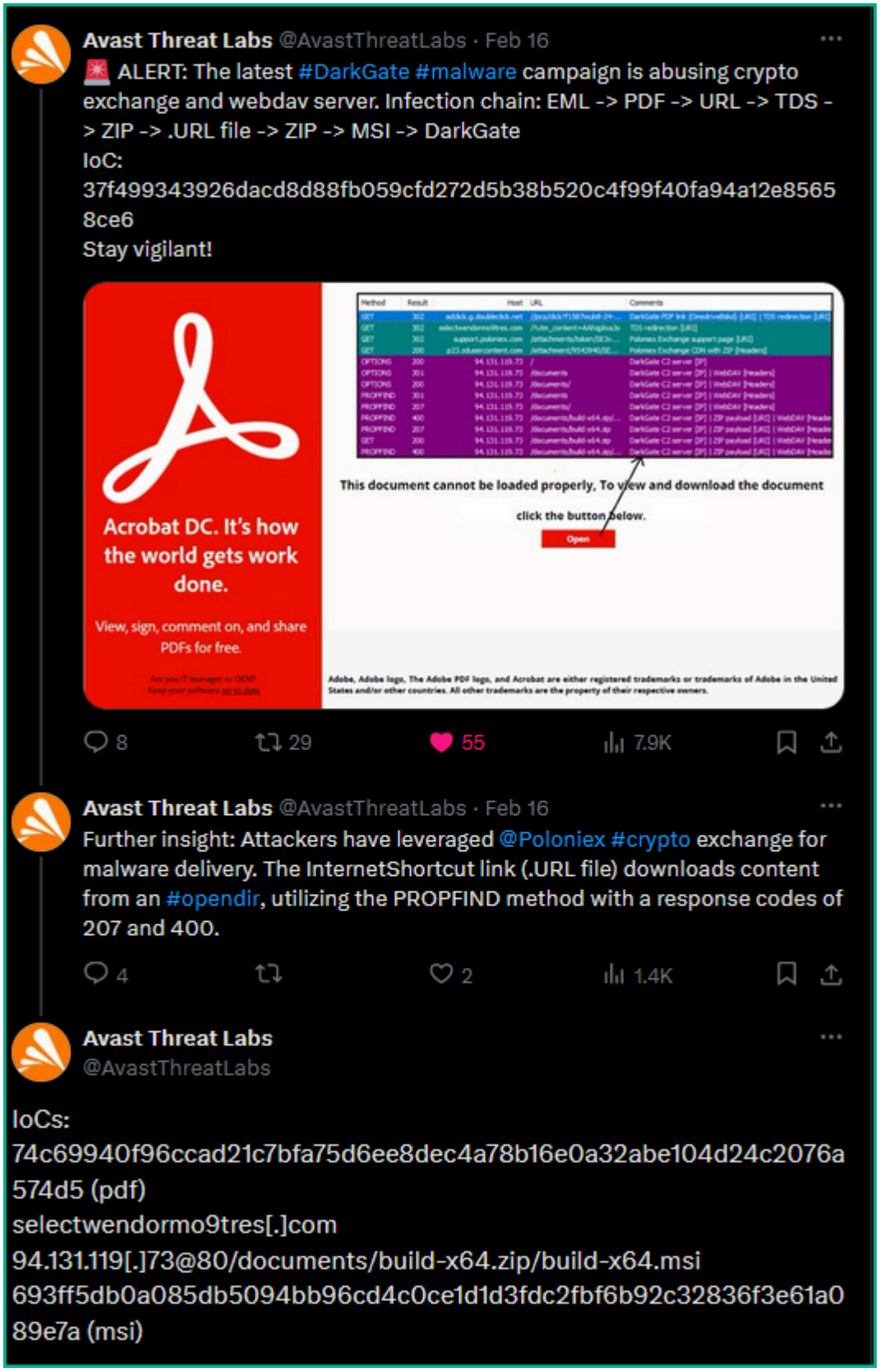
[QuickNote] DarkGate – Make AutoIt Great Again

Published: 2024-06-06 · Archived: 2026-04-10 03:08:46 UTC

1 Votes

1. Context

In the first quarter of 2024, @AvastThreatLabs observed a **DarkGate** campaign distributed via malicious PDF files:



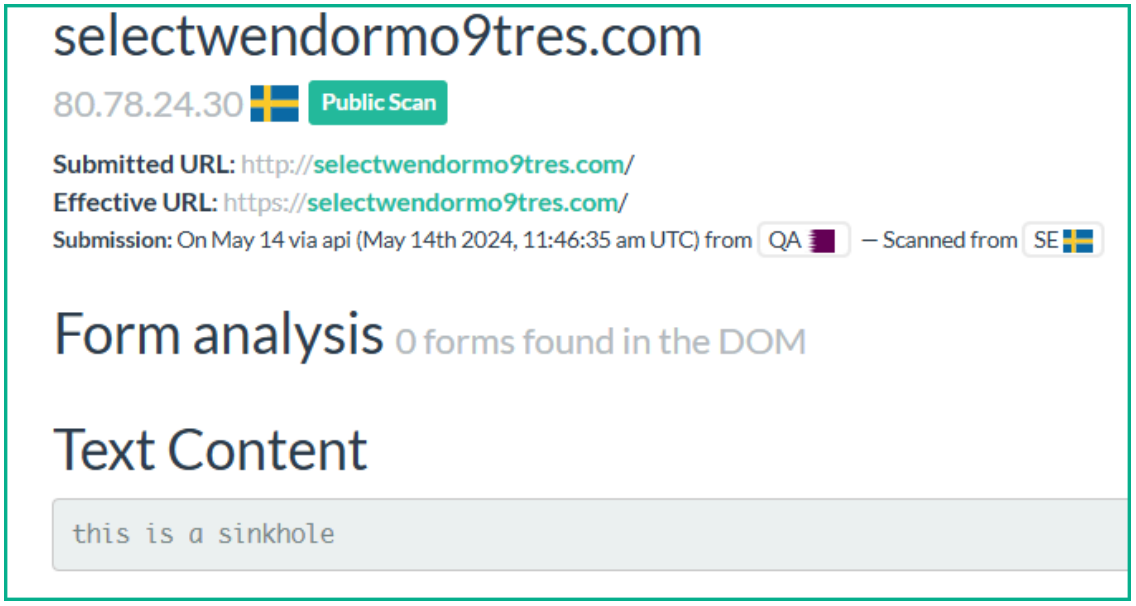
The technique of using PDF files to lure users into downloading and executing files containing malware is not new. I have previously discussed this technique in my presentation titled Unveiling Qakbot: Exploring one of the Most Active Threat Actors at the **Security Bootcamp 2023 (SBC2023)** conference. We can search for more

information about the domain `selectwendormo9tres[.]com` on **VirusTotal** reveal that multiple samples are used to access this domain for downloading files.

Communicating Files (10)

Scanned	Detections	Type	Name
2024-02-28	11 / 58	PDF	case_-2024_6833292639.pdf
2024-05-30	23 / 64	PDF	February_-2024_3398702636.pdf
2024-05-29	23 / 65	PDF	feb_-2024_9804218831-1.pdf
2024-02-27	13 / 59	PDF	case_-2023_4449725749.pdf
2024-05-13	35 / 64	PDF	74c69940f96ccad21c7bfa75d6ee8dec4a78b16e0a32abe104d24c2076a574d5.pdf
2024-02-28	10 / 59	PDF	info_-2023_3443310460.pdf
2024-02-13	1 / 61	PDF	feb_-2023_8692935239.pdf
2024-02-29	7 / 59	PDF	cb3a1ec802a7304fda367d9c5dec6c48e4e7f25a1c8c0534db212f74c8b3b81d
2024-04-08	22 / 56	PDF	message[.]pdf
2024-02-27	8 / 59	PDF	feb_-2023_4630979418.pdf

However, the domain is currently being used as a sinkhole, which helps users avoid downloading malicious content. Sinkholes are a valuable defense mechanism in cybersecurity. They redirect malicious traffic to a controlled server, preventing it from reaching its intended malicious destination.

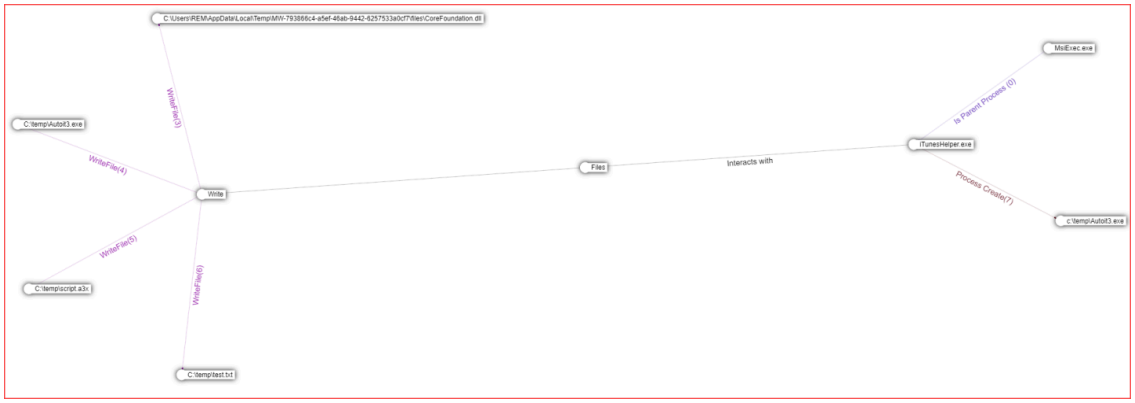


Therefore, in this article, I will utilize the IOCs released by @AvastThreatLabs to conduct a thorough analysis:

1. case_-2023_4824647818.pdf
2. build-x64.msi

2. Execution Flow Summary

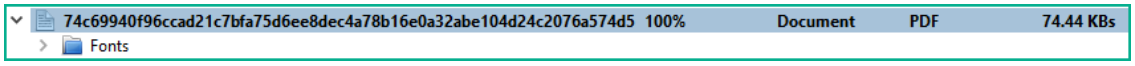
Here’s the high-level illustration of the malware execution flow:



