Challenges and Approaches of Cracking Ransomware

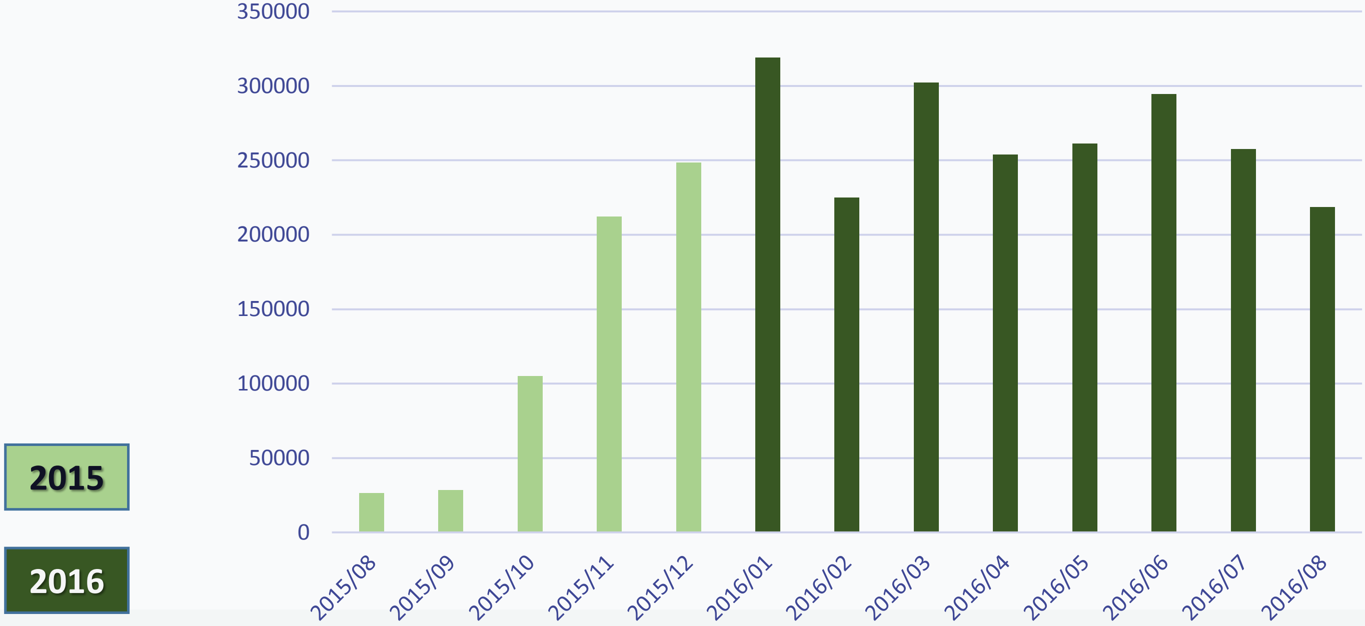
Hasherezade (@hasherezade) - malware analyst, technical blogger

Malwarebytes

Agenda

1. Short background
2. Hunting weak points - tips and tricks
3. My experience on real-life examples (7ev3n, Petya, DMALocker, Chimera)

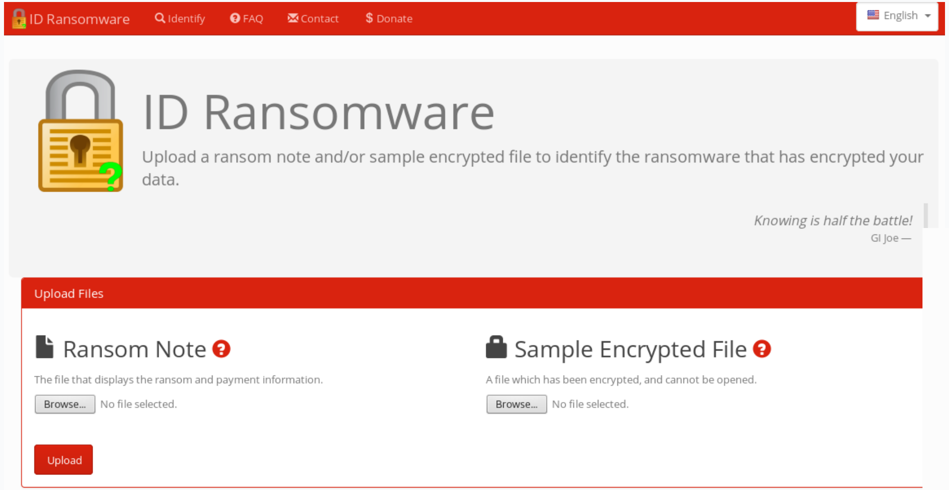
Ransomware - trends (2015-2016)



Ransomware detections per month
(source: Malwarebytes telemetry)

Malwarebytes

Ransomware - varieties



ID Ransomware

Upload a ransom note and/or sample encrypted file to identify the ransomware that has encrypted your data.

Knowing is half the battle!
GI Joe —

Upload Files

Ransom Note

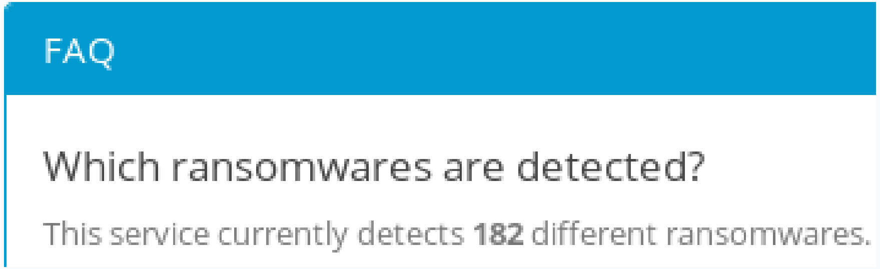
The file that displays the ransom and payment information.

No file selected.

Sample Encrypted File

A file which has been encrypted, and cannot be opened.

No file selected.



FAQ

Which ransomwares are detected?

This service currently detects **182** different ransomwares.

182 types and counting...

https://id-ransomware.malwarehunterteam.com

Malwarebytes

Ransomware – encryption algorithms

- Most popular: **AES + RSA**
  - **AES** to encrypt files, **RSA** to encrypt **random AES key**
- Other observed:
  - AES (only), RSA (only), Salsa20, ChaCha, TripleDES, XTEA, XOR, custom...

http://www.nyxbone.com/malware/RansomwareOverview.html

Ransomware - successful recovery attempts

- 7ev3n, XORist, Bart – weak **encryption algorithm**
- Petya – **mistakes** in cryptography **implementation**
- DMA Locker, CryptXXX – weak **key generator**
- Cerber – **server-side** vulnerability
- Chimera – **leaked** keys
- ...

