

# Retread Ransomware

---

 [blog.reversinglabs.com/blog/retread-ransomware](https://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.

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code is the same as in Figure 1. A red box highlights the instructions from 0x0040187d to 0x00401884. Below the assembly, the debugger shows registers (edx=FFFFFF72, eax=00778B24) and a memory dump. The dump shows hex values and their ASCII equivalents. A red box highlights the hex values 44 71 75 73 73 67 5D 71 7B 7F 7E 00, which correspond to the ASCII string 'Dqussg]q[~.cgwy'.

**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'

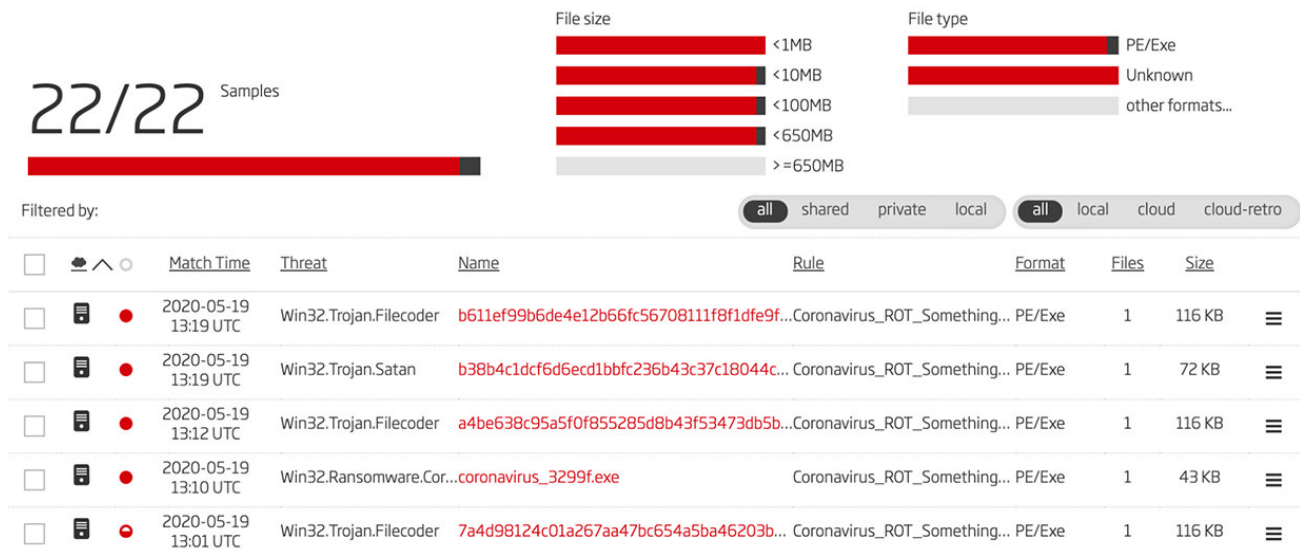
```

**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 }
```

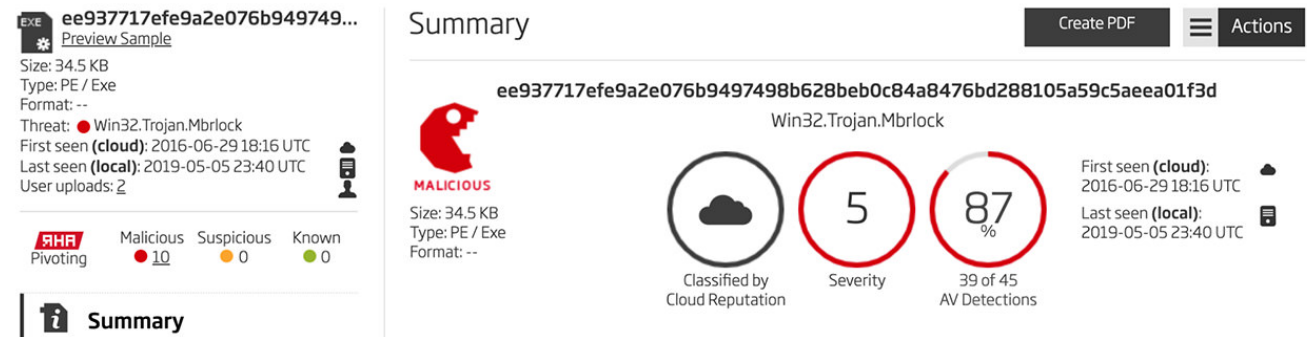
By then deploying this YARA rule as a retrohunt in the [Titanium Platform](#), a number of additional samples are found. The results of this retrohunt are shown in Figure 4.



