The Curious Case of DeathRansom: Part I

By Minh Tran

Published: 2020-01-02 · Archived: 2026-04-05 15:44:46 UTC

A FortiGuard Labs Threat Analysis Report

Introduction

Ransomware is certainly a significant global threat. According to one recent report, ransomware is estimated to have cost businesses more than $8 billion in 2018, up from just $1 billion in 2016, while this year alone losses for the healthcare industry have already reached $25 billion.

Part of this increase is due to the rise of Ransomware as a Service, with variants such as GandCrab generating as much as $2 billion in revenue for its developers, and our observation in the FortiGuard Labs Threat Landscape Report for Q3 that two additional ransomware families – Sodinokibi and Nemty – have now been deployed as RaaS solutions as well. Another part of the reason for this growth is that cybercriminals continue to develop new ransomware variants. This dramatic escalation of ransomware over the past few years is part of the reason why we here at FortiGuard Labs keep such close track of it.

And recently, our threat radar detected a new ransomware variant that we break down for you in this threat analysis, ominously called DeathRansom. In this analysis, we will first be looking at a version with a SHA256 sample of:

7C2DBAD516D18D2C1C21ECC5792BC232F7B34DADC1BC19E967190D79174131D1

Subsequent samples exhibit slightly different behaviors, as we describe later in our analysis.

```
.rdata:0040DE38                 align 10h
.rdata:0040DE40 ; Debug Directory entries
.rdata:0040DE40                 dd 0                    ; Characteristics
.rdata:0040DE44                 dd 5DCFB52Eh            ; TimeDateStamp: Sat Nov 16 08:37:02 2019
.rdata:0040DE48                 dw 0                    ; MajorVersion
.rdata:0040DE4A                 dw 0                    ; MinorVersion
.rdata:0040DE4C                 dd 0Dh                  ; Type: IMAGE_DEBUG_TYPE_POGO
.rdata:0040DE50                 dd 0D4h                 ; SizeOfData
.rdata:0040DE54                 dd rva aGctl            ; AddressOfRawData
.rdata:0040DE58                 dd 0CE78h               ; PointerToRawData
.rdata:0040DE5C                 dd 0                    ; Characteristics
.rdata:0040DE60                 dd 5DCFB52Eh            ; TimeDateStamp: Sat Nov 16 08:37:02 2019
.rdata:0040DE64                 dw 0                    ; MajorVersion
.rdata:0040DE66                 dw 0                    ; MinorVersion
.rdata:0040DE68                 dd 0Eh                  ; Type: IMAGE_DEBUG_TYPE_ILTCG
.rdata:0040DE6C                 dd 0                    ; SizeOfData
.rdata:0040DE70                 dd 0                    ; AddressOfRawData
.rdata:0040DE74                 dd 0                    ; PointerToRawData
```

Figure 1. TimeDateStamp of the sample

As you can see, the **TimeDateStamp** of this sample is very new (Sat Nov 16 08:37:02 2019), so naturally, we decided to dig deeper and learn more.

The Workflow of the DeathRansom Ransomware

At a high level, this ransomware follows a sensible design: it scans and encrypts files on local and network drives.

```
124   SHEmptyRecycleBinA(0, 0, 1u);
125   mtr_processNetwork(&v28, 0);                  // Process Network Drives
126   if ( GetLogicalDriveStringsW(0x7FFFu, &Buffer) )// Logical Drives
127   {
128     v17 = &Buffer;
129     if ( Buffer )
130     {
131       do
132       {
133         v18 = GetProcessHeap();
134         v19 = (WCHAR *)HeapAlloc(v18, 8u, 0xFFFEu);
135         v20 = v19;
136         if ( v19 )
137         {
138           wnsprintfW(v19, 0x7FFF, (const char *)L"\\\\?\\%c:", *v17);// Drives:     C: D: etc
139           mtr_processDir(&v28, v20);
140           v21 = GetProcessHeap();
141           HeapFree(v21, 0, v20);
142         }
```

Figure 2. start()

It's clear that right off the bat (in **start()**), that DeathRansom gets right to the business of being a ransomware: by enumerating and encrypting files. To enumerate network resources, the malware uses standard Windows APIs (**WNetOpenEnumW**, **WNetEnumResourceW** etc.)

