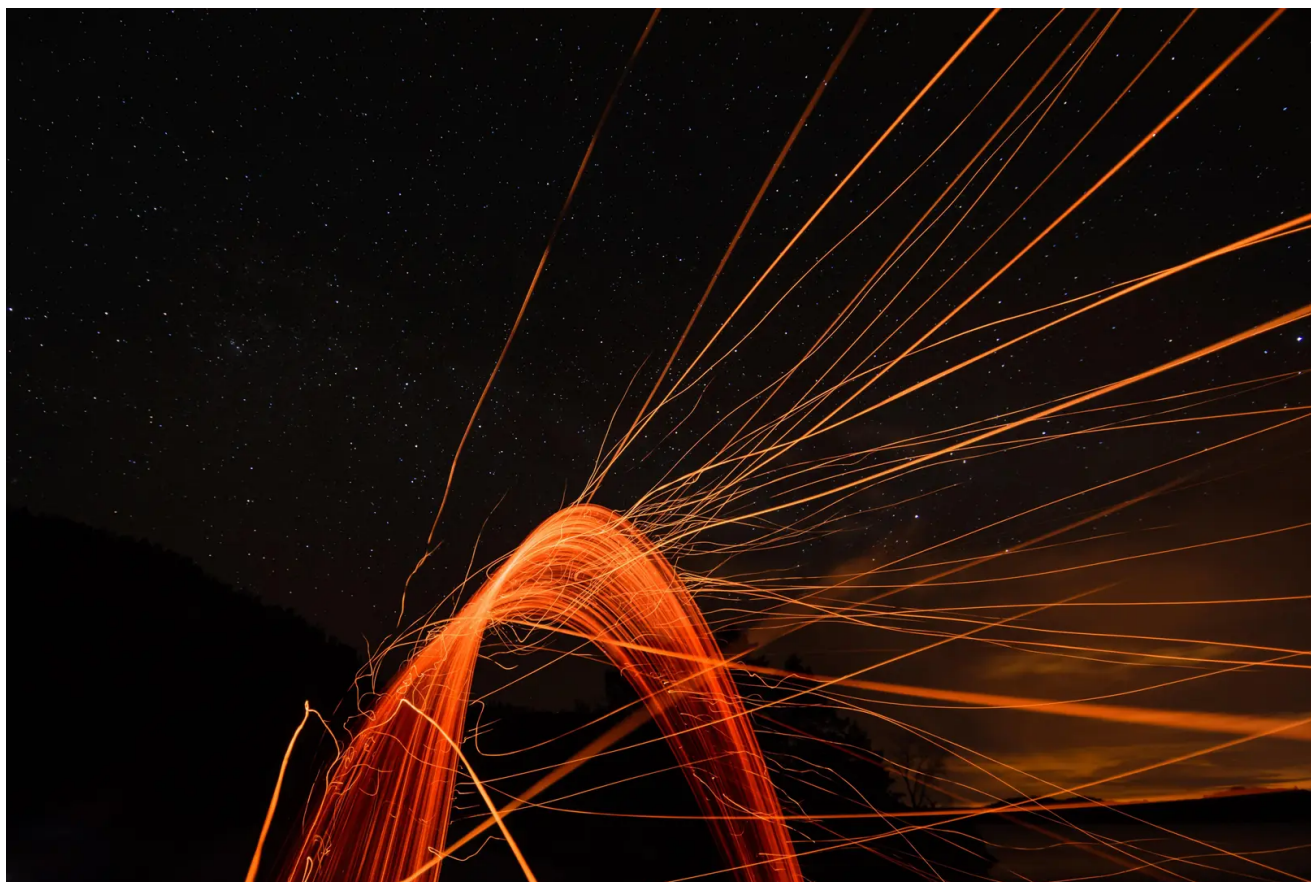


Analysis of CaddyWiper - Wiper Targeting Ukraine

truesec.com/hub/blog/analysis-of-caddywiper-wiper-targeting-ukraine



On the March 14, 2022, security company ESET found a third destructive wiper that has been deployed in Ukraine, called CaddyWiper. It has parts that are created to destroy data quickly and in several ways. ESET published their first initial analysis on [Twitter](#)

Sample analyzed, SHA256:

a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea

Truesec has looked into the wiper to understand its inner workings and find ways to detect the malware.

