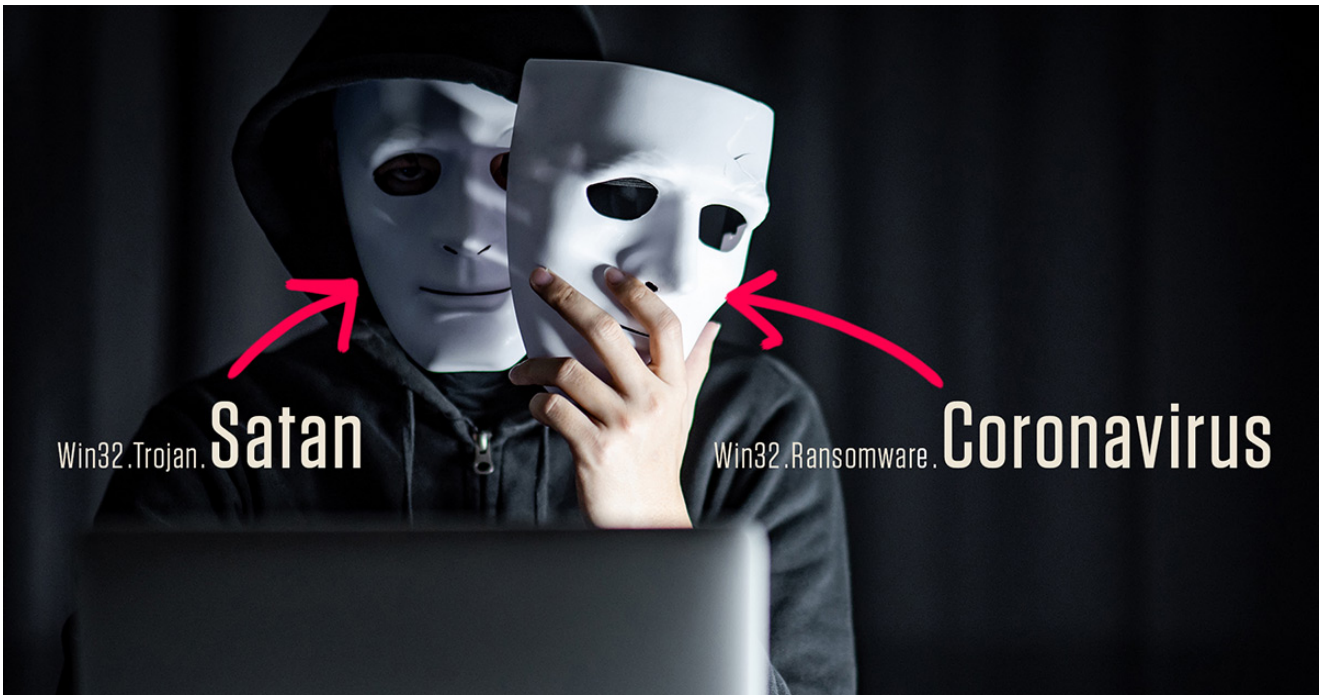
# Retread Ransomware

blog.reversinglabs.com/blog/retread-ransomware

Threat Research | June 5, 2020

Blog Author
Robert Simmons, Independent malware researcher and threat researcher at ReversingLabs. Read More...




In March of 2020, MalwareHunterTeam discovered a downloader which installed both a KPot infostealer as well as a second payload which was a ransomware variant that used the string "CoronaVirus". This sample was leveraging ongoing current events and appears to be some form of cover for or distraction from the infostealer trojan that was installed alongside it. Via code analysis of this "CoronaVirus" sample, it is clear that it reuses a large amount of code from a four year old sample of ransomware detected as "Satana".

This older malware sample was first seen on June 29, 2016. This old sample and the recent one share two decoding algorithms that are used to hide strings, code, and a little, tiny PE file. This embedded file is run via a modification to the BootExecute registry key. The new file has some changes to the strings that are encoded along with the encoded data hiding the ransom note and the Bitcoin wallet addresses used to collect payment.

These are new addresses with a number of payments made during the same time that the new file was observed in the wild. The ransom amount demanded is quite small: $50. All in all, this is a strange executable. What follows is an examination of the two decoding algorithms and a method for discovering the old samples based on YARA rules that focus on these algorithms.

#### String Decoding

Many malware families obfuscate the strings that are used during their execution as a way to foil static analysis and detections based on the decoded forms of the strings. In the samples examined here, the string encoding is very simple. It is a substitution cipher which shifts one character then subtracts the position index of the character in the encoded string to reveal the decoded character. This algorithm is highlighted in Figure 1.

```
                        loc_401873:
0x00401873 8B55FC       mov     edx, dword [ebp+var_4]
0x00401876 C6040200     mov     byte [edx+eax], 0x0
0x0040187a C60000       mov     byte [eax], 0x0
0x0040187d 8A1406       mov     dl, byte [esi+eax]
0x00401880 2AD1         sub     dl, cl
0x00401882 FECA         dec     dl
0x00401884 881407       mov     byte [edi+eax], dl
0x00401887 41           inc     ecx
0x00401888 40           inc     eax
0x00401889 3B4DF8       cmp     ecx, dword [ebp+var_8]
0x0040188c 76E5         jbe     loc_401873
```

**Figure 1: String Decoding Algorithm**

By focusing on the algorithm itself, which can be more stable than surrounding code when reused, one may locate other malware samples that could be related to the sample being analyzed. Figure 2 shows the same algorithm, but in a debugger where the first encoded string that is operated on in the sample can be seen in the dump at the bottom.

