Security Analysis of Cryptsetup/LUKS

Ubuntu Privacy Remix Team <info@privacy-cd.org>

August 12, 2012

Contents

1. Introduction

Since version 12.04 Ubuntu Privacy Remix besides TrueCrypt supports Cryptsetup with LUKS[1] as well for volume encryption. For this reason we analyzed the security of Cryptsetup/LUKS just as that of TrueCrypt (see [UPR2011]). This analysis serves the same purpose as that of TrueCrypt, that is to help people to form their own sound opinion on the security of the encryption. As Ubuntu Privacy Remix 12.04 offers the choice between these two encryption programs we every so often compare these programs and their peculiarities in this analysis. Again we invite everyone reading this to criticize our method and findings or make suggestions for further analysis.

Cryptsetup with LUKS is exclusively developed for Linux. But the encryption software FreeOTFE developed by Sarah Dean for Windows and PDAs supports the LUKS format also. So volumes encrypted with Cryptsetup/LUKS on Linux can be opened with FreeOTFE on Windows or on a PDA. However, we will not analyze FreeOTFE in the following. Cryptsetup is distributed under the *"Gnu General Public License"* version 2 (GPL, v2). Therefore – in contrast with TrueCrypt – there are no

1 LUKS is the acronym for "Linux Unified Key Setup".

legal problems with the openness of the sources of Cryptsetup.

Compared to TrueCrypt Cryptsetup isn't such a monolithic program. It rather depends on libraries from other sources and delegates the actual encryption to a greater degree to the Linux kernel as TrueCrypt does on Linux. Among the libraries `libgcrypt` from the GnuPG project is salient. Cryptsetup uses it for calculating cryptographic hash values. Nevertheless, we restricted our analysis to the sources of Cryptsetup itself. Beyond that we merely verified the hash values calculated by `libgcrypt` with a test program we wrote for that purpose.

2. Analyzed Version of Cryptsetup

We have chosen the version 1.4.1 of Cryptsetup used by Ubuntu 12.04 LTS for our analysis although there exist versions 1.4.3 and 1.5.0 in the meantime. The source code of this may be downloaded in Ubuntu 12.04 LTS with the command

```
apt-get source cryptsetup
```

which gives the three files

- `cryptsetup_1.4.1.orig.tar.bz2`
- `cryptsetup_1.4.1-2ubuntu4.debian.tar.gz`
- `cryptsetup_1.4.1-2ubuntu4.dsc`

The first file is identical with the source code archive `cryptsetup-1.4.1.tar.bz2` as it could be downloaded from http://code.google.com/p/cryptsetup/. The second file only contains code for building the Debian package as well as code necessary for the integration of Cryptsetup into the Ubuntu system. The third file finally only contains a description for the Debian package.

Data of Cryptsetup 1.4.1

Website:	http://code.google.com/p/cryptsetup/
Analyzed version:	`Cryptsetup 1.4.1`
Analyzed source code archive:	`cryptsetup-1.4.1.tar.bz2`
MD5 fingerprint:	`9253b3f29abf5c6f333eb74128b0df1c`
SHA1 fingerprint:	`32608be5b146a7bd3999129b086bad8b66c085b9`

3. Compiling Cryptsetup from Sources

In order to build Debian packages on Ubuntu 12.04 LTS from the source code which has been downloaded as described in section 2 you have to install the tools for building Debian packages with the command

```
sudo apt-get install dpkg-dev
```

Then you have to install the development versions of the libraries on which Cryptsetup depends with

```
sudo apt-get install libgcrypt11-dev libdevmapper-dev libpopt-dev \
uuid-dev libselinux1-dev libsepol1-dev
```

Finally some additional development tools are needed and can be installed with

```
sudo apt-get install libtool autoconf automake autopoint gettext \
debhelper xsltproc docbook-xsl po-debconf
```

Thereafter you unpack the source code archive `cryptsetup_1.4.1.orig.tar.gz` and the code from the archive `cryptsetup_1.4.1-2ubuntu4.debian.tar.gz` for building the Debian packages with the command

```
dpkg-source -x cryptsetup_1.4.1-2ubuntu4.dsc
```

After changing to the newly created directory with

```
cd cryptsetup-1.4.1
```

the Debian packages are built with

```
dpkg-buildpackage -b -uc
```

The option `-b` has the effect that only binary packages are built and `-uc` tells the command not to sign the file `cryptsetup_1.4.1-2ubuntu4_i386.changes` with GnuPG[2]. It is created together with the packages and contains their checksums. On a 32-bit-x86 system the packages

- `cryptsetup_1.4.1-2ubuntu4_i386.deb`
- `cryptsetup-bin_1.4.1-2ubuntu4_i386.deb`
- `cryptsetup-udeb_1.4.1-2ubuntu4_i386.udeb`
- `libcryptsetup4_1.4.1-2ubuntu4_i386.deb`
- `libcryptsetup4-udeb_1.4.1-2ubuntu4_i386.udeb`
- `libcryptsetup-dev_1.4.1-2ubuntu4_i386.deb`

are to be found one level up in the directory hierarchy. On other processor architectures corresponding packages are created where `i386` in the names is replaced by a notation of the respective architecture. Files with ending `.udeb` are packages in the so-called micro-Debian format which are needed in the boot process of the computer if it already has to decrypt some file system.