Malware Execution

According to the Twitter post by ESET the wiper is deployed by group policy to the infected system. Once run, as administrator, the system will crash and the following screen will be displayed.



Your PC ran into a problem and needs to restart. We'll restart for you.



For more information about this issue and possible fixes, visit <https://www.windows.com/stopcode>

If you call a support person, give them this info:
Stop code: CRITICAL_PROCESS_DIED

Once the computer is rebooted it crashes and will not start anymore and prompt that it cannot locate the operating system.

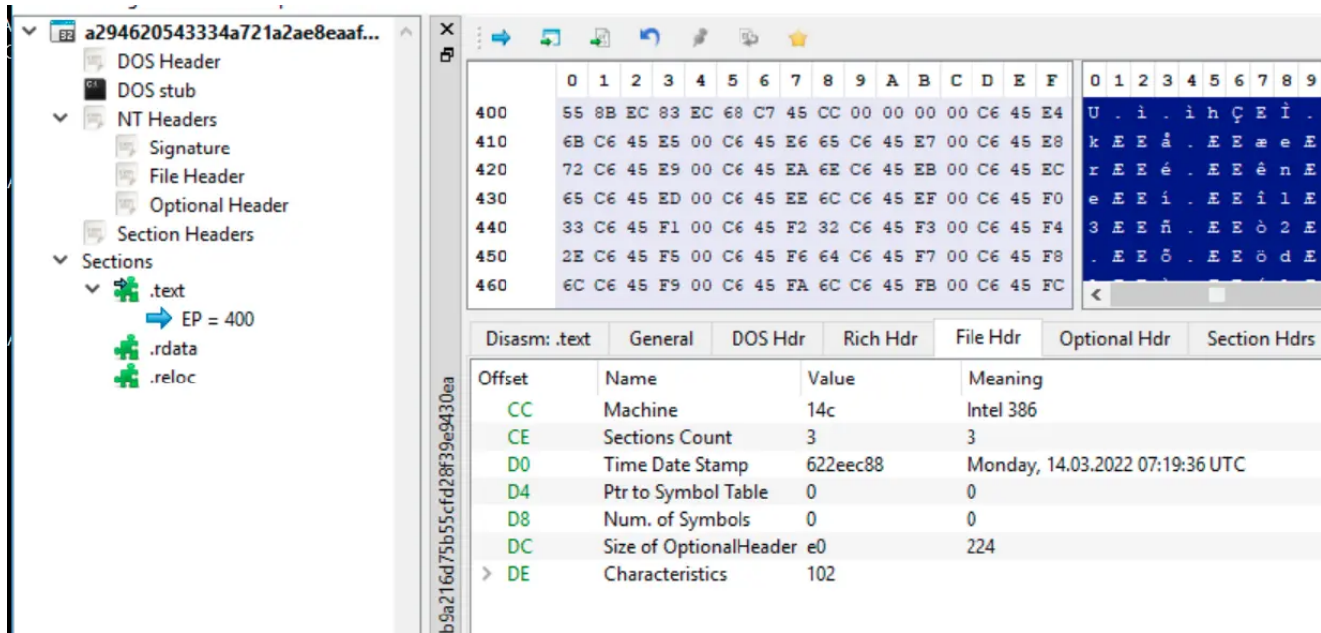
```
Network boot from Intel E1000e
Copyright (C) 2003-2018 VMware, Inc.
Copyright (C) 1997-2000 Intel Corporation

CLIENT MAC ADDR: 00 0C 29 5E BD B5  GUID: 564D8DB3-18B4-56E2-5417-73F3415EBDB5
PXE-E53: No boot filename received

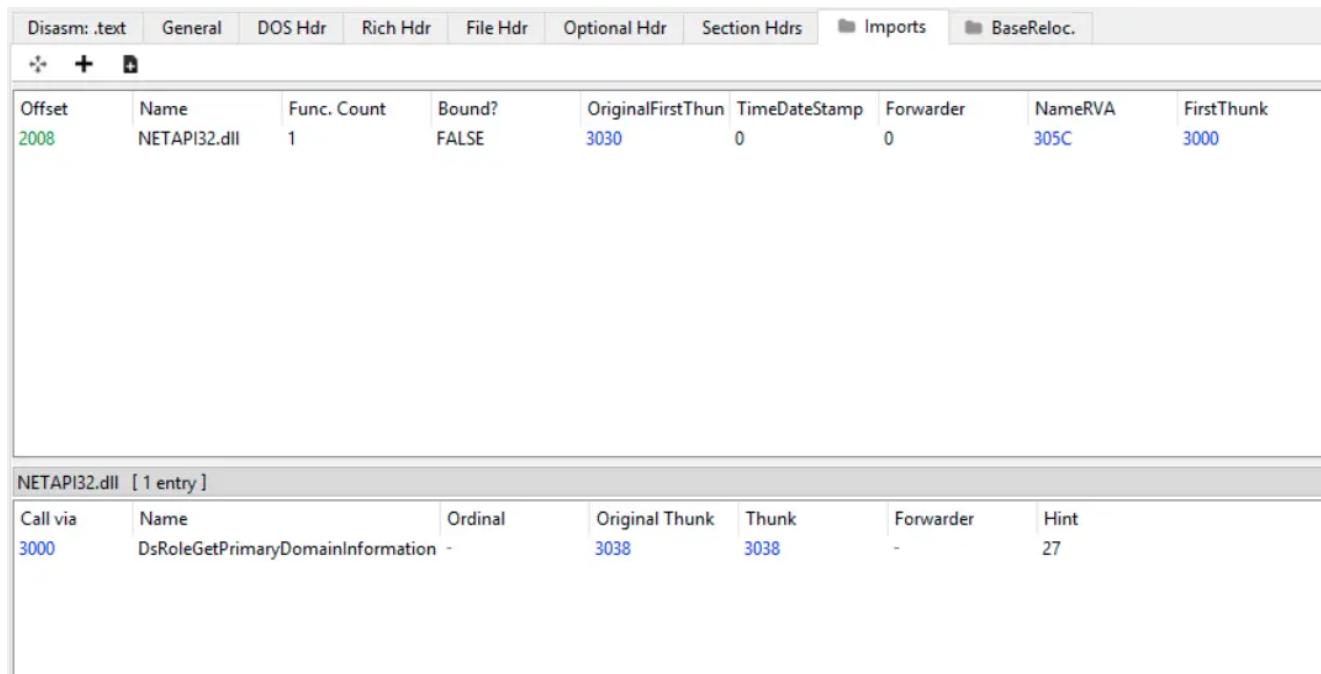
PXE-M0F: Exiting Intel PXE ROM.
Operating System not found
```