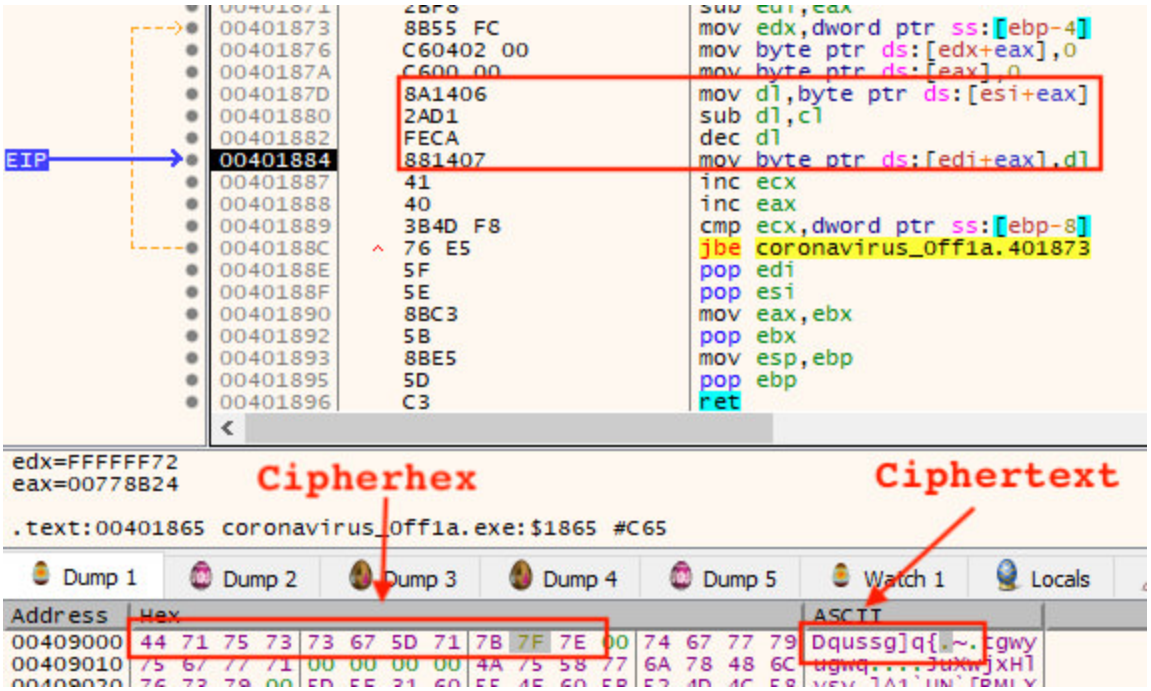



**Figure 2: First Encoded String in Sample**

This same algorithm can be easily translated into Python and the strings from the sample decoded without executing the sample. A small Python program for decoding these strings is shown in Figure 3. The first string in the sample, which decodes to "CoronaVirus", is shown here.

```
[2]: cipherstring = 'Dqussg]q{\x7f~'

[3]: print(cipherstring)
     Dqussg]q{~

[4]: len(cipherstring)
[4]: 11

[5]: chars = list()

     for index, letter in enumerate(cipherstring):
         temp = ord(letter) - index
         temp -= 1
         chars.append(chr(temp))

[6]: ''.join(chars)
[6]: 'CoronaVirus'
```

**Figure 3: Python Implementation of String Decoder Algorithm**

Taking only the bytes of the instructions that comprise this algorithm, a byte string for a YARA rule is identified.

```
$op1 = { 8A 14 06 2A D1 FE CA 88 14 07 }
```

By then deploying this YARA rule as a retrohunt in the Titanium Platform, a number of additional samples are found. The results of this retrohunt are shown in Figure 4.

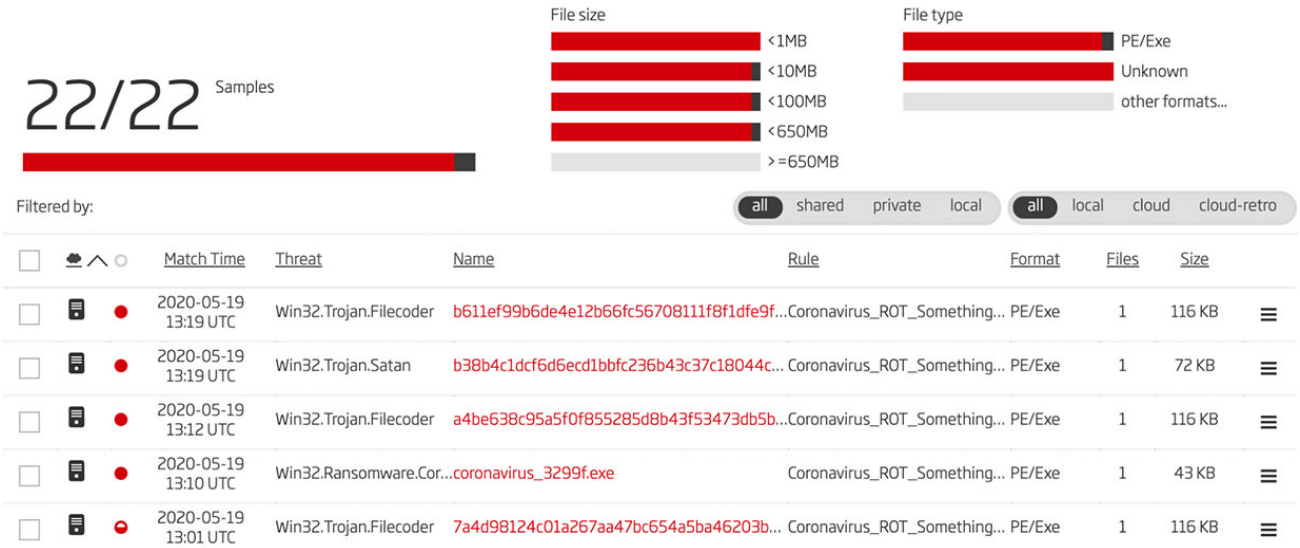



**Figure 4: Retrohunt Results**

### Old Satana Sample

In the results of the retrohunt there are a number of samples from 2016. The sample with the earliest 'first seen' date is shown in Figure 5.

