

DMA Locker Strikes Back | Malwarebytes Labs

By hasherezade

Published: 2016-02-08 · Archived: 2026-04-05 19:21:38 UTC

A few days ago we published a post about a new ransomware – DMA Locker (read more [here](#)). At that time, it was using a pretty simple way of storing keys. Having the original sample was enough to recover files. Unfortunately, the latest version (discovered February 8) comes with several improvements and RSA key. Let's take a look at the changes.

DMA Locker in recent campaigns have been found installed by the attackers via Remote Desktop (similar distribution method was used by [LeChiffre ransomware](#)).

[\[UPDATE\] READ ABOUT THE LATEST VERSION OF DMA LOCKER: 4.0](#)

UPDATE: version 3.0 (discovered 22-th Feb) fixed the bug in the cryptography implementation. Due to this fact, encrypted files cannot be recovered by external tools (although it was possible in case of the earlier version, described in this article). Sorry, but our decryptor can no longer help!

PREVENTION TIP: *Create these files to protect yourself from this version of DMA Locker. Content doesn't matter. In presence of these files, the program will go by other path of execution and display the red message only – but not deploy the encryption.*

- *C:Documents and SettingsAll Usersdecrypting.txt*
- *C:Documents and SettingsAll Usersstart.txt*
- *C:ProgramDatadecrypting.txt*
- *C:ProgramDatastart.txt*

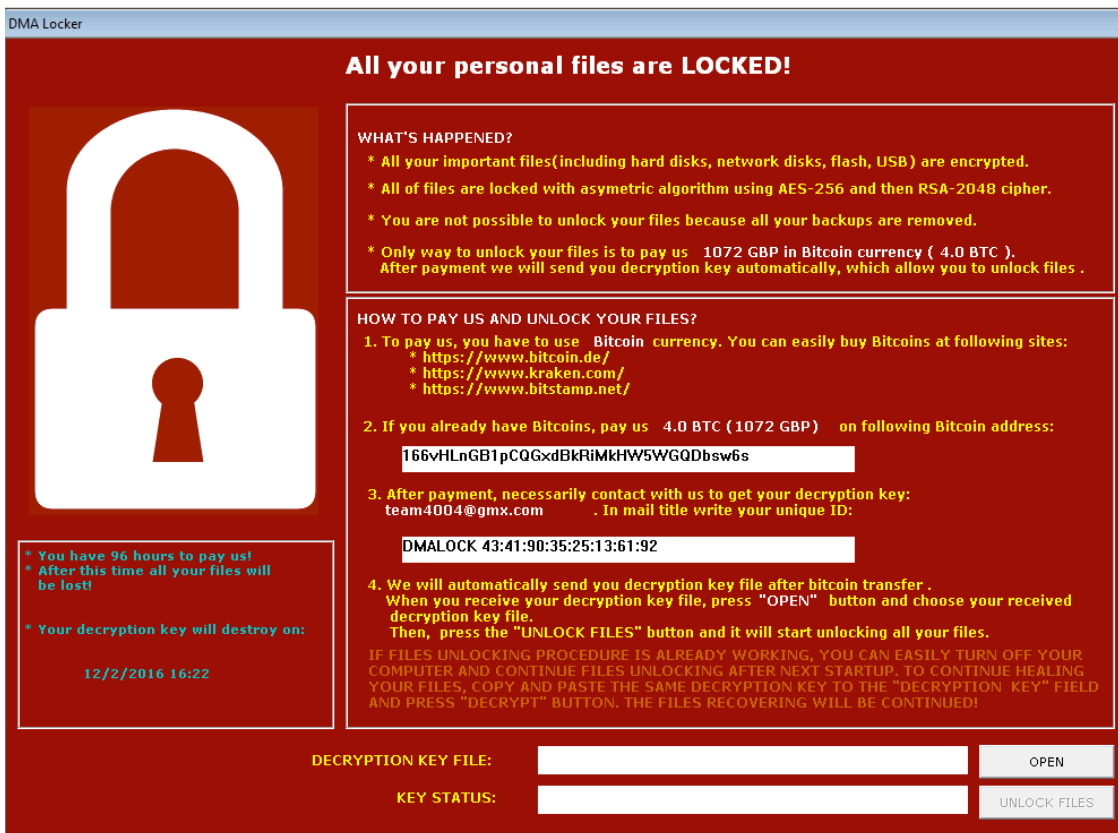
This trick works only as a PREVENTION – once your files are encrypted, it is not going to help. For more info about why it happens, please read this post.