Static Analysis

Investigating the time stamp for the sample, it indicates that is compiled on March 14, 2022, showing that it was done just before the attack was conducted.



Looking at the Import Address table, there is only one function called, DsRoleGetPrimaryDomainInformation, indicating that there are more functionalities in the malware that are hidden from static tools.



If the sample is opened in a disassembler, in this case Ghidra, it can be seen that it uses a lot of stack strings for obfuscation.

```

00401000    PUSH    EBP
00401001    MOV     EBP, ESP
00401003    SUB     ESP, 0x68
00401006    MOV     dword ptr [EBP + local_38], 0x0
0040100d    MOV     byte ptr [EBP + local_20], 0x6b
00401011    MOV     byte ptr [EBP + local_1f], 0x0
00401015    MOV     byte ptr [EBP + local_1e], 0x65
00401019    MOV     byte ptr [EBP + local_1d], 0x0
0040101d    MOV     byte ptr [EBP + local_1c], 0x72
00401021    MOV     byte ptr [EBP + local_1b], 0x0
00401025    MOV     byte ptr [EBP + local_1a], 0x6e
00401029    MOV     byte ptr [EBP + local_19], 0x0
0040102d    MOV     byte ptr [EBP + local_18], 0x65
00401031    MOV     byte ptr [EBP + local_17], 0x0
00401035    MOV     byte ptr [EBP + local_16], 0x6c
00401039    MOV     byte ptr [EBP + local_15], 0x0
0040103d    MOV     byte ptr [EBP + local_14], 0x33
00401041    MOV     byte ptr [EBP + local_13], 0x0
00401045    MOV     byte ptr [EBP + local_12], 0x32
00401049    MOV     byte ptr [EBP + local_11], 0x0
0040104d    MOV     byte ptr [EBP + local_10], 0x2e
00401051    MOV     byte ptr [EBP + local_f], 0x0
00401055    MOV     byte ptr [EBP + local_e], 0x64
00401059    MOV     byte ptr [EBP + local_d], 0x0
0040105d    MOV     byte ptr [EBP + local_c], 0x6c
00401061    MOV     byte ptr [EBP + local_b], 0x0
00401065    MOV     byte ptr [EBP + local_a], 0x6c
00401069    MOV     byte ptr [EBP + local_9], 0x0
0040106d    MOV     byte ptr [EBP + local_8], 0x0
00401071    MOV     byte ptr [EBP + local_7], 0x0
00401075    MOV     byte ptr [EBP + local_4c], 0x61

