Hard disk encryption with DM-Crypt, LUKS, and cryptsetup

SECRET MESSAGES

If you're serious about keeping secrets, try hard disk encryption with DM-Crypt and LUKS.

BY CLEMENS FRUHWIRTH AND MARKUS SCHUSTER

File encryption is a popular means for ensuring the security and privacy of file-based data. An intruder who breaks through your firewall won't be able to read your private files if they are encrypted, right?

Actually, selective file encryption provided by utilities such as GnuPG covers some of your tracks, but it may not cover all of them. An intruder can still learn about your system – and maybe even reconstruct some of your file data – by snooping through secret files, temporary files, configuration data, and command histories. The */var/spool/cups* directory, for example, could yield a treasure trove of data about files you might have printed in the past, and tools such as the Gnome Thumbnail Factory could be storing an unencrypted thumbnail of your encrypted images.

Rather than combing through every action performed on every file to remove any trace of the data, Linux users can choose to encrypt data at a deeper level using DM-Crypt. The *dm-crypt* kernel module works at the block device level, enabling users to encrypt whole partitions. The process is transparent to the application, provided the user has been granted access to the data. DM-Crypt encrypts the so-called backing device (the physical disk) and uses a virtual block device to provide access to the cleartext content below */dev/mapper*. Users can access this block device to set up and mount the filesystem. This article examines the technology that underlies DM-Crypt and the new LUKS (Linux Unified Key Setup) management tool.

THE AUTHORS

Clemens Fruhwirth is the author of LUKS and a white paper entitled "New Methods in Hard Disk Encryption," which defines the underlying theories. Clemens is also the inventor of ESSIV and the implementor of LRW-AES and EME for Linux.

Markus Schuster is a system integrator with Bits & Bytes (a Bavarian IT service provider); he refers to himself as a free software all-rounder and has been using LUKS ever since its inception.

En Route to a Crypto Setup

DM-Crypt builds on a flexible layer known as the device mapper. Device mapper modules are configured via so-called DM Tables – simple text files that specify how the device mapper should handle access to areas of the virtual disk. The *dmsetup* program parses these text files and uses *ioctl()* calls to pass the details to the kernel.

The DM table format for DM-Crypt is very clumsy for daily use. The software expects the key to be a fixed length hexadecimal string. The module uses the key to encrypt the block device data. However, storing the key permanently in a DM table file is just like leaving your door key hanging on the door knob. Instead, the key needs to be entered whenever you mount the device.

Typing up to 32 hex characters from memory may not be easy, but *cryptsetup* can help. *cryptsetup* is a tool that generates a cryptographic key from a (more simple) pass phrase, then passes the key to the kernel. Figure 1 shows you the cryptsetup environment.

Two important cryptsetup features can be parametrized: key generation and encryption. The former specifies how cryptsetup will generate a key from a password supplied by a user. This defaults to a hash algorithm, which gives the user the freedom of selecting a password of any length. The hash will compress the information to provide a fixed number of bytes. Figure 1 shows cryptsetup using its defaults: the Ripemd-160 hash generates a 256-bit key.

Two parameters need to be selected for the encryption process: the algorithm and the mode. cryptsetup passes these parameters and the derived key to the kernel, and the DM-Crypt module coordinates the procedure, using the Crypto-API to handle encryption.

Use the Force, LUKS

Unfortunately, there is a downside to cryptsetup. It separates the details on what to do with a set of encrypted information from the encrypted information. The cryptsetup parameters are mostly located in scripts or configuration files which, obviously, can't be on the encrypted partitions. If you lose these files or can't remember the settings for a portable disk, you will lose access to your encrypted data. LUKS (Linux Unified Key Setup) removes this segregation.

LUKS is a formal standard [3], implemented by the cryptsetup-LUKS tool [4] (Figure 2). The latter is a fork of the original cryptsetup. LUKS defines a header for DM-Crypt partitions (Figure 3); the header includes all the information for safe key generation. As the header is part of the encrypted partition, the settings are always available right where they are needed.

cryptsetup-LUKS and the original cryptsetup also differ with respect to the way they generate a key from a passphrase (Figure 2). LUKS password management is based on three concepts: key hierarchies, PBKDF2, and anti-forensic information storage.

Secure Password Management

The legacy cryptsetup application passes the key, which is generated from the password, directly to the kernel. The major drawback to this approach is that the software needs to re-encrypt all data whenever the password is changed. cryptsetup-LUKS introduces an additional password management layer to remove this need. The key hierarchy inserts an extra encryption layer between the derived key and the key used by the kernel to protect the data on the partition. Thus, the derived key only protects the so-called master key. which encrypts the data on the partition (Figure 2).