Imports | Functions window | Strings window

```
1  DWORD __thiscall mtr_processNetwork(void *this, LPNETRESOURCEW lpNetResource)
2  {
3    void *v2; // ebx
4    DWORD result; // eax
5    SIZE_T v4; // ST10_4
6    HANDLE v5; // eax
7    struct _NETRESOURCEW *v6; // eax
8    LPNETRESOURCEW v7; // esi
9    DWORD v8; // edi
10   LPCWSTR *targetDir; // esi
11   HANDLE v10; // eax
12   HANDLE hEnum; // [esp+10h] [ebp-Ch]
13   SIZE_T dwBytes; // [esp+14h] [ebp-8h]
14   DWORD cCount; // [esp+18h] [ebp-4h]
15   LPNETRESOURCEW lpNetResourcea; // [esp+24h] [ebp+8h]
16
17   cCount = -1;
18   v2 = this;
19   dwBytes = 0x4000;
20   result = WNetOpenEnumW(2u, 0, 0x13u, lpNetResource, &hEnum);// RESOURCE_GLOBALNET
21   if ( !result )
22   {
23     v4 = dwBytes;
24     v5 = GetProcessHeap();
25     v6 = (struct _NETRESOURCEW *)HeapAlloc(v5, 8u, v4);
26     v7 = v6;
27     lpNetResourcea = v6;
28     if ( v6 )
29     {
30       if ( !WNetEnumResourceW(hEnum, &cCount, v6, &dwBytes) )
31       {
32         do
33         {
34           v8 = 0;
35           if ( cCount )
36           {
37             targetDir = (LPCWSTR *)&v7->lpRemoteName;
38             do
39             {
40               if ( *(_BYTE *)(targetDir - 2) & 2 )
41                 mtr_processNetwork(v2, (LPNETRESOURCEW)(targetDir - 5));// recursive function
42               else
43                 mtr_processDir(v2, *targetDir); // main
44               ++v8;
45               targetDir += 8;
```

Figure 3. Recursively Scanning Network Resources

Inside this function (**processNetwork**), it recursively scans network resources until it hits a normal directory, at which point it processes it like a directory (**processDir**).

```
28    InterlockedExchangeAdd(&Interlock, 1);
29    QueueUserWorkItem(mtr_renameFiles, v4, 0);  // ThreadPool
30    v5 = GetProcessHeap();
31    v6 = (WCHAR *)HeapAlloc(v5, 8u, 0xFFFEu);
32    newItem = v6;
33    if ( v6 )
34    {
35      v8 = v2;
36      wnsprintfW(v6, 0x7FFF, (const char *)L"%s\\*", v2);
37      hFindFile = FindFirstFileW(newItem, &FindFileData);
38      if ( hFindFile != (HANDLE)-1 )
39      {
40        do
41        {
42          wnsprintfW(newItem, 0x7FFF, (const char *)L"%s\\%s", v8, FindFileData.cFileName);
43          if ( FindFileData.dwFileAttributes & 0x10
44            && lstrcmpW(FindFileData.cFileName, L"..")
45            && lstrcmpW(FindFileData.cFileName, L".") )
46          {
47            v9 = 0;
48            while ( 1 )
49            {
50              v10 = exclusion2[v9];
51              v11 = CharLowerW(FindFileData.cFileName);
52              if ( StrStrW(v11, v10) )
53                break;
54              ++v9;
55              if ( v9 >= 6 )
56              {
57                mtr_processDir(v17, newItem);   // recursive
58                break;
59              }
60            }
61            v8 = targetDir;
62          }
63        }
64        while ( FindNextFileW(hFindFile, &FindFileData) );
65        FindClose(hFindFile);
66      }
```

Figure 4. processDir()

Just like **processNetwork, processDir** is also recursive. The difference is that it needs to perform some sanity checks to make sure that the item is indeed a folder (but not **“.”** or **“..”**), and further, that the item is not excluded (see **exclusion2**, below).

```
.rdata:0040D35B                 db    0
.rdata:0040D35C                 dd offset sub_401790
.rdata:0040D360                 dd offset sub_4017D0
.rdata:0040D364 exclusion1      dd offset aReadMeTxt    ; DATA XREF: mtr_renameFiles:loc_40A753↑r
.rdata:0040D364                                         ; "read_me.txt"
.rdata:0040D368                 dd offset aAutoexecBat  ; "autoexec.bat"
.rdata:0040D36C                 dd offset aDesktopIni   ; "desktop.ini"
.rdata:0040D370                 dd offset aAutorunInf   ; "autorun.inf"
.rdata:0040D374                 dd offset aNtuserDat    ; "ntuser.dat"
.rdata:0040D378                 dd offset aIconcacheDb  ; "iconcache.db"
.rdata:0040D37C                 dd offset aBootsectBak  ; "bootsect.bak"
.rdata:0040D380                 dd offset aBootIni      ; "boot.ini"
.rdata:0040D384                 dd offset aNtuserDatLog ; "ntuser.dat.log"
.rdata:0040D388                 dd offset aThumbsDb     ; "thumbs.db"
.rdata:0040D38C exclusion2      dd offset aProgramdata  ; DATA XREF: mtr_processDir:loc_40A960↑r
.rdata:0040D38C                                         ; "programdata"
.rdata:0040D390                 dd offset aRecycleBin   ; "$recycle.bin"
.rdata:0040D394                 dd offset aProgramFiles ; "program files"
.rdata:0040D398                 dd offset aWindows      ; "windows"
.rdata:0040D39C                 dd offset aAllUsers     ; "all users"
.rdata:0040D3A0                 dd offset aAppdata      ; "appdata"
```

Figure 5. Exclusions

We can see here that the malware author has made a number of reasonable choices, including:

- Excluding important Windows folders (Program Files, Windows, etc) to avoid rendering the system unusable
- When it comes to files, similar checks also occur.
- DeathRansom also avoids “encrypting” the systems files (ntuser.dat, etc)