4. Methodology of Analysis

The development of LUKS by Clemens Fruhwirth was theoretically substantiated by two publications (see [Fru2004] and [Fru2005]) which we studied first together with the specification of the format of LUKS [Fru2011]. We then carefully read the source code contained in the archive `cryptsetup-1.4.1.tar.bz2`. An encryption or decryption isn't performed in this code. Where it is needed Cryptsetup charges the Linux kernel with this task via the device mapper interface. The solution of this task then proceeds exactly the same as with TrueCrypt which uses the same interface to assign the job of encryption and decryption of the volume to the Linux kernel[3]. Cryptsetup goes even further than TrueCrypt by assigning the job of decryption of the master key also to the Linux kernel. As we already analyzed this encryption and decryption in the Linux kernel in the context of

2 If you want to sign this file you have to replace the option `-uc` with `-k` followed by the key ID without any space in between. If the GnuPG signing key isn't given by this option the command would try to use the key of the last-mentioned maintainer in `debian/changelog` which would fail if you haven't created a GnuPG key with the name of that maintainer.

3 There only is a distinction when the CBC-ESSIV mode of LUKS is used which is unknown to TrueCrypt.

our analysis of TrueCrypt (see [UPR2011]) there was no need for us to re-examine this.

For the derivation of the key encrypting the master key from the password as well as for other security features Cryptsetup with LUKS needs cryptographic hash algorithms. For the calculation of hash values it solely uses the library `libgcrypt11` from the GnuPG project. We abstained from analyzing the source code of this library as well. Instead we wrote the program `hashtest` which calculates all sorts of hash values with this library. For all hash algorithms used by Cryptsetup we then calculated a vast number of hash values for distinct inputs and compared them to reference values which have been calculated independently. For the calculation of reference values among others the programs `md5sum`, `sha1sum`, `sha256sum` and `sha512sum` as well as `openssl` have been used. The compared values always matched. Our `hashtest` program also tests the capability of `libgcrypt11` to calculate HMAC-values. These values likewise always matched the values calculated by our code.

Moreover we wrote the program `luksanalyzer` which decrypts and analyzes the LUKS header as it has been written to disk with Cryptsetup/LUKS. It also uses the library `libgcrypt11` for hash algorithms and for decryption with cipher algorithms. All other security features of Cryptsetup with LUKS have been reimplemented independently by us in this program. We have analyzed a lot of headers of test volumes created with Cryptsetup/LUKS. All relevant combinations of ciphers, hash algorithms and encryption modes have been checked by this means.

5. The Programs luksanalyzer and hashtest

We wrote the programs `luksanalyzer` and `hashtest` for the purpose of our analysis. The source code of both is maid available together in the archive `luksanalyzer.tar.gz` under the *"GNU General Public* License" version 3 or optionally any later version (GPL, v3+).

The Program luksanalyzer

The program `luksanalyzer` must only be applied to LUKS-encrypted volumes created for test purposes. Otherwise the encryption will be compromised by the output of the master key. It analyzes the LUKS partition header and key slots for an encrypted partition and has the following options to be given on the command line:

- **-h, --help:** print a help message and exit.
- **-d, --dump *file*:** dump the decrypted anti-forensic key to the given file.
- **-o, --out *file*:** write the output to the given file, without this option output is written to the terminal.
- **-p, --passphrase p*assword*:** use the given password for analyzing, this option is mandatory.
- **-s, --slot *key-number*:** analyze only the given key slot (a number between 0 and 7), without this option all slots will be tried.
- **-v, --volume *device-file*:** analyze the volume on the given device, this option is mandatory.

The output of let's say the command

```
luksanalyzer –p test –s 0 –v /dev/sdb1
```

