Linux.Encoder.0 technical writeup: a story about light-weight cryptanalysis and blind reverse engineering

Radu Caragea
Bitdefender

November 17, 2015

Why Linux.Encoder.0 you ask? Since the recent ransomware making the news is Linux.Encoder.1 we should start with that first.

1 Linux.Encoder.1 background

This ransomware prides itself on its encryption:

```
Your personal files are encrypted!
Encryption was produced using a unique public key
RSA-2048 generated for this computer.
```

Analyzing the binary, the behaviour becomes clear:

- a few priority directories are encrypted first
- then all other directories get encrypted too

How exactly is encryption carried out using the RSA key?

The routine that encrypts one file first obtains file metadata:

- size
- permissions

A new file with the extension ".encrypted" is created and a special header is inserted at the beginning of the file to allow decryption. This header contains:

- 4 bytes holding the old permissions
- 4 bytes holding a size (`enc_key_sz`)
- `enc_key_sz` bytes holding the encrypted key that encrypts the content. More on this later
- 16 bytes used as an IV in the encryption

A keen reader will observe that since the sample does not store ownership information (uid/gid) the encryption/decryption process is actually not completely lossless.

An example of a file after encryption follows:

```
# hexdump -Cv photo_01.jpg.encrypted
00000000  a4 81 00 00 00 01 00 00  61 83 01 ba b4 91 30 a5  |........a.....0.|
00000010  02 98 31 fd 66 54 44 6d  13 cd e6 a3 65 59 9e d4  |..1.fTDm....eY..|
00000020  8b b3 f7 39 3e 8b c4 4b  dd 26 c0 22 6e cf bb 7c  |...9>..K.&."n..||
00000030  89 96 4a 09 16 00 8d 89  67 a0 5d d4 f8 f5 f4 d8  |..J.....g.].....|
00000040  5a 6b f1 ab 07 64 4d 43  f1 9d 48 ec 30 c1 36 48  |Zk...dMC..H.0.6H|
00000050  f7 50 d0 10 e1 4d 20 dd  ab c8 7f 10 97 96 14 df  |.P...M .........|
00000060  47 82 5b ed 5d 1f a5 50  e8 1b 8a dc fd 71 3b b1  |G.[.]..P.....q;.|
00000070  73 77 ec 7d 9c d7 7b e5  2e 5e b9 6d 4e 94 70 52  |sw.}..{..^.mN.pR|
00000080  31 2d fe d8 ff 56 c1 59  26 97 02 94 5f 94 ac b8  |1-...V.Y&..._...|
00000090  aa 41 01 6d 10 49 db ee  e9 b7 13 14 d1 7b 61 a1  |.A.m.I.......{a.|
000000a0  c8 d9 68 38 d0 94 66 6a  81 c0 d8 9b 35 25 28 77  |..h8..fj....5%(w|
000000b0  8a 04 3f 94 b4 14 21 71  1d af 46 ca bc 0d bc 45  |..?...!q..F....E|
000000c0  51 c7 ef a1 83 7a 86 c3  c4 b9 c6 b3 10 43 f4 04  |Q....z.......C..|
000000d0  bb 09 26 7c 21 9c 54 55  14 4c 61 e2 2d 99 0c 92  |..&|!.TU.La.-...|
000000e0  2b be 86 c3 12 ee 3a 92  92 ab 57 3a 2b f4 75 15  |+.....:...W:+.u.|
000000f0  05 37 a5 c1 87 3c a2 c5  7f c5 d9 cd ce 31 b6 01  |.7...<.......1..|
00000100  1a 89 58 08 11 86 8d 71  80 b4 d3 0c ff 26 cd a2  |..X....q.....&..|
00000110  80 6c f5 98 a3 2c ce 9e  6b a0 45 93 27 05 8a 3c  |.l...,..k.E.'..<|
00000120  33 47 cf de e2 0e 41 ae  5f 8e 2d c8 ba e8 a9 cb  |3G....A._.-.....|
00000130  64 ca d7 45 17 d7 f9 41  e9 ad 00 3a 06 83 01 92  |d..E...A...:....|
```

Since asymmetric encryption algorithms are overwhelmingly slower than symmetric algorithms, the traditional solution is a hybrid approach:

- pick a random symmetric key to encrypt a large input with symmetric encryption (in our case AES)
- encrypt only the symmetric key with the public RSA key.