Certainly, this list is not comprehensive, but it does show that the author does have some skills and knowledge about system programming.

```
27    v3 = GetProcessHeap();
28    v4 = (WCHAR *)HeapAlloc(v3, 8u, 0xFFFEu);
29    fullPath = v4;
30    if ( v4 )
31    {
32      wnsprintfW(v4, 0x7FFF, (const char *)L"%s\\*", path);
33      v6 = FindFirstFileW(fullPath, &FindFileData);
34      v19 = v6;
35      if ( v6 != (HANDLE)-1 )
36      {
37        v7 = v6;
38        do
39        {
40          wnsprintfW(fullPath, 0x7FFF, (const char *)L"%s\\%s", v2, FindFileData.cFileName);
41          if ( !(FindFileData.dwFileAttributes & 0x10) )
42          {
43            v8 = CreateFileW(fullPath, 0xC0000000, 7u, 0, 3u, 0x80u, 0);// OPEN_EXISTING
44            if ( v8 != (HANDLE)-1 && !StrStrW(FindFileData.cFileName, L".wctc") )// not renamed yet
45            {
46              v9 = 0;
47              do
48              {
49                v10 = exclusion1[v9];           // Exclusions
50                v11 = CharLowerW(FindFileData.cFileName);
51                if ( StrStrW(v11, v10) )
52                {
53                  CloseHandle(v8);
54                  goto LABEL_15;
55                }
56                ++v9;
57              }
58              while ( v9 < 10 );
59              v12 = GetProcessHeap();
60              v13 = (WCHAR *)HeapAlloc(v12, 8u, 0xFFFEu);
61              v14 = v13;
62              if ( v13 )
63              {
64                wnsprintfW(v13, 0x7FFF, (const char *)L"%s.wctc", fullPath);// .wctc
65                CloseHandle(v8);
66                MoveFileW(fullPath, v14);       // rename not encrypt file!
67                v15 = GetProcessHeap();
68                HeapFree(v15, 0, v14);
69              }
70              InterlockedExchangeAdd(&dword_40F064, 1);
71              QueueUserWorkItem(Function, v8, 0);
72LABEL_15:
73              v2 = path;
74            }
75            v7 = v19;
76          }
77        }
78        while ( FindNextFileW(v7, &FindFileData) );
79        FindClose(v7);
```

Figure 6. “Encrypting” Files

An astute reader may have noticed that DeathRansom does **not** really encrypt file content. In this case, victims only have to rename the affected files (hint: remove the extension) to restore the system back to normal.

