Masaryk University
Faculty of Informatics

Argon2 security margin for disk encryption passwords

Master's Thesis

Bc. Vojtěch Polášek

Brno, Spring 2019

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

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 [17]. *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 based on *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 [22]. 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 [13] [26]. 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 [17, 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 based on 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 20, 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 [14]. 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 [30] 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 [11], FileVault software used by macOS [1], CipherShed disk encryption software [8], Veracrypt disk encryption software [14] 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 [7].

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 [12]. 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 [11]. Currently the latest version is LUKS on-disk specification version 2 [7].

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, SHA256 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 17, 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, PBKDF2 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 [10]. 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 other 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. The number of iterations is either benchmarked or it can be manually specified through `--pbkdf-force-iterations`.

Note that the number of parallel threads will be always at most 4 and it will decrease if not enough CPUs are online at the time of its usage. In Cryptsetup version 2.6.0 the default unlocking time is set at 2000 miliseconds and the default memory cost is set at 1048576 kB. Both of default values can be changed at compilation time.

2.4 Attacks on PBKDFs

History of password cracking is as old as computer passwords them selves. People crack passwords for two main reasons. Either they want to recover a forgotten password, or they want to recover password of someone else, later being able to use it for authentication purposes. There exist two main techniques for password cracking; brute force attacks and dictionary attacks. The first form of attacks tries to guess the password by trying all passwords of given length composed of all combinations of given characters. Dictionary attacks exploit the fact that people often use passwords containing words found in language dictionaries. It means that if an attacker tries paswords containing dictionary words or permutations, they have quite good chance of success.

At the begining passwords were stored on computer systems in plain text, protected only by the fact that users shouldn't be able to read passwords of other users. But soon it appeared that plain text passwords can be revealed for example by badly designed software permissions. This is shown in case of Allan Scherr, who misused capabilities of a printing program to print out whole password file [9]page 37. Since that time, passwords started to be hashed.

. This time marks begining of the never-ending fight between authors of hash functions and people trying to crack passwords hashed by them. First hash functions were really simple and they were definitely not cryptographically secure, such as hash mechanism used in Multics. This mechanism squared numerical form of each password and applied a bit mask with AND operation [28]. This increased number of guesses but only negligibly compared to modern functions.

The first cryptographic hash came with Robert Morris and his Crypt function. Crypt used up to Unix 6th edition mimicked the M209 cipher machine from World War II. and proved not very secure because the algorithm could be recoded in a way which allowed to test passwords in very short period of time (1.25 milliseconds per password) [24]. Later version used since Unix 7th edition employed the DES block cipher. This cipher was at that time very slow if implemented in software. The password entry program also introduced two new concepts; automated proactive password strength checking and *cryptographic salt* (12 bit random number at that time). Currently for example the LUKS2 specification uses 32 bytes long salt.

During 1980s there happened some password cracking contests and hash functions were also improved. Some of them were made deliberately slow to slow down potential attacker. If this measure was not effective enough, more iterations of hashing function could be used. During 1990s many password cracking programs appeared including John the Ripper, Crack, LOphtCrack etc. [18]. Most of those programs tried to improve the cracking speed by optimizing underlying algorithms and later by using CPU parallelism.

The concept of *KDFs* started being studied in late 1990s. The RFC 2898 for PBKDF2 was released in 2000 and it started being used primarily for key derivation in many applications such as WinZip, Opendocument, Truecrypt or Android. However, it was also used for actual password hashing for example in Mac OS X 10.8. Note that PBKDF1 exists, but it is not recommended because of its limited key length (20 bytes at best) and it is provded solely for backward compatibility. PBKDF2 introduced configurable pseudorandom function, number of iterations and derived key length. These parameters allow flexibility while choosing trade-off between security and user experience.