Ransomware - successful recovery attempts

- Tesla Crypt – **failed to protect AES keys** – weak keys for the ECDH (Elliptic Curve Diffie-Hellman) algorithm [2][3][4]
- Torrent Locker (2014) – **failed to initialize AES CTR** properly (invalid initalization vector - introduced a possibility of known-plaintext attack) [1]
- And many more...

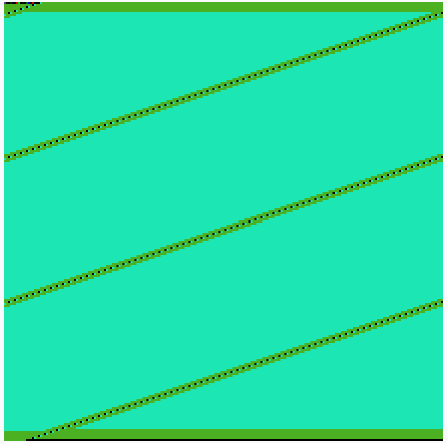
How to find the weak points?

1. Identifying the encryption algorithm
    - Visualization is your friend!
2. Checking the implementation correctness
3. Identifying the key generator
    - Is the key unique for each file?
4. Identifying how the key is stored

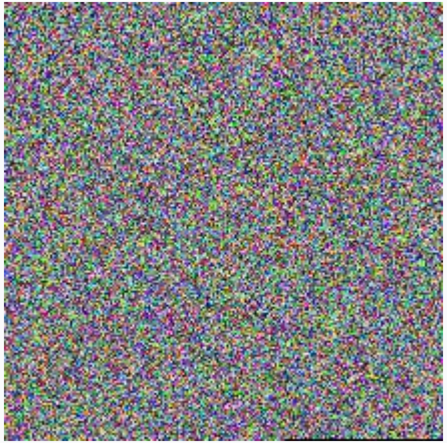
Identifying the encryption algorithm

What can the visualization tell us?

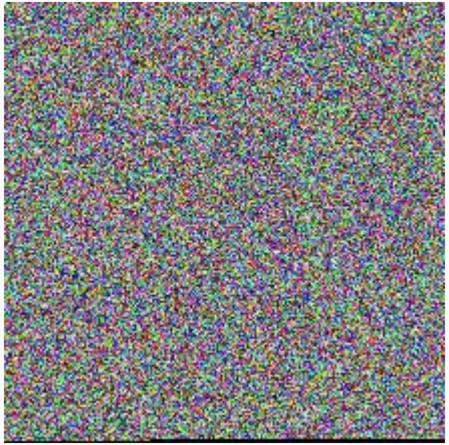
Original file



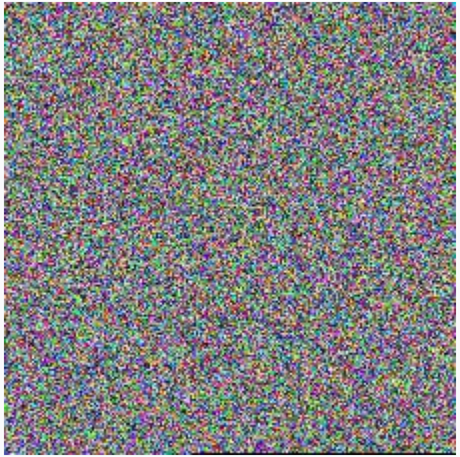
Encrypted by Cerber



Encrypted by Locky



Encrypted by zCrypt



High entropy, no patterns visible:

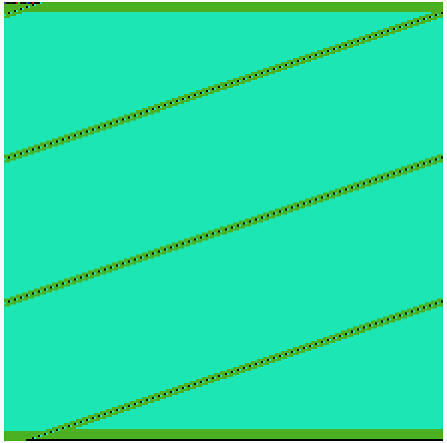
often: stream ciphers/chained blocks (i.e. AES CBC), rarely: RSA

https://github.com/hasherezade/crypto_utils/blob/master/file2png.py

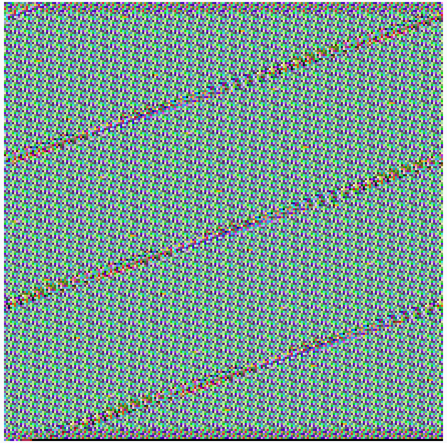
Identifying the encryption algorithm

What the visualization can tell us?

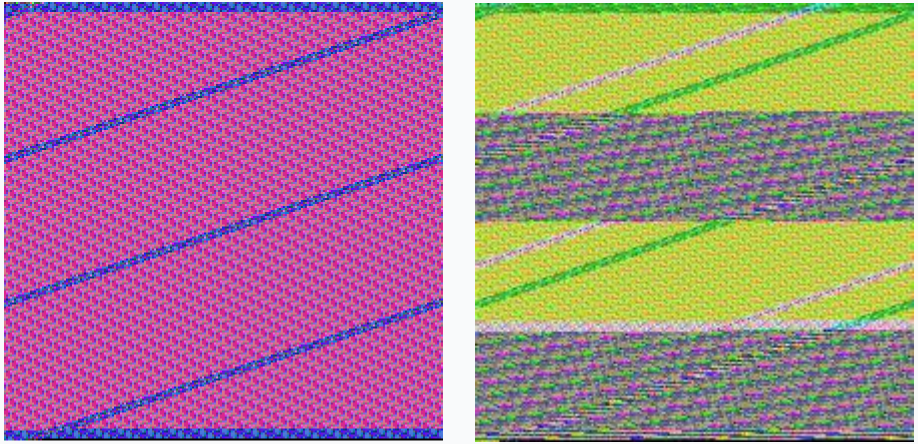
Original file



Encrypted by DMA Locker



Encrypted by 7ev3n



Lower entropy, patterns visible:

block ciphers (i.e. AES ECB), possible: XOR & XOR-based

https://github.com/hasherezade/crypto_utils/blob/master/file2png.py

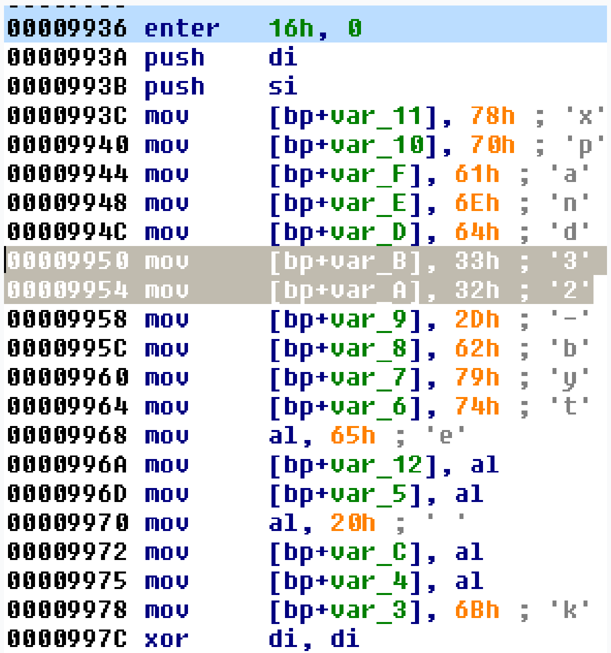
Identifying the encryption algorithm

Find the file encryption function:

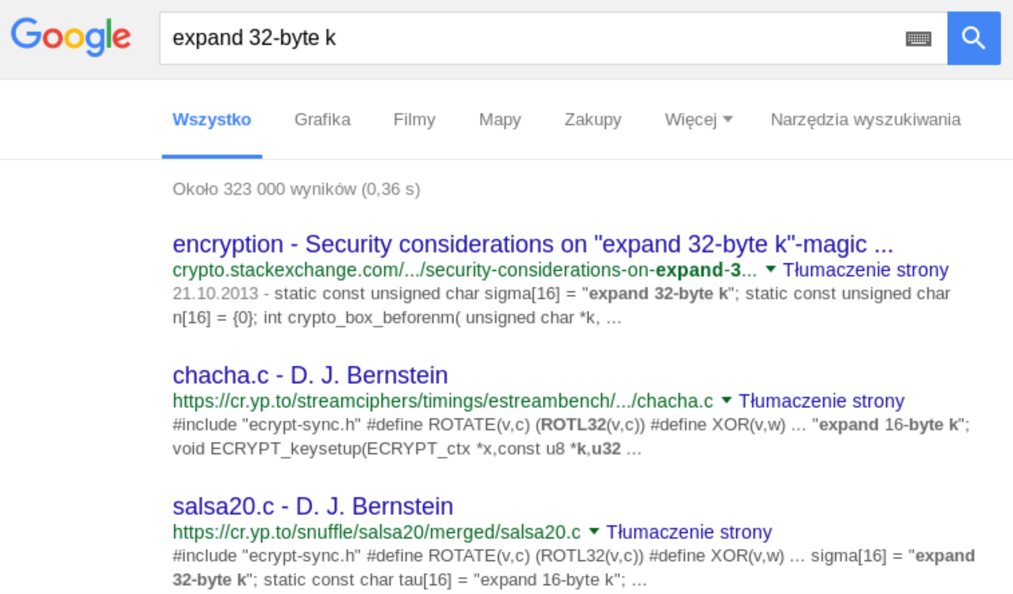
1. Where the content is **read** from the file
2. Where the content is **written** to the file
3. Search the call to the encryption function in between 1 and 2!
4. Search from where the encryption key comes
5. Search how the key is stored after use

Identifying the encryption algorithm

Searching in the code: typical constants, keywords...



```
00009936 enter    16h, 0
0000993A push     di
0000993B push     si
0000993C mov      [bp+var_11], 78h ; 'x'
00009940 mov      [bp+var_10], 70h ; 'p'
00009944 mov      [bp+var_F], 61h ; 'a'
00009948 mov      [bp+var_E], 6Eh ; 'n'
0000994C mov      [bp+var_D], 64h ; 'd'
00009950 mov      [bp+var_B], 33h ; '3'
00009954 mov      [bp+var_A], 32h ; '2'
00009958 mov      [bp+var_9], 2Dh ; '-'
0000995C mov      [bp+var_8], 62h ; 'b'
00009960 mov      [bp+var_7], 79h ; 'y'
00009964 mov      [bp+var_6], 74h ; 't'
00009968 mov      al, 65h ; 'e'
0000996A mov      [bp+var_12], al
0000996D mov      [bp+var_5], al
00009970 mov      al, 20h ; ' '
00009972 mov      [bp+var_C], al
00009975 mov      [bp+var_4], al
00009978 mov      [bp+var_3], 6Bh ; 'k'
0000997C xor      di, di
```



Około 323 000 wyników (0,36 s)

encryption - Security considerations on "expand 32-byte k"-magic ...
crypto.stackexchange.com/.../security-considerations-on-**expand**-3... ▾ Tłumaczenie strony
21.10.2013 - static const unsigned char sigma[16] = "**expand 32-byte k**"; static const unsigned char n[16] = {0}; int crypto_box_beforenm(unsigned char *k, ...

chacha.c - D. J. Bernstein
https://cr.yp.to/streamciphers/timings/estreambench/.../chacha.c ▾ Tłumaczenie strony
#include "ecrypt-sync.h" #define ROTATE(v,c) (**ROTL32**(v,c)) #define XOR(v,w) ... "**expand** 16-**byte k**"; void ECRYPT_keysetup(ECRYPT_ctx *x,const u8 ***k,u32** ...

salsa20.c - D. J. Bernstein
https://cr.yp.to/snuffle/salsa20/merged/salsa20.c ▾ Tłumaczenie strony
#include "ecrypt-sync.h" #define ROTATE(v,c) (ROTL32(v,c)) #define XOR(v,w) ... sigma[16] = "**expand 32-byte k**"; static const char tau[16] = "expand 16-byte k"; ...