In theory this should be watertight. In practice, though, we have to deal with computers which are, of course, deterministic. Generating random numbers is not easy thing, and many people get it wrong like, luckily for us, in this case.

How are the IV and the key generated? Inspecting the pseudocode in IDA gives us enough information:

```
[...]
// IV generation
do
        IV[idx++] = rand();
while ( idx != 0x10 );

//AES key generation
 char alphabet[] = "abcdefghijklmnopqrstuvwxyz\
        ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,.-#'?!";
 [...]
```

```
 sz = 16;
while ( i < sz )
        key[i++] = alphabet[rand() % 69];
[...]
```

As readers on /r/netsec pointed out, it bears a striking resemblance to an "open-source ransomware": `https://github.com/utkusen/hidden-tear/blob/master/hidden-tear/hidden-tear/Form1.cs#L107`

Using rand() in this case is an arguably borderline safe use of random number generation. However, computers being deterministic, what would prevent a person from repeating the same steps and obtaining the same results for rand(), thus obtaining the IVs and the keys? Easy! The PRNG must be seeded with a random value using srand(). However, in this case srand is seeded using a value that is anything but random:

```
time_seed = time(0);
srand(time_seed);
```

Although it isn't random, it's still a bit of a hassle to determine. Recall that the program starts encrypting immediately when started and creates new files. Aside from actual contents, filesystems offer metadata regarding stored files containing:

- creation timestamp
- access timestamp
- last modification timestamp
- permissions
- ownership

The solution is to determine when the encryption started by sorting by the creation timestamp of the encrypted files. This allows us to estimate the value that the time(0) call could have returned: trying a few values around this timestamp is enough. But what exactly are we looking for and how do we know we've found it?

Since we can mimic the PRNG completely, we can obtain an indefinite amount of pairs of (key, IV) starting from a timestamp. Since the IV is stored in the files, we know exactly when we've reached the correct timestamp. Using this primitive of pair generation we then build a dictionary with the key being the IV and the value being the AES key. In the end, this dictionary allows us to obtain the AES key that was encrypted in the "key_sz bytes" part of the special header and decrypt all the files successfully. We packed this algorithm into a few scripts and let the people free themselves from the encryption cage.

It was not long before we encountered a person who had non-trivial problems with the decryption. Moreover, the files encrypted were dated "Aug 25" although the suspicion was that this "first" Linux ransomware appeared in late October-early November. We then asked the person to give us the ransom note, some encrypted files and, if possible, the encryption binary.

Before long, we realized we were actually dealing with a predecessor of Linux.Encoder.1

2 Linux.Encoder.0 intro

The Ransom note was similar to the previous one:

```
Your personal files are encrypted! Encryption was produced
using a unique public key RSA-2048 generated for this computer.
To decrypt files you need to obtain the private key.
The single copy of the private key, which will allow to decrypt
the files, located on a secret server on the Internet.  After
that, nobody and never will be able to restore files...
To obtain the private key for this computer, which will
automatically decrypt files, you need to pay 4.6 bitcoins
(~1000 USD).
Without key, you will never be able to get your original files
back.

---------------------------------------------
!!!!!!!!!!!!!!!!!!!!  PURSE FOR PAYMNET:
13JqaSsVg2tVBpbbvbwgF2jzdK5Sn2rQ68      !!!!!!!!!!!!!!!!!!!!
After you made payment, you should send email to
X.X.X.X@mailinator.com - which must contains you're BTC wallet.
After this, our system will automatically checks payment and
send to your email private key for decryption.
If you have any questions about payment, you can send also
to X.X.X.X@mailinator.com
```

The note is similar, aside from the mailinator address and the ransom amount:

```
To obtain the private key and php script for this computer,
which will automatically decrypt files, you need to pay
1 bitcoin(s) (~420 USD).
```

By checking the blockchain, we ascertained that 7 people had paid this fairly high amount around 25 August when the attack took place: `https://blockchain.info/address/13JqaSsVg2tVBpbbvbwgF2jzdK5Sn2rQ68` and they cashed out around 2015-11-10 18:47:58. Searching for 13JqaSsVg2tVBpbbvbwgF2jzdK5Sn2rQ68 on the Internet shows more sites have been infected.

However, having no sample meant that decryption (if it's not exactly the same as the recent sample) is virtually impossible. Or is it?