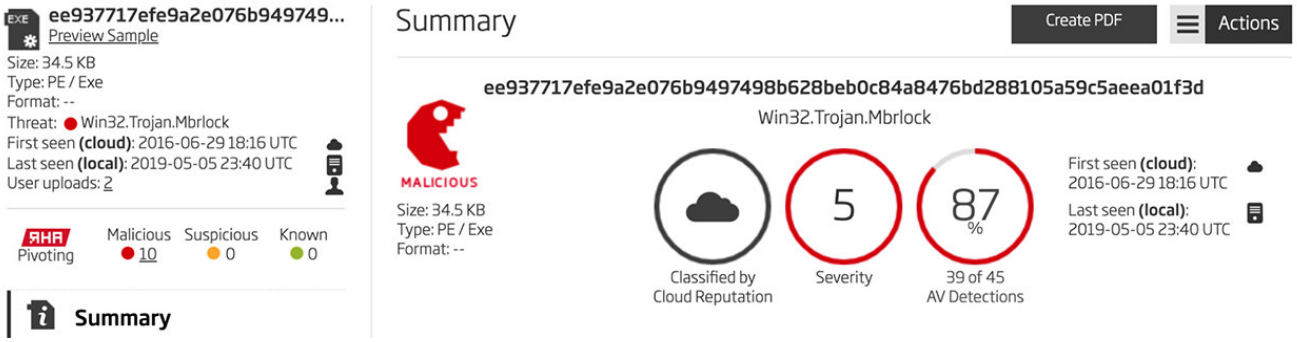



**Figure 5: Earliest Related Satana Sample from 2016**

This file is well detected as Satana via the TitaniumCore YARA classification as well as being classified as ransomware by the TitaniumCore Machine Learning classification. Both of these classifications are seen in Figure 6.

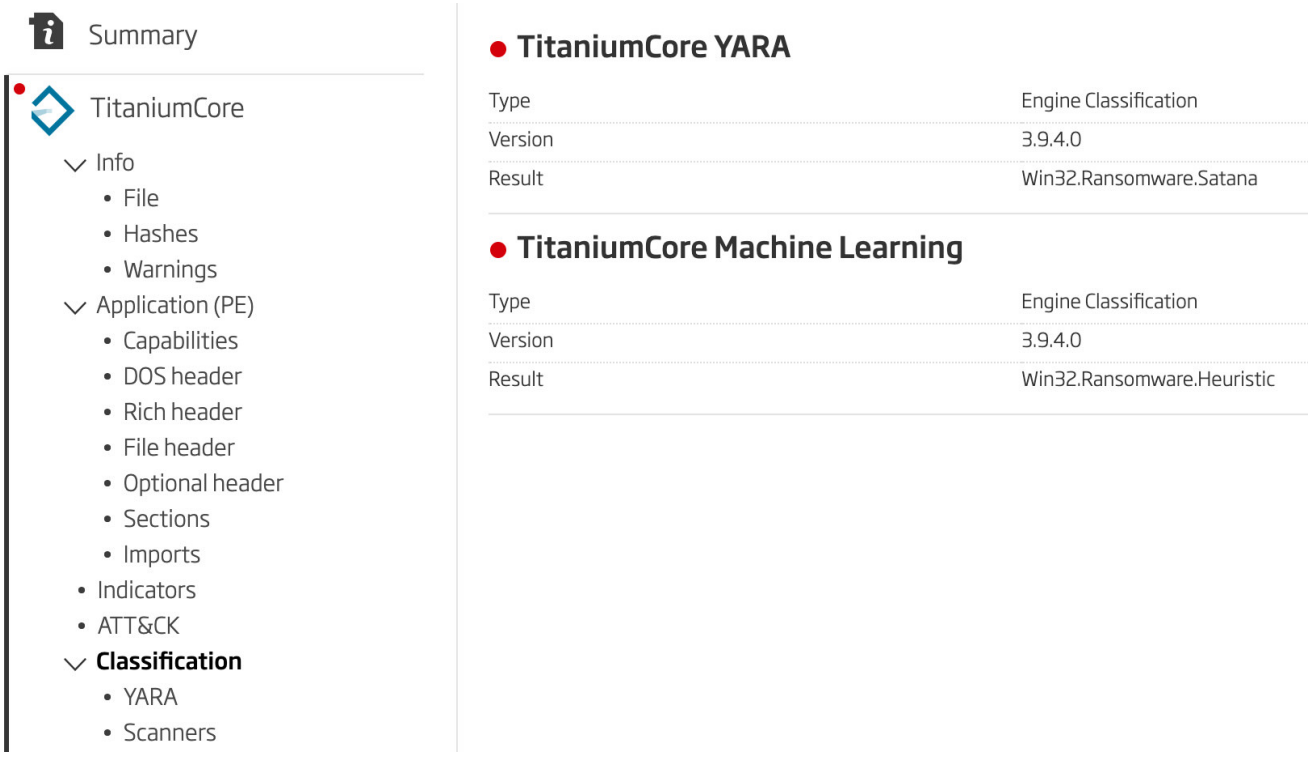



**Figure 6: Detection and Classification**

In addition to this first sample, there are nine other samples that have been identified as related to the first based on the ReversingLabs Hash Algorithm (RHA). These other samples are seen in Figure 7.