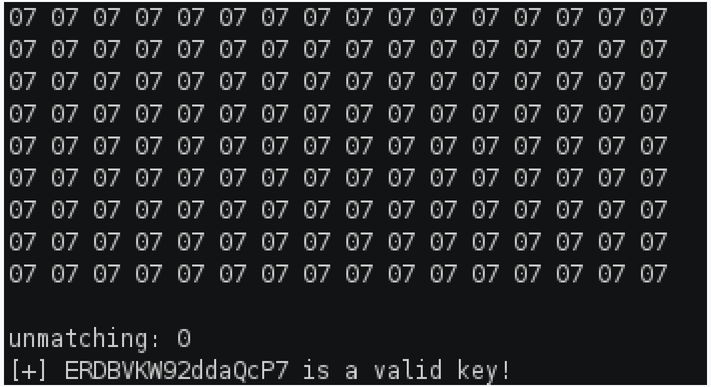
Malwarebytes

Checking the correctness of implementation

- Fast check:
  - Dump the key from malware’s memory
  - Save the file encrypted by the malware
  - Encrypt the original file by a valid implementation of the identified algorithm
  - Compare the results

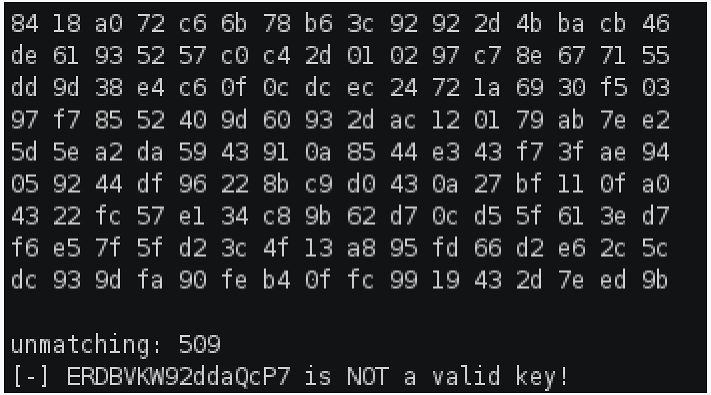
Checking the correctness of implementation

Comparing the output of given algorithm vs the valid one can give us hints!



```
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07
07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07

unmatching: 0
[+] ERDBVKW92ddaQcP7 is a valid key!
```



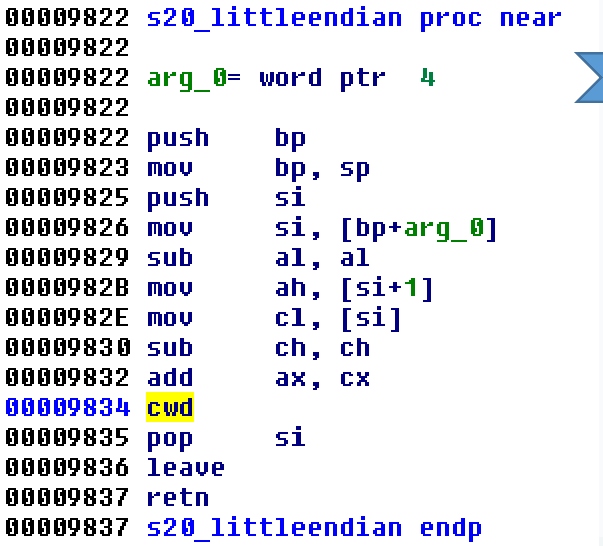
```
84 18 a0 72 c6 6b 78 b6 3c 92 92 2d 4b ba cb 46
de 61 93 52 57 c0 c4 2d 01 02 97 c7 8e 67 71 55
dd 9d 38 e4 c6 0f 0c dc ec 24 72 1a 69 30 f5 03
97 f7 85 52 40 9d 60 93 2d ac 12 01 79 ab 7e e2
5d 5e a2 da 59 43 91 0a 85 44 e3 43 f7 3f ae 94
05 92 44 df 96 22 8b c9 d0 43 0a 27 bf 11 0f a0
43 22 fc 57 e1 34 c8 9b 62 d7 0c d5 5f 61 3e d7
f6 e5 7f 5f d2 3c 4f 13 a8 95 fd 66 d2 e6 2c 5c
dc 93 9d fa 90 fe b4 0f fc 99 19 43 2d 7e ed 9b

unmatching: 509
[-] ERDBVKW92ddaQcP7 is NOT a valid key!
```

https://asciinema.org/a/87388

Checking the implementation correctnes

- Analysis of the algorithm implementation and comparing with the correct code



```
00009822 s20_littleendian proc near
00009822
00009822 arg_0= word ptr  4
00009822
00009822 push    bp
00009823 mov     bp, sp
00009825 push    si
00009826 mov     si, [bp+arg_0]
00009829 sub     al, al
0000982B mov     ah, [si+1]
0000982E mov     cl, [si]
00009830 sub     ch, ch
00009832 add     ax, cx
00009834 cwd
00009835 pop     si
00009836 leave
00009837 retn
00009837 s20_littleendian endp
```

```
static int16_t s20_littleendian(uint8_t *b)
{
  return b[0] +
         (b[1] << 8);
//...
}
```

Versus - the same function from the valid Salsa20:

```
static uint32_t s20_littleendian(uint8_t *b)
{
  return b[0] +
         (b[1] << 8)} +
         (b[2] << 16) +
         (b[3] << 24);
}
```

Identifying the key generator

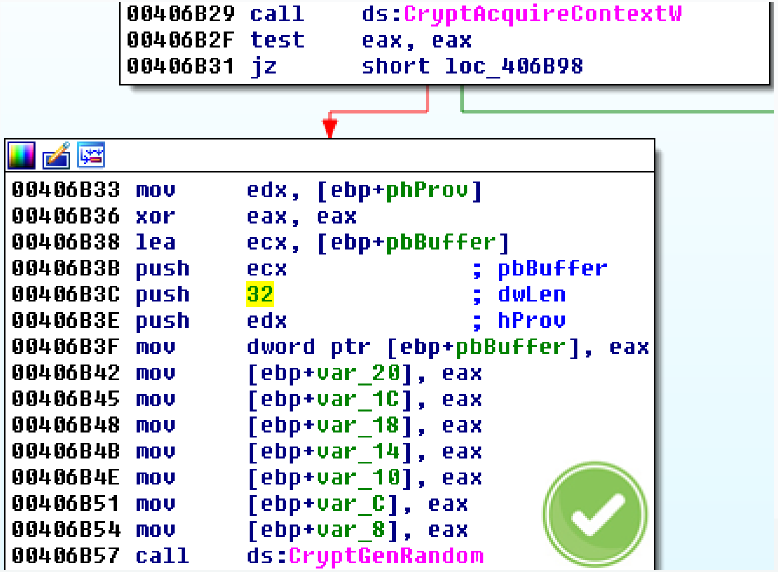
- Is the key unique for each file?
- Make a simple test:
  - let the ransomware encrypt two identical files
  - is the output same?

