Masaryk University
Faculty of Informatics

Argon2 security margin for disk encryption passwords

Master's Thesis

Bc. Vojtěch Polášek

Brno, Fall 2018

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Vojtěch Polášek

Advisor: Ing. Milan Brož

Abstract

«abstract»

Keywords

«keywords»

Contents

1 Introduction **3**

2 Password hashing and key derivation functions **5**

2.1 *Definitions* 5

2.2 *Why do we need PBKDFs?* 6

2.3 *PBKDFs and disk encryption* 7

2.3.1 LUKS 8

2.3.2 usage of PBKDFs in LUKS version 1 8

2.3.3 Usage of PBKDFs in LUKS version 2 11

2.4 *Attacks on PBKDFs* 12

3 State of the art PBKDFs **13**

3.1 *PBKDF2* 13

3.2 *Argon2* 16

3.2.1 Algorithm 16

3.3 *Scrypt* 16

3.3.1 Algorithm 16

3.4 *Other PBKDFs* 16

4 The price of an attack **17**

4.1 *Attacker* 17

4.2 *Tools* 17

4.2.1 Hardware 17

4.2.2 Software 17

4.3 *Cost model* 17

4.3.1 Hardware 17

4.3.2 Energy 17

4.3.3 Software 17

4.4 *Real world cost estimation* 17

5 Attacking LUKS **19**

5.1 *Testing methodology* 19

5.2 *Analysis of results* 19

6 Conclusions **21**

A LUKS attack results **27**

List of Tables

Contents

1 Introduction

2 Password hashing and key derivation functions

2.1 Definitions

This thesis deals with various cryptographic terms including password hashing and key derivation. This section briefly explains some of them. Note that reader of this thesis is expected to have at least basic knowlege of cryptography and computer security.

A *hashing function* is a function which receives an input of arbitrary length and produces an output of specified shorter length, effectively compressing the input. These functions are used in many areas such as effective data retrieval [11]. *Cryptographic hashing functions* are subset of *hashing functions* and they have to meet certain properties, namely preimage resistance, second preimage resistance and collision resistance. We will consider only cryptographic hash functions in this thesis.

Password hashing is a process in which a password is supplied to a hash function. This is de facto standard method of storing of saved passwords in operating systems and applications. In case that an attacker gets hold of such password hashes, it should be ractically infeasible for an attacker to derive he original password. Therefore hashing functions are used also in process of password verification, during which the entered password is hashed and compared with the stored hash.

This thesis is not going to deal with *password hashing*. However, many *key derivation functions* described below meet desired properties for being used in the password hashing process. However, this thesis is focused primarily on key derivation and verification of cryptographic keys during disk encryption.

Key derivation functions are contained in subset of *hash functions*. Their basic purpose is to take an input and produce an output which can be used as a cryptographic key. The input is usually a password or other material such as biometric sample converted into binary form. These materials could be of course used as cryptographic keys on their own but they often lack properties of a good cryptographic key such as sufficient entropy or length.

A *cryptographic salt* is often used during process of key derivation. The purpose of salt is to prevent attacks which use precomputed tables such as Rainbow tables [13]. Salt introduces another factor which influences a derived key. It means that it is no longer dependent only on passhprase. For example suppose that 32 bit long integer is used as a salt. In that case there are 2^{32} possible keys derived from the same passphrase. This makes precomputing attacks effectively infeasible. Salt is usualy stored unobfuscated together with hashed material.

2.2 Why do we need PBKDFs?

Today, as more and more private information is stored on various kinds of media and transfered over the Internet, it is becoming crucial to protect it from being accessed or changed by unauthorised actors. Although there are several interesting authentication options such as biometrics, passwords or passphrases are still the most common method.

Considering passwords we are facing a problematic situation. Organisations and services provide guidelines or requirements which should help an user to choose a strong password [8] [14]. Important parameters are password length (in characters), password complexity, uniqueness and others. By complexity I mean amount and diversity of used characters (letters, numbers, symbols, emojis…) and by uniqueness I mean the fact that the password does not contain easily guessable or predictable sequences. See mentioned policies for example.