then typically looks like the following:

```
Analysis of LUKS Volume on /dev/sdb1
```

```
Header:
================================================================================
    0: 4c 55 4b 53 ba be                                        | magic byte sequence for
LUKS partition header
    6: 00 01                                                    | version: 1
    8: 61 65 73 00 00 00 00 00 00 00 00 00 00 00 00 00          | ...
   24: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00          | cipher name: aes
   40: 78 74 73 2d 70 6c 61 69 6e 36 34 00 00 00 00 00          | ...
   56: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00          | cipher mode: xts-
plain64
   72: 73 68 61 35 31 32 00 00 00 00 00 00 00 00 00 00          | ...
   88: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00          | hash specification:
sha512
  104: 00 00 0f c8                                              | payload offset (in
sectors): 4040
  108: 00 00 00 40                                              | key bytes: 64
  112: 75 9b bf 34 7b 73 8b 38 02 82 c9 8a f3 82 89 dd          | ...
  128: 28 01 c0 70                                              | master key checksum
  132: 4a c5 91 43 ed da 70 2c 4e b7 c6 7b 02 6c c4 d2          | ...
  148: 76 54 b7 ca d0 0f 3e 61 cc 2e 6f 22 12 89 2b 66          | PBKDF2 salt for master
key checksum
  164: 00 00 b6 ec                                              | PBKDF2 iterations for
master key checksum: 46828
  168: 66 38 62 65 31 61 39 39 2d 38 30 30 65 2d 34 61          | ...
  184: 66 34 2d 62 33 38 66 2d 35 35 37 63 66 62 30 64          | ...
  200: 66 37 63 33 00 00 00 00                                  | UUID of the partition:
f8be1a99-800e-4af4-b38f-557cfb0df7c3
================================================================================

KeySlot 0:
--------------------------------------------------------------------------------
  208: 00 ac 71 f3                                              | state of keyslot:
enabled
  212: 00 02 77 c1                                              | PBKDF2 iterations for
passphrase of keyslot: 161729
  216: 2c c7 56 b0 df ef 08 80 c9 f4 c3 5d ee fd 10 dd          | ...
  232: 54 5b 40 ab 1e a0 ff 33 13 1a ea 90 42 8e 1d a9          | PBKDF2 salt for keyslot
  248: 00 00 00 08                                              | start sector of key
material for keyslot: 8
  252: 00 00 0f a0                                              | number of anti-forensic
stripes for keyslot: 4000
--------------------------------------------------------------------------------
mk: 0: 7a 5b 94 26 db 26 27 30 3c fc d0 34 b8 a4 d4 4b          | ...
mk:16: 78 8f c7 fe 6d 42 1e f4 09 b2 08 c1 06 c6 30 e1          | ...
mk:32: 5e fa 23 f6 6f 4b 38 3f dd b7 ac 9e f6 f7 c9 a1          | ...
mk:48: cf 9a e7 2f c3 88 1e 10 c4 58 4c 03 fb 93 a4 81          | master key (verified
with checksum)
--------------------------------------------------------------------------------
```

The Program hashtest

The program `hashtest` calculates hash and HMAC values either for a password given on the command line or for the contents of a file. For HMAC values you also have to give the key to the HMAC algorithm on the command line. For them two calculations are compared, one with the internal HMAC capability of `libgcrypt11` and the other with an external HMAC coded in `hashtest` using only hash values calculated with `libgcrypt11`. In order to verify the hash values they have to be double-checked by an independent calculation with other programs like `md5sum`, `sha1sum`, `sha256sum` and `sha512sum` or `openssl dgst`. The options of `hashtest` are:

- **-h, --help:** print a help message and exit.
- **-d, --digest *Algorithm*:** test the given digest algorithm. Strings identifying the digest algorithms are case-insensitive.
- **-f, --file *file*:** calculate hash or HMAC of the contents of the given file.
- **-k, --key *key*:** calculate a HMAC with the given key instead of a simple hash. This is done internally using the HMAC capability of `libgcrypt11` and externally using only the hashing capability of `libgcrypt11`. Both values are printed and have to be the same. If not an error message is printed.
- **-p, --passphrase *password*:** calculate hash or HMAC of the given passphrase.
- **-v, --version:** print the version number of libgcrypt11 and the digest algorithms supported by it with their internal identification number.

HMAC values may also be calculated independently with the command

`openssl dgst –`*algorithm* `–hmac` *key file*

or

`echo –n` *password* | `openssl dgst –`*algorithm* `–hmac` *key*

respectively for verification. Here *algorithm* has to be replaced with one of the algorithms `sha1`, `sha256`, `sha512` or `ripemd160` just as by `hashtest` and *key*, *file* and *password* also have to be the same as given to the program `hashtest`.

6. Findings of Analysis

The License of Cryptsetup

Cryptsetup is distributed under the *"Gnu General Public License"* version 2 (GPL, v2). In contrast with TrueCrypt, we therefore have no problem with the license. The maintainers of Cryptsetup even discuss and possibly accept change requests. This is quite in contrast with the habit of the developers of TrueCrypt who ignore all proposals of changes.

Website and Documentation of Cryptsetup/LUKS

The website http://code.google.com/p/cryptsetup/ offers all versions of Cryptsetup for download which ever have been published. A Wiki on that website contains interesting articles and discussions on Cryptsetup and LUKS. Bug reports, issues and proposals for enhancements may be followed up in a bug-tracker there. In order to add new issues to that bug-tracker you need a previously created Google account.

Finally also the source code is accessible in the version control system `git` on that website. This may be downloaded with the command

```
git clone https://code.google.com/p/cryptsetup/
```

with up to date changes and the history of the code. For this you only need to install the version control system `git` on your computer. Of course, write access to this version control system is limited to the developers of the project. Other developers, however, are facilitated by this if they want to follow the current development of Cryptsetup or even to submit concrete code changes themselves.

Cipher Algorithms in Cryptsetup/LUKS

Cryptsetup can set up a LUKS-encrypted volume with any cipher supported by the Linux kernel. However, [Fru2011] specifies only five cipher algorithms to be chosen for LUKS-encrypted volumes:

- `aes`,
- `twofish`,
- `serpent`,
- `cast5` (also known as CAST-128) and
- `cast6` (also know as CAST-256).