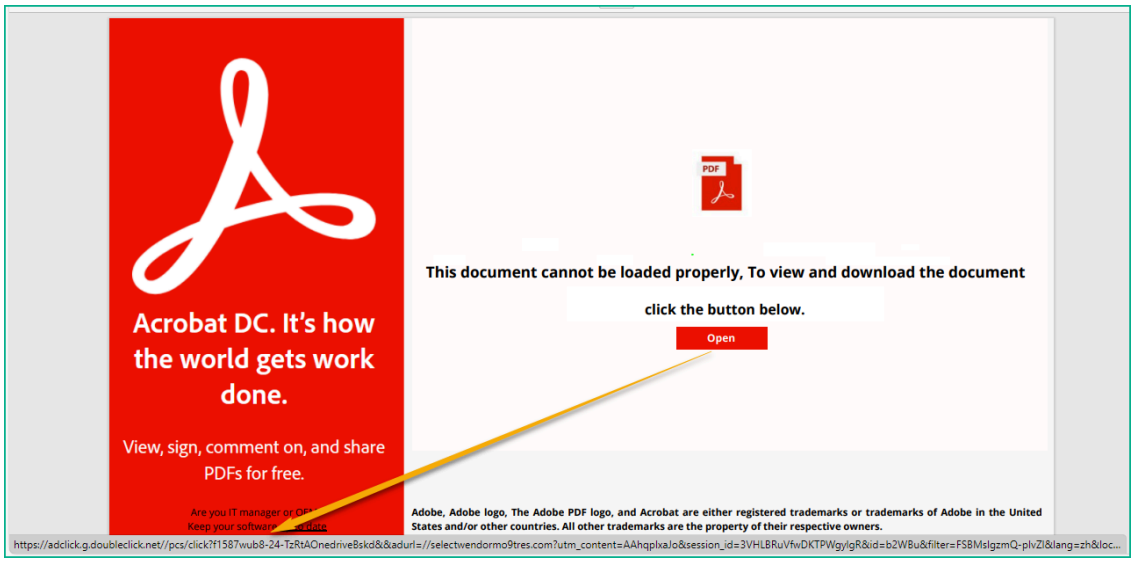
3. Technical Analysis

3.1. Analyze PDF file

The PDF file is quite small in size:



Normal users, when opening this file, will be prompted to click the “ `Open` ” button, which will then lead them to the following link to download a malicious file: `hxxps[:]//adclick[.]g.doubleclick[.]net//pcs/click?f1587wub8-24-TzRtAOnedriveBskd&&adurl=//selectwendormo9tres[.]com?utm_content=AAhqplxaJo&session_id=3VHLBRuVfwDKTPWgylgR&id=b2WBu&filter=FSBMsIgzmQ-pIvZl&lang=zh&locale=US`



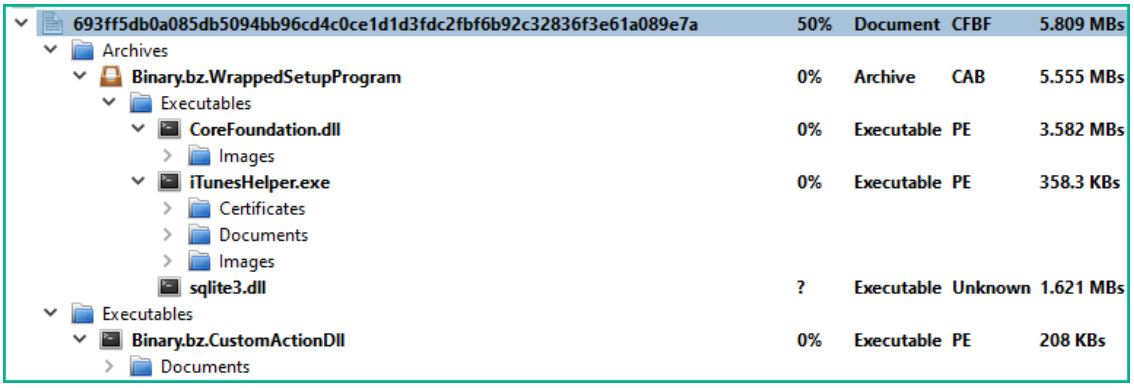
Similar to the February_-2024_3398702636.pdf file, this file will lure users to access the following link: `hxxps[:]//adclick[.]g.doubleclick[.]net//pcs/click?f6879wbk1-2024-CnPlUOnedriveFrkd&&adurl=//selectwendormo9tres[.]com?utm_content=orHchiCJYv&session_id=QLuKkySnvaDSwwY9mSwT&id=Uwgtv&filter=PBeVeNVqPB-dEdGa&lang=es&locale=DE`

Quick comparison of the links above:

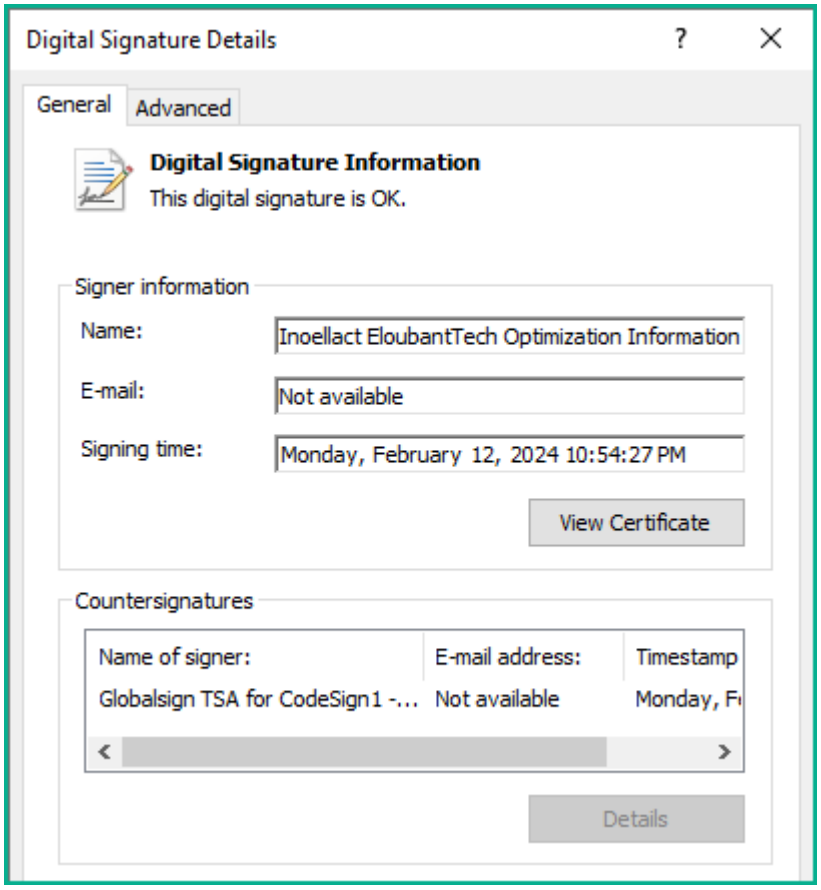
According to @AvastThreatLabs, these links will download an MSI file named “ `build-x64.msi` “. If users trust and execute this MSI file, it will infect their system with the **DarkGate** malware.

3.2. Analyze MSI file

The structure of the `build-x64.msi` file is as follows:



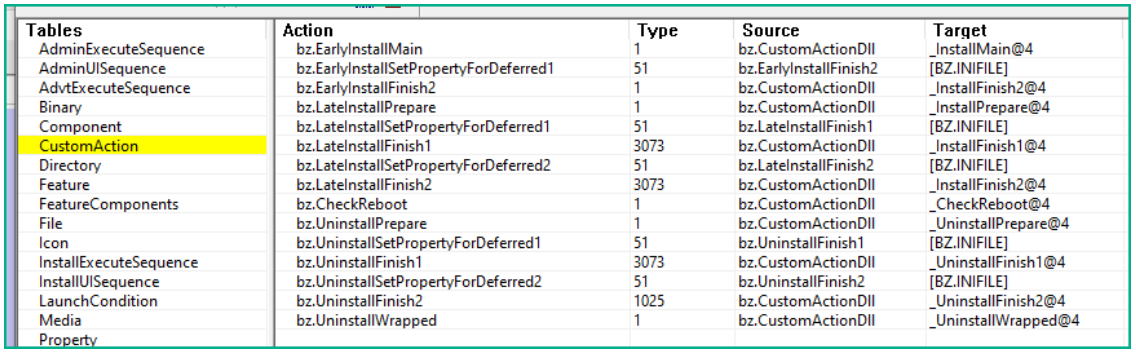
It uses a valid digital signature:



```
[+] Included certificates:
  - Subject: CN=GlobalSign GCC R45 EV CodeSigning CA 2020, O=GlobalSign nv-sa, C=BE
    Issuer: CN=GlobalSign Code Signing Root R45, O=GlobalSign nv-sa, C=BE
    Serial: 159159760011286741492753271723304908269
    Valid from: 2020-07-28 00:00:00+00:00
    Valid to: 2030-07-28 00:00:00+00:00
  - Subject: CN=Inoellact EloubantTech Optimization Information Co.\, Ltd., O=Inoellact
EloubantTech Optimization Information Co.\, Ltd., STREET=Room 502\, No.22\, Jiangbu
Road\, Dali Town\, Nanhai District, L=Foshan, ST=Guangdong, C=CN,
1.3.6.1.4.1.311.60.2.1.1=Foshan, 1.3.6.1.4.1.311.60.2.1.2=Guangdong,
1.3.6.1.4.1.311.60.2.1.3=CN, 2.5.4.5=91440605MACRJLFMXL, 2.5.4.15=Private Organization
    Issuer: CN=GlobalSign GCC R45 EV CodeSigning CA 2020, O=GlobalSign nv-sa, C=BE
    Serial: 338308593044112860324718746 7
    Valid from: 2024-01-26 09:28:10+00:00
    Valid to: 2025-01-26 09:28:10+00:00

  [+] Signer:
    Issuer: CN=GlobalSign GCC R45 EV CodeSigning CA 2020, O=GlobalSign nv-sa, C=BE
    Serial: 338308593044112860324718746 7
    Program name: None
    More info: None

  [+] Countersigner (nested RFC3161):
    Issuer: CN=GlobalSign Timestamping CA - SHA384 - G4, O=GlobalSign nv-sa, C=BE
    Serial: 225629295226172135187772734291733796 2
    Signing time: 2024-02-12 15:54:27+00:00
    Included certificates:
      - Subject: CN=Globalsign TSA for CodeSign1 - R6, O=GlobalSign nv-sa, C=BE
        Issuer: CN=GlobalSign Timestamping CA - SHA384 - G4, O=GlobalSign nv-sa, C=BE
        Serial: 225629295226172135187772734291733796 2
        Valid from: 2022-04-06 07:45:38+00:00
        Valid to: 2033-05-08 07:45:38+00:00
      - Subject: CN=GlobalSign Timestamping CA - SHA384 - G4, O=GlobalSign nv-sa, C=BE
        Issuer: CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R6
        Serial: 152301165417217153014605563764
        Valid from: 2018-06-20 00:00:00+00:00
        Valid to: 2034-12-10 00:00:00+00:00
      - Subject: CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R6
        Issuer: CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R6
        Serial: 141776661797344498925267030161953 7
        Valid from: 2014-12-10 00:00:00+00:00
        Valid to: 2034-12-10 00:00:00+00:00
```