To change the password, cryptsetup-LUKS decrypts the master key using the old password, re-encrypts the key using the new password, and overwrites the copy of the old master key with the new value. As the cleartext master key is not affected by this process, the encrypted partition data remains valid. This can save you half a day's work if you need to decrypt 120GBytes; the key hierarchy reduces the time needed to change a password to just a few seconds.

LUKS stores the encrypted master in the partition header without imposing a single copy restriction. To support multiple passwords for a single partition, LUKS can store multiple, equivalent copies of the master key and encrypt each one of them with a different string. Each

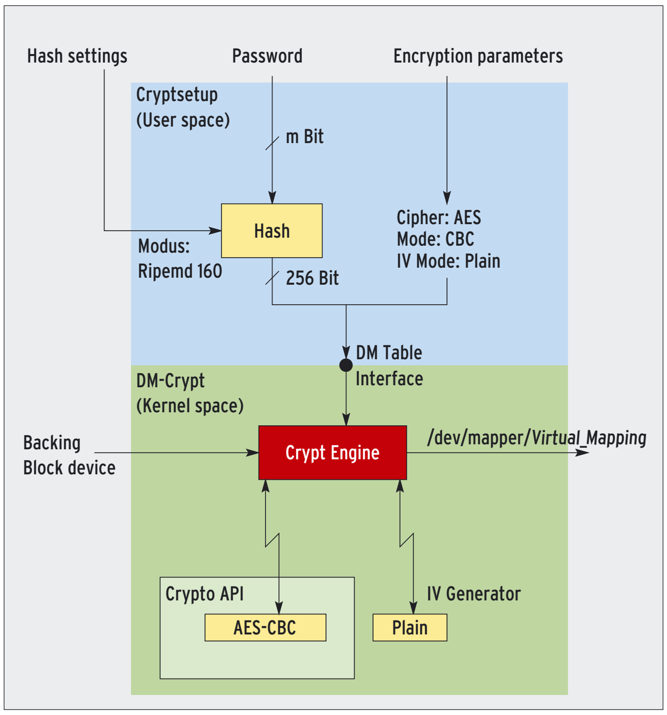


Figure 1: cryptsetup (top) prompts the user for a password and uses a hash to create a fixed length key, which it then passes on to the kernel (center). DM-Crypt (bottom) uses the key to encrypt and decrypt data on the hard disk (or backing block device).

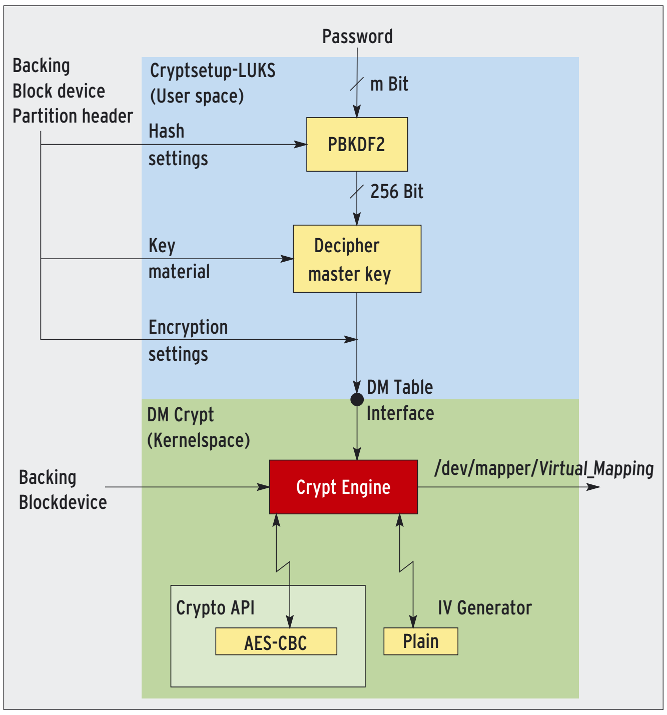


Figure 2: cryptsetup-LUKS stores the parameters for the encrypted partition in the backing block device partition header (top left). The derived key protects the master key, which encrypts the data on the partition.