They have to be given to Cryptsetup in the parameter `-c` or `--cipher` exactly as specified here in lower case letters. Rijndael, Twofish and Serpent were finalists and CAST6 a candidate in the world wide competition by which Rijndael was selected by NIST (National Institute of Standards and Technology of the USA) for Advanced Encryption Standard (AES) in October 2000 (see [NIST2001]). All ciphers have been designed by teams of well-known cryptographers and scrutinized publicly and intensively in the process of that competition. No security holes have been found in these ciphers.

CAST5 is a predecessor of CAST6 and operates with a block length of only 64 Bit compared with 128 Bit block length of the other algorithms. Although CAST5 is the default cipher in GnuPG and in some versions of PGP we do not recommend this cipher for volume encryption because of the low block length.

AES distinguishes itself among the other specified ciphers by being an international standard. On this basis we recommend the choice of AES. AES just as Twofish, Serpent and CAST6 permits key sizes of 128, 192 and 256 bits. Generally the security increases with the size of the key. Therefore a key size of 256 recommends itself[4].

In contrast with TrueCrypt, Cryptsetup does not provide combinations of several ciphers. This is an advantage of TrueCrypt as a multiple encryption with several ciphers with independent keys is as strong as the strongest cipher in the cascade. If one of those ciphers gets broken the whole encryption is still safe as it is protected by another unbroken cipher. Application of multiple ciphers, on the other hand, slows down the process of encryption and decryption. There, however, is already a discussion on http://code.google.com/p/cryptsetup/wiki/LUKSSpec20BrainStorming whether a new version 2.0 of the LUKS specification should allow cipher cascades.

Modes of Encryption in Cryptsetup/LUKS

The cipher algorithms are operating on blocks of 16 bytes (128 bits) except for CAST5 which is operating on blocks of 8 bytes (64 bits). So as not to encrypt blocks with identical contents to identical blocks within the encrypted volume the encryption is modified slightly from one block to the other. This is called the encryption mode.

As for ciphers the supported modes of encryption in Cryptsetup with or without LUKS only depend on the Linux kernel. The modes specified in [Fru2011] are

- `ecb`,
- `cbc-plain`,
- `cbc-essiv:`*hash* and

4 If the XTS mode of encryption is chosen, you have to specify twice that key size when creating an encrypted volume with Cryptsetup. This is so since XTS needs two different keys and Cryptsetup expects in its option `--key-size` or `-s` respectively the specification of the total size of both keys joined together.

- `xts-plain64`.

Just as the cipher, the encryption mode has to be given to Cryptsetup in the parameter `-c` or `--cipher` exactly as specified here in lower case letters appended to the cipher and separated from it by a minus sign. In `cbc-essiv:`*hash hash* has to be replaced by a valid hash algorithm (`sha1`, `sha256`, `sha512` or `ripemd160`).

In ECB (electronic code-book) mode every block is encrypted separately and independently of the other blocks and of its position in the volume. So this mode does not solve the problem of modifying the encryption. Therefore, it is absolutely not suitable for a serious volume encryption.

In the plain CBC (cipher-block chaining) mode every plain-text block is combined with the previous cipher-text block by an exclusive-or (XOR) operation before it is encrypted. This chaining is cut at every sector boundary and re-initialized with the sector number which converted to 32-bit and to little-endian replaces the previous cipher-text block for the first block in every sector. This mode has considerable weaknesses as exposed by Clemens Fruhwirth in chapter 4 of [Fru2005]. We refer to the content leak problem, the “watermarking attack”, the data modification leak and the “malleability weakness” of the plain CBC mode discussed there. Due to these weaknesses we do not recommend using `cbc-plain` as encryption mode in Cryptsetup.

The CBC-ESSIV mode (cipher-block chaining – encrypted salt-sector initialization vector) was invented for LUKS by Clemens Fruhwirth. In this mode the hash algorithm specified after the colon is used to create a second key as hash value of the first one. With this second key the sector number converted to 64-bit and to little-endian is encrypted. That encrypted sector number then re-initializes the chaining in each sector as the plain sector number does in plain CBC mode. For this mode the length of the hash value must be the same as the key length of the cipher. By the initialization of the CBC mode in the sectors with an encrypted value which therefore is unknown to an attacker the “watermarking attack” is made impossible. The other three less serious weaknesses of the plain CBC mode remain in the CBC-ESSIV mode.

The origin of the XTS mode was the XEX (XOR – Encrypt – XOR) mode invented by Phillip Rogaway (see [Rog2004]). It uses a construction of Liskov, Rivest and Wagner (see [LRW2002]) but avoids a vulnerability of their LRW mode when the second LRW key itself is encrypted and stored. In 2007 XEX was standardized by IEEE (Institute of Electrical and Electronics Engineers, Inc.) with a modification and an extension (see [IEEE2007]) with the new acronym XTS (XEX Tweakable Block Cipher with Cipher Text Stealing). The modification was that two keys are used in XTS mode instead of one key in XEX mode. The extension only concerns the case when the size of the encrypted data units isn't a multiple of the block length of the cipher algorithm. For this case the cipher text stealing is needed. For LUKS this doesn't matter as the sector size of 512 bytes is a multiple of the 16 byte or 8 byte block lengths of the algorithms specified by LUKS. In January 2010 the XTS mode in conjunction with AES was approved by NIST (National Institute of Standards and Technology, U.S. Department of Commerce) as a national standard for the USA (see [NIST2010]).