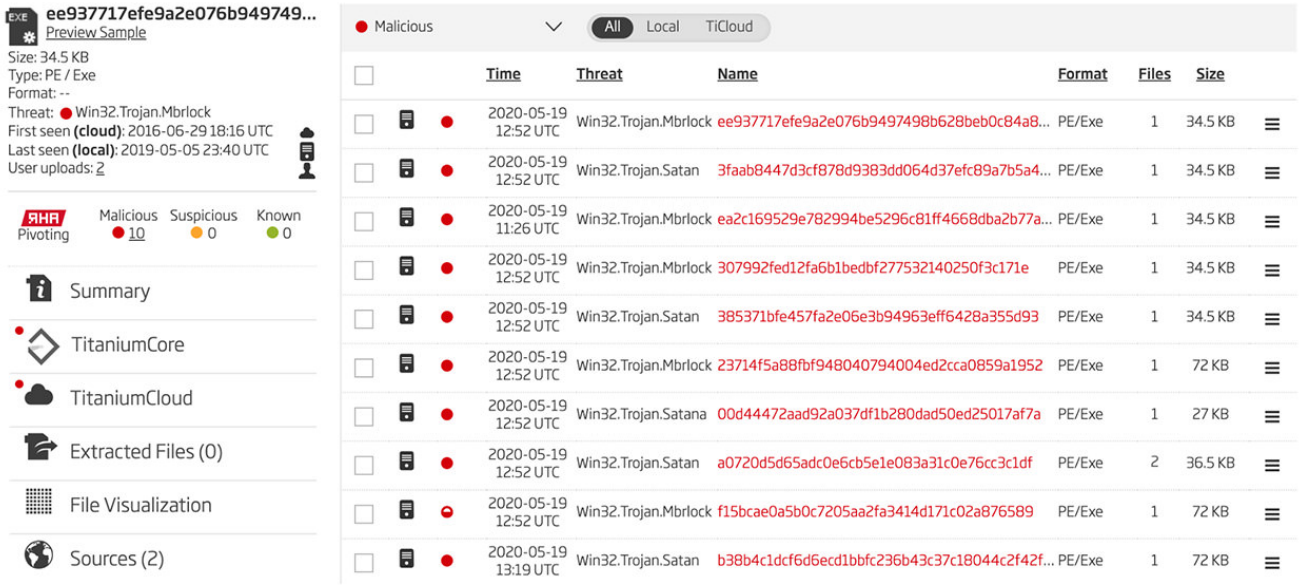



**Figure 7: Cluster of Ten Files Related via RHA**

The earliest files in the cluster, each of which are also first seen in 2016, are very minor changes to the earliest file. These may be researchers testing or other types of analysis artifacts. One such minor change is highlighted in Figure 8 which shows a side-by-side

comparison of the two files using HexFiend.

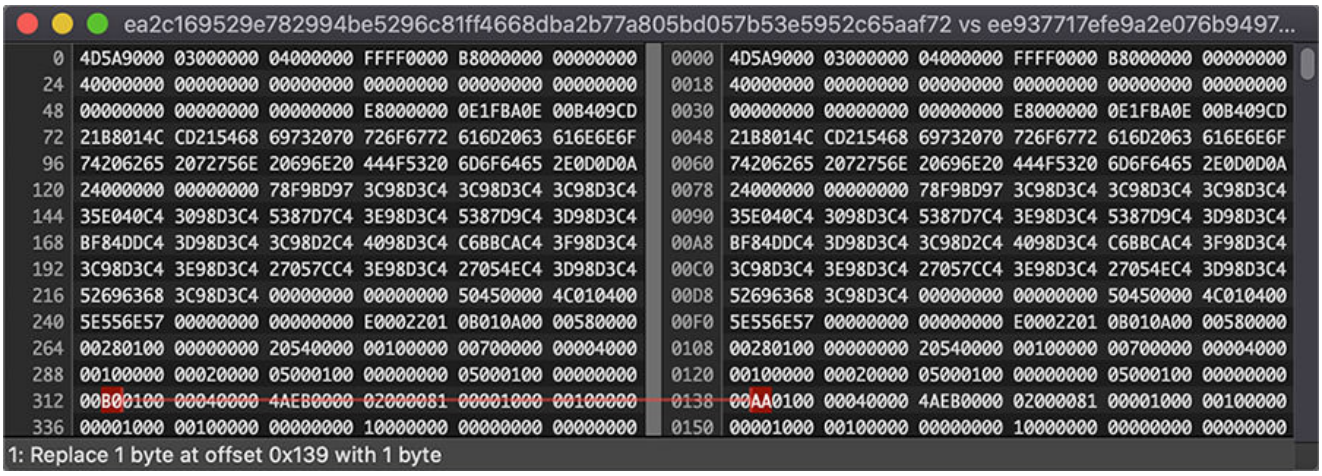



**Figure 8: Side-by-side Comparison of Satana Samples**

More recently observed files from this same cluster are clearly analysis artifacts from researchers or analysis tools. Many of them simply have the encoded strings decoded in the sample and no other changes. Knowing that all the files in the cluster are essentially the same file, only the earliest sample (ee937717efe9a2e076b9497498b628beb0c84a8476bd288105a59c5aeea01f3d) is used for comparison with the CoronaVirus sample.

#### Data Decoding

Another common method used to encode data to avoid detection and identification is the "exclusive or" operation or XOR. The sample analyzed here uses this operation to hide large blobs of data. It specifically encodes the ransom note as well as a very small PE file which is used to display a variation of the ransom note on reboot. The location where this XOR operation is carried out is shown in Figure 9.

