What users should know about Full Disk Encryption based on LUKS [⋆]

Simone Bossi [⋆⋆] and Andrea Visconti [⋆⋆⋆]

Cryptography and Coding Laboratory (CLUB),
Department of Computer Science,
Università degli Studi di Milano
`http://www.club.di.unimi.it/`

`simone.bossi2@studenti.unimi.it,andrea.visconti@unimi.it`

Abstract. Mobile devices, laptops, and USB memory usually store large amounts of sensitive information frequently unprotected. Unauthorized access to or release of such information could reveal business secrets, users habits, non-public data or anything else. Full Disk Encryption (FDE) solutions might help users to protect sensitive data in the event that devices are lost or stolen. In this paper we focus on the security of Linux Unified Key Setup (LUKS) specifications, the most common FDE solution implemented in Linux based operating systems. In particular, we analyze the key management process used to compute and store the encryption key, and the solution adopted to mitigate the problem of brute force attacks based on weak user passwords. Our testing activities show that unwitting users can significantly reduce the security of a LUKS implementation by setting specific hash functions and aggressive power management options.

Keywords: Linux Unified Key Setup (LUKS), Password-Based Key Derivation Function 2 (PBKDF2), Full Disk Encryption (FDE), hash functions, HMAC.

1 Introduction

Nowadays, mobile devices, laptops, USB memory are convenient and easy to use. They are fast becoming the preferred choice of companies, customers and employees, especially by those who are on the

[⋆] A slightly different version of this paper appeared in the Proceedings of the 14th International Conference on Cryptology and Network Security (CANS 2015), Springer International Publishing, LNCS 9476.

[⋆⋆] Part of this work was performed as part of the author's B.Sc. thesis, under the supervision of Dr. Andrea Visconti

[⋆⋆⋆] Corresponding author: andrea.visconti@unimi.it move. These devices usually store large amounts of sensitive information frequently unprotected. If such devices are lost or stolen, the risk of unauthorized disclosure of confidential, sensitive, or classified information is very high and the impact to the affected companies is potentially billions of dollars [15]. However computer users are not the only ones who do not pay attention to security when it comes to protecting sensitive data. Many operating systems store temporary files/swap partitions on hard drive and a number of problems arises when these files contain sensitive data [10].

A possible solution is to encrypt the whole hard disk. Full Disk Encryption (FDE) solutions, also known as "On-Disk Encryption" or "Whole Disk Encryption", work by encrypting every single bit of data that resides on a storage device — i.e., operating systems, applications, swap partitions, user's files, and so on. FDE solutions aim to provide data security, even in the event that an encrypted device is lost or stolen. All information is encrypted/decrypted on the fly, automatically and transparently. Without the encryption key, the data stored on the disk remains inaccessible to any users (regular or malicious).

One of the main issues facing Full Disk Encryption solutions is the password management. Indeed, the master key used to encrypt the whole disk is stored on it. A well-known solution to this problem, is to adopt a two level key hierarchy [16] but sometimes it is not enough (e.g. two level key hierarchy adopted by Android 3-4.3 [6]), and a number of questions arise. Could the choice of specific cryptographic parameters significantly reduce the security of a FDE solution? How should users choose cryptographic parameters that best meet security requirements? Could external factors (i.e. power management options) affect the security of a FDE solution?

In this paper we try to find answers to these questions, evaluating the level of security provided by Linux Unified Key Setup, the most common Full Disk Encryption specification implemented in Linux based operating systems. In particular, we analyze the key management process used to derive the encryption key, and how the choice of specific hash functions and aggressive power management options may affect the security of a FDE solution.

The remainder of the paper is organized as follows. In Section 2, we introduce the problems of managing passwords and the solution adopted. In Section 3 we describe the LUKS design. In Section 4 we analyze the key management process used by LUKS implementations, explaining the possible weaknesses found. Finally, discussion and conclusions are drawn in Section 5.