In the XTS mode implemented as `xts-plain64` in the Linux kernel a second key encrypts the sector number converted to 64-bit and to little-endian. This results in the “tweak” for the first block in the sector. A multiplication in the Galois field $GF(2^{128})$ modifies the “tweak” for each subsequent block. This “tweak” then is combined with the block by an exclusive-or (XOR) operation once before and again after the encryption with the first key. As this requires 128-bit blocks the cipher CAST5 can't be used with the XTS mode in LUKS. Besides `xts-plain64` the Linux kernel also implements the mode `xts-plain`. It handles the sector number as a 32-bit number but otherwise does not differ from `xts-plain64`. This has the disadvantage that sectors with the same contents are equally encrypted already after 2^{32} sectors or 2 TiB. Therefore this mode isn't suitable for encryption of large volumes and for that reason it isn't specified in [Fru2011] as a valid mode in Cryptsetup/LUKS.

The XTS mode is not vulnerable to the content leak problem. In CBC and in CBC-ESSIV mode this occurs with a very small probability which however increases with the size of the encrypted volume if two 128-bit blocks happen to be encrypted with the same result by accident. The XTS mode is

also not vulnerable to the “watermarking attack”. In it the attacker is able to provoke the situation of the content leak problem that two different blocks are encrypted with the same result and if this actually occurs he knows that the data tagged by him have been written to the encrypted volume.

The concept of non-malleability has been introduced to cryptography by Dolev, Dwork and Naor (see [DDN2000]). Roughly speaking it means that given a cipher text an attacker can't find a different cipher text such that the corresponding plain texts are related to each other in any predefined relation other than being different. As in CBC mode in CBC-ESSIV mode every bit of a block can be flipped intentionally at the cost of randomizing the preceding block. This isn't possible in XTS mode. It is only “malleable” in the sense that an attacker can change whole 128-bit blocks in an uncontrolled manner while keeping other blocks unchanged. The substantial “malleability” of CBC-ESSIV mode, however, is a serious weakness in our opinion. We therefore recommend to use the mode `xts-plain64` in Cryptsetup/LUKS. The mode `xts-plain64` is identical with the mode of encryption TrueCrypt uses since version 5.

The data modification leak remains in all modes specified by [Fru2011]. An attacker watching the encrypted volume at two distinct points in time may deduce from the comparison of his observations which blocks have changed in the meantime. But this weakness is inevitable for a volume encryption as the complete encrypted volume can't be rewritten with every change of some data in it. This could only be mitigated if larger units – for example entire 512-byte sectors instead of 16-byte blocks – were selected as units of data encryption. This is the goal pursued by the development of another standard of IEEE (see [IEEE2010]). We consider Cryptsetup/LUKS with the XTS mode to be secure even though it does not support this standard.

Derivation of User Keys from Passwords

In Cryptsetup/LUKS a user key is derived from his password by the *“Password-Based Key Derivation Function 2”* (PBKDF2) as it is described in the standard PKCS #5 of RSA Laboratories (see [PKCS5v20] and [RFC2898]) which has been updated in 2006 (see [PKCS5v21]) to incorporate new hash algorithms. In Cryptsetup/LUKS eight different “slots” for up to eight different user keys are available. With each user key which has been activated the volume can be decrypted. This offers a greater flexibility for team usage as TrueCrypt does where only one password may decrypt the volume. After an encrypted volume has been created with the command `cryptsetup luksFormat` more user keys may be added with the command `cryptsetup luksAddKey`. For that only one already valid password has to be supplied. With one of the commands `cryptsetup luksRemoveKey` or `cryptsetup luksKillSlot` each single user key may be deleted. In the first case you have to give the password to be deleted. In the second case the slot number of the user key to be deleted must be specified.

In PBKDF2 the necessary key length is divided into blocks of the length of the hash value for the used hash algorithm. The last block may be smaller if necessary. Then the salt value stored in the key slot is supplemented with a four byte representation of the block number. With this supplemented salt value and the password a HMAC value is calculated. This procedure is iterated by calculating a new HMAC value from the last HMAC value and the password again and again. The final HMAC value makes the block of the key. For the last key block this still has to be truncated if its length is smaller than the length of the hash value.

The standard PKCS #5 recommends at least 1000 iterations for the derivation of the user key from the password. The point of this is is to make the derivation of the user key from the password expensive. No method is known by which an attacker might abridge this procedure. If he checks a large number of potential passwords from a “dictionary” specially prepared for this purpose he has to devote this large effort to each password from his dictionary. So his total computational effort is pushed up tremendously by the iteration count.

TrueCrypt also uses PBKDF2 from the standard PKCS #5 for derivation of the user key from the password. The main difference between Cryptsetup/LUKS on the one hand and TrueCrypt on the other hand is the iteration count. TrueCrypt for the hash algorithms SHA-512 and Whirlpool exactly

executes the recommended minimum of 1000 iterations. For the old hash algorithm Ripemd-160 it does 2000 iterations. Cryptsetup/LUKS on the other hand does a benchmark test on the computer where it is running how many iterations it can do within one second. With the option `--iter-time` or `-i` any other time may be chosen therefore and specified in milliseconds. Then the result of this benchmark test will be the number of iterations for the derivation of the user key from the password. On up to date hardware the thus chosen number of iterations will be many times over the constant values chosen by TrueCrypt. On very fast processors it may even reach millions.