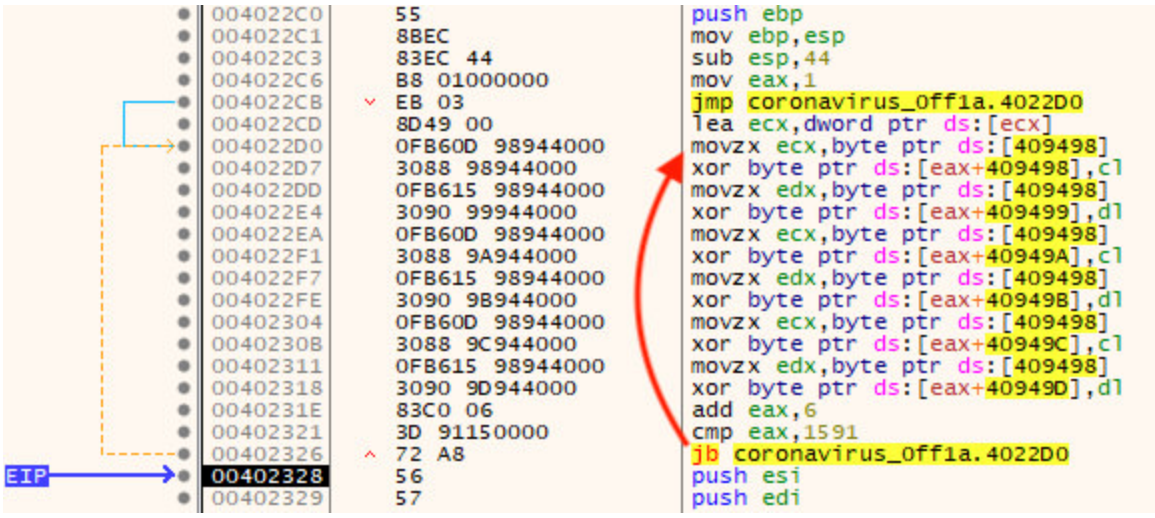



**Figure 9: XOR Decoding Process**

This is a loop which uses the first byte of the data as the key and then operates using that key on the rest of the bytes in the data. The encoded data before this operation has started is shown in Figure 10. The XOR key is the very first byte, 0x58.

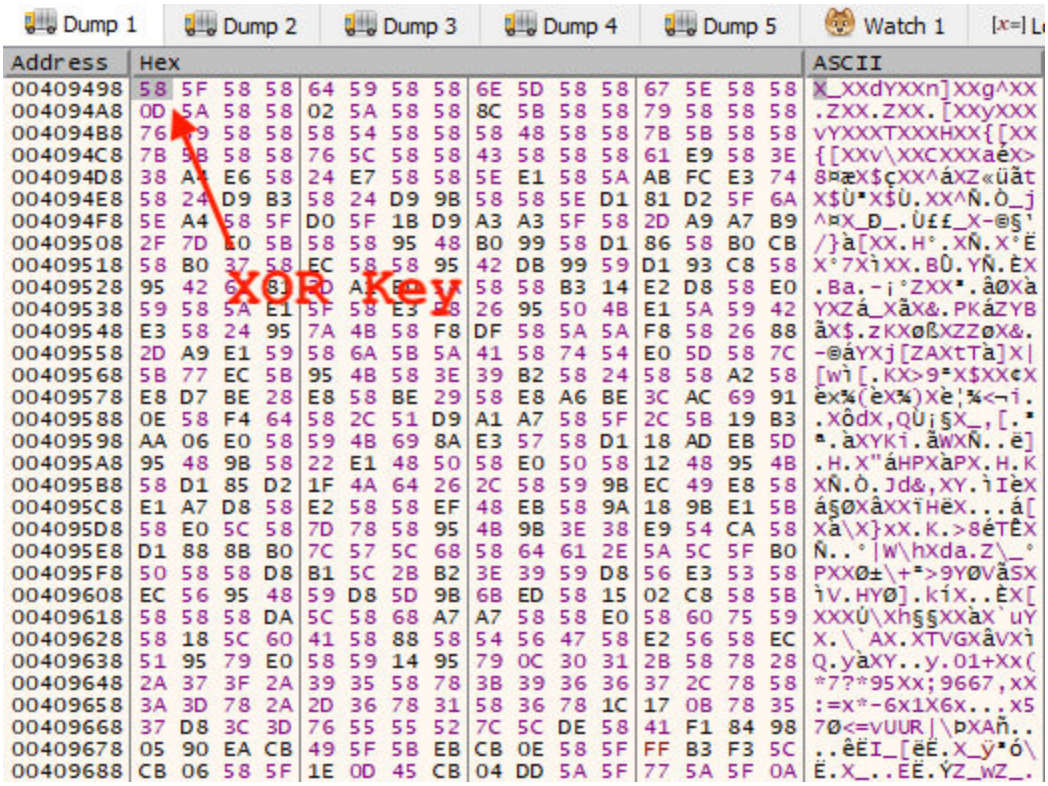



**Figure 10: First Byte of Encoded Data is XOR Key**

After this decoding process is complete, the embedded PE file begins to appear. This data is still compressed, but the file magic "MZ" and the DOS stub string are clear. The decoded version of the data shown above is seen in Figure 11.