However, in 2007 there appeared first password crackers using parrallel computing capabitilies of GPUs [4]. Other password crackers followed; Whitepixel in 2010, Oclhashcat in 2012, John the ripper in 2012. Until 2012 software could recover primarily MD5 and NTLM hashes, but Oclhashcat introduced recovery of many other hashes.

As far as PBKDF2 is considered, its resistance to password cracking using GPUs or even ASICs/FPGAs is currently not ideal [20]section 7. PBKDF2 does not offer support for parallelism while used as suggested. However, at the same time its low memory requirements and

GPU friendly algorithm (see algorithm 1 bring advantage to the attacker. As shown in [20], it was possible to improve cracking speed of LUKS passwords fourty times. This required rewriting the function for GPUs. Moreover, it is shown that proper optimization of underlying algorithms can greatly increase PBKDF2 performance even without GPUs. This was shown after analysis of closed source Oclhashcat in [25].

As shown in [29], there exist other attacks not connected with GPU. This paper shows that an attacker can save 50 % of PBKDF2 cpu operations if the PBKDF2 is not implemented according to suggested performance improvements described in RFC 2104 [19] and NIST FIPS PUB 198 [15]. In this case it is possible to precompute first message block of underlying keyed hash function (used as PRF) and replace it with resulting constant in subsequent operations. See lines 11–13 in algorithm 1. Note that HMAC function is not described in this thesis, see [29]subsection 4.1.

In [29]subsection 4.2 there is shown that an attacker can omit considerable amount of XOR operations while using SHA1 as a pseudorandom function within PBKDF2 because this operation is sometimes performed on two blocks containing only zeroes. Additionally, more XOR operations can be omitted because of padding characters which are constant and some XOR operations in this case just zero out them selves. Finally, in subsection 4.3, it is shown that it is possible to precompute the word expansion part of the second message block of a keyed hash function. The block is password independent and can be thus precomputed. However, this saves only negligible amount of time compared to previous attacks described in this paragraph.

In 2009 Colin Percival suggested that to defend against usage of parallel computing, PBKDFs should fulfill requirements of memory-hard functions [23]. As a reaction to previously mentioned problems of PBKDF2, the Password hashing competition was held from 2013 to 2015 to select a new function for password hashing. The winner is Argon2 described in section 3.2 and it is indeed a memory-hard function. Of course that does not mean that memory-hard functions are not prone to attacks.

In a paper from 2016, authors show that there still exists an attack which can decrease computational complexity of Argon2I-B function. It was shown that at that time it was needed to configure at least ten passes of Argon2I-B to mitigate this attack. At the time of releasing the paper, the IRTF proposal suggested only six passes for "paranoid" situations. Fortunately authors of Argon2 reacted to those attacks and improved the function, sothat the attack is not effective against Argon2I anymore except when runnin with only signle pass. Details and rationale can be found in [5]subsection 5.2.

3 State of the art PBKDFs

3.1 PBKDF2

PBKDF2 is a password-based key derivation function defined in RFC 8018 [16]. 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 [27]. 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 [20]section 4.1 for more details.

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

Algorithm 1: PBKDF2 function algorithm

3.2 Argon2

As mentioned in very brief history of attacks on password hashes in section 2.4, the Argon2 function is the winner of Password Hashing Competition. Argon2 is a hash function belonging to the set of memory-hard functions as defined in [23]. As defined in section 2.1 PBKDFs are subset of hashing functions and Argon2 can be definitely used as PBKDF. Current version of the function is 1.3 and the latest IETF draft is [6].

The function quickly fills up given amount of memory and performs a sequence of computations over values stored in this memory. The Argon2 comes in three versions which differ in the way in which data in the memory matrix (described further below) is processed. Argon2D, Argon2I and Argon2ID. See subsection 3.2.3 for detailed description.

Argon2 is used as the default PBKDF in LUKS version 2. Note that according to [10] the default PBKDF can be configured during compilation.