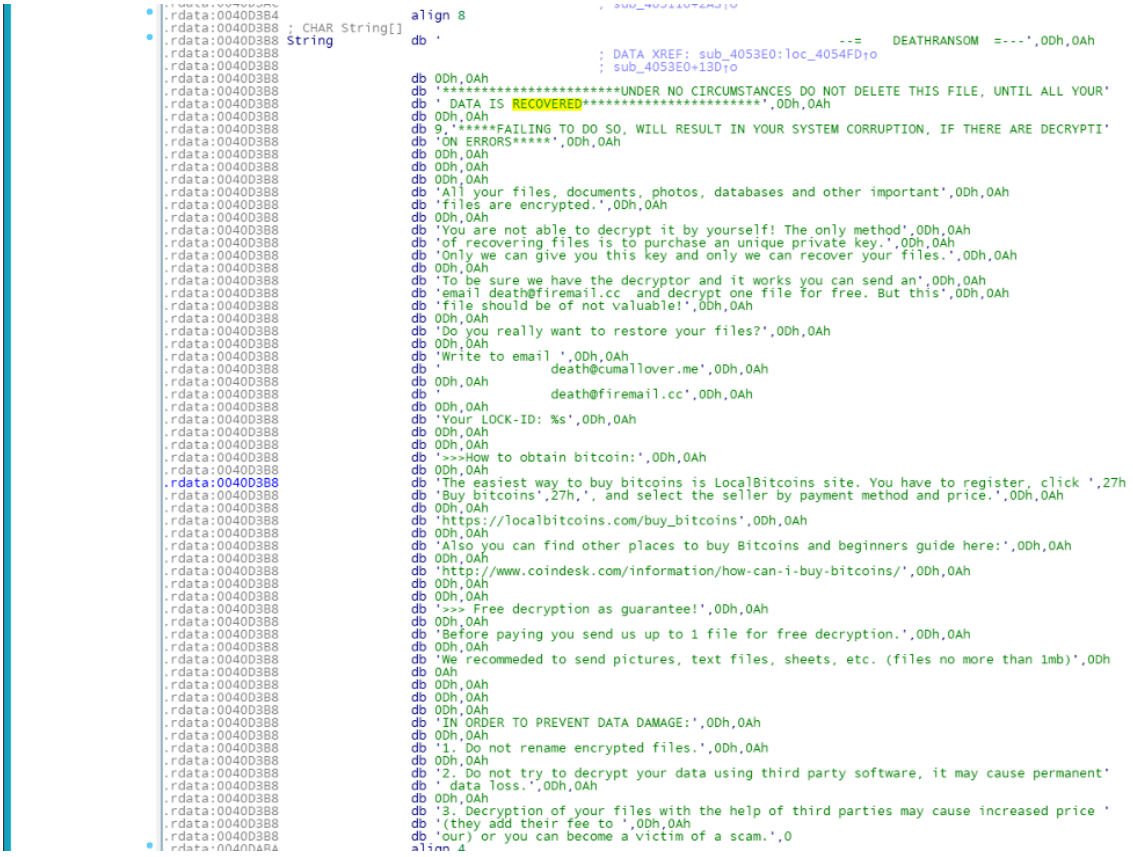

Figure 7. Ransom Note

In spite of this, like almost all ransomware, DeathRansom still displays a ransom note called read_me.txt. In that note one can see clearly that the author calls the malware DEATHRANSOM. The email also informs its victims that they need to contact the culprits at **death@cxxxxxxver.me** and **death@fxxxxxxx.cc**

Other Interesting Technical Details

When the malware launches, but before it begins “encoding” files, it performs some interesting checks about the languages used in the victim’s system.

Figure 8. Language Checks

We can use LangID to look up the names of these language from the authoritative source at Microsoft.

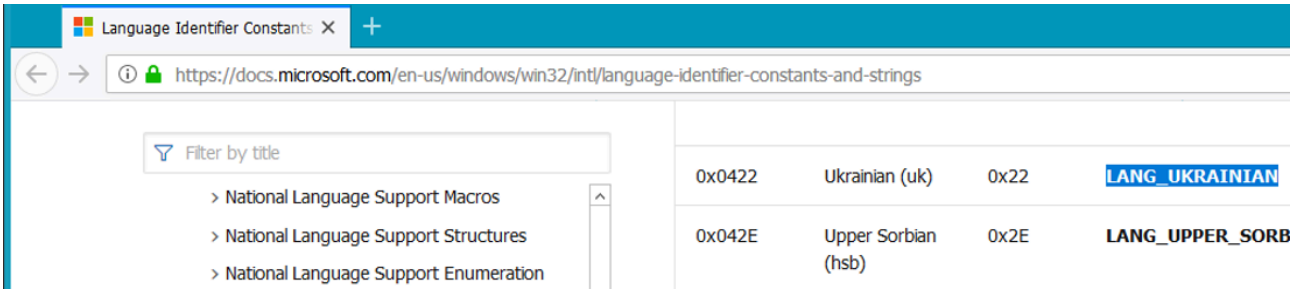

Figure 9. Language Identifier Constants and Strings

Interestingly, DeathRansom checks not just for one language but several languages, but we can still see a clear pattern: it avoids infecting systems in Eastern European countries.

We also managed to extract some interesting information from an undocumented header (specifically, the Rich Signature):