2 Password management

An important problem to solve in FDE solutions is the password management. Users know they need to generate a strong password and change it frequently. But the process of changing encryption password brings with it a series of problems, indeed, if a FDE solution has been implemented using a master key which encrypts/decrypts the whole hard disk — i.e., single key schema — changing the master key means re-encrypt all the data with the new key. This process can be very time consuming and cause unacceptable unavailability of data.

A well-known solution to this problem, is to adopt a two level key hierarchy. A strong master key generated by the system is used to encrypt/decrypt whole hard disk. Such key have to be split, encrypted with a secret user key — each user has their own secret key — and stored on the device itself. The master key is unique but a number of encrypted master key are stored on disk, one for each user. This approach has a main advantage. If we set a new secret user key, the encrypted master key stored on disk changes but the master key does not. Hence, users can change password frequently without re-encrypting all the data.

But, what happens when a device is lost or stolen? Is the two level key hierarchy method strong enough to protect our sensitive data? When devices are lost or stolen, it is desirable that the master key cannot be decrypted by anyone. Unfortunately, master keys are protected with user keys which are usually short and lack entropy. Hence, an attacker would try to guess them constructing a list of possible passwords. A solution to this problem is described in [12]. Morris and Thompson suggest to combine a user password with a salt to generate a key. This approach allows to compute several possible keys for each user password. The effect is to discourage an attacker from precomputing a list of possible keys. Another solution described in literature [16] is to derive the key using a Key Derivation

Function (KDF). This approach tries to slow down the computation of malicious users to mitigate the problem of brute force attacks. In particular, the KDF allows legitimate users to spend a moderate amount of time on key derivation, while inserts CPU-intensive operations on the attacker side.

To face the problems of password management described in this section, it is possible to adopt a solution based on a two level key hierarchy and protect the master key using both salt and key derivation function.

2.1 PBKDF2: A key derivation function

PBKDF2 is a Password-Based Key Derivation Function described in PKCS #5 [16], [13]. For providing better resistance against brute force attacks, PBKDF2 introduce CPU-intensive operations. These operations are based on an iterated pseudorandom function (PRF) which maps input values to a derived key. The most important properties to assure is that the iterated pseudorandom function is cycle free. If this is not so, a malicious user can avoid the CPU-intensive operations and, as described in [18], get the derived key by executing a set of functionally-equivalent instructions.

PBKDF2 inputs a pseudorandom function PRF, the user password p, a random salt s, an iteration count c, and the desired length len of the derived key. It outputs a derived key $DerKey$.

$$DerKey = PBKDF2(PRF, p, s, c, len) \tag{1}$$

More precisely, the derived key is computed as follows:

$$DerKey = T_1||T_2||\dots||T_{len} \tag{2}$$

where

$$T_1 = Function(p, s, c, 1)$$

$$T_2 = Function(p, s, c, 2)$$

$$\vdots$$

$$T_{len} = Function(p, s, c, len).$$

Each single block T_i — i.e., $T_i = Function(p, s, c, i)$ — is computed as

$$T_i = U_1 \oplus U_2 \oplus ... \oplus U_c \tag{3}$$

where

$$U_1 = PRF(p, s||i)$$

$$U_2 = PRF(p, U_1)$$

$$\vdots$$

$$U_c = PRF(p, U_{c-1})$$

The pseudorandom function applied to derive a key can be a hash function [14], cipher, or HMAC [3], [4], and [11]. In the sequel, unless otherwise specified, by PRF we will refer to HMAC with the SHA-1 hash function, which is the default as per [16], [9].

3 Linux Unified Key Setup

The Linux Unified Key Setup (LUKS) is a disk-encryption specification commonly implemented in Linux based operating systems. It is a platform-independent standard on-disk format developed by Clemens Fruhwirth in 2004 [9,8]. LUKS is based on a two level key hierarchy. It protects the master key using PBKDF2 as key derivation function. To solve the problem of data remanence — i.e., data continues to exist on hard disk even after it has been deleted — an anti-forensic splitter (AF-splitter) is adopted. This AF-splitter inflates and splits the master key before storing it on disk and, furthermore, uses a hash function as diffusion element.