```

To investigate the stack strings, and reveal what they are hiding, first the tool FLOSS was run on the sample that gave the following output.

FLOSS static ASCII strings
!This program cannot be run in DOS mode.
Rich%
.text
`.rdata
@.reloc
DsRoleGetPrimaryDomainInformation
NETAPI32.dll

FLOSS static Unicode strings
jjjjjjjj0040113A
jjjjjj
FLOSS decoded 13 strings
C:\Users\
C:\Users\
FindFirstFileA
kernel32.dll
D:\\
D:*
WriteFile\
advapi32.dll
SetEntriesInAclA
LookupPrivilegeValueA
DeviceIoControl
CreateFileW
Wkernel32.dll
FLOSS extracted 38 stackstrings
C:\Users
netapi32.dll
kernel32.dll
advapi32.dll
CreateFileA
kernel32.dll
FindFirstFileA
OpenProcessToken
CreateFileW
AdjustTokenPrivileges
Wkernel32.dll
FreeSid
SetEntriesInAclA
AllocateAndInitializeSid
LocalFree
SetFilePointer
LookupPrivilegeValueA
LocalAlloc
LoadLibraryA
GetLastError
advapi32.dll
FindClose
kernel32.dll
DeviceIoControl
CloseHandle
CloseHandle
\kernel32.dll
CloseHandle


```
SetTakeOwnershipPrivilege
advapi32.dll
\\.\PHYSICALDRIVE9
kernel32.dll
LocalFree
FindNextFileA
GetFileSize
GetCurrentProcess
WriteFile
SetNamedSecurityInfoA
```

To give context for the stacked strings the tool CAPA was used to find the different locations in the code where stacked strings are used.

```
contain obfuscated stackstrings (8 matches)
namespace anti-analysis/obfuscation/string/stackstring
scope basic block
matches 0x401000
0x40114A
0x4011D0
0x401750
0x401A10
0x402025
0x40215E
0x4022A0
```

To get an overview of the intent of each function in relation to where the different stack strings are used for obfuscation, API calls and libraries are mapped to every function that CAPA found in the sample.

```
0x401000 kernel32.dll, advapi32.dll, LoadLibraryA, netapi32.dll
0x40114A netapi32.dll, netapi32.dll
0x4011D0 DeviceIoControl, kernel32.dll, CreateFileW, CloseHandle, \\.\PHYSICALDRIVE9
0x401750 advapi32.dll, LookupPrivilegeValueA, AdjustTokenPrivileges, GetLastErrorc
0x401A10 advapi32.dll, SetEntriesInAclA, AllocateAndInitializeSid,
SetNamedSecurityInfoA, kernel32.dll, GetCurrentProcess, OpenProcessToken
0x402025 SetTakeOwnershipPrivilege, FreeSid, LocalFree, CloseHandle
0x40215E FreeSid, LocalFree, CloseHandle
0x4022A0 FindFirstFileA, kernel32.dll, FindNextFileA, CreateFileA, GetFileSize,
LocalAlloc, SetFilePointer, WriteFile, LocalFree, CloseHandle, FindClose
```