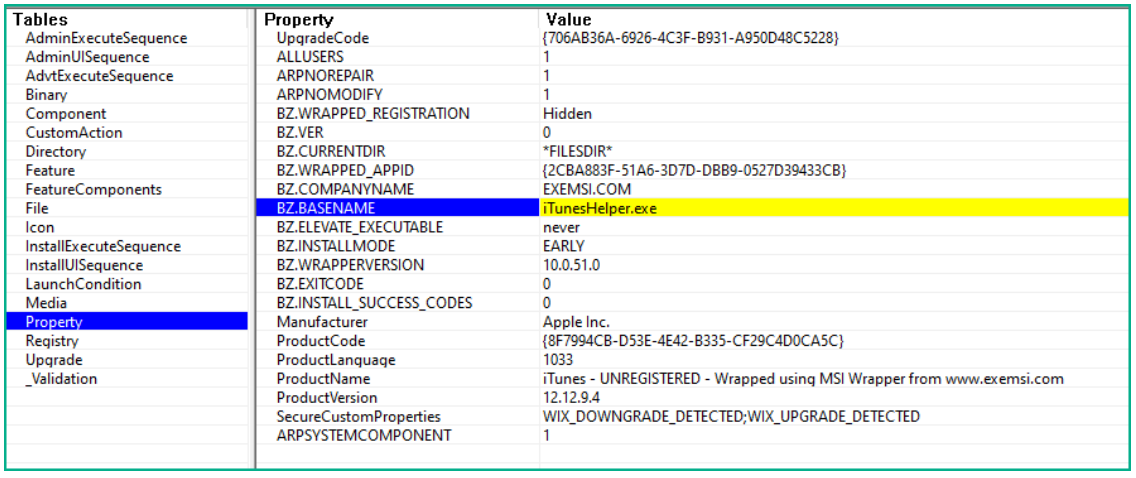
```
 [+] Digest algorithm: openssl_sha256
[+] Digest: 69d4769c1244a1dbd1222c91f7efd4d39bc3b46edb5c9a53061cee3843e33932
```

Further examination using the Orca tool reveals that the `CustomAction` section executes the `bz.CustomActionDll` file:



Tables	Action	Type	Source	Target
AdminExecuteSequence	bz.EarlyInstallMain	1	bz.CustomActionDll	_InstallMain@4
AdminUISequence	bz.EarlyInstallSetPropertyForDeferred1	51	bz.EarlyInstallFinish2	[BZ.INIFILE]
AdvtExecuteSequence	bz.EarlyInstallFinish2	1	bz.CustomActionDll	_InstallFinish2@4
Binary	bz.LateInstallPrepare	1	bz.CustomActionDll	_InstallPrepare@4
Component	bz.LateInstallSetPropertyForDeferred1	51	bz.LateInstallFinish1	[BZ.INIFILE]
CustomAction	bz.LateInstallFinish1	3073	bz.CustomActionDll	_InstallFinish1@4
Directory	bz.LateInstallSetPropertyForDeferred2	51	bz.LateInstallFinish2	[BZ.INIFILE]
Feature	bz.LateInstallFinish2	3073	bz.CustomActionDll	_InstallFinish2@4
FeatureComponents	bz.CheckReboot	1	bz.CustomActionDll	_CheckReboot@4
File	bz.UninstallPrepare	1	bz.CustomActionDll	_UninstallPrepare@4
Icon	bz.UninstallSetPropertyForDeferred1	51	bz.UninstallFinish1	[BZ.INIFILE]
InstallExecuteSequence	bz.UninstallFinish1	3073	bz.CustomActionDll	_UninstallFinish1@4
InstallUISequence	bz.UninstallSetPropertyForDeferred2	51	bz.UninstallFinish2	[BZ.INIFILE]
LaunchCondition	bz.UninstallFinish2	1025	bz.CustomActionDll	_UninstallFinish2@4
Media	bz.UninstallWrapped	1	bz.CustomActionDll	_UninstallWrapped@4
Property				

The `Property` section has property field related to `iTunesHelper.exe` file:



Tables	Property	Value
AdminExecuteSequence	UpgradeCode	{706AB36A-6926-4C3F-B931-A950D48C5228}
AdminUISequence	ALLUSERS	1
AdvtExecuteSequence	ARPNOREPAIR	1
Binary	ARPNOMODIFY	1
Component	BZ.WRAPPED_REGISTRATION	Hidden
CustomAction	BZ.VER	0
Directory	BZ.CURRENTDIR	*FILESDIR*
Feature	BZ.WRAPPED_APPID	{2CBA883F-51A6-3D7D-DBB9-0527D39433CB}
FeatureComponents	BZ.COMPANYNAME	EXEMSI.COM
File	BZ.BASENAME	iTunesHelper.exe
Icon	BZ.ELEVATE_EXECUTABLE	never
InstallExecuteSequence	BZ.INSTALLMODE	EARLY
InstallUISequence	BZ.WRAPPERVERSION	10.0.51.0
LaunchCondition	BZ.EXITCODE	0
Media	BZ.INSTALL_SUCCESS_CODES	0
Property	Manufacturer	Apple Inc.
Registry	ProductCode	{8F7994CB-D53E-4E42-B335-CF29C4D0CA5C}
Upgrade	ProductLanguage	1033
_Validation	ProductName	iTunes - UNREGISTERED - Wrapped using MSI Wrapper from www.exemsi.com
	ProductVersion	12.12.9.4
	SecureCustomProperties	WIX_DOWNGRADE_DETECTED;WIX_UPGRADE_DETECTED
	ARPSYSTEMCOMPONENT	1

3.3. Analyze bz.CustomActionDll file

A quick review of the `bz.CustomActionDll` code reveals that it will interact with the `iTunesHelper.exe` file located in the `bz.WrappedSetupProgram` cab file:

```
v48 = (sub_10003FF0(Path + 0xFFFFFFFC) + 4);
LOBYTE(v66) = 3;
sub_100093D0(&v48, L"msiwrapper.ini", 0xE);
v43 = v7;
v62 = &v43;
v54 = v48 - 0x10;
v8 = sub_10003FF0((v48 - 0x10));
sub_10009840(v63, (v8 + 4));
LOBYTE(v66) = 4;
sub_1000C3B0(L"BZ.WRAPPED_APPID", a1, &v52);
LOBYTE(v66) = 5;
sub_10009F50(v63, v52, L"WrappedApplicationId");
sub_1000C3B0(L"BZ.WRAPPED_REGISTRATION", a1, &v51);
LOBYTE(v66) = 6;
sub_10009F50(v63, v51, L"WrappedRegistration");
sub_1000C3B0(L"BZ.INSTALL_SUCCESS_CODES", a1, &v50);
LOBYTE(v66) = 7;
sub_10009F50(v63, v50, L"InstallSuccessCodes");
sub_10009F50(v63, v47, L"ElevationMode");
v46 = (sub_10003FF0(Path + 0xFFFFFFFC) + 4);
LOBYTE(v66) = 8;
sub_100093D0(&v46, L"files", 5);
sub_100093D0(&v46, L"\\", 1);
sub_1000C3B0(L"BZ.BASENAME", a1, &v49);
LOBYTE(v66) = 9;
if ( !*(v49 - 0xC) )
{
  sub_10009610(L"Unable to get base name of wrapped setup.", a1);
  LOBYTE(v66) = 8;
```

3.4. Analyze iTunesHelper.exe file

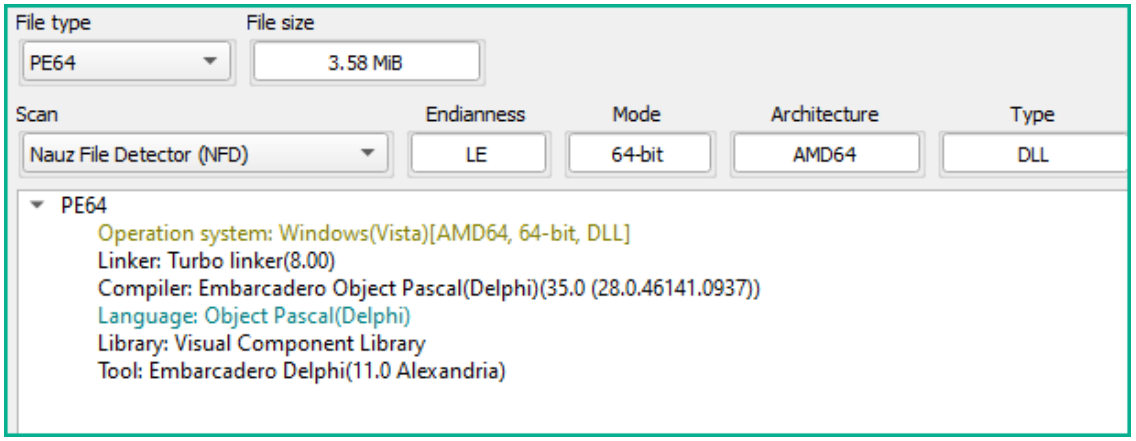
The `iTunesHelper.exe` file is a genuine Apple file, with its Pdb path is: `D:\BWA\6B22E293-2BF5-0\iTunesWin-1200.12.12.9.4\srcroot\iTunes\iPodSupport\(Win32)\BuildResults\Production64\bin\iTunesHelper.pdb`. Examining the **Import Directory** of this file reveals that it loads a DLL file called `CoreFoundation.dll`.

Module Name	Imports (n)	OriginalF…	TimeDateStamp	ForwarderChain	Name	FirstThunk
CoreFoundation.dll	5	0003D310	00000000	00000000	0003D89E	0002E058
SETUPAPI.dll	7	0003D720	00000000	00000000	0003D97C	0002E468
SHLWAPI.dll	2	0003D760	00000000	00000000	0003D9AE	0002E4A8
KERNEL32.dll	115	0003D340	00000000	00000000	0003DD84	0002E088
USER32.dll	14	0003D778	00000000	00000000	0003DE76	0002E4C0
ADVAPI32.dll	10	0003D2B8	00000000	00000000	0003DF2A	0002E000
ole32.dll	9	0003D7F0	00000000	00000000	0003DFF8	0002E538
OLEAUT32.dll	7	0003D6E0	00000000	00000000	0003E002	0002E428

The `CoreFoundation.dll` file is included within the `bz.WrappedSetupProgram` cab file. This indicates that the attacker employed the Dll Side Loading techniques to load a DLL file containing code responsible for executing the task of malware propagation.