A LUKS partition has a simple layout (see Figure 1). It includes the partition header, the key material (KM_1, KM_2, ..., KM_8), and the user encrypted data.

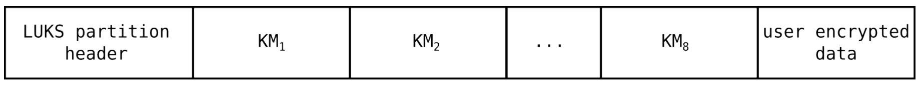

Fig. 1. LUKS partition header

The partition header contains information about salt, iteration counts, key slots (eight), used cipher, cipher mode, key length, hash function, master key checksum, start sector of key material, and so on [8]. Among all these parameters, we look more closely at salt and iteration counts because they allow to mitigate brute force attacks. In particular, the salt is fetched from a random source [9], while the iteration counts are automatically computed by making some run-time tests when the encrypted partition is generated. Salt and iteration counts are stored in plain text in LUKS partition header.

In addition, the solution adopted by LUKS has as many user key as there are key slots. Therefore, the same master key can be encrypted with eight different user keys, and stored in one of the eight key material sections.

3.1 Master key recovery

In order to recover the master key, we need a valid LUKS partition header. When a user key is provided, it unlocks one of the eight key slots. As shown in Figure 2, PBKDF2, an anti-forensic splitter, and a cipher are used to compute the master key. Such a key in turn will unlock the encrypted data.

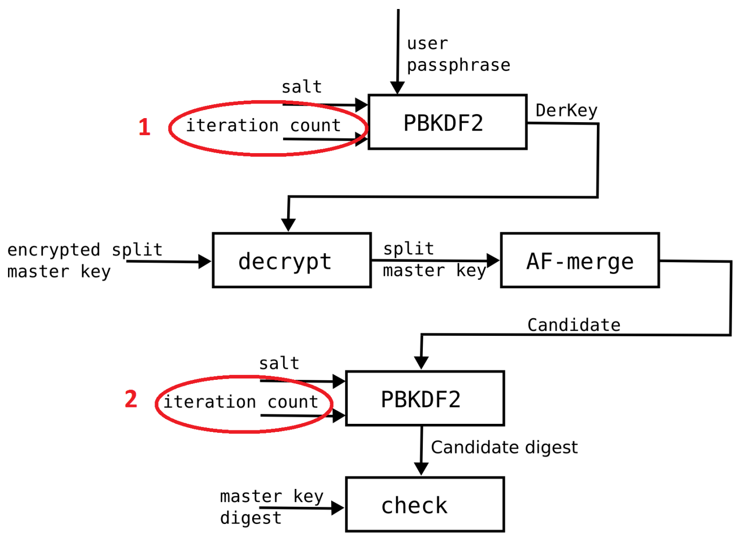

Fig. 2. Master key recovery process

More precisely, the following algorithm is processed:

Algorithm 1: Master key recovery process

```
1  Read the user password/passphare $p$;
2  Read salt $s$ from active key slot;
3  Read first iteration count $c$ from active key slot;
4  Use PBKDF2 to compute derived key $DerKey$;
5  Read the start sector of key material from active key slot;
6  Read the split master key from key material;
7  Decrypt the split master key using derived key $DerKey$;
8  Merge the split encrypted master key and obtain a candidate
   master key;
9  Read the second iteration count for computing the master key
   digest;
10 Use PBKDF2 to compute the candidate master key digest;
11 Compare such digest with those stored in the partition header;
12 If equal, the recovery is successful. Otherwise, the candidate is
   not the correct master key.
```

4 Analysis of a LUKS implementation

In Linux world, LUKS implementations are based on cryptsetup and dm-crypt. In order to mitigate the problem of brute force attacks based on weak user passwords, LUKS combined the ideas of salt and key derivation function (i.e., PBKDF2). Because salt parameter is known and user password may be guessed, we focus on iteration counts and their ability to slow down a brute force attack as much as possible. In particular, we try to understand where and how the iteration counts are used, how the choice of specific hash functions may affect the iteration count computation, and how unwitting users might significantly reduce the security of a LUKS implementation by setting aggressive power management options.