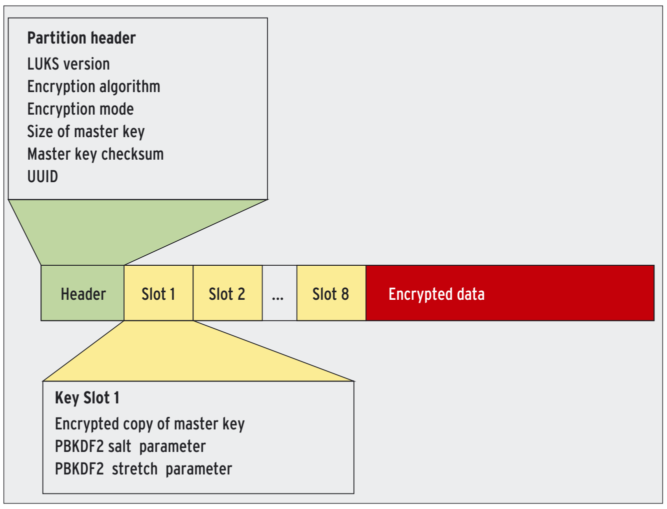


Figure 3: LUKS adds the parameters needed by cryptsetup-LUKS to generate the key from a password entered by a user to the header of the encrypted partition. Each key slot contains an encrypted copy of the master key which DM-Crypt uses for data protection.

of these passwords gives the user access to the cleartext content on the disk. This is particularly useful if you wish to store a contingency password or give multiple users separate credentials. LUKS reserves enough key slots in the header for up to eight passwords (Figure 3.)

Better than a Hash

Just like cryptsetup, LUKS needs a hash algorithm to convert an arbitrary-length password into a fixed number of bytes. To do so, LUKS uses the generic PBKDF2 approach (Password-Based Key Derive Function, Version 2). PBKDF2 is a PKCS#5 (Public Key Cryptography Standard 5) component. PKCS#5 was specified in RFC 2898 [5]. Among other things, PBKDF2 uses salting and stretching to prevent dictionary attacks.

Users prefer short, easily-remembered passwords. Dates of birth and pet names are much more common than random 22-character strings. Unfortunately, you need at least 22 characters to represent a 128-bit key. But there are not many people who would relish the thought of remembering, or even typing *Sq5woq7501VUE5irAXau.a* every day. A useful derivation function satisfies both requirements: the user can type an easily-remembered password, while the function generates a more complex key.

An algorithm that blows up a short password to provide 128 bits of key material, needs to bridge the so-called entropy gap, that is the gap between the degree of randomness in the password domain and the key domain. Simple padding would produce a bigger key, but it would be no more random than the password, and thus it would be just as easily guessed.

Let's imagine that a user entered only English words; this would restrict the scope of the password domain and not provide enough entropy. An attacker could simply run a dictionary attack instead of trying the 2^{128} keys that a 128-bit key space provides. An English dictionary, for example, may have less than 2^{20} entries. This total of 2^{20} is 108 powers less than the full key space; a fatal reduction, as almost anyone could attack a 20-bit key.

To counteract this problem, PBKDF2 uses a deliberately complex function to derive the key from the password. Although this takes a while, the legitimate user will not mind, because the operation is a once off. An attacker would need to try 2^{20} phrases. If each call takes a second, this would take 12 days (2^{20} seconds). If the user combines two words to form a password, the attack could take up to 30,000 years (2^{40} seconds.) This artificial barrier is referred to as stretching. PBKDF2 uses a stretching function that involves infinitely variable computational effort.

Salting and Stretching

But this is not enough to stop determined attackers. An attacker could create an enormous table containing the input and output from the stretching function to remove the need for number crunching during future attacks. To prevent this from working, PBKDF2 adds a randomly selected string to the password before generating the key. LUKS stores the cleartext version of the string in the partition header.

Now, the attacker needs more than just the PBKDF2 hash for every word in the dictionary. In fact, the attacker would need the hashes for each word in the dictionary and for every combination of the appended string. The longer the salt, the bigger the attacker's table would need to be. PBKDF2 pushes the size of the table to an unimaginable scale. The universe has fewer atoms than the number of entries the universal dictionary would need to contain every single PBKDF2 combination.

With all hope of using tables dwindling, attackers are forced back to number crunching. The legacy Unix password mechanism uses a similar approach, by the way: however, the salt is a lot shorter in this case (12 bits stored in the first two digits.)

Shredding

As we mentioned earlier, data shredding on magnetic storage devices is very difficult to perform [2]. To effectively change or delete passwords in the key hierarchy, it is vital to completely destroy the old copy of the master key. With a bit of luck, a user might hit the right hard disk sector after several attempts and physically overwrite the old master key. But luck is something that users and cryptographers don't typically rely on.

