Linux on Z and LinuxONE

Pervasive Encryption for Data Volumes September 2020



SC34-2782-04

Note

Before using this document, be sure to read the information in "Notices" on page 91.

This edition applies to the software components listed in <u>"Software prerequisites" on page 9</u> and to all subsequent versions and modifications until otherwise indicated in new editions.

[©] Copyright International Business Machines Corporation 2018, 2020.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

I

About this document	v
Distribution hints	v
Summary of changes	V
Chapter 1. Protected and secure volume encryption	1
Chanter 2. Infractivity concents	2
Terminology	
Chapter 3. Setting up the infrastructure	9
Prerequisites	9
Planning	
Loading required modules and components	
Chapter 4. Sample system for data volume encryption	15
Chapter 5. Working with encrypted volumes	
Creating a volume for pervasive encryption	
Opening an encrypted volume	22
Encrypting an unencrypted volume with a secure key	
Re-encrypting a volume from clear key to secure key	
Chapter 6. Managing keys	35
Managing a secure key repository	35
Managing secure LUKS2 volume keys	43
Validating a secure key	
Changing master keys and re-enciphering secure keys	
Sharing master keys across cryptographic coprocessors	
Replacing a cryptographic coprocessor	
Chapter 7. Problem resolution and recovery	
Verifying your configuration	55
Valid physical block size combinations of LVM physical volumes	56
Troubleshooting problems in your environment	
Chapter 8 Recovering secure key encrypted volumes	59
Pacavariant and values from an invalid secure key	50
Pecovering encrypted volumes with a secure key from the repository	
Recovering encrypted volumes with a secure key nom the repository	
Chapter 9. Encrypting volumes without LUKS	63
Volume encryption with cryptsetup plain mode	
Encrypting an unencrypted volume using plain mode	
Changing a master key using plain mode	
Opening an encrypted volume in plain mode	66
Chapter 10. Encrypting swap disks with protected keys	69
Setting up an encrypted swap disk	

Appendix A. zkey - Managing secure keys	71
Appendix B. zkey-cryptsetup - Managing LUKS2 volume keys	85
Accessibility	89
Notices	91
Index	

About this document

This document describes an infrastructure for encrypting volumes using protected and secure keys for encrypting and decrypting data. This infrastructure for protected volume encryption provides end-to-end protection for data at-rest for Linux on IBM Z° and IBM LinuxONE. This publication informs about the required setup and describes various scenarios that deal with the data management on the encrypted volumes, with key management, and with tasks of backup, recovery, and migration.

You can find the latest version of this publication in the Linux[®] on Z library on the developerWorks[®] website at:

www.ibm.com/developerworks/linux/linux390/documentation_dev.html
and on the IBM® Knowledge Center at:
https:https://www.ibm.com/support/knowledgecenter/linuxonibm/
com.ibm.linux.z.lxdc/lxdc_linuxonz.html

• For a video with information on the features and advantages of the infrastructure for protected volume encryption, click or enter the following URL:

https://youtu.be/jDK3ZwEdX4I

• For an illustration of the ease of setting up data volumes for pervasive encryption, watch this video:

https://youtu.be/t2Ph_h0LcsQ

Distribution hints

This publication provides information that is based on the minimum level of required upstream features. Support in a particular Linux distribution might differ.

If your distribution does not include the features that are required for using the infrastructure for protected volume encryption to its full extent, you might have to install them manually.

Summary of changes

Track the changes of this document for each new edition.

Edition SC34-2782-01

This update of the original document SC34-2782-00 describes how to avoid problems that may occur when migrating LVM volumes.

Edition SC34-2782-02

This edition documents valid physical block size combinations of LVM physical volumes.

Edition SC34-2782-03

A new feature of the infrastructure for protected volume encryption enhances the paes_s390 and the pkey kernel modules to allow using randomly generated protected keys without requiring a cryptographic coprocessor. This is mainly useful for encrypted swap disks, or any other cases where the keys may be ephemeral, that means, their lifetime does not extend over different boot cycles or machine migrations.

Edition SC34-2782-04

- The **zkey** and **zkey-cryptsetup** tools are enhanced with new functions:
 - During creation of a secure key, or modification of APQNs associated with a certain secure key, the tools perform a cross check whether all associated APQNs have the same master key.

- The display of the master key register during a **zkey validate** command for a secure key is enhanced with the output of the master key verification pattern in this register. The master key verification pattern is displayed even when the secure key is no longer valid.
- *PBKDF2* is now automatically used as password based key derivation function for key slots when using secure keys. This avoids out-of-memory errors when using the LUKS2 default *Argon2i*.
- A batch-mode option is added to the zkey cryptsetup subcommand and to the zkeycryptsetup reeincipher subcommand. This option suppresses any user confirmation and thus enables a better automation.
- Also, to enable a better automation of commands generated by the zkey cryptsetup function, you can specify values for --keyfile-size or --keyfile-offset to control which part of the key file is used as passphrase. These options and values are passed to the generated commands.
- You can specify the same common passphrase options when generating crypttab entries using the **zkey crypttab** function.
- Both the **zkey** and the **zkey-cryptsetup** commands are enhanced to support generation and management of CCA AES CIPHER keys. Up to now, only CCA AES DATA keys have been supported. This was transparent for the user. Now you can specify the key type at key generation time.
- Also, starting with IBM 4769 Crypto Express7 feature (CEX7S) on IBM z15, both the zkey and the zkey-cryptsetup commands are enhanced to support generation and management of EP11 AES secure keys for use in CPACF protected key operations.

Chapter 1. Protected and secure volume encryption

One of the most valuable assets of a modern enterprise is data that is typically stored on storage systems. This data is often referred to as data at-rest. The method of choice to protect such data in storage systems is end-to-end data encryption. Data is encrypted by the Linux instance applying controlled protected and secure keys. Therefore, the data is protected whenever it leaves the system.



Figure 1. End-to-end encryption

Figure 1 on page 1 shows that end-to-end encryption protects the data during the complete journey from the operating system on the IBM Z mainframe through the cryptographic hardware, the SAN cable and the SAN adapters into the storage server cache, and finally on the storage devices.

In Linux, the most popular method for end-to-end data at-rest encryption is full volume encryption using the dm-crypt kernel component. dm-crypt reads encrypted sectors from a block device (disk, partition, or logical volume), decrypts the data in the sector, and writes it to the reading component (for example, into the page cache).

In the opposite direction, dm-crypt reads data from a buffer of the writing component (for example from the page cache), encrypts the data, and writes it into a sector of the block device. The decryption and encryption happens transparently as long as the application uses file I/O to a file system mounted on a dm-crypt device.

The challenge is to manage the cryptographic keys needed to open an encrypted volume:

- · How can these keys be stored securely?
- How can these keys be protected from being discovered while they are in use?
- How can these keys be protected from being stolen by an intruder?
- How can these keys be protected from discovery by a service engineer that analyzes a system dump?

To manage the challenge of storing cryptographic keys and associating these keys with the volume they encrypt, the Linux **cryptsetup** utility applies the Linux Unified Key Setup (LUKS) volume format. This format provides protection by passphrases and stores passphrase-protected volume keys in the header of the volume. The **cryptsetup** utility uses LUKS to store information (for example about the used cipher and the volume key) for setting up and configuring dm-crypt. On opening a LUKS formatted volume, a

passphrase is required. This might be a valid procedure for personnel computers and laptops, but is not a viable solution to manage a server farm with hundreds of volumes.

The purpose of protected volume encryption is to provide end-to-end encryption of data at-rest in a way that the keys needed to decrypt or encrypt your data are inherently secure. That means that the keys must be represented by objects that cannot be used as keys outside of your system. These objects are secure keys and protected keys. How these types of keys cooperate in the infrastructure for protected volume encryption is described in Chapter 2, "Infrastructure concepts," on page 3.

Chapter 2. Infrastructure concepts

You can exploit the infrastructure for protected volume encryption for transparently encrypting and decrypting Linux block devices using secure and protected keys. For Linux on Z, block devices can be, for example, partitions of ECKD disks (also referred to as DASDs), SCSI disks or partitions, logical volumes, NVMe volumes, loopback devices, or network block devices.



Figure 2. Hardware components of the infrastructure for protected volume encryption

The infrastructure for protected volume encryption makes use of both secure keys and protected keys:

- Secure keys are processed in a Crypto Express cryptographic coprocessor configured in CCA mode (shortly referred to as CCA coprocessor) or configured in EP11 mode (EP11 coprocessor). Secure keys are persistent key objects that can be safely stored on unprotected media, because they are protected by a FIPS 140-2 Level 4 certified hardware security module. Using these keys requires access to a domain of a CCA or EP11 coprocessor where a specific AES master key is activated.
- Protected keys are volatile keys encrypted by the IBM firmware master key of an LPAR or a virtual machine. They are created by a specific instance of an LPAR or a z/VM[®] guest and can only be used by the instance of the operating system that created the protected key. Also, they are only valid as long as the LPAR or the virtual machine that generated the key is running. The operating system has no access to the plain values of protected keys, and protected keys are useless on any other system.

Protected keys support high performance AES cryptography using the acceleration of cryptographic operations provided by the IBM Z Central Processor Assist for Cryptographic Functions (CPACF), which is implemented in the IBM Z CPUs.

In the infrastructure for protected volume encryption, a protected key can securely be derived from a secure key using a CCA or EP11 coprocessor. The coprocessor and the IBM Z mainframe then cooperate to perform the required re-wrapping operation.

Therefore neither secure nor protected keys are of any use outside their system. Even if stolen, they cannot be used to decipher data stored in the storage system or in transit through the SAN from a system owned by adversary persons.

The secure key, which is stored on the volume (for example, in the LUKS2 header) is actually the effective key wrapped by the tamper-proof master key of the domain. When the volume contents need to be accessed, the secure key is unwrapped from the master key by the cryptographic coprocessor to obtain

L

I

I

the effective key. The effective key is passed through a secure channel to the CPACF where it is rewrapped by a temporary firmware master key specific to the LPAR or virtual guest. This re-wrapped key is called the protected key. The effective key inside the protected key cannot be discovered by the operating system. The effective key is unwrapped from the protected key inside the CPACF and used in cryptographic functions performed by the CPACF whenever the Linux kernel (dm-crypt) reads and writes data to the disk volumes.

The infrastructure for protected volume encryption consists of the following components:

- The IBM Z cryptographic hardware:
 - Crypto Express cryptographic coprocessors configured as CCA coprocessors or as EP11 coprocessors.
 - CPACF
- The Linux kernel with:
 - The paes_s390 module supporting protected key AES cryptography (also called the PAES cipher)
 - The pkey module for generating secure keys and deriving protected keys from secure keys
 - dm-crypt, preferably supporting LUKS2
- The Linux **cryptsetup** utility managing the LUKS on-disk format for volumes, preferably supporting LUKS2, to store cryptographic information to set up and configure dm-crypt. Such volumes are referred to as dm-crypt volumes in this documentation.
- The **zkey** and the **zkey-cryptsetup** utilities from the s390-tools package to generate and manage secure keys.

Figure 3 on page 4 illustrates the setup of the infrastructure for protected volume encryption. It presents a view on the installed Linux components and shows how they cooperate to implement the functionality of the infrastructure for protected volume encryption with an end-to-end encryption of volumes. An end user can transparently encrypt or decrypt the data of an application without any further interaction.



Figure 3. Volume encryption - system administrator view

This document mainly explains how to set up and manage dm-crypt volumes encrypted with the PAES cipher using the LUKS2 format. Alternatively, it is possible to use the plain format together with the PAES cipher. This is described in Chapter 9, "Encrypting volumes without LUKS," on page 63.

Note: You cannot set up a dm-crypt volume using the PAES cipher with **cryptsetup** and LUKS version 1. Use **cryptsetup** version 2.0.3 or newer in combination with PAES and the LUKS2 format.

The IBM Z cryptographic hardware

The concepts and features of the infrastructure for protected volume encryption solely work within the IBM Z cryptographic hardware environment. This environment is a combination of hardware components and processing infrastructures that provide a rich array of cryptographic capabilities and functions. Among these, especially the functions for encrypting or decrypting data and for managing keys are exploited by the infrastructure for protected volume encryption.

The core hardware component of the IBM Z cryptographic hardware environment are devices that are called *cryptographic coprocessors*. Sometimes you can read the synonymous terms *cryptographic adapter* or *cryptographic card*.

An IBM Z cryptographic hardware environment can consist of several cryptographic coprocessors. For the infrastructure for protected volume encryption, use at least two of them to provide the reliability that is expected from an IBM Z mainframe. Crypto Express cryptographic coprocessors are partitioned into multiple domains, where each of the domains maintains a domain-specific master key.

To access and securely manage multiple cryptographic coprocessors with their domains and master keys, you require a Trusted Key Entry workstation (TKE) providing the following features:

- Loading and maintaining master keys and operational keys into the cryptographic coprocessor.
- Providing a cryptographic hardware migration feature. The TKE allows you to collect data from one cryptographic coprocessor and apply the data to another coprocessor.

A TKE comprises the following components:

- The hardware: a workstation with a cryptographic coprocessor
- Smart card readers and smart cards.

The TKE version is closely associated with the host platforms that it supports.

The PAES cipher

The PAES cipher is used by the infrastructure for protected volume encryption to enable the utilization of a protected key for the AES algorithm. The PAES cipher converts a secure key into a protected key for enhanced performance using the pkey module.

For more information, see "Cipher mode considerations" on page 13.

cryptsetup

The Linux **cryptsetup** open source utility allows you to manage dm-crypt volumes. In particular it supports formatting volumes with LUKS1 or LUKS2 and also supports managing formatted dm-crypt volumes.

LUKS (LUKS1 and LUKS2)

LUKS stands for Linux Unified Key Setup. By providing a standard on-disk-format, it does not only facilitate compatibility among distributions, but also provides secure management of multiple user passphrases. LUKS stores all necessary setup information in the volume header, enabling users to transport or migrate their data seamlessly. The LUKS volume encryption key (LVEK), which is the key with which the volume is encrypted, is also stored as part of the header. Of course, the LVEK is not stored in clear, but is encrypted by a key derived from a user's passphrase.

A LUKS header contains multiple key slots. Each key slot can contain the LVEK encrypted by a user passphrase. The user must enter the correct passphrase to open the key slot. Alternatively, a key file that

contains the passphrase can be used for an unattended opening of a key slot (for example, at Linux startup).

The **cryptsetup** utility manages the LUKS header and requests the passphrase to decrypt the LVEK and to create a logical device using the dm-crypt device mapper target.

LUKS2 is a new header format introduced with **cryptsetup** version 2.0. It covers all possibilities of the previous LUKS version (LUKS1) and additionally provides various extensions.

To format a device with a LUKS2 header, use **cryptsetup** together with the **luksFormat** subcommand and the LUKS2-type option as shown in <u>"Creating a volume for pervasive encryption" on page 17</u>. When opening an encrypted volume, the LUKS format is automatically recognized.

For more information about LUKS, read the following documentation:

LUKS1 On-Disk Format Specification

LUKS2 On-Disk Format Specification

Terminology

Often, multiple different terms are used to denote the same technical construct. This publication defines a certain terminology and uses it consistently throughout all topics.

cryptographic coprocessor

A Crypto Express cryptographic coprocessor is often also referred to as cryptographic card or cryptographic adapter or just adapter. If a cryptographic coprocessor is tamper-proof and performs cryptographic operations on cryptographic keys protected by the coprocessor, then such a coprocessor is called a *hardware security module (HSM)*.

A cryptographic coprocessor is divided into multiple domains, also called *AP queues*. Each *AP queue* acts as an independent cryptographic device (HSM) with its own state, including its own master key.

If this publication mentions a cryptographic coprocessor, this term is used as a synonym for a coprocessor configured in CCA mode (CCA coprocessor) or configured in EP11 mode (EP11 coprocessor).

APQN

AP queues are identified by their adjunct processor queue number (APQN). An APQN designates the combination of a cryptographic coprocessor (adapter) and a domain. In the infrastructure for protected volume encryption, for example, the APQN 03.0039 specifies the domain 0039 on the cryptographic coprocessor with ID 03.

clear key

A clear key is a key in plain text. That is, the bit pattern of a clear key is the one that is used in the mathematical description of a cipher. Therefore, whoever knows the clear key can perform cryptographic operations (like encrypt or decrypt) using that clear key.

Important: Use IBM-provided utilities and hardware to generate a clear key. You should never explicitly generate a clear key unless this operation is performed in a clean room environment. Otherwise you risk being observed during the clear key generation, or some software components still contain some remains of the generated clear key.

master key

A master key is a wrapping key or a key-encrypting-key (KEK) used to encrypt a key. In the IBM Z cryptographic hardware, a master key is protected from any access from outside this hardware.

There are two types of master keys in the IBM Z cryptographic hardware environment:

HSM master key

Each domain of a Crypto Express cryptographic coprocessor can contain active master keys which are used to generate secure keys. The IBM Crypto Express CCA coprocessor actually can maintain four master keys per domain: one to wrap DES/Triple DES keys, one to wrap AES (and HMAC)

keys, one to wrap RSA keys, and one to wrap ECC keys. In the context to this document, only AES master keys are of importance.

Starting with IBM z15, the infrastructure for protected volume encryption also supports the use of the latest generation of Crypto Express cards (CEX7S) as HSMs in EP11 mode (CEX7P). An EP11 cryptographic coprocessor (CEX7P) uses an AES master key for encryption of the secure key. In the EP11 documentation, this master key is often referred to as *wrapping key*.

firmware master key

For each virtual server (LPAR or guest), the firmware maintains master keys in storage, which are used to generate protected keys. These master keys are protected against operating system access. In the context of this document, only the AES master keys of the firmware are of importance.

effective key

An effective key is a plain text key. This term is used in connection with wrapped keys. For example, for a secure key consisting of a key wrapped by a master key, that key is the effective key of the secure key. If a protected key is derived from a secure key, then both the protected key and the secure key have the same effective key. The effective key of a multiply wrapped key is the innermost plain text key.

protected key

A protected key is a key encrypted by a firmware master key. The effective key of the protected key cannot be discovered by the operating system. Protected keys can be used in cryptographic functions performed by the CPACF component of the IFL (CPU) and therefore are performed at CPU speed.

A protected key is only valid inside the virtual server instance that generated the protected key. A protected key of a virtual server can be derived from a secure key of an HSM attached to that virtual server. In that case, the effective keys of the protected key and the effective key of that secure key are the same.

secure key

A secure key is a key encrypted by an HSM master key. Secure keys can be used in cryptographic functions performed by the HSM. Thus, each cryptographic function on a secure key requires I/O operations to the HSM. A secure key is only valid in the HSM it was generated in.

A secure AES key generated in a domain of an IBM Crypto Express coprocessor (configured in CCA or EP11 mode) can be transformed into a protected AES key for the virtual server attached to that domain of the CCA or EP11 coprocessor. Then the effective key of the generating secure key and the effective key of the derived protected key are the same.

In the infrastructure for protected volume encryption, you can decide between various types of secure keys: a CCA AES DATA key, a CCA AES CIPHER key, and an EP11 AES secure key. For information about the differences between these types, read the <u>"Secure key considerations" on page 12</u>.

hardware security module (HSM)

A hardware security module is a tamper protected cryptographic device that protects secrets (typically master keys) from being inspected. IBM Crypto Express CCA coprocessors and EP11 coprocessors are certified as HSMs. Each domain of an IBM Crypto Express coprocessor constitutes a (virtual) HSM and maintains a domain-specific master key or a set of master keys.

volume

L

A volume refers to a Linux block device (for example, a DASD partition or a SCSI disk, or a logical volume). This publication also uses the term disk or disk partition when referring to an example of a volume.

LUKS1 or LUKS2 volume key

A key used to encrypt and decrypt the user data on a volume formatted in LUKS1 or LUKS2 format. A key slot in the LUKS1 or LUKS2 header stores a wrapped copy of this volume key, where the wrapping key is derived from the user's passphrase. In the infrastructure for protected volume encryption, the LUKS2 volume key is a secure key. Hence, the effective volume key is twofold protected: it is encrypted by an AES master key from a CCA or EP11 coprocessor and by a wrapping key or KEK derived from a passphrase. Therefore, to unlock a LUKS2 volume, a passphrase - provided interactively or from a key file - is required to decrypt the outer wrapping.

The security provided by the passphrase is typically much lower than that provided by the wrapping AES master key. Therefore the password may be exposed without any loss of security. When a secure key for the PAES cipher is provided to dm-crypt in order to open a volume, it automatically transforms this secure key into a protected key that can be interpreted by the CPACF. The actual effective key of the LUKS2 volume key is never exposed to the operating system.

Chapter 3. Setting up the infrastructure

It is important to be aware of some general considerations about the infrastructure for protected volume encryption, about the hardware and software prerequisites and how to set up and configure all required components.

Prerequisites

Here is a list of hardware and software components that are required for configuring the infrastructure for protected volume encryption.

Hardware prerequisites

- An IBM z14[®], z13[®], z13s[®], or any IBM LinuxONE machine with the CPACF feature installed. CPACF requires specific microcode to be loaded which you can order as no-charge feature code (LIC #3863).
- For redundancy, two or more IBM Crypto Express5S or Crypto Express6S cryptographic coprocessors configured in CCA coprocessor mode or two or more Crypto Express7S coprocessors configured in EP11 mode.
- A Trusted Key Entry (TKE) workstation. For CCA coprocessors you need to install the CSUTKEcat TKE daemon from the CCA library to handle administrative commands between the TKE and the cryptographic coprocessors. For non-production CCA environments you can use the utilities from the CCA package instead of the TKE to perform operations on cryptographic coprocessors.
- For EP11 coprocessors, you need to install the EP11 TKE daemon (ep11TKEd) from the EP11 host library version 3.0.0 or later, which is listening on port 50004 for administrative TKE commands.

Download the required libraries from:

http://www.ibm.com/security/cryptocards

• Volumes to be encrypted (for example, SCSI or DASD volumes). For DASD volumes, you can only encrypt partitions, not the complete DASD.

Generally speaking, any block device can be encrypted using the infrastructure for protected volume encryption.

Software prerequisites

• Any Linux distribution that includes the pkey and paes_s390 kernel modules and a dm-crypt version that supports LUKS2. Linux kernel upstream version 4.12 or later includes the required support for secure keys of type CCA-AESDATA. There might be distributions that have older kernel versions where the required modules have been back-ported.

Support of secure keys of type CCA-AESCIPHER requires Linux kernel upstream version 5.4 (or older versions where the required modules have been back-ported).

Support of secure keys of type EP11-AES requires Linux kernel upstream version 5.6 (or older versions where the required modules have been back-ported).

The pkey kernel module requires permission for the AES key import functions. To grant this permission, go to the security settings of the applicable LPAR on the *Hardware Management Console* (HMC). In the CPACF Key Management Operations section, select the Permit AES Key import functions option. For z/VM guests, the LPAR in which the hypervisor runs requires this option.