As shown by researches, users tend to circumvent such policies by finding loopholes in them.

What more, passwords them selves are not good cryptographic material which should be used as a cryptographic key. There are surprisingly many reasons. They are usually not sufficiently long. Because they are composed of printable characters, they do not meet the requirement of being uniformly distributed. If they should be remembered, they will probably contain dictionary words, which lessens their entropy even more. See [11, section 5.6.4] for short but interesting analysis.

PBKDF stands for password-based key derivation function. The goal of these functions is to derive one or more cryptographic keys from a password or a passphrase. This key should be pseudorandom and sufficiently long to make brute-force guessing as time-consuming as possible. As stated above, they are a subset of cryptographic hashing functions.

Lately PBKDFs are taking another specific task. Due to availability of GPUs, FPGAs and ASICs, there are new possibilities in running functions in parallel computing environment [see 12, chapter 4]. This increases effectivity of brute-force attacks. PBKDFs try to defend against such attacks by using salt and and function-specific parameters like iteration count etc. See section 2.4 for more details.

Examples of PBKDFs include Argon2, PBKDF2, Scrypt, Yscrypt and more. See chapter 3 for comparison of several functions.

2.3 PBKDFs and disk encryption

Disk encryption is a very good use case for usage of PBKDFs. Used encryption algorithms require cryptographic keys of certain length [9]. It is also important to consider the fact, that it is usually not desirable to change the encryption key often because reencryption of whole disk takes considerable amount of time. Let aside the fact that if an attacker gains permanent access to such an encrypted disk, the key cannot be changed at all and they may have extensive time period during which they can manage to crack the key.

By looking at [16] we can see that PBKDFs are used in many types of disk encryption software. Note that this list mentions only PBKDF2 as this has been most used PBKDF since recent times. PBKDF2 is for example used in LUKS version 1 [6], FileVault software used by macOS [1], CipherShed disk encryption software [4], Veracrypt disk encryption software [9] and more.

In 2013 there was initiated a new open competition called Password hashing competition. Its goal was to find a new password hashing function which would resist new attacks devised against those functions [2]. The winner was function named Argon2. It is already used for example in LUKS version 2 [3].

2.3.1 LUKS

LUKS stands for Linux Unified Key Setup. This project started to be developed by Clemens Fruhwirth as a reaction to several incompatible disk encryption schemes which coexisted at the same time at the begining of 21st century. At certain point there existed three incompatible disk encryption schemes which varried from Linux distribution to Linux distribution. If an user created an encrypted disk, they couldn't be sure if they will be able to encrypt the disk with a different distribution or even with a new version of the same distribution.

LUKS began as a metadata format for storing information about cryptographic key setup. However, Fruhwirth discovered that to design a proper metadata format, he needs to knoww enough information about key setup process [7]. Therefore, he created TKS1 and TKS2. These are templates for the key setup process. Together with LUKS they ensure safe and standardized key management during disk encryption. After some user feedback, LUKS on-disk specification version 1.0 was created in 2005 [6]. Currently the latest version is LUKS on-disk specification version 2 [3].

The reference implementation of both versions of LUKS is called libcryptsetup. The userspace interface is called Cryptsetup. In the following two subsections, default parameters and information concerning command line switches are specific to this implementation.

2.3.2 usage of PBKDFs in LUKS version 1

PBKDF2 function is used as a key derivation function in LUKS version 1. It is used during master key initialisation, adding of a new password, master key recovery, and also during password changing because this operation is actually composed of previously mentioned operations. During all operations it internaly uses a hash algorithm specified by user during initialisation of the LUKS header. By default, SHA1 algorithm is used.

During initialisation, the PBKDF2 function is used to create a checksum of a master key. This key is subsequently used for symmetric encryption of actual data stored on the encrypted disk. The function receives following parameters:

masterKey a new randomly generated master key of user specified length

phdr.mk-digest-salt a random number 32 bytes long which is used to prevent attacks against password using precomputed tables [see 11, section 5.6.3]

phdr.mk-digest-iteration-count number of iterations for PBKDF2, see section 3.1

LUKSDIGESTSIZE length of he computed digest in bytes, default is 20

The generated 20 bytes long checksum is stored in the LUKS header together with the iteration count and salt. Please note that the *phdr.mk-digest-iteration-count* parameter is obtained by performing a benchmark with minimum of 1000 iterations.