3 Linux.Encoder.0 forensic analysis

Provided with some encrypted files, I started to look into why the decryption is not working:

```
$ ls *.encrypted
dump.txt.encrypted
phpMyAdmin-3.2.4-english.tar.gz.encrypted
rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted
```

```
unixbench-5.1.2-1.el5.rf.i386.rpm.encrypted

$ hexdump -Cv dump.txt.encrypted  | head
00000000  00 01 00 00 8a 0b ab 39  36 4b 37 d6 19 b6 11 91  |.......96K7.....|
00000010  93 d6 3b 10 c8 6f dc b2  c2 4a 57 a3 65 e6 e4 12  |..;..o...JW.e...|
00000020  2a 0d 2b 1f 94 aa 61 68  41 62 26 ee 0c 2b 3f 84  |*.+...ahAb&..+?.|
00000030  8c f1 8d f7 4e 2f 04 0a  36 d7 22 17 60 d3 fd 1a  |....N/..6.".`...|
00000040  18 76 b6 da 6f 88 04 1f  3f 86 22 c5 3a 36 72 dd  |.v..o...?.".:6r.|
00000050  a2 1d 67 3a 0e 21 2e 8a  97 56 c5 32 41 a3 d9 21  |..g:.!...V.2A..!|
00000060  bd 8d e6 02 0a e8 e8 be  2f 1e 70 af 4d 72 1f b1  |......../.p.Mr..|
00000070  21 a1 18 8b f9 0a 0b b6  82 73 e5 55 9c 31 86 62  |!........s.U.1.b|
00000080  d0 ef 33 f4 93 f4 6a af  5e 01 bf c7 58 f1 57 79  |..3...j.^...X.Wy|
00000090  50 a5 13 cd cb d5 0b e5  a7 3c 3d 08 30 17 f7 96  |P........<=.0...|
```

The header looks slightly familiar but the first 4 bytes containing the permissions are gone. Nothing else stands out immediately and modifying the algorithm to account for the missing permissions reveals no timestamp. Does the file even have an IV? Does it use the current time? We have no idea without the sample so another approach should be used.

3.1 Idea: block or stream?

I first thought that it might be another encryption method, such as RC4. How could this be ascertained? Well, in Encoder.1 because AES is used the file sizes are of the form $4 + 4 + 256 + 16 + 16 \cdot k$ because AES is a 16-byte block cipher, so always an even size. By using a stream cipher, the encrypted content can have arbitrary lengths so the total file size can be odd. Let's check:

```
$ stat --printf "%s %n\n" *.encrypted
6274526 dump.txt.encrypted
1889865 phpMyAdmin-3.2.4-english.tar.gz.encrypted
16974 rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted
142659 unixbench-5.1.2-1.el5.rf.i386.rpm.encrypted
```

Bingo! We have two files that have an odd file size, so it could be a stream cipher. But which one? And what is the key?