Note: This action is only required for distributions that contain Linux kernels prior to upstream version 4.20. Also, the appropriate updates have been back-ported to several distributions, so that you do no longer need to manually apply the described permission for these distributions.

• The **cryptsetup** utility version 2.0.3 or later is required to configure an encrypted volume.

• The **zkey** utility from the <u>s390-tools</u> package (upstream version 2.6.0 or later). Use this utility to generate and manage secure keys.

Support of secure keys of type CCA-AESCIPHER requires s390-tools package upstream version 2.12.

Support of secure keys of type EP11-AES requires s390-tools package upstream version 2.13.

Note: s390-tools versions might differ among various distributions, because the **zkey** utility might have been back-ported to earlier versions.

- The **zkey-cryptsetup** utility from the <u>s390-tools package</u> (upstream version 2.6.0 or later, also see previous note). Use this utility to support an AES master key change in order to avoid loss of data on volumes encrypted using the PAES cipher.
- The CCA 6.0 package or later from the <u>software-package selection page</u> is required to support secure keys of type CCA-AESCIPHER and CCA-AESDATA. It is also required to connect the TKE workstation to cryptographic coprocessors configured in CCA mode.

For the support of secure keys of type EP11-AES, an IBM 4769 Crypto Express7 adapter configured in EP11 mode is required (CEX7P) and the EP11 host library version 3.0.0 or later must be installed. This library is also required for using the TKE to manage cryptographic coprocessors configured in EP11 mode.

Assumptions

L

In this documentation, it is assumed that the following setup has been installed:

- A Linux instance is installed as a z/VM or KVM guest virtual machine or in LPAR mode. Linux as a KVM guest can access cryptographic coprocessors starting with the following distributions:
 - Red Hat Enterprise Linux 8.0
 - SUSE Linux Enterprise Server 15 SP1
 - Ubuntu 18.04 LTS
- Two cryptographic coprocessor domains are configured to the LPAR or as dedicated adapters to the z/VM or KVM guest virtual machine.
 - The two domains have the same domain ID and are located on two distinct cryptographic coprocessors.
 - The same AES master key is set on the domains of both cryptographic coprocessors. For information
 on how to set a CCA master key, refer to *How to set an AES master key*. For setting an EP11 master
 key, refer to *Exploiting Enterprise PKCS #11 using openCryptoki*.

For comprehensive information about the TKE refer to:

TKE 9.1 PDF or TKE 9.1 Resource link

• The volumes to be encrypted are configured to be persistently available to the Linux instance.



Figure 4. System configuration for protected volume encryption

Planning

Before you start to configure and set up your infrastructure for protected volume encryption, you need to decide a series of aspects such as the type of the used devices, the number of required cryptographic coprocessors, or the way you want to manage the applied secure keys.

Device considerations

For Linux on Z and LinuxONE, you can apply the infrastructure for protected volume encryption on block devices (for example, on DASD devices or on SCSI-over-FCP attached devices).

For DASDs, use partitions for disk encryption only. For volumes attached through SCSI-over-FCP, you can either use partitions or the full block device.

For more information on SCSI devices, read How to use FC-attached SCSI devices with Linux on System z.

Cryptographic coprocessor considerations

Encrypting volumes with a secure key requires that the Linux instances have access to domains of a cryptographic coprocessor configured in CCA or EP11 coprocessor mode, depending on which key types you want to use. These domains contain the AES master key.

Whenever the master key or the coprocessor is changed, then appropriate actions must be taken to retain access to the data. Once the master key is lost, the data on the volumes cannot be recovered anymore.

Use standard procedures to set your AES master keys through the Trusted Key Entry workstation. Your master key should be stored in multiple parts on a set of smart cards. Preserve these smart cards in safe places. In case of a broken cryptographic coprocessor or other disaster, you can use these smart cards to configure a domain on another coprocessor with that master key.

Also, to safeguard against the loss of a master key during the operation of an operating system, consider keeping the same master key on the domains of two or more separate cryptographic coprocessors accessible by that system.

In addition, there might be circumstances where you need to access volumes on different Linux instances attached to different cryptographic coprocessors, being encrypted with the same secure key. In such cases, use the Trusted Key Entry workstation to set the same master key on the same domains on two or more separate cryptographic coprocessors.

Important: If the master key used to create the secure key is lost, you can recover encrypted data only if the clear key, which you used to initially create the secure key, is still available. This is often not the case, especially when you generated your secure key from a random key on the cryptographic coprocessor. Therefore, generate the master key parts on the Trusted Key Entry workstation and store them safely on separate smart cards. Without a master key, you could decrypt your data only if the clear key would still be available.

Secure key considerations

Creating and managing secure keys depends on your security policies. You can use the **zkey** command to generate a secure key and store it in a secure key repository or in a specified binary secure key file.

As a prerequisite, the pkey kernel module must be loaded, and an AES master key must exist on a domain of the cryptographic coprocessor. Refer to <u>How to set an AES master key</u> for information about setting a CCA master key. For setting an EP11 master key, refer to <u>Exploiting Enterprise PKCS #11 using</u> openCryptoki.

For comprehensive information about the TKE refer to:

The **zkey** command issues a request to a domain of a cryptographic coprocessor to generate a secure key by wrapping a randomly generated plain text key with an existing master key. Or, you can pass a binary input file containing a clear key to the **zkey** command for secure key generation.

Using the --key-type parameter of the **zkey** command, you can decide between various types of secure keys: a CCA AES DATA key, a CCA AES CIPHER key, or an EP11 AES key for use with pervasive volume encryption. An AES DATA secure key is generated as the default, if you omit this parameter.

With an AES CIPHER secure key, certain attributes are cryptographically bound to the key. These attributes may limit the usage of the key, for example, restrict the export or the usability scope. So this key type is assumed to be even more secure than the default AES DATA key. To generate an AES CIPHER secure key, a CEX6C or later coprocessor is required. If more than one domain or cryptographic coprocessors are used, the master key setup needs to be identical for each APQN.

If your installation uses cryptographic coprocessors configured in EP11 mode, you can work with EP11 AES secure keys starting with CEX7P adapters (see the following note).

Note: The support for CCA AES CIPHER keys and for EP11 AES secure keys depends on the version of your distribution. The support of EP11 secure key starts with IBM 4769 Crypto Express7 features on IBM z15 configured in EP11 mode (CEX7P). Furthermore EP11 firmware and CEC millicode support which is available with IBM z15, and bundle S14 is required.

The different secure key types have different sizes (in bytes):

- CCA AES DATA: 64, 128 for XTS
- CCA AES CIPHER: 136, 272 for XTS
- EP11 AES: 320, 640 for XTS

In various commands and command outputs, the key size is indicated in bits.

- Support for AES CIPHER keys starts with:
 - Red Hat Enterprise Linux 8.2
 - SUSE Linux Enterprise Server 15 SP2
 - Ubuntu 20.04

- At the time of writing, support for EP11 AES secure keys starts with:
 - Ubuntu 20.04

Do not explicitly generate a clear key into a binary file to subsequently transform it into a secure key, unless you can perform this operation in a safe clean room environment. Otherwise you risk being observed during the clear key generation, or some software components still contain some remains of the generated clear key.

When providing the clear key in an input file, this file should be kept at a secure place (clean room), or it should be securely erased after creation of the secure key, for example, with the **shred** command. The secure key itself does not need to be kept securely, because it can only be processed within a cryptographic coprocessor that contains the adequate AES master key.

The clear keys themselves are never stored persistently on the cryptographic coprocessor, but can only be reconstructed by decrypting the secure key with the pertaining master key within the cryptographic coprocessor.

For safety and security reasons, protect the secure key file as described in <u>"Managing a secure key</u> repository" on page 35.

If you format a volume with LUKS2 and a secure key, the secure key is encrypted and stored in a key slot of the LUKS2 header. During secure key generation, you can decide to additionally store the secure key in a secure key repository for archive purposes. For more information, see <u>"Managing a secure key</u> repository" on page 35.

For more information about the **zkey** command, read <u>Appendix A</u>, "zkey - Managing secure keys," on page 71 and also see the **zkey** man page.

Cipher mode considerations

The cryptographic algorithm used by the infrastructure for protected volume encryption is called PAES cipher and is implemented by the paes_s390 kernel module. When you format a volume using LUKS2, you need to specify this PAES cipher together with a block cipher mode of your choice.

The PAES cipher supports the following block cipher modes:

xts

XTS: XEX-based tweaked-codebook mode with ciphertext stealing.

cbc

CBC: Cipher Block Chaining

ctr

CTR: Counter mode

ecb

ECB: Electronic Codebook

Important: In the infrastructure for protected volume encryption, only use the XTS cipher mode. This XEX-based tweaked-codebook mode with ciphertext stealing (XTS) is the block cipher mode recommended by the NIST to encrypt data at-rest. In this mode, the plaintext blocks are XOR-ed with the previous ciphertext block before being encrypted by the block cipher. And in contrast to CBC, it is not vulnerable against code injection attacks.

Restriction: The ESSIV (encrypted sector salt initial vector) mode to generate initialization vectors cannot be used with **cryptsetup** and the PAES cipher.

Loading required modules and components

The Linux kernel must include support for the protected AES cipher (paes_s390) which automatically includes the pkey module.

Normally, the paes_s390 module is automatically loaded, if required. In this case, the pkey module is also automatically loaded.

If errors occur nevertheless, first check whether the required module is already loaded:

grep <module_name> /proc/modules

If required, you can load the paes_s390 module or the pkey module with the **modprobe** command. There are no module parameters.

modprobe <module_name>

Chapter 4. Sample system for data volume encryption

A sample system configuration is used to illustrate most of the tasks documented in this publication.

Hardware components

I

- Two CEX6C cryptographic coprocessors
 - Up to 10 physical SCSI volumes with a capacity of 20 GB each

Alias names disk1, disk2, disk3, ... are assigned to these volumes

- A Trusted Key Entry workstation (TKE) to maintain the cryptographic coprocessors
- A Central Processor Assist for Cryptographic Functions (CPACF) is configured and enabled
- A CPC containing CPs (general purpose processores) or IFLs

Software components

The sample system is running with the following software:

- A Linux kernel with:
 - all required modules loaded: paes_s390 and pkey
 - Logical Volume Manager
 - device mapper
 - dm-crypt
 - cryptographic device driver (zcrypt)
- s390-tools: zkey and zkey-cryptsetup
- cryptsetup version 2.0.3

Configuration

The used volumes are configured as SCSI volumes in a multipath setup, using two FCP adapters. For more information about SCSI devices, read How to use FC-attached SCSI devices with Linux on z Systems.

It is assumed that the multipath SCSI volume configuration is transparent to users of the infrastructure for protected volume encryption.

SCSI multipath volumes appear under /dev/mapper/. For ease of use, readable alias names like disk1, disk2, and so on are applied.

The sample system configuration exploits the Logical Volume Manager (LVM). Alternatively, you can set up the infrastructure for protected volume encryption without LVM. In this case, you need to adapt the described procedures accordingly.

An identical AES master key (AES MK) is set on the first and second cryptographic coprocessor (CEX6C I and CEX6C II in Figure 5 on page 16) in the same domain on both.

Figure 5 on page 16 shows the configuration of the sample system after you finished the tasks described in "Creating a volume for pervasive encryption" on page 17.



Figure 5. Sample system configuration

In the logical volume group MY_VOLGROUP, intentionally only 35 GB are allocated instead of the maximum of 40 GB to logical volume LV3. The LUKS2 headers occupy some space that is not available as volume capacity. Also, you can keep some reserve space for defining a small new logical volume LV4 for future use, or to enlarge an existing logical volume (for example, LV1).

Chapter 5. Working with encrypted volumes

Learn how to configure an encrypted volume (or a disk or partition). Content that you write to such an adequately configured volume is transparently encrypted or decrypted depending on the I/O request.

Note: If you use an LVM configuration, apply the encryption on the physical volume level. That is, encrypt /dev/mapper/disk<n> and then add the encrypted volume to the LVM volume group.

Remember: In the context of this document, the term *volume* also stands for an entire disk, or a partition on a disk, or for any other block device in Linux on Z and LinuxONE.

The documented tasks and scenarios in this publication focus on volumes formatted with LUKS2. A procedure how to work with volumes in *plain mode* is presented in <u>Chapter 9</u>, "Encrypting volumes without LUKS," on page 63.

The following topics provide further details:

- "Creating a volume for pervasive encryption" on page 17
- "Opening an encrypted volume" on page 22
- "Encrypting an unencrypted volume with a secure key" on page 25
- "Re-encrypting a volume from clear key to secure key" on page 31

Creating a volume for pervasive encryption

With **cryptsetup**, you can conveniently set up volume encryption. It formats the volume and performs the necessary device mapper setup tasks.

Before you begin

As a prerequisite, you require free disk devices or partitions on disk devices that are configured to be persistently available to your Linux instance.

About this task

To create a volume that will receive encrypted content, you need to know the encryption method for this volume, the wanted key size, and the name and location of the volume. This information is passed to **cryptsetup** as input parameters.

Also, you must define a passphrase for each volume that you want to encrypt. This passphrase encrypts the LUKS volume encryption key (LVEK) and you can use it for all interactive setup actions on the encrypted volume. In the infrastructure for protected volume encryption, the LVEK is a secure key. Because secure keys are already encrypted by a master key, these passphrases are of limited relevance to security.

For automated opening of the encrypted volume during system startup, you can use a key file containing some random data that fulfills the same purpose as a passphrase.

Procedure

1. Generate a secure key in the secure key repository using the **zkey** utility.

You can generate secure keys (AES DATA keys, AES CIPHER keys as shown in the example, or EP11 AES secure keys) with a length of 128, 192, or 256 bits. When generating secure keys for the XTS cipher mode, only 128 or 256 bit keys are supported. For XTS cipher mode keys, two secure key parts are generated and concatenated to each other.

For the scenario, issue the following command for the XTS cipher mode with a key name of secure_xtskey1 and a key length of 256 bits. Create four different keys with names secure_xtskey1 up to secure_xtskey4.

```
# zkey generate --name secure_xtskey1 --key-type CCA-AESCIPHER --keybits 256 --xts \
--volumes /dev/mapper/disk1:enc-disk1 --volume-type LUKS2 \
--apqns 03.0039,04.0039 --sector-size 4096
...
# zkey generate --name secure_xtskey4 --key-type CCA-AESCIPHER --keybits 256 --xts \
--volumes /dev/mapper/disk4:enc-disk4 --volume-type LUKS2 \
--apqns 03.0039,04.0039 --sector-size 4096
```

The keys to be encrypted by the AES master key are generated by random inside a domain of the cryptographic coprocessor and are thus never exposed in clear outside the coprocessor. After the successful generation of the secure key, the randomly generated key is destroyed by the coprocessor. This is possible because during cryptographic operations within the coprocessor, effective keys are always reconstructed by decrypting the secure key with the master key.

In the shown example, you specify the volumes that are to be encrypted with the new secure key, and you also associate the APQNs where the related master key is located.

Use a different key for each encrypted volume. Only for certain reasons, if you want to use a particular secure key for selected volumes, generate only one key and associate it to all these selected volumes.

Note: With the optional parameter **--sector-size**, you can set the size of the blocks to be encrypted or decrypted. It must be a power of two and in the range 512 - 4096 bytes. **cryptsetup** chooses a default sector size of 512 bytes. Using 4096-byte sectors provides the best performance on Linux on Z and LinuxONE.

zkey list
Key : secure_xtskey1
Description :
Secure key size : 272 bytes
Clear key size : 512 bits
XTS type key : Yes
Key type : CCA-AESCIPHER
Volumes : /dev/mapper/disk1:enc-disk1
APQNs : 03.0039
04.0039
Key file name : /etc/zkey/repository/secure_xtskey1.skey
Sector size : 4096 bytes
Volume type : LUKS2
Verification pattern : 303344b12b8258840fa11852a4ecc6d5
84c7a867f893a5dcc0d499557c45bee6
Created : 2020-08-20 15:27:20
Changed : (never)
Re-enciphered : (never)

You can list information about the generated secure keys:

2. Generate the commands to format the volume to be encrypted using **cryptsetup**. The formatting process writes the LUKS2 header onto the volume. During formatting, the LUKS2 volume encryption key, which is the previously generated secure key, is encrypted with a KEK that is derived from the passphrase or from a provided key file.

When you format a volume, you need to specify values for the following parameters, which are retrieved or calculated during command generation:

type

L

L

L

I

The LUKS2 type of formatting.

cipher

The PAES cipher, its operation mode (XTS), and the algorithm to generate the initialization vector (plain64).

master-key-file

The location and name of the secure key file generated in step "1" on page 17.

key-size

I

I

The size of the secure key file in bits. In our example, an AES CIPHER key used with the XTS cipher mode is generated with a size of 272 bytes = 2176 bits. (An AES DATA key has 64 bytes = 512 bits. An EP11 AES key used with XTS of size 640 bytes has a file size of 5120 bits).

In addition, you specify the name of the volume. In our case, this is /dev/mapper/disk1 through /dev/mapper/disk4.

For the sample system, invoke the **zkey cryptsetup** command for the first volume as follows:

zkey cryptsetup --volumes /dev/mapper/disk1 cryptsetup luksFormat --type luks2 --pbkdf pbkdf2 --master-key-file '/etc/zkey/repository/secure_xtskey1.skey' --key-size 2176 --cipher paes-xts-plain64 --sector-size 4096 /dev/mapper/disk1 zkey-cryptsetup setvp /dev/mapper/disk1

Two commands are generated and displayed as shown:

- the cryptsetup luksFormat command to format the volume
- the **zkey-cryptsetup setvp** command to set the verification pattern into the LUKS2 header. The verification pattern is used to identify the valid effective key during recovery actions.
- 3. Run the generated commands.

Copy and paste the generated commands into the command line.

Note: If the you use an older distribution, the --pbkdf pbkdf2 option may be missing in the generated command. In such a case, edit the command by adding option --pbkdf pbkdf2. This is to avoid an out-of-memory error for a LUKS2 volume during automated opening via /etc/crypttab at system startup. For more information, see <u>"Out-of-memory errors when opening a LUKS2 volume"</u> on page 58.

For each volume, define a passphrase like disk1pw up to disk4pw.

```
# cryptsetup luksFormat --type luks2 --pbkdf pbkdf2 \
--master-key-file '/etc/zkey/repository/secure_xtskey1.skey' \
--key-size 2176 --cipher paes-xts-plain64 --sector-size 4096 /dev/mapper/disk1
WARNING!
=======
This will overwrite data on /dev/mapper/disk1 irrevocably.
Are you sure? (Type uppercase yes): YES
Enter passphrase for /dev/mapper/disk1: disk1pw
Verify passphrase: disk1pw
# zkey-cryptsetup setvp /dev/mapper/disk1
Enter passphrase for '/dev/mapper/disk1': disk1pw
```

When you type the passphrase, it is not displayed.

Tip: In newer distributions where the --pbkdf pbkdf2 option is automatically generated, you can use the --run option to let the generated commands run directly after creation, without the need to copy and paste:

zkey cryptsetup --volumes /dev/mapper/disk1 --run

4. Open the volume using a generated command:

```
# zkey cryptsetup --volumes /dev/mapper/disk1 --open --run
```

The following command is generated and immediately run:

cryptsetup luksOpen /dev/mapper/disk1 enc-disk1

When prompted, enter the passphrase from step "3" on page 19.

This creates a device-mapper device named /dev/mapper/enc-disk1 (see step "5" on page 20).

Note: Repeat steps <u>1</u> through <u>"4" on page 19</u> with the appropriate file names and volume names for each volume that you want to encrypt.

5. Optional: Check the result of step <u>"4" on page 19</u> with the command **ls** /dev/mapper/:

ls /dev/mapper control disk2 disk4 disk6 disk8 enc-disk2 enc-disk4 disk1 disk3 disk5 disk7 enc-disk1 enc-disk3

As of now, any I/O operation to or from /dev/mapper/enc-disk1 is transparently encrypted or decrypted onto the /dev/mapper/disk1 device. Do not write to this device directly. The same is valid for disks disk2 through disk4.

- 6. Create a key file and specify it in an entry in /etc/crypttab to persistently configure the opening at system startup.
 - a. Create the key file named disk1.key in directory /etc/luks_keys/ with random content with a size of 4096 bytes (specified by bs=1024 count=4). If the directory does not yet exist, you must create it first.

```
# dd if=/dev/urandom of=/etc/luks_keys/disk1.key bs=1024 count=4
# chmod 0400 /etc/luks_keys/disk1.key
```

The second command makes the key file readable by the root user only. Repeat these commands for each volume adequately.

b. Add a key slot with the previously generated key file:

```
# cryptsetup luksAddKey --pbkdf pbkdf2 /dev/mapper/disk1 /etc/luks_keys/disk1.key
```

You must also specify option --pbkdf pbkdf2 with the **luksAddKey** command to avoid an outof-memory error for a LUKS2 volume during automated opening via /etc/crypttab at system startup. For more information, see <u>"Out-of-memory errors when opening a LUKS2 volume" on</u> page 58.

Repeat this for each disk. When prompted, enter the passphrase from step "3" on page 19.

c. Edit /etc/crypttab and add one entry for each encrypted volume:

```
# /etc/crypttab
∄
# See crypttab(5) for more information.
#
                                             <key file>
/etc/luks_keys/disk1.key
                                                                             <options>
# <target name> <source device>
  enc-disk1
               /dev/mapper/disk1
                                                                             luks
  enc-disk2
                  /dev/mapper/disk2
                                             /etc/luks_keys/disk2.key
                                                                             luks
                                             /etc/luks_keys/disk3.key
/etc/luks_keys/disk4.key
                  /dev/mapper/disk3
  enc-disk3
                                                                             luks
  enc-disk4
                  /dev/mapper/disk4
                                                                             luks
```

The format of the /etc/crypttab file depends on your Linux distribution. See the crypttab man page for more details.

Alternatively you can have a skeleton for a crypttab entry automatically generated by the **zkey crypttab** command:

```
# zkey crypttab --volumes /dev/mapper/disk<n>
```

You only need to add the **<key file>** column manually.

You can, however, specify the key file with the command, then you do not need to add the **key** file> column manually:

```
# zkey crypttab --volumes /dev/mapper/disk<n> --key-file /etc/luks_keys/disk<n>.key
```

- 7. Prepare your volumes for being managed by LVM. For this purpose, you need to perform the following sub-steps:
 - a. Define and initialize the physical volumes that you plan to use in order to make them available to LVM. Use the **pvcreate** command. Each such command creates what is called an LVM physical volume.
 - b. Define an LVM volume group where you combine the LVM physical volumes that you want to manage within such a group. Use the vgcreate command.
 - c. Add logical volumes to the volume group using the lvcreate command. With the -n option, you specify a meaningful name for the logical volume. With the -L option, you specify a size, which can be independent from the sizes of any existing hardware disks, except for the overall capacity.