The function is optimized for X86 architecture using improvements in handling of cache and memory access in recent Intel and AMD processors. The function can be implemented on specialised hardware, but the previously mentioned fact makes this implementation possibly very slow and expensive and even specialised ASICs shouldn't acquire significant benefit even if they employ large areas of memory. However, no implementations of any Argon2 mode for specialised hardware are known so far.

3.2.1 Operation

The function expects following primary input parameters. Primary means that parameters must always be supplied by user.

P the message of any length from 0 to $2^{32}-1$ bytes. In case of PBKDF this is the passphrase.

S nonce with length from 8 to $2^{32}-1$ bytes. In case of PBKDF this is the *cryptographic salt*.

The function also accepts secondary inputs, which do not need to be supplied.

p degree of parallelism as an integer with a value from 1 to $2^{24}-1$. The value determines the number of parallel computational chains to be run. Chains are not independent, synchronisation occurs.

τ tag length in bytes in range from 4 to $2^{32}-1$. This determines an output of the function, in case of PBKDF key length.

m memory size as an integer number in range from $8p$ to $2^{32}-1$. The integer determines amount of kilobytes of memory which should be used for computation of the function.

t number of iterations as an integer in range from 1 to $2^{32}-1$. This parameter is used to tune the length of the function run by specifying number of iterations.

v one byte version umber, currently hardcoded to 0x13.

K signifies a secret value (key) with length from 0 to $2^{32}-1$ bytes. By default no key is assumed.

X associated data with length from 0 to $2^{32}-1$ bytes.

y type (mode) of Argon2 to be used. 0 for Argon2D, 1 for Argon2I, 2 for Argon2ID.

Argon2 makes use of the permutation function P which is based on Blake2b hashfunction [3]. The function actually copies blake2b design but additionally it uses 64 bit multiplications. This particular modification makes the function more complicated to implement and optimize for ASICs, while the running speed on X86 processors should be degraded only negligibly. See [6]section 3.6 for detailed explanation.

Another internal function of Argon2 is compression function G which internally uses previously mentioned function P. G accepts two 1024 bytes long inputs and produces one 1024 bytes long output. Let X and Y be the inputs. Firstly, the function XORs them:

$R = X \oplus Y$

R is treated as a 8×8 matrix of 16 byte registers. The function P is first applied to every row of the matrix and then to every column.

The result is denoted as Z. Finally, the result is computed as $Z \oplus R$. For more detailed description see [6]section 3.5.

Argon2 makes use of two more hash functions. The H^X function where X denotes the output length in bytes and the whole function is again based on Blake2b. Finally, the variable length hash function H'^X based on H^X defined in [6]section 3.3 is also used.

The argon2 operation can be described as follows. The emphasized variables are Argon2 input parameters, primary and secondary parameters are not distinguished. The numbers in brackets denote the line numbers in algorithm 2.

1. initialisation of the block H_0
2. allocation of the memory according to m and p parameters. The real size of allocated memory is denoted with m' in 1024 byte blocks. Note that the memory is treated as a matrix $B[I][J]$ with p rows and $q = m'/p$ columns.
3. compute $B[i][0]$ for $0 <= i < p$
4. compute $B[i][1[$ for $0 <= i < p$
5. compute $B[i][j]$ for $0 <= i < p$ and $2 <= j < q$. This step is different for every Argon2 mode.
6. if $t>$ 1 then repeat step 5 with slight change for every iteration
7. the final block C is computed
8. the output tag is computed

3.2.2 algorithm

This subsection describes Argon2 through pseudocode. In addition to input parameters mentioned earlier, following notations will be used:

$||$ concatenation of two strings

floor(x) function returns the largest integer which is not bigger than x ceil(x) function returns the smallest integer which is not smaller than x

LE32(x) converts 32 bit long integer x to byte string in little endian

length(s) returns length of string s in bytes as a 32 bytes long integer

allocate(x) allocates x bytes of memory

3.2.3 Differences in Argon2 versions

Argon2 comes in three versions:

Argon2D data-dependent variant of Argon2. This version uses previously computed data while performing computations and memory access. It is recommended to be used for cryptocurrencies and other proof-of-work applications as well for hashing on backend servers. In general, this version is suited for an environment where no side-channel timing attacks are expected. It provides better protection against brute-force attacks usin specialised hardware and tradeoff attacks.