Analyzed sample

[28b44669d6e7bc7ede7f5586a938b1cb](#)

Behavioral analysis

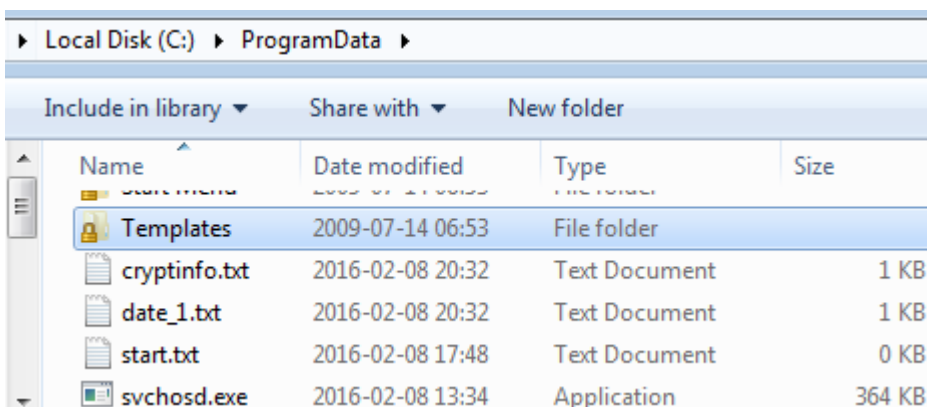
Again we are alerted by a red window – almost identical like before, only the locker image is added:



This time the key necessary to decrypt files must be supplied not as a text, but as RSA key file. Author added also key validation.



Similarly, it drops files in **C:\ProgramData** (or **C:\Documents and Settings\All Users**). Now, the dropped copy is named **svchosd.exe**.



And created registry keys to autorun the file and to autodisplay ransom note via notepad at system startup.

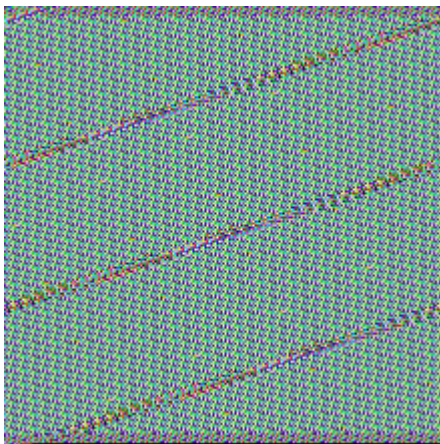
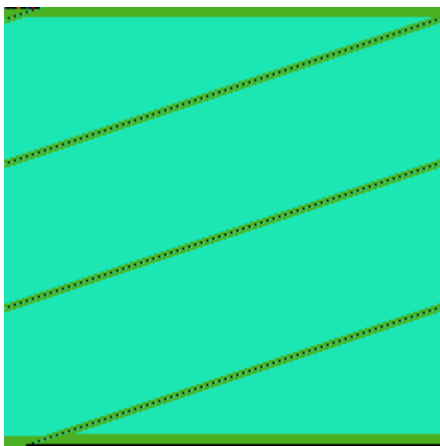
Encrypted files again have unchanged extensions – they can be only recognized by 8 byte long prefix at the beginning of the content. In the previous edition it was “**ABCXYZ11**“, in current it is “**!DMALOCK**“:

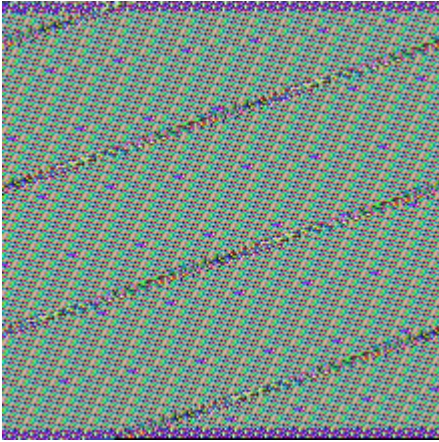
```
00000000 21 44 4d 41 4c 4f 43 4b ef 02 37 2a b3 56 ff 3c |!DMALOCK..7*.V.<|
00000010 c1 af b9 d9 f2 5f cf 4b 7f 18 5f 28 c3 1d 41 d9 |....._K..(..A.|
00000020 d3 27 8b 7f 7f 97 ee 38 5a f8 37 ab a6 18 df 3a |.'.....8Z.7.....:|
00000030 2f 53 6a ac 9d 48 02 3f 35 1a 8f fb f3 97 95 01 |/Sj..H.?5.....|
00000040 59 2b c4 2f d9 1f ce 3f c5 7f 1f 35 1b 98 49 e1 |Y+./...?...5..I.|
00000050 f0 ba d1 8e 17 64 8f ad ea 95 f6 ae b7 a1 c2 93 |.....d.....|
00000060 69 c3 32 9c 8c 10 f5 3f 9c bf 3f 8d c1 71 5c a9 |i.2....?...?.q\.
```