3.5. Analyze CoreFoundation.dll file

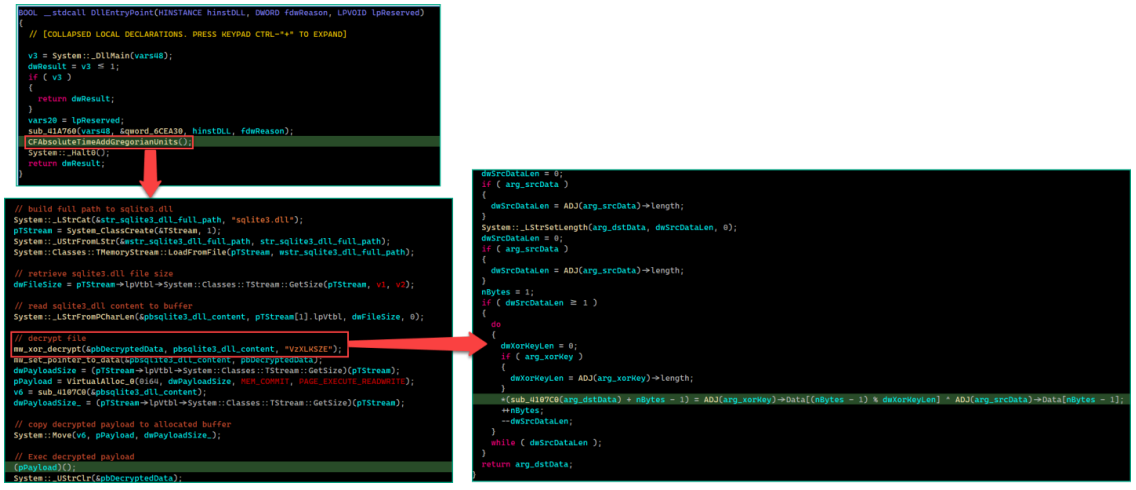
This is a 64-bit DLL without a digital signature:



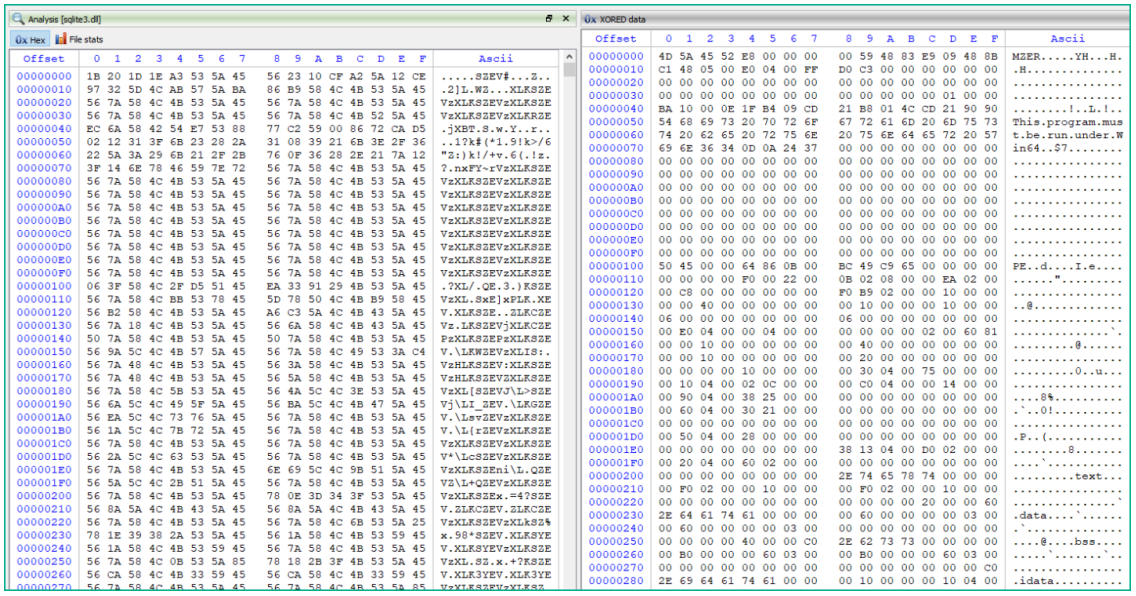
It has the following information:

```
FileDescription: Project1
FileVersion: 1.0.0.0
ProgramID: com.embarcadero.Project1
ProductName: Project1
ProductVersion: 1.0.0.0
```

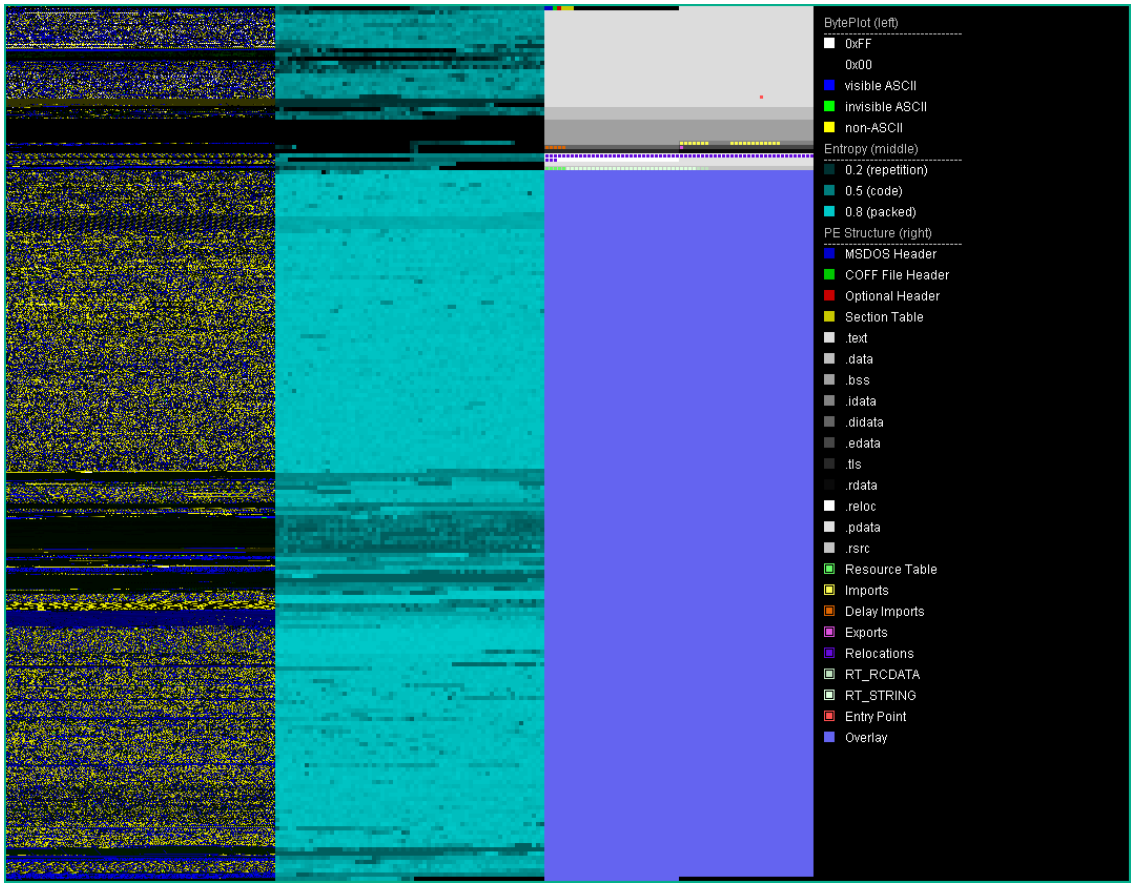
Loading this DLL file into IDA, the `CFAbsoluteTimeAddGregorianUnits()` function is called from `DllEntryPoint`. The function's task is to read encrypted data stored in the `sqlite3.dll` file into a buffer, decrypt it using XOR with the decryption key "`VzXLKSZE`", and finally execute the decrypted payload:



The `sqlite3.dll` file before and after the decryption process:



The decrypted file (`sqlite3_xored_VzXLKSZE.dll`) contains a PE file with `SizeOfImage = 0x4E000` .

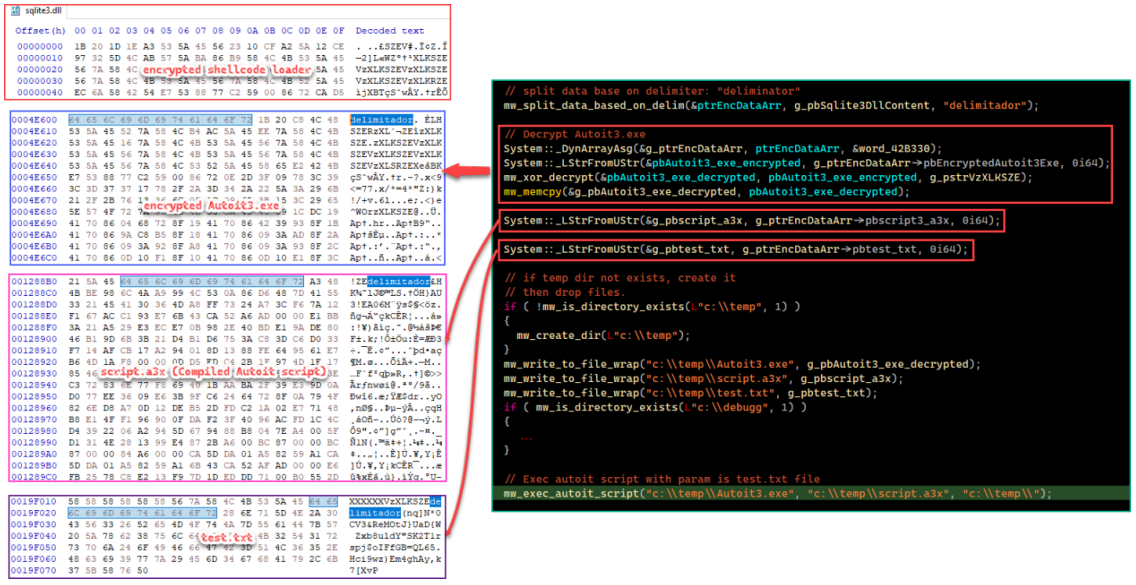


3.6. Analyze sqlite3_xored_VzXLKSZE.dll file

The file has a header of “ `MZER` ”, enabling it to execute as a shellcode. This shellcode maps the file into memory and then calls the OEP address of the file to execute the main code. Its code first reads the contents of the original “ `sqlite3.dll` ” file into the memory:

```
// Assign xor key to global var
mw_memcpy(&g_pstrVzXLKSZE, "VzXLKSZE");
if ( mw_is_directory_exists(L"c:\\debugg", 1) )
{
  ...
}

// retrieve full path to sqlite3.dll
System::ParamStr(&pwstrCurrExecutionFilePath, 0);
System::Sysutils::ExtractFileDir(&pwstrCurrExecutionDir, pwstrCurrExecutionFilePath);
System::Ansistrings::IncludeTrailingPathDelimiter(&pwstrCurrExecutionDirPlusBackslash, pwstrCurrExecutionDir);
System::_LStrFromUStr(&g_pstrSqlite3DllFullPath, pwstrCurrExecutionDirPlusBackslash, 0i64);
mw_append_str(&g_pstrSqlite3DllFullPath, "sqlite3.dll");

// Read content of sqlite3.dll to buffer
mw_read_file_content(&pbSqlite3DllContent, g_pstrSqlite3DllFullPath);
mw_memcpy(&g_pbSqlite3DllContent, pbSqlite3DllContent);
```

Next, it performs a search for the string “ `delimitador` ” and use it to separate the data sections corresponding to the files. Then, decrypt Autoit3.exe file using the same decryption key as above, which is “ `VzXLKSZE` ”. Check if the directory `c:\temp` exists, if it does not exist, create it and drop the following files into it: `Autoit3.exe, script.a3x, test.txt` . Finally, use the `CreateProcessA` API to execute `Autoit3.exe` for loading AutoIt script is `script.a3x` . The script will take the file `test.txt` as a parameter.