4.1 Iteration counts: where and how

Two iteration counts are involved in the key management process. The first iteration count is used to compute derived key (see point

4, Algorithm 1), while the second one is involved in the master key checksum process (see points 9-10-11, Algorithm 1).

Table 1. Average iteration counts involved in the key derivation process

CPU	OS	sha1	sha512	sha256	ripemd160
Intel Atom z520	Debian 7.7 x86	31,035	7,019	18,567	29,491
Intel Core 2 Duo T6670	Kali 1.0 x86	151,772	22,821	67,634	111,791
Intel Pentium 3556U	Xubuntu 14.04 x64	126,617	50,082	77,379	103,287
Intel Core i3 2310M	Fedora 20 x64	136,375	50,107	77,682	111,536
Intel Pentium T4500	Ubuntu 12.04 x64	147,904	56,380	85,167	119,366
Intel Core i5 3320M	Debian 7.7 x64	232,203	88,843	139,985	196,209
Intel Core i7 2860QM	Kubuntu 14.04 x64	248,671	90,225	123,904	179,947
Intel Core i7 4710MQ	ArchLinux x64	588,761	302,148	392,916	350,378

Table 2. Average iteration counts involved in the master key checksum process

CPU	OS	sha1	sha512	sha256	ripemd160
Intel Atom z520	Debian 7.7 x86	7,826	1,702	4,668	7,327
Intel Core 2 Duo T6670	Kali 1.0 x86	37,761	5,752	27,498	16,764
Intel Pentium 3556U	Xubuntu 14.04 x64	31,419	12,406	19,318	25,659
Intel Core i3 2310M	Fedora 20 x64	33,903	12,657	19,307	27,718
Intel Pentium T4500	Ubuntu 12.04 x64	36,913	14,009	21,495	29,951
Intel Core i5 3320M	Debian 7.7 x64	58,218	22,026	34,802	49,138
Intel Core i7 2860QM	Kubuntu 14.04 x64	54,371	19,353	30,926	44,927
Intel Core i7 4710MQ	ArchLinux x64	147,727	75,570	98,929	87,572