Example:

L

```
# pvcreate /dev/mapper/enc-disk1
# pvcreate /dev/mapper/enc-disk2
# pvcreate /dev/mapper/enc-disk3
# pvcreate /dev/mapper/enc-disk4
# vgcreate MY VOLGROUP /dev/mapper/enc-disk*
# lvcreate -L 10GB MY_VOLGROUP -n LV1
# lvcreate -L 30GB MY_VOLGROUP -n LV2
# lvcreate -L 35GB MY_VOLGROUP -n LV3
# ls /dev/mapper
control disk3 disk6 enc-disk1 enc-disk4
disk1 disk4 disk7 enc-disk2 MY_VOLGROUP-LV1
disk2 disk5 disk8 enc-disk3 MY_VOLGROUP-LV2
                                                                              MY VOLGROUP-LV3
```

Note: For logical volume LV3, only 35 GB are defined, though 40 GB physical space are available. Thus you can enlarge the volume for further purposes.

8. Create a file system - in the example, an ext4 file system - on the encrypted LVM logical volume created in step "7" on page 21 or the encrypted volume created in step "4" on page 19 (if you did not use LVM).

mkfs.ext4 /dev/mapper/MY_VOLGROUP-LV1
mkfs.ext4 /dev/mapper/MY_VOLGROUP-LV2
mkfs.ext4 /dev/mapper/MY_VOLGROUP-LV3

The kernel transparently encrypts or decrypts I/O requests to or from the encrypted volume. Thus, the end-to-end encryption for the data at-rest is implemented.

- 9. Create mount points and update /etc/fstab to later mount the file systems on the encrypted volumes.
 - a. Create the mount points:

```
# mkdir /crypted_lv1
# mkdir /crypted_lv2
# mkdir /crypted_lv3
```

b. Edit file /etc/fstab and add similar entries like follows:

```
/dev/mapper/MY_VOLGROUP-LV1 /crypted_lv1 ext4 defaults 0 0
/dev/mapper/MY_VOLGROUP-LV2 /crypted_lv2 ext4 defaults 0 0
/dev/mapper/MY_VOLGROUP-LV3 /crypted_lv3 ext4 defaults 0 0
```

10. Mount the file system through /etc/fstab.

```
# mount /crypted_lv1
# mount /crypted_lv2
# mount /crypted_lv3
```

Note: To let users issue the **mount** command for a particular mount point, add the user option to the entry for this mount point in /etc/fstab.

Results

You created four encrypted volumes as LVM physical volumes grouped into three LVM logical volumes. Each of these LVM logical volumes contains an empty file system ready to accept files. All content you save onto these LVM logical volumes is automatically encrypted.

The result of this task is illustrated in Figure 5 on page 16.

What to do next

- Create permissions for users to access data on the mounted file system.
- Now a user can start to read and write data on the mounted file system, which is transparently encrypted or decrypted.

For example, issue:

```
$ echo 'what is secret' > /crypted_lv1/mysecret
$ ls /crypted_lv1
$ cat /crypted_lv1/mysecret
```

Note: You might encounter an out-of-memory error when opening a LUKS2 volume either during manual opening or during automated opening via /etc/crypttab at system startup. In such a case, read <u>"Out-of-memory errors when opening a LUKS2 volume" on page 58 for problem resolution information.</u>

Opening an encrypted volume

The need to open an encrypted volume can occur during normal runtime, or during Linux startup. Special processing is necessary for opening an encrypted volume at early startup time (for example, if it is part of an LVM volume group on which the root file system resides).

To open an encrypted volume during runtime, it is sufficient to perform step <u>"4" on page 19</u> of <u>"Creating a</u> volume for pervasive encryption" on page 17.

This topic presents two use cases:

- 1. "Automatically opening encrypted volumes at Linux startup" on page 22
- 2. "Opening and mounting an encrypted volume at user login" on page 23

Automatically opening encrypted volumes at Linux startup

Automatically opening one or more volumes at Linux startup allows you to perform automated reboots. For an automatic unattended startup, a key file is required. For an automatic attended startup, you can also use passphrases.

Before you begin

Determine the encrypted volumes that are required during the Linux startup.

Procedure

1. For each encrypted volume that is required during the Linux startup, you need to edit /etc/ crypttab. Add an entry for each required volume. For examples of valid /etc/crypttab entries, read step <u>"6" on page 20</u> in <u>"Creating a volume for</u> pervasive encryption" on page 17.

 If one or more of the encrypted volumes are required to mount the root file system, ensure that the /etc/crypttab and the referenced secure key files are included in the initial RAM disk. See your Linux distribution documentation for including additional files into the RAM disk and re-creating the RAM disk.

Opening and mounting an encrypted volume at user login

Automatically opening one or more volumes at user login has the advantage that only a certain user can access the data.

Before you begin

Important: At the time of writing, LUKS2 support for *pam_mount* is provided by the following distributions:

- Ubuntu 19.04
- SUSE Linux Enterprise Server 12 SP5
- SUSE Linux Enterprise Server 15 SP1

Without LUKS2 support, the functionality described in this topic does not work.

You must install the *pam_mount* package. See the web site at <u>http://pam-mount.sourceforge.net/</u>. Some Linux distributions provide the *pam_mount* package.

Ensure that the pam_mount package is configured and the pam_mount.so PAM module is used in the auth and session sections of the PAM configuration files. Your Linux distribution might already perform this for you. See also the pam_mount man page for more information.

About this task

In the scenario you create a user called alice. The home directory for this user is stored on an encrypted volume. The encrypted volume is opened when the user logs in and respectively closed at logout. The encrypted volume in this example is /dev/mapper/disk10.

Procedure

1. Create a user and set an initial password.

For example, issue:

```
# useradd -G users -m -s /bin/bash alice
# passwd alice
Enter new UNIX password: alice
Retype new UNIX password: alice
passwd: password updated successfully
```

2. Create a secure key in the secure key repository for user alice with the **zkey** command.

I.

```
# zkey generate --name user-alice-xts --key-type CCA-AESCIPHER --keybits 256 --xts \
--volumes /dev/mapper/disk10:enc-disk10 --volume-type LUKS2 \
--apqns 03.0039,04.0039 --sector-size 4096
```

Format and open the encrypted volume /dev/mapper/disk10 and create a file system that is later mounted as home directory for user alice.

For example:

```
# zkey cryptsetup --volumes /dev/mapper/disk10 --run
Executing: cryptsetup luksFormat --type luks2 --pbkdf pbkdf2
--master-key-file '/etc/zkey/repository/user-alice-xts.skey'
--key-size 2176 --cipher paes-xts-plain64 --sector-size 4096 /dev/mapper/disk10
WARNING!
==========
This will overwrite data on /dev/mapper/disk10 irrevocably.
Are you sure? (Type uppercase yes): YES
Enter passphrase for /dev/mapper/disk10: disk10pw
Verify passphrase: disk10pw
Executing: zkey-cryptsetup setvp /dev/mapper/disk10
Enter passphrase for '/dev/mapper/disk10': disk10pw
# cryptsetup luksOpen /dev/mapper/disk10 user-enc-alice
# mkfs.ext4 -L USER_ALICE /dev/mapper/user-enc-alice
# cryptsetup luksClose user-enc-alice
```

If you plan to open the volume automatically during system startup, ensure that the --pbkdf pbkdf2 option is generated for the **cryptsetup luksFormat** command. Otherwise, insert this option manually before running the command. Also use the --pbkdf pbkdf2 option with the **cryptsetup luksAddKey** command in the alternatives described in steps <u>"4" on page 24</u> and <u>"5" on page 24</u>. This is to avoid an out-of-memory error for a LUKS2 volume during automated opening via /etc/crypttab at system startup. For more information, read <u>"Out-of-memory errors when opening a LUKS2 volume" on page 58</u>.

You can optionally mount the file system temporarily to copy or migrate existing files for the user.

4. Perform Alternative 1:

L

L

Disk passphrase and user password can be different. Therefore, you must provide the disk passphrase in a key file. The pam_mount module retrieves it from there. The advantage is that the user can change his passphrase anytime without influencing the disk passphrase. However, the disk passphrase is available in clear text (in our example, in file /etc/pam_mount_keys/alice.key). You might have to create this /etc/pam_mount_keys/ directory in advance. Ensure to establish the correct access rights.

a) Create a file containing the disk passphrase.

This file should contain a disk passphrase or a random character sequence. Example: /etc/pam_mount_keys/alice.key

b) Add the disk passphrase to a new LUKS2 key slot on your volume (see also step <u>"6" on page 20</u> in <u>"Creating a volume for pervasive encryption" on page 17</u>).

```
# cryptsetup luksAddKey --pbkdf pbkdf2 /dev/mapper/disk10 /etc/pam_mount_keys/alice.key
```

c) Edit the pam_mount configuration file /etc/security/pam_mount.conf.xml. Add a volume definition for alice.

```
<volume user="alice" path="/dev/mapper/disk10" mountpoint="~"
fstype="crypt" fskeycipher="none"
fskeypath="/etc/pam_mount_keys/alice.key" />
```

See also the *pam_mount.conf* man page for details.

5. Perform Alternative 2:

Disk passphrase is the same as the user password. In this case, you add an additional key-slot to the LUKS header on your volume. The advantage is the enhanced security: the disk passphrase is nowhere available in clear text.

Note: When the user password changes, the passphrase for the encrypted volume must also be changed.

a) Add the disk passphrase to a new LUKS2 key slot on your volume.

You are prompted for the phassphrase that you have specified when formatting the volume in step "3" on page 23. In this scenario, it is disk10pw.

```
# cryptsetup luksAddKey --pbkdf pbkdf2 /dev/mapper/disk10
Enter any existing passphrase: disk10pw
Enter new passphrase for key slot: alice
Verify passphrase: alice
```

b) Edit the pam_mount configuration file /etc/security/pam_mount.conf.xml.

```
<volume user="alice" path="/dev/mapper/disk10" mountpoint="~"
fstype="crypt" />
```

Results

I.

Now alice can log in to the Linux instance. The pam_mount PAM module opens the encrypted volume and creates a device in the /dev/mapper/ directory, for example, /dev/mapper/_dev_dm_21 which is then mounted as /home/alice.

```
# ssh alice@localhost
alice@localhost's password: alice
Welcome to your favourite Linux distribution
Last login: Mon Aug 06 16:28:45 2018 from 127.0.0.1
alice@localhost:~$ df | grep alice
/dev/mapper/_dev_dm_21 20507216 45080 19397384 1% /home/alice
alice@localhost:~$
```

Encrypting an unencrypted volume with a secure key

If you want to integrate unencrypted data residing on a volume into the infrastructure for protected volume encryption, you need to perform the task to transform an unencrypted volume into an encrypted one.

This topic presents two methods with which you can achieve this task:

- 1. For LVM physical volumes, you can use the **pvmove** command (see <u>"Migrating to an encrypted LVM</u> physical volume" on page 25).
- 2. You can copy existing content to a new encrypted volume and delete the original volume (see "Migrating data to a new encrypted volume" on page 31).

After you have migrated the data from the unencrypted volumes to the encrypted ones, be sure to securely delete any unencrypted data according to your security policies. For example, you can use **badblocks** or **shred** to overwrite unencrypted data with random data multiple times.

Migrating to an encrypted LVM physical volume

Read about a method for integrating unencrypted data into the infrastructure for protected volume encryption for an environment which uses LVM for volume management.

Before you begin

Prerequisite is an existing LVM volume group called MIGR_VOLGROUP as shown in Figure 6 on page 27. This volume group at first comprises disk5 and disk6 with a capacity of 20 GB each, which are the unencrypted LVM physical volumes. The total available capacity of MIGR_VOLGROUP therefore comprises 40 GB. The space available on the new encrypted disks enc_disk7 and enc_disk8 is somewhat less than 40 GB, because the LUKS2 header occupies some physical space on them, which is not accessible from LV1 and LV2. Therefore, the total space of the LVM logical volumes LV1 and LV2 must be less than 40 GB. In our example, only 35 GB are allocated for LV1 and LV2, which leaves 5 GB free space. This is of course, much more than the LUKS2 headers will occupy.

Note: Check the physical block sizes of your source physical volumes. If the block sizes are smaller than the block sizes of your target physical volumes, the file systems on the logical volumes that reside on the source physical volumes might get corrupted during migration and your data might be lost. This is, because the file systems of the logical volumes are aligned to the block size of the source physical volumes. When moved to a target physical volume with a larger block size, the alignment is now incorrect and the data might become corrupt.

New LVM versions perform this check and reject actions that may corrupt the data.

The **--sector-size** parameter of a dm-crypt volume influences the physical block size. A dm-crypt volume encrypted with a sector size larger than the default 512 bytes results in a device with a physical block size of either the used sector size or the block size of the underlying device, whatever is higher. Therefore, take care to format your target volume with a sector size equal to the block size of your source volume.

To query the physical block size of a device, use the following command:

blockdev --getpbsz <device>

I

For information about valid block size combinations between volumes involved in the infrastructure for protected volume encryption refer to <u>"Valid physical block size combinations of LVM physical volumes"</u> on page 56.

Use the **pvs**, **1vs**, and **cat** commands as shown to verify the existing sample set up:

pvs PV VG Fmt Attr PSize PFree /dev/mapper/disk5 MIGR_VOLGROUP lvm2 a-- <20.00g 0 /dev/mapper/disk6 MIGR_VOLGROUP lvm2 a-- <20.00g 4.99g # lvs LV VG Attr LSize Pool Origin Data% Meta% Move Log Cpy%Sync Convert LV1 MIGR_VOLGROUP -wi-a---- 20.00g LV2 MIGR_VOLGROUP -wi-a---- 15.00g # cat /etc/fstab /dev/mapper/MIGR_VOLGROUP-LV1 /migr_lv1 ext4 defaults 0 0 /dev/mapper/MIGR_VOLGROUP-LV2 /migr_lv2 ext4 defaults 0 0

About this task

In step <u>"7" on page 29 of this scenario you create two new encrypted LVM physical volumes enc_disk7</u> and enc_disk8 as a target for the encrypted data and integrate them into the MIGR_VOLGROUP LVM volume group. Then you migrate the unencrypted volumes disk5 and disk6 to the encrypted ones. Finally, you remove the unencrypted volumes from the volume group.



Figure 6. Migration scenario

Procedure

I

L

1. Create a new secure key in the secure key repository for encrypting volumes for use as LVM physical volumes in the volume group MIGR_VOLGROUP.

To do so, issue the following commands:

```
# zkey generate --name secure_xtskey7 --key-type CCA-AESCIPHER --keybits 256 --xts \
--volumes /dev/mapper/disk7:enc-disk7 --volume-type LUKS2 \
--apqns 03.0039,04.0039
# zkey generate --name secure_xtskey8 --key-type CCA-AESCIPHER --keybits 256 --xts \
--volumes /dev/mapper/disk8:enc-disk8 --volume-type LUKS2 \
--apqns 03.0039,04.0039
```

Note: In this procedure, it is assumed that the unencrypted source volumes have a physical block size of 512 bytes. Therefore, you should format the encrypted target volumes enc-disk7 and enc-disk8 with sector size 512, too, to ensure a successful migration. Only if you are sure that the unencrypted source volumes already have a physical block size of 4096 bytes (or more), you can add parameter

--sector-size 4096

to both shown zkey generate commands.

2. Generate the commands to format the volumes to be encrypted using **cryptsetup**. The formatting process writes the LUKS2 header onto the volumes.

Invoke the **zkey cryptsetup** command for the first volume as follows:

```
# zkey cryptsetup --volumes /dev/mapper/disk7
```

cryptsetup luksFormat --type luks2 --pbkdf pbkdf2 --master-key-file '/etc/zkey/repository/secure_xtskey7.skey' --key-size 2176 --cipher paes-xts-plain64 /dev/mapper/disk7

zkey-cryptsetup setvp /dev/mapper/disk7

Two commands are generated:

- the cryptsetup luksFormat command to format the volume
- the **zkey-cryptsetup** setvp command to set the verification pattern into the LUKS2 header. The verification pattern is used to identify the valid effective key during recovery actions.

3. Run the generated commands.

Copy and paste the generated commands into the command line. Alternatively, you can skip this step if you run the **zkey cryptsetup** command from step "2" on page 27 using the --run.

Note: If the you use an older distribution, the --pbkdf pbkdf2 option may be missing in the generated command. In such a case, edit the command by adding option --pbkdf pbkdf2. This is to avoid an out-of-memory error for a LUKS2 volume during automated opening via /etc/crypttab at system startup. For more information, see <u>"Out-of-memory errors when opening a LUKS2 volume" on page 58.</u>

cryptsetup luksFormat --type luks2 --pbkdf pbkdf2 \
--master-key-file '/etc/zkey/repository/secure_xtskey7.skey' \
--key-size 2176 --cipher paes-xts-plain64 /dev/mapper/disk7
WARNING!
=======
This will overwrite data on /dev/mapper/disk7 irrevocably.
Are you sure? (Type uppercase yes): YES
Enter passphrase for /dev/mapper/disk7: disk7pw
Verify passphrase: disk7pw
zkey-cryptsetup setvp /dev/mapper/disk7
Enter passphrase for '/dev/mapper/disk7

When you type the passphrases, these are not displayed.

Repeat this for disk8 accordingly.

4. Open the volumes:

cryptsetup luksOpen /dev/mapper/disk7 enc-disk7
cryptsetup luksOpen /dev/mapper/disk8 enc-disk8

Alternatively, you can use the **zkey cryptsetup** command with option - - open (and with option - - run) to generate (and run) the **luksOpen** commands:

zkey cryptsetup --volumes /dev/mapper/disk7 --open [--run]
zkey cryptsetup --volumes /dev/mapper/disk8 --open [--run]

- 5. Create a key file and specify it in an entry in /etc/crypttab to persistently configure the opening at system startup.
 - a. Create the key file named disk7.key in directory /etc/luks_keys/ with random content with a size of 4096 bytes, specified by bs=1024 count=4. If the directory does not yet exist, you must create it first.

```
# dd if=/dev/urandom of=/etc/luks_keys/disk7.key bs=1024 count=4
# chmod 0400 /etc/luks_keys/disk7.key
```

The second command makes the key file readable by the root user only. Repeat these commands for each volume appropriately.

b. Add a key slot with the previously generated key file.

cryptsetup luksAddKey --pbkdf pbkdf2 /dev/mapper/disk7 /etc/luks_keys/disk7.key

You must also specify option --pbkdf pbkdf2 with the **luksAddKey** command to avoid an outof-memory error for a LUKS2 volume during automated opening via /etc/crypttab at system startup. For more information, see <u>"Out-of-memory errors when opening a LUKS2 volume" on page</u> <u>58</u>.

c. Edit /etc/crypttab and add one entry for each encrypted volume:

```
# /etc/crypttab
#
# See crypttab(5) for more information.
#
# <target name> <source device>
enc-disk7 /dev/mapper/disk7 /
enc-disk8 /dev/mapper/disk8 //
```

<key file> /etc/luks_keys/disk7.key /etc/luks_keys/disk8.key <options> luks luks

The format of the /etc/crypttab file depends on your Linux distribution. See the crypttab man page for more details.

Alternatively you can have a skeleton for a crypttab entry automatically generated by the **zkey crypttab** command:

zkey crypttab --volumes /dev/mapper/disk<n>

You only need to add the **<key file>** column manually. You can, however, specify the key file with the command, then you do not need to add the **<key file>** column manually:

zkey crypttab --volumes /dev/mapper/disk<n> --key-file /etc/luks_keys/disk<n>.key

6. Create the target LVM physical volumes to which you want to migrate the encrypted data.

```
# pvcreate /dev/mapper/enc-disk7
# pvcreate /dev/mapper/enc-disk8
```

L

7. Extend the volume group MIGR_VOLGROUP by adding the two encrypted physical volumes. Display the result using the **pvs** command.

#	vgextend MIGR_VOLGROU	<pre>P /dev/mapper/e</pre>	enc-d:	isk7 ,	/dev/mapp	per/enc-disk8
#	pvs	NC	Et	۸ <u>ـــ</u> ــ		DEmo
	PV	VG	Fmτ	Αττι	PSize	PFIEE
	/dev/mapper/disk5	MIGR_VOLGROUP	lvm2	a	<20.00g	Θ
	/dev/mapper/disk6	MIGR_VOLGROUP	lvm2	a	<20.00g	4.99g
	/dev/mapper/enc-disk7	MIGR_VOLGROUP	lvm2	a	19.99g	19.99g
	/dev/mapper/enc-disk8	MIGR VOLGROUP	lvm2	a	19.99g	19.99g

The physical size (**PSize**) of 19.99 GB for enc-disk7 and enc-disk8 (instead of 20.00 GB) is due to the LUKS2 header information stored on these volumes.

8. Move the content from the unencrypted physical volumes disk5 and disk6 to the encrypted physical volumes enc-disk7 and enc-disk8 using the **pvmove** command.

This step requires several sub-steps. This is due to the fact that the encrypted physical volumes enc_disk7 and enc_disk8 are somewhat smaller than disk5 and disk6, because the LUKS2 header occupies a small amount of space that is not available to the encrypted physical volume. Thus, trying to move disk5 onto enc_disk7 would fail. Instead, use the approach described in the following sub-steps that allows LVM to move the data to any available space on the physical volumes belonging to the MIGR_VOLGROUP volume group.

Note: The described scenario only works if the total space occupied on the unencrypted physical volumes (disk5 and disk6) is less or equal to the space available on the encrypted physical volumes enc_disk7 and enc_disk8; that is, if the MIGR_VOLGROUP volume group uses less than two times 19.99 GB (= 39.98 GB). In this scenario, only 35 GB are used. If the volume group would occupy the full 40 GB, then another encrypted physical volume (for example, /dev/mapper/enc-disk9) would be required or disk7 and disk8 must be larger 20 GB.

a) Move the physical content from disk5 onto some other physical volume somewhere within the same LVM volume group.

Type the following **pvmove** command without specifying a target. If you verify the result with the **pvs** command, and compare it with the **pvs** output from step <u>"7" on page 29</u>, you see that in our scenario, the biggest portion of disk5 was moved to enc_disk7, which no longer has physical free space (**PFree = 0**). The rest of disk5 was moved to disk6, whose free space decreased from **PFree = 4.99g** to **PFree <4.99g**.