**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.

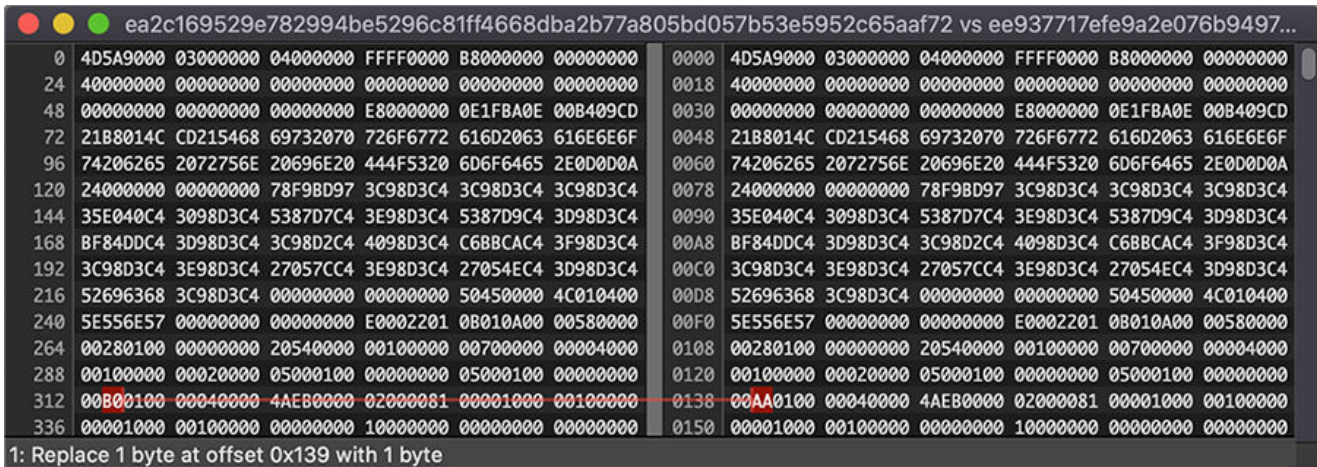


**Figure 5: Earliest Related Satana Sample from 2016**

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.



comparison of the two files using HexFiend.