This choice of the iteration count with a benchmark test in our opinion is an essential advantage of Cryptsetup/LUKS over TrueCrypt. The password mostly is the weakest point in the security of a volume encryption and dictionary attacks become better and better with the increasing speed of hardware sold at a time. This is encountered by Cryptsetup/LUKS by pushing up the amount of work for a dictionary attack to the same extent as hardware gets faster. Today an attacker has to pay about 100 to 1000 times more for the computing power for a dictionary attack on a Cryptsetup/LUKS encrypted volume created on current hardware than he has to pay for the same attack on a TrueCrypt volume encrypted with the same password.

After a program for volume encryption has derived the user key from the supplied password it has to verify in a second step that this key and therefore the password is correct. TrueCrypt does this by decrypting the remainder of the header and checking for the string "`TRUE`" at a predetermined position in the decrypted header. In addition it calculates two CRC-32 checksums and compares them with values from the decrypted header.

Cryptsetup/LUKS follows another procedure for the verification. It decrypts with the derived user key the key material of the key slot used. For the resulting master key a checksum is calculated again by PBKDF2 from the standard PKCS #5 and a salt value from the partition header. This is compared with the value of that checksum stored in the header. So this second step also is much more expensive in Cryptsetup/LUKS as it is in TrueCrypt where it is negligible. However, an attacker could try to reduce the computing effort for the verification by decrypting the first few sectors of the volume with the calculated master key and checking whether they contain a known file system or only random data.

Anti-forensic Information Splitting of Keys

Anti-forensic data storage (see [Fru2004] and [Fru2005]) is a feature specially developed for LUKS by Clemens Fruhwirth. It was meant as a counter-measure against recovery of a deleted user key from remnant magnetizations on a magnetic hard disk in a forensic laboratory. No other volume encryption program has such a feature except for FreeOTFE which supports the LUKS format. For modern solid-state drives (SSD), USB flash drives and compact flash memory cards the problem of forensic data recovery is even more serious as the wear leveling algorithms in the controller of such media might prevent a user key from being actually deleted at all.

The idea of anti-forensic information splitting in LUKS is to enlarge the size of every storage region for the master key encrypted with one of the user keys such that all parts of this storage region are required in order to recover the master key. This aims at a marked increase in probability that at least one part of the key can't be recovered from remnant magnetizations or has escaped the wear leveling algorithm preventing it from being actually deleted. If this has been achieved the whole key and also any part of it can't be recovered.

For this purpose the information of the master key is split to a number of so-called "stripes" each having the size of the master key. In principle according to the LUKS standard this number may be chosen arbitrarily. However, Cryptsetup on creating a new encrypted volume always chooses the fixed value 4000 for this which can't be changed by an option of Cryptsetup. All stripes except for the last one are filled with random values. The last stripe is calculated such that from all stripes together the master key may be deduced. The calculation goes as follows: With the hash algorithm, also used to derive the user key from the password, the information in the stripe is diffused and then added to the next stripe by an exclusive-or (XOR) operation. Thereafter the modified next

stripe is processed the same way. In the last step the result of the diffusion is added by an XOR-operation to the master key which makes the last stripe.

For the cipher algorithms specified in LUKS according to [Fru2011] the greatest key length is 256 bits or 32 bytes respectively. With the XTS mode, which needs two keys, this doubles to 512 bits or 64 bytes respectively. Hence, the storage region for the split master key may have at most 256000 bytes which is 250 KiB. Only after this splitting the storage region is encrypted with the key derived from the password of the user and with the chosen encryption mode. Finally this is written to the data medium.

Manufacturers of solid-state drives (SSD), USB flash drives and compact flash memory cards only publish very scanty information on their wear leveling algorithms. Documents available on the internet (see e.g. [NIC2012]) with respect to the blocks on which the wear leveling algorithms operate talk about block sizes ranging from 16 KiB to 1 MiB depending on the size of the SSD. According to that information 4000 stripes in anti-forensic information splitting today are no longer sufficient in order to decrease the probability of forensic recovery of a deleted key on a larger solid-state drive.

The Random Number Generator in Cryptsetup

Cryptsetup does not implement an own pseudo-random number generator but uses that of the Linux system – either from the device `/dev/urandom` or from `/dev/random`. In this respect it does not differ substantially from TrueCrypt on a Linux system. In [UPR2011] we criticized the random number generator of TrueCrypt. This likewise applies to Cryptsetup. Weaknesses of the pseudo-random number generator of Linux are in-depth analyzed in [GPR2006] by Gutterman, Pinkas and Reinman. However, the right place to fix these weaknesses is the Linux kernel and not Cryptsetup. But the Linux kernel developers rejected the criticism of Gutterman, Pinkas and Reinman as mostly academic (see [Edge2006] and [Tso2006]). They were only willing to do some small corrections.