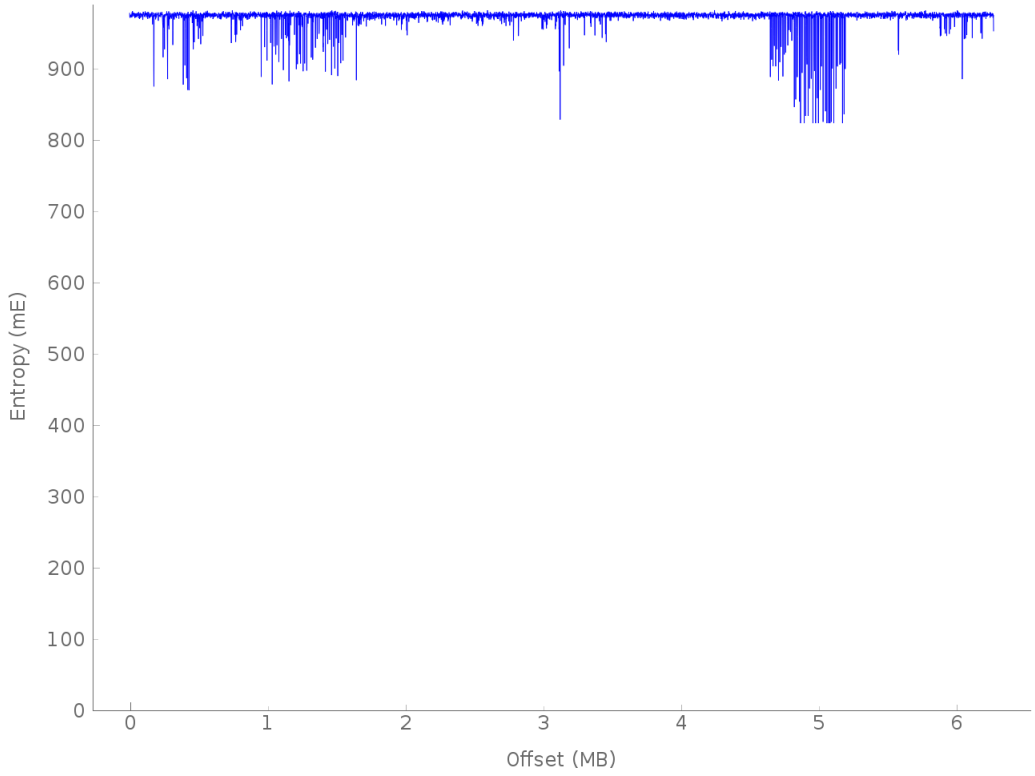
3.2 Idea: Known plaintext size

Since the rpm files are publicly available on the Internet, we can obtain before and after images of files. Downloading an rpm revealed the following:

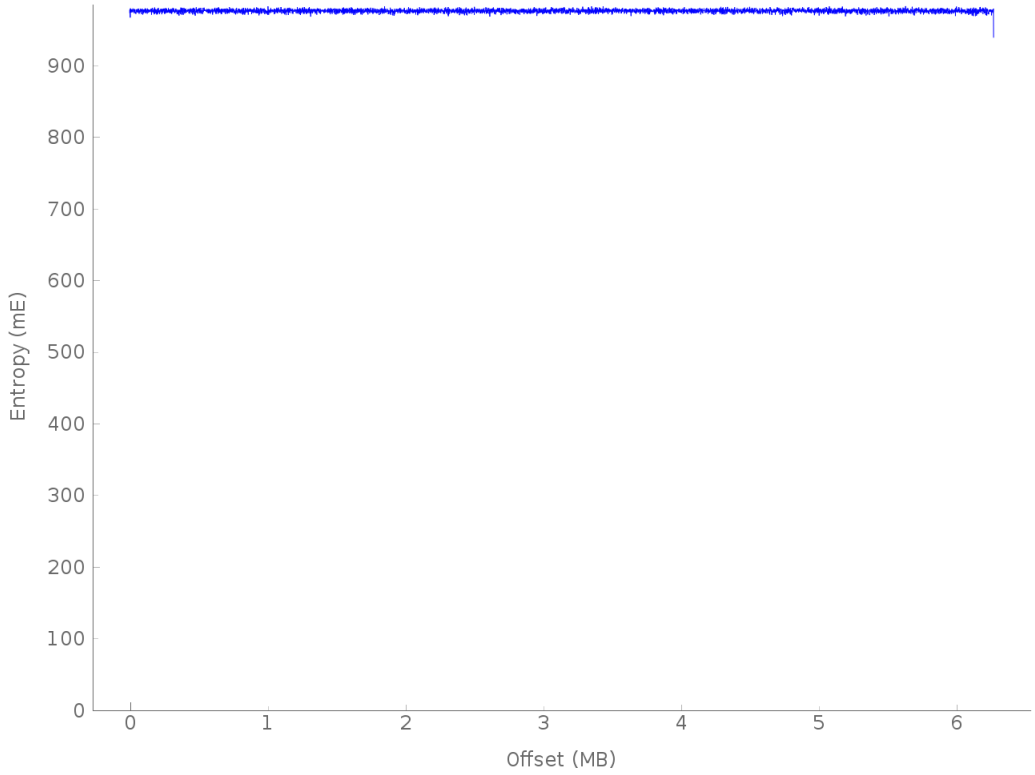
```
$ stat --printf "%s %n\n" rpmforge*
16698 rpmforge-release-0.3.6-1.el5.rf.i386.rpm
16974 rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted
$ echo $((16974 - 16698))
276
```

We already know the header is at least $4 + 256 = 260$ bytes but there are 16 bytes extra. Could it be an IV? Common stream ciphers (for an attacker of this capability) that might be used could be RC4, AES-CTR or something homebrew. But RC4 does not use both a key and an IV (or 16 bytes; we don't know yet if it's an IV).