The hard disk firmware actively combats data shredding, as its major concern is data safety. One way a hard disk provides more safety is by remapping bad blocks, a simple technique for detecting sectors that are hard to read. The firmware automatically copies these sectors to an area of the disk specially reserved for this purpose and redirects any future read or write operations for the original sector to the copy.

The original sector can't be deleted from this point onwards, as the firmware will redirect any write attempts to the reserved zone. Unfortunately, this could leave fragments of the key on the hard disk, meaning that a data recovery expert, or a determined hacker, could still access the fragments using modified firmware.

This is a big problem for LUKS master keys, which are very small in comparison to the sector size (128, 192 or 256 bits for AES) and thus easily fit into a single sector. All it would take would be for the firmware to decide to redirect this sector to the reserved zone while the old password was active. Neither SCSI nor

IDE have commands that provide access to the original sector.

Beating the Data Recovery Experts

The author of LUKS introduced Anti-Forensic Information Splitting (or AF Splitter for short) to confound recovery experts. To reduce the statistical probability of traces of deleted files surviving on magnetic storage media, AF Splitter expands the data by a factor of four thousand. The expanded data is not redundant; the complete record is always needed to recover the master key. The counterpart to information splitting is merging. That is, the original data is merged in memory where data can easily be deleted.

AF Splitter distributes the original data (Variable x) based on the formula $x = a_1 + a_2 + a_3 + \ldots + a_{4000}$. The algorithm generates the variables a_1 through a_{3999} randomly, and calculates a_{4000} to make the equation balance. The Merger adds the elements a_i requiring every single element to do so; there is no redundancy. If a single element is missing, the equation can't be solved and x can't be calculated.

To shred the data, just one of the 4000 sectors involved in the process needs to be overwritten, as the merging process needs the whole of the expanded record. Of course, it is a lot easier to hit one of the 4000 sectors. The statistics show that this works really well, as you can read at [1]. Thanks to AF Splitter, passwords can be changed without leaving any traces behind. In combination with key hierarchies and PBKDF2, this gives users quality password management for encrypted DM-Crypt partitions.

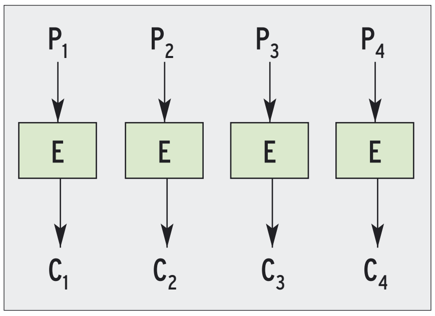


Figure 4: ECB encryption mode (Electronic Code Book) encyphers each block of cleartext independently of all other blocks. This means that the same input P_i to the encryption function E will result in identical output C_i.

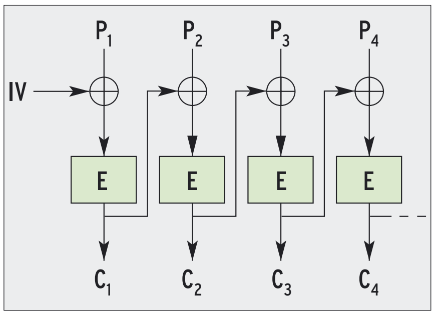


Figure 5: CBC encryption mode (Cipher Block Chaining) XORs the results of one round of encryption XORs with the following block. This ensures that identical blocks of cleartext will produce different cipher text results.

Safe Data Storage

What users mainly expect of an encrypted disk is encryption. DM-Crypt provides two cipher modes: ECB (Electronic Code Book) and CBC (Cipher Block Chaining). Both modes are subject to a few vulnerabilities, all of which are solved by the most promising candidate at present LRW-AES [6] [7] (LRW: Liskov, Rivest, Wagner; AES: Advanced Encryption Standard).

ECB (Figure 4) doesn't really deserve the Cipher Mode label: it stores each individual block cipher result without performing any additional calculations with it. This means that for each key, cleartext will always lead to the same cipher text. Expressed mathematically, ECB is a bijective function of cleartext into the cipher text domain. This is a dangerous trait if an attacker knows the cleartext for an encrypted block due to standardized filesystem headers, for example.