Argon2I data-independent version. This version does not rely on previously computed data while performing calculations in memory. The version is recommended for key derivation and password hashing, where side-channel timing attacks are more probable because an adversary can have physical access to the machine. The mode is slower because it performs multiple passes over memory to defend against tradeoff attacks.

Argon2ID combination of previous versions. In this mode the function behaves as Argon2I during computation of the first half of the first iteration over the memory. During remaining operation the Argon2D mode is used. The mode combines benefits of mitigation of side-channel timing attacks and brute force attacks.

The difference lies in line number xx in the algorithm 2. In particular indexes l and z are computed differently within different Argon2 versions. The allocated memory is represented as a matrix $B[I][J]$ with

```
input  : P, S, p, $\tau$, m, t, v, K, X, y
output : TAG
$H_0 \leftarrow$ H64(LE32($p$) || LE32($\tau$) || LE32($M$) || LE32($t$) || LE32($v$) ||
  LE32($y$) || LE32(length($P$)) || $P$ || LE32(length($S$)) || $S$ ||
  LE32(length($K$)) || $K$ || LE32(length($X$)) || $X$)
$M' \leftarrow 4 \times p \times$ floor($m/4p$)
$B \leftarrow$ allocate($m'$)
for $i \leftarrow 0$ to $p-1$ do
    $B[i][0] \leftarrow H'^{128}$($H_0$ || LE32($0$) || LE32($i$))
end
for $i \leftarrow 0$ to $-1p$ do
    $B[i][1] \leftarrow H'^{128}$($H_0$ || LE32($1$) || LE32($i$))
end
for $i \leftarrow 0$ to $p-1$ do
    for $j \leftarrow 2$ to $q$ do
        $B[i][j] \leftarrow$ G($B[i][j-1], B[l][z]$)
        /* indexes l and z are computed differently for
           every Argon2 version, see subsection 3.2.3    */
    end
end
if $t > 1$ then
    for $i \leftarrow 0$ to $p-1$ do
        $B[i][0] \leftarrow$ G($B[i][q-1], B[l][z]$)
        for $j \leftarrow 1$ to $q$ do
            $B[i][j] \leftarrow$ G($B[i][j-1], B[l][z]$)
        end
    end
end
$C \leftarrow B[0][q-1] \oplus B[1][q-1] \oplus \ldots \oplus B[p-1][q-1]$
return $H'^{\tau}(C)$
```

Algorithm 2: Argon2 function algorithm

p rows and $q = m'/p$ columns. Before computing indexes l and z, values J_1 and J_2 have to be calculated. This is the difference among Argon2 versions. AFter calculating of these values, further processing needs to be done but this stays the same for all versions.

Rows of memory are called lanes and there are p lanes according to the parameter p, which signifies degree of parallelism. Moreover, the memory is divided into 4 vertical slices, where number of slices is denoted as S. Every intersection of a lane and a slice is called a segment. Segments belonging to the same slice are computed in parallel and they must not reference blocks of each other.

Argon2d computes J_1 and J_2 based on data stored in memory. Therefore it is data-dependent.

$J_1 \leftarrow int32(extract(B[i][j-1], 1))$

$J_2 \leftarrow int32(extract(B[i][j-1], 2))$

Where the function int32(s) converts the 32 bit string s into non-negative integer represented in little endian. The function extract(s, i) extracts ith set 32 bits long from bitstring s while s is indexed from 0.