In the best scenario possible, this could be a faulty xor encryption and although highly unlikely, it's worth trying an entropy analysis using binwalk.



The spikes are unusual for a supposedly secure encryption that also employs an IV or similar. For comparison, a file of the same size obtained from /dev/urandom looks like:



3.3 Idea: block segmentation

By using xortool (greetz to hellman from MSLC), I ran a quick analysis to see possible block lengths:

```
$ xortool rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted
The most probable key lengths:
   2:   10.8%
   4:   12.0%
   6:   9.3%
   8:   13.0%
  10:   8.2%
  12:   8.7%
  14:   7.0%
  16:   12.5%
  32:   10.6%
  64:   8.0%
```

Not a very promising result but it seems that for 4, 8 and 16, there is something. On a hunch I tried to see if there are repeated blocks of the biggest length: 16

```
$ xxd -p -c 16 rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted | sort
 ↪  | uniq -c | sort -n | tail
      1 fefe1ea7565294ec985770397dead1d0
      1 ff6f5bf5b16c439a6a56fe6d2d622337
      2 a2c4dab80fdb2161e026b6067d4ca90e
      2 dec0ef5efdfb8f8b3b11315b58c05ad8
      3 77c2be2309d72974f6db38be6ff1c8cb
```

```
      3 db78c7d02f94a3443348b1ab06e6f02f
      3 f21c3d4c9f140983b6045104809d633c
      4 63db8466b2f75c5c8814c3289e11141d
      4 8e9577882422f9c1ff2717162fba9cfd
      5 df427bc59f5f6dab4767fe400d244a4f
```

If the file was encrypted using AES-CBC like the Nov sample, the probability of these blocks repeating would have been ZERO. But this might still be AES-ECB. How can we tell? Split the blocks at length 8 and apply the same procedure:

```
$ xxd -p -c 8 rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted | sort
 ↪  | uniq -c | sort -n | tail
      4 3b11315b58c05ad8
      4 63db8466b2f75c5c
      4 8814c3289e11141d
      4 8e9577882422f9c1
      4 b6045104809d633c
      4 db78c7d02f94a344
      4 f6db38be6ff1c8cb
      4 ff2717162fba9cfd
      5 4767fe400d244a4f
      5 df427bc59f5f6dab
```

Now here's something more interesting:

```
$ xxd -p -c 16 rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted | sort
 ↪  | uniq | grep f6db38be6ff1c8cb
02c2be2309d72974f6db38be6ff1c8cb
77c2be2309d72974f6db38be6ff1c8cb
```

Two blocks that differ by only one byte. Good news: this is also not AES-ECB, it actually looks like a plain XOR.

3.4 Idea: known plaintext content

Remembering that we have the full contents of the file both before and after encryption, we then try xoring the two:

```
$ python xor.py rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted rpmforge-release-0
00000000  9f 14 09 83 b6 04 51 04  80 9d 63 3c 77 c2 be 23  |......Q...c<w..#|
00000010  09 d7 29 74 f6 db 38 be  6f f1 c8 cb db 78 c7 d0  |..)t..8.o....x..|
00000020  2f 94 a3 44 33 48 b1 ab  06 e6 f0 2f df 42 7b c5  |/..D3H...../.B{.|
00000030  9f 5f 6d ab 47 67 fe 40  0d 24 4a 4f 8e 95 77 88  |._m.Gg.@.$JO..w.|
00000040  24 22 f9 c1 ff 27 17 16  2f ba 9c fd de c0 ef 5e  |$"...'../......^|
00000050  fd fb 8f 8b 3b 11 31 5b  58 c0 5a d8 63 db 84 66  |....;.1[X.Z.c..f|
00000060  b2 f7 5c 5c 88 14 c3 28  9e 11 14 1d a2 c4 da b8  |..\\...(........|
00000070  0f db 21 61 e0 26 b6 06  7d 4c a9 0e f2 1c 3d 4c  |..!a.&..}L....=L|
00000080  9f 14 09 83 b6 04 51 04  80 9d 63 3c 77 c2 be 23  |......Q...c<w..#|
```

```
00000090  09 d7 29 74 f6 db 38 be  6f f1 c8 cb db 78 c7 d0  |..)t..8.o....x..|
000000a0  2f 94 a3 44 33 48 b1 ab  06 e6 f0 2f df 42 7b c5  |/..D3H...../.B{.|
000000b0  9f 5f 6d ab 47 67 fe 40  0d 24 4a 4f 8e 95 77 88  |._m.Gg.@.$JO..w.|
```

```
$ python xor.py rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted
↪  rpmforge-release-0.3.6-1.el5.rf.i386.rpm | xxd -c 128 -p | sort |
↪  uniq -c
      1 9f140983b6045104809d633c77c2be2309d72974\
        f6db38be6ff1c8cbdb78c7d02f94a3443348b1ab\
        06e6f02fdf427bc59f5f6dab4767fe400d24
    130 9f140983b6045104809d633c77c2be2309d72974\
        f6db38be6ff1c8cbdb78c7d02f94a3443348b1ab\
        06e6f02fdf427bc59f5f6dab4767fe400d244a4f\
        8e9577882422f9c1ff2717162fba9cfddec0ef5e\
        fdfb8f8b3b11315b58c05ad863db8466b2f75c5c\
        8814c3289e11141da2c4dab80fdb2161e026b606\
        7d4ca90ef21c3d4c
```