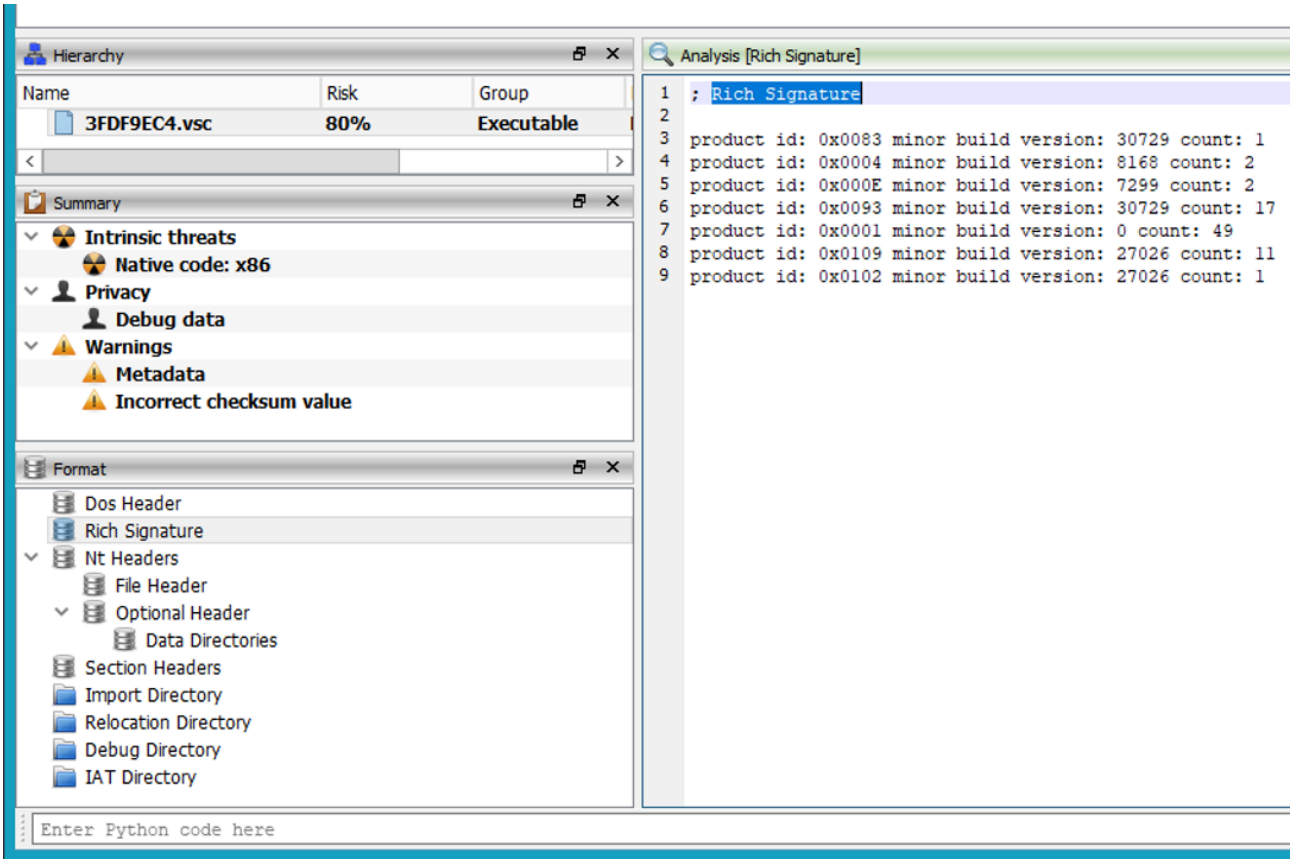

Figure 10. Rich Signature

Based on this information, we were able to figure out the product used to make this malware. Notice the following extra marks used in the descriptions:

+ [C] - object files produced by C compiler

+ [IMP] - DLL import record in library file

+ [LNK] - files produced by a linker

product id: 0x0083 = MSVS2008 [C]

product id: 0x0004 = MSVS6 [LNK]

product id: 0x000E = ?

product id: 0x0093 = MSVS2008 [IMP]

product id: 0x0001 Objects without @comp.id

product id: 0x0109 = ?

product id: 0x0102 = VS 2019 / 2017 / 2015 [LNK]

It's interesting that a product from 1998 (e.g. MSVS6) is still being used to make malware.

New Version Encrypts Files

Recently, we found a new version of DeathRansom, and the primary change is that the malware now actually encrypts files. Our analysis is focused on a sample with the SHA256 hash of:

ab828f0e0555f88e3005387cb523f221a1933bbd7db4f05902a1e5cc289e7ba4.

The new version of this ransomware uses a combination of Curve25519 algorithm for the Elliptic Curve Diffie-Hellman (ECDH) key exchange scheme, Salsa20, RSA-2048, AES-256 ECB, and a simple block XOR algorithm to encrypt files.

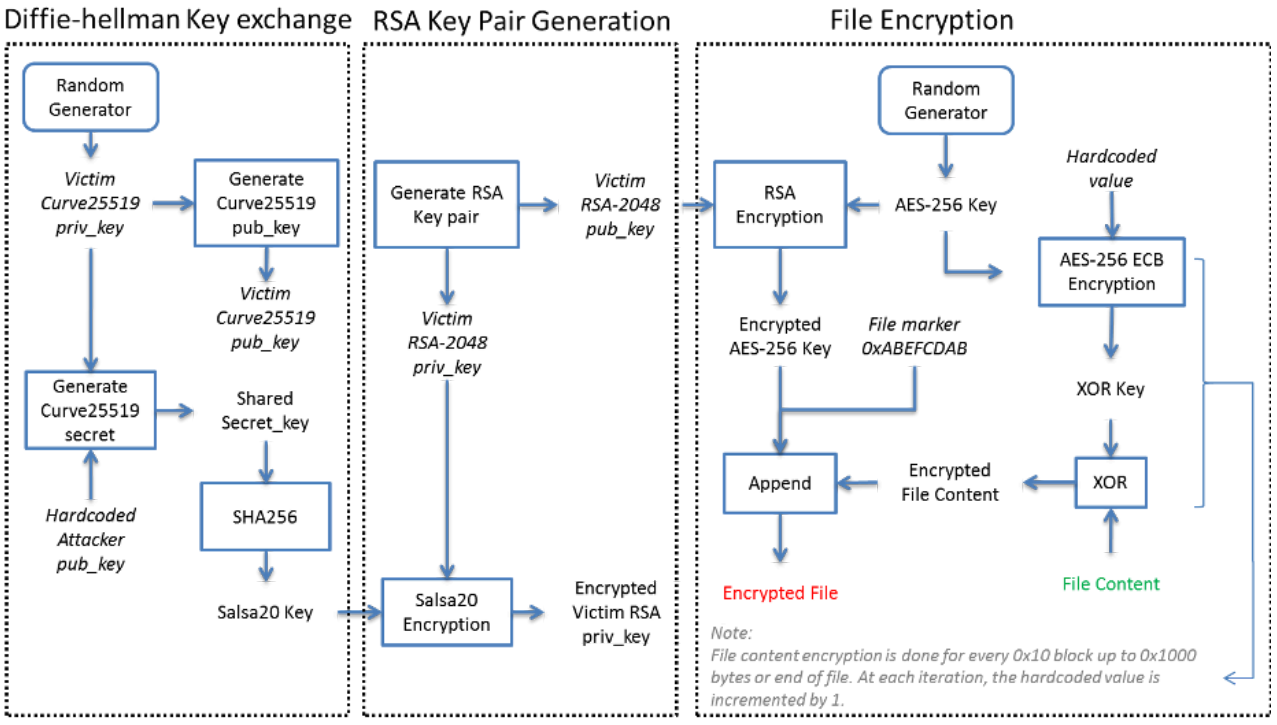

Figure 11. Key generation and file encryption

Diffie-Hellman Key Exchange

1. A random 32-byte value is generated using *advapi32.SystemFunction036* (the same as RtlGenRandom). This serves as the victim's curve25519 private key to the ECDH key exchange.