If the attacker knows that the first sector on the encrypted partition starts with a series of zeros, the attacker also knows where else zeros are encrypted. The attacker does not need a key for this, but can simply compare all the cipher text blocks with the start of the partition. If the attacker discovers identical blocks, he or she knows that the decrypted content at this position on the disk comprises zeros. The same principle applies to any other block of cleartext.

Hide and Seek

There are basically two methods to hide these redundancies in the cleartext. One

ESSIV

To generate a watermark, the attacker needs to create two identical sectors on the disk. The aim is to manipulate the encryption mechanism in a way that gives two identical results from encrypting two sectors on the disk. In Figure 5 you can see the attacker can identify all input values for P_i, but not the IV. This is the value used to modify the first cleartext, as shown in Figure 6a.

Watermarking undermines this by applying P_1-1 rather than P_1 to the second sector. The IV for the sector two is one greater than the IV for sector one. This incrementation can be compensated for by subtracting 1 from P_1 (Figure 6b). If the attacker sets all subsequent P_i just like with the first sector, the cipher texts are identical.

ESSIV (Encrypted Salt Sector IV) resolves this issue. It passes the sector number to a function, the result of which depends on the secret key (Figure 6c). The attacker can no longer manipulate P_1 in sector two to compensate for the IV difference. The attacker does not have the key required to calculate the IVs.

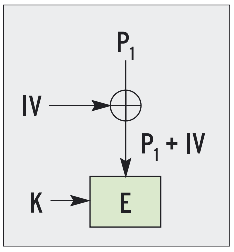


Figure 6a: Traditional CBC starts encrypting by XOR-ing the IV with the first cleartext block.

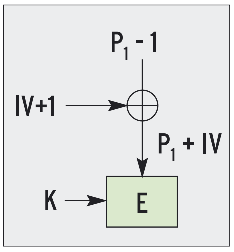


Figure 6b: Watermarking compensates for the changing IV by reversing the change in P_1.

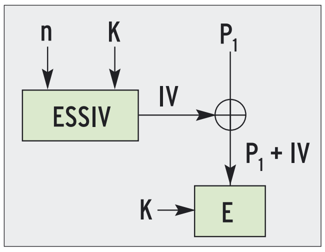


Figure 6c: ESSIV prevents the would-be attacker from calculating the IV because the attacker does not know the secret key material K.

Advertisement

approach is to add another component to the encryption process; this component needs to be unique for each location on the disk, for example the hard disk position. This would mean that identical cleartext blocks stored at different positions on the disk would lead to different encryption results.

The second approach uses encryption modes that take encrypted blocks into account. The easiest way of implementing this is to use recursion. CBC (Cipher Block Chaining) may be simple, but it remains an effective, recursive encryption mode. It XORs the cipher text from the last block of cipher text with the current cleartext. CBC then encrypts the modified cleartext and applies the results to the next block of cleartext.

Take a look at Figure 5 to see how CBC works. Even if several contiguous blocks of cleartext were identical, recursion causes a kind of snowball effect. This link means that identical cleartext blocks are modified using different cipher text results.

Snowball Effect

One characteristic of this kind of recursion is that the first round of encryption has an effect on all subsequent rounds. This is not useful for hard disk encryption, where the whole of the partition would need to be re-encrypted if the content of the first sector changes. The typical answer here is to view each sector as the result of a recursive function and to process each sector independently from the rest.

This leads to a familiar problem: two sectors with identical cleartext result in the same cipher text. Although sectors are a lot bigger than the blocks in a block cipher, the content can still be identical – just imagine a user creating multiple copies of a file, for example. This is where the first trick applies: the sector number changes the encryption by specifying the initialization vector (IV, Figure 5). Two different modifications of the first cleartext trigger different snowball effects and lead to different cipher texts.

The standard variant of DM-Crypt applies the sector number directly as the IV. This is referred to as plain IV generation. Unfortunately, this approach is vulnerable to watermarking attacks, where an attacker crafts data so that he or she

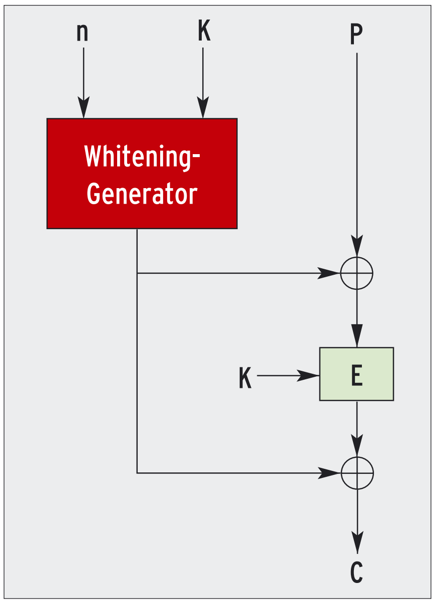


Figure 7: The LRW encryption mode does not use recursion. It prevents ECB style attacks (Figure 4) by adding whitening. The whitening factor is calculated by reference to the hard disk position n and the secret key K.

can rediscover that data without knowing the key.

