Hardening File Storage Crypto Systems By Utilizing Real Random Numbers.

Glenn Larsson, Sweden
[*Ichinin@SUESPAMMERS.org*]

Real random numbers are normally unwanted in Crypto systems when we are storing data for later retrieval. Exceptions when real random numbers are necessary are key generation and key exchange/negotiation.

I present a way to utilize real random numbers in cryptography to make attacks upon file storage systems harder: The idea is to use a random salt to harden security, the same way as one would go about to store passwords on a Unix system.

INTRODUCTION:

Most file encryptors use a single, static key (K) to encrypt multiple files. Making a single compromised key a total simultaneous compromise of ALL files.

The problem.

If one keystream is recovered from a known plaintext attack (i.e. known header) usually the whole system fails because every file is encrypted using the same K. In the model presented in this paper, no other files can be recovered using one recovered keystream.

This system is only valid if the master key (K) can be protected (which also is a problem for all file system protected; at runtime, all encryption systems are vulnerable to key theft and other attacks.)

With this in mind, we move on.

TERMINOLOGY:

Hash() = A One way hash algorithm (i.e. MD5)

RN = a random number, (My example app uses size 2^16)

PP = Passphrase

K = Key

K' = Key derived from Hashing (K, RN and PP)

ENCRYPTION:

1) A real random number (RN) is generated, it is used to generate a random encryption key.

K' = Hash(RN & K & PP)

2) We now have K' (Subgroup of K), we use that to encrypt the message.

3) When that is done we simply concatenate:

Hash(NOT(RN) & K & PP)

to the ciphertext.

[Since finding out K is equivalent to breaking the hash algorithm, we do not worry (much) about that.]

DECRYPTION:

1) The program go to the end of the ciphertext and extract the hash.

2) We provide K and PP, the program starts guessing the right RN by comparing Every Hash(RN & Key & PP) with the stored hash. Since RN is small we can do this in a reasonable amount of time.

3) With the correct RN in hand, we invert (NOT) in the 2^16 sized "keyspace"; we can now reconstruct K'.

NOTE:

RN need to be LARGE! 2^16 are fast, but insecure: it will most likely produce two similar keystreams once in a while, so if enough keystreams are recovered, the system could be compromised anyway(!), so a RN sized 2^24 to 2^32 (or more) would be recommendable.

However, it should be fast enough to find if (or rather "when") you decide to change the Encryption key or Passphrase.

AN OBSERVATION:

When generating K', the filename could also be added to produce an even more unique keystream, this does not increase security by one bit, but it do increase the number of keystreams in the system. This would produce a subset of keys for one specific file, and the RN would guarantee that no K' would be the same for multiple copies of the same file, and no files of the same type (i.e. executable) would ever produce the same ciphertext.

AN EXAMPLE:

K = "12345"
PP = "Open sesame"
RN = 16705 (asc: "AA")

K' = Hash (K & PP & RN) = Hash ("12345Open sesameAA")

K' becomes 0F1B10E79BFFE91E38CE4685DBC0CF846B90FC3F,

We encrypt data with K' and append:

Hash(K & RN) = Hash ("12345AA")
(9E7F98A4E5165F6CC441015BAC5F0F664047DE2F)

To decrypt, we extract the appended hash from the ciphertext, we denote this "HTest".

We now guess RN by bruteforcing the small space of RN:

```
For (RN = 0; RN < 65536; RN++)
{
  if (HASH(RN & K) == HTest) {
        // we guessed correct RN, we can now decrypt(!)
          Decrypt (K & PP & RN);
        }
}
```

REFERENCES:

None

DOCUMENT HISTORY

```
- Fixed flaw in protocol; to reconstruct K’ from
appended hash, one have to provide K & PP, then the
program searches for RN, when found it performs NOT(RN)
to find out the correct RN.
```