2. Using the Curve25519 algorithm, the victim's Curve25519 public key is derived from the victim's Curve25519 private key. This is the main information needed by the attacker to eventually decrypt the victim's files. This is included in the registry "HKCU\Software\Wacatac\private" as shown below.

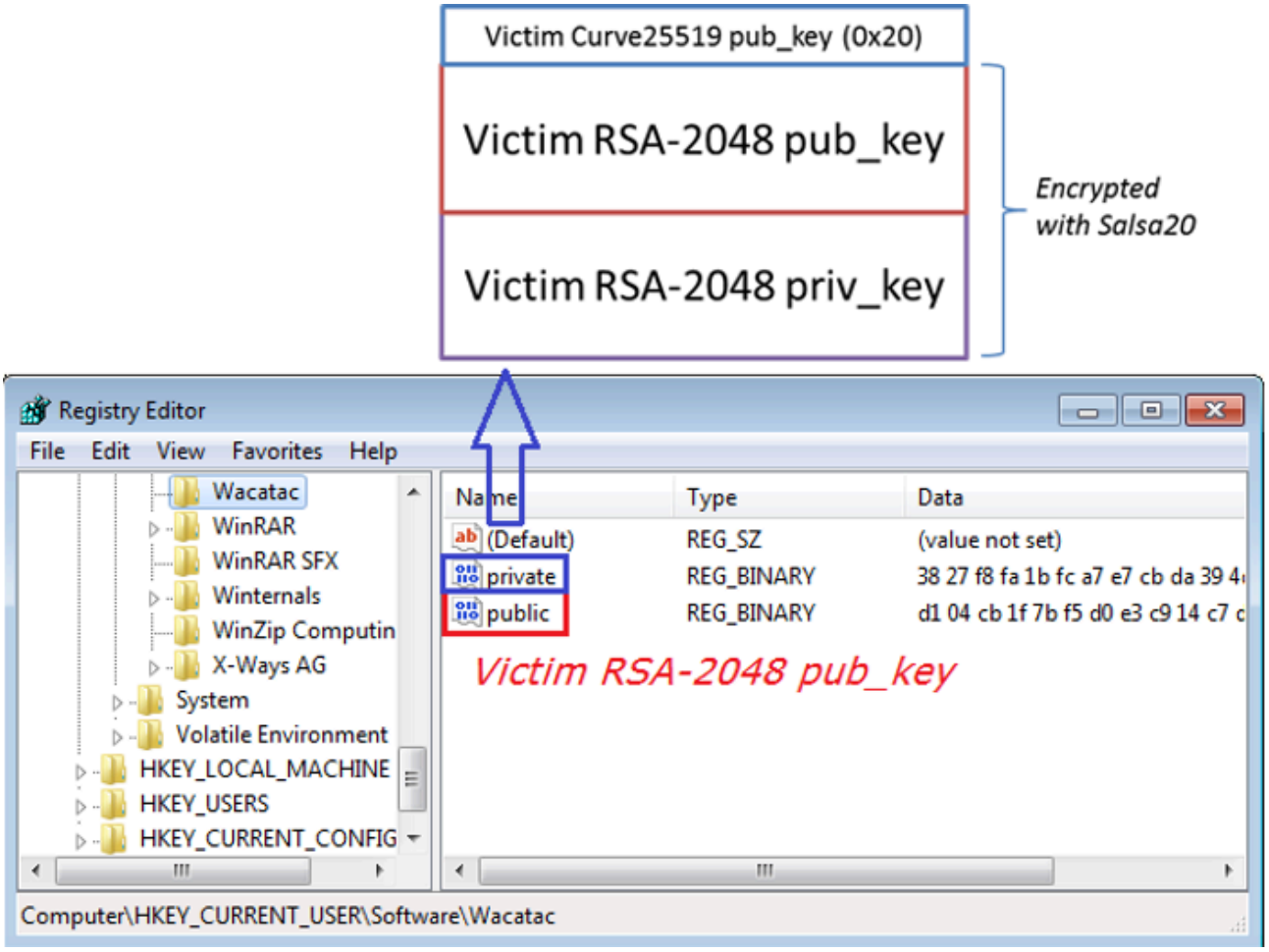

Figure 12. Added Registries

3. Using the Curve25519 algorithm, a shared secret key is derived from a 32-byte hardcoded value (the attacker's Curve25519 public key) and the previously generated victim's Curve25519 private key described in step 1.