Experiment

Let's compare how the encrypted files look

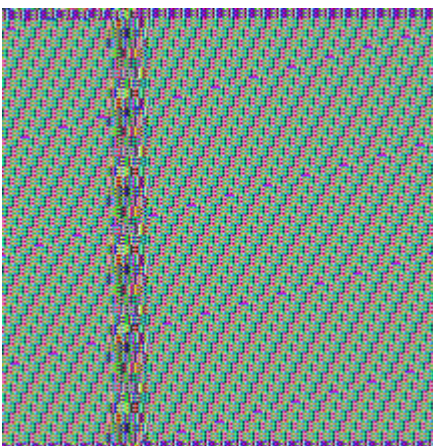
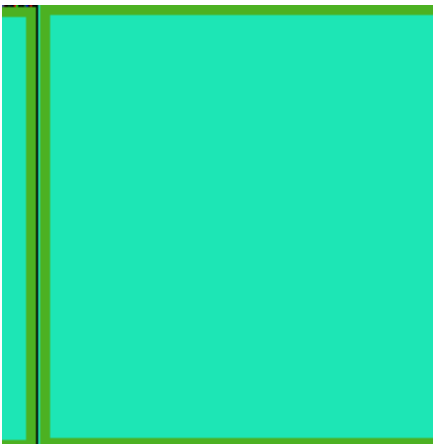
From the left we can see visualizations of raw bytes of following files: original, encrypted by previous DMA Locker, encrypted by current DMA Locker





Previous DMA Locker(middle picture) was encrypting files by AES-256 ECB mode, applied on 16 byte long chunks of input. Now (last picture), also repetitive patterns exist – so probably AES-256 ECB mode was used again.

However, pay attention to the strips in the BMP – in a new file they are shifted a bit more. It would suggest that the header is longer than previously. Let's visualize the same files with a different width, to make sure that this impression is right. The header of the file is visualized as a line at the top left corner – it ends where the vertical line starts.



Now it is visible clearly – the header is really longer. Why? To answer this question, code analysis is required – but it can signify, that some additional data have been stored there (it can be for example the AES key, encrypted

by RSA).

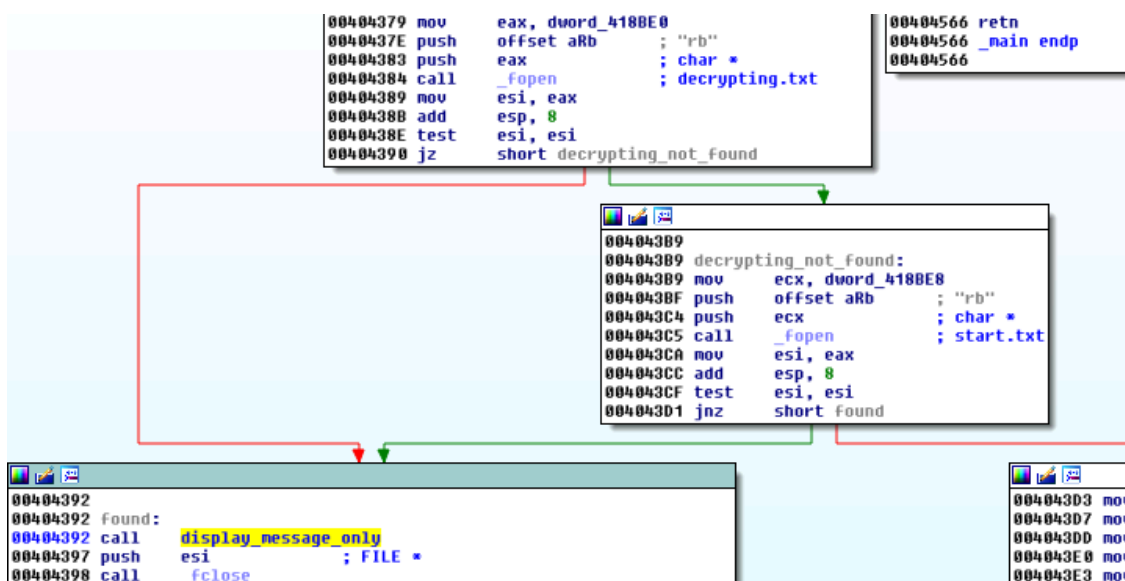
Inside

When does the encryption start?