- b) Remove disk5 from the MIGR_VOLGROUP volume group.
 - # vgreduce MIGR_VOLGROUP /dev/mapper/disk5 Removed "/dev/mapper/disk5" from volume group "MIGR_VOLGROUP"
- c) Now move disk6 onto some other physical volume within MIGR_VOLGROUP and verify the result as shown in step "8.a" on page 29:

```
# pvmove /dev/mapper/disk6
/dev/mapper/disk6: Moved: 0.03%
/dev/mapper/disk6: Moved: 0.05%
...
/dev/mapper/disk6: Moved: 100.00%
```

d) Remove disk6 from the MIGR_VOLGROUP volume group:

```
vgreduce MIGR_VOLGROUP /dev/mapper/disk6
  Removed "/dev/mapper/disk6" from volume group "MIGR_VOLGROUP"
```

Verify the result with the **pvs** command to see that disk5 and disk6 are free and the encrypted volumes enc_disk7 and enc_disk8 are used (**PFree = 0 / PFree = 4.98g**

ŧF	pvs					
	PV	VG	Fmt	Attr	PSize	PFree
	/dev/mapper/disk5		lvm2		20.00g	20.00g
	/dev/mapper/disk6		lvm2		20.00g	20.00g
	/dev/mapper/enc-disk7	MIGR_VOLGROUP	lvm2	a	19.99g	0
	/dev/mapper/enc-disk8	MIGR_VOLGROUP	lvm2	a	19.99g	4.98g

Results

After migrating all unencrypted physical volumes to encrypted ones, you achieve a transparent encryption of all data in the MIGR_VOLGROUP LVM volume group. The **pvs** command from step <u>"8.d" on page 30</u> shows that the source LVM physical volumes disk5 and disk6 are now completely free (**20.00g** in both **PFree** and **PSize**), and the corresponding encrypted target LVM physical volumes enc-disk7 and enc-disk8 are respectively written with data.

What to do next

For security reasons you should next remove the unencrypted source volumes disk5 and disk6 from LVM:

pvremove /dev/mapper/disk5
pvremove /dev/mapper/disk6
Finally, you should securely delete all data from these unencrypted volumes.

Migrating data to a new encrypted volume

The method to integrate unencrypted data into the infrastructure for protected volume encryption described in this topic works in an environment without using LVM.

Before you begin

You require a free volume that has sufficient space. Ensure that this volume is persistently configured to your Linux instance.

Procedure

1. Create a new encrypted volume using the free volume.

Ensure that the newly encrypted volume is formatted with a file system. For this purpose, complete steps <u>"1" on page 17</u> through <u>"8" on page 21</u> from <u>"Creating a volume for pervasive encryption" on</u> page 17. If you do not want to use LVM, adapt the steps accordingly.

2. Mount the file system on the newly encrypted volume on a temporary mount point. For example:

mount /dev/mapper/new-enc-disk /mnt

3. Copy the contents (meaning files and directories) from the unencrypted volume to the newly created file system.

For example, to copy all data from /path/to-be-encrypted/data/ to the file system on the encrypted volume that is mounted on /mnt/, issue:

rsync -av /path/to-be-encrypted/data/ /mnt/

4. Replace the /etc/fstab entry of the existing unencrypted volume with the device-mapper device of the encrypted volume.

If the encrypted volume is required at startup time, ensure that an appropriate entry in /etc/ crypttab exists. You have already created this entry in step 3 in <u>"Creating a volume for pervasive encryption" on page 17</u>.

Results

The data is now on an encrypted volume in the infrastructure for protected volume encryption.

What to do next

You should now securely delete the unencrypted data according to your security policies from / path/to-be-encrypted/data/.

Re-encrypting a volume from clear key to secure key

You can re-encrypt a volume that had previously been encrypted with a clear key, with a new secure key.

This topic presents two use cases where a re-encryption with a secure key is desired for volumes that have been encrypted with a clear key only:

- You might have a volume that is encrypted with a clear key in LUKS1 or in plain mode, which you now want to encrypt with a secure key (see <u>"Re-encrypting from clear key to secure key onto a new volume"</u> on page 32). In this scenario, you need a new separate volume. The LUKS2 format is applied to the new volume.
- Your volume might be encrypted with a clear key using the LUKS1 format, and you now want to apply encryption with a secure key. Use the methods of the infrastructure for protected volume encryption to automatically convert to the LUKS2 format using secure key encryption (see <u>"Re-encrypting a LUKS</u> volume from clear key to secure key on the same volume" on page 32).

In both use cases it is your goal to provide enhanced security that is applied by secure keys in contrast to the previously used clear keys.

Re-encrypting from clear key to secure key onto a new volume

In this use case, you learn how to decrypt a volume that had been encrypted with a clear key in LUKS1 or in plain mode, and how to re-encrypt it with a secure key according to the infrastructure for protected volume encryption. The re-encrypted data is written to a new volume in this scenario.

Before you begin

You need a free volume, or a free partition on a volume that has sufficient space. Ensure that this volume is persistently configured to your Linux instance.

About this task

You can either have the clear-key encrypted volume as a stand-alone volume or it can be a physical volume as part of an LVM volume group.

In this procedure, you use the known clear key used for the volume encryption to decrypt the data and then use the tools of the infrastructure for protected volume encryption to re-encrypt the volume using a generated secure key.

Procedure

- 1. Open the encrypted volume.
- 2. Perform the data migration.
 - a) **With LVM:** Perform the procedure as described in <u>"Migrating to an encrypted LVM physical volume"</u> on page 25.

If you have enough free space in your LVM volume group, you can migrate one physical volume after the other, because you do not need temporary disk space to hold the migration data.

- b) **Non LVM:** Perform the procedure as described in <u>"Migrating data to a new encrypted volume" on page 31</u>.
- 3. Update your system configuration to use the new encrypted volume. You might require changes in several configuration files, for example, /etc/crypttab and /etc/fstab, depending on the usage of the encrypted volume.

Re-encrypting a LUKS volume from clear key to secure key on the same volume

This use case demonstrates how to convert a LUKS1 or LUKS2 volume that is encrypted with a clear key only, into a secure-key encrypted LUKS2 volume.

Before you begin

The described procedure is only possible with **cryptsetup** version 2.0.4 or later.

Note: At the time of writing, the *LUKS2 online reencryption* feature, that was added to **cryptsetup** with version 2.2.0, did not work with PAES cipher.

Important: Ensure that you have a backup copy of the volume that you want to re-encrypt. If the system crashes or a media error occurs during re-encryption, your original data might be destroyed.

About this task

You can either have the encrypted volume as a stand-alone volume or as a physical volume as part of an LVM volume group. In this use case, it is assumed that the volume is already protected by a clear key managed by LUKS1 or LUKS2.

Procedure

1. If your volume uses the LUKS1 format, you must convert it to LUKS2.

Use the **cryptsetup convert** command for this purpose. This requires that the volume is not mounted and not opened. Close it first using **cryptsetup luksClose**.

Important: Always create a header backup before performing this operation. Refer to the **cryptsetup** man page for more information.

```
# cryptsetup convert --type luks2 /dev/mapper/disk9
WARNING!
This operation will convert /dev/mapper/disk9 to LUKS2 format.
Are you sure? (Type uppercase yes): YES
# cryptsetup luksDump /dev/mapper/disk9
LUKS header information
Version:
                2
Epoch:
Metadata area: 12288 bytes
                5d6495ba-b6f9-43c5-883f-dff56f10c72a
UUID:
Label:
                (no label)
Laber.
Subsystem:
                (no subsystem)
Flags:
               (no flags)
. . .
```

2. Generate a secure key for /dev/mapper/disk9 in the secure key repository. See step <u>"1" on page 17</u> from "Creating a volume for pervasive encryption" on page 17.

```
# zkey generate --name secure_xtskey9 --key-type CCA-AESCIPHER --keybits 256 --xts \
--volumes /dev/mapper/disk9:enc-disk9 --volume-type LUKS2 \
--apqns 03.0039,04.0039
```

- This creates the secure key file secure_xtskey9.skey as an AES CIPHER key in the directory of the secure key repository: /etc/zkey/repository/.
- 3. Re-encrypt the volume with the generated secure key and the PAES cipher.

cryptsetup-reencrypt /dev/mapper/disk9 --cipher paes-xts-plain64 \
 --master-key-file /etc/zkey/repository/secure_xtskey9.skey --key-size 2176
Enter passphrase for key slot 0: disk9pw
...
Progress: 63.7%, ETA 03:43, 13004 MiB written, speed 34.2 MiB/s
...
Finished, time 11:22.750, 20478 MiB written, speed 30.0 MiB/s

4. Set the verification pattern into the LUKS2 header using the **zkey-cryptsetup** command.

zkey-cryptsetup setvp /dev/mapper/disk9

Results

L

I

You now have the original volume re-encrypted with a secure key using the LUKS2 format.

34 Linux on Z and LinuxONE: Pervasive Encryption for Data VolumesSeptember 2020

Chapter 6. Managing keys

Besides the generation of master keys and secure keys, these keys require reliable and secure procedures for their management during their life times.

"Secure key considerations" on page 12 describes how to obtain a secure key. Once created, there are various scenarios that require updating, exchanging, or storing secure keys.

The following subtopics provide specific information:

- "Managing a secure key repository" on page 35
- "Managing secure LUKS2 volume keys" on page 43
- "Validating a secure key" on page 46
- "Changing master keys and re-enciphering secure keys" on page 48
- "Sharing master keys across cryptographic coprocessors" on page 52
- "Replacing a cryptographic coprocessor " on page 52

You can use the **zkey** utility to obtain validation information about an existing secure key. See <u>"Validating</u> a secure key" on page 46 for more information.

Managing a secure key repository

L

You should keep the generated secure keys in a repository because each time the master key changes on the cryptographic coprocessor, you must re-encipher the secure keys with the new master key. The **zkey** command supports all tasks in the context of secure key management. These tasks are introduced and presented with an example.

For detailed information refer to the **zkey** man page or to <u>Appendix A</u>, "zkey - Managing secure keys," on page 71.

When storing the secure key in a key repository, additional information, such as a description of the key, can be associated with a secure key. You can associate a secure key with one or multiple cryptographic coprocessors (APQNs) that are set up with the same CCA AES or EP11 AES master key. You can also associate a secure key with one or multiple volumes (block devices), which are encrypted using dm-crypt with the secure key. The volume association also contains the device-mapper name, separated by a colon, used with dm-crypt. A specific volume can only be associated with one secure key.

Keep a copy of the secure keys used for disk encryption in a secure key repository. That way, you can easily re-encipher the secure keys in case of a master key change even for archived disks. LUKS2 disks contain the secure key in the LUKS2 header. You would normally re-encipher these volume keys when the master key changes. When the disk is already in the archive, this might be cumbersome. If you have a valid secure key in the repository, that has been re-enciphered with every master key change, you can still recover such an archived volume, even after multiple master key changes. For details, refer to <u>Chapter 8</u>, "Recovering secure key encrypted volumes," on page 59.

Deciding about the location of the secure key repository

The default repository location of the secure key repository is /etc/zkey/repository. Set environment variable ZKEY_REPOSITORY to point to a different location for the secure key repository.

Keys stored in a secure key repository inherit the permissions from the repository directory (except write access for other users, which is always denied). The default repository location is created with group *zkeyadm* as owner and mode 770. Thus all secure keys created in that repository are owned by group *zkeyadm*. Anyone that is supposed to access secure keys in the secure key repository must be part of group *zkeyadm*.

If you select a location using the environment variable, you can decide about the access permissions of that directory.

Keep a backup copy of the secure key repository.

Generating AES secure keys

You can generate a secure key in the secure key repository using the **zkey generate** command with the --name option.

You can specify additional information, such as a textual description of the key. You can associate a secure key with one or multiple cryptographic coprocessors (APQNs) that are set up with the same CCA AES or EP11 AES master key. You can also associate a secure key with one or multiple volumes (block devices), which are encrypted using dm-crypt with the secure key. The volume association also contains the device-mapper name, separated by a colon, used with dm-crypt. A specific volume can only be associated with one secure key.

Example:

```
# zkey generate --name secure_xtskey1 --keybits 256 --xts \
    --description "This is our secure key in a repository" \
    --volumes /dev/mapper/disk1:enc-disk1 --volume-type LUKS2 \
    --apqns 03.0039,04.0039 --sector-size 4096
```

With parameter -K or --keytype, you can also specify a key type for the generated secure key. Valid values are CCA-AESDATA to generate AES DATA keys, or CCA-AESCIPHER to generate an AES CIPHER key. To generate an EP11 AES secure key for use on CEX7S or later cryptographic coprocessors configured in EP11 mode (CEX7P), specify key type EP11-AES.

Note: Linux allows hot-plugging of cryptographic coprocessors (APQNs). You might need to update the APQN associations with the **zkey change** command when an APQN had been added to or removed from the Linux instance.

The **zkey** tool checks, whether the master key is the same for all of the APQNs that are to be associated with a secure key during creation, or for all APQNs that are involved in a modification of a secure key. In cases, where no APQNs are associated with the generation or validation of a secure key (either outside or within the secure key repository), all APQNs available on the system are checked.

Warnings are issued if the APQNs do not fulfill the requirements to ensure that all APQNs have the same master key. In case of inconsistencies, a table of all associated APQNs with its master key verification patterns is displayed, together with warning or information messages, for example:

Warning messages normally prevent the generation or validation of master keys, while information messages allow the desired action.

Starting with CEX7P cryptographic coprocessors, you can generate an EP11 secure key, for example:

zkey gen -N sec-ep11-key -K EP11-AES -a 0a.0036,0b.0036 --xts

Validating secure AES keys

Using the **zkey** utility, you can obtain validation information about an existing secure key stored in the secure key repository.

This task is described in detail in "Validating a secure key from the secure key repository" on page 46.

Re-enciphering AES secure keys

Use the **zkey reencipher** command to re-encipher an existing secure key with a new master key. An AES secure key must be re-enciphered when the AES master key of the CCA or EP11 cryptographic coprocessor changes.

The CCA cryptographic coprocessor has three different registers to store master keys:

- The CURRENT register contains the current master key.
- The OLD register contains the previously used master key. Secure keys enciphered with the master key contained in the OLD register can still be used until the master key is changed again.
- The NEW register contains the new master key to be set. The master key in the NEW register cannot be used until it is made the current master key. You can pro-actively re-encipher a secure key with the NEW master key before this key is made the CURRENT key. Use the --to-new option to do this.

Note: An EP11 cryptographic coprocessor has only two registers to store master keys, CURRENT and NEW. Therefore, old master keys are no longer available as soon as a new master key is set CURRENT.

Use the --from-old option to re-encipher a secure key that is currently enciphered with the master key in the OLD register with the master key in the CURRENT register. If both the --from-old and --to-new options are specified, a secure key that is currently enciphered with the master key in the OLD register is re-enciphered with the master key in the NEW register. If both options are omitted, **zkey** automatically detects whether the secure key is currently enciphered with the master key in the OLD register or with the master key in the CURRENT register. If currently enciphered with the master key in the OLD register, it is re-enciphered with the master key in the CURRENT register. If currently enciphered with the master key in the OLD register, it is re-enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the NEW register. If for this case the NEW register does not contain a valid master key, then the re-encipher operation fails.

To re-encipher secure keys contained in the secure key repository, specify the name of the key or a pattern containing wildcards using the --name option. When wildcards are used you must quote the value. You can also specify the --apqns option to re-encipher those secure keys that are associated with the specified cryptographic coprocessors (APQNs). You can use wildcards for the APQN specification. When wildcards are used you must quote the value. If both options --name and --apqns are specified then all secure keys contained in the key repository that match both patterns are re-enciphered. If both options are omitted, then all secure keys contained in the key repository are re-enciphered.

Re-enciphering a secure key contained in the secure key repository can be performed in-place, or in staged mode:

In-place

I

L

immediately replaces the secure key in the repository with the re-enciphered secure key. Reenciphering from OLD to CURRENT is performed in-place per default. You can use option --inplace to force an in-place re-enciphering for the CURRENT to NEW case. Be aware that a secure key that was re-enciphered in-place from CURRENT to NEW is no longer valid, until the new CCA or EP11 master key has been made the current one.

Staged mode

means that the re-enciphered secure key is stored in a separate file in the secure key repository. Thus the current secure key is still valid at this point. Once the new CCA or EP11 master key has been set and made active, you must rerun the re-encipher command with option --complete to complete the staged re-enciphering. Re-enciphering from CURRENT to NEW is performed in staged mode per default. For CCA master keys, you can use option --staged to force a staged re-enciphering for the OLD to CURRENT case.

Note: The **reencipher** command requires the CCA host library (libcsulcca.so) to be installed for the support of CCA secure keys. It requires the Linux on Z Enterprise PKCS #11 (EP11) Support Program (EP11 host library) to be installed for support of secure keys of type EP11-AES.

Examples: Assuming there is only one secure key (secure_xtskey1) matching the filters in the --name and --apqns options, the following three examples deliver the same result:

```
# zkey reencipher --name secure_xtskey1
# zkey reencipher --apqns 03.0039 --staged
# zkey reencipher --name "sec*" --apqns "*.0039" --staged
Re-enciphering key 'secure_xtskey1'
The secure key is currently enciphered with the CURRENT master key and is
being re-enciphered with the NEW master key
Staged re-enciphering is initiated for key 'secure_xtskey1'. After the NEW
master key has been set to become the CURRENT master key run 'zkey reencipher'
with option '--complete' to complete the re-enciphering process
1 keys re-enciphered, 0 keys skipped, 0 keys failed to re-encipher
```

Importing AES secure keys into the secure key repository

Use the **zkey import** command to import an existing secure key contained in a file into the secure key repository.

When importing a secure key in a key repository, additional information can be associated with a secure key using the --description, --volumes, --apqns, or the --sector-size options.

Example:

```
# zkey import seckey.bin --name imported_seckey
# zkey import seckey.bin --name imported_seckey \
--description "This is an imported secure key" \
--volumes /dev/mapper/disk1:enc-disk1 --volume-type LUKS2 \
--apqns 03.0039,04.0039
```

To import a secure LUKS2 volume key from a volume encrypted with that secure key, you need to first export the volume key into a binary file and then import it into the secure key repository and associate the volume with it.

Example:

```
# cryptsetup luksDump /dev/mapper/disk<n> --dump-master-key \
    --master-key-file seckey.bin
WARNING!
========
Header dump with volume key is sensitive information
which allows access to encrypted partition without passphrase.
This dump should be always stored encrypted on safe place.
Are you sure? (Type uppercase yes): YES
Enter passphrase for /dev/mapper/disk<n>: disk<n>pw
```

zkey import seckey.bin --name imported_key_of_disk<n> \
--volumes /dev/mapper/disk<n>:enc-disk<n>

Notes:

- A **# touch seckey.bin** command may be required prior to the **cryptsetup luksDump** command. The command creates an empty target file for the secure key to be retrieved. Hereafter, the secure key is contained in this file. This is due to a bug in **cryptsetup** versions 2.0.3 and 2.0.4. In **cryptsetup** 2.0.5, this bug is fixed. The command **cryptsetup luksDump** creates the file itself, and will fail when the file is already available.
- The **zkey import** command can also import AES CIPHER keys and EP11 secure keys. When importing an AES CIPHER key, additional checks are performed on this key, such as a check of the history section of the secure key. If a potentially insecure setting is detected, you are prompted to confirm the import. Imported AES CIPHER keys are restricted for export in any way, regardless of the export setting of the imported keys. The only export that is kept allowed is the export to CPACF protected keys, so that these keys can be used with the PAES cipher.

• The **zkey import** command requires the CCA host library (libcsulcca.so) to be installed when secure keys of type CCA-AESCIPHER are imported. For the supported environments and downloads, see:

http://www.ibm.com/security/cryptocards

Exporting AES secure keys from the secure key repository

Use the **zkey export** command to export a secure key contained in the secure key repository to a file in the file system.

Specify the name of the key that is to be exported using the --name option. You cannot use wildcards. The exported secure key also remains in the secure key repository.

Example:

zkey export seckey.bin --name secure_xtskey

Listing AES secure keys contained in the secure key repository

Use the zkey **list** command to display a list of secure keys contained in the secure key repository.

You can filter the displayed list by key name, key type, associated volumes, associated cryptographic coprocessors (APQNs), and volume type. You can use wildcards for the key name, associated APQNs, and associated volumes. The device-mapper name of an associated volume can be omitted. If it is specified, then only those keys are listed that are associated with the specified volume and device-mapper name. The **list** command displays the attributes of the secure keys, such as key sizes, whether it is a secure key that can be used for the XTS cipher mode, the textual description, associated cryptographic coprocessors (APQNs) and volumes, the sector size, the key verification pattern, and time stamps for key creation, last modification and last re-encipherment.

Examples: Assuming there is only one secure key (secure_xtskey1) matching the specified filters, the following examples deliver the same result:

```
# zkey list
# zkey list --name "secure*"
# zkey list --apqns "*.0039"
# zkey list --volumes "/dev/mapper/disk*"
# zkey list --volumes "/dev/mapper/disk*"
# zkey list --volumes "*:enc-disk*" --apqns "*.0039"
Key : secure_xtskey1
Description : This is our secure key in a repository
Secure key size : 272 bytes
Clear key size : 512 bits
XTS type key : Yes
Key type : CCA-AESCIPHER
Volumes : /dev/mapper/disk1:enc-disk1
APQNs : 03.0039
Key file name : /etc/zkey/repository/secure_xtskey1.skey
Sector size : (system default)
Volume type : LUKS2
Verification pattern : ac08c5d154374a247d6bbbae047ab9f8
541575915e764f6e35817b56bcf7c999
Created : 2020-08-20 16:57:32
Changed : (never)
```

Removing AES secure keys

L

L

Use the **zkey remove** command to remove an existing secure key from the secure key repository.

Specify the name of the key that is to be removed using the --name option. You cannot use wildcards. The **remove** command prompts for a confirmation, unless you specify the --force option.

Note: When removing a secure key that is associated with one or multiple volumes, and the key's volume type is PLAIN, a message informs you about the associated volumes. When the secure key is removed, these volumes can no longer be used, unless you have a backup of the secure key.

For keys with volume type LUKS2 no such message is issued, because the secure key is contained in the LUKS2 header.

Examples:

volume type LUKS2

```
#zkey remove --name secure_xtskey1
zkey: Remove key 'secure_xtskey1'? y
```

volume type PLAIN

```
#zkey remove --name secure_xtskey1
When you remove key 'secure_xtskey1' the following volumes will no longer be
usable:
   /dev/mapper/disk1:enc-disk1
zkey: Remove key 'secure_xtskey1'? y
```

Changing AES secure keys

Use the **zkey change** command to change the description, the associated volumes, the associated cryptographic coprocessors (APQNs), the sector size, and the volume type of a secure key contained in the secure key repository.

Specify the name of the key that is to be changed using the --name option. You cannot use wildcards.

You can set, replace, add, or remove volume and cryptographic coprocessor (APQN) associations. To set or replace an association, specify the association with the --volumes or the --apqns options. To add an association, specify the new association prefixed with a + sign with the --volumes or the --apqns option. To remove an association, specify the association to remove prefixed with a - sign with the -volumes or the --apqns option. You cannot mix + and - in one specification. You can either add or remove or set the associations with one command.

Note: The secure key itself cannot be changed, only information about the secure key is changed. To rename a secure key, use the **rename** command. To re-encipher a secure key with a new CCA or EP11 master key, use the **reencipher** command.

Example:

I

```
# zkey change --name secure_xtskey1 --volumes +/dev/mapper/disk2:enc-disk2
# zkey change --name secure_xtskey1 --apqns -04.0039
# zkey change --name secure_xtskey1 --volume-type plain
# zkey change --name secure_xtskey1 --sector-size 4096
```

Note: Linux allows hot-plugging of cryptographic coprocessors (APQNs). You might need to update the APQN associations when an APQN had been added to or removed from the Linux instance.