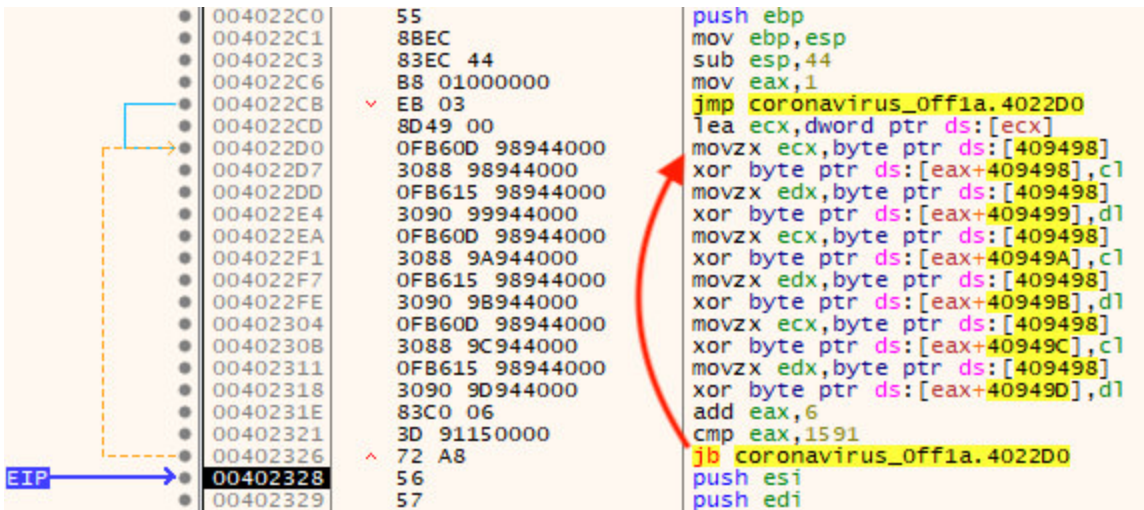


**Figure 8: Side-by-side Comparison of Satana Samples**

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.

Address	Hex	ASCII
00409498	58 5F 58 58 64 59 58 58 6E 5D 58 58 67 5E 58 58	X_XXdyXXn]XXg^XX
004094A8	0D 5A 58 58 02 5A 58 58 8C 58 58 58 79 58 58 58	.ZXX.ZXX.[XXyXXX
004094B8	76 59 58 58 58 54 58 58 58 48 58 58 7B 58 58 58	VYXXXTXXXHXX{[XX
004094C8	7B 58 58 58 76 5C 58 58 43 58 58 58 61 E9 58 3E	{[XXv\XXCXXXaéx>
004094D8	38 A1 E6 58 24 E7 58 58 5E E1 58 5A AB FC E3 74	8pæx\$cXX^âxz«üât
004094E8	58 24 D9 B3 58 24 D9 98 58 58 5E D1 81 D2 5F 6A	X\$Û*x\$Û.XX^Ñ.Ô_j
004094F8	5E A4 58 5F D0 5F 1B D9 A3 A3 5F 58 2D A9 A7 B9	^PX_D_Üff_X-@§'
00409508	2F 7D 10 58 58 58 95 48 B0 99 58 D1 86 58 B0 CB	/j]â[XX.H^,xÑ.X^É
00409518	58 B0 37 58 EC 58 58 95 42 DB 99 59 D1 93 C8 58	X^7XîXX.BÛ.YÑ.ÈX
00409528	95 42 68 58 58 58 83 14 E2 D8 58 E0 .Ba.-j^ZXX^..â0Xa	
00409538	59 58 5A E1 5F 58 E3 28 26 95 50 48 E1 5A 59 42	YXZâ_xâx&.PKâZYB
00409548	E3 58 24 95 7A 48 58 F8 DF 58 5A 5A F8 58 26 88	âx\$.zKXoßXZZ0x&.
00409558	2D A9 E1 59 58 6A 58 5A 41 58 74 54 E0 5D 58 7C	-@âYxj[ZAXtTâ]X
00409568	58 77 EC 58 95 48 58 3E 39 B2 58 24 58 58 A2 58	[w [,KX>9^X\$XXcX
00409578	E8 D7 BE 28 E8 58 BE 29 58 E8 A6 BE 3C AC 69 91	èx%(èX%)Xè;%<-i.
00409588	0E 58 F4 64 58 2C 51 D9 A1 A7 58 5F 2C 5B 19 B3	.xôdx,QUi\$X_,[.°
00409598	AA 06 E0 58 59 48 69 8A E3 57 58 D1 18 AD EB 5D	°.âXYki..âwxÑ..è]
004095A8	95 48 9B 58 22 E1 48 50 58 E0 50 58 12 48 95 48	.H.X"âHPXâPX.H.K
004095B8	58 D1 85 D2 1F 4A 64 26 2C 58 59 9B EC 49 E8 58	XÑ.Ô.Jd&,XY.îIèX
004095C8	E1 A7 D8 58 E2 58 58 EF 48 EB 58 9A 18 9B E1 58	â§0xâXXîHèX...â[
004095D8	58 E0 5C 58 7D 78 58 95 48 9B 3E 38 E9 54 CA 58	Xâ\X}xx.K.>8éTÈX
004095E8	D1 88 8B B0 7C 57 5C 68 58 64 61 2E 5A 5C 5F B0	Ñ..° w\hxda.Z\_^
004095F8	50 58 58 D8 B1 5C 2B B2 3E 39 59 D8 56 E3 53 58	PXX0±\+>9Y0VâSX
00409608	EC 56 95 48 59 D8 5D 98 6B ED 58 15 02 C8 58 58	iv.HY0).kiX..ÈX[
00409618	58 58 58 DA 5C 58 68 A7 A7 58 58 E0 58 60 75 59	XXXÛ\Xh\$§XXâx`uY
00409628	58 18 5C 60 41 58 88 58 54 56 47 58 E2 56 58 EC	X.\`AX.XTVGXâVXî
00409638	51 95 79 E0 58 59 14 95 79 0C 30 31 2B 58 78 28	Q.yâXY..y.01+XX(
00409648	2A 37 3F 2A 39 35 58 78 3B 39 36 36 37 2C 78 58	*7?*95Xx;9667,XX
00409658	3A 3D 78 2A 2D 36 78 31 58 36 78 1C 17 0B 78 35	:=x*-6x1X6x...x5
00409668	37 D8 3C 3D 76 55 55 52 7C 5C DE 58 41 F1 84 98	70<=vUUR `pXâñ..
00409678	05 90 EA CB 49 5F 58 EB CB 0E 58 5F FF B3 F3 5C	..èËI_[èÉ.X_ÿ*0\
00409688	CB 06 58 5F 1E 0D 45 CB 04 DD 5A 5F 77 5A 5F 0A	È.X_..ÈÈ.YZ_wZ_.