Identifying the key generator

- What is used for code generation?
  - Hardware identifiers?
  - Random generator? Weak or strong?

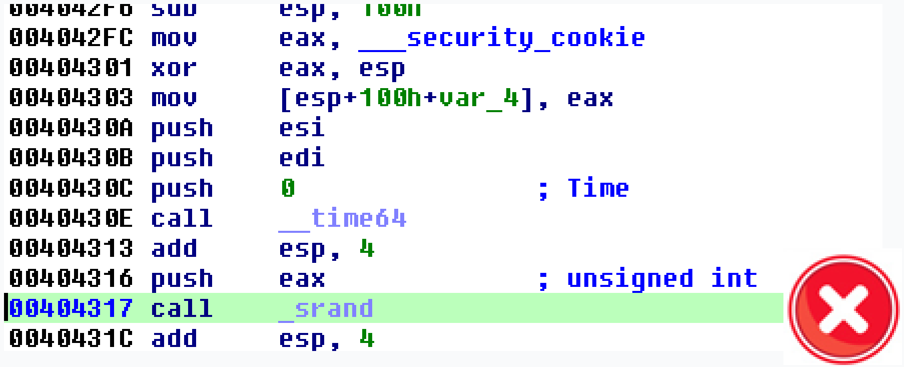
Random generator: weak or strong?

- Strong: CryptGenRandom, RtlGenRandom (SystemFunction036)
- Weak: i.e. rand() initialized by the current time



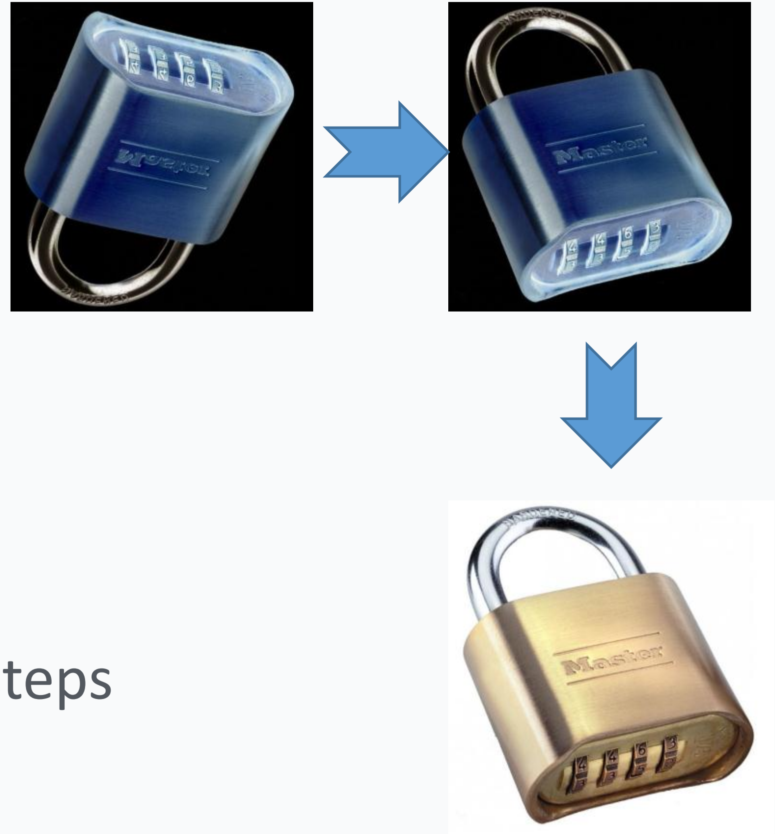
```
00406B29 call    ds:CryptAcquireContextW
00406B2F test    eax, eax
00406B31 jz      short loc_406B98
```

```
00406B33 mov     edx, [ebp+phProv]
00406B36 xor     eax, eax
00406B38 lea     ecx, [ebp+pbBuffer]
00406B3B push    ecx                     ; pbBuffer
00406B3C push    32                      ; dwLen
00406B3E push    edx                     ; hProv
00406B3F mov     dword ptr [ebp+pbBuffer], eax
00406B42 mov     [ebp+var_20], eax
00406B45 mov     [ebp+var_1C], eax
00406B48 mov     [ebp+var_18], eax
00406B4B mov     [ebp+var_14], eax
00406B4E mov     [ebp+var_10], eax
00406B51 mov     [ebp+var_C], eax
00406B54 mov     [ebp+var_8], eax
00406B57 call    ds:CryptGenRandom
```



```
004042F6 sub     esp, 100h
004042FC mov     eax, ___security_cookie
00404301 xor     eax, esp
00404303 mov     [esp+100h+var_4], eax
0040430A push    esi
0040430B push    edi
0040430C push    0               ; Time
0040430E call    __time64
00404313 add     esp, 4
00404316 push    eax             ; unsigned int
00404317 call    _srand
0040431C add     esp, 4
```

Exploiting the weak algorithm: Example – 7ev3n



Challenge:

- reverse the custom algorithm

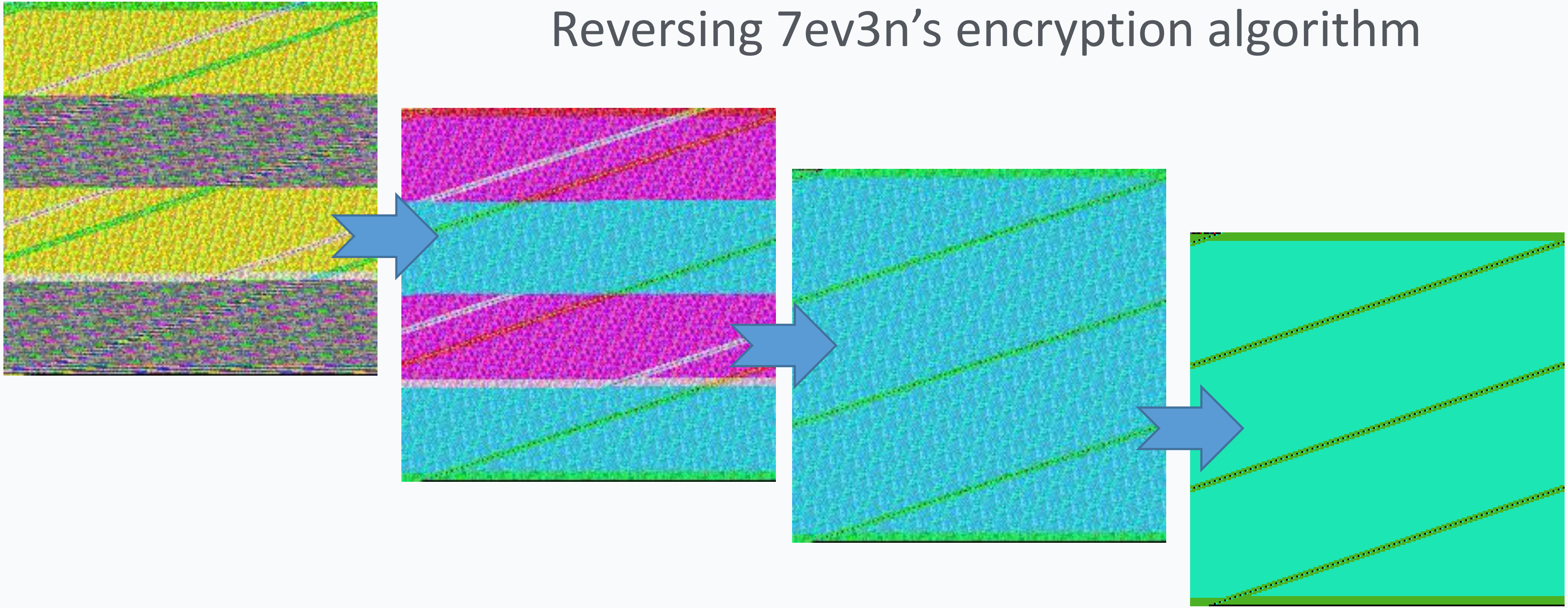
Approach:

- Analyze the code and reverse the steps
- Implement the decoder

https://blog.malwarebytes.com/threat-analysis/2016/05/7ev3n-ransomware/
https://github.com/hasherezade/malware_analysis/tree/master/7ev3n

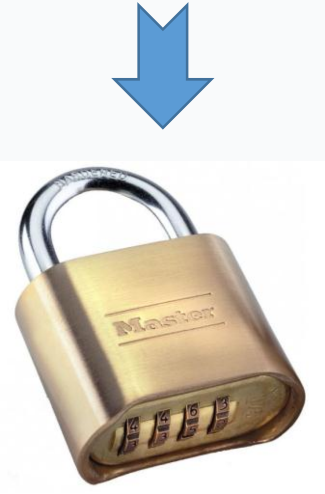
Exploiting the weak algorithm: Example – 7ev3n

Reversing 7ev3n's encryption algorithm



https://blog.malwarebytes.com/threat-analysis/2016/05/7ev3n-ransomware/
https://github.com/hasherezade/malware_analysis/tree/master/7ev3n

Exploiting the weak algorithm: Example – 7ev3n



Difficulties:

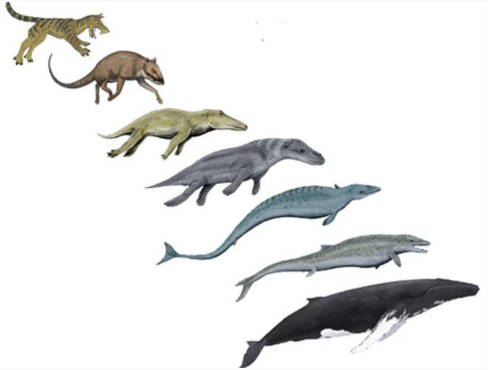
- Many variants of the custom algorithm (no generic solution)
- Additional data required (i.e. path to the file)

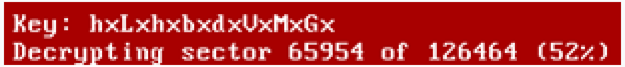
https://blog.malwarebytes.com/threat-analysis/2016/05/7ev3n-ransomware/
https://github.com/hasherezade/malware_analysis/tree/master/7ev3n

Exploiting the implementation vulnerability: Example – Petya

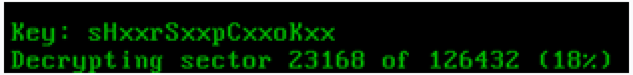
Challenge:

- find the key (8 characters from 54 character set)





```
Key: hxLxhxbxdxVxMxGx
Decrypting sector 65954 of 126464 (52%)
```



```
Key: sHxxrSxxpCxxoKxx
Decrypting sector 23168 of 126432 (18%)
```

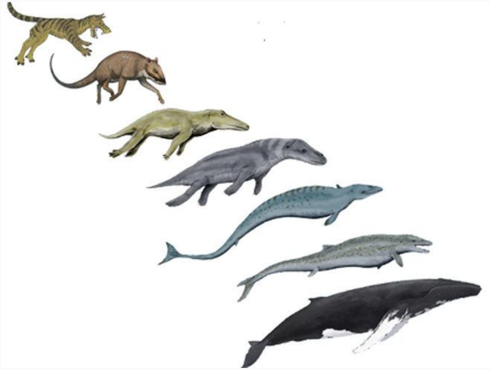
hxLxhxbxdxVxMxGx   sHxxrSxxpCxxoKxx

Exploiting the implementation vulnerability: Example – Petya

Approach:

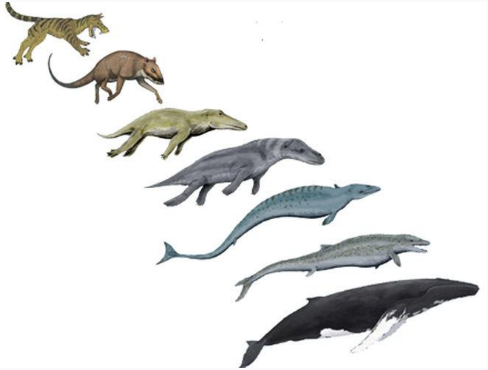
- Reimplement the corrupt version of Salsa20
- Search the key space (using dumped validation buffer and nonce)
- Possible to bruteforce

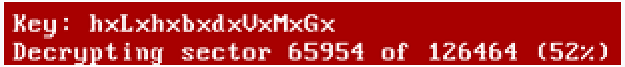
  (54 ^ 8 = 72301961339136)



Exploiting the implementation vulnerability: Example – Petya

- Interesting observation by @leo_and_stone (only in Red Petya):
  - Due to the specifics of the vulnerability, **we can measure the progress in cracking**
  - **Genetic algorithms can be used, to make the correct key “evolve”**