At the beginning of execution, (as in the previous version) the [malware](#) terminates applications used for backups. Also, adds registry keys for its persistence. Then, execution of the main function may follow 3 alternative paths.

- if system is already infected -> do not deploy encryption, only display the red window with ransom note
- system is not yet infected, malware is not yet installed (current file name is different than the expected one – **svchostd.exe**) -> install the malware in **ProgramData** and then deploy again the dropped file
- system is not yet infected, but malware is installed -> **deploy encryption**, after finishing display the red window with ransom note

The recognition, in which state is the system, is performed basing on the presence of some predefined files. Presence of file **decrypting.txt** informs that system is already infected. File **start.txt** informs that encrypting started (and no need to start it again):



Knowing this fact, we can easily drop those files by our own and fake that our system is infected. It will prevent this version of DMA Locker from attacking our system (it will display the ransom note but not touch our files).

How does the encryption work?

This time the author decided to practice what he preached and really used RSA key (previous version supplied to the encrypting function just a text key, read from the end of the original sample).

```

00E244C7 . CALL svchosd.00E258ED
00E244C8 . PUSH EAX
00E244C9 . CALL svchosd.00E258ED
00E244CA . ADD ESP,0x8
00E244DB . LEA EAX,DWORD PTR SS:[ESP+0x68]
00E244DC . PUSH 0x0
00E244DD . PUSH EAX
00E244DE . CALL svchosd.00E22820 encrypt_logical_drives
00E244DF . ADD ESP,0x8
    
```

Stack address=0014FD84
EAX=0014FD88

Address	Hex dump	ASCII
0014FD84	1A 00 00 00 06 02 00 00 00 A4 00 00 52 53 41 31	+...@...A..RSA1
0014FD94	00 04 00 00 01 00 01 00 2F DD 6F FB F6 AE 6E C1	.♦..0.0./Tou<<n+
0014FDA4	66 0B EB C1 B5 4A FC 98 76 3B 66 12 B2 02 07 7E	f[]0-AJAsv;f#0.~
0014FDB4	DF 4C 85 5B 1A F4 8A 78 98 EB DF 85 B3 36 1F 01	■L[]+>0x\$0■[]670
0014FDC4	DE 45 A9 A1 B8 B7 23 83 71 9A FB F5 08 63 2A EF	0Ee[]E#3quü\$éc*
0014FDD4	5F B4 07 EC A8 39 91 15 A9 24 B8 7D EC A3 FA 5A	-iY9L3es\$yü`Z
0014FDE4	45 07 46 68 B9 73 3E 68 16 63 24 C9 70 B4 6B 74	E·Fh[]s>k.c\$[]p[]kt
0014FDF4	EF E8 02 E6 FB 45 08 C0 6E A7 9D 08 76 00 C5 B1	`R0\$üEé`n2L0v0H\$
0014FE04	00 E6 35 09 E8 0B F7 37 FF 8F 96 91 00 4A 36 71	.35°A[]_7 CPL.J6q
0014FE14	BA 73 C0 80 30 01 31 8E 00 FE 14 00 62 61 E2 00	s+C=0iA.[]ba0.
0014FE24	E2 32 F1 42 70 FE 14 00 5A 5E E2 00 01 00 00 00	02~Bp[]_Z^0.0...
0014FE34	08 19 86 00 38 19 86 00 BA 31 F1 42 00 00 00 00	0+6.8+6.[]1~B....
0014FE44	00 00 00 00 00 E0 FD 7F 5C FE 14 00 00 00 00 000x0~[].....
0014FE54	00 00 00 00 3C FE 14 00 09 A6 FF 3E AC FE 14 00<[]_.2 >C[].
0014FE64	60 69 E2 00 BA 8A 06 42 00 00 00 00 7C FE 14 00	!e.S\$B.....![][]
0014FE74	45 3C 39 76 00 E0 FD 7F BC FE 14 00 F5 37 50 77	E<9v.0x0°[]_.37Pw
0014FE84	00 E0 FD 7F B4 CA 1C 77 00 00 00 00 00 00 00 00	.0x0H~Lw.....