Execution Flow

Upon start the wiper uses the API call DsRoleGetPrimaryDomainInformation to check if the computer is the primary domain controller by comparing to the hard coded value 0x5, that comes from the struct DSROLE_MACHINE_ROLE. If it is the primary domain controller it will exit. This is probably done because the threat actor is using the domain controller as the source of distribution of the wiper and not to ruin its own foothold.

```
00401135    PUSH    EAX
00401136    PUSH    0x1
00401138    PUSH    0x0
0040113a    CALL    dword ptr [->NETAPI32.DLL::DsRoleGetPrimaryDomainInformation]
00401140    MOV     ECX, dword ptr [EBP + local_3c]
00401143    CMP    dword ptr [ECX], 0x5
00401146    JNZ    LAB_0040114a
00401148    JMP    LAB_004011c4
```

The next part of the wiper is the file destruction part. It calls the function 0x4022A0 that iterates over the files, using the API calls that are resolved from the stack strings, and writes over the first 0xA00000 bytes with zeros.

```

004029aa    CMP     dword ptr [EBP + local_e60], 0x0
004029b1    JNC     LAB_004029b8
004029b3    JMP     LAB_00402a51

                                LAB_004029b8                                XREF[1]: 004029b1(j)
004029b8    CMP     dword ptr [EBP + local_e60], 0xa00000
004029c2    JBE     LAB_004029ce
004029c4    MOV     dword ptr [EBP + local_e60], 0xa00000

```

Untitled (C:) paddedFile.bin

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
009FFE70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFE80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFE90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFEA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFEB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFEC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFED0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFEE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFEF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFF90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFFA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFFB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFFC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFFD0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFFE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
009FFFF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00A00000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY
00A00010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY
00A00020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY
00A00030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY
00A00040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY
00A00050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY
00A00060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY
00A00070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	YYYYYYYYYYYYYYYY

Then the wiper loops through the alphabet (0x18), starting with D all the way up to Z and then one additional iteration, and applies the data destruction from the function in 0x4022A0 to the files in every partition it finds.


```

00401198 JMP LAB_004011a3
LAB_0040119a XREF[1]: 004011bd(j)
0040119a MOV ECX, dword ptr [EBP + local_6c]
0040119d ADD ECX, 0x1
004011a0 MOV dword ptr [EBP + local_6c], ECX
LAB_004011a3 XREF[1]: 00401198(j)
004011a3 CMP dword ptr [EBP + local_6c], 0x18
004011a7 JNC LAB_004011bf
004011a9 LEA EDX=>local_24, [EBP + -0x20]
004011ac PUSH EDX
004011ad CALL FUN_004022a0 ; undefined FUN_004022a0(char * param_1)
004011b2 ADD ESP, 0x4
004011b5 MOV AL, byte ptr [EBP + local_24]
004011b8 ADD AL, 0x1
004011ba MOV byte ptr [EBP + local_24], AL
004011bd JMP LAB_0040119a

```

This is the last iteration and has gone past Z to Z+1.

00401198	EB 09	jmp a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd2
0040119A	8B4D 98	mov ecx,dword ptr ss:[ebp-68]
0040119D	83C1 01	add ecx,1
004011A0	894D 98	mov dword ptr ss:[ebp-68],ecx
004011A3	837D 98 18	cmp dword ptr ss:[ebp-68],18
004011A7	73 16	jae a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd
004011A9	8D55 E0	lea edx,dword ptr ss:[ebp-20]
004011AC	52	push edx
004011AD	E8 EE100000	call <a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd>
004011B2	83C4 04	add esp,4
004011B5	8A45 E0	mov al,byte ptr ss:[ebp-20]
004011B8	04 01	add al,1
004011BA	8845 E0	mov byte ptr ss:[ebp-20],al
004011BD	EB DB	jmp a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd
004011BF	E8 0C000000	call <a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd>

Lastly the wiper loops through a list of open raw access to \\.\PHYSICALDRIVE9 - \\.\PHYSICALDRIVE0 and writing to it using IOCTL_DISK_SET_DRIVE_LAYOUT_EX (0x7c054) by using the API DeviceIoControl. By doing so it erases the Master Boot Record.