Watermarks can contain up to 5 bits of information [1]. An attacker could add watermarks to emails, which he or she would then send to the victim to find out where the victim stores the messages. Watermarks could also be added to MP3 files, images, or other files that a suspicious boss could easily send to a member of staff. This makes attacks on a user’s privacy possible. Without needing to decrypt, the spy has access to information about the victim’s hard disk.

The ESSIV (Encrypted Salt-Sector IV) IV generator prevents this. Watermarking assumes a simple relationship between the IVs for two contiguous sectors. This is true in the case of plain; the IV for sector n is followed by the IV $n + 1$ for the next sector. ESSIV adds complexity to the sequence, making it impossible for attackers to calculate the sequence without knowing at least part of the secret key (see the “ESSIV” box.)

Data Whitening

You may be wondering why DM-Crypt uses a mix of recursion and manipulation based on the hard disk position where the latter would be quite sufficient on its own. There is a historic reason for using CBC: it is a tried and trusted approach, and the properties of

CBC have been investigated by many people. The alternatives, which entirely rely on the block number, are young in cryptographic terms.

LRW is one encryption mode that integrates the block number into the encryption routine in a simple and effective way. First, LRW calculates a whitening factor based on the secret key and the block number. It then adds the whitening factor to the cleartext and encrypts the sums, before adding the whitening factor again (Figure 7). These two steps are known as pre-whitening and post-whitening. They link the cipher text with a hard disk position to achieve different encryption results for identical cleartext stored on different parts of the disk

LRW removes the known vulnerabilities associated with CBC while improving performance. Whereas CBC does not scale well for multiple processors, as each recursive step is based on the results of the previous step, LRW can use multiple, parallel processors. The LUKS author, who is also the co-author of this article, Clemens Fruhwirth, has implemented and tested LRW for DM-Crypt, and the release is imminent.

Foiled by the Kernel

This said, LRW is not currently available for DM-Crypt. Linux' high/low memory management design means that kernel modules process more than two high memory data areas. The LRW implementation is based on an attempted generic re-implementation of Scatterwalk (part of Crypto-API), which should be capable of accessing an arbitrary number of high memory areas simultaneously. Due to the current restriction to two memory areas, a generic implementation would not achieve what the author intended, and this has led to him dropping the attempt in frustration [8].

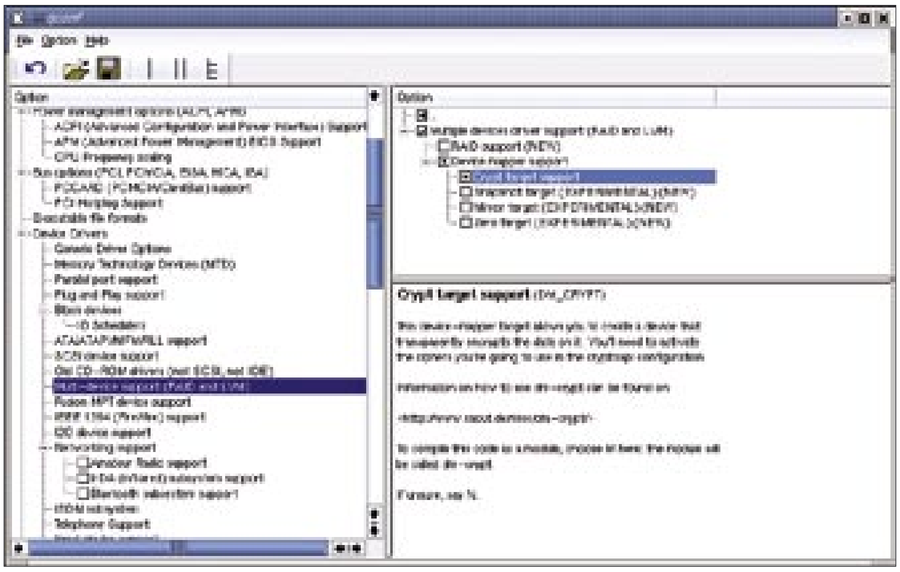


Figure 8: The device mapper is located below Device Drivers | Multi-device support (RAID and LVM) | Device mapper support in the kernel configuration. Crypt target support is required for DM-Crypt.