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.



Address	Hex	ASCII
00409498	58 07 00 00 3C 01 00 00 36 05 00 00 3F 06 00 00	X...<...6...?...
004094A8	55 02 00 00 5A 02 00 00 D4 03 00 00 21 00 00 00	U...Z...ô...!...
004094B8	2E 01 00 00 00 0C 00 00 D4 10 00 00 23 03 00 00	.....#.....
004094C8	23 03 00 00 2E 04 00 00 18 00 00 00 39 B1 00 66	#.....9±.f
004094D8	60 FC BE 00 7C BF 00 00 06 B9 00 02 F3 A4 BB 2C	`ü% ¿...'.ô»»,
004094E8	00 7C 81 EB 00 7C 81 C3 00 00 06 89 D9 8A 07 32	. .ë .A...Ü..2
004094F8	06 FC 00 07 88 07 43 81 FB FB 07 00 75 F1 FF E1	.ü...C.üü.unÿä
00409508	77 25 B8 03 00 00 CD 10 E8 C1 00 89 DE 00 E8 93	w%...i.ëA..p.e.
00409518	00 E8 6F 00 84 00 00 CD 1A 83 C1 01 89 CB 90 00	.eo...I..A..É..
00409528	CD 1A 39 D9 75 F9 E8 02 00 00 EB 4C BA 80 00 B8	i.9üüë...èL°...
00409538	01 00 02 B9 07 00 BB 00 7E CD 08 13 89 02 01 1A	...'....~i...'
00409548	B8 00 7C D0 22 13 00 A0 87 00 02 02 A7 00 7E D0	» i".....~D
00409558	75 F1 B9 01 00 32 03 02 19 00 2C 0C B8 05 00 24	un'.2.....\$
00409568	03 2F B4 03 CD 13 00 66 61 EA 00 7C 00 00 FA 00	./..i..faë. .ü.
00409578	B0 8F E6 70 80 00 E6 71 00 B0 FE E6 64 F1 31 C9	°.ap°.æq°.pædô1É
00409588	56 00 AC 3C 00 74 09 81 F9 FF 00 07 74 03 41 EB	V.-<.t..uy'.t.Aë
00409598	F2 5E B8 00 01 13 31 D2 B8 0F 00 89 40 F5 13 05	ò^...10'...@ò*.
004095A8	CD 10 C3 00 7A B9 10 08 00 B8 08 00 4A 10 C0 13	i.Ä.z'...J.i.
004095B8	00 89 DD 8A 47 12 3C 7E 74 00 01 C3 B4 11 B0 00	.Y.G.<~t..A'..
004095C8	B9 FF 80 00 BA 00 00 B7 10 B3 00 C2 40 C3 B9 63	'y...°.A@A'.
004095D8	00 B8 04 00 25 20 00 CD 13 C3 66 60 B1 0C 92 00	...%..i.Af'±...
004095E8	89 D0 D3 E8 24 0F 04 30 00 3C 39 76 02 04 07 E0	Ð0ë\$.O.<9v...ë
004095F8	08 00 00 80 E9 04 73 EA 66 61 01 80 0E B8 08 00	...ë.sëfa...»
00409608	B4 0E CD 10 01 80 05 C3 33 B5 00 4D 5A 90 00 03	..I....A3µ.MZ...
00409618	00 00 00 82 04 00 30 FF FF 00 00 B8 00 38 2D 01	.....öyy...8-
00409628	00 40 04 38 19 00 D0 00 0C 0E 1F 00 BA 0E 00 B4	..@.8..D.....°
00409638	09 CD 21 B8 00 01 4C CD 21 54 68 69 73 00 20 70	.i!...LI!This. p
00409648	72 6F 67 72 61 6D 00 20 63 61 6E 6E 6F 74 20 00	rogram. cannot
00409658	62 65 20 72 75 6E 20 69 00 6E 20 44 4F 53 20 6D	be run i.n DOS m
00409668	6F 80 64 65 2E 0D 0D 0A 24 04 86 00 19 A9 DC C0	o.de....\$....@ÜA
00409678	5D C8 B2 93 11 07 03 B3 93 56 00 07 A7 EB AB 04	jE...v...šë«
00409688	93 5E 00 07 46 55 1D 93 5C 85 02 07 2F 02 07 52	.Ä..FU..\'.../..R