Renaming AES secure keys

Use the **zkey rename** command to rename a secure key in the secure key repository.

Specify the name of the key that is to be renamed using the --name option and the new name using the --new-name option. You cannot use wildcards.

Note: When renaming a secure key that is associated with one or multiple volumes, and the key's volume type is PLAIN, a message informs you about the associated volumes. When the secure key is renamed, these volumes can no longer be used, unless you change the name of the secure key in the **cryptsetup plainOpen** commands and in the /etc/crypttab entries.

For keys with volume type LUKS2 no such message is issued, because the secure key is contained in the LUKS2 header.

Examples:

volume type LUKS2

```
# zkey rename --name secure_xtskey1 --new-name secure_xtskey2
```

volume type PLAIN

```
# zkey rename --name secure_xtskey1 --new-name secure_xtskey2
The following volumes are associated with the renamed key 'secure_xtskey2'. You
should adjust the corresponding crypttab entries and 'cryptsetup plainOpen'
commands to use the new name.
   /dev/mapper/disk1:enc-disk1
```

Copying AES secure keys in the secure key repository

Use the **zkey copy** command to copy (duplicate) an existing secure key in the secure key repository under a new name.

Specify the name of the key that is to be copied using the --name option and the name of the copied key using the --new-name option. You cannot use wildcards.

Note: When copying a secure key, the volume associations are not copied because a specific volume can only be associated with a single secure key. Specify the --volumes option to associate different volumes with the copied secure key, or use the **change** command to associate volumes afterwards.

Examples:

Generating crypttab entries for encrypted volumes

Use the **zkey crypttab** command to generate crypttab entries using the plain or LUKS2 dm-crypt mode for volumes that are associated with secure keys contained in the secure key repository.

Specify the --volumes option to limit the list of volumes where crypttab entries are generated for. You can use wildcards. If wildcards are used you must quote the value. The device-mapper name of an associated volume can be omitted. If it is specified, then only those volumes with a matching volume name and device-mapper name are selected. Specify the --volume-type option to generate crypttab entries for the specified volume type only.

The options that you specify with the **zkey crypttab** command are used to generate the crypttab entries with the options at the desired places. For LUKS2 volumes, a passphrase is required. You are prompted for the passphrase during system startup when crypttab is evaluated, unless option --keyfile is specified. Option --tries specifies how often a passphrase can be re-entered. When option -key-file is specified, the passphrase is read from the specified file. You can specify options -keyfile-offset and --keyfile-size to control which part of the key file is used as passphrase. These options are passed to the generated crypttab entries and are only available if zkey has been compiled with LUKS2 support enabled.

Examples:

- for volume type LUKS2

```
# zkey crypttab
# zkey crypttab --volumes /dev/mapper/disk1
# zkey crypttab --volume-type luks2
```

enc-disk1 /dev/mapper/disk1

Note: To use automated opening of the encrypted volume with a key file during system startup, you must adapt the generated crypttab entry. Follow the instructions from step <u>"6" on page 20</u> in <u>"Creating a</u> volume for pervasive encryption" on page 17.

```
- for volume type PLAIN
```

I

```
# zkey crypttab --volume-type plain
```

enc-disk2 /dev/mapper/disk2 /etc/zkey/repository/secure_xtskey2.skey \ plain,cipher=paes-xts-plain64,size=2176,hash=plain

- for LUKS2 disk1 with options:

<pre># zkey crypttab</pre>	volumes /dev/mapper/	disk1key-file /etc/luks-keys/	all_disks.key
enc-disk1	/dev/mapper/disk1	/etc/luks-keys/all_disks.key	luks
# zkey crypttabvolumes /dev/mapper/disk1key-file /etc/luks-keys/all_disks.key \ keyfile-offset 27keyfile-size 8tries 3			
enc-disk1	/dev/mapper/disk1 keyfile-offset=27 keyf	/etc/luks-keys/all_disks.key ile-size=8 tries=3	luks,\

Generating cryptsetup commands for encrypted volumes

- Use the **zkey cryptsetup** command to generate **cryptsetup plain0pen**, **cryptsetup luks0pen**, or **cryptsetup luksFormat** commands for volumes that are associated with secure keys contained in the secure key repository.
 - For LUKS2 volumes, the **zkey cryptsetup** command generates **cryptsetup luksFormat** commands.
 - With option --open, the command generates **cryptsetup luksOpen** commands for LUKS2 volumes or **cryptsetup plainOpen** for plain mode volumes (same as without --open option).
 - For plain mode volumes, this command only generates **cryptsetup** plainOpen commands.
 - With the --format option, you can limit the generated commands to only generate **cryptsetup luksFormat** commands for LUKS2 volumes, and skip plain mode volumes. Without the --format and --volume-type options, **zkey cryptsetup** generates the commands dependent from the detected volume type:
 - For PLAIN volumes, a **cryptsetup plain0pen** command is generated.
 - For LUKS2 volumes, a **cryptsetup luksFormat** command is generated.

Specify the --volumes option to limit the list of volumes where **cryptsetup** commands are generated for. You can use wildcards. When wildcards are used you must quote the value. The device-mapper name of an associated volume can be omitted. If it is specified, then only those volumes with the specified volume and device-mapper name are selected. Specify the --volume-type option to generate **cryptsetup** commands for the specified volume type only. Specify the --run option to run the generated **cryptsetup** commands.

For LUKS2 volumes, a passphrase is required. You are prompted for the passphrase when running the generated commands, unless option --key-file is specified. Option --tries specifies how often a passphrase can be reentered. When option --key-file is specified, the passphrase is read from the specified file. You can specify options --keyfile-offset and --keyfile-size to control which part of the key file is used as passphrase. These options are only available if zkey has been compiled with LUKS2 support enabled.

To avoid cryptsetup confirmation questions, you can specify the --batch-mode option.

All these options are passed to the generated **cryptsetup** command(s) and behave in the same way as with using **cryptsetup** originally.

Examples:

L

L

I

- for volume type LUKS2

- for volume type PLAIN

zkey cryptsetup --volume-type plain

```
cryptsetup plainOpen --key-file '/etc/zkey/repository/secure_xtskey2.skey' \
--key-size 2176 \
--cipher paes-xts-plain64 /dev/mapper/disk2 enc-disk2
```

- for volume type LUKS2, reading the passphrase from a key file, and let the generated commands run in batch mode. The supposed content of the /etc/luks2-keys/passphrases.txt key file is:

abcdefgh ijklmnop qrstuvw xyzabcde fghijklm nopqrstu

The targeted passphrase xyzabcde starts at offset 27.

Managing secure LUKS2 volume keys

Use **zkey-cryptsetup** to validate and re-encipher secure LUKS2 volume keys of volumes encrypted with LUKS2 and the PAES cipher. These secure LUKS2 volume keys of type AES are produced in two steps: First, a random plain text key is wrapped with an AES master key of a cryptographic coprocessor. Then this secure AES key is again wrapped by LUKS2 with a key derived from a user passphrase or key file. The result is a secure LUKS2 volume key of type AES (sometimes shortly referred to as AES volume key in this documentation).

When you open a key slot contained in the LUKS2 header of the volume using **zkey-cryptsetup**, a passphrase is required. You are prompted for the passphrase, unless option --key-file is specified. Option --tries specifies how often a passphrase can be re-entered. When option --key-file is specified, the passphrase is read from the specified file. You can specify options --keyfile-offset and --keyfile-size to control which part of the key file is used as passphrase. These options behave in the same way as with **cryptsetup**.

For detailed information refer to the **zkey-cryptsetup** man page or to <u>Appendix B, " zkey-cryptsetup -</u> Managing LUKS2 volume keys," on page 85.

To encrypt a volume using LUKS2 and the PAES cipher, generate a secure AES key in a specified file using the **zkey** command. Then format the device with **cryptsetup luksFormat** using the generated secure AES key (see "Creating a volume for pervasive encryption" on page 17).

Validate a secure LUKS2 volume key

Using the **zkey-cryptsetup validate** command, you can obtain validation information about a secure key in the LUKS2 header of an encrypted volume.

You can find more detailed information about the validation of LUKS2 volume keys in <u>"Validating a secure</u> key used with a LUKS2 volume" on page 47.

Re-encipher a secure LUKS2 volume key

Use the **zkey-cryptsetup reencipher** command to re-encipher a secure LUKS2 volume key of a volume encrypted with LUKS2 and the PAES cipher.

A secure AES volume key must be re-enciphered when the AES master key of the cryptographic coprocessor in CCA or EP11 coprocessor mode changes. Such a coprocessor has three different registers to store master keys: the CURRENT, the OLD, and the NEW register, as described in <u>"Re-enciphering AES</u> secure keys" on page 37. Note that EP11 coprocessors do not contain an OLD register.

zkey-cryptsetup automatically detects whether the secure volume key is currently enciphered with the master key in the OLD register (not valid for EP11 secure keys, no OLD register on EP11 coprocessors) or with the master key in the CURRENT register. If currently enciphered with the master key in the OLD register, it is re-enciphered with the master key in the CURRENT register (not valid for EP11 secure keys). If it is currently enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the CURRENT register does not contain a valid master key, then the re-encipher operation fails.

Re-enciphering a secure volume key of a volume encrypted with LUKS2 and the PAES cipher can be performed in-place, or in staged mode:

In-place

L

L

L

L

immediately replaces the secure volume key in the LUKS2 header of the encrypted volume with the re-enciphered secure volume key. Re- enciphering from OLD to CURRENT is performed in-place per default. You can use option --in-place to force an in-place re-enciphering for the CURRENT to NEW case. Be aware that an encrypted volume with a secure volume key that was re-enciphered in-place from CURRENT to NEW is no longer usable, until the new CCA or EP11 master key has been made the current one.

Staged mode

means that the re-enciphered secure volume key is stored in a separate (unbound) key slot in the LUKS2 header of the encrypted volume. Thus all key slots containing the current secure volume key are still valid at this point. Once the new CCA or EP11 master key has been set (made active), you must rerun the re-encipher command with option --complete to complete the staged re-enciphering. When completing the staged re-enciphering, the (unbound) key slot containing the re-enciphered secure volume key becomes the active key slot and, optionally, all key slots containing the old secure volume key are removed. Re-enciphering from CURRENT to NEW is performed in staged mode per default. You can use option --staged to force a staged re-enciphering for the OLD to CURRENT case.

The NEW register of a cryptographic coprocessor may contain a new master key to be set. This master key in the NEW register cannot be used until it is made the current master key. Use the --to-new option of the **zkey-cryptsetup reencipher** command to pro-actively re-encipher a secure key with the NEW master key before this key is made the CURRENT key.

Use the --from-old option to re-encipher a secure volume key that is currently enciphered with the master key in the OLD register with the master key in the CURRENT register.

When re-enciphering or setting LUKS2 volume keys using the **zkey-cryptsetup reencipher** command, you can specify the --batch-mode option to suppress confirmation questions. All

confirmations when running the underlying **cryptsetup** commands are assumed to be answered with YES. This allows better automation of **zkey-cryptsetup** commands.

Note: For information about **zkey-cryptsetup**, and how to avoid the need to enter a passphrase when opening a key slot contained in the LUKS2 header of a volume, read <u>Appendix B</u>, "zkey-cryptsetup - Managing LUKS2 volume keys," on page 85.

Examples:

To re-encipher the secure key of the encrypted volume /dev/mapper/disk1 in staged mode and complete it later:

zkey-cryptsetup reencipher /dev/mapper/disk1 --staged

Enter passphrase for '/dev/mapper/disk1': disk1pw The secure volume key of device '/dev/mapper/disk1' is enciphered with the CURRENT master key and is being re-enciphered with the NEW master key. Staged re-enciphering is initiated for device '/dev/mapper/disk1'. After the NEW master key has been set to become the CURRENT master key, run 'zkey-cryptsetup reencipher' with option '--complete' to complete the re-enciphering process.

zkey-cryptsetup reencipher /dev/mapper/disk1 --complete

To re-encipher the secure key of the encrypted volume /dev/mapper/disk1 in in-place mode and in batch mode:

zkey-cryptsetup reencipher /dev/mapper/disk1 --in-place --batch-mode

Set a verification pattern of the secure LUKS2 volume key

Use the **setvp** command to set a verification pattern of the secure LUKS2 volume key of a volume encrypted with LUKS2 and the PAES cipher.

The verification pattern identifies the effective key used to encrypt the volume's data. The verification pattern is stored in a token named paes-verification-pattern in the LUKS2 header.

Note: Set the verification pattern right after formatting the volume using **cryptsetup luksFormat**.

For information about **zkey-cryptsetup**, and how to avoid the need to enter a passphrase when opening a key slot contained in the LUKS2 header of a volume, read <u>Appendix B</u>, "zkey-cryptsetup - Managing LUKS2 volume keys," on page 85.

Example: To set the verification pattern of the secure key of the encrypted volume /dev/mapper/disk1:

zkey-cryptsetup setvp /dev/mapper/disk1

Set a new secure LUKS2 volume key

I

I

L

Use the **zkey-cryptsetup setkey** command to set a new secure LUKS2 volume key for a volume encrypted with LUKS2 and the PAES cipher.

Use this command to recover from an invalid secure AES volume key contained in the LUKS2 header. Such a key can become invalid when the CCA or EP11 master key changed without re-enciphering the secure volume key.

You can recover the secure volume key only if you have a copy of the secure key in a file, and this copy was re-enciphered when the CCA or EP11 master key has been changed. Thus, the copy of the secure key must be currently enciphered with the master key in the CURRENT or OLD master key register. Specify the secure key file with option --master-key-file to set this secure key as the new volume key. Remember that OLD master key registers are not available on EP11 coprocessors.

In case the LUKS2 header of the volume contains a verification pattern token, it is used to ensure that the new volume key contains the same effective key. If no verification pattern token is available, then you are prompted to confirm that the specified secure key is the correct one.



L

Attention: If you set a wrong secure key you will lose all the data on the encrypted volume.

Example: To set the secure key contained in file seckey.key as the new key for the encrypted volume /dev/mapper/disk1:

zkey-cryptsetup setkey /dev/mapper/disk1 --master-key-file seckey.key

Note: For information about **zkey-cryptsetup**, and how to avoid the need to enter a passphrase when opening a key slot contained in the LUKS2 header of a volume, read <u>Appendix B</u>, "zkey-cryptsetup - Managing LUKS2 volume keys," on page 85.

Validating a secure key

You can obtain validation information about a secure key. This is helpful in a scenario where you need to re-encipher a secure key due to a master key change.

The following subtopics provide specific information:

- "Validating a secure key from the secure key repository" on page 46
- "Validating a secure key used with a LUKS2 volume" on page 47
- "Validating a secure key from a file" on page 47

Validating a secure key from the secure key repository

Using the **zkey validate** command, you can obtain validation information about a secure key stored in the secure key repository.

Specifying the **zkey validate** command checks if the specified secure key is valid. It also displays further attributes of this secure key, such as the key size, key type, whether it is a secure key that can be used for the XTS cipher mode, and the register (CURRENT or OLD) where the master key resides with which the secure key is enciphered, together with the master key verification pattern (MKVP). EP11 MKVPs are 16 bytes long, CCA MKVPs are 8 bytes long.

Example command with output for a valid secure key:

```
# zkey validate --name secure xtskev1
Key
                                         : secure xtskey1
           Status
                                        : Valid
           Description
          Description :

Secure key size : 272 bytes

Clear key size : 512 bits

XTS type key : Yes

Key type : CCA-AESCIPHER

Enciphered with : CURRENT master key (MKVP: 26d69731a66f4255)

Volumes : /dev/mapper/disk1:enc-disk1

APQNs : 03.0039

04.0039
           Wey file name04.0039Sector size: /etc/zkey/repository/secure_xtskey1.skeyVolume type: LUKS2
           Verification pattern : 303344b12b8258840fa11852a4ecc6d5
                            .
84c7a867f893a5dcc0d499557c45bee6
2020-07-23 15:27:20
           Created
           Changed
                                         : (never)
           Re-enciphered : (never)
1 keys are valid, 0 keys are invalid, 0 warnings
```

The displayed verification pattern can be used to identify the effective key contained in this secure key. Any secure key with the same verification pattern contains the same effective key. If the secure key is not valid because the master key with which it was wrapped is no longer available, the **zkey** utility shows a similar output as for a valid secure key, however, with Status: Invalid, and some other properties are indicated as (unknown).

The **zkey** tool checks, whether the master key is the same for all of the APQNs associated with a secure key during validation of a secure key. In cases, where no APQNs are associated with the validation of a secure key, then all APQNs available on the system are checked.

For more information, also refer to "Generating AES secure keys" on page 36 and to the **zkey** man page.

Validating a secure key used with a LUKS2 volume

Using the **zkey-cryptsetup validate** command, you can obtain validation information about a secure key in the LUKS2 header of an encrypted volume.

Specifying the **zkey-cryptsetup validate** command checks if the specified LUKS2 volume contains a valid secure key. It also displays further attributes of its secure key, such as the key size, whether it is a secure key that can be used for the XTS cipher mode, and the master key register (CURRENT or OLD) with which the secure key is enciphered. It also displays the verification pattern of the secure key, if available, that is, if it had been set using the **setvp** command.

For further information about master key registers, see "Re-enciphering AES secure keys" on page 37.

Example: To validate the secure key of the encrypted volume /dev/mapper/disk<n> and display its attributes, enter:

```
# zkey-cryptsetup validate /dev/mapper/disk<n>
Enter passphrase for '/dev/mapper/disk<>>': disk<n>pw
Validation of secure volume key of device '/dev/mapper/disk<n>':
  Status:
                            Valid
                            272 bytes
  Secure key size:
  XTS type key:
                            Yes
  Key type:
                            CCA-AESCIPHER
                            512 bits
  Clear key size:
                            CURRENT master key (MKVP: 26d69731a66f4255)
  Enciphered with:
  Verification pattern: 477a8608f06743569e2c62fc5fe00085
                            08b18a80d7616094eceaa746be4a2edd
```

If the secure key is not valid because the master key with which it was wrapped is no longer available, the **zkey** utility shows a similar output as for a valid secure key, but with Status: **Invalid**, and some other properties are indicated as (unknown).

Note: For information about **zkey-cryptsetup** and how to avoid the need to enter a passphrase, read Appendix B, " zkey-cryptsetup - Managing LUKS2 volume keys," on page 85.

Validating a secure key from a file

L

L

L

L

Using the **zkey validate** command, you can obtain validation information about an existing secure key stored in a binary file.

Specifying the **zkey validate** command checks if the specified file contains a valid secure key. It also displays further attributes of this secure key, such as the key size, whether it is a secure key that can be used for the XTS cipher mode, and the master key register (CURRENT or OLD) with which the secure key is enciphered.

Example command with output for a valid secure key:

```
# zkey validate secure_xtskey1.bin
Validation of secure key in file 'secure_xtskey1.bin':
Status: Valid
Secure key size: 272 bytes
Key type: CCA-AESCIPHER
Clear key size: 512 bits
XTS type key: Yes
Enciphered with: CURRENT master key (MKVP: 26d69731a66f4255)
Verification pattern: 0aa2b29a40c946de9b6ae4c7410ffaa2
96ad1b20d242c8e5847c821aacbb80bf
```

The displayed verification pattern can be used to identify the effective key contained in this secure key. Any secure key with the same verification pattern contains the same effective key.

If the secure key is not valid because the master key with which it was wrapped is no longer available, the **zkey** utility shows an error message:

zkey: Failed to validate a secure key: No such device zkey: The secure key in file 'seckey.bin' is not valid

The *No such device* message indicates that there is no cryptographic coprocessor available with the master key that was used to wrap this secure key.

For more information, also refer to the **zkey** man page.

Changing master keys and re-enciphering secure keys

Your security policies might require that a new master key must be generated on the cryptographic coprocessors in certain time intervals. Hereafter, you need to re-encipher all the secure keys that have been generated with the current master key. If a new master key must be used to re-encipher the secure key, the re-enciphering of the applicable secure keys depends on where these are stored: either in a secure key repository, or as a volume key in a LUKS2 header, or just in a file in the file system.

The information presented in this topic is valid for both CCA and EP11 master keys. How to set an EP11 master key is outlined in the EP11 documentation: *Exploiting Enterprise PKCS #11 using openCryptoki*. How to change a CCA master key is documented in *How to set an AES master key*.

For security and ease of use, always store your secure keys in a secure key repository. To manage a required change of the master key, you can then use the **zkey** utility to perform the required reencryption. Read "Re-enciphering secure keys from a repository" on page 48 for more information.

If the secure keys are stored as volume keys in the LUKS2 header of your volume, you can use the **zkeycryptsetup** utility to perform the re-encryption. In this case, read <u>"Re-enciphering LUKS2 volume keys"</u> on page 49.

Also, you might have saved your secure key in a file. In this case, too, you can use the **zkey** utility to perform the re-encryption. Read <u>"Re-enciphering secure keys from a file" on page 51</u> for more information.

Re-enciphering secure keys from a repository

Read how to use the **zkey reencipher** command to re-encipher a secure key that is stored in a secure key repository.

Before you begin

This task requires that a new CCA or EP11 AES master key has been set on the attached cryptographic coprocessors with the TKE. Using the **zkey** utility for re-enciphering a CCA AES secure key requires the IBM CCA host library (libcsulcca.so). For re-enciphering an EP11 AES secure key, **zkey** requires the Linux on Z Enterprise PKCS #11 (EP11) Support Program (EP11 host library) to be installed.

To obtain information about your current secure key, perform the procedure described in <u>"Validating a</u> secure key from the secure key repository" on page 46.

Procedure

I

1. Load the key parts for a new CCA or EP11 AES master key using the TKE.

If you use multiple APQNs with the same master key, load the same new master key on these APQNs. Do not yet set the new master key active at this time.

2. Re-encipher all secure keys contained in the secure key repository that are associated with the APQN for which you change the master key.

For example, you can use a command similar to the following:

zkey reencipher --apqns <apqn1,apqn2,...>

Note:

You can re-encipher a secure key that is currently enciphered under the master key in the CURRENT register of the CCA or EP11 coprocessor to the master key in the NEW register, as long as the new master key has not been activated (set). For this purpose, use option --to-new with the **zkey** utility.

You can also re-encipher a secure AES DATA or AES CIPHER key that is currently enciphered under the master key in the OLD register of the cryptographic adapter to the master key in the CURRENT register. For this purpose, use option --from-old with the **zkey** utility. This option is not available for EP11 AES keys, as cryptographic coprocessors in EP11 mode do not have an OLD register.

If both options --from-old and --to-new are specified, a secure key that is currently enciphered with the master key in the OLD register is re-enciphered with the master key in the NEW register.

Finally, you can use the auto-detection function of **zkey**. The utility detects whether the secure key is enciphered with a master key from the OLD or from the CURRENT register and re-enciphers the secure key with the appropriate new master key as described.