In contrast to the previous edition, where one AES key was used for all the files, here a new random key is generated per every file.

As you can see – in example below the randomly generated key was **MRNW9KSC5JRCeT4uJVmI2AOS7JUjPQc6**

```

00111E63 . ADD ESP,0x8
00111E66 . LEA EDI,[LOCAL_78]
00111E6C . MOV BYTE PTR SS:[EBP-0x138],BL
00111E72 . MOV DWORD PTR SS:[EBP-0x137],EAX
00111E78 . MOV DWORD PTR SS:[EBP-0x133],EAX
00111E7E . MOV DWORD PTR SS:[EBP-0x12F],EAX
00111E84 . MOV DWORD PTR SS:[EBP-0x12B],EAX
00111E8A . MOV DWORD PTR SS:[EBP-0x127],EAX
00111E90 . MOV DWORD PTR SS:[EBP-0x123],EAX
00111E96 . MOV DWORD PTR SS:[EBP-0x11F],EAX
00111E9C . MOV WORD PTR SS:[EBP-0x11B],AX
00111EA3 . MOV BYTE PTR SS:[EBP-0x119],AL
0011EA9 . CALL svchosd.00114570 make_random_AES_key
00111EAE . MOV [LOCAL_100],EBX
00111FB4 . CMP [LOCAL_101],EBX
    
```

00114570=svchosd.00114570

Address	Hex dump	ASCII
002AE3F0	4D 52 4E 57 39 4B 53 43 35 4A 52 43 65 54 34 75	MRNW9KSC5JRCeT4u
002AE400	4A 56 6D 49 32 41 4F 53 37 4A 55 6A 50 51 63 36	JUmI2AOS7JUjPQc6
002AE410	74 ED 2A 00 08 00 15 C0 70 E7 2A 00 1B 00 00 00	t?*.[]_.\$°p\$*.+...
002AE420	00 00 F6 76 E4 E4 2A 00 54 E4 2A 00 5E 7C 02 77	..+v[]n[]*.Tn*.^;0w

Then, the key is used in the same way like the previous one – to encrypt 16 byte long chunk with AES ECB mode.

Below – buffer before encryption (fragment of the input is selected on the hex dump – it is a header of a PNG file):

```

001119F0 . LEA ECX, DWORD PTR SS:[ESP+0x10]
001119F4 . MOV DWORD PTR SS:[ESP+0x80], EDX
001119FB . CALL suchosd.001113B0          init
00111A00 . MOV EAX, ECX
00111A02 . PUSH EAX
00111A03 . LEA ESI, DWORD PTR SS:[ESP+0x78]
00111A07 . CALL suchosd.001114D0          Arg1 = 0177F7F9
00111A0C . ADD ESP, 0x4                  encrypt_chunk
00111A0F . LEA EAX, DWORD PTR SS:[ESP+0x30]
00111A13 . MOV ECX, 0x20
00111A18 . JMP SHORT suchosd.00111A20
    
```

ECX=0177F798
EAX=0177F7F9

Address	Hex dump	ASCII
0177F7A8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0177F7B8	4D 52 4E 57 39 4B 53 43 35 4A 52 43 65 54 34 75	MRNW9KSC5JRCeT4u
0177F7C8	4A 56 6D 49 32 41 4F 53 37 4A 55 6A 50 51 63 36	JUmI2A0S7JUjP0c6
0177F7D8	7A B9 D0 38 88 10 90 3E 31 85 29 1B E5 5D B0 13	zif08t>E>1A)+hI!!
0177F7E8	29 61 4F 2D 46 DF 75 13 12 CC 49 8D DA 80 25 2E)a0-Fu!!#I2 rC%.
0177F7F8	00 00 00 00 89 50 4E 47 0D 0A 1A 0A 00 00 00 00PNG..+....
0177F808	49 48 44 52 E9 8E 73 49 1C F8 77 01 45 3C 17 77	IHDRUAsIL°w0E<#w
0177F818	58 4E 60 00 5C F8 77 01 F5 37 03 77 58 4E 60 00	XN'Uw0S7#wx
0177F828	0C F4 5C 74 00 00 00 00 00 00 00 00 58 4E 60 00	.\t.....XN'