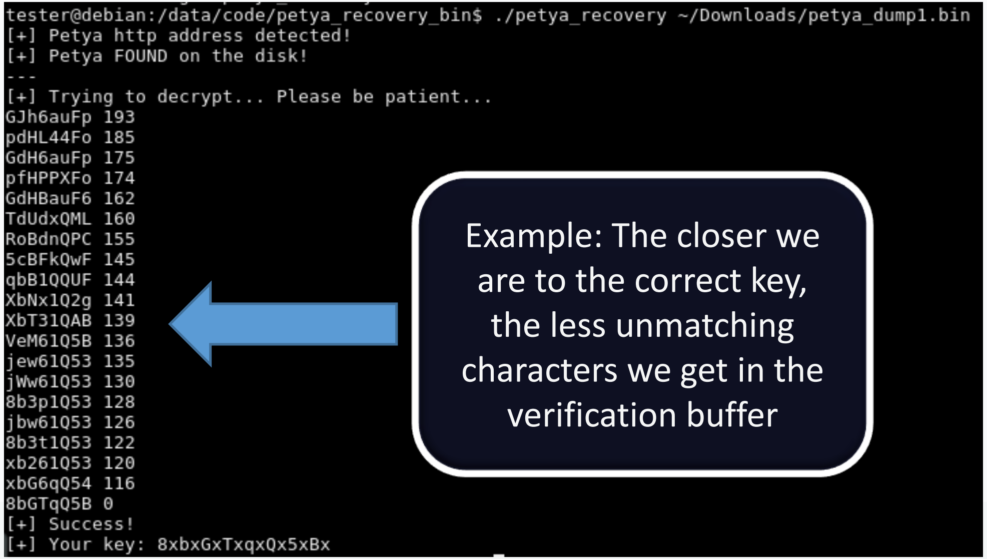




```
Key: hxLxhxbxdxVxMxGx
Decrypting sector 65954 of 126464 (52%)
```

Exploiting the implementation vulnerability: Example – Petya

- When the genetic approach works?
  - Only in cases when we can measure the progress!



```
tester@debian:/data/code/petya_recovery_bin$ ./petya_recovery ~/Downloads/petya_dump1.bin
[+] Petya http address detected!
[+] Petya FOUND on the disk!
---
[+] Trying to decrypt... Please be patient...
GJh6auFp 193
pdHL44Fo 185
GdH6auFp 175
pfHPPXFo 174
GdHBauF6 162
TdUdxQML 160
RoBdnQPC 155
5cBFkQwF 145
qbB1QQUF 144
XbNx1Q2g 141
XbT31QAB 139
VeM61Q5B 136
jew61Q53 135
jWw61Q53 130
8b3p1Q53 128
jbw61Q53 126
8b3t1Q53 122
xb261Q53 120
xbG6qQ54 116
8bGTqQ5B 0
[+] Success!
[+] Your key: 8xbxGxTxqxQx5xBx
```

Example: The closer we are to the correct key, the less unmatching characters we get in the verification buffer

Demo of Genetic Algorithms applied:

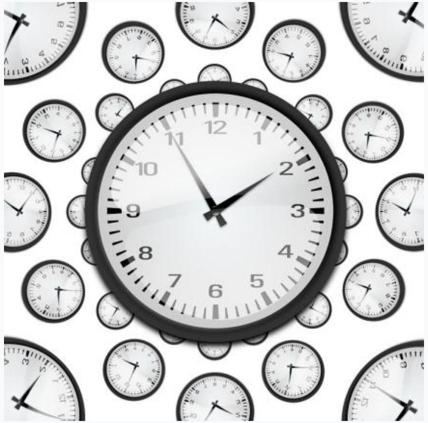
1) Red Petya :
- https://asciinema.org/a/87075

2) Green Petya :
- https://asciinema.org/a/87077

Exploiting the weak key generator: Example – DMA Locker

Challenge:

- Find the seed (**start time**), used to initialize **rand()**
- Then, find a correct key for each file

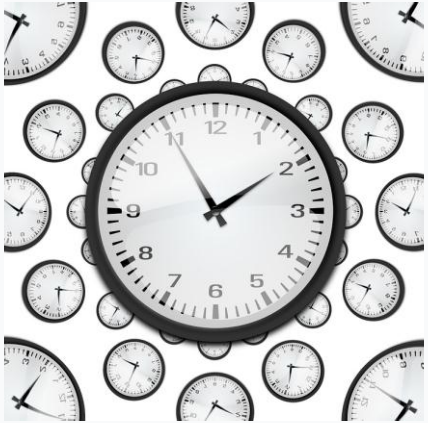


https://github.com/hasherezade/dma_unlocker

Exploiting the weak key generator: Example – DMA Locker

Approach:

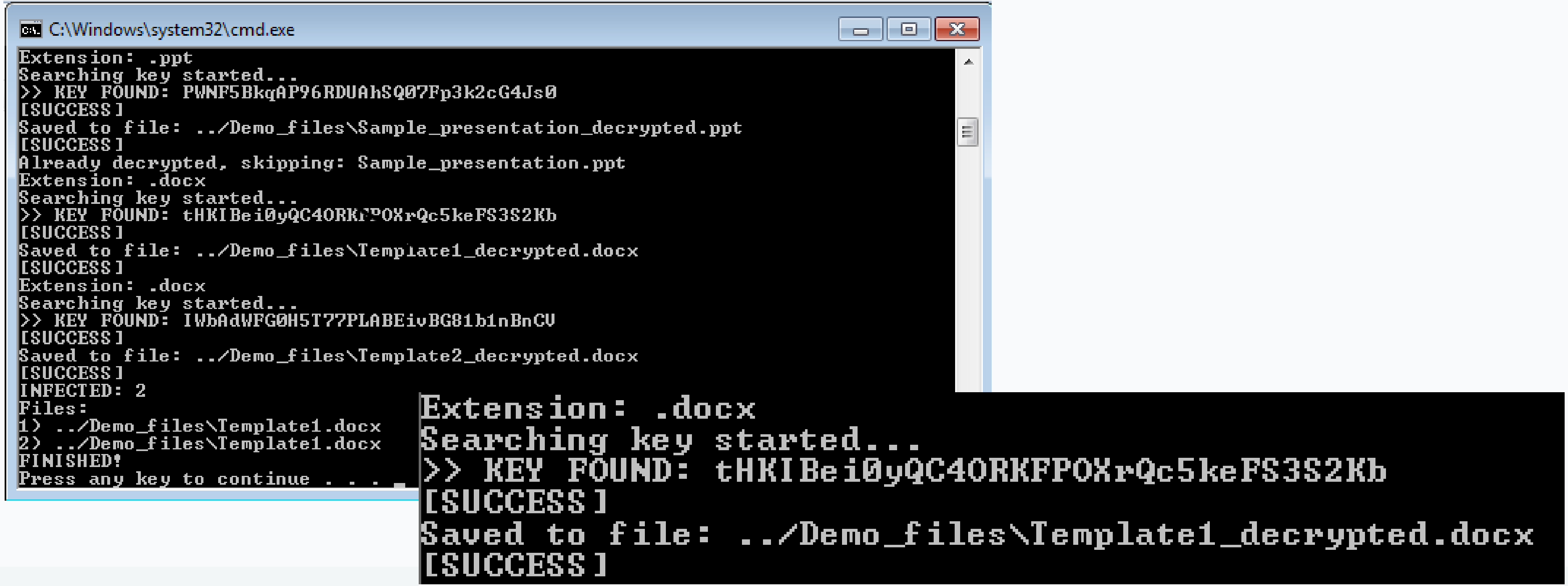
- approximate the **timestamp** by knowing date of the ransom note and/or file modification timestamp
- Validate the key by header typical for file format



