TKS1 - An anti-forensic, two level, and iterated key setup scheme

Clemens Fruhwirth <clemens@endorphin.org>

January 13, 2009

Abstract

This paper sketches the problems connected with usual hard disk encryption setups. It introduces the reader to PBKDF2, a password based key derive function, which provides better resistance against brute force attacks based on entropy weak user passwords. It proposes to use a two level hierarchy of cryptographic keys to provide the ability to change passwords and drafts solutions to the key storage problem arising when using two levels of cryptography due to the fact, that given the abilities of recent forensic data recovery methods, data can't be destroyed on magnetic storage media reliably.

The frowardness of many modern operating system to store data, like swap or temporary files, on disk at will raises the question, if traces of sensitive data ends up on solid storage media) As Peter Gutmann pointed out in his seminal paper[2], several difficulties arise when sensitive data, written to temporary storage, is expurgated. Instead of repressing the stubbornness of OS the approach of encrypting the whole hard disk seems to be tempting. This paper will focus on the key management for the realtime disk cipher used for encrypting the block device.

1 The dictionary attack

User suplied passwords have usually two unwanted properties. They are too short and sometimes even based on dictionary words. Both emerge due to the user's preference for easy rememberable short passphrases. From a cryptographers point of view, the problem with short strings as well as English words is, that they tremendously lack entropy. A password generated by the standard Unix command `mkpasswd` will give you a 13 character random string where the term "character" refers to a set of 64 symbols[1]. This yields 78 bits[2]of entropy. Given this symbol set, the user would have to remember 42 random characters to provide enough entropy to form a 256 bit key.

The usual passphrase length is nowhere around that, neither is it random generated. Choosing a regular English word with length 10 will yield 12 bits entropy[3]. The entropy gap to a 128 bit key or worse, a 256 bit key, is highly

[1] A-Z, a-z, 0-9, '/', '.'

[2] $\prod_1^{13} 2^6$

[3] 1.2 bits per character

visible. A potential attacker could easily traverse a dictionary instead of the whole password domain. Even if these 10 characters are random generated, the entropy would be 50 bits, when using 64 symbols/character. In 1999 distributed.net compromised a 56 bit DES key within 22 hours. So 10 characters are not secure either.

As technology advances, larger key sizes become attackable by brute force in less time. One might think one just has to raise the key size, and therefor raise the length of the passphrase to be safe. Since technology advances much faster than humans capability to remember arbitrary passwords do, the gap to the feasibility of this attack becomes smaller everyday.

The following scheme will not improve this insufficient entropy and therefor an insufficient attackable complexity, but it artificially hampers the computation of a single brute force operation. A single brute force attack consists of setting up the cipher with a passphrase to be attacked, computing the encryption of a cipher block likely to contain known information and checking the result against plausible outcomes. Since making the regular decryption slower is not desirable, because it would also hamper the legitimate usage of the system, the item of interest is the key setup phase.

By inserting a CPU intensive transformation into the key setup path the single brute force operation can be made much more expensive to carry out without imparing the regular use. Mathematically it's a map of the given password to the key domain, which doesn't need to be invertible. Hash functions qualify best for this purpose, as they are well defined for variable sized input. To make them CPU intensive, one can just iterate them for a number of times. PBKDF2, standardized in RFC 2898 [1], is based on this idea.

A potential attacker would have to have access to a precomputed dictionary, storing $f(password)$ instead of $password$, to be unchallenged by this measure, where f is the CPU intensive function. A complete enumeration of the password domain generated from by `mkpasswd` (of the size 2^{78}), would require 8 yottabytes[4] storage, which might be achievable one day[5]. Therefor, instead of utilizing a generic function, $f(password)$, it's advisable to make them depend not only on the password, but also on a constant random value s, commonly referred to as salt. By choosing s at the time of the initial creation of a cryptography volume, an attacker would have to have a dictionary available, which contains $f(password, s)$ for all possible $password$ and s, to be unchallenged. By making the domain of s larger, the storage requirements of a precomputed dictionary becomes larger as well. Since s doesn't have to be secret, it can be stored on disk and therefor it is relieved of any size constraints imposed by the limited human's ability to remember random content. The solution is quite simple, make s that large, that storaging a dictionary is unlike to become feasible for the timespawn one would like to keep the data secret.