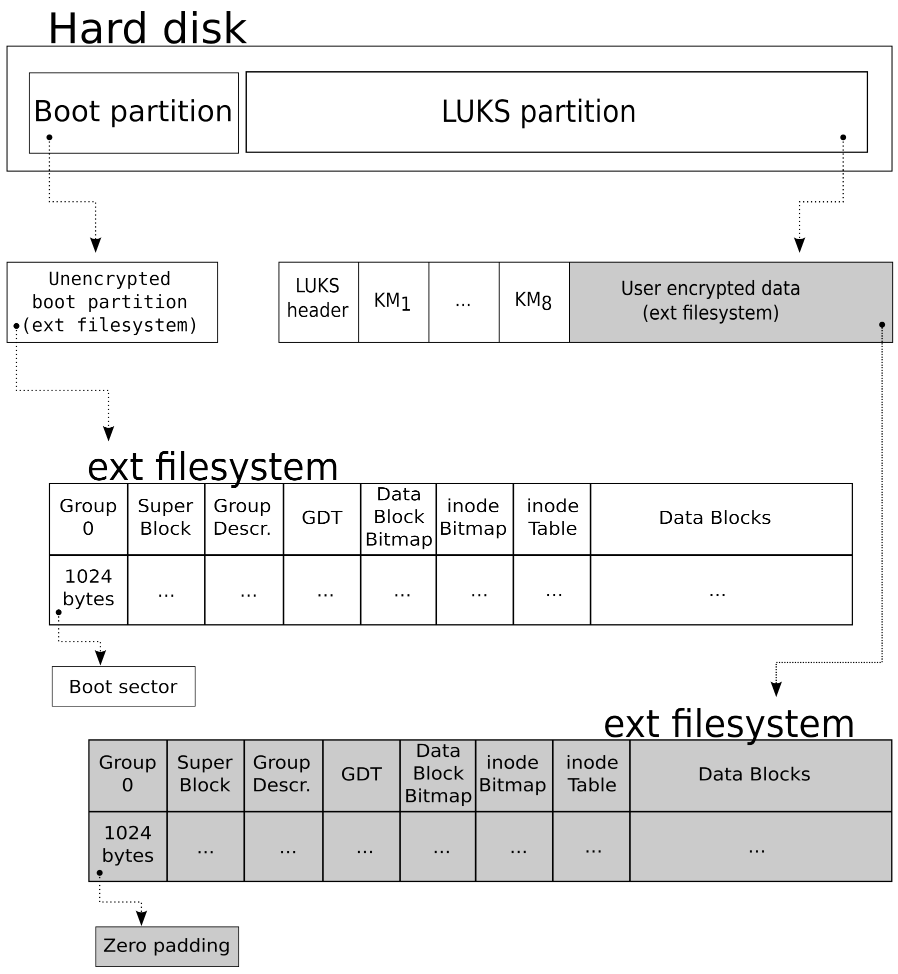

Fig. 3. The first 1024 bytes on EXT-family file systems

We experimentally observed that about 75-80% of the computational effort required to compute a derived key is generated by first iteration count (see Table 1), while the remaining 20-25% by second one (see Table 2). Unfortunately, the master key checksum process can be avoided exploiting the well known problem of file system structure. Indeed, on EXT-family file systems the first 1024 bytes are reserved for the boot sector (see Figure 3, unencrypted boot partition). When unused — recall that a hard disk can contain several partitions, each with their own boot sectors — it is set to zeros (see Figure 3, user encrypted data).

By decrypting the first bytes of the user encrypted data and checking if such bytes are zeros, we are able to understand if the candidate key is the correct master key or not. Hence, we substitute points 9-10-11 of Algorithm 1 with a decryption operation.

This means that, for all encrypted LUKS partitions the second iteration count can be avoided and the computational effort required to compute the master key can be reduced by about 20-25%.

4.2 Iteration counts and hash functions

To better understand how the iteration counts are handled — recall that they are automatically computed by making some run-time tests — we experimentally collected several partition headers related to a number of encrypted devices.

We installed on our laptops a 32-bit or 64-bit operating system (e.g. Debian 7.7 x86, Fedora 20 x64, Kubuntu 14.04 x64, Kali 1.0 x86, and so on), libgcrypt 1.6.3 [2] and cryptsetup 1.6.6 [1]. These are the latest releases available at the time of testing.

To be sure that such values are not conditioned by external factors, e.g. running programs, we collected 3200 partition headers. More precisely, for each processor (eight) and each hash function (four) listed in Tables 1 and 2, we execute 100 runs for a total of $8 \times 4 \times 100 = 3200$ partition headers collected. Then, we read salt and iteration counts stored in each partition header. Tables 1 and 2 shown the average values collected. Notice that the variation across runs is observed to be less than 0.4%.

As expected, devices with a different hardware configuration generate different iteration count values. For example, the values collected for SHA1 run on average between 588,761 (Intel Core i7 4710MQ) and 31,035 (Intel Atom z520), with higher values corresponding to a more powerful processor.

Surprisingly, even small changes in software, such as choose a different function of the SHA family, may considerably decrease the iteration count values. Notice the differences between 67,634 and 22,821 (Intel Core due duo T6670, SHA256 vs SHA512), or 18,567 and 7,019 (Intel Atom z520, SHA256 vs SHA512), or 139,985 and 88,843 (Intel Core i5 3320M, SHA256 vs SHA512). This abnormal behavior was not found in all cases tested. For example, it is partially mitigated in i7 4710MQ processor where the average values collected are 392,916 and 302,148 (Intel Core i7 4710MQ, SHA256 vs SHA512).