During adding of a new password, PBKDF is used to process the passphrase supplied by user. It receives following parameters:

password a passphrase supplied by an user

ks.salt randomly generated salt used for this particular key slot with length of 32 bytes

ks.iteration-count number of PBKDF2 iterations

MasterKeyLength length of the derived key

The resulting key derived from the passphrase is used to encrypt the master key. This key has to be present in memory either because initialisation happened recently or it was successfully recovered through a different key stored in different key slot. Together with the encrypted master key, the salt and iteration count are also written into the key slot.

Note that the ks.iteration-count parameter can be influenced by user in several ways [5]. One possibility is to specify number of iterations directly with the command line option `--pbkdf-force-iterations`.

Another option is to specify the iteration time through `-i` or `--iter-time` command line options. This option expects a number which signifies number of milliseconds which should be spent calculating the hash. A benchmark is used to calculate number of iterations which corresponds to this time. If no option is specified then the default iteration time of 2000 milliseconds is used.

The function is also used during the master key recovery. This process is performed while unlocking the encrypted partition. During key recovery the PBKDF is actually used twice for every key slot until the master key is decrypted or there are no more key slots to try. First it is used to derive a decryption key from a passphrase supplied by an user. Then this decryption key is used to decrypt the encrypted master key stored in the current key slot. The result is called the candidate master key because we are still not sure if the passphrase was correct. This candidate is again hashed with PBKDF and finally compared with hash of the master key stored in the LUKS header. If hashes match then the passphrase was entered correctly and the master key can be used.

In the first case the function receives following parameters:

pwd user supplied passphrase

ks.salt the salt value read from currently tried key slot

ks.iteration-count the iteration count read from currently tried key slot

masterKeyLength length of the derived key

In the second case the function receives following parameters:

masterKeyCandidate candidate master key, see above

ph.mk-digest-salt the salt value which was used during the initialisation phase and stored in the header

ph.mk-digest-iter number of PBKDF2 iterations used during the initialisation phase and also stored in the header

LUKS_DIGEST_SIZE 20 bytes

The process of changing a password is composed of previously mentioned operations and hence I am not mentioning it here in greater details. To sum it up, firstly the master key is recovered, then a new password is added to a new key slot and the previous one is revoked.

Both password-based encryption and password checking require additional cryptographic primitives which process the derived key. For encryption, reference implementation of LUKS uses aes-xts-plain64 and for hashing it uses sha256. Usage of PBKDF2 requires underlying pseudorandom function. In case of LUKS version 1, default PRF is SHA1. Alternatively, it is possible to choose SHA256, SHA512, ripemd160 or whirlpool.

2.3.3 Usage of PBKDFs in LUKS version 2

LUKS version 2 extends LUKS version 1 and uses similar principles. Therefore, I will focus on differences between version 1 and 2 which are related to usage of PBKDFs. For detailed list of changes see [ ]luks2section 1.1.

The new version supports configurable algorithms for encryption, hashing and also key derivation. That means that the set of algorithms can be extended as new are developed and some are obsoleted. Note that there still exist some requirements for algorithms provided by cryptographic backend [ ]luks2section 4.6. The backend has to support SHA-1 and SHA-256 hashing functions, PBKDF2, Argon2i and Argon2id key derivation functions and AES-XTS symmetric encryption.

LUKS2 introduces PBKDF memory hard functions Argon2i and Argon2id which are described in 3.2. Argon2 functions should offer increased resistance to brute-force attacks.

The volume key digest is no longer limited by length of 20 bytes, because it no longer relies on SHA-1 hashing function. The processes described in subsection 2.3.2 are the same in LUKS version 2. The same applies for situations in which PBKDFs are used.