Further one has to make sure that by repeatedly applying f the codomain of f does not degenerated into a smaller set of values. To prevent such drifts, the function f should not simply iterate a hash function, but also compute a result based on it's intermediate values. XOR-ing the intermediate values together should provide enough safety against this possible degeneration.[6]

[4] 2^{30} Petabytes

[5] Cryptographers tend to be satisfied only when the storage complexity exceeds 10^{78}, the approximate numbers of atoms in the universe.

[6] For a discussion of this subject see `http://www.google.com/groups?threadm=3BC854A7.`

PBKDF2 implements these three key concepts: iterations, salt and degeneration countermeasures. Further PBKDF2 can generate an output of arbitrary length by concatenating an "initial vector" to the salt, therefor altering all successive computations. The following function F yields the i-th block derived from P, the password, S, the salt, and c, the iteration depth. PRF is a pseudorandom function, usually a hash function in a HMAC setup [3].

$$\begin{aligned} F(P,S,c,i) &= U_1 \oplus U_2 \oplus \ldots \oplus U_c \\ U_1 &= PRF(P, S \ || \ INT(i)), \\ U_2 &= PRF(P, U_1), \\ &\ldots \\ U_c &= PRF(P, U_{c-1}) \end{aligned}$$

The feasibility of attacking this scheme depends on the ratio of the user's CPU power to the processing power available to an attacker. The user can always adjust the CPU intensity of the hash process to approximately take 5 seconds. That amount of time seems reasonable to wait for a password confirmation. If an attacker has the same amount of processing power at his disposal as the user, he would require about 10^{16} years to enumerate the passphrase domain generated by `mkpasswd`, assuming a single operation only consists of 5 seconds key setup time and the decryption time is marginal[7]. Even for commercial entities information security requirement should not exceed a timespan of 1000 years. Therefor an attacker must be 10^{13} times ahead of the user in terms of processing power. The gap between ordinary workstations and supercomputer is far from being in magnitudes like 1:10^{13}. As a little illustration, the Earth Simulator Computer[8] deployed in Japan can carry out 35 TFLOPS in contrast to Pentium 4 which can compute about 0.5 GFLOPS. FLOPS are not ideal for benchmarking actually processing power, however but it demonstrates there is a gap of the magnitude of 1:70000 between regular user hardware and supercomputers. So, the gap of 1:10^{13} is not likely to be filled, not even with special manufactured hardware.

2 Two level encryption

If the password is compromised, for example by someone standing behind the user while typing, it becomes evident, that it's very unhandy to re-encrypt the whole hard disk with a new passphrase[9]. The solution to this is to build a cryptographic hierarchy. The hard disk will be encrypted with a generated master key, which is in turn encrypted with a user supplied passphrase and stored on disk. When the user wants to open his cryptographic storage, the passphrase is entered and the master key is thus recovered and can be used to decrypt the whole hard disk. In case of compromise, the master key can be

`56005C4%40no.spam.please`

[7] $2^{78} keys * 5 seconds/key * \frac{3110400 seconds}{1 year}$

[8] `http://en.wikipedia.org/wiki/Earth_Simulator`

[9] This can only be done when the master key itself has not been compromised.

encrypted with a new passphrase and the instance of the master key encrypted by the compromised key can be destroyed. The master key should never be written to any storage media, it should only be kept in memory, or should even be destroyed after the key setup of the underlaying cipher is completed.

3 Anti-forensic data storage

Hard disk have a long long memory. Even if you think data is gone, even if you have overwritten the whole disk with zeros, even if you invoked the security-erase ATA command of your IDE hard disk, data can be easily recovered, if not special care is taken to destroy it properly. The two level encryption scheme, which stores the encrypted master key on disk, must take extra percaution, since data is not guaranteed to be ever erased. Bad block remapping of modern firmwares supports data safety but weakens the opposite, data destruction.