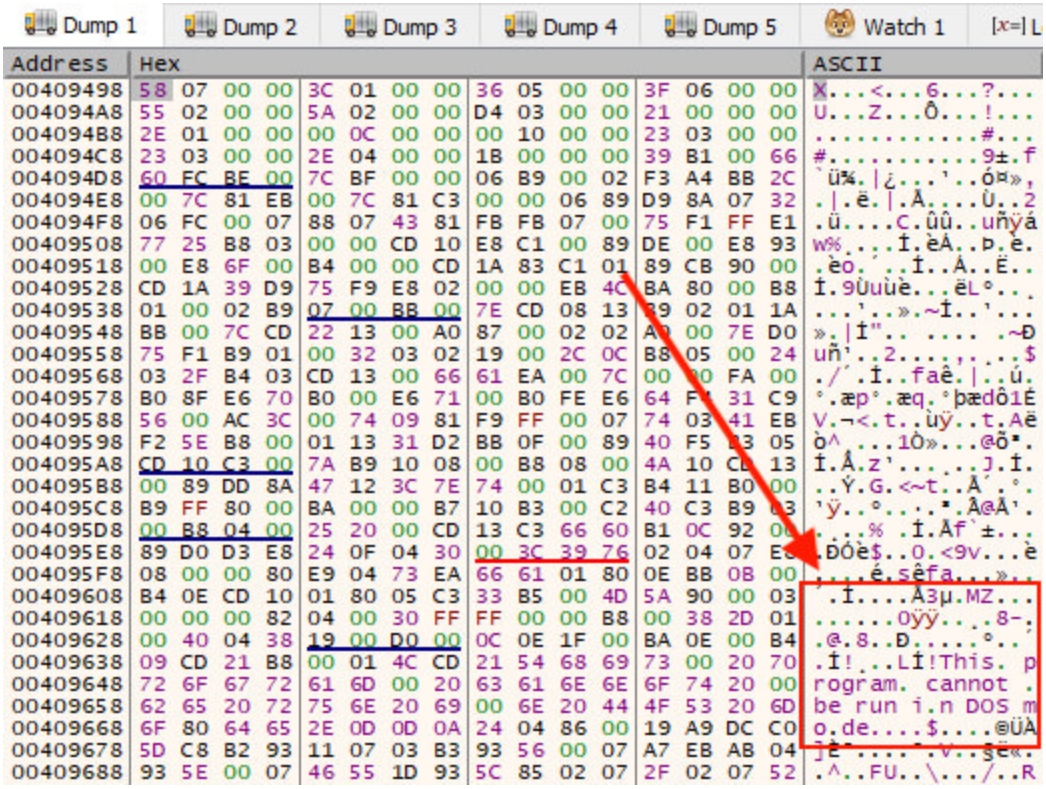



**Figure 11: Decoded Data with PE File Starting to Appear**

The next step after decoding the data is to decompress it. The instructions that perform this action utilize the library function "RtlDecompressBuffer". The function name is loaded dynamically by decoding a string using the string decoding capability examined earlier. As puzzling as it sounds, something like a comic book supervillain telling you their next step before doing something evil, the meaning of these instructions is stated clearly by the additional string, "DeCompressor", which is not encoded or hidden in any way. These instructions as seen in the debugger are shown in Figure 12.

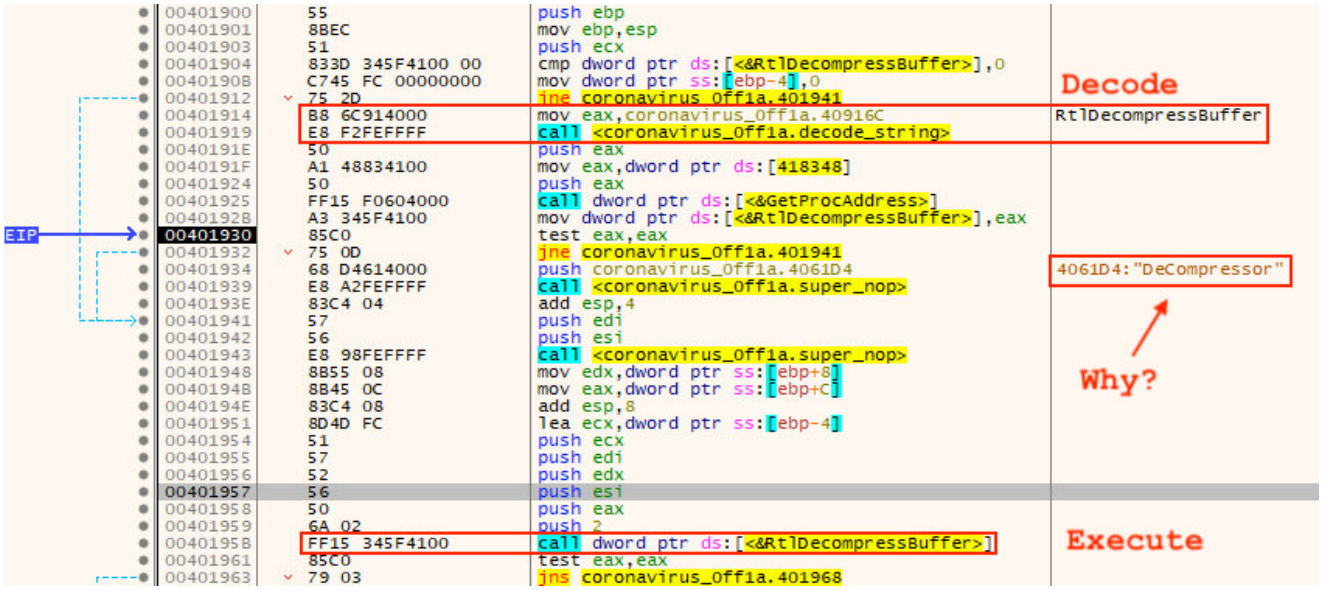



**Figure 12: Disassembled Decompression Instructions**

Aside from the tiny PE file and the ransom note template, the decoded list of Bitcoin wallet addresses is provided. One of these addresses is selected to be used with the ransom note template to generate the note that is dropped by the ransomware. These decoded and decompressed Bitcoin wallet addresses are shown in Figure 13.

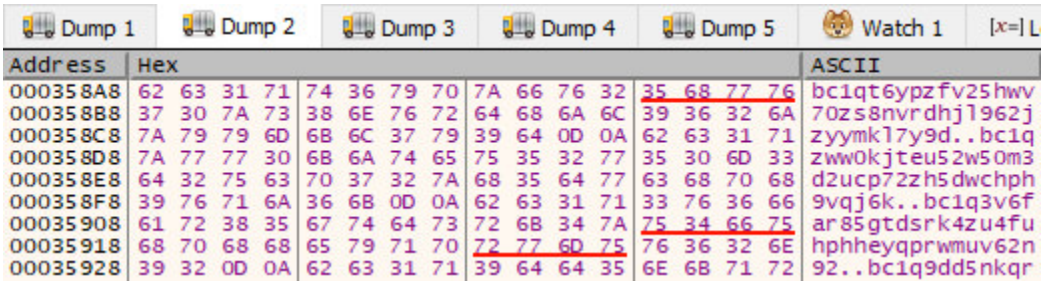



**Figure 13: Decoded and Decompressed Bitcoin Wallet Addresses**

These addresses are listed in Table 1. The list of wallet addresses in the sample have two duplicates, so the total set of addresses has 22 members each of which is of type Bech32 SegregatedWitness address format.