The approach adopted by LUKS in defining iteration count values does not always sound good. We found it curious that the iteration counts related to SHA-256/512 are considerably smaller than those of SHA-1. Although there is no reason why this should not happen when we talk about the security against password guessing, from an user's point of view, SHA-256 and SHA-512 are still considered more secure than SHA-1, therefore a FDE solution based on SHA-2 is expected to be stronger. We notice that the CPU time spent to compute a list of master key candidates based on SHA-256/512 costs less than one based on SHA-1. Hence, it is easier to attack a FDE solution which makes use of a safer hash function (e.g., SHA256 or SHA512) rather than one which uses a less secure function (e.g., SHA-1).

Furthermore, the computational time spent to compute a list of master key candidates does not only depend on the iteration count values. Even the number of fingerprints required to compute a single iteration affects the total execution time. Indeed, assuming that the decryption function involved in the master key recovery process is AES (i.e. the default choice), we need a 256 bits derived key. A SHA-1 fingerprint is only 160 bits in length and cannot be used as derived key. As described in Equation 2, a second fingerprint is necessary — i.e., $DerKey = T_1||T_2$. On the other hand, SHA-256 and SHA-512 generate enough bits to compute a derived key, hence $DerKey = T_1$. This means that, at equal iteration count values, a FDE solution based on HMAC-SHA1 slow down the brute force process better than one based on HMAC-SHA256 or HMAC-SHA512.

To point out this finding, we set the first iteration count to 500,000, and try to compute a list of 250,000 master key candidates using a number of hash functions. Figure 4 can help us to visualize the time necessary to execute a brute force attack on a i7 processor. Note that the gap between SHA-1, SHA-256, and SHA-512 hash functions is partially mitigated by compensatory mechanisms such as using a computationally more complex hash function.

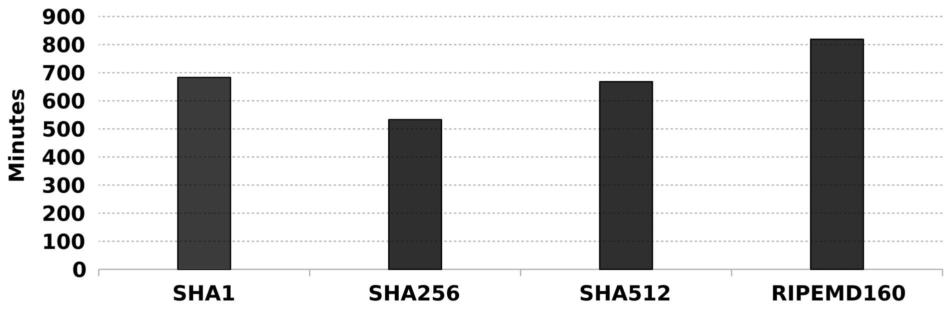

Fig. 4. Time spent to compute a list of 250,000 master key candidates

4.3 Iteration counts and power management

Another important feature that users have to take into account during encryption operations are the power management options. A common way to increase the battery life of devices is to enable aggressive power saving policies. Such policies save power, but they also impact performance by lowering CPU clock speed. Hence, the iteration count values fall down even further.

Table 3. Maximum and minimum CPU frequency of some devices

CPU	OS	Max Freq (Plugged)	Min Freq (Unplugged)
Intel Atom z520	Debian 7.7 x86	1.33 GHz	0.80 GHz
Intel Pentium 3556U	Xubuntu 14.04 x64	1.70 GHz	0.80 GHz
Intel Core i7 4710MQ	ArchLinux x64	3.50 GHz	1.20 GHz

To better understand this behavior, we install a well-known Linux power management package (i.e., Laptop Mode Tools package version 1.66) and reduce the CPU frequency as much as possible (see Table 3). Then, we run a number of tests and experimental results are reported in Table 4.