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.

Address	Hex	Disassembly	Comments
00401900	55	push ebp	
00401901	8BEC	mov ebp,esp	
00401903	51	push ecx	
00401904	833D 345F4100 00	cmp dword ptr ds:[<&RtlDecompressBuffer>],0	
00401908	C745 FC 00000000	mov dword ptr ss:[ebp-4],0	
00401912	75 2D	jne coronavirus_Off1a.401941	
00401914	B8 6C914000	mov eax,coronavirus_Off1a.40916C	
00401919	E8 F2FEFFFF	call <coronavirus_Off1a.decode_string>	RtlDecompressBuffer
0040191E	50	push eax	
0040191F	A1 48834100	mov eax,dword ptr ds:[418348]	
00401924	50	push eax	
00401925	FF15 F0604000	call dword ptr ds:[<&GetProcAddress>]	
00401928	A3 345F4100	mov dword ptr ds:[<&RtlDecompressBuffer>],eax	
00401930	85C0	test eax,eax	
00401932	75 0D	jne coronavirus_Off1a.401941	
00401934	68 D4614000	push coronavirus_Off1a.4061D4	4061D4: "DeCompressor"
00401939	E8 A2FEFFFF	call <coronavirus_Off1a.super_nop>	
0040193E	83C4 04	add esp,4	
00401941	57	push edi	
00401942	56	push esi	
00401943	E8 98FEFFFF	call <coronavirus_Off1a.super_nop>	
00401948	8B55 08	mov edx,dword ptr ss:[ebp+8]	
0040194B	8B45 0C	mov eax,dword ptr ss:[ebp+C]	
0040194E	83C4 08	add esp,8	
00401951	8D4D FC	lea ecx,dword ptr ss:[ebp-4]	
00401954	51	push ecx	
00401955	57	push edi	
00401956	52	push edx	
00401957	56	push esi	
00401958	50	push eax	
00401959	6A 02	push 2	
0040195B	FF15 345F4100	call dword ptr ds:[<&RtlDecompressBuffer>]	Execute
00401961	85C0	test eax,eax	
00401963	79 03	jns coronavirus_Off1a.401968	

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.