Re-enciphering a secure key contained in the secure key repository can be performed in-place, or in staged mode. Staged mode means that the re-enciphered secure key is stored in a separate file in the secure key repository. Thus the current secure key is still valid at this point. Once the new CCA or EP11 master key has been set (made active), you need to rerun the **reencipher** command with option -- complete to complete the staged re-enciphering. Re-enciphering from CURRENT to NEW is performed in staged mode by default. You can use option --staged to force a staged re-enciphering for the OLD to CURRENT case.

Examples:

L

zkey reencipher --apqns <apqn1,apqn2,...> --from-old --staged

zkey reencipher --apqns <apqn1,apqn2,...> --to-new --staged

zkey re-encipher --apqns <apqn1,apqn2,...> --staged

3. Now set the new master key active. Use the functions provided by the TKE for this purpose.

4. Complete the re-enciphering:

zkey reencipher --apqns <apqn1,apqn2,...> --complete

Note: To re-encipher a single secure key stored in the key repository, run:

zkey reencipher --name secure_key1

This re-enciphers the secure key with the name secure_key1.

For more information, see "Managing a secure key repository" on page 35.

Re-enciphering LUKS2 volume keys

If a new master key must be used to re-encipher a secure key used as a LUKS2 volume key, you can use the **zkey-cryptsetup reencipher** command to perform this task.

Before you begin

This task requires that a new CCA or EP11 master key has been set on the attached cryptographic coprocessors with the TKE. Using the **zkey-cryptsetup** utility for re-enciphering a LUKS2 volume key encrypted with a CCA master key requires the IBM CCA host library (libcsulcca.so) to be installed. For re-enciphering a LUKS2 volume key encrypted with an EP11 AES secure key, **zkey-cryptsetup** requires the Linux on Z Enterprise PKCS #11 (EP11) Support Program (EP11 host library) to be installed.

To obtain information about your current secure key (which is the same as the LUKS2 volume key), perform the procedure described in "Validating a secure key used with a LUKS2 volume" on page 47.

Note: You need to perform this procedure for all volumes encrypted with a secure LUKS2 volume key.

If the disks are encrypted with secure keys stored in the secure key repository, then you can use the following command to get a list of keys and their associated volumes that require re-encipherment when changing the master key of a specific APQN:

```
zkey list --apqns <apqn1,apqn2,...>
```

Procedure

- 1. Load the CCA key parts for a new AES master key or EP11 master key using the TKE. Do not yet set the new master key active at this time.
- 2. Re-encipher the LUKS2 volume key.

The **zkey-cryptsetup** command automatically detects whether the secure volume key is enciphered with a master key from the OLD or from the CURRENT register and re-enciphers the secure volume key with the appropriate new master key in the NEW register. You can explicitly specify this processing using the options --from-old or --to-new. Detecting a master key from the OLD register is not possible for EP11 master keys.

In addition, re-enciphering a secure volume key can be performed in-place, or in staged mode.

Example:

I

```
# zkey-cryptsetup reencipher /dev/mapper/disk1 --staged
Enter passphrase for '/dev/mapper/disk1': disk1pw
The secure volume key of device '/dev/mapper/disk1' is enciphered with the
CURRENT master key and is being re-enciphered with the NEW master key.
Staged re-enciphering is initiated for device '/dev/mapper/disk1'. After the NEW
master key has been set to become the CURRENT master key, run 'zkey-cryptsetup
reencipher' with option '--complete' to complete the re-enciphering process.
```

- 3. Now set the new CCA or EP11 AES master key active.
- 4. Complete the re-enciphering:

zkey-cryptsetup reencipher /dev/mapper/disk1 --complete

Note: If you used several key slots on your LUKS2 volume before, only the one for which you entered the passphrase is kept. All other key slots are removed during re-enciphering. You need to add them again using **cryptsetup luksAddKey**.

This applies for example, if you use additional key slots for automatic opening volumes at system startup.

For more information on staged re-enciphering of keys and the **zkey-cryptsetup** command, read both, <u>"Re-encipher a secure LUKS2 volume key" on page 44 and Appendix B</u>, <u>"zkey-cryptsetup -</u> <u>Managing LUKS2 volume keys," on page 85</u>. In addition, you can also refer to the **zkey-cryptsetup** man page.

5. Reset the pbkdf option to use PBKDF2.

This step is not required when using the **zkey-cryptsetup** tool from an <u>s390-tools</u> package upstream version 2.9.0 or later.

The re-enciphering process of the **zkey-cryptsetup** tool from an <u>s390-tools package</u> older than version 2.9.0 sets the value of option pbkdf of the key slot with the interactive passphrase to the default Argon2i password-based key derivation function (PBKDF), no matter which value you had set before.

As described in <u>"Out-of-memory errors when opening a LUKS2 volume" on page 58</u>, the use of the Argon2i key derivation function may cause an out-of-memory error when opening a LUKS2 volume. To

avoid such an error during automatic unlocking of the encrypted volume at system startup, change the value of option pbkdf to use the PBKDF2 key derivation function:

```
# cryptsetup luksConvertKey --pbkdf pbkdf2 /dev/mapper/disk1
Enter passphrase for keyslot to be converted:
disk1pw
```

Re-enciphering secure keys from a file

If a new master key must be used to re-encipher a secure key stored in a binary file, you can use the **zkey reencipher** command to perform this task.

Before you begin

This task requires that a new CCA or EP11 master key has been set on the attached cryptographic coprocessors with the TKE.. Using the **zkey** utility for re-enciphering a CCA AES secure key requires the IBM CCA host library (libcsulcca.so). For re-enciphering an EP11 AES secure key, **zkey** requires the Linux on Z Enterprise PKCS #11 (EP11) Support Program (EP11 host library) to be installed.

To obtain information about your current secure key, perform the procedure described in <u>"Validating a</u> secure key from a file" on page 47.

Procedure

L

L

L

1. Load the CCA key parts for a new AES master key or EP11 master key using the TKE. Do not yet set the new master key active at this time.

2. Re-encipher the secure key.

For example, you can use a command similar to the following:

zkey reencipher <oldSK_binary_file> [--output <newSK_binary_file>]

Note:

You can re-encipher a secure key that is currently enciphered under the master key in the CURRENT register of the CCA or EP11 coprocessor to the master key in the NEW register, as long as the new master key has not been activated (set). For this purpose, use option --to-new with the **zkey** utility.

For CCA secure keys only, you can also re-encipher a secure key which is currently enciphered under the master key in the OLD register of the cryptographic adapter to the master key in the CURRENT register. For this purpose, use option --from-old with the **zkey** utility. A cryptographic coprocessor configured in EP11 mode does not have an OLD register. So the --from-old option is not available.

If both options --from-old and --to-new are specified, a secure key that is currently enciphered with the master key in the OLD register is re-enciphered with the master key in the NEW register.

Finally, you can use the auto-detection function of **zkey**. The utility detects whether the secure key is enciphered with a master key from the OLD or from the CURRENT register and re-enciphers the secure key with the appropriate new master key as described.

For more information, also refer to the **zkey** man page.

Examples:

zkey reencipher securekey.bin --from-old [--output securekey2.bin]

zkey reencipher securekey.bin --to-new [--output securekey2.bin]

zkey re-encipher securekey.bin [--output securekey2.bin]

3. Now set the new CCA or EP11 master key active. Use the functions provided by the TKE for this purpose.

If you stored the re-enciphered secure key in a separate file ([--output <newSK_binary_file>]), from now on you should use the new secure key file. You can still use the original secure key file until you change the master key again, because the previous master key is still available in the OLD register (not valid for EP11 coprocessors).

Sharing master keys across cryptographic coprocessors

If you share master keys on different redundant cryptographic coprocessors, you can provide for high availability of your encrypted data.

In case of a required CCA or EP11 master key change, you must set the same master key on all used cryptographic coprocessors. In this case, generate the master key or the master key parts on one or more smart cards and use these smart cards as the source for loading the key on all cryptographic coprocessors.

For information on how to set a master key, refer to <u>How to set an AES master key</u> in the IBM Knowledge Center.

Replacing a cryptographic coprocessor

I

I

The reasons why you might want to exchange a cryptographic coprocessor which is in use for volume encryption is that you either want to upgrade to a new model, or you want to switch from a CCA coprocessor to a EP11 coprocessor, or that you need to replace the old coprocessor due to a defect. The described scenarios include the case where you want to continue to use the old master key on the new coprocessor, as well as the cases where you want to use a different master key with or without the clear key being available.

The following subtopics provide specific information:

- "Replacing with the same master key" on page 52
- "Replacing with a different master key" on page 53

The scenario of replacing a CCA coprocessor by a EP11 coprocessor is equivalent with changing an encrypted volume from being encrypted with a CCA secure key (AES CIPHER or CCA-AESDATA) to being encrypted with an EP11 AES secure key (EP11-AES). Therefore, the required steps are similar as for "Replacing with a different master key" on page 53 with the appropriate differences.

A scenario where you first set the new master key on the old and on the new cryptographic coprocessor, then re-encipher the volume using the old coprocessor with the new master key, and then replace the old cryptographic coprocessor with the new one is not described here explicitly, because this is very similar to the use cases described in "Changing master keys and re-enciphering secure keys" on page 48.

Replacing with the same master key

You might have to replace a cryptographic coprocessor without the need to also change the current master key (for example, when you want to upgrade to a new coprocessor model long before your security policies require a master key change).

About this task

With a few differences only, this is the same scenario as documented in <u>"Sharing master keys across</u> <u>cryptographic coprocessors</u>" on page 52. The new cryptographic coprocessor is in a way the same as one of the coprocessors sharing the same master key.

Important: Save the master key parts on one or more smart cards, because this facilitates the key management in many scenarios. If the master key is lost, there is no way to decrypt the data.

Procedure

1. Set the current master key from the old cryptographic coprocessor on the new coprocessor so that both coprocessors have the same master key.

2. Start to use the new cryptographic coprocessor when working with your associated volumes.

In case you are using a secure key stored in the secure key repository, and the secure key is associated with one or multiple APQNs, you should update the association using **zkey change** command with option --apqns. For details see <u>"Changing AES secure keys" on page 40</u>.

Replacing with a different master key

In this scenario, the old cryptographic coprocessor still exists. All the secure keys generated on this coprocessor were generated from random keys.

Before you begin

As an alternative to the procedure described here, you can create a new encrypted volume and migrate the data there (see <u>"Migrating to an encrypted LVM physical volume" on page 25</u> or <u>"Migrating data to a</u> new encrypted volume" on page 31.

Important: If you use the approach described in this task, be sure to have a backup of the data on your volume in case the system crashes or a media error occurs during the re-encryption. In contrast to the mentioned alternatives, this re-encryption works on the volume in-place.

Procedure

1. Generate a new secure key on the new cryptographic coprocessor using **zkey generate**.

Store the new secure key in the secure key repository.

To generate a secure key on a specific APQN, use option --apqns with the **zkey generate** command and specify the desired APQN. This is only possible when generating secure keys in the secure key repository. For example, enter:

```
# zkey generate --name new_secure_xtskey --keybits 256 --xts \
--apqns <card.domain>
```

Because the new secure key now also contains a new effective key, a next step for re-encrypting is required.

2. Use the **cryptsetup-reencrypt** command to re-encipher all LUKS2 volumes that use the old secure key, with the new secure key from the secure key repository.

The new LUKS2 volume key is stored in the LUKS2 header of the volumes at the end of the reencryption.

For example, enter:

cryptsetup-reencrypt /dev/mapper/disk<n> --cipher paes-xts-plain64 \
 --master-key-file /etc/zkey/repository/new_secure_xtskey.skey --key-size 1024

You can retrieve information about the just-created secure key file using the **zkey list** command. To limit the returned list to the desired secure key, use option - name to specify the secure key name. You find the secure key file name indicated by: **Key file name** (see <u>"Listing AES secure keys contained</u> in the secure key repository" on page 39).

Important: Ensure that you have a backup copy of the volume that you want to re-encrypt. In case the system crashes or a media error occurs during re-encryption, your original data might be destroyed.

3. Set the verification pattern into the LUKS2 header using the **zkey-cryptsetup** command.

Issue a command similar to the following:

zkey-cryptsetup setvp /dev/mapper/disk<n>

4. Remove the old cryptographic coprocessor and start using the new one.

If you previously stored the old secure key in the secure key repository, remove it now and associate the new secure key with the re-encrypted volume. For example, enter:

```
# zkey remove --name old_secure_xtskey
# zkey change --name new_secure_xtskey --volumes /dev/mapper/disk<n>:enc-disk<n> \
        --volume-type LUKS2 --sector-size 4096
```

Note: Ensure that the sector size for encryption or decryption of the volume is the same for the old and the new secure key. The example assumes that a sector size of 4096 bytes was used for encrypting or decrypting the volume with both the old and the new secure key.

For details see <u>"Removing AES secure keys" on page 39</u> and <u>"Changing AES secure keys" on page 40</u>.

Chapter 7. Problem resolution and recovery

The infrastructure for protected volume encryption offers several resources to assist you in cases when problems occur. You can query and verify information about your secure keys and you can utilize debug tools.

The following topics provide further details:

- "Verifying your configuration" on page 55
- "Troubleshooting problems in your environment" on page 57
- Chapter 8, "Recovering secure key encrypted volumes," on page 59

Verifying your configuration

You can use several commands for verifying certain aspects of your pervasive encryption configuration.

Checking required kernel modules

Secure key volume encryption requires the pkey and paes_s390 kernel modules (see also <u>"Prerequisites"</u> on page 9).

To check if these kernel modules are loaded, use the **1smod** command:

```
# lsmod | grep pkey
pkey 24576 1 paes_s390
zcrypt 69632 4 pkey,zcrypt_cex4
# lsmod | grep paes
paes_s390 16384 0
pkey 24576 1 paes_s390
```

If the modules are not loaded, use the **modprobe** to load them:

```
# modprobe pkey
# modprobe paes_s390
```

I

As the paes_s390 module requires the pkey module, pkey is also loaded together with the paes_s390 module by the shown command.

Checking available cryptographic coprocessors

Secure key volume encryption requires IBM Crypto Express5S or Crypto Express6S adapters in CCA coprocessor mode (CEX5C or CEX6C) or Crypto Express7S adapters in EP11 mode (CEX7P).

Use the **lszcrypt** command to list the available cryptographic coprocessors:

# lszcrypt CARD.DOMAIN	TYPE	MODE	STATUS	REQUEST_CNT
02	CEX5A	Accelerator	online	0
02.004c	CEX5A	Accelerator	online	0
03	CEX5C	CCA-Coproc	online	13000
03.004c	CEX5C	CCA-Coproc	online	13000
05	CEX5P	EP11-Coproc	online	81213
05.004c	CEX5P	EP11-Coproc	online	81213

For more details, refer to chapter *Generic cryptographic device driver* in *Device Drivers, Features, and Commands*, SC33-8411 available on the IBM Knowledge Center at

Valid physical block size combinations of LVM physical volumes

Before migrating data from an unencrypted volume to an encrypted LVM logical volume, check the physical block sizes of the involved volumes to ensure that you use only allowed block size combinations to avoid file system corruption and data loss. New LVM versions perform this check and reject actions that may corrupt the data.

If in your migration scenario the physical block size of a target physical volume (PV) is larger than the physical block size of the source PV, the file systems residing on logical volumes backed by the source PV may become corrupted. This is because the file system is aligned to the physical block size of the source PV and cannot be mapped to a larger block size. Therefore, never use a target volume with a larger block size than your source volumes.

The **--sector-size** parameter of a dm-crypt volume influences the physical block size. A dm-crypt volume encrypted with a sector size larger than the default 512 bytes results in a device with a physical block size of either the used sector size or the block size of the underlying device, whatever is higher.

This is the case for LUKS2 as well as for a plain mode dm-crypt volume. LUKS1 only supports the default block size of 512 bytes.

Migrating data from an unencrypted PV onto a dm-crypt encrypted PV might thus corrupt the file systems on the logical volumes backed by the source PV if the target dm-crypt volume uses a sector size of 4096 bytes, and the unencrypted source PV uses a physical block size of 512 bytes.

To query the physical block size of a device, use the following command:

blockdev --getpbsz <device>

If the file systems on the affected LVs were created with a 4096 block size, then the problem does not occur. The file system may choose different block sizes based on various parameters. Thus, on larger volumes, it is very likely that the file system block size is 4096 by default.

<u>Table 1 on page 56</u> shows combinations of source and target physical volumes where migration can be performed. <u>Table 2 on page 57</u> shows combinations of source and target physical volumes where migration might corrupt the data on the source volumes.

Note: The tables offer a selection of possible combinations. There might be numerous other combinations available in your environment. The general rule is that you should never extend an existing LVM volume group with a device that has a larger physical block size than the existing physical volumes of the volume group.

Table 1. Allowed combinations		
Source physical volume	Target physical volume	
Comparison of block sizes		
physical block size = n (for example, n = 4096)	physical block size = n (for example, n = 4096)	
physical block size = n	physical block size < n	
Comparison of block sizes on source PV and sector sizes on target dm-crypt volume		
physical block size = 512	dm-crypt volume with sector-size = 512 (default)	
physical block size = 4096	dm-crypt volume with sector-size = 512 (default)	
physical block size = 4096	dm-crypt volume with sector-size = 4096	
Comparison of block sizes of specific device types (selection)		

Table 1. Allowed combinations (continued)			
Source physical volume	Target physical volume		
SCSI-disk, SCSI-partition, SCSI-multipath-device with block size = 512	dm-crypt volume with sector-size = 512 (default)		
DASD-partition (4096)	dm-crypt volume with sector-size = 512 (default)		
DASD-partition (4096)	dm-crypt volume with sector-size = 4096		
Loopback-device (512)	dm-crypt volume with sector-size = 512 (default)		
Other device mapper device (512)	dm-crypt volume with sector-size = 512 (default)		
Other device mapper device with physical block size > 512	dm-crypt volume with sector-size = 512 (default)		

Table 2. Combinations of physical block sizes that might lead to data corruption

Source physical volume	Target physical volume
physical block size = n (for example, n = 512)	physical block size > n (for example, 4096)
physical block size = 512	dm-crypt volume with sector-size = 4096
SCSI-disk, SCSI-partition, SCSI-multipath-device with block size = 512	dm-crypt volume with sector-size = 4096
Loopback-device (512)	dm-crypt volume with sector-size = 4096
Other device mapper device with physical block size 512 < n < 4096	dm-crypt volume with sector-size = 4096

Troubleshooting problems in your environment

For an efficient troubleshooting, you can access various resources for debugging problems with the components of the infrastructure for protected volume encryption.

Debugging zkey problems

If the **zkey** utility encounters an error, you can use the **--verbose** option of the utility to show additional messages.

Especially for the **reencipher** command of **zkey**, the **--verbose** option displays the CCA return and reason codes of the CSNBKTC (Key Token Change) verb used by **zkey**. CCA return and reason codes are documented in *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*, SC33-8294.

For problems when working with EP11 AES secure keys, refer to *Exploiting Enterprise PKCS #11 using openCryptoki*.

Debugging zkey-cryptsetup problems

If the **zkey-cryptsetup** utility encounters an error, you can use the **--verbose** option or the **--debug** option of the utility to show additional messages. The **--debug** option also causes debug messages from **libcryptsetup** to be issued. The **zkey-cryptsetup** command uses the functions provided by **cryptsetup**.

Especially for the **reencipher** command of **zkey-cryptsetup**, the **--verbose** option displays the CCA return and reason codes of the CSNBKTC (Key Token Change) verb used by **zkey-cryptsetup**. CCA

return and reason codes are documented in *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*, SC33-8294.

For problems when working with EP11 AES secure keys, refer to *Exploiting Enterprise PKCS #11 using openCryptoki*.

Debugging cryptsetup problems

In case **cryptsetup** encounters an error, check the syslog for additional messages from the device mapper and dm-crypt.

In addition, use option **--debug** to see additional debugging messages. These messages show you where the error occurred and help you to find the reason of the failure.

The same debugging procedure is valid for problems with the cryptsetup-reencrypt command.

Debugging the pkey kernel module

The pkey driver uses the s390 debug feature. Usually the debugfs file system is mounted at /sys/ kernel/debug and the **cryptsetup** driver appears as own directory at /sys/kernel/debug/ s390dbf/pkey. By default the **dbf** level is set to 3 and all error messages within the driver are valued to 3 also. Thus any error message can be extracted as ASCII text by reading from the sprintf pseudo file:

cat /sys/kernel/debug/s390dbf/pkey/sprintf

In order to get debug messages, too, you can set the **dbf** level to 6:

echo 6 >/sys/kernel/debug/s390dbf/pkey/level

Out-of-memory errors when opening a LUKS2 volume

You might encounter an out-of-memory error when opening a LUKS2 volume either during manual opening or during automated opening via /etc/crypttab at system startup. This is most probably caused by the fact that the LUKS2 format by default uses the Argon2i key derivation function, which is a so-called memory-hard function. It requires a certain amount of physical memory to make dictionary attacks more costly. To reduce the required amount of memory, the following steps can be helpful:

- Use LUKS2 with PBKDF2 instead of Argon2i as the key derivation function. Add the --pbkdf pbkdf2 option when using the **luksFormat** or **luksAddKey** commands.
- Decrease the amount of memory for Argon2i functions. For example, to use up to 256 KB, add the -- pbkdf-memory 256 option to the **luksFormat** or **luksAddKey** commands.

Since the infrastructure for protected volume encryption uses secure keys as volume keys, the security of the key derivation function used to derive the key to encrypt the volume key in the LUKS key slots is of less relevance.

In order to check which key derivation function is used by an encrypted disk, use **cryptsetup luksDump**. It displays the key derivation function for each key slot. To change the key derivation function for an existing key slot, use **cryptsetup luksConvertKey**. Refer to the cryptsetup man page for more details.

Chapter 8. Recovering secure key encrypted volumes

There is no way to recover data encrypted with a secure key, if the corresponding master key is lost. Therefore, keep the master key at a very safe and secure place outside the HSM. If the used secure key is lost, you can recover your data only if certain prerequisites are fulfilled.

The following scenarios are discussed:

- "Recovering encrypted volumes from an invalid secure key" on page 59
- "Recovering encrypted volumes with a secure key from the repository" on page 60

Recovering encrypted volumes from an invalid secure key

This recovery scenario is only possible if the clear key that was used to generate the secure key is still available in a binary file. This can be the case in environments where you intentionally want to save the clear key due to your security policy. Use this approach only if you kept your clear key absolutely safe in a clean room environment, where it is protected against being exposed to non-authorized persons. A secure key that contains an encrypted insecure clear key is of no added value compared to the insecure clear key.

Procedure

1. Generate a new secure key from a binary clear key input file on the new cryptographic coprocessor. Use the **zkey generate** command with option --clearkey.

Store the new secure key in the secure key repository.