Note that the reduction of the iteration count values is proportional to the reduction of the CPU frequency. Indeed, for the i7 Core tested, power save settings imply a lowering of iterations by about a factor 3. Pentium, instead, has half the iteration counts, and Atom has about a third less. These results suggest that power saving policies might

Table 4. Power saving policies and their impact on the iteration count values

	SHA1		SHA512	
CPU	Plugged	Unplugged	Plugged	Unplugged
Intel Atom z520	31,035	18,693	7,019	4,288
Intel Pentium 3556U	126,617	62,969	50,082	25,161
Intel Core i7 4710MQ	588,761	202,143	302,148	104,216
	SHA256		RIPEMD	
CPU	Plugged	Unplugged	Plugged	Unplugged
Intel Atom z520	18,567	11,094	29,491	17,813
Intel Pentium 3556U	77,379	38,714	103,287	51,603
Intel Core i7 4710MQ	392,916	135,207	350,378	121,483

have an important impact on the iteration count values, hence, on the strength of the FDE solution adopted.

4.4 Testing

Our testing activity is not intended to decrypt a FDE solution — PBKDF2 can be parallelized on GPU architecture or specialized hardware (ASIC/FPGA) and interested readers can find more information about this topic in [5], [7], and [17] — but only to evaluate how the choice of PBKDF2 parameters and power management options can affect the security of a full disk encryption solution.

We implemented a brute-force attack based on a password-list of 250,000 master keys. Cryptographic hash functions and PBKDF2 have been implemented using standard OpenSSL library. We run our code on a laptop equipped with an i7 4710MQ processor. No GPUs have been used. The brute force attack has been executed six times. For each CPU listed in Table 3, we target two LUKS partitions collected using the following configuration options:

1. default iteration count values, AES-256 XTS mode, HMAC-SHA1, laptop plugged in;
2. default iteration count values, AES-256 XTS mode, HMAC-SHA512, laptop unplugged;

Figure 5 visualizes the time spent attacking a FDE solution. Although this is a toy example — 250,00 master keys are an approximation of the size of a dictionary — we can easily identify the gap between different kinds of approach. The second approach

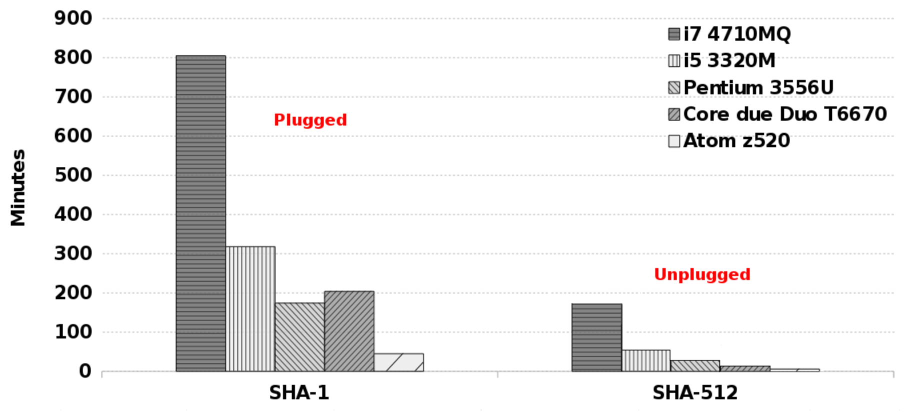

Fig. 5. A toy example: time spent attacking a FDE solution

abruptly reduce the timeframe for brute forcing, showing how the simple choice of configuration parameters may affect a FDE solution based on LUKS. Note that such an attack takes into account all the weaknesses described in Section 4.1, 4.2, 4.3 and in [18].

5 Discussion and conclusions

In this paper, we addressed the security of a Full Disk Encryption solution based on LUKS specification. Such a solution aims to prevent data leakage even in the event that devices are lost or stolen. We analyzed the key management process used to compute and store the encryption key and how the problem of brute force attacks based on weak user passwords has been mitigated.

We identify a number of issues that should be assessed and faced when a full disk encryption is implemented.

– Firstly, the iteration count values are used to slow down a brute force attack, therefore, they should not be too small. Experimental results show that sometimes they are.
– Secondly, power management options should not affect the strength of a FDE solution. Testing results show that aggressive power-saving approaches may have a relevant impact on the iteration count values, hence, on the strength of the solution adopted.