| |
|---|
| bc1qc9axh3fq2ypgcd92j582v9khfrn52strql7ztn |
| bc1qzww0kjteu52w50m3d2ucp72zh5dwchph9vqj6k |
| bc1q5e8pwyk9rqtq400agngmq5h23cuz42x0wlqw3q |
| bc1qrkp9cx6svxguxupx9p0z5ss4nmyr4fwhvgkasg |
| bc1qjl0ufmwct84ww69zwyxe99gext7za6qkyhx200 |
| bc1q6ryyex33jxgr946u3jyre66uey07e2xy3v2cah |
| bc1qlmu9xk8wdnydnlcvy9uvcepzklcv7kxyhk8ymy |
| bc1qftwqsaw57v6cstwrdvclmkz63plvf5q3vqvw4k |
| bc1qegps92ddvgv8t45lfcn02afsjlyf7mynuqvpmm |
| bc1quwc6yqgdcgm6z2663vpjm9cgtfwf7mhk2n7gtn |
| bc1q9dd5nkqrxsny93r9u09jwq8agvkf04afxh67jg |
| bc1qt6ypzfv25hwv70zs8nvrdhjl962jzyymkl7y9d |
| bc1qgd3nj0486k35ra42a550ntyafdr7s5lmyzjn29 |
| bc1q8r42fm7kwg68dts3w70qah79n5emt5m76rus5u |
| bc1q3v6far85gtdsrk4zu4fuhphheyqprwmuv62n92 |
| bc1qpvguajy4rxr7743hzuwfmz32krzzfcjl9rf0qx |

| |
|---|
| bc1qe9gj2sj3an73dq37vpe34xflc83yjp4u3pzfgz |
| bc1qpaksevt2w6cqdeqjvm8dapvz66y3hs3jjy4x66 |
| bc1qt3uf3wx569z5z9wdeanuj4rwq7m06grhxt96v3 |
| bc1qzv6h2zaedjgduc6xmyn86hdsu0skuunt9lhkxn |
| bc1q2x9h8wlh2cuxjd9rv94v6syz7lpxxk2wwrndmv |
| bc1qxsjfw0jftfr9n5urdyh7xmz4cspls8qefuyetr |

**Table 1: Bitcoin Wallet Addresses**

The ransom note template has four locations where string replacement is used to generate the final note text that is dropped to the ransom note file. These locations in the template are highlighted in Figure 14.

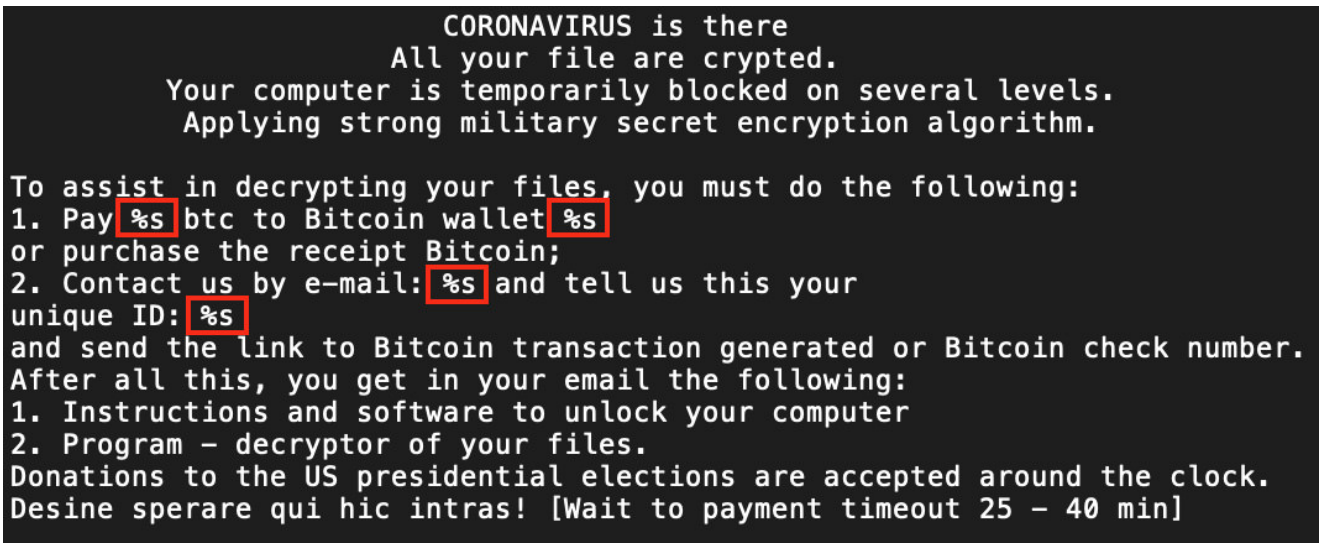



**Figure 14: Ransom Note Template**

The email address, coronaVi2022[@]protonmail[.]ch, along with the amount demanded in Bitcoin (BTC), 0.008, are both hard coded. The unique ID is generated by concatenating the drive serial number collected from PhysicalDrive0 and the HwProfileGuid and then generating an MD5 from this string. The Bitcoin wallet address is chosen from the list decoded and decompressed as documented above. These four values just before the call to sprintf can be seen in Figure 15.

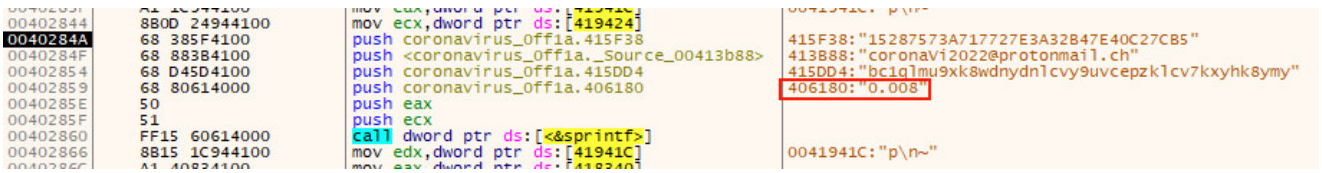


**Figure 15: String Replacement into Ransom Note Template**

The payment demand of 0.008 BTC, interestingly, does not line up with any of the seven payments made to the set of wallet addresses from the binary. Each is a single payment. Some of the payments are larger than the demand and some are smaller than the demand. It is unknown whether these payments are in fact connected with the ransomware.