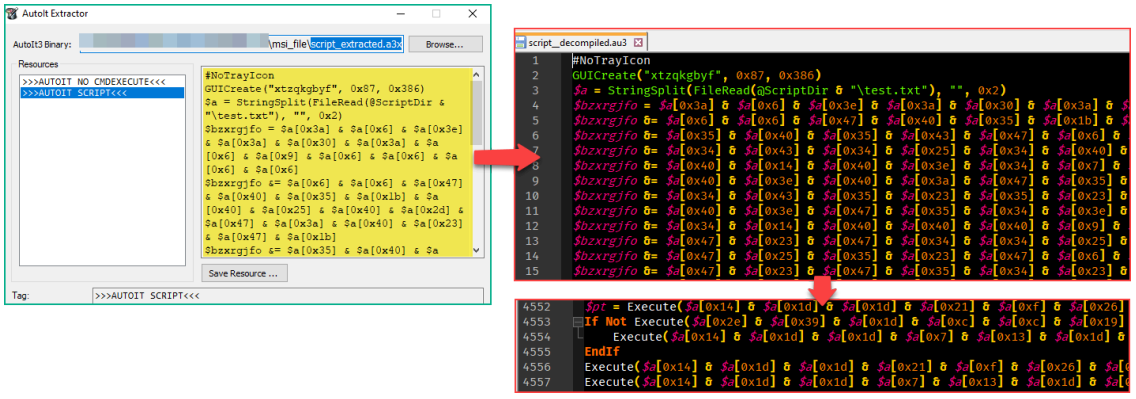
3.7. Extract and analyze AutoIt script

Based on the analysis above, the compiled script’s content can be extracted from `0x1288BE` to `0x19F01E` (size: `0x76760` bytes):

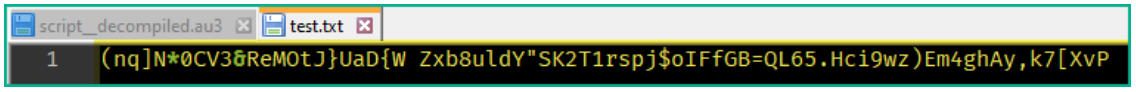
script_extracted.a3x

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	£HK¾˜lJ©™LS.†ÖH}
00000010	41	55	33	21	45	41	30	36	4D	A8	FF	73	24	A7	3C	F6	AU3!EA06M¨ÿs$§<ö
00000020	7A	12	F1	67	AC	C1	93	E7	6B	43	CA	52	A6	AD	00	00	z.ñg¬Á"çkCÊR¦...
00000030	E1	BB	3A	21	A5	29	E3	EC	E7	0B	98	2E	40	BD	E1	9A	á»:!¥)ãìç.˜.@½áš
00000040	DE	80	46	B1	9D	6B	3B	21	D4	B1	D6	75	3A	C8	3D	C6	Þ€F±.k;!Ô±Öu:È=Æ
00000050	D0	33	F7	14	AF	CB	17	A2	94	01	8D	13	88	FE	64	95	Ð3÷.¯Ë.¢”...ˆþd•
00000060	61	E7	B6	4D	1A	F8	00	00	0D	D5	ED	C4	2B	1F	97	4D	aç¶M.ø...ÕíÄ+.—M
00000070	1E	17	85	46	B4	66	B2	71	FE	BB	52	2C	2E	86	5D	A9	...…F´f²qþ»R,.†]©
00000080	3E	3E	C3	72	83	6E	77	F8	69	40	1B	AA	BA	2F	39	E3	>>Ãrƒnwøi@.ª°/9ã
00000090	9D	0A	D0	77	EE	36	09	E6	3B	9F	C6	24	64	72	8F	0A	..Ðwî6.æ;ŸÆ$dr..
000000A0	79	4F	82	6E	D8	A7	0D	12	DE	B5	2D	FD	C2	1A	02	E7	yO‚nØ§..Þµ-ýÂ..ç
000000B0	71	48	B8	E1	4F	F1	96	90	0F	DA	F2	3F	40	96	AC	FD	qH¸áOñ–..Úò?@–¬ý
000000C0	1C	4C	D4	39	22	06	A2	94	5D	67	94	88	B8	04	7E	A4	.LÔ9".¢”]g”ˆ¸.~¤
000000D0	00	5F	D1	31	4E	28	13	99	E4	87	2B	A6	00	BC	87	00	._Ñ1N(.™ä‡+¦.¼‡.
000000E0	00	BC	87	00	00	84	A6	00	00	CA	5D	DA	01	A5	82	59	.¼‡..„¦..Ê]Ú.¥‚Y
000000F0	A1	CA	5D	DA	01	A5	82	59	A1	6B	43	CA	52	AF	AD	00	¡Ê]Ú.¥‚Y¡kCÊR¯..
00000100	00	E6	FB	25	78	C8	E2	13	F9	7D	1D	ED	DD	71	00	B0	.æû%xÈâ.ù}.íÝq.°
00000110	55	2D	AC	9A	D5	28	15	D4	F0	CF	25	E4	CF	11	8E	56	U-¬šÕ(.ÔðÏ%äÏ.ŽV

By leveraging the **AutoIt Extractor** tool, we can effortlessly extract the original script:

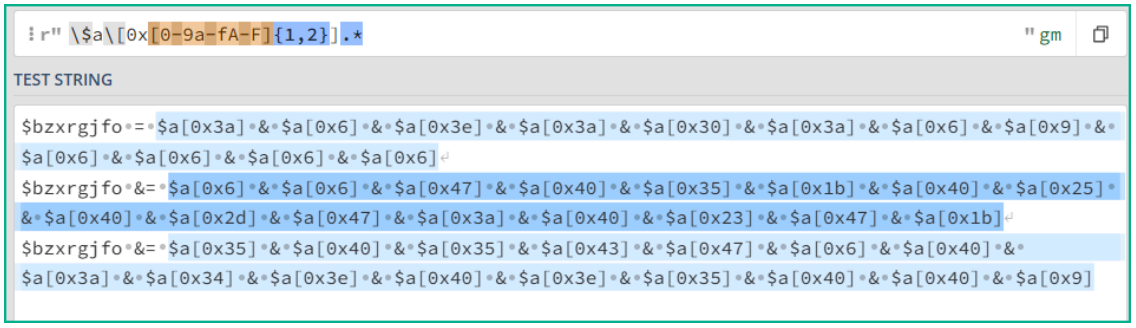


Analyzing the script, it uses the FileRead function to read the contents from the `test.txt` file. Then, it employs StringSplit function on the retrieved string to generate an array of characters. Based on this character array stored in the $a variable, corresponding strings are constructed. The contents of the `test.txt` file is as follows:

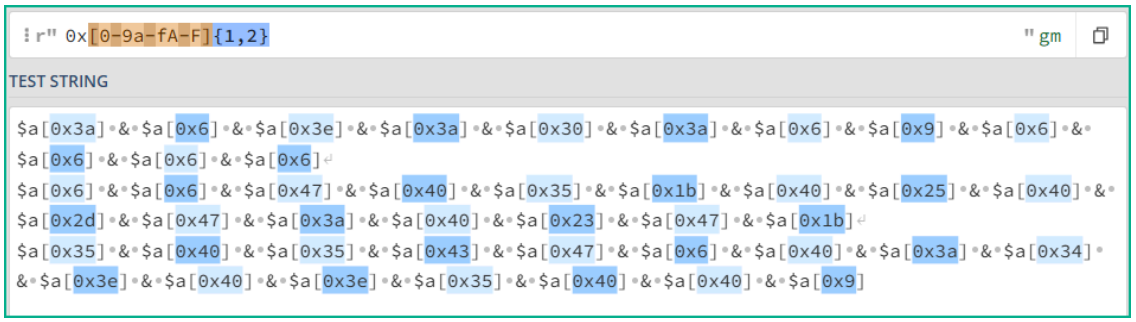


However, since the script is **4557 lines** long, manual replacement is not feasible. An automated script needs to be written to perform this de-obfuscation task and restore the corresponding strings. To achieve this, I utilize two regular expressions as follows:

- `r"\$a\[0x[0-9a-fA-F]{1,2}].*"` : This regex extracts the entire encoded string after variable `$bzxrgjfo =`



- `r"0x[0-9a-fA-F]{1,2}"` : This regex extracts all hexadecimal codes within the string.



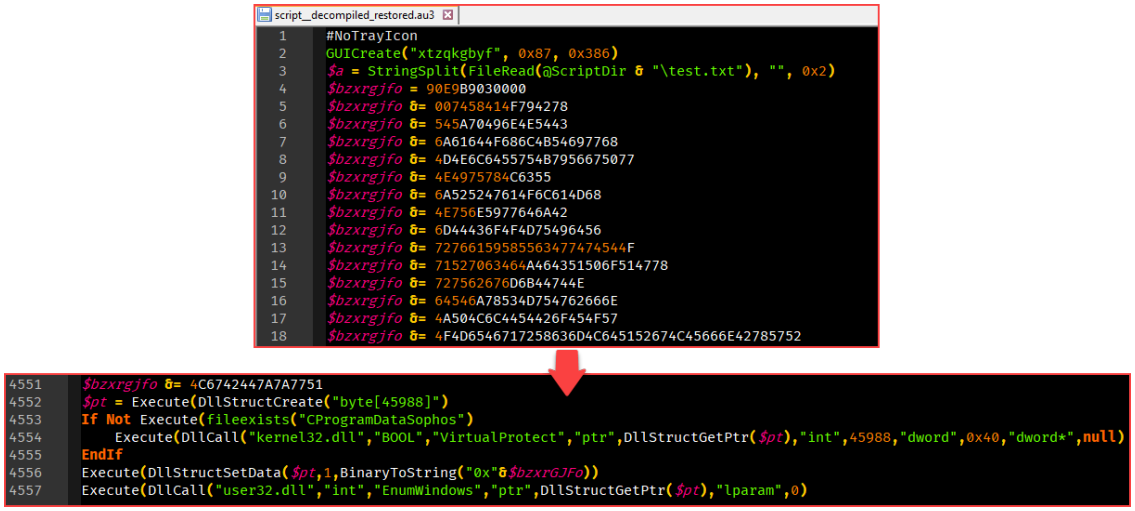
The entire Python code I wrote to recover the script is as follows:

```
from argparse import ArgumentParser
import sys
import re

# Regular expression pattern to match encoded string
pattern_1 = r"\$a\[0x[0-9a-fA-F]{1,2}].*"
# Regular expression pattern to match hex codes
pattern_2 = r"0x[0-9a-fA-F]{1,2}"

# Define a dictionary to map num to characters
char_map = {0: '(', 1: 'n', 2: 'q', 3: ']', 4: 'N', 5: '*', 6: '0', 7: 'C', 8: 'V', 9: '3', 10: '&',
            11: 'R', 12: 'e', 13: 'M', 14: 'O', 15: 't', 16: 'J', 17: '}', 18: 'U', 19: 'a', 20: 'D'
            21: '{', 22: 'W', 23: ' ', 24: 'Z', 25: 'x', 26: 'b', 27: '8', 28: 'u', 29: 'l', 30: 'd'
            31: 'Y', 32: '"', 33: 'S', 34: 'K', 35: '2', 36: 'T', 37: '1', 38: 'r', 39: 's', 40: 'p'
            41: 'j', 42: '$', 43: 'o', 44: 'I', 45: 'F', 46: 'f', 47: 'G', 48: 'B', 49: '=', 50: 'Q'
            51: 'L', 52: '6', 53: '5', 54: '.', 55: 'H', 56: 'c', 57: 'i', 58: '9', 59: 'w', 60: 'z'
            61: ')', 62: 'E', 63: 'm', 64: '4', 65: 'g', 66: 'h', 67: 'A', 68: 'y', 69: ',', 70: 'k'
            71: '7', 72: '[', 73: 'X', 74: 'v', 75: 'P'}

def mapping_char(listCharCode):
    message = ""
    for num in listCharCode:
        message += char_map.get(num)
    # return the final message
    return message
```

```
def extract_index_from_str(text):
    # Find all matches of the regex pattern
    matches = re.findall(pattern_2, text)

    # Convert hex codes to integers
    idxArr = [int(hex_code, 16) for hex_code in matches]

    return idxArr

def main():
    parser = ArgumentParser(description="Deobfuscating DarkGate AutoIt script!!")
    parser.add_argument("-i", "--input_file", dest='input_file', metavar='INPUT_FILE', type=str, requ
    parser.add_argument("-o", "--output_file", dest='output_file', metavar='OUTPUT_FILE', type=str,

    args = parser.parse_args()

    try:
        fin = open(args.input_file, "r")
    except IOError as e:
        print("Could not open file %s - %s" % (args.input_file, e))
        return 1

    in_lines = fin.readlines()
    fin.close()

    out_lines = []
    for line in in_lines:
        matched_str = re.search(pattern_1, line)
        if matched_str:
            strEncoded = matched_str.group(0)
            idxArray = extract_index_from_str(strEncoded)
            strDecoded = mapping_char(idxArray)
            line = line.replace(strEncoded,strDecoded)

        out_lines.append(line)

    if out_lines:
        try:
            fout = open(args.output_file, "w+")
        except IOError as e:
            print("Coult not write to file %s - %s" % (args.output_file, e))
            fout = sys.stdout

        for line in out_lines:
            fout.write(line)
        fout.close()
```

```
        print("Deobfuscating done :)")
        return  0   # 0 = EXIT_SUCESS
    return 1    # 1 is EXIT_FAILURE

if __name__ == "__main__":
    sys.exit(main())
```