To generate a secure key on a specific APQN, use option - - apqns with the **zkey generate** command and specify the desired APQN. This is only possible when generating secure keys in the secure key repository. Also, you can already associate the affected volumes with the new secure key. For example, enter:

1

```
# zkey generate --name new_secure_xtskey --clearkey old_clearkey.bin --key-type CCA-AESCIPHER \
    --xts --apqns <card.domain> --volumes /dev/mapper/disk<n>:enc-disk<n> \
    --volume-type LUKS2 --sector-size 4096
```

Because the new secure key contains the same old clear key, the volume needs not be re-encrypted, but the new created secure key must be set as the LUKS volume key.

2. Use the zkey-cryptsetup setkey command to set the new LUKS2 volume key.

For example, enter:

```
# zkey-cryptsetup setkey /dev/mapper/disk<n> \
--master-key-file /etc/zkey/repository/new_secure_xtskey.skey
Enter passphrase for '/dev/mapper/disk<n>': disk<n>pw
```

You can retrieve information about the secure key file using the **zkey list** command. To limit the returned list to the desired secure key, use option - - name to specify the secure key name. You find the secure key file name indicated by: **Key file name** (see <u>"Listing AES secure keys contained in the</u> secure key repository" on page 39).

If you had set a verification pattern into the LUKS2 header of the volume using the **zkey-cryptsetup setvp** command, this pattern is used to ensure that the new volume key contains the same clear key. If no verification pattern is available, then you are prompted to confirm that the specified secure key is the correct one.

Important: If you set an incorrect secure key you will lose all the data on the encrypted volume.

For details see Appendix B, "zkey-cryptsetup - Managing LUKS2 volume keys," on page 85.

3. Reset the pbkdf option to use PBKDF2

This step is not required when using the **zkey-cryptsetup** tool from an <u>s390-tools</u> package upstream version 2.9.0 or later.

Setting a new LUKS2 volume key with **zkey-cryptsetup** sets the value of option pbkdf of the key slot with the interactive passphrase to the default Argon2i password-based key derivation function (PBKDF), no matter which value you had set before.

As described in <u>"Out-of-memory errors when opening a LUKS2 volume" on page 58</u>, the use of the Argon2i key derivation function may cause an out-of-memory error when opening a LUKS2 volume. To avoid such an error during automatic unlocking of the encrypted volume at system startup, change the value of option pbkdf to use the PBKDF2 key derivation function:

```
# cryptsetup luksConvertKey --pbkdf pbkdf2 /dev/mapper/disk<n>
Enter passphrase for keyslot to be converted:
disk<n>pw
```

4. Remove the old cryptographic coprocessor and start using the new one.

If you had stored the old secure key in the secure key repository, remove it now. For example, enter:

zkey remove --name old_secure_xtskey

For details see "Removing AES secure keys" on page 39 and "Changing AES secure keys" on page 40.

Results

Your affected volumes are ready for use, exploiting the new cryptographic coprocessor.

Recovering encrypted volumes with a secure key from the repository

Read about the procedure how to recover encrypted data from a volume whose volume key is no longer valid (for example, if the volume has been archived a long time ago).

About this task

Recovering a secure key encrypted partition whose volume key is no longer valid is only possible if a valid copy of this key is available in the secure key repository. This copy of the secure key must have been reenciphered in the repository each time the master key changed on the cryptographic coprocessor.

Procedure

1. Optional: Ensure that the secure key in the repository matches the volume key in the LUKS2 header.

This action is only possible if a verification pattern was previously set into the LUKS2 header. To check, enter the following commands and compare the verification patterns. The **zkey-cryptsetup validate** command checks the validity of the volume key on the volume and displays its verification pattern. The **zkey validate** command checks the validity of the specified secure key in the repository and also displays its verification pattern.

```
# zkey-cryptsetup validate /dev/mapper/disk<n>
Validation of secure volume key of device '/dev/mapper/disk<n>':
  Status:272 bytesSecure key size:272 bytesXTS type key:YesKey type:CCA-AESCIPHERClear key size:(unknown)Enciphered with:(unknown) (MKVP: 880f72e3759217e7Verification pattern:477a8608f06743569e2c62fc5fe0008508b18a80d7616094eceaa746be4a2edd
    Status: Invalid
Secure key size: 272 bytes
                                                                         (MKVP: 880f72e3759217e7)
ATTENTION: The secure volume key is not valid.
# zkey validate --name <name of key>
                                                   : <name of key>
Key
Status
                                                             : Valid
               Status

Description :

Secure key size : 272 bytes

Clear key size : 512 bits

XTS type key : Yes

Key type: CCA-AESCIPHER

Enciphered with : CURRENT master key (MKVP: 26d69731a66f4255)

Volumes :/dev/mapper/disk<n>:enc-disk<n>

APQNs : 03.0039

04.0039

      Nights
      .00309

      04.0039

      Key file name
      : /etc/zkey/repository/<name_of_key>.skey

      Sector size
      : (system default)

      Volume type
      : LUKS2

      Verification pattern
      : 477a8608f06743569e2c62fc5fe00085

      Other type
      : 0039

                O8b18a8Created: 2018-08Changed: (never)Re-enciphered: (never)
                                                              08b18a80d7616094eceaa746be4a2edd
                                                             : 2018-08-21 17:19:53
```

If you omit this step, and the keys mismatch, the **zkey-cryptsetup setkey** from step <u>"2" on page</u> 61 will reject to set the key, provided that the verification pattern is available in the LUKS2 header.

2. Set the secure key from the secure key repository as the new volume key.

Enter the following command:

| | |

I

```
# zkey-cryptsetup setkey /dev/mapper/disk<n> \
--master-key-file /etc/zkey/repository/<name_of_key>.skey
Enter passphrase for '/dev/mapper/disk<n>': disk<n>pw
```

The verification pattern is used to ensure that the new volume key contains the same effective key as the old volume key. If no verification pattern is available, then you are prompted to confirm that the specified secure key is the correct one.

Important: If you set an incorrect secure key you will lose all the data on the encrypted volume.

3. Reset the pbkdf option to use PBKDF2

This step is not required when using the **zkey-cryptsetup** tool from an <u>s390-tools</u> package upstream version 2.9.0 or later.

Setting a new LUKS2 volume key with **zkey-cryptsetup** sets the value of option pbkdf of the key slot with the interactive passphrase to the default Argon2i password-based key derivation function (PBKDF), no matter which value you had set before.

As described in <u>"Out-of-memory errors when opening a LUKS2 volume" on page 58</u>, the use of the Argon2i key derivation function may cause an out-of-memory error when opening a LUKS2 volume. To avoid such an error during automatic unlocking of the encrypted volume at system startup, change the value of option pbkdf to use the PBKDF2 key derivation function:

```
# cryptsetup luksConvertKey --pbkdf pbkdf2 /dev/mapper/disk<n>
Enter passphrase for keyslot to be converted:
disk<n>pw
```

62 Linux on Z and LinuxONE: Pervasive Encryption for Data VolumesSeptember 2020

Chapter 9. Encrypting volumes without LUKS

In an environment where you do not want or cannot use encrypted volumes formatted with LUKS2, you can use encrypted volumes in plain mode as an alternative. This way, you can exploit the features of the infrastructure for protected volume encryption in the **cryptsetup** plain mode as described in the contained subtopics.

In plain mode, the keys used to open the volume are not protected by a passphrase in contrast to LUKS. In the infrastructure for protected volume encryption, these keys are secure keys and therefore do not need an additonal encryption using a passphrase. Secure keys are usable only on systems that have access to a cryptographic coprocessor with the correct master key.

In plain mode, you can also use the **zkey** utility to manage a secure key repository that helps you to work with encrypted volumes in plain mode. It allows to associate secure keys with volumes and knows the volume type. Therefore, it can generate the required commands to open a plain-mode volume for you.

The following topics are discussed:

- "Volume encryption with cryptsetup plain mode" on page 63
- "Encrypting an unencrypted volume using plain mode" on page 65
- "Changing a master key using plain mode" on page 65
- "Opening an encrypted volume in plain mode" on page 66

Volume encryption with cryptsetup plain mode

Setting up volume encryption using **cryptsetup** plain mode entails generating secure keys and creating logical volumes.

Before you begin

Make sure that the software prerequisites are met as described in <u>"Software prerequisites" on page 9</u>. Note that the plain mode also works with **cryptsetup** versions prior to 2.0.3.

Based on the sample system environment as shown in Figure 5 on page 16, the procedure documented here uses the first partition on a multipath SCSI disk /dev/mapper/disk1.

About this task

It is of advantage to store the secure keys in the secure key repository as shown in this procedure. This enables you to also open archived volumes (for example, provided you have re-enciphered these keys with each master key change). For more information, refer to <u>"Managing a secure key repository" on page</u> 35.

Procedure

I

1. Use the **zkey** utility to generate a secure AES DATA key in a secure key repository.

Issue the following command using the XTS cipher mode:

```
# zkey generate --name secure_xtskey1 --key-type CCA-AESDATA --keybits 256 --xts \
--volumes /dev/mapper/disk1:enc-disk1 --volume-type PLAIN \
--apqns 03.0039,04.0039
```

In the example, the generated secure key is of type AES DATA (--key-type CCA-AESDATA) and is stored in the secure key repository. Secure keys of types AES CIPHER and EP11 AES are also supported in plain mode.

The key to be wrapped is generated by random inside the cryptographic coprocessor and is thus never exposed in clear.

The --key-type

You can have a secure key per volume or share a secure key among volumes. In the previous example, the secure key named secure_xtskey1 is associated with volume /dev/mapper/disk1 in plain mode and uses the device mapper name enc-disk1.

Do not encrypt multiple volumes with the same key. If, for certain reasons, you want to do this nevertheless, then you can specify the **--volumes** parameter as shown in this example:

--volumes /dev/mapper/disk1:enc-disk1,/dev/mapper/disk2:enc-disk2,...

This associates all listed volumes with the same secure key.

Note: You can also specify the **--sector-size** parameter with the **plainOpen** command. However, automatic opening of plain mode volumes during system startup might not work, depending on the used **systemd** version. Ensure that **systemd** supports the sector size option in /etc/crypttab before you create plain-mode encrypted volumes with a sector size different than the default (512 bytes).

2. Use **zkey cryptsetup** to generate the command for creating an encrypted logical volume in plain format.

```
# zkey cryptsetup --volumes /dev/mapper/disk1
cryptsetup plainOpen --key-file '/etc/zkey/repository/secure_xtskey1.skey'
--key-size 1024 --cipher paes-xts-plain64 /dev/mapper/disk1 enc-disk1
```

In the generated **cryptsetup** command, **plainOpen** is used to open the volume and to assign a logical volume name to the opened volume. The new logical volume is created in /dev/mapper.

The generated **plainOpen** command specifies:

- The location and name of the secure key file.
- The key size (in bits). For cipher mode XTS, the size of an AES DATA key is 128*8=1024.
- The PAES cipher and its operation mode (in the example, XTS).
- The name of the volume.
- A name of your choice for the logical volume.
- 3. Run the generated command.

Either copy and paste the generated command into the command line or you use the --run option to execute it:

```
# zkey cryptsetup --volumes /dev/mapper/disk1 --run
Executing: cryptsetup plainOpen --key-file '/etc/zkey/repository/secure_xtskey1.skey'
--key-size 1024 --cipher paes-xts-plain64 /dev/mapper/disk1 enc-disk1
```

You can check the result of this step with the command **ls** /dev/mapper/. Any I/O operation to or from /dev/mapper/enc-disk1 is then transparently encrypted or decrypted onto the /dev/mapper/disk1 volume. As of now, do not write to this volume directly.

4. Open the volume during the system startup.

Use the **zkey crypttab** command to generate an entry in /etc/crypttab to persistently configure an opening during system startup.

Copy the generated crypttab entry into file /etc/crypttab to configure unlocking during system startup. The generated output must be in one line. Each line describes an encrypted volume and assigns the secure key to be used for encryption and decryption of the volume:

```
# /etc/crypttab
#
# See crypttab(5) for more information.
#
#
#
#
#
# Target Source device Key file Options
enc-disk1 /dev/mapper/disk1 /etc/zkey/repository/secure_xtskey1.skey plain,cipher=paes-xts-plain64,
size=1024,hash=plain
```

The format of the /etc/crypttab file depends on your Linux distribution. See the crypttab man page for more details.

Note: The /etc/crypttab file might not be located on an encrypted volume.

What to do next

Once you have opened an encrypted logical volume either with the **cryptsetup** command (step <u>"2" on</u> page 64), or implicitly during system startup (step <u>"4" on page 64</u>), you can use the opened volume /dev/mapper/enc-disk1 like any other block device. Typical next steps are:

- If you want to manage your encrypted volumes using LVM, create LVM physical volumes and add them to an LVM volume group.
- Create a file system on the encrypted logical volume.
- Create a mount point and update /etc/fstab to later mount the file system on the encrypted logical volume or LVM logical volume.

Encrypting an unencrypted volume using plain mode

If you want to integrate unencrypted data residing on a volume into the infrastructure for protected volume encryption using plain mode, you need to perform the task to transform an unencrypted partition into an encrypted one.

This topic presents two methods with which you can achieve this task:

- 1. For LVM physical volumes, you can use the **pvmove** LVM command. Refer to the procedure described in <u>"Migrating to an encrypted LVM physical volume" on page 25</u> and perform the steps according to plain mode.
- 2. You can copy existing content to a new encrypted volume in plain mode and delete the original data. Refer to the procedure described in <u>"Migrating data to a new encrypted volume" on page 31</u> and perform the steps according to plain mode.

Note: An encrypted volume in plain mode does not contain a LUKS header, thus the full size of the volume is available for use.

After you have migrated the data from the unencrypted volume to the encrypted one, be sure to securely delete any unencrypted data according to your security policies. For example, you can use **badblocks** or **shred** to overwrite unencrypted data with random data multiple times.

Changing a master key using plain mode

In plain mode, you must specify the secure key when opening the volume. For a master key change, you must re-encipher these secure keys with the new master key.

If the secure keys are stored in a file or in a secure key repository, you can use the **zkey** utility to perform the re-enciphering. Read <u>"Re-enciphering secure keys from a file" on page 51</u> or <u>"Re-enciphering secure keys from a repository"</u> on page 48 for more information.

Opening an encrypted volume in plain mode

The need to open an encrypted volume can occur during normal runtime or during Linux startup. Special processing is required if the volume is required early in the system startup process (for example, if it is part of an LVM volume group on which the root file system resides).

Automatically opening encrypted volumes in plain mode at Linux system startup

Automatically opening one or more volumes at Linux startup allows you to perform automated reboots.

For each encrypted volume that is required during the Linux startup, yon need to edit /etc/crypttab. Add an entry for each required volume. For examples of valid /etc/crypttab entries, read the information from step "4" on page 64 in "Volume encryption with cryptsetup plain mode " on page 63.

Opening and mounting an encrypted volume at user login in plain mode

Automatically opening one or more partitions at user login has the advantage that only a certain user can access the data.

Before you begin

You must install the *pam_mount* package. See the web site at <u>http://pam-mount.sourceforge.net/</u>. Some Linux distributions provide a *pam_mount* package.

Ensure that the pam_mount package is configured and the pam_mount.so PAM module is used in the auth and session sections of the PAM configuration files. Your Linux distribution might already perform this for you. See also the pam_mount man page for more information.

About this task

In this scenario you create a user called alice. The home directory for this user is stored on an encrypted volume. The encrypted volume is opened when the user logs in and is respectively closed at logout. The encrypted volume in this example is /dev/mapper/disk10.

Procedure

1. Create a user and set an initial password.

For example, issue:

```
# useradd -G users -m -s /bin/bash alice
# passwd alice
Enter new UNIX password: alice
Retype new UNIX password: alice
passwd: password updated successfully
```

2. Create a secure key in the secure key repository for user alice with the **zkey** command.

```
# zkey generate --name user-alice-xts --keybits 256 --xts \
--volumes /dev/mapper/disk10:user-enc-alice --volume-type PLAIN \
--apqns 03.0039,04.0039
```

Note: Do not use the **--sector-size** parameter here, because pam_mount does not support sector sizes other than the default (512 bytes).

3. Format and open the encrypted volume, /dev/mapper/disk10, and create a file system that is later mounted as home directory for user alice.

For example:
```
# zkey cryptsetup --volumes /dev/mapper/disk10 --run
Executing: cryptsetup plainOpen --key-file '/etc/zkey/repository/user-alice-xts.skey'
--key-size 1024 --cipher paes-xts-plain64 /dev/mapper/disk10 user-enc-alice
# mkfs.ext4 -L USER_ALICE /dev/mapper/user-enc-alice
# cryptsetup close user-enc-alice
```

You can optionally mount the file system temporarily to copy or migrate existing files for the user.

4. Edit the pam_mount configuration file /etc/security/pam_mount.conf.xml. Add a volume definition for alice.

```
<volume user="alice" path="/dev/mapper/disk10" mountpoint="~"
    fstype="crypt" fskeycipher="none"
    fskeypath="/etc/zkey/repository/user-alice-xts.skey"
    cipher="paes-xts-plain64" fskeyhash="plain"/>
```

See also the *pam_mount.conf* man page for details.

Results

Now alice can log in to the Linux instance. The pam_mount PAM module opens the encrypted volume and creates a device under /dev/mapper/ (for example, /dev/mapper/_dev_dm_21) which is then mounted as /home/alice.

```
# ssh alice@localhost
alice@localhost's password: alice
Welcome to your favourite Linux distribution
Last login: Mon Aug 06 16:28:45 2018 from 127.0.0.1
alice@localhost:~$ df | grep alice
/dev/mapper/_dev_dm_21 20507216 45080 19397384 1% /home/alice
alice@localhost:~$
```

68 Linux on Z and LinuxONE: Pervasive Encryption for Data VolumesSeptember 2020

Chapter 10. Encrypting swap disks with protected keys

Within the infrastructure for protected volume encryption, you can generate random protected AES keys without requiring a cryptographic coprocessor. Use these keys for encrypting swap disks, or for other use cases, where keys may be ephemeral.

You can generate volatile protected keys from random data without requiring a cryptographic coprocessor in two ways:

- A program or tool can read from one of the binary read-only sysfs attributes which are located in the /sys/devices/virtual/misc/pkey/protkey directory. Each time such an attribute is read, a new random AES protected-key token of the corresponding format is returned. Refer to the applicable *Device Drivers, Features, and Commands* for information about available key token formats.
- You can issue ioctl calls on the misc character device /dev/pkey to generate and handle protected keys, for example, PKEY_GENPROTKEY. Refer to the applicable *Device Drivers, Features, and Commands* for more information about available ioctl calls.

During the generation process, the underlying effective key is never exposed in clear in memory. The paes_s390 kernel module can use these protected keys in the same way as a protected key derived from a secure key.

This feature is mainly useful for encrypting swap disks, or for any other use cases where the keys may be ephemeral, that means, that their life time does not extend over different boot cycles or machine migrations.

Important: The protected key is volatile and cannot be recreated if lost, for example during a reboot. Do not use protected keys that are generated from random data to encrypt persistent data. Use such a protected key only to protect transient data. Especially, KVM guest migration, z/VM live guest relocation in a single system image (SSI), or suspend or resume actions are not supported with such randomly generated protected keys.

If you set up your environment as described in <u>"Setting up an encrypted swap disk" on page 69</u>, a volatile random protected key is automatically generated to be used for swap disks.

Setting up an encrypted swap disk

You can use a volatile protected key generated by the pkey device driver to encrypt a swap disk.

About this task

Because swap disks are discarded on reboot, volatile encryption keys are an option. You can generate volatile protected keys or secure keys from random data.

Important: Use a protected key based on random data only for cases where the key is not needed after a reboot. In particular, do not use such a key with:

- KVM guest migration
- z/VM live guest relocation in a single system image (SSI)
- Suspend and resume

Procedure

1. Add an entry to /etc/crypttab.

To encrypt the swap device using a protected key, the entry must point to one of the sysfs attributes within the /sys/devices/virtual/misc/pkey/protkey/ directory. Use the attribute for the required key type (see <u>Chapter 10</u>, "Encrypting swap disks with protected keys," on page 69).

For example:

∦ <name> swap_disk</name>	<device> /dev/mapper/disk99</device>	<pre><password> /sys/devices/virtual/misc/pkey/protkey/protkey_aes_256_xts</password></pre>	<pre><options> swap,\ cipher=paes-xts-plain64,\ size=1280</options></pre>
-------------------------------	--	---	---

The entry must be all in one line without continuation characters.

The swap option causes an **mkswap** command to be performed after the dm-crypt volume is set up.

Tip: Consider adding the sector-size=4096 option to increase the performance of dm-crypt encrypted disks with large block sizes.

2. Add an entry to /etc/fstab to use the device-mapper device named swap_disk as swap device: For example:

<pre># <filesystem></filesystem></pre>	<dir></dir>	<type></type>	<options></options>	<dump></dump>	<pass></pass>
/dev/mapper/swap_disk	none	swap	defaults	0	0

3. Ensure that the pkey kernel module is loaded during system startup before /etc/crypttab is evaluated.

Ensure that a configuration file such as pkey.conf or modules.conf is in the .../modules-load.d/ directory. The configuration file must contain:

pkey

The .conf file(s) in .../modules-load.d/ contain the modules to be loaded early during startup, before the swap disk is initiated.

Results

During system startup, /etc/crypttab is evaluated, and a dm-crypt device is set up in plain mode as a swap device, using an AES protected key in XTS cipher mode. The random protected AES key is read from /sys/devices/virtual/misc/pkey/protkey/protkey_aes_256_xts. Its size is 2x80 bytes, which is 1280 bits.

The swap option causes that an mkswap is performed after the dm-crypt volume has been set up. The entry in /etc/fstab then causes the device-mapper device named swap_disk to be used as swap device.

Linux now runs with a swap device that is encrypted with a protected key.

What to do next

For reasons of security, you might consider to use a secure key instead of a protected key for encrypting swap disks. In such a case, you can generate a new random secure key from a cryptographic coprocessor using another set of sysfs attributes. You do not need to store and manage the secure key within the secure key repository, because you want to generate a new secure key at boot time, that is, each time you use a new swap disk.

A new random secure key is generated by a cryptographic coprocessor when reading from the ccadata_aes_256_xts attribute in the /sys/devices/virtual/misc/pkey/ccdata/ directory.

Add an entry to /etc/crypttab, similar to the one for protected keys, but now using a sysfs attribute from the .../ccadata/ directory.

‡ <name></name>	<device></device>	<pre><password></password></pre>	<options></options>
swap_disk	/dev/mapper/disk99	/sys/devices/virtual/misc/pkey/ccadata/ccadata_aes_256_xts	swap,\
			cipher=paes-xts-plain64,\
			size=1024

The required entry to /etc/fstab is the same as for protected keys.

Appendix A. zkey - Managing secure keys

Use the **zkey** command to generate, validate, and re-encipher secure AES keys for Crypto Express CCA and EP11 coprocessors.

The **zkey** command is used in the context of a cryptographic domain of a Crypto Express adapter in CCA coprocessor mode or in EP11 mode. At least one domain must have an AES master key configured. Such a domain of a cryptographic coprocessor maintains three registers for master keys: The OLD, the NEW, and the CURRENT. In the following description, OLD, NEW, and CURRENT refer to the master key registers of the applicable domain.

