         lea eax,dword ptr ss:[ebp-14]   [ebp-14]:"PE"
04530B87    E8 8CCAFEFF     call 451D618
04530B8C    8B45 EC         mov eax,dword ptr ss:[ebp-14]   [ebp-14]:"PE"
04530B8F    BA 3C0F5304     mov edx,4530F3C                 4530F3C:"SYSTEM"
04530B94    E8 4B3CFBFF     call 44E47E4
04530B99    75 1D           jne 4530BB8
04530B9B    A1 FC645304     mov eax,dword ptr ds:[45364FC]
04530BA0    C600 01         mov byte ptr ds:[eax],1
04530BA3    B8 4C0F5304     mov eax,4530F4C                 4530F4C:"c:\\temp\\ssy"
04530BA8    E8 1B3BFFFF     call 45246C8
04530BAD    84C0            test al,al
04530BAF    75 0F           jne 4530BC0
04530BB1    E8 2230FFFF     call 4523BD8
04530BB6    EB 08           jmp 4530BC0
04530BB8    A1 FC645304     mov eax,dword ptr ds:[45364FC]
04530BBD    C600 00         mov byte ptr ds:[eax],0
04530BC0    8D45 E8         lea eax,dword ptr ss:[ebp-18]
04530BC3    E8 50CAFEFF     call 451D618
04530BC8    8B45 E8         mov eax,dword ptr ss:[ebp-18]
04530BCB    BA 600F5304     mov edx,4530F60                 4530F60:"SafeMode"
04530BD0    E8 0F3CFBFF     call 44E47E4
```

DarkGate final payload entrypoint

The following is an example of the configuration extracted from the DarkGate payload:



DarkGate configuration example

In order to facilitate the final DarkGate payload extraction Netskope Threat Labs created a script to automate this process.

Netskope Detection

- Netskope Threat Protection
  - Win32.Trojan.TurtleLoader
  - Win32.Trojan.DarkGate
- Netskope Advanced Threat Protection provides proactive coverage against this threat.
  - Gen.Malware.Detect.By.StHeur indicates a sample that was detected using static analysis
  - Gen.Malware.Detect.By.Sandbox indicates a sample that was detected by our cloud sandbox

Conclusions

Although DarkGate is a threat created years ago it has been very active recently. Several campaigns involving different delivery and loading methods have been used, as well as new malware features being added, which requires a lot of action from the security community.

Netskope Threat Labs will continue to track how the DarkGate malware evolves and its TTP.

IOCs

All the IOCs related to this campaign, scripts, and the Yara rules can be found in our GitHub repository.