The same chunk encrypted (result in bytes -> “55 0F 94 4C B0 98 81 DB F4 57 8A 98 92 2C 09 14”)

```

001119F0 . LEA ECX, DWORD PTR SS:[ESP+0x10]
001119F4 . MOV DWORD PTR SS:[ESP+0x80], EDX
001119FB . CALL suchosd.001113B0          init
00111A00 . MOV EAX, ECX
00111A02 . PUSH EAX
00111A03 . LEA ESI, DWORD PTR SS:[ESP+0x78]
00111A07 . CALL suchosd.001114D0          Arg1 = 0177F7FC
00111A0C . ADD ESP, 0x4                  encrypt_chunk
00111A0F . LEA EAX, DWORD PTR SS:[ESP+0x30]
00111A13 . MOV ECX, 0x20
00111A18 . JMP SHORT suchosd.00111A20
    
```

ESP=0177F784

Address	Hex dump	ASCII
0177F7A8	29 61 4F 2D 46 DF 75 13 12 CC 49 8D DA 80 25 2E)a0-Fu!!#I2 rC%.
0177F7B8	4D 52 4E 57 39 4B 53 43 35 4A 52 43 65 54 34 75	MRNW9KSC5JRCeT4u
0177F7C8	4A 56 6D 49 32 41 4F 53 37 4A 55 6A 50 51 63 36	JUmI2A0S7JUjP0c6
0177F7D8	7A B9 D0 38 88 10 90 3E 31 85 29 1B E5 5D B0 13	zif08t>E>1A)+hI!!
0177F7E8	29 61 4F 2D 46 DF 75 13 12 CC 49 8D DA 80 25 2E)a0-Fu!!#I2 rC%.
0177F7F8	00 00 00 00 55 0F 94 4C B0 98 81 DB F4 57 8A 98U0L#s0#00s
0177F808	92 2C 09 14 E9 8E 73 49 1C F8 77 01 45 3C 17 77	l.,UAsIL°w0E<#w
0177F818	58 4E 60 00 5C F8 77 01 F5 37 03 77 58 4E 60 00	XN'Uw0S7#wx
0177F828	0C F4 5C 74 00 00 00 00 00 00 00 00 58 4E 60 00	.\t.....XN'

After use, the random AES key is RSA encrypted:

```

00112035 . CALL suchosd.001187C0
0011203A . MOV ECX, [ARG.2]             RSA_key
0011203D . ADD ESP, 0xC
00112040 . LEA EDX, [LOCAL.100]
00112046 . PUSH EDX
00112047 . PUSH EDI
00112048 . LEA EAX, [LOCAL.78]
0011204E . PUSH EAX
0011204F . PUSH ECX
00112050 . MOV [LOCAL.100], 0x0
0011205A . CALL suchosd.001145E0
0011205F . PUSH EBX
    
```