Argon2i computes indexes by running two rounds of the compression function G in counter mode in the following way

$x \leftarrow G(ZERO, LE64(r)||LE64(l)||LE64(s)||LE64(m')||LE64(t)||LE64(y)||LE64(i)||ZE$

The result x is 64 bits long and therefore it can be viewed as two 32 bit strings

$x_1||x_2 \leftarrow x$

$j_1 \leftarrow int32(x_1)$

$j_2 \leftarrow int32(x_2)$

Inputs for the function G are described below. Note that no data from memory blocks are used. therefore Argon2i is data-independent.

r the pass number

l the lane number

s the slice number

m' total number of memory blocks

t total number of passes

y Argon2 mode

i counter starting from 1 in every segment

ZERO string of zeroes, in the first case 1024 bytes long, in the second case 968 bytes long

Argon2id behaves as Argon2i if the pass number is 0 and at the same time the slice number is 0 or 1. Othervise it behaves like Argon2D.

Further computations common for all Argon2 modes are described in [6]3.4.2.

3.3 Scrypt

Scrypt is a PBKDF introduced by Colin Percival in [23]. At the time of releasing the paper (2009) problems of PBKDF2 and its parallel computation were already well-known. Percival came with two new concepts. He defined a memory-hard algorithm as an algorithm which performs asymptotically almost the same number of operations compared to number of accessed memory locations. The second concept is called sequential memory-hard function. This definition describes a function which can be computed by a memory-hard sequential algorithm and at the same time even the fastest parallel algorithm cannot asymptotically reach a significantly lower cost.

Both concepts came as a reaction to increasing computing power of parallel hardware.

3.4 Other PBKDFs

4 The price of an attack

This chapter focuses on creating a price model for potential attacker trying to gain unauthorized access to a disk volume encrypted with LUKS2 with Argon2 used as PBKDF. First an attacker and available hardware and software options are briefly described. Then the actual price model is introduced and applied to real world examples.

4.1 Attacker

For the purpose of this thesis an attacker is defined as an entity which gained unauthorized access to a LUKS2 volume header with Argon2 used as PBKDF. The header can be part of actual encrypted volume or it can be a detached header stored for example for backup purposes. The volume is not damaged. The attacker has moderate knowledge of information security and computers in general but does not want to invest the time in searching for vulnerabilities in Argon2 or Cryptsetup. The attacker decides to use brute-force or dictionary attack.

The attacker currently does not have any hardware with sufficient computing power and amount of RAM to be used for cracking the passphrase. However, suppose that the attacker has sufficient financial resources to purchase needed hardware or rent cloud computing resources.

4.1.1 Hardware

While considering cracking a password or a hash, there are several types of hardware to consider. An attacker can use powerful CPUs which can be cheaper solution compared to other possibilities but they are also usually slower because of their generic nature. The fact which is crucial while considering Argon2 is that they can have access to relatively large volume of RAM without degrading their computing performance. It is also less time-consuming to eventually optimize the hash algorithm for CPU than for other types of hardware.

The next option often used in this process is to use GPUs. Their architecture is suitable for computing many parallel identical tasks as is the case for hash cracking. GPUs showed to be effective against

PBKDF2 [20]. However, the price of a GPU might be higher than the price of a CPU. To use full computing power of GPU it is important that the data being processed is copied into GPU global memory which limited. This fact could reduce their effectivity in this particular case considering possible high memory demands of Argon2.

Last two options are FPGAs and ASICs. They represent two groups of hardware which can be optimized for highly specific tasks. FPGAs are in general less powerful but they can be easily reprogrammed after production. ASICs can incorporate parts which can be later reprogrammed but definitely not to the same extend as FPGAs. ASICs are also more expensive than FPGAs when produced in small volumes. Moreover, to the best of the author's knowledge, there does not exist any publicly available implementation of Argon2 for FPGAs or ASICs and it would take for an attacker nonnegligible amount of time to create one.