– Thirdly, from an user's point of view a FDE solution based on HMAC-SHA256, or HMAC-SHA512, is expected to be much stronger than one based on SHA-1, and be far more resistant to brute-force attacks. Our testing disprove this.
– Fourthly, the well-known problem of EXT family file system (i.e. the first block group contains the boot record or is set to zero) allows attackers to substitute the master key checksum process by a simple decryption operation. The CPU-intensive operations used to compute a derived key should not be avoided by executing a set of functionally-equivalent instructions.
– Fifthly, master keys stored on disk are protected with user keys which should have a minimum length requirement in order to prevent a brute force attack. We experimentally observed that a number of distribution such as Debian, Ubuntu, and ArchLinux have no minimum length requirement, while Fedora has (but only eight characters).

Our testing activities show that unwitting users can significantly reduced the security of LUKS by setting "stronger" hash function (e.g. HAMC-SHA512 or HAMC-SHA256) and enabling aggressive power management options. Because attacks always get better and Moore's Law will continue to march forward, we strongly suggest to increase default iteration count values whenever a user key is defined. Unfortunately, the most common user approach is to leave the default values unchanged, although a number of parameters can be easily adjusted by user as desired.

6 Acknowledgment

This research was partially supported by a grant from Università degli Studi di Milano, "Piano di Sostegno alla Ricerca UNIMI 2015-2017".

References

1. Cryptsetup 1.6.6 release. `https://gitlab.com/cryptsetup/cryptsetup` (2015)
2. Libgcrypt 1.6.3 release. `https://www.gnu.org/software/libgcrypt/` (2015)

3. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Proc. of Advances in Cryptology—CRYPTO96. pp. 1–15. Springer (1996)
4. Bellare, M., Canetti, R., Krawczyk, H.: Message authentication using hash functions—the hmac construction. RSA Laboratories CryptoBytes 2(1), 12–15 (1996)
5. Dürmuth, M., Güneysu, T., Kasper, M., Paar, C., Yalcin, T., Zimmermann, R.: Evaluation of standardized password-based key derivation against parallel processing platforms. In: Proc. of ESORICS12, pp. 716–733. Springer (2012)
6. Elenkov, N.: Android Security Internals. No Starch Press (2014)
7. Frederiksen, T.K.: Using cuda for exhaustive password recovery (2011), `http://daimi.au.dk/~jot2re/cuda/resources/report.pdf`
8. Fruhwirth, C.: New methods in hard disk encryption (2005), `http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf`
9. Fruhwirth, C.: LUKS On-Disk Format Specification Version 1.2.1 (2011), `http://wiki.cryptsetup.googlecode.com/git/LUKS-standard/on-disk-format.pdf`
10. Gutmann, P.: Secure deletion of data from magnetic and solid-state memory (1996), `https://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`
11. Krawczyk, H., Bellare, M., Canetti, R.: Hmac: Keyed-hashing for message authentication. Internet RFC 2104 (1998)
12. Morris, R., Thompson, K.: Password security: A case history. Communications of the ACM 22(11), 594–597 (1979)
13. NIST: SP 800-132: Recommendation for password-based key derivation (2010)
14. NIST: FIPS PUB 180-4: Secure Hash Standard (Mar 2012), `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`
15. Ponemon Institute: The billion dollar lost laptop problem (2010), `http://newsroom.intel.com/servlet/JiveServlet/download/1544-16-3132/The_Billion_Dollar_Lost_Laptop_Study.pdf`
16. RSA Laboratories: Pkcs #5 v2.1: Password based cryptography standard (2012)
17. Schober, M.: Efficient password and key recovery using graphic cards. Diploma Thesis, Ruhr-Universität Bochum (2010)
18. Visconti, A., Bossi, S., Ragab, H., Caló, A.: On the weaknesses of PBKDF2. In: Proc. of the 14th International Conference on Cryptology and Network Security, CANS 2015. Springer International Publishing, LNCS 9476 (2015)