Stack SS:[002AE534]=002AF9D8
ECX=002AF9D8

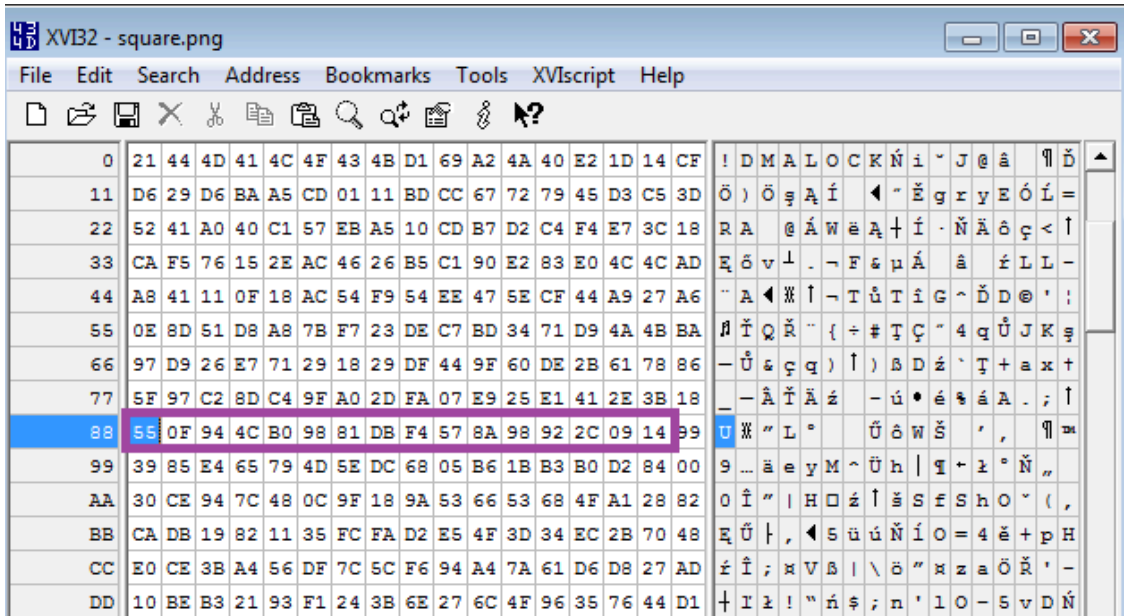
Address	Hex dump	ASCII
002AF9D8	06 02 00 00 00 A4 00 00 52 53 41 31 00 04 00 00	00...A..RSA1.00
002AF9E8	01 00 01 00 2F 0D 6F FB F6 AE 6E C1 66 0B EB C1	0.0./Tou+en+f004
002AF9F8	B5 4A FC 98 76 3B 66 12 B2 02 07 7E DF 4C 85 5B	AJRsv;f#00.~MLA[
002AFA08	1A F4 8A 78 98 EB DF 85 B3 36 1F 01 DE 45 A9 A1	+°xs0°m 6°00Eei
002AFA18	BB 87 23 83 71 9A FB F5 D8 63 2A EF 5F B4 07 EC	ηE#aquúšéc*~iú
002AFA28	A8 39 91 15 A9 24 B8 7D EC A3 FA 5A 45 07 46 68	E9Lšes\$yú'ZE-Fh
002AFA38	B9 73 3E 6B 16 63 24 C9 70 B4 68 74 EF E8 02 E6	ij's>k~cšfipkt'R0S
002AFA48	FB 45 D8 C0 6E A7 90 0B 76 D0 C5 B1 00 E6 35 D9	úEš°nZk0v0H.85°
002AFA58	E8 D8 F7 37 FF 8F 96 91 0D 4A 36 71 BA 73 C0 80	° 7 CPL.J6all's°C
002AFA68	3D 01 31 8E 00 B4 11 00 84 FA 2A 00 05 80 2E 48	=01A.-+4.š.*.š.C.H
002AFA78	90 B4 11 00 C4 FA 2A 00 5A 5E 11 00 01 00 00 00	E+4.-.*.Z^4.0...
002AFA88	08 19 60 00 38 19 60 00 B9 83 2E 48 00 00 00 00	0+š+ar.H...
002AFA98	00 00 00 00 00 60 FD 7F B0 FA 2A 00 00 00 00 00y°0
002AFAB8	00 00 00 00 90 FA 2A 00 8F 59 A0 CC 00 FB 2A 00E*.CVšif.ú*
002AFAB8	60 69 11 00 0D 3C 16 48 00 00 00 00 D0 FA 2A 00	*H...<.H....0.*.

and then, appended to the beginning of the AES encrypted file (just after the “!DMALOCK” signature):

```

0040202B push    1024             ; size_t
00402030 mov     edi, eax
00402032 push    0               ; int
00402034 push    edi             ; void *
00402035 call   _memset
0040203A mov     ecx, [ebp+RSA_key]
0040203D add     esp, 0Ch
00402040 lea    edx, [ebp+var_190]
00402046 push    edx             ; int
00402047 push    edi             ; void *
00402048 lea    eax, [ebp+generated_AES_key]
0040204E push    eax             ; int
0040204F push    ecx             ; int
00402050 mov     [ebp+var_190], 0
0040205A call   encrypt_key_with_RSA
0040205F push    ebx             ; FILE *
00402060 push    8               ; size_t
00402062 push    1               ; size_t
00402064 push    offset aDmalock ; "!DMALOCK"
00402069 call   fwrite          ; write 'magic' prefix
0040206E mov     edx, [ebp+var_190]
00402074 push    ebx             ; FILE *
00402075 push    edx             ; size_t
00402076 push    1               ; size_t
00402078 push    edi             ; void *
00402079 call   fwrite          ; write encrypted AES key
    
```