As stated in subsection 2.3.2, in case of PBKDF2 user can influence number of iterations directly or specify approximate time required for processing of the passphrase. This stays the same for LUKS2. Functions from Argon2 family introduce two additional parameters; memory cost and parallel cost. Both parameters can be specified through

`--pbkdf-memory` and `--pbkdf-parallel` command line parameters respectively. If options are not specified, they are benchmarked.

2.4 Attacks on PBKDFs

3 State of the art PBKDFs

3.1 PBKDF2

PBKDF2 is a password-based key derivation function defined in RFC 8018 [10]. This RFC thoroughly describes two use cases of PBKDF2; password-based encryption scheme and password-based message authentication scheme. Other mentioned use cases include password checking and derivation of multiple keys from one password. As shown in 2.3.2, LUKS version 1 uses PBKDF2 for password checking and derivation of key for encryption or decryption of master key.

the function requires four input parameters; passphrase, cryptographic salt, iteration count and length of a key to be derived. Moreover, the function requires a pseudorandom function (PRF) which is used in process of key derivation.

The term *Passphrase* in this context means any data which are source for subsequent key derivation process. Usually it is a password entered by user. The *cryptographic salt* is represented by randomly generated number.

The purpose of *iteration count* parameter is to defend against brute force and dictionary attacks performed on PBKDF2. The iteration count prolongs the time which is needed to derive a single key. Technically, the iteration count signifies number of successive runs of chosen PRF for every block of the derived key. In general, the *iteration count* should be chosen as large as possible, taking into account the fact that the processing time should be acceptable for the end user [15]. According to the cited document, minimum iteration count should be 1000 iterations and for critical security systems a count of 10000000 iterations is appropriate.

The function is described by following algorithm. Verbal description is also provided. Following abbreviations and conventions are used in the algorithm and description:

P an octet string representing a passphrase

S an octet string representing a cryptographic salt

C a positive integer representing iteration count dkLen a positive integer representing length of the derived key counted in octets

DKan octet string representing the derived key

PRF - a pseudorandom function

hLen - length of output of chosen pseudorandom function counted in octets

CEIL(x) the ceiling function returning the smallest integer which is greater or equal to X

F a helper function for better description

|| concatenation of strings

INT(x) a big§-endian encoding of integer x

At the begining the algorithm checks if the desired length of the key does not exceed $2^{32} - 1$. If it does, it exits immediatelly. Then it processes the input and creates output key in blocks. Every block has length of hLen octects, except for the last one which can have shorter length.

In the pseudocode there is defined function *F* which is applied to every block. Results of such applications are finally concatenated and returned as the resulting derived key. This function performs *c* iterations of underlying pseudorandom function *PRF*. The *PRF* takes a passhprse as the first parameter and result of previous iteration as the second parameter. The only exception is the first iteration where the second parameter is concatenation of salt and binary representation of the block index *i*. Results of all iterations are xored and returned as a particular block of the derived key. Finally, all blocks are concatenated and returned as the derived key.

Notice that the function F can be rewritten to be quickly computed in parallel computing environments such as GPUs. See [12]section 4.1 for more details.

```
input  : P, S, C, dkLen
output : DK
if $dkLen > (2^{32} - 1) * hLen$ then
    return Derived key too long
end
$L \leftarrow$ CEIL($dkLen/hLen$)
/* l is the number of hLen-octet blocks in the derived
   key                                                  */
$r \leftarrow dkLen - (l - 1) * hLen$ /* r is the number of octets in
   the last block                                       */
for $i \leftarrow 1$ to $l$ do
    $T_i \leftarrow$ F($p, s, c, i$)
end
return $t_1||t_2|| \ldots t_l[0 \ldots (r-1)]$
Function $F$ $(s, p, c, i)$
    $u_1 \leftarrow$ PRF($P, S \,||\, INT(i)$)
    for $j \leftarrow 2$ to $c$ do
        $u_j \leftarrow$ PRF($P, u_{j-1}$)
    end
    return $u_1 \oplus u_2 \oplus \ldots \oplus u_c$
```