For the time being, DM-Crypt is the most secure cipher mode implementation for CBC-ESSIV – until someone who is not fazed by the useless and endless discussions on the kernel mailing list [9] steps in and develops a suitable Scatterwalk variant. The authors of this article would be very pleased to see this happen. The math for LRW has been completed and implemented to comply with standards.

Installation

To use DM-Crypt, cryptsetup, and LUKS you need a few kernel modules and a user-space tool. The options for DM-Crypt are hidden below *Device Drivers | Multi-device support | Device mapper support* in the kernel configuration and below *Crypt target support* (Figure 8) in the same section. Note that you need to select *Prompt for development and/or incomplete code/drivers* below *Code maturity level options*, otherwise the Crypt-Target stays hidden.

As DM-Crypt relies on Crypto-API functions, you need to select at least one algorithm in *Cryptographic options | Cryptographic API* (Figure 9). The authors recommend AES. One encryption algorithm is all you need; the cryptsetup-LUKS user-space tool handles the operations, such as hashing to generate the key from the password.

Most Linux distributions set these options by default. You can enter *modprobe dm-crypt* to check. The command should

Listing 1: cryptsetup-LUKS

```
$ dd if=/dev/zero of=verysecret.loop bs=52428800 count=1
1+0 records in
1+0 records out
$ losetup /dev/loop0 verysecret.loop
$ cryptsetup -c aes-cbc-essiv:sha256 -y -s 256 luksFormat /dev/loop0

WARNING!
========
This will overwrite data on /dev/loop0 irrevocably.

Are you sure? (Type uppercase yes): YES
Enter LUKS passphrase: ******
Verify passphrase: ******
$ cryptsetup luksOpen /dev/loop0 verysecret
Enter LUKS passphrase: ******
key slot 0 unlocked.
$ mkfs.xfs /dev/mapper/verysecret
[...]
$ mount /dev/mapper/verysecret /mnt
$ umount /mnt
$ cryptsetup luksClose verysecret
$ cryptsetup luksAddKey /dev/loop0
Enter any LUKS passphrase: ******
key slot 0 unlocked.
Enter new passphrase for key slot: ******
$ cryptsetup luksDelKey /dev/loop0 0
losetup -d /dev/loop0
```

work fine. DM-Crypt became an official Linux component with kernel 2.6.4; the ESSIV IV generator needs at least kernel version 2.6.10.

The LUKS user-space component is available as a download from [4]. There are packages for Debian, Gentoo, Suse and Red Hat; cryptsetup-luks is a standard component in Fedora 4. Users with other distributions can follow standard procedure, *./configure && make && make install* to build and install, assuming that Libpopt, Libgcrypt (Version 1.1.42 or later), and Libdevmapper are installed.

cryptsetup-LUKS

The binary answers to the name of *cryptsetup* and supports several actions. It joins the ranks of tools that allow Linux admins to assign filesystems to block devices and to mount these filesystems. Listing 1 gives you an example. To keep this as uninvasive as possible, the *dd* call in line 1 creates a 50MByte container, which is enabled as a block device using a loop in line 4.

Initially, the most important cryptsetup action is *luksFormat*, which prepares the backing block device (the loopback device in our case) for use in the encrypted environment. This is also the step where you need to decide on an encryption algorithm. The formating action needs the block device, and optionally a file, the content of which will be used as the password. LUKS refers to this file as the key file. The following parameters are useful:

- *-c* specifies the algorithm and for recent kernel versions, the chaining mode and IV generator. These three parameters must be separated by simple dashes (default: *aes-cbc-plain*). The safest variant at present is *aes-cbc-essiv:sha256*.
- *-y* tells cryptsetup to ask twice for the password to avoid typing errors. This parameter does not make sense in combination with a key file.
- *-s* specifies the length of the encryption key.

In line 5 of Listing 1, you can see the complete call. By typing *YES* in line 11, the user confirms that existing data might be lost. The user then sets the first password (line 12.)

Mapping for a Filesystem

To use the block device we just prepared, cryptsetup-LUKS needs to map the physical block device to the virtual block device. The *luksOpen* action takes care of this (line 14.) If the password is stored in a file (see *luksFormat*), cryptsetup needs the *-d* parameter followed by the name of the key file. In our example, the user types a password (line 15.)