Note: A cryptographic coprocessor configured in EP11 mode does not contain an OLD master key register.

You can either manage your keys as key files in the file system or use a secure key repository. You can set up the secure key repository with the environment variable ZKEY_REPOSITORY. The default repository is /etc/zkey/repository.

Note: If multiple domains on potentially multiple CCA or EP11 coprocessors coprocessors are available, the **zkey** command may use any of those domains, and therefore, coprocessors.



I

Attention: Handle your cryptographic keys with care. The loss of a key can result in data loss.

Prerequisites

- The **zkey** command requires access to a cryptographic domain that holds an AES master key.
- The **zkey** command requires the pkey kernel module. For more information, see chapter *Protected key device driver* in *Device Drivers, Features, and Commands*, SC33-8411 available on the IBM Knowledge Center at

www.ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_lib.html.

• The **reencipher** subcommand requires the CCA host library (libcsulcca.so) to be installed for secure keys of type CCA-AESDATA or CCA-AESCIPHER. This library is part of the CCA software package.

The **reencipher** subcommand requires the Linux on Z Enterprise PKCS #11 (EP11) Support Program (EP11 host library) for secure keys of type EP11-AES to be installed.

To download the required packages, go to <u>www.ibm.com/security/cryptocards</u> and proceed to the software download page for your IBM cryptographic coprocessor version.



where

subcommand

is described in the following sections:

- "Generating a secure key" on page 72
- "Validating secure keys" on page 74
- "Re-encipher secure keys" on page 75
- "Import a secure key into the the secure key repository" on page 76
- "Export a secure key from the secure key repository" on page 77
- "List secure keys in the secure key repository" on page 78
- "Delete secure keys from the secure key repository" on page 79

- "Change the properties of secure keys in the secure key repository" on page 79
- "Rename a secure key in the secure key repository" on page 80
- "Copy a secure key in the secure key repository" on page 81
- "Generate a crypttab entry for a volume" on page 81
- "Generate a cryptsetup command for a volume" on page 82
- "Convert an AES DATA key into an AES CIPHER key" on page 84

-V or --verbose

displays additional information during processing.

-h or --help

I

displays help information for the command. Specify zkey subcommand -h to get help for a subcommand.

-v or --version

displays the version number of **zkey**.

Generating a secure key

zkey syntax - generating a secure key → zkey → gen -N <name> -d <text> -d <text> -t <volume-type> gen <secure-key-file> -K <CCA-AESDATA -K <text> -K <text>

where

gen or generate

generates a new secure AES key either randomly within the cryptographic coprocessor, or from a clear AES key specified as input. You can store the key in a file or in the secure key repository.

-N <name> or --name <name>

specifies the name of the secure key. The key is generated into the secure key repository. The name can contain any of the following characters:

0-9 A-Z a-z ! @ # \$ & ^ ~ () = +

-t <volume-type> or --volume-type <volume-type>

Optional. Specifies the volume type of the associated volumes used with dm-crypt. Possible values are PLAIN or LUKS2 (not case-sensitive). If omitted, LUKS2 is used. This option is only available if **zkey** has been compiled with LUKS2 support enabled. If LUKS2 support is not enabled, the default volume type is plain.

-d <text> or --description <text>

Optional. Describes the secure key.

-l <volumes> or --volumes <volumes>

Optional. You can associate volumes with a key. Each volume association specifies the name of the block device (for example /dev/mapper/disk1) and the device mapper name separated by a colon. Separate multiple volume associations with a comma, for example:

-l /dev/mapper/disk1:enc-disk1,/dev/mapper/disk2:enc-disk2

-a <AP queues> or --apqns <AP queues>

Optional. You can associate AP queue numbers with a key. Each AP queue association specifies a device number and domain separated by a period. Separate multiple AP queue associations with a comma, for example:

When at least one APQN is specified, then the first one is used to generate the key. If no APQNs are specified, then an APQN is selected automatically.

All specified APQNs must be online, unless the --no-apqn-check option is specified.

--no-apqn-check

Optional. Disable checking whether the specified APQNs are available and have the same master key. With this option, a currently non-existing APQN can be associated with a secure key. This is useful to associate APQNs that exist only on other systems, such as disaster recovery systems, but not on the current system. When generating keys, at least one of the specified APQNs must be available to generate the key.

-S <bytes> or --sector-size <bytes>

Optional. Specifies the sector size in bytes used with dm-crypt. It must be a power of two in the range 512 - 4096 bytes. The default sector size is 512 bytes.

<secure-key-file>

specifies the name of the file that holds the secure key if you are not using a repository.

-K <key-type> or --key-type <key-type>

Optional. Specifies the type of the secure key. To generate an AES DATA key, specify CCA-AESDATA, to generate an AES CIPHER key, specify CCA-AESCIPHER. To generate an EP11 AES secure key for use on cryptographic coprocessors configured in EP11 mode, specify EP11-AES. If this parameter is omitted, then an AES DATA key (CCA-AESDATA) is generated as default.

This applies to both, keys stored in the secure key repository, as well as keys stored in a file outside of the secure key repository.

-k or --keybits <size>

specifies the size of the AES key to be generated in bits. Valid sizes are 128, 192, and 256 bits. Secure keys for use with the XTS cipher mode can only use keys of 128 or 256 bits. The default is 256 bits.

-x or --xts

generates a secure AES key for the XTS cipher mode. A secure AES key for the XTS cipher mode consist of two concatenated secure keys.

-c or --clearkey <clear-key-file>

specifies a file path that contains the clear AES key in binary form. If the --keybits option is omitted, the size of the specified file determines the size of the AES key. If the --keybits option is specified, the size of the specified file must match the specified key size. Valid file sizes are 16, 24, or 32 bytes, and for the XTS cipher mode 32 or 64 bytes.

Examples

L

• To generate a 256-bit secure AES DATA key and store it in file seckey.bin:

zkey generate seckey.bin

• To generate a 128-bit secure AES CIPHER key for the XTS cipher mode and store it in file seckey.bin:

zkey generate seckey.bin --keybits 128 --xts -K CCA-AESCIPHER

• To generate a secure AES DATA key from the clear key in file clearkey.bin and store it in file seckey.bin:

zkey generate seckey.bin --clearkey clearkey.bin

 To generate a random 256-bit secure AES DATA key and store it in the secure key repository under the name seckey:

zkey generate --name seckey

 To generate a random 256-bit secure AES DATA key, store it in thesecure key repository under the name seckey, and associate it with block device /dev/mapper/disk1 and device-mapper name enc-disk1, and associate it with AP queue 03.004c:

zkey generate --name seckey --volumes /dev/mapper/disk1:enc-disk1 --apqns 03.004c

Validating secure keys



where

L

val or validate

checks whether the specified file contains a valid secure key. It also displays the attributes of the secure key, such as key size, whether it is a secure key that can be used for XTS cipher mode, and the master key register with which the secure key is enciphered.

<secure-key-file>

specifies the name of the file that holds the secure key if you are not using the secure key repository.

-N <name> or --name <key name>

specifies the key name. You can validate a specific key by specifying its name, or you can use wildcards to match keys to be validated. If no key name is specified, all keys in the secure key repository are validated.

-a <AP queues> or --apqns <AP queues>

Optional. Validate those secure keys which are associated with the specified cryptographic adapters (APQNs). You can use wildcards for the APQN specification. When wildcards are used you must quote the value. If both option - - name and option - - apqns are specified then all secure keys contained in the secure key repository that match both patterns are validated.

--no-apqn-check

Optional. Disable checking whether the specified APQNs are available.

Examples

• To validate the secure key in seckey.bin and display its attributes:

zkey validate seckey.bin

• To validate the secure key seckey in the secure key repository and display its attributes.

```
zkey validate --name seckey
```

Re-encipher secure keys



where

I

re or reencipher

re-enciphers an existing secure key with a new master key. A secure key must be re-enciphered when the master key of the CCA or EP11 coprocessor changes.

<secure_key_file>

specifies the name of the file that holds the secure key when you are not using the secure key repository.

-N <name> or --name <name>

specifies the key name. You can re-encipher a specific key by specifying its name, or you can use wildcards to match keys to be re-enciphered.

-a <AP queues> or --apqns <AP queues>

Optional. Specifies one or more AP queues for keys in the secure key repository. All keys that are associated with the specified AP queues are re-enciphered. You can use wildcards. Separate multiple AP queues with a comma. If neither key name nor AP queue is specified, then all keys are re-enciphered. If both key name and AP queues are specified then only those keys that match both specifications are re-enciphered.

-i or --in-place

forces an in-place re-enciphering. This is the default for OLD to CURRENT.

-s or --staged

stores the key in a third file (*<key-name>*.renc) in the secure key repository. The key in *<key-name>*.skey is still valid. Once a new master key has been set, you must rerun the **reencipher** command with option --complete. This copies the file *<key-name>*.renc to *<key-name>*.skey and thus completes the staged re-enciphering. Re-enciphering from CURRENT to NEW is by default done in staged mode.

-p or --complete

completes a staged re-enciphering.

-n or --to-new

re-enciphers a secure key that is currently enciphered with the master key in the CURRENT register with the master key in the NEW register.

-o or --from-old

re-enciphers a secure key that is currently enciphered with the master key in the OLD register with the master key in the CURRENT register.

If both options are specified, a secure key that is currently enciphered with the master key in the OLD register is re-enciphered with the master key in the NEW register.

If both options are omitted, **zkey** automatically detects whether the secure key is currently enciphered with the master key in the OLD register or with the master key in the CURRENT register.

If currently enciphered with the master key in the OLD register, it is re-enciphered with the master key in the CURRENT register. If it is currently enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the NEW register.

Note: This option is not available for secure keys of type EP11-AES, because an EP11 cryptographic coprocessor has only two registers to store master keys, CURRENT and NEW.

-f or --output <output_file>

specifies the name of the output file to which the re-enciphered secure key is written if you are not using the secure key repository. If this option is omitted, the re-enciphered secure key is replaced in the file that currently contains the secure key.

Examples

• To re-encipher the secure key in seckey.bin, which is currently enciphered with the master key in the OLD register with the master key in the CURRENT register, and replace the secure key in seckey.bin with the re-enciphered key:

zkey reencipher seckey.bin --from-old

• To re-encipher the secure key in seckey.bin, which is currently enciphered with the master key in the CURRENT register, with the master key in the NEW register, and save the re-enciphered secure key to seckey2.bin:

zkey reencipher seckey.bin --to-new --output seckey2.bin

• To re-encipher the secure key seckey in the the secure key repository.

zkey reencipher --name seckey

 Re-enciphers all secure keys contained in the secure key repository that are associated with AP queue 03.004c.

zkey reencipher --apqns 03.004c

Import a secure key into the the secure key repository



where:

im or import

imports a secure key into the secure key repository.

<secure_key_file>

specifies the name of the file that holds the secure key that you want to import.

-N <key_name> or --name <key_name> specifies the name of the secure key.

-t <volume_type> or --volume-type <volume_type>

Optional. Specifies the volume type of the associated volumes used with dm-crypt. Possible values are PLAIN orLUKS2 (not case-sensitive). If omitted, LUKS2 is used. This option is only available if **zkey** has been compiled with LUKS2 support enabled. If LUKS2 support is not enabled, the default volume type is plain.

-d <description> or --description <description>

Optional. Describes the secure key.

-l <volumes> or --volumes <volumes>

Optional. You can associate volumes with a key. Each volume association specifies the name of the block device (for example /dev/mapper/disk1) and the device mapper name separated by a colon. Separate multiple volume associations with a comma; for example:

-1 /dev/mapper/disk1:enc-disk1,/dev/mapper/disk2:enc-disk2

-a <AP queues> or --apqns <AP queues>

Optional. You can associate AP queue numbers with a key in the . Each AP queue association specifies an device number and domain separated by a period. Separate multiple AP queue associations with a comma; for example:

-a 00.0005,00.001f,00.004d

All specified APQNs must be online, unless option --no-apqn-check is specified.

--no-apqn-check

L

Optional. Do not check whether the specified APQNs are available. Use this option to associate APQNs with a secure AES key that are currently not available.

-S <bytes> or --sector-size <bytes>

Optional. Specifies the sector size in bytes used with dm-crypt. The value must be a power of two in the range 512 - 4096 bytes. The default sector size is 512 bytes.

Note: The **zkey import** command can also import CCA AES CIPHER secure keys and EP11 AES secure keys. When importing an AES CIPHER key, additional checks are performed on this key, such as a check of the history section of the secure key. If a potentially insecure setting is detected, you are prompted to confirm the import. Imported AES CIPHER keys are restricted for export in any way, regardless of the export setting of the imported keys. The only export that is kept allowed is the export to CPACF protected keys, so that these keys can be used with the PAES cipher.

Export a secure key from the secure key repository

zkey syntax - export a secure key from the repository

► zkey — ex — <secure_key_file> — -N <name> →

where

ex or export

exports a key specified by name.

<secure_key_file>

specifies the name of the file that holds the secure key after you have exported it.

-N <name> or --name <name>

specifies the name of the secure key that you want to export.



where

li or list

lists a key specified by name. You can use wildcards to match keys to be listed. If no name, volume, or AP queue is specified, all keys are listed.

-N <key_name> or --name <key_name>

Optional. Specifies the name of the secure key.

-t <volume_type> or --volume-type <volume_type>

Optional. Specifies the volume type of the associated volumes used with dm-crypt. Possible values are PLAIN orLUKS2 (not case-sensitive). Only keys with the specified volume type are listed. This option is only available if **zkey** has been compiled with LUKS2 support enabled.

-l <volumes> or --volumes <volumes>

Optional. Specifies one or more volumes and device mapper names associated with a key. Separate multiple volumes with a comma. All keys that are associated with the specified volumes are listed.

You can use wildcards for the volumes specification. If a device mapper name is specified as part of the volume association, then it is used as part of the filter. If no device mapper name is specified as part of a volume association, only the volume itself is used as filter.

-a <AP queues> or --apqns <AP queues>

Optional. Specifies one or more AP queues of crypto adapters. Separate multiple AP queues with a comma. All keys that are associated with the specified AP queues are listed. You can use wildcards for the AP queue specification.

-K <key type> or --key-type <key type>

Optional. Specifies the type of the secure keys to be listed. Valid values are CCA-AESDATA, CCA-AESCIPHER, or EP11-AES.

Examples

• To list all secure keys in the secure key repository and display their attributes:

zkey list

• To list all secure keys in the secure key repository with names ending with "key", and of type AES CIPHER keys:

zkey list --name "*key" -K CCA-AESCIPHER

• To filter your secure keys in the secure key repository for EP11 AES secure keys:

zkey list -K EP11-AES



where:

rem or remove

deletes a key specified by name.

-N <key_name> or --name <key_name> Specifies the name of the secure key.

-F or --force

Optional. Does not prompt for confirmation of key removal.

Change the properties of secure keys in the secure key repository



where

ch or change

changes the description, volume association or AP queue association of a key specified by name.

-N <key_name> or --name <key_name>

Specifies the name of the secure key.

-t <volume_type> or --volume-type <volume_type>

Optional. Specifies the volume type of the associated volumes used with dm-crypt. Possible values are PLAIN orLUKS2 (not case-sensitive). If omitted, LUKS2 is used. This option is only available if **zkey** has been compiled with LUKS2 support enabled. If LUKS2 support is not enabled, the default volume type is plain.

-d <description> or --description <description>

Optional. Specifies a description of the secure key. If a description exists, it is overwritten.

-l <volumes> or --volumes <volumes>

Optional. Specifies one or more volumes and device mapper names associated with a key. Separate multiple volumes with a comma. All keys that are associated with the specified volumes are listed.

You can use add, delete, or overwrite volume specifications:

- To add volumes to the current associations, prefix the list of volumes that are to be added with a plus sign (+).
- To remove volumes from the current associations, prefix the list of volumes that are to be removed with a minus sign (-).
- If neither minus nor plus is used, then the volume associations are set, overwriting any existing associations.

In one command, you can either add, delete, or set volume associations.

-a <AP queues> or --apqns <AP queues>

Optional. Specifies one or more AP queues of crypto adapters. Separate multiple AP queues with a comma.

You can use add, delete, or overwrite AP queue specifications:

- To add AP queues to the current associations, prefix the list of volumes that are to be added with a plus sign (+).
- To remove AP queues from the current associations, prefix the list of volumes that are to be removed with a minus sign (-).
- To set AP queues, do not use neither minus nor plus sign. Then the AP queues associations are set, overwriting any existing associations.

In one command, you can either add, delete, or set AP queue associations.

All APQNs being added or set (replaced) must be online, unless option --no-apqn-check is specified.

--no-apqn-check

Optional. Do not check whether the specified APQNs are available. Use this option to associate APQNs with a secure AES key that are currently not available.

-S <bytes> or --sector-size <bytes>

Optional. Specifies the sector size in bytes used with **dm-crypt**. The value must be a power of two in the range 512 - 4096 bytes. The default sector size is 512 bytes. If you specify 0, the default is used.

Examples

• To change the secure key seckey in the secure key repository and add volume /dev/mapper/disk2 with device-mapper name enc-disk2 to the list of associated volumes of this secure key:

```
zkey change --name seckey --volumes +/dev/mapper/disk2:enc-disk2
```

• To change the secure key seckey in the secure key repository and remove AP queue 03.004c from the list of associated AP queues:

zkey change --name seckey --apqns -03.004c

Rename a secure key in the secure key repository

zkey syntax - rename a secure key in a key repository

► zkey — ren -N <old_key_name> -w <new_key_name> -⊷

where

ren or rename

renames a key specified by name.

- -N <old_key_name> or --name <old_key_name> specifies the name of the secure key.
- -w <new_key_name> or --newname <new_key_name>
 specifies the name to which the secure key is renamed.

zkey syntax - copy a secure key in the key repo	ository
➡ zkey — co -N < <i>key_name></i> -w < <i>new_key_name></i>	-l <volumes></volumes>

where:

co or copy

copies a key specified by name. Volume associations are not copied.

-N <key_name> or --name <key_name> specifies the name of the secure key.

-w <new_key_name> or --newname <new_key_name>
 specifies the name to which the secure key is copied.

-l <volumes> or --volumes <volumes>

Optional. You can associate volumes with the copied key. Each volume association specifies the name of the block device (for example /dev/mapper/disk1) and the device mapper name separated by a colon. Separate multiple volume associations with a comma; for example:

-1 /dev/mapper/disk1:enc-disk1,/dev/mapper/disk2:enc-disk2

Generate a crypttab entry for a volume



where

cryptt or crypttab

generates crypttab entries for one or more volumes and any device mapper names. crypttab auto-mounts encrypted volumes during system start-up. If no volumes are specified, crypttab entries are generated for all volumes associated with a key.

-t <volume_type> or --volume-type <volume_type>

Optional. Specifies the volume type of the associated volumes used with dm-crypt. Possible values are PLAIN orLUKS2 (not case-sensitive). Only keys with the specified volume type are selected to generate crypttab entries for. This option is only available if **zkey** has been compiled with LUKS2 support enabled.

-l <volumes> or --volumes <volumes>

Optional. Specifies one or more volumes and device mapper names associated with a key. Separate multiple volumes with a comma. All keys that are associated with the specified volumes are listed.

You can use wildcards for the volumes association. If a device mapper name is specified as part of the volume association, then it is used as part of the filter. If no device mapper name is specified as part of a volume association, then only the volume itself is used as filter.

--key-file <*file_name>*

Reads the passphrase from the specified file. If this option is omitted, then you are prompted to enter the passphrase interactively during system startup. The crypttab entries are generated for LUKS2 volumes based on this option, which is only available if **zkey** has been compiled with LUKS2 support enabled.

--keyfile-offset <bytes>

Specifies the number of bytes to skip before starting to read in the file specified with option --key-file. If omitted, the file is read from the beginning. When option --key-file is not specified, this option is ignored. The crypttab entries are generated for LUKS2 volumes based on this option, which is only available if **zkey** has been compiled with LUKS2 support enabled. Not all distributions support the keyfile-offset option in crypttab entries.

--keyfile-size <bytes>

Specifies the number of bytes to be read from the beginning of the file specified with option --key-file. If omitted, the file is read until the end. When --keyfile-offset is also specified, reading starts at the offset. When option --key-file is not specified, this option is ignored. The crypttab entries are generated for LUKS2 volumes based on this option, which is only available if **zkey** has been compiled with LUKS2 support enabled. Not all distributions support the keyfile-size option in crypttab entries.

--tries <number>

Specifies how often the interactive input of the passphrase can be re-entered during system startup. The default is 3 times. When option --key-file is specified, this option is ignored, and the passphrase is read only once from the file. This option is passed to the generated crypttab entries for LUKS2 volumes, and is only available if **zkey** has been compiled with LUKS2 support enabled.

Example

To generate crypttab entries for all volumes that match the pattern /dev/mapper/disk*:

zkey crypttab --volumes "/dev/mapper/disk*"

Generate a cryptsetup command for a volume



where

crypts or cryptsetup

generates **cryptsetup** commands for one or more volumes and device mapper names. **cryptsetup** commands mount the encrypted volumes. If no volumes are specified, **cryptsetup** commands are generated for all volumes associated with a key in the secure key repository.

-t <volume_type> or --volume-type <volume_type>

Optional. Specifies the volume type of the associated volumes used with dm-crypt. Possible values are PLAIN or LUKS2 (not case-sensitive). Only keys with the specified volume type are selected to

generate **cryptsetup** commands for. This option is only available if **zkey** has been compiled with LUKS2 support enabled.

-l <volumes> or --volumes <volumes>

Optional. Specifies one or more volumes and device mapper names associated with a key. Separate multiple volumes with a comma. All keys that are associated with the specified volumes are listed.

You can use wildcards for the volumes association. If a device mapper name is specified as part of the volume association, it is used as part of the filter. If no device mapper name is specified as part of a volume association, only the volume itself is used as filter.

--open

Optional. Generates **cryptsetup luksOpen** or **cryptsetup plainOpen** commands. For a plain volume type, this is the default. This option can not be specified together with the --format option, and is only available if **zkey** has been compiled with LUKS2 support enabled.

--format

Optional. Limits the generated commands to only generate **cryptsetup luksFormat** commands for LUKS2 volumes (without generating **cryptsetup luksOpen** commands), and skip plain mode volumes. For a LUKS2 volume type, this is the default. If specified for a plain volume type, then no command is generated. This option can not be specified together with the --open option, and is only available if **zkey** has been compiled with LUKS2 support enabled.

--key-file <file_name>

Reads the passphrase from the specified file. If this option is omitted, or if the file-name is - (a dash), then you are prompted to enter the passphrase interactively. This option is passed to the generated command(s) for LUKS2 volumes, and is only available if **zkey** has been compiled with LUKS2 support enabled.

--keyfile-offset <bytes>

Specifies the number of bytes to skip before starting to read in the file specified with option --key-file