4.1.2 Software

There does not exist any publicly available software offering feature to crack passphrases of LUKS2 headers. There exists a simple proof of concept program distributed with Cryptsetup called Dict_search. It mounts dictionary attack against specified device supporting Truecrypt and LUKS1. After slight modifications it can be used also against LUKS2 volumes. It uses API of libcryptsetup. Therefore it obviously introduces slight overhead by creating and deleting structures pertaining to Cryptsetup.

Then there is a possibility of extracting the master key from the captured LUKS2 header and trying to find the right password by comparing it to it after being hashed. This removes some slight overload mentioned above. Support for cracking of Argon2 hashes appeared in well-known open source password auditing software John the ripper in July 2016. This allows to use some infrastructure offered by the framework ut the hash function is not optimized in any way.

Ondrej Mosnáček created an experimental program for benchmarking speed of Argon2 while running on CPUs and GPUs [21]. The project can use both CUDA and OpenCL technologies to run on multiple GPUs or CPUs. The program currently does not perform password cracking, passwords are generated randomly and they are not compared to any hash. But slight modifications to the project could turn it into a password cracker.

4.2 Price model

Based on previous assumptions of an attacker's options I try to create a price model which will estimate costs connected with finding the right passphrase to unlock the LUKS2 encrypted volume. These costs include purchase of devices and electricity costs. The model is based on...

4.3 Real world cost estimation

5 Analysis of LUKS2

This chapter deals with the actual practical research performed as a part of the thesis. The first part deals with collecting number of Argon2 parameters under various circumstances and their analysis. The second part introduces experimental attack against LUKS2 headers created with chosen parameters based on previous part. These results are then applied to the price model introduced in chapter 4.

5.1 LUKS2 and Argon2 parameters

As mentioned in section 2.3.3, there are two ways of defining three parameters needed for Argon2 operation. They can be either specified manually or Cryptsetup can estimate some of them through benchmarking process. Here I will deal only with the second option.

The most important parameters for the benchmark are unlocking time (default of 2000 miliseconds) and Aargon2 memory cost (1048576 kB). The degree of parallelism can be at most 4 and will be lowered if not enough CPUs are available. The main aim of the benchmarking function is to find such Argon2 parameters that the time of hash calculation is as closest to the given unlocking time as possible. This is reached by increasing or decreasing memory cost and number of Argon2 iterations. At the same time the memory cost parameter cannot exceed the user supplyed volume of memory and it cannot exceed half of available physical memory. The source code of the function can be found in the file lib/crypto_backend/pbkdf_check.c.

5.1.1 Benchmarking tool

As a part of my thesis I performed collection of Argon2 parameters estimated by the function mentioned above. I created a small tool which can perform series of benchmarks with given parameters and output results in human-readable or machine-readable format. The source code of the tool is part of the thesis (bude na Githubu).

The tool accepts the following parameters. Run the tool without command line options to see exact format of parameters.

- memory cost

- degree of parallelism
- required unlocking time
- number of repetitions of the benchmark

The tool can output results in a text format or in a CSV format including headers for better machine processing. The tool does not perform actual benchmarking, it use API of Libcryptsetup library. Therefore the library is required for it to work. The tool does not create any cryptsetup volumes and does not require root privileges.

5.1.2 Collecting of real world parameters

I used the previously mentioned tool to collect real data on real physical hardware. I tried to simulate various environments in which Cryptsetup can be used to create and unlock encrypted volumes. I decided to use real hardware and not virtual machines because of possible inaccuracies during measurement caused by virtualization layer.

The first hardware configuration consists of Lenovo Thinkpad P50 with ... to be completed

The second configuration comprises Raspberry Pi 3 with ... to be completed. As shown this configuration has relatively low computing power and limited amount of memory. This combination was chosen because it can show potential problems of the benchmarking function.