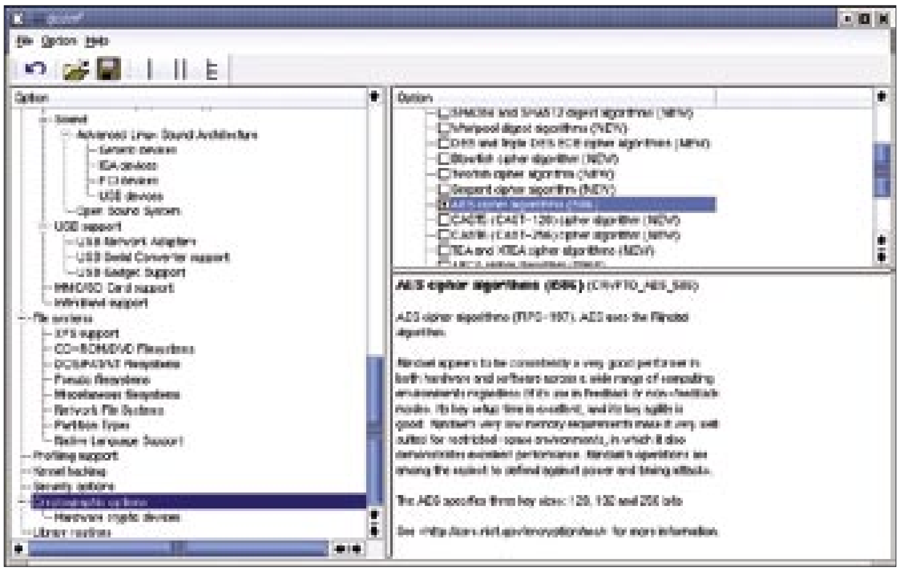
cryptsetup-LUKS automatically creates the block device with the specified name of *verysecret* below */dev/mapper/*. The *mkfs.xfs* call in line 17 puts an XFS filesystem on the device. The result can be mounted as shown in line 19. Don't forget to unmount before applying changes (line 20.)

Cleaning Up

After finishing your work, it makes sense to unmap to avoid opening up a vector to attackers and spies. The *luksClose* action takes care of this.

As mentioned previously, cryptsetup-LUKS can manage multiple passwords per block device. This makes it easy to change a compromised password without re-encrypting your data. The *luksAdd Key* expects the physical block device as a parameter (Listing 1, line 22.) After typing any current password, the tool prompts for an additional, new password. You can also specify a key file.

The *luksDelKey* action (line 26) removes an existing password. It expects the physical block device and the key slot to be deleted as parameters. The latter is the storage location for the key. As cryptsetup-LUKS manages eight passwords by default, key slots 0 through 7 are typically all you need. The program will tell you which key slot a password is stored in when you call *luksOpen* (line 16) or *luksAddKey* (line 24.) ■


AES cipher algorithms (i586) (CRYPTO_AES_586)

AES cipher algorithms (FIPS-197). AES uses the Rijndael algorithm.

[illegible]

The AES specifies three key sizes: 128, 192 and 256 bits

See [illegible] for more information.

Figure 9: As DM-Crypt relies on the Crypto-API for encryption, you need to select at least one algorithm in Cryptographic options | Cryptographic API. AES is the algorithm of choice right now.

INFO

[1] Clemens Fruhwirth, "New Methods in Hard Disk Encryption": *http://clemens.endorphin.org/publications*

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

[3] Clemens Fruhwirth, "LUKS On-Disk Format Specification": *http://luks.endorphin.org/LUKS-on-disk-format.pdf*

[4] LUKS Software: *http://luks.endorphin.org*

[5] RFC 2898, "PKCS #5: Password-Based Cryptography Specification Version 2.0": *http://rfc.net/rfc2898.html*

[6] Moses Liskov, Ronald L. Rivest und David Wagner, "Tweakable Block Ciphers": *http://www.cs.berkeley.edu/~daw/papers/tweak-crypto02.pdf*

[7] IEEE, "Draft Proposal for Tweakable Narrow-block Encryption": *http://www.siswg.org/docs/LRW-AES-10-19-2004.pdf*

[8] Clemens Fruhwirth, "LRW for Linux is dead": *http://grouper.ieee.org/groups/1619/email/msg00253.html*

[9] Kernel mailing list discussion on Scatterwalk changes: *http://lkml.org/lkml/2005/1/24/54*