Algorithm 1: PBKDF2 function algorithm

3.2 Argon2

3.2.1 Algorithm

3.3 Scrypt

3.3.1 Algorithm

3.4 Other PBKDFs

4 The price of an attack

4.1 Attacker

4.2 Tools

4.2.1 Hardware

4.2.2 Software

4.3 Cost model

4.3.1 Hardware

4.3.2 Energy

4.3.3 Software

4.4 Real world cost estimation

5 Attacking LUKS

5.1 Testing methodology

5.2 Analysis of results

6 Conclusions

Bibliography

[1] J. Appelbaum and R.-P. Weinman. (Dec. 29, 2006). Unlocking filevault - an analysis of apple's disk encryption system, [Online]. Available: `https://events.ccc.de/congress/2006/Fahrplan/attachments/1244-23C3VileFault.pdf` (visited on 10/28/2018).

[2] J.-P. Aumasson. (Dec. 6, 2015). Password hashing competition, [Online]. Available: `https://password-hashing.net/` (visited on 09/08/2018).

[3] M. Broz, *Luks2 on-disk format specification*, version 1.0.0, Aug. 2, 2018. [Online]. Available: `https://gitlab.com/cryptsetup/cryptsetup/blob/a1b606803f6d50e0476867fe9d284399504559a3/docs/on-disk-format-luks2.pdf` (visited on 09/17/2018).

[4] CipherShed Project, *Ciphershed - secure encryption software*, version 0.7.3.0, Dec. 19, 2014. [Online]. Available: `https://github.com/CipherShed/CipherShed/raw/v0.7.3.0-dev/doc/userdocs/guide/CipherShed-User-Guide-0.7.3.0.pdf` (visited on 10/28/2018).

[5] *Cryptsetup(8(maintenance commands*.

[6] C. Fruhwirth, *Luks1 on-disk format specification*, version 1.2.3, Jan. 20, 2018. [Online]. Available: `https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf` (visited on 09/18/2018).

[7] C. Fruhwirth, "New methods in hard disk encryption", Aug. 2005. [Online]. Available: `http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf` (visited on 10/03/2018).

[8] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Y.-Y. Choong, K. K. Greene, and M. F. Theofanos. (Jun. 2017). Nist special publication 800-63b, Digital identity guidelines - authentication and lyfecycle management, [Online]. Available: `https://doi.org/10.6028/NIST.SP.800-63b` (visited on 10/25/2018).

[9] IDRIX, *Veracrypt - free open source disk encryption with strong security for the paranoid*, Veracrypt Documentation - Header Key Derivation, Salt, and Iteration Count.
[Online]. Available: `https://www.veracrypt.fr/en/Header%20Key%20Derivation.html` (visited on 10/28/2018).

[10] B. Kaliski and A. Rush, "Pkcs #5: Password-based cryptography specification", RFC Editor, RFC 8018, Jan. 2017, p. 40.
[Online]. Available: `https://tools.ietf.org/html/rfc8018` (visited on 10/28/2018).

[11] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, ser. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2014, ISBN: 9781466570269.
[Online]. Available: `https://books.google.cz/books?id=OWZYBQAAQBAJ`.

[12] O. MOSNÁČEK, "Key derivation functions and their gpu implementations", Bachelor thesis, Masaryk university, Faculty of informatics, Brno, 2015. [Online]. Available: `https://is.muni.cz/th/sah52` (visited on 09/05/2018).

[13] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off", in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 617–630, ISBN: 978-3-540-45146-4.

[14] SANS Institute. (Oct. 2017). Password construction guidelines, [Online]. Available: `https://www.sans.org/security-resources/policies/general/pdf/password-construction-guidelines` (visited on 10/25/2018).

[15] M. S. Turan, E. B. Barker, W. E. Burr, and L. Chen, "Sp 800-132. recommendation for password-based key derivation: Part 1: Storage applications", Gaithersburg, MD, United States, Tech. Rep., 2010.

[16] Wikipedia contributors. (2018). List of pbkdf2 implementations — Wikipedia, the free encyclopedia, [Online]. Available: `https://en.wikipedia.org/w/index.php?title=List_of_`

PBKDF2_implementations&oldid=863291654 (visited on 10/28/2018).

A LUKS attack results