Address	Hex	ASCII
000358A8	62 63 31 71 74 36 79 70 7A 66 76 32 35 68 77 76	bc1qt6ypzfv25hvw
000358B8	37 30 7A 73 38 6E 76 72 64 68 6A 6C 39 36 32 6A	70zs8nvrhdj1962j
000358C8	7A 79 79 6D 68 6C 37 79 39 64 0D 0A 62 63 31 71	zyymk17y9d..bc1q
000358D8	7A 77 77 30 68 6A 74 65 75 35 32 77 35 30 6D 33	zww0kjteu52w50m3
000358E8	64 32 75 63 70 37 32 7A 68 35 64 77 63 68 70 68	d2ucp72zh5dwchph
000358F8	39 76 71 6A 36 68 0D 0A 62 63 31 71 33 76 36 66	9vqj6k..bc1q3v6f
00035908	61 72 38 35 67 74 64 73 72 68 34 7A 75 34 66 75	ar85gtdsrk4zu4fu
00035918	68 70 68 68 65 79 71 70 72 77 6D 75 76 36 32 6E	hphheyqprwmu62n
00035928	39 32 0D 0A 62 63 31 71 39 64 64 35 6E 68 71 72	92..bc1q9dd5nkqr

**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

---

- bc1qftwqsaw57v6cstwrdrvclmkz63plvf5q3vqvw4k

---

- bc1qegps92ddvgv8t45lfcn02afsilyf7mynuqvpmmm

---

- bc1quwc6yqgdcm6z2663vpjm9cgtfwf7mhk2n7gtn

---

- bc1q9dd5nkqrxsny93r9u09jwq8agvfkf04afxh67jg

---

- bc1qt6ypzfv25hvw70zs8nvrhdj1962jzyymk17y9d

---

- bc1qgd3nj0486k35ra42a550ntyafdr7s5lmyzjn29

---

- bc1q8r42fm7kwwg68dts3w70qah79n5emt5m76rus5u

---

- bc1q3v6far85gtdsrk4zu4fuhphheyqprwmu62n92

---

- bc1qpvguajy4rxr7743hzuwfmz32krzzfcjl9rf0qx

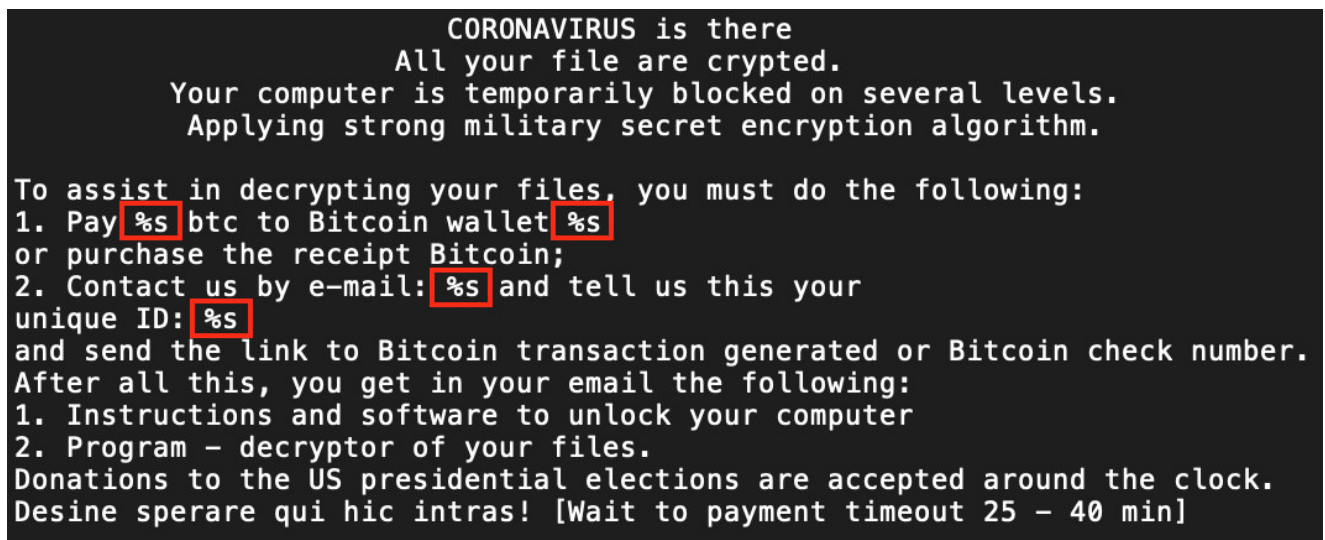
```

bc1qe9gj2sj3an73dq37vpe34xflc83yjp4u3pzfgz
bc1qpaksevt2w6cqdeqjvm8dapvz66y3hs3jyy4x66
bc1qt3uf3wx569z5z9wdeanuj4rwq7m06grhxt96v3
bc1qzv6h2zaedjgduc6xmyn86hdsu0skuunt9lhkxn
bc1q2x9h8wlh2cuxjd9rv94v6syz7lpxxk2wwrndmv
bc1qxsjfw0jftfr9n5urdyh7xmz4cspls8qefuyetr