4. The SHA256 hash of the shared secret key is computed, which will be used later on.

RSA Key Pair Generation

5. An RSA-2048 key pair is generated

6. The RSA-2048 private key is encrypted using Salsa20, with the secret key's SHA256 hash (from step 4, above) and then included in the registry file "HKCU\Software\Wacatac\private" (see Fig 12).

7. The RSA-2048 public key is written in the registry file “HKCU\Software\Wacatac\public” (see Fig 12).

File Encryption

8. A random 32-byte value is generated using the advapi32.SystemFunction036, which is used as the AES-256 key. This is done for every file.

9. A 16-byte hardcoded value is encrypted with AES-256 ECB, using the previously generated random value as the AES-256 key.

10. The result from step 9 is the used as the key to a 16-byte block XOR operation with the content of the targeted file.

11. Repeat steps 9 and 10 while incrementing the hard-coded value by 1 on each loop until it encrypts the whole file or until it encrypts 4 kilobytes (4096 bytes).

12. The AES key (from step 8) is encrypted using the victim’s RSA-2048 public key (generated in step 5)

13. The encrypted AES key and the marker 0xABEFCDAB, is then appended to the encrypted file.

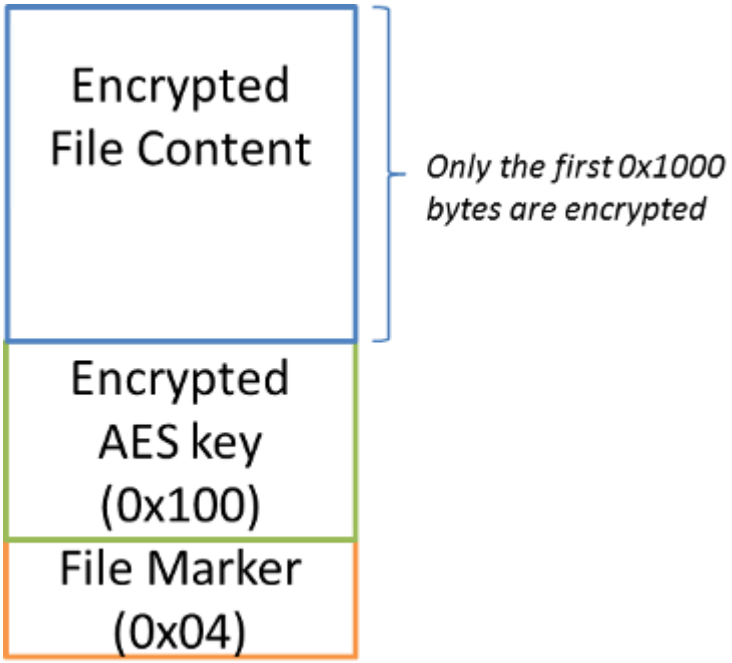

Figure 13. Encrypted file format

Ultimately, for the victim files to be decrypted the shared secret must be generated. And for that, at least one of the following pairs is needed:

- The victim’s curve25519 public key (which can be obtained from registry or ransom note) and the attacker’s curve25519 private key (possessed by attacker)
- The attacker’s curve25519 public key (embedded in the binary) and the victim’s curve25519 private key (lost after the malware’s execution)

NOTE: Further investigation of this ransomware's encryption behavior in underway to check for any implementation flaws. We will be releasing updates for any new findings.

After encryption, it drops a ransom note in every directory. It includes a LOCK-ID that is unique for every user. This is the same data that can be found in the "HKCU\Software\Wacatac\private" registry and encoded in base64.