https://github.com/hasherezade/dma_unlocker

Exploiting the weak key generator: Example – DMA Locker

DMA Unlocker



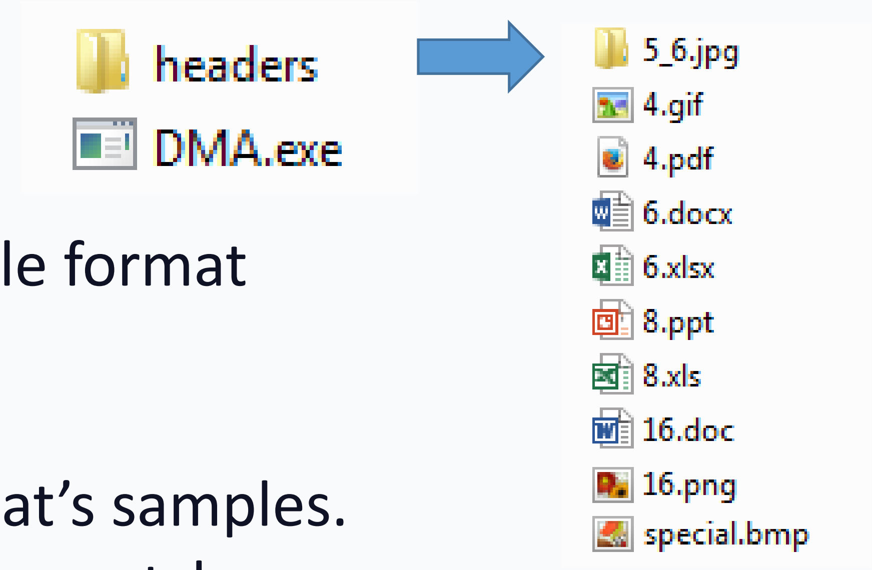
```
Extension: .ppt
Searching key started...
>> KEY FOUND: PWNF5BkqAP96RDUAhSQ07Fp3k2cG4Js0
[SUCCESS]
Saved to file: ../Demo_files\Sample_presentation_decrypted.ppt
[SUCCESS]
Already decrypted, skipping: Sample_presentation.ppt
Extension: .docx
Searching key started...
>> KEY FOUND: tHKIBei0yQC4ORKFPOXrQc5keFS3S2Kb
[SUCCESS]
Saved to file: ../Demo_files\Template1_decrypted.docx
[SUCCESS]
Extension: .docx
Searching key started...
>> KEY FOUND: IWbAdWFG0H5T77PLABEivBG81b1nBnCV
[SUCCESS]
Saved to file: ../Demo_files\Template2_decrypted.docx
[SUCCESS]
INFECTED: 2
Files:
1) ../Demo_files\Template1.docx
2) ../Demo_files\Template1.docx
FINISHED!
Press any key to continue . . .
```

```
Extension: .docx
Searching key started...
>> KEY FOUND: tHKIBei0yQC4ORKFPOXrQc5keFS3S2Kb
[SUCCESS]
Saved to file: ../Demo_files\Template1_decrypted.docx
[SUCCESS]
```

https://github.com/hasherezade/dma_unlocker

Exploiting the weak key generator: Example – DMA Locker

DMA Unlocker



headers
DMA.exe

5_6.jpg
4.gif
4.pdf
6.docx
6.xlsx
8.ppt
8.xls
16.doc
16.png
special.bmp

- **Challenge:**

easy adding support for a new file format

- **Solution:**

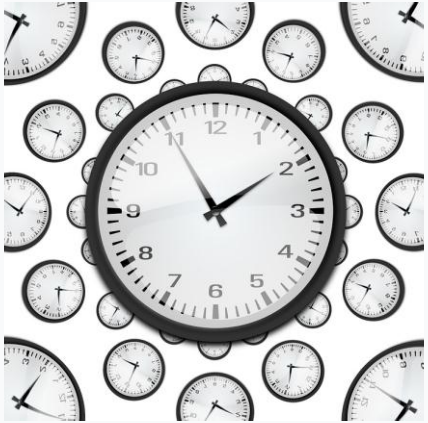
Make a folder that is set of format’s samples.
File name is a **number of bytes** to match.
Some formats needs to be handled in a special way...

https://github.com/hasherezade/dma_unlocker

Exploiting the weak key generator: Example – DMA Locker

Difficulties:

- Some file types are hard to validate
- Finding one seed is not enough



https://github.com/hasherezade/dma_unlocker

Making use of the leaked keys: Example – Chimera

Challenge:

- **find the proper key** for the particular victim

https://blog.malwarebytes.com/cybercrime/2016/08/decrypting-chimera-ransomware/

Making use of the leaked keys: Example – Chimera

Approach:

- Use/implement the decryption algorithm
- Make a "dictionary" attack on the encrypted file (using as a dictionary set of leaked keys)

https://blog.malwarebytes.com/cybercrime/2016/08/decrypting-chimera-ransomware/

Conclusions

- Cryptography is difficult: multiple places where the implementation can go wrong
- Some people still ignore the advice to not roll own crypto
- Ransomware authors keep improving their products, so the decryptors have a short life span...
- The most important is prevention

Additional material

- [1] http://digital-forensics.sans.org/blog/2014/09/09/torrentlocker-unlocked
- [2] http://www.bleepingcomputer.com/news/security/teslacrypt-decrypted-flaw-in-teslacrypt-allows-victims-to-recover-their-files/
- [3] http://blog.talosintel.com/2016/03/teslacrypt-301-tales-from-crypto.html
- [4] https://github.com/Googulator/TeslaCrack

Questions? Remarks?

Read more:

- https://blog.malwarebytes.com/?s=ransomware
- https://hshrzd.wordpress.com/category/malware-decryptor/

Thank You!