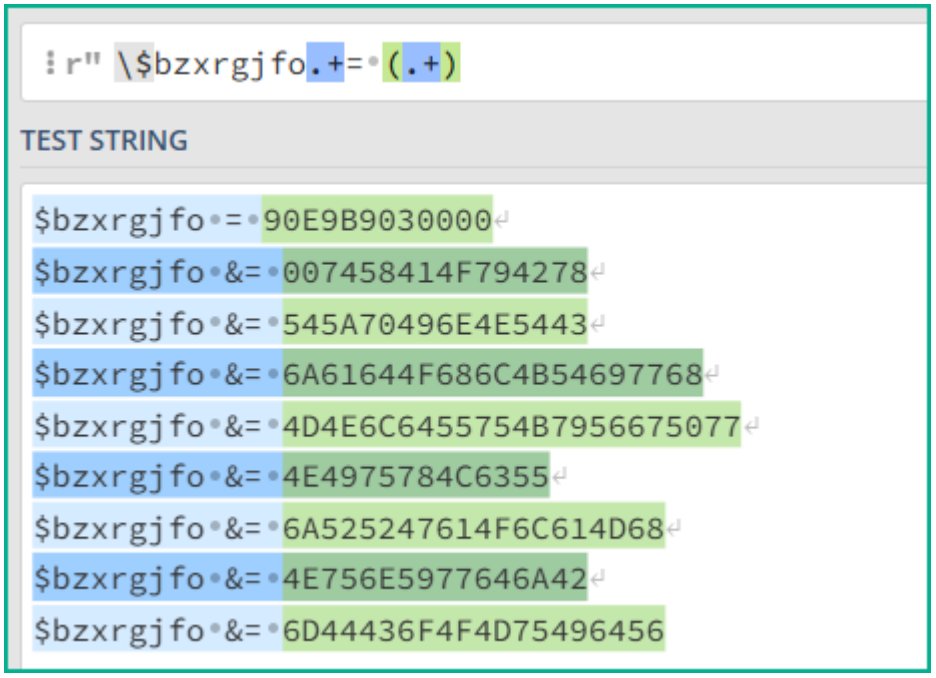
Here is the final result:



As a result, the variable `$bzxrGJFo` will store the shellcode, which is 45,988 bytes in size. The script copies this shellcode to the variable `$pt`, and utilizes the `EnumWindows` function to execute it.

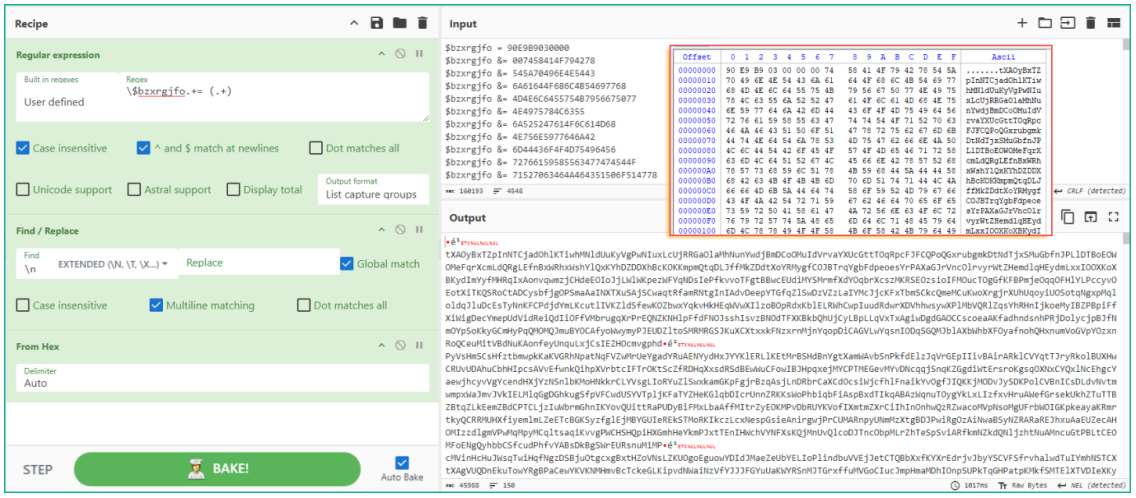
3.8. Extract and analyze DarkGate shellcode loader

For extracting the shellcode content from the above script, I used the following regex: `"\$bzxrgjfo.+= (.+)"`

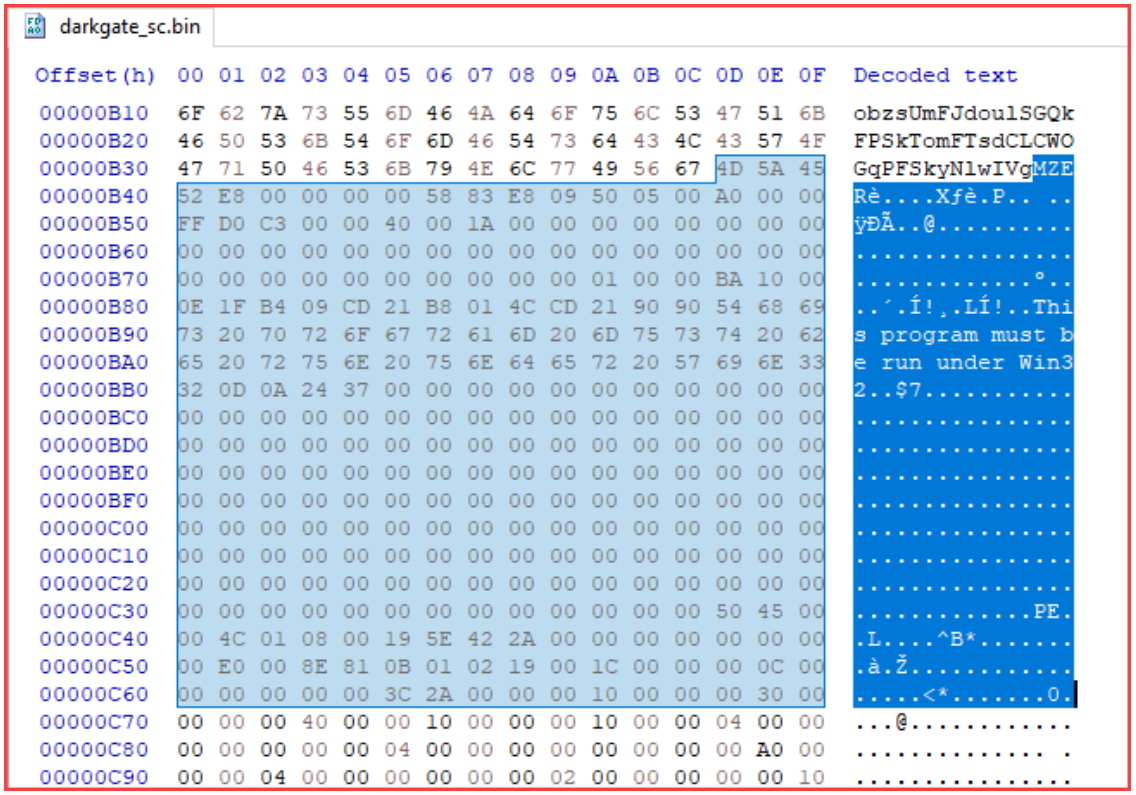


Utilizing **CyberChef**, the shellcode was successfully dumped as follows:

```
recipe=Regular_expression('User%20defined','%5C%5C$bzxrgjfo.%2B%3D%20(.%2B)',true,true,false,false,f
```



This shellcode will perform the task of loading and executing a PE file in memory:



This shellcode's complete loader code is as follows:

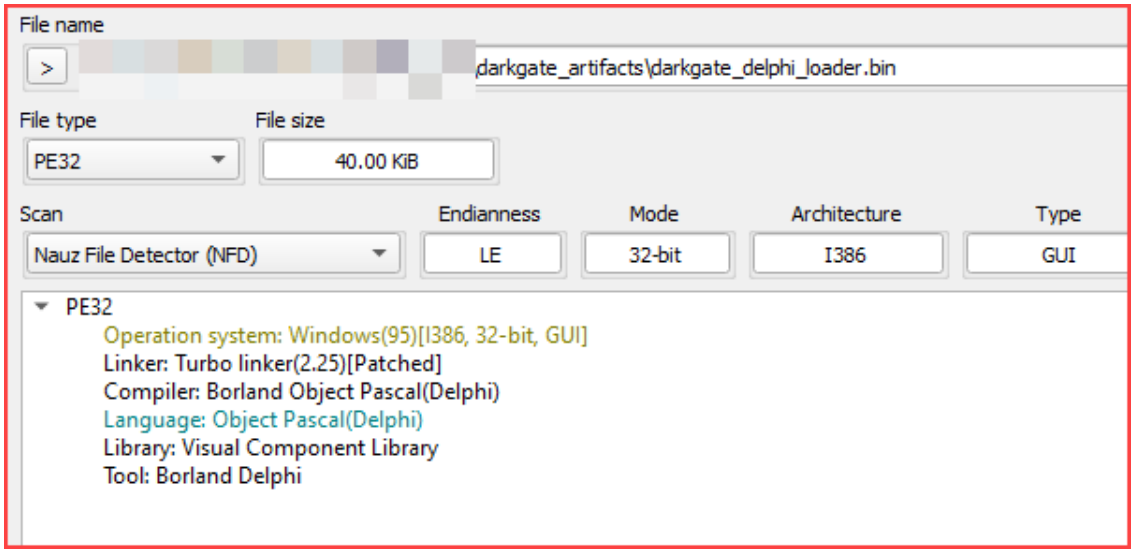
```
unsigned int __stdcall drg_shellcode_loader(PE_BASE arg_dgDelphiLoaderBaseAddr)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
```

```
if ( !drg_resolve_LoadLibraryA_GetProcAddress(&mw_resolved_api) )
{
  return -1u;
}
if ( arg_dgDelphiLoaderBaseAddr.dos_hdr->e_magic != IMAGE_DOS_SIGNATURE )
{
  return -2u;
}
pNtHeaders = (arg_dgDelphiLoaderBaseAddr.baseAddress + arg_dgDelphiLoaderBaseAddr.dos_hdr->e_lfanew
if ( pNtHeaders->Signature != IMAGE_NT_SIGNATURE )
{
  return -2u;
}
v3 = arg_dgDelphiLoaderBaseAddr.dos_hdr->e_res[2];
switch ( v3 )
{
  case 2:
    return 0x4DF;
  case 3:
    if ( (pNtHeaders->FileHeader.Characteristics & IMAGE_FILE_DLL) == 0 )
    {
      return 0x4DF;
    }
    result = ((arg_dgDelphiLoaderBaseAddr.baseAddress + pNtHeaders->OptionalHeader.AddressOfEntryP
               arg_dgDelphiLoaderBaseAddr.baseAddress,
               0,
               0);
    if ( result )
    {
      LOBYTE(arg_dgDelphiLoaderBaseAddr.dos_hdr->e_res[2]) = 2;
    }
    return result;
  case 0:
    if ( !pNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress
    {
      return -3u;
    }
    if ( !drg_perform_base_relocate(
            &pNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddr
            arg_dgDelphiLoaderBaseAddr,
            pNtHeaders->OptionalHeader.ImageBase) )
    {
      return -4u;
    }
    if ( pNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress
      && !drg_build_import_table(
            mw_resolved_api.LoadLibraryA,
```

```
              mw_resolved_api.GetProcAddress,
              pNtHeaders->OptionalHeader.DataDirectory[IMAGE_FILE_IMPORT_DIRECTORY].VirtualAddress,
              pNtHeaders->OptionalHeader.DataDirectory[IMAGE_FILE_IMPORT_DIRECTORY].Size,
              arg_dgDelphiLoaderBaseAddr) )
      {
        return -5u;
      }
      if ( pNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress )
      {
        drg_exec_tls_callbacks(
          &pNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS],
          arg_dgDelphiLoaderBaseAddr);
      }
      break;
  }
  LOBYTE(arg_dgDelphiLoaderBaseAddr.dos_hdr->e_res[2]) = 1;
  mw_EntryPointAddr = (arg_dgDelphiLoaderBaseAddr.baseAddress + pNtHeaders->OptionalHeader.AddressOf
  LOBYTE(arg_dgDelphiLoaderBaseAddr.dos_hdr->e_res[2]) = 2;
  if ( (pNtHeaders->FileHeader.Characteristics & IMAGE_FILE_DLL) == 0 )
  {
    return mw_EntryPointAddr();
  }
  result = (mw_EntryPointAddr)(arg_dgDelphiLoaderBaseAddr.baseAddress, 1, 0);
  if ( result )
  {
    LOBYTE(arg_dgDelphiLoaderBaseAddr.dos_hdr->e_res[2]) = 3;
  }
  return result;
}
```

3.9. Analyze DarkGate Delphi loader

As analyzed above, the shellcode loader’s function is to load and execute a PE file in memory. For easier analysis, let’s extract this PE. This PE is coded in Delphi:



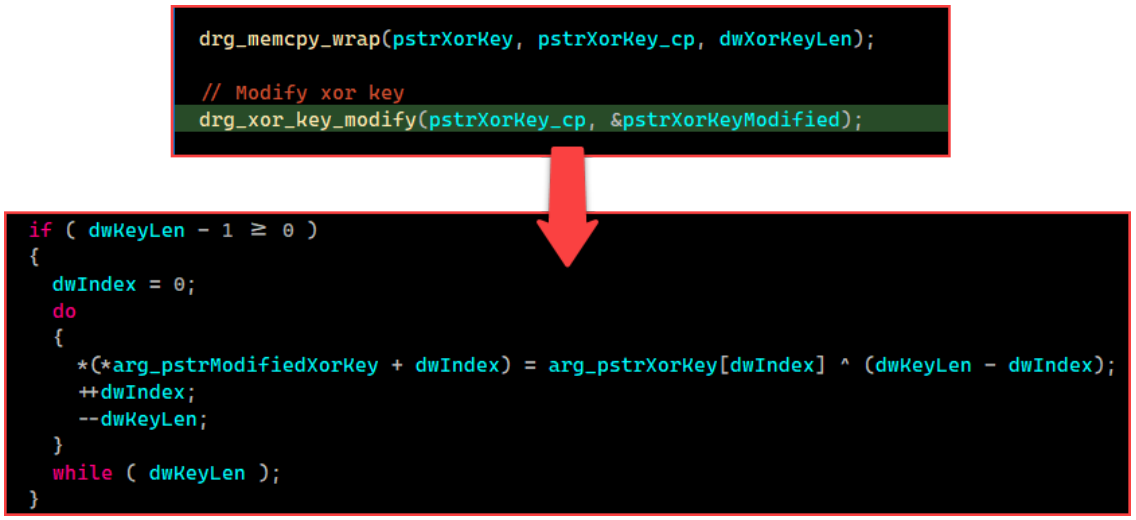
The primary function of this loader is to decrypt the final payload, which is the DarkGate payload, from the AutoIt script, map it into memory, and execute it. To accomplish this task, it first assigns “ `VzXLKSZE` ” to the global variable. This string is then used as a marker to retrieve the encrypted final payload and is subsequently modified for use as an XOR key for decrypting the final payload. Next, it reads the contents of the AutoIt script is `script.a3x` into a buffer.

```
// assign original xor key to global var; this key will be used as marker
// and then modified when peform decrypting final payload
drg_memcpy(&g_pstrVzXLKSZE, "VzXLKSZE");

// returns the path of AutoIt script (script.a3x).
System::ParamStr(1, &g_pstrAutoItScriptPath);
if ( !g_pstrAutoItScriptPath )
{
  MessageBoxA(0, "x", Caption, 0);
}

// read the content of script.a3x to buffer
drg_read_file_content_wrap(g_pstrAutoItScriptPath, &pAutoItContent);
drg_memcpy(&g_pTmpVar, pAutoItContent);
if ( !g_pTmpVar )
{
  Sysutils::ExtractFileName(g_pstrAutoItScriptPath, (int)&v8);
  drg_memcpy(&g_pstrAutoItScriptPath, v8);
  drg_read_file_content_wrap(g_pstrAutoItScriptPath, &result);
  drg_memcpy(&g_pTmpVar, result);
}
if ( !g_pTmpVar )
{
  MessageBoxA(0, "n", Caption, 0);
}
```

Next, utilizing the aforementioned marker string, extract the encrypted payload from the data of the AutoIt script, decrypt the DarkGate payload, and execute the decrypted payload.

```
// retrieve pointer to encrypted payload based on marker
drg_find_and_copy_data_based_on_marker(g_pTmpVar, g_pstrVzXLKSZE, &pDataArray);

// Modify xor key
// Xor with modified key to get the final payload
drg_memcpy(&g_pTmpVar, pDataArray->pbEncPayload);
drg_xor_crypt(g_pTmpVar, g_pstrVzXLKSZE, &pbDarkGatePayload);
drg_memcpy(&g_pTmpVar, pbDarkGatePayload);

// Exec final payload (DarkGate malware)
drg_mapping_and_exec_final_payload(g_pTmpVar);
```

The decryption process of the DarkGate payload is carried out through the `drg_xor_crypt` function. In this function, the initial string " `VzXLKSZE` " is converted into a new string using the `drg_xor_key_modify` function.



Rewrite the above pseudo-code in Python as follows:

```
# modify marker
l = len(marker)
mod_key = ''
for c in marker:
    k = c ^ l
    l -= 1
    mod_key += chr(k)
```

After obtaining the new key, use this key to decrypt the DarkGate payload:

```
dwSrcDataLen = System::_linkproc__ DynArrayLength(arg_pSrcData);
System::_linkproc__ LStrSetLength(ppDecData, dwSrcDataLen);
idx1 = 0;

// xor with modified key to get the final payload
length = drg_get_length(pSrcData_cp);
if ( length ≥ 0 )
{
  dwDecDataLen = length + 1;
  idx2 = 0;
  do
  {
    pDecData = drg_get_pointer_to_buffer(ppDecData);
    pDecData[idx2] = pstrXorKeyModified[idx1] ^ pSrcData_cp[idx2];
    dwModifiedXorKeyLen = System::_linkproc__ DynArrayLength(pstrXorKeyModified);
    idx1 = (idx1 + pstrXorKeyModified[idx1]) % dwModifiedXorKeyLen;
    if ( !idx1 )
    {
      idx1 = System::_linkproc__ DynArrayLength(pstrXorKeyModified) - 1;
    }
    ++idx2;
    --dwDecDataLen;
  }
  while ( dwDecDataLen );
}
```

The above pseudo-code is rewritten as a function as follows:

```
def xor_crypt(enc_data, key):
    dec_bytes = list()
    idx = 0
    key_len = len(key)
    for i in range(len(enc_data)):
        dec_bytes.append(((enc_data[i] ^ ord(key[idx]))) & 0xFF)
        idx = (idx + ord(key[idx])) % key_len
        if idx == 0:
            idx = key_len -1

    dec_data = ''.join(chr(c) for c in dec_bytes).encode('latin-1')
    return dec_data
```