If the probability p to destroy a certain block of data is $0 < p < 1$, then the probability that the block survives is of course $1 - p$. Given a set of data of the size l, the probability to destroy the whole block becomes worse since p^l becomes smaller as l becomes larger. But the probability that the whole block survives, $(1 - p)^l$, becomes smaller as well with an increasing l. If $(1 - p)^l$ is becoming smaller, than $1 - (1 - p)^l$ must become larger, which is exactly the chance, that the whole block does not survive. The reader should notice the subtile difference between "whole block is destroyed" and "whole block does not survive". The former means that all items are destroyed. The later means that one data item or more is destroyed.

Usually one is not able to control p, but the size of the data set l is arbitrary. By making l larger, one can make the chance of destroying at least one data block arbitrary large. For instance, if the odds are against us and the hard disk exhibits $p = 0.01$, we can almost reliable destroy a single data block in a 1000 item data set with a probability of 0.99995.

The remaining task is to make the destruction of a single data block crucial for the usability of the whole data set. A key is usually just a single data item with the length of 16 or 32 byte (128 bits or 256 bits). We will just treat the master key, which should be made reliably purgeable, as data set from now on, since this technique can be used for arbitrary data. The point is to distribute the information of a single data item D uniformly to all data items of another data set S, where "uniformly" refers to the property that any data item is equally important to extract the information.

An easy approach is to create the interdependency for a data set S, $S = s_1, s_2, \ldots s_n$, is by generating $s_1 \ldots s_{n-1}$ random data items and computing s_n so that $s_1 \oplus s_2 \oplus s_3 \ldots \oplus s_n = D$ ($\oplus$ denotes the XOR operation). The reconstruction is done by carring out the left-side of the equation, XOR-ing all data items together. If one item s_i is missing, D can't be reconstructed, since an arbitrary s_i effects the entire D.

This scheme can be enhanced to include diffusion of information, so that the k-th bit of an arbitrary s_i does not only affect the k-th bit of D but the entire D. To achieve this diffusion, we insert a diffusion element in the chain of XORs. A crytographic hash function can be used as such element, but since it might not output sufficiently large data, it will be processed a few times with an increasing number, similar to an initial vector, prepended to the complete

dataset to produce enough data. As a hash function is usually required to be non-invertible, we can not choose it's output. Therefor, the last diffusion will be omitted. This will degrade security slightly, so when computing destruction probabilities the last element shall never be taken into account.

As we can destroy a single undeterminated data item quite easily as shown in the previous paragraph, and as a single missing data item makes the base information unrecoverable, data items can be made reliable erasable. A sample implementation of this scheme is the AFsplitter, short for anti-forensic splitter, which can be found under `http://clemens.endorphin.org/AFsplitter`. As illustration, you can find the overall composition in Figure 1. H denotes the diffusion element, which is likely to be a hash[10] and Z denotes the zero vector. In the splitting phase, s_1 to s_{n-1} are random generated and the intermediate result I is computed. Then s_n computed as $s_n = D \oplus I$. When recovering the base information, the whole chain is computed as shown resulting in D, the original data item.

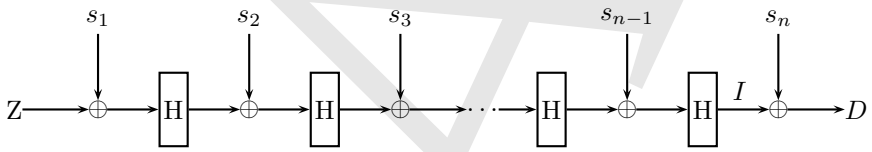

Figure 1: AFsplitter

3.1 Choosing the size of the inflated information

As the diffusion element in our design will cause a bit error in the data storage to have devastating effect on the output, we will focus on the probability of destroying bits, or in other words producing bit errors. Given the probability p that a bit is destroyed, $1 - p$ denotes the probability that a bit survives. The probability, that a data set with l items, each item k bits long, survives, is $(1 - p)^{lk}$. Our aim is to make this probability equal to the probability of guessing the master key of our datastore, which is $\frac{1}{2^k}$, where k denotes the key size (in bits).