00401433	8D4D 98	lea ecx,dword ptr ss:[ebp-68]	
00401426	898D F47FFFFF	mov dword ptr ss:[ebp-80C],ecx	
0040142C	6A 00	push 0	
0040142E	68 80000000	push 80	
00401433	6A 03	push 3	
00401435	6A 00	push 0	
00401437	6A 03	push 3	
00401439	68 000000C0	push C0000000	
0040143E	8B95 F47FFFFF	mov edx,dword ptr ss:[ebp-80C]	
00401444	52	push edx	
00401445	FF95 FCF7FFFF	call dword ptr ss:[ebp-804]	
00401448	8945 FC	mov dword ptr ss:[ebp-4],eax	
0040144E	837D FC FF	cmp dword ptr ss:[ebp-4],FFFFFFFF	
00401452	74 2C	jne a294620543334a721a2ae8eaf9680a0786f4b9	
00401454	6A 00	push 0	
00401456	8D85 F8F7FFFF	lea eax,dword ptr ss:[ebp-808]	
0040145C	50	push eax	
0040145D	6A 00	push 0	
0040145F	6A 00	push 0	
00401461	68 80070000	push 780	
00401466	8D8D 10F8FFFF	lea ecx,dword ptr ss:[ebp-7F0]	
0040146C	51	push ecx	
0040146D	68 54C00700	push 7C054	
00401472	8B55 FC	mov edx,dword ptr ss:[ebp-4]	
00401475	52	push edx	
00401476	FF55 94	call dword ptr ss:[ebp-6C]	
00401479	8B45 FC	mov eax,dword ptr ss:[ebp-4]	
0040147C	50	push eax	
0040147D	FF55 F8	call dword ptr ss:[ebp-8]	
00401480	8A4D BA	mov cl,byte ptr ss:[ebp-46]	
00401483	8A4D BA	mov byte ptr ss:[ebp-46],cl	
00401486	8A55 BA	mov d1,byte ptr ss:[ebp-46]	
00401489	80EA 01	sub d1,1	
0040148C	8855 BA	mov byte ptr ss:[ebp-46],d1	
0040148F	8B85 0CF8FFFF	mov eax,dword ptr ss:[ebp-7F4]	
00401495	8B8D 0CF8FFFF	mov ecx,dword ptr ss:[ebp-7F4]	
00401498	83E9 01	sub ecx,1	
0040149E	898D 0CF8FFFF	mov dword ptr ss:[ebp-7F4],ecx	
004014A4	85C0	test eax,eax	
004014A6	0F85 77FFFFFF	jne a294620543334a721a2ae8eaf9680a0786f4b	