After decoding the DarkGate payload, the loader calls the function `drg_mapping_and_exec_final_payload` to execute this payload directly from memory. The entire code of the `drg_mapping_and_exec_final_payload` function is as follows:

```
void __fastcall drg_mapping_and_exec_final_payload(char *arg_pFinalPayload)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  drg_perform_refCnt(arg_pFinalPayload);
  v10[2] = &savedregs;
  v10[1] = System::_16747;
  v10[0] = NtCurrentTeb()->NtTib.ExceptionList;
```

```
__writefsdword(0, v10);
pFinalPayload = drg_get_pointer_to_buffer(&arg_pFinalPayload);

// copy dos header
qmemcpy(&pPayloadDosHdr, pFinalPayload, sizeof(pPayloadDosHdr));

// copy nt headers
qmemcpy(&pPayloadNtHdr, &pFinalPayload[pPayloadDosHdr.e_lfanew], sizeof(pPayloadNtHdr));
dwPayloadSize = System::__linkproc__ DynArrayLength(arg_pFinalPayload);
pMappedPayload = VirtualAlloc(0, 8 * dwPayloadSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

// copy section data
dwNumberOfSections = pPayloadNtHdr.FileHeader.NumberOfSections;
dwCount = 0;
while ( 1 )
{
  qmemcpy(
    &pPayloadSectionHdr,
    &pFinalPayload[0x28 * dwCount + 0xF8 + pPayloadDosHdr.e_lfanew],
    sizeof(pPayloadSectionHdr));
  if ( pPayloadSectionHdr.SizeOfRawData )
  {
    drg_qmemcpy_wrap(
      &pMappedPayload[pPayloadSectionHdr.VirtualAddress],
      &pFinalPayload[pPayloadSectionHdr.PointerToRawData],
      pPayloadSectionHdr.SizeOfRawData);
  }
  ++dwCount;

  // build import table
  if ( !--dwNumberOfSections )
  {
    for ( import_desc = &pMappedPayload[pPayloadNtHdr.OptionalHeader.DataDirectory[IMAGE_DIRECTORY
          ;
          ++import_desc )
    {
      dwNameRva = import_desc->Name;
      if ( !dwNameRva )
      {
        break;
      }
      dll_handle = LoadLibraryA(&pMappedPayload[dwNameRva]);
      if ( dll_handle != INVALID_HANDLE_VALUE )
      {
        if ( import_desc->anonymous_0.OriginalFirstThunk )
        {
          thunkRef = &pMappedPayload[import_desc->anonymous_0.OriginalFirstThunk];
```

```
      }
      else
      {
        thunkRef = &pMappedPayload[import_desc->FirstThunk];
      }
      for ( funcRef = &pMappedPayload[import_desc->FirstThunk]; ; ++funcRef )
      {
        thunkData = *thunkRef;
        if ( !*thunkRef )
        {
          break;
        }
        if ( thunkData >= 0 )
        {
          apiAddr = GetProcAddress(dll_handle, &thunkData->Name[pMappedPayload]);
        }
        else
        {
          apiAddr = GetProcAddress(dll_handle, *thunkRef);
        }
        *funcRef = apiAddr;
        ++thunkRef;
      }
    }
  }

  // perform base relocation
  pBase = &pMappedPayload[pPayloadNtHdr.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASER
  for ( relocation = &pMappedPayload[pPayloadNtHdr.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_
        relocation - pBase < pPayloadNtHdr.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BA
        relocation = (relocation + relocation->SizeOfBlock) )
  {
    v11 = relocation->SizeOfBlock - IMAGE_SIZEOF_BASE_RELOCATION;
    dwRelocationCount = System::__linkproc__ TRUNC(v11 / 2.0);
    v15 = relocation + 1;
    do
    {
      if ( LOWORD(v15->VirtualAddress) >> 0xC == IMAGE_REL_BASED_HIGHLOW )
      {
        relocEntry = &pMappedPayload[relocation->VirtualAddress + (v15->VirtualAddress & 0xFFF)]
        *relocEntry += &pMappedPayload[-pPayloadNtHdr.OptionalHeader.ImageBase];
      }
      v15 = (v15 + 2);
      --dwRelocationCount;
    }
    while ( dwRelocationCount );
  }
```

```
      // Execute final DarkGate payload
      __asm { jmp     eax }
    }
  }
}
```

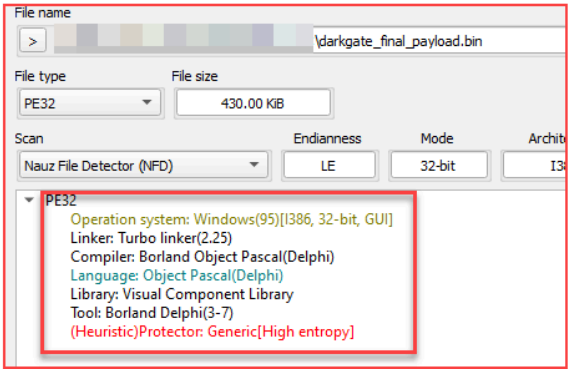
Based on everything analyzed above, we can completely rewrite the script to extract DarkGate payload from AutoIt script as follows:

```
from argparse import ArgumentParser

def xor_crypt(enc_data, key):
    dec_bytes = list()
    idx = 0
    key_len = len(key)
    for i in range(len(enc_data)):
        dec_bytes.append(((enc_data[i] ^ ord(key[idx]))) & 0xFF)
        idx = (idx + ord(key[idx])) % key_len
        if idx == 0:
            idx = key_len -1

    dec_data = ''.join(chr(c) for c in dec_bytes).encode('latin-1')
    return dec_data

def get_payload(enc_file, marker_file, out_file):
    marker = open(marker_file, 'rb').read()
    enc_data = open(enc_file, 'rb').read()

    # modify marker
    l = len(marker)
    mod_key = ''
    for c in marker:
        k = c ^ l
        l -= 1
        mod_key += chr(k)

    blobs = enc_data.split(marker)

    darkgate_enc_final_payload = blobs[1]
    darkgate_payload = xor_crypt(darkgate_enc_final_payload, mod_key)

    open(out_file, 'wb').write(darkgate_payload)
    print("\nSaved to {}".format(out_file))

def main():
    # Add arguments
```

```
    parser = ArgumentParser(description="Get final DarkGate payload!")
    parser.add_argument("-i", "--input_file", dest='input_file', metavar='INPUT_FILE', type=str, req
    parser.add_argument("-m", "--marker_file", dest='marker_file', metavar='MARKER_FILE', type=str,
    parser.add_argument("-o", "--output_file", dest='output_file', metavar='OUTPUT_FILE', type=str,

    # Parse arguments
    args = parser.parse_args()
    get_payload(args.input_file, args.marker_file, args.output_file)

if __name__ == "__main__":
    main()
```

Here is the final result:

```
λ python extract_final_darkgate_payload_from_autoit.py -h
usage: extract_final_darkgate_payload_from_autoit.py [-h] -i INPUT_FILE -m MARKER_FILE -o OUTPUT_FILE

Get final DarkGate payload!

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT_FILE, --input_file INPUT_FILE
                        Please specify extracted AutoIt binary file!
  -m MARKER_FILE, --marker_file MARKER_FILE
                        Please specify marker file!
  -o OUTPUT_FILE, --output_file OUTPUT_FILE
                        Write decrypted payload to output file

λ python extract_final_darkgate_payload_from_autoit.py -i script.a3x -m marker.bin -o darkgate_final_payload.bin

Saved to darkgate_final_payload.bin
```



3.10. Analyze DarkGate payload

I will not go into detailed analysis of the DarkGate payload but rather focus on extracting the configuration it uses. The payload I am analyzing is version `6.1.9` of DarkGate:

```
CODE:0045CB33        mov     eax, ds:g_strDarkGateVersion ; arg_pstrDest
CODE:0045CB38        mov     edx, offset str_619     ; "6.1.9"
CODE:0045CB3D        call    System::_linkproc__ LStrAsg(void *,void *)
```

From the `main` function, it will call the `drg_decode_configuration (0x0042FB0C)` to perform configuration decoding. In this function, it reads the encrypted config stored at address `0045E524` in section **.DATA** and calls the function `drg_decrypt_config (0x00432534)` to perform the main decryption task with the initial key " `ckcilIcconnh` ":

Similar to the analysis with the loader, before performing decryption, the initial key will be changed using the `drg_xor_key_modify (0x004324E4)` :

```
if ( dwKeyLen - 1 < 0 )
{
  return dwIndex;
}
dwIndex = 0;
do
{
  (*arg_pstrModifiedXorKey)[dwIndex] = arg_pstrXorKey[dwIndex] ^ (dwKeyLen - dwIndex);
  ++dwIndex;
  --dwKeyLen;
}
while ( dwKeyLen );
```

Then, use the modified key above to perform configuration decryption:

```
// Modify xor key
drg_xor_key_modify(pstrXorKey_cp, &pstrModifiedXorKey);
v12 = System::_linkproc__ DynArrayLength(arg_pbEncConfig);
System::_linkproc__ LStrSetLength(arg_pbDecConfig, v12);
dwIndex1 = 0;
dwDecConfigLen = drg_get_length(pbEncConfig_cp);

// decrypt data
if ( dwDecConfigLen >= 0 )
{
  dwEncConfigLen = dwDecConfigLen + 1;
  dwIndex2 = 0;
  do
  {
    pbDecConfig = System::UniqueStringA(arg_pbDecConfig);
    pbDecConfig[dwIndex2] = pstrModifiedXorKey[dwIndex1] ^ pbEncConfig_cp[dwIndex2];
    dwModifiedXorKeyLen = System::_linkproc__ DynArrayLength(pstrModifiedXorKey);
    dwIndex1 = (dwIndex1 + pstrModifiedXorKey[dwIndex1]) % dwModifiedXorKeyLen;
    if ( !dwIndex1 )
    {
      dwIndex1 = System::_linkproc__ DynArrayLength(pstrModifiedXorKey) - 1;
    }
    ++dwIndex2;
    --dwEncConfigLen;
  }
  while ( dwEncConfigLen );
}
```

Thus, based on the code written for loader analysis, we can use it to decode the configuration of this campaign as follows:

Based on this configuration information, along with payload analysis, we can describe as follows:

```
0=prodomainnameeforappru.com| //c2 domain
8=No // show_fake_error
11=DarkGate //Fake error message caption
12=R0ijS0qCVITtS0e6xeZ //Fake error message. Custom Base64-encoded -> "Hello world!" (charsets: "zLA
13=6
14=Yes
15=443 //c2 port number
1=Yes //setup persistence
3=Yes //anti_vm based on display devices
4=Yes //check_disk
18=50 //min_disk
6=Yes //anti_vm based on display devices
7=No //check_ram
19=7000 //min_ram
5=Yes //check_xeon
21=No
22=Yes
23=No
25=admin888 //campaign ID
26=No // perform process injection using Process Hollowing technique
27=VzXLKSZE //xor_key or marker
28=No
29=1
tabla=(nq]N*0CV3&ReMOtJ}UaD{W Zxb8uldY"SK2T1rspj$oIFfGB=QL65.Hci9wz)Em4ghAy,k7[XvP //test.txt data t
```

4. Refs

- Tips For Analyzing Delphi Binaries in IDA (Danabot)
- Malpedia DarkGate
- My script