On both devices I performed the following steps:

1. install required libraries and headers
2. download and unpack the source code of Cryptsetup version 2.6.0
3. configure and compile Cryptsetup without support for udev and blkid to minimize required dependencies, see below for remarks concerning configuration options
4. compile the benchmarking tool linking it to the Libcryptsetup compiled in the previous step
5. perform hardware limitations (amount of available CPUs, amount of available memory)

6. run series of benchmarks over unlocking times of 1000, 2000, 3000, 4000, 5000, 10000 and 20000 miliseconds with 1, 2, 3, 4, 6 and 8 parallel threads with every benchmark repeated 100 times

5.2 Attacking LUKS2

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] J.-p. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. (Jan. 29, 2013). Blake2: Simpler, smaller, fast as md5, [Online]. Available: https://blake2.net/blake2.pdf (visited on 12/26/2018).

[4] A. Belenko. (2007). Faster password recovery with modern gpu, [Online]. Available: https://www.elcomsoft.com/presentations/faster_password_recovery_with_modern_GPUs.pdf (visited on 12/21/2018).

[5] A. Biryukov, D. Dinu, and D. Khovratovich. (Mar. 24, 2017). Argon2: The memory-hard function for password hashing and other applications, [Online]. Available: https://www.cryptolux.org/images/0/0d/Argon2.pdf (visited on 09/29/2018).

[6] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "The memory-hard argon2 password hash and proof-of-work function", IETF Secretariat, Internet-Draft draft-irtf-cfrg-argon2-04, Nov. 2018. [Online]. Available: http://www.ietf.org/internet-drafts/draft-irtf-cfrg-argon2-04.txt.

[7] 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).

[8] 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).

[9] F. Corbató, M. Dagget, R. Daley, P. Denning, D. A. Grier, R. Mills, R. Roach, and A. Scherr. (2011). Compatible time-sharing system (1961-1973), Fiftieth aniversary commemorative overview. D. Walden and T. Van Vleck, Eds., [Online]. Available: http://www.multicians.org/thvv/compatible-time-sharing-system.pdf (visited on 12/21/2018).

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

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] P. J. Bond, U. Secretary, A. L. Bement, and W. Mehuron Director, "Fips pub 198", May 2002.

[16] 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).

[17] 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.

[18] D. Kennedy. (May 29, 2015). Of history & hashes: A brief history of password storage, transmission, & cracking, [Online]. Available: https://www.trustedsec.com/2015/05/passwordstorage/ (visited on 12/20/2018).

[19] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication", RFC Editor, RFC 2104, Feb. 1997, p. 11. [Online]. Available: https://tools.ietf.org/html/rfc2104 (visited on 12/21/2018).

[20] 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).

[21] O. Mosnáček. (2019). Argon2-gpu, [Online]. Available: https://gitlab.com/omos/argon2-gpu (visited on 03/16/2019).

[22] 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.

[23] C. PERCIVAL, "Stronger key derivation via sequential memory-hard functions", Jan. 2009.

[24] A. Peslyak and S. Marechal. (2012). Password security: Past, present, future, [Online]. Available: https://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/Passwords12-The-Future-Of-Hashing.pdf (visited on 10/21/2018).

[25] A. Ruddick and J. Yan, "Acceleration attacks on pbkdf2: Or, what is inside the black-box of oclhashcat?", in *Proceedings of the 10th USENIX Conference on Offensive Technologies*, ser. WOOT'16, Austin, TX: USENIX Association, 2016, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=3027019.3027020 (visited on 11/28/2018).

[26] 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).

[27] 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.

[28] T. Van Vleck. (Feb. 15, 1995). Multics security, [Online]. Available: http://www.multicians.org/security.html (visited on 12/21/2018).

[29] A. Visconti, S. Bossi, H. Ragab, and A. Calò, "On the weaknesses of pbkdf2", Dec. 2015. DOI: 10.1007/978-3-319-26823-1_9.

[30] 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 (visited on 10/28/2018).

A LUKS attack results