In [UPR2011] we pushed the pseudo-random number generator Yarrow (see [KSF1999]). Gutterman, Pinkas and Reinman on the other hand recommend the architecture of the pseudo-random number generator of Barak and Halevi (see [BH2006]) which has a formal proof of security. It deals with the following three properties of the pseudo-random number generator:

- *Resilience.* The output of the generator looks like genuine random data to an attacker with no knowledge of it's internal state. This holds even if that attacker from some time on has complete control over data that is used to refresh the internal state.
- *Forward security.* Past output of the generator may not be deduced neither in total nor in part by an attacker who learns the internal state of the generator at a later time.
- *Backward security/Break-in recovery.* An attacker who learns the internal state of the generator at some time cannot deduce any predictions on future output of the generator after it has been refreshed with data of sufficient entropy unknown to the attacker.

In [GPR2006] the lacking forward security of the Linux pseudo-random number generator is criticized. The delineated attack however requires a very high computing power. In a secure environment such as Ubuntu privacy remix where events like mouse movements and keyboard input unpredictable for the attacker are also available for the refreshment of the generator this criticism is irrelevant in practice. This is so because the secure environment prevents that an attacker may learn the internal state of the pseudo-random number generator. In order to enhance this security ensured by the system we recommend to shut down the computer instead of leaving it unattended when it is no longer needed after an encrypted volume has been created. Otherwise an attacker might access the system, possibly acquire root privileges and then reading the internal state of the pseudo-random number generator from kernel memory.

The devices `/dev/urandom` and `/dev/random` on a Linux system are distinguished by their behavior in case that according to the estimation of the system not enough entropy from new random

events has refreshed the internal state of the generator. In that case `/dev/random` blocks in contrast to `/dev/urandom` which continues to provide pseudo-random values. The recommendation on Linux is to use `/dev/random` only for the most valuable long-lived cryptographic keys and to use `/dev/urandom` for everything else. Cryptsetup by default uses `/dev/urandom`. This, however, may be changed with the option `--use-random` on the command line so that `/dev/random` will be used. If mouse movements or keyboard input from the user are available as a source of entropy which could prevent the device from blocking forever, we recommend to use this option.

The Format of LUKS Headers

Cryptsetup/LUKS writes a header at the beginning of an encrypted partition. This header starts with the partition header which has a size of 208 bytes. Then eight slots for up to eight different user keys follow each having a size of 48 bytes. The partition header and the eight slots are stored unencrypted. After that space for key material is reserved where for each of the eight slots the master key enlarged by anti-forensic information splitting and encrypted with a key derived from the users password may be stored. The size of this reserved region depends on the size of the master key. But it is always aligned to sector boundaries.

The following table specifies the format of the partition header.

Start	Size	Description of the field
0	6	Magic label consisting of the bytes `'L'`, `'U'`, `'K'`, `'S'`, `0xba`, `0xbe`
6	2	Version number of the LUKS format, currently `0x00`, `0x01`
8	32	Name of the cipher
40	32	Name of the mode of encryption
72	32	Name of the hash algorithm
104	4	Number of the Sector, where the encrypted payload begins
108	4	Size of the master key in bytes
112	20	Value of the checksum calculated by PBKDF2 for the master key
132	32	Salt value for calculating the checksum by PBKDF2
164	4	Iteration count for calculating the checksum by PBKDF2
168	40	UUID of the partition

The format of each of the eight slots is specified in the following table.

Start	Size	Description of the field
0	4	State of slot, enabled (`0x00`, `0xac`, `0x71`, `0xf3`) or disabled (`0x00`, `0x00`, `0xde`, `0xad`)
4	4	Iteration count for calculating the user key by PBKDF2
8	32	Salt value for calculating the user key by PBKDF2
40	4	Number of the sector, where the key material for the slot starts
44	4	Number of anti-forensic stripes

These eight slots follow one immediately after the other subsequent to the partition header. The specification of start values for the fields here are to be interpreted relative to the start of the slot.

7. Conclusion

Cryptsetup with LUKS is a highly secure program for encrypting whole data media or partitions thereupon. The encryption algorithms and other security mechanism it implements comply with the current state of the art in cryptography. We found no back door or security-related mistake in the published source code. If you use this program in a secure environment such as Ubuntu privacy remix you may assume with high certainty that no one can get access to the data stored in your volumes as long as they are closed, the passwords are really good and the attacker doesn't apply highly advanced methods below the layer of the operation system, such as BIOS rootkits, hardware keyloggers or video surveillance. A special strong point of Cryptsetup with LUKS is its high power of resistance against dictionary attacks. This resisting power is adapted to the increased speed of hardware when new encrypted volumes are created on new up to date hardware.

Just as for TrueCrypt (see [UPR2011]) we recommend for Cryptsetup as well to compile your binaries yourself from the source code if you don't want to put blind confidence in the vendor of your Linux distribution. For Ubuntu privacy remix 12.04 we did that. In our guide for building this live DVD this is also advised. For that purpose we described in section 3 in detail the steps to build a Debian package from that sources. In section 2 we listed the MD5 and SHA1 fingerprints of the source code we analyzed. So you may check that you have the same source code if you want to compile it yourself. In order to do so use the programs `md5sum` and `sha1sum` with the source code archive as a parameter.