The key repeats itself from 0x80 and the last part is truncated as the file size is not a multiple of 128.

Now we know the encryption context is the following:

- one block of 16 bytes in the file (IV?)
- one block of ?? (maximum 0x100) bytes encrypted with RSA in the file (key?)
- these two somehow combine and form a stream of 128 bytes that is used as a repeated XOR mask.

3.5 Idea: LCG output

At this point I was pretty much stuck, baffled by the sheer number of possibilities of encryption. However, taking a step back provided some headway. Assuming the ransomware authors were lazy and didn't change so much between these two versions, the extra 16 bytes in the file might come from rand() seeded with time(0), maybe even the 128 keystream too.

It's worth noting that there are multiple implementations of srand()/rand() and, by trying a few of them, I came upon the correct one: it was from glibc (which was different from the Nov sample). I first generated 512 values of rand() using srand(file_timestamp) and assumed that the IV was generated the same: rand() % 256:

```
$ foresee glibc  -c 512  -s 1440465460 rand
1184999731 515046404 743410982 755965648 1550080474 263744381 1731232729
......
$ foresee glibc  -c 512  -s 1440465460 rand > values

$ python dump.py values | xxd -p -c 100000
330426d0da7dd94bcf11525b54ee9aecb9de2b0afa3254079214c1eb19f4734df8991dd216f61de60
86f415c5ddc4816ba7320b4a674bc38887d23a27196ef692f0c3b4503582b0bc86d672549af3c0323\
```

```
5cb8c9d1740159f124fb63baeacce9f7082efa605a0528c76c4e101b8a133ee6cb07b73f0811312c0\
c94e6f760cfee68fee8c958edf11f593f2f74c942b3b00eba674dc3787eef8512d67c73a56adba352\
a4fb3f961a98d5490c9f8cbf4f9a7ab6e73d2f662cb4780230eba89ac74bec6b472b0161c3d7abcf7\
6378fc5d1097bb846aa1e725e97758e821d28496814b5af3fb611028dbcd203f361c8c46a447cb0ee\
9b224d3297dbb4b404fe1d18b3cc5869dd5af7992cfa8c8dc350f707cda7f568ca429a611e4e16224\
c333afffff9269dded6076195a03a71d539e2420461a88105c22717a71879cbdbad7bdba6926975686\
0d70e01017fe64b52284fb3c0d0b992f7d13a004b05ebf871b79f0411047c71db7a72ecea59283c81\
67f04238a9d5307b1f30c6151cbe86c44d8ad541f7471d61c9fa4c131288948a78e6b312bbe39dcb2\
453d0310257054fe1da81d9219f4aeb9986feac0f93267879e99b25cd28f0e17cc1227f2827bf09f2\
30d313c01dff59a4fe05a4812c2cfb05b810d2d101b44dd2d6bcfafe6bf4f09cc8045ce603b68af1b\
c2f72d84c6dedf47eb0c5806503534bb04e3a1c332aa8fb3f05d0a

### extract the IV
$ xxd -s 0x104 -l 16 -p -c 16 rpmforge-release-0.3.6-1.el5.rf.i386.rpm.encrypted
a7f568ca429a611e4e16224c333affff
```