```

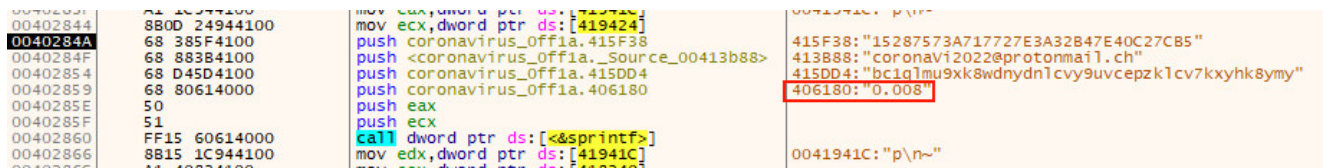
**Table 1: Bitcoin Wallet Addresses**

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.

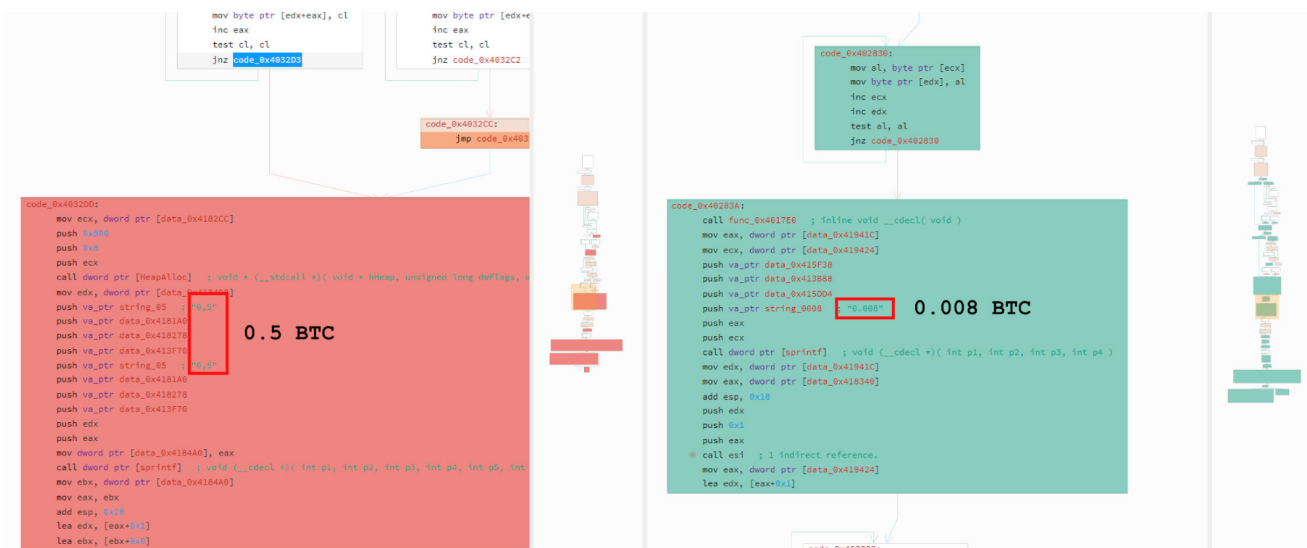
### Old Satana Sample Differences

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_0x405860	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. Totalling 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**