From the analysis in section 6 we draw the following conclusions. They are meant as a summary. For the rational behind them see section 6.

- The encryption algorithms AES, Twofish, Serpent and CAST6 are well suited. As an international standard AES is particularly recommendable. For the mode of encryption XTS should be selected. Compared to the CBC-ESSIV mode it has the advantage of being hardly malleable. Other modes of encryption in LUKS are out of the question. The hash algorithm should best be selected such that the length of the hash value matches the length of the key. For one of the above mentioned encryption algorithms with the XTS mode SHA512 fits best. For one of these algorithms with the CBC-ESSIV mode SHA256 is suitable.
- We recommend to use the option `--use-random` of Cryptsetup and to wiggle with the mouse for about a minute before and during the creation of a new encrypted volume with Cryptsetup/LUKS.
- With the option `--iter-time` or `-i` respectively and a time in milliseconds of significantly more than 1000 the resistance against dictionary attacks can be further enhanced. This is particularly advisable on older hardware. For this increased security you have to accept the disadvantage that later opening of the volume will always take longer as determined by the given time.
- If a team or group of up to eight people has to work with a common encrypted volume each team member may use his own password. New user keys may be added to the encrypted volume and old user keys may be removed.

Bibliography

[BH2006] Boaz Barak and Shai Halevi: *A model and architecture for pseudo-random generation with applications to /dev/random*, 2006, http://eprint.iacr.org/2005/029.pdf

[DDN2000] Danny Dolev, Cynthia Dwork and Moni Naor: *Non-Malleable Cryptography*, 2000, http://www.wisdom.weizmann.ac.il/~naor/PAPERS/nmc.ps

[Edge2006] Jake Edge: *Holes in the Linux random number generator?*, 2006, http://lwn.net/Articles/184925/

[Fru2004] Clemens Fruhwirth: *TKS1 - An anti-forensic, two level, and iterated key setup scheme*, 2004, http://clemens.endorphin.org/TKS1-draft.pdf

[Fru2005] Clemens Fruhwirth: *New Methods in Hard Disk Encryption*, 2005, http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf

[Fru2011] Clemens Fruhwirth: *LUKS On-Disk Format Specification Version 1.2.1*, 2011, http://wiki.cryptsetup.googlecode.com/git/LUKS-standard/on-disk-format.pdf

[GPR2006] Zvi Gutterman, Benny Pinkas and Tzachy Reinman: *Analysis of the Linux Random Number Generator*, 2006, http://www.pinkas.net/PAPERS/gpr06.pdf

[IEEE2007] Security in Storage Working Group of the IEEE Computer Society Committee: *Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*, Institute of Electrical and Electronics Engineers, Inc., Standard P1619-2007, 2007, http://standards.ieee.org/findstds/standard/1619-2007.html

[IEEE2010] Security in Storage Working Group of the IEEE Computer Society Committee: *IEEE Standard for Wide-Block Encryption for Shared Storage Media*, Institute of Electrical and Electronics Engineers, Inc., Standard 1619.2-2010, 2010, http://standards.ieee.org/findstds/standard/1619.2-2010.html

[KSF1999] John Kelsey, Bruce Schneier and Niels Ferguson: *Yarrow-160: Notes on the Design of the Yarrow Cryptographic Pseudorandom Number Generator*, 1999, http://www.schneier.com/paper-yarrow.html

[LRW2002] Moses Liskov, Ronald L. Rivest and David Wagner: *Tweakable Block Ciphers*, in *Advances in Cryptology - CRYPTO 2002*, ed. by M. Yung, Lecture Notes in Computer Science (Springer, Berlin, 2002)

[NIC2012] National Instruments Corporation: *Understanding and Extending the Life of my Solid-State Drive*, 2012, http://www.ni.com/white-paper/10126/en

[NIST2001] National Institute of Standards and Technology, U.S. Department of Commerce: *AES*, 2001, http://csrc.nist.gov/archive/aes/index.html

[NIST2010] Morris Dworkin: *Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices*, National Institute of Standards and Technology, Special Publication 800-38E, 2010, http://csrc.nist.gov/publications/nistpubs/800-38E/nist-sp-800-38E.pdf

[PKCS5v20] RSA Laboratories: *PKCS #5 v2.0: Password-Based Cryptography Standard*, 1999, ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf

[PKCS5v21] RSA Laboratories: *PKCS #5 v2.1: Password-Based Cryptography Standard*, 2006, ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2_1.pdf

[RFC2898] Burt Kaliski: *Password-Based Cryptography Specification - Version 2.0*, Internet Engineering Task Force, Request for Comments: 2898 (RFC 2898), 2000, http://www.ietf.org/rfc/rfc2898.txt

[Rog2004] Phillip Rogaway: *Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC*, 2004, http://www.cs.ucdavis.edu/~rogaway/papers/offsets.pdf

[Tso2006] Theodore Ts'o: *Re: /dev/random on Linux*, 2006, http://lwn.net/Articles/184928/

[UPR2011] Ubuntu Privacy Remix Team: *Security Analysis of TrueCrypt 7.0a with an Attack on the Keyfile Algorithm*, 2011, http://www.privacy-cd.org/downloads/truecrypt_7.0a-analysis-en.pdf