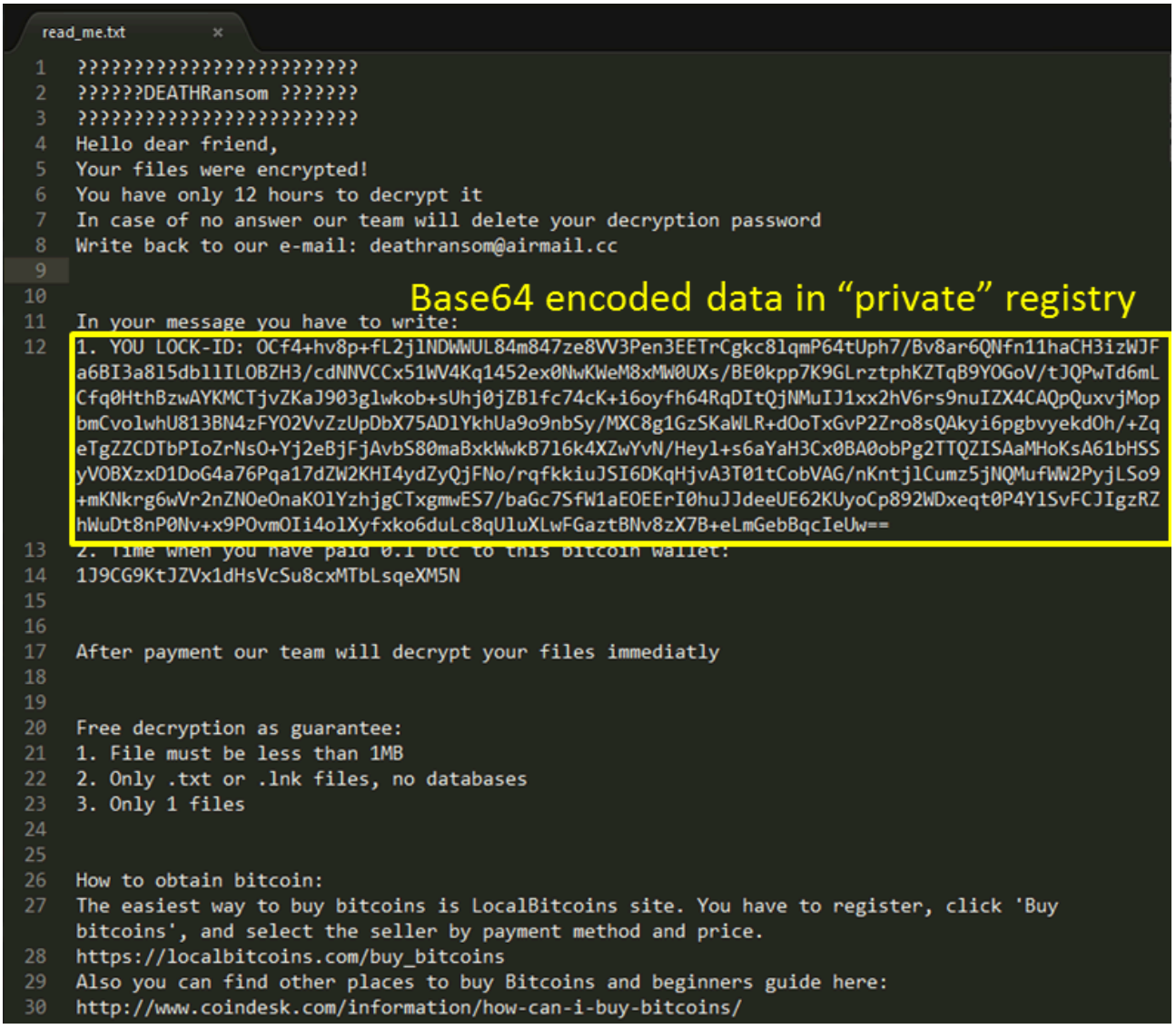

Figure 14. New Ransom Note

Aside from a connection to *hxxps://iplogger[.]org/1Zqq77* to get the victim's public IP address, the ransomware does not have any other network communication. However, as we discussed in the previous section, to generate the shared secret key needed to decrypt the files the attacker must obtain the victim's curve25519 public key. This is why the instruction to the victim in the ransom note is for them to send the LOCK-ID through email.

Solution

Internal testing by FortiGuard Labs shows that all networks and devices being protected by FortiGate solutions running the latest updates were automatically protected from this malware. In addition:

- The FortiGuard Web Filtering service blocks hxxps://iplogger[.]org/1Zqq77
- The FortiGuard Antivirus service detects the SHA 256 samples used in this analysis as the following:

    7C2DBAD516D18D2C1C21ECC5792BC232F7B34DADC1BC19E967190D79174131D1

    - This is detected as W32/Filecoder.B!tr.ransom

    13D263FB19D866BB929F45677A9DCBB683DF5E1FA2E1B856FDE905629366C5E1

    AB828F0E0555F88E3005387CB523F221A1933BBD7DB4F05902A1E5CC289E7BA4

    - These are detected as W32/Kryptik.ANT!tr

Finally, as part of our membership in the Cyber Threat Alliance, details of this threat were shared in real time with other Alliance members to help create better protections for customers.

Conclusion

DeathRansom is a new malware. Naturally, things are moving fast. In our experience, a malware author changes the malware often over time to improve features or to avoid detection. In fact, after our intital assessment of the first version of DeathRansom we noticed a short article describing how DeathRansom had begun encrypting files, making them inaccessible. Our research confirmed that, indeed, the malware author has addressed the missing part of the initial release and DeathRansom has now become a “proper” ransomware.

In our follow-up analysis we try to shed light on how DeathRansom can be associated with other attacks, and who may be behind the creation of this malicious program. Continue reading to learn more in DeathRansom Part II: Attribution.

The author would like to thank Artem Semenchenko, Rommel Joven and Joie Salvio for additional insights during the research process.

IOCs

Samples:

7C2DBAD516D18D2C1C21ECC5792BC232F7B34DADC1BC19E967190D79174131D1

13D263FB19D866BB929F45677A9DCBB683DF5E1FA2E1B856FDE905629366C5E1

AB828F0E0555F88E3005387CB523F221A1933BBD7DB4F05902A1E5CC289E7BA4

Network:

hxxps://iplogger[.]org/1Zqq77

Learn more about how FortiGuard Labs provides unmatched security and intelligence services using integrated AI systems. Find out about the FortiGuard Security Services portfolio and sign up for our weekly FortiGuard Threat Brief.

Read about the FortiGuard Security Rating Service, which provides security audits and best practices to guide customers in designing, implementing, and maintaining the security posture best suited for their organization.