Detection

Since the wiper is using stack strings for obfuscation of the part that interacts with the disk, that part can be used as Yara rule for detection.

```
rule caddy_wiper {
  meta:
    description = "Search for caddy wiper"
    author = "Truesec"
    reference = "truesec.se"
    date = "2022-03-14"
    hash1 = "a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea"
    strings:
      $x1 = {c6 45 ?? 5c c6 45 ?? 00 c6 45 ?? 5c c6 45 ?? 00 c6 45 ?? 2e c6 45 ?? 00 c6 45
      ?? 5c c6 45 ?? 00 c6 45 ?? 50 c6 45 ?? 00 c6 45 ?? 48 c6 45 ?? 00 c6 45 ?? 59 c6 45
      ?? 00 c6 45 ?? 53 c6 45 ?? 00 c6 45 ?? 49 c6 45 ?? 00 c6 45 ?? 43 c6 45 ?? 00 c6 45
      ?? 41 c6 45 ?? 00 c6 45 ?? 4c c6 45 ?? 00 c6 45 ?? 44 c6 45 ?? 00 c6 45 ?? 52 c6 45
      ?? 00 c6 45 ?? 49 c6 45 ?? 00 c6 45 ?? 56 c6 45 ?? 00 c6 45 ?? 45} //Stack strings
for \\.\\PHYSICALDRIVE
      $x2 = {c6 45 ?? 44 c6 45 ?? 65 c6 45 ?? 76 c6 45 ?? 69 c6 45 ?? 63 c6 45 ?? 65 c6 45
      ?? 49 c6 45 ?? 6f c6 45 ?? 43 c6 45 ?? 6f c6 45 ?? 6e c6 45 ?? 74 c6 45 ?? 72 c6 45
      ?? 6f c6 45 ?? 6c c6 45 ?? 00 c6 45 ?? 6b c6 45 ?? 00 c6 45 ?? 65 c6 45 ?? 00 c6 45
      ?? 72 c6 45 ?? 00 c6 45 ?? 6e c6 45 ?? 00 c6 45 ?? 65 c6 45 ?? 00 c6 45 ?? 6c c6 45
      ?? 00 c6 45 ?? 33 c6 45 ?? 00 c6 45 ?? 32 c6 45 ?? 00 c6 45 ?? 2e c6 45 ?? 00 c6 45
      ?? 64 c6 45 ?? 00 c6 45 ?? 6c c6 45 ?? 00 c6 45 ?? 6c c6 45 ?? 00 c6 45 ?? 00 c6 45
      ?? 00 c6 45 ?? 43 c6 45 ?? 72 c6 45 ?? 65 c6 45 ?? 61 c6 45 ?? 74 c6 45 ?? 65 c6 45
      ?? 46 c6 45 ?? 69 c6 45 ?? 6c c6 45 ?? 65 c6 45 ?? 57} //Stack strings for
DeviceIoControl, kernel32.dll, CreateFileW

      $a1 = {c6 85 ?? fe ff ff 61 c6 85 ?? fe ff ff 00 c6 85 ?? fe ff ff 64 c6 85 ?? fe ff
      ff 00 c6 85 ?? fe ff ff 76 c6 85 ?? fe ff ff 00 c6 85 ?? fe ff ff 61 c6 85 ?? fe ff
      ff 00 c6 85 ?? fe ff ff 70 c6 85 ?? fe ff ff 00 c6 85 ?? fe ff ff 69 c6 85 ?? fe ff
      ff 00 c6 85 ?? fe ff ff 33 c6 85 ?? fe ff ff 00 c6 85 ?? fe ff ff 32 c6 85 ?? fe ff
      ff 00 c6 85 ?? fe ff ff 2e c6 85 ?? fe ff ff 00 c6 85 ?? fe ff ff 64 c6 85 ?? fe ff
      ff 00 c6 85 ?? fe ff ff 6c c6 85 ?? fe ff ff 00 c6 85 ?? fe ff ff 6c c6 85 ?? fe ff
      ff 00 c6 85 ?? fe ff ff 00 c6 85 ?? fe ff ff 00 c6 85 ?? ff ff ff 53 c6 85 ?? ff ff
      ff 65 c6 85 ?? ff ff ff 74 c6 85 ?? ff ff ff 45 c6 85 ?? ff ff ff 6e c6 85 ?? ff ff
      ff 74 c6 85 ?? ff ff ff 72 c6 85 ?? ff ff ff 69 c6 85 ?? ff ff ff 65 c6 85 ?? ff ff
      ff 73 c6 85 ?? ff ff ff 49 c6 85 ?? ff ff ff 6e c6 85 ?? ff ff ff 41 c6 85 ?? ff ff
      ff 63 c6 85 ?? ff ff ff 6c c6 85 ?? ff ff ff 41} //Stack strings for advapi32.dll
SetEntriesInAclA

      $a2 = {c6 85 ?? ff ff ff 41 c6 85 ?? ff ff ff 6c c6 85 ?? ff ff ff 6c c6 85 ?? ff ff
      ff 6f c6 85 ?? ff ff ff 63 c6 85 ?? ff ff ff 61 c6 85 ?? ff ff ff 74 c6 85 ?? ff ff
      ff 65 c6 85 ?? ff ff ff 41 c6 85 ?? ff ff ff 6e c6 85 ?? ff ff ff 64 c6 85 ?? ff ff
      ff 49 c6 85 ?? ff ff ff 6e c6 85 ?? ff ff ff 69 c6 85 ?? ff ff ff 74 c6 85 ?? ff ff
      ff 69 c6 45 ?? 61 c6 45 ?? 6c c6 45 ?? 69 c6 45 ?? 7a c6 45 ?? 65 c6 45 ?? 53 c6 45
      ?? 69 c6 45 ?? 64} //Stack strings for AllocateAndInitializeSid

    condition:
      uint16(0) == 0x5A4D
      and any of ($x*)
      or all of ($a*) and filesize < 50000
}
```