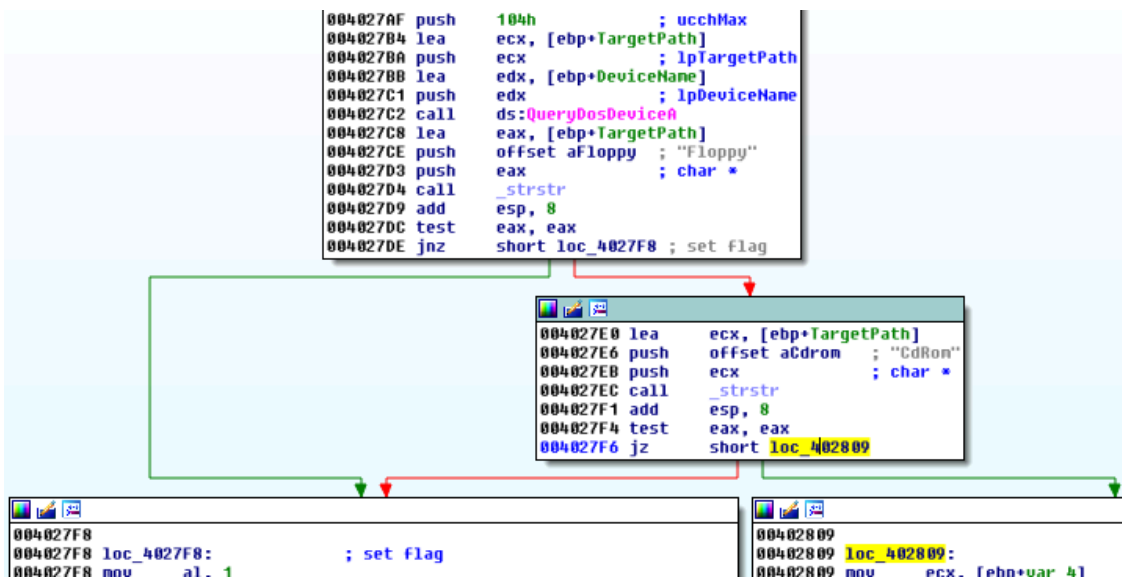
We can see that now the AES encrypted content starts with offset 0x88 (compare the selected part with the above example showing AES encryption result):



What is attacked?

As previous, attacked are logical disks as well as network shares.

This sample introduced also check against Floppy and CD using [QueryDosDeviceA](#) (floppy and CD are skipped):



Like in the previous version, skipped are some predefined folders:

```

00402633 sub     esp, zcn
00402636 push   esi
00402637 mov     [ebp+var_2C], offset aWindows ; "\\Windows\\"
0040263E mov     [ebp+var_28], offset aWindows_0 ; "\\WINDOWS\\"
00402645 mov     [ebp+var_24], offset aProgramFiles ; "\\Program Files\\"
0040264C mov     [ebp+var_20], offset aProgramFilesX8 ; "\\Program Files (x86)\\"
00402653 mov     [ebp+var_1C], offset aGames ; "Games"
0040265A mov     [ebp+var_18], offset aTemp ; "\\Temp"
00402661 mov     [ebp+var_14], offset aSamplePictures ; "\\Sample Pictures"
00402668 mov     [ebp+var_10], offset aSampleMusic ; "\\Sample Music"
0040266F mov     [ebp+var_C], offset aCache ; "\\cache"
00402676 mov     [ebp+var_8], offset aCache_0 ; "\\Cache"
0040267D xor     esi, esi
0040267E nop
    
```

...and file extensions:

```

004026B7 mov     [ebp+var_30], offset a_exe ; ".exe"
004026BE mov     [ebp+var_2C], offset a_nsi ; ".nsi"
004026C5 mov     [ebp+var_28], offset a_dll ; ".dll"
004026CC mov     [ebp+var_24], offset a_pif ; ".pif"
004026D3 mov     [ebp+var_20], offset a_scr ; ".scr"
004026DA mov     [ebp+var_1C], offset a_sys ; ".sys"
004026E1 mov     [ebp+var_18], offset a_nsp ; ".nsp"
004026E8 mov     [ebp+var_14], offset a_com ; ".com"
004026EF mov     [ebp+var_10], offset a_lnk ; ".lnk"
004026F6 mov     [ebp+var_C], offset a_hta ; ".hta"
004026FD mov     [ebp+var_8], offset a_cpl ; ".cpl"
00402704 mov     [ebp+var_4], offset a_msc ; ".msc"
0040270B xor     esi, esi
    
```

Conclusion

The author of this malware, despite appearing inexperienced in programming, seems to be very determined to gradually improve the quality of the product. The disparity between the quality of the first edition, second (described in [the previous article](#)) and the third (current) is significant. We will keep eye on the evolution of this malware family and provide you with updates and possible tips on dealing with this threat.

About the author

Unpacks malware with as much joy as a kid unpacking candies.

Source: <https://blog.malwarebytes.com/threat-analysis/2016/02/dma-locker-strikes-back/>