### Old Satana Sample Differences

The majority of the instructions that handle the generation of randomness and moving that random data around in memory are all identical between the Satana and the Coronavirus samples. Library and compiler added code along with these identical functions are shown in Figure 16 as analyzed using Relyze's binary difference feature.

Everything | Equal | Modified | Removed | Added

| Difference ▾ | Item Type | Diff Type | Item A | Item B |
|---|---|---|---|---|
| 0.00% | Function | Equal | _allrem1 | _allrem1 |
| 0.00% | Function | Equal | _seh_longjmp_unwind | _seh_longjmp_unwind |
| 0.00% | Function | Equal | anonymous_library_func1 | __unwind_handler |
| 0.00% | Function | Equal | func_0x405B60 | func_0x405D8C |
| 0.00% | Function | Equal | func_0x402794 | func_0x4019E4 |
| 0.00% | Function | Equal | memset1 | memset1 |
| 0.00% | Function | Equal | __matherr | __matherr |
| 0.00% | Function | Equal | memcpy1 | memcpy1 |
| 0.00% | Function | Equal | func_0x401E80 | func_0x401170 |
| 0.00% | Function | Equal | func_0x402290 | func_0x401580 |
| 0.00% | Function | Equal | func_0x402060 | func_0x401350 |
| 0.00% | Function | Equal | anonymous_library_func | __abnormal_termination |
| 0.00% | Function | Equal | func_0x401DB0 | func_0x401000 |
| 0.00% | Function | Equal | _chkstk1 | _chkstk1 |
| 0.00% | Function | Equal | func_0x402EA0 | func_0x401FA0 |
| 0.00% | Function | Equal | func_0x402BF0 | func_0x401E40 |
| 0.00% | Function | Equal | func_0x402740 | func_0x401990 |
| 0.00% | Function | Equal | func_0x402780 | func_0x4019D0 |
| 0.00% | Function | Equal | func_0x405C38 | func_0x405E64 |
| 0.00% | Function | Equal | RtlUnwind1 | RtlUnwind1 |
| 0.00% | Function | Equal | func_0x405C7A | func_0x405EA6 |

**Figure 16: Binary Difference Showing Identical Functions**

There are a number of functions that have some amount of change according to binary difference, however, most of these instruction changes do not have a significant effect on the malware sample's behavior. The strings that are operated on are different and one significant location of change is the BTC demand amount which was 0.5 BTC in the older sample and 0.008 BTC in the newest. This change in price in the disassembly is shown in Figure 17.

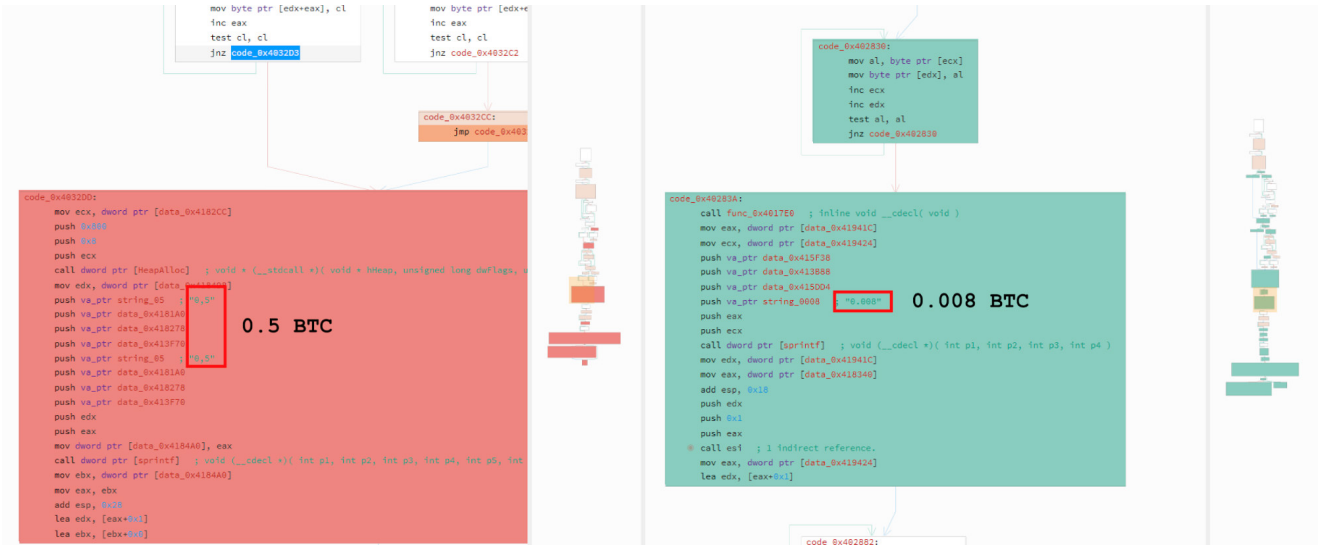



**Figure 17: Difference in Hard-Coded BTC Demand**

### Conclusion

As can be seen through comparison between the older Satana sample and the new Coronavirus sample, these two are very closely related. This old sample, or the source code for it, has been repurposed and redeployed as "CoronaVirus" ransomware. However, due to the distribution alongside the KPot sample, the very low demand amount of approximately $70, and the payments that are oddly larger or smaller than the demand: this may be a faux ransomware campaign. Totaling all the payments received, the set of wallet addresses only collected 0.10417322 BTC or about $930, a strangely low amount.

### YARA Rule

```
private rule WindowsPE
  {
      condition:
          uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550
  }

  rule SubstCipher_MinusIndex
  {
      meta:
          author = "Malware Utkonos"
          date = "2020-04-13"
          exemplar =
"ee937717efe9a2e076b9497498b628beb0c84a8476bd288105a59c5aeea01f3d"
      strings:
          $op1 = { 8A 14 06 2A D1 FE CA 88 14 07 }
      condition:
          WindowsPE and all of them
  }
```

## MORE BLOG ARTICLES