Notice the IV appears in the raw hex dump and it's also 16-byte block aligned. We do this with all the files, neatly splitting the output:

```
$ python test.py  | xxd -p -c 16
330426d0da7dd94bcf11525b54ee9aec
b9de2b0afa3254079214c1eb19f4734d
f8991dd216f61de6086f415c5ddc4816  # IV from phpMyAdmin-3.2.4-english.tar.gz.encrypted
ba7320b4a674bc38887d23a27196ef69
2f0c3b4503582b0bc86d672549af3c03
235cb8c9d1740159f124fb63baeacce9
f7082efa605a0528c76c4e101b8a133e
e6cb07b73f0811312c0c94e6f760cfee
68fee8c958edf11f593f2f74c942b3b0
0eba674dc3787eef8512d67c73a56adb
a352a4fb3f961a98d5490c9f8cbf4f9a
7ab6e73d2f662cb4780230eba89ac74b
ec6b472b0161c3d7abcf76378fc5d109
7bb846aa1e725e97758e821d28496814
b5af3fb611028dbcd203f361c8c46a44  # IV from unixbench-5.1.2-1.el5.rf.i386.rpm.encrypte
7cb0ee9b224d3297dbb4b404fe1d18b3
cc5869dd5af7992cfa8c8dc350f707cd
a7f568ca429a611e4e16224c333affff  # IV from rpmforge-release-0.3.6-1.el5.rf.i386.rpm.e
9269dded6076195a03a71d539e242046
1a88105c22717a71879cbdbad7bdba69
269756860d70e01017fe64b52284fb3c
0d0b992f7d13a004b05ebf871b79f041
1047c71db7a72ecea59283c8167f0423
8a9d5307b1f30c6151cbe86c44d8ad54
1f7471d61c9fa4c131288948a78e6b31
2bbe39dcb2453d0310257054fe1da81d
9219f4aeb9986feac0f93267879e99b2
5cd28f0e17cc1227f2827bf09f230d31
3c01dff59a4fe05a4812c2cfb05b810d
```

```
2d101b44dd2d6bcfafe6bf4f09cc8045  # IV from dump.txt.encrypted
ce603b68af1bc2f72d84c6dedf47eb0c
5806503534bb04e3a1c332aa8fb3f05d
```

3.6 Idea: decrypt to encrypt

A few patterns emerge: it seems that, for each file to be encrypted, there are 32 calls to rand() (to generate a key?) and then 16 more to generate the IV. What encryption algorithm uses a 32-element key and a 16-byte additional value and produces 128 bytes of pseudo-random keystream? Maybe AES-256 with a 32-byte key and 16-byte IV that encrypts something fixed? Judging from the Nov sample it could use AES-CBC.

To validate this wild assumption we would need to know when we are correct. Since the encryption could be of something static then the decryption from two different keystreams (from two different files) should decode to something static!

I first tried to generate the 32-byte key like the IV: rand()%256 but had no luck. Then I remembered that in the Nov sample the AES key is calculated differently:

```
[...]
//AES key generation
char alphabet[] = "abcdefghijklmnopqrstuvwxyz\
        ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,.-#'?!";
[...]
sz = 16;
while ( i < sz )
        key[i++] = alphabet[rand() % 69];
[...]
```

Applying this on the 32 rand() outputs provided the final solution:

```
$ tail test.py
iv_bytes = ""
for i in iv:
        iv_bytes += chr( i % 256)

c = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,.-#'?!"
key_bytes = ""
for i in key:
        key_bytes += c[ i % len(c) ]
cipher = AES.new(key_bytes[0:16], AES.MODE_CBC, iv_bytes)
print cipher.decrypt(mask).encode('hex')
$ python test.py
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Jackpot! So it actually uses AES-128 and drops bytes 16-32 from the 48 calls to rand() for one file. In short, the algorithm is the following:

```
srand(time(0));
for each file:
   generate 16-byte IV and 32-byte Key,
   encrypt 8 blocks of 00 using AES-128 in CBC mode with this IV and key
   apply this xor mask to the input file
```

4 Conclusion

- Never leave crypto to amateurs. You either know what you're doing or you fail tragically. This is one of the few cases where bad crypto is actually good.
- Decryption can be done even without the exact infection sample. Lots of leaps of faith and educated guesses are required, however.
- The ransomware era has just begun. Expect plenty of cases like this (hopefully with the same gaping security flaws).
- But most importantly, patch your systems regularly to avoid becoming a victim.