$$(1 - p)^{lk} = (\frac{1}{2})^k$$

By applying ln to both sides and removing k from the equation, the following form is derived.

$$l = \frac{-\ln 2}{\ln(1 - p)} \tag{1}$$

Note that l does not depend on the length of the key k. To give the reader a sense for the magnitudes, selected values of p with the corresponding multiplication factor l are presented in the following table,

[10] To construct enough data from a hash with fixed output size, hash $h(s_i||0), h(s_i||1), h(s_i||2) \ldots$ until enough data is obtained

p	l
0.05	13.5134
0.01	68.9676
0.001	692.801
0.0001	6931.13

4 Assembling TKS1

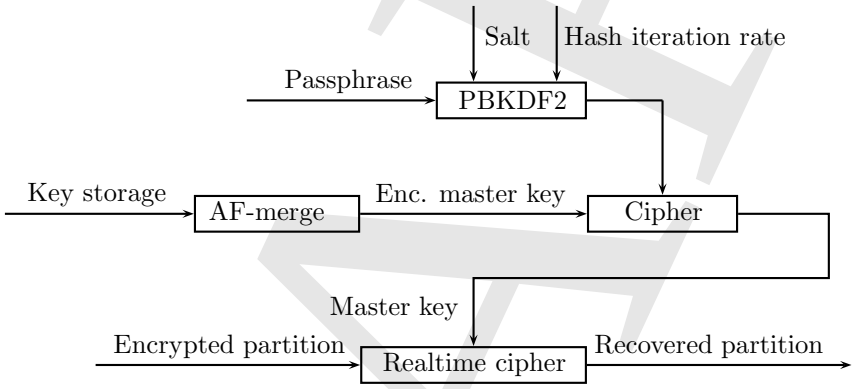

Figure 2: TKS1 scheme

Figure 2 depicts the overall structure of the advised solutions from previous sections. The salt as well, as the AF-splitted master key, is coming from a key storage. The salt can be saved without splitting, since it's not sensitive to the security of the system. The passphrase comes from an entropy-weak source like the user's keyboard input.

The initialization of the system is straight forward:

1. generate a master key
2. generate a salt to use for PBKDF2
3. choose an appropriate hash iteration rate by benchmarking the system
4. let the user enter the passphrase
5. process the passphrase with PBKDF2 and thus obtaining the password derived key
6. setup the master key cipher with the password derived key
7. encrypt the master key with the master key cipher
8. AF-split the encrypted master key
9. save the AF-splitted encrypted master key, the iteration rate and the password salt to storage
10. (setup the realtime cipher with the master key)

11. (destroy master key copy in memory)

The recovery of an encrypted volume happens as follows, where step 3-5 are identical to generation-mode steps 4-6:

1. read salt and iteration rate from key storage
2. read the AF-splitted encrypted master key from storage and AF-merge in memory and thus obtaining the encrypted master key.
3. let the user enter the passphrase
4. process the passphrase with PBKDF2 (salt and iteration as parameter) and thus obtaining the password derived key
5. setup the master key cipher with the password derived key
6. decrypt the encrypted master key with the master key cipher
7. setup the realtime cipher with the master key
8. (destroy master key copy in memory)

When a password has been compromised, the master key can be recovered as shown above, but instead of using it for the realtime cipher, it can be re-encrypted using a new password derived with PBKDF2. The master key encrypted with the old, compromised password can be easily destroyed as proposed in Gutmann's paper[2].

5 Reference implementation

LUKS, Linux Unified Key Setup, is a sample implementation of TKS1. More information about this project can be found under
`http://clemens.endorphin.org/LUKS`

References

[1] B. Kaliski, RSA Laboratories. RFC 2898; PKCS #5: Password-Based Cryptography Specification Version 2.0. `http://www.faqs.org/rfcs/rfc2898.html`, 1996-1997.

[2] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. `http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`, 1996.

[3] M. Bellare, R. Canetti, and H. Krawczyk. The HMAC papers. `http://www.cs.ucsd.edu/users/mihir/papers/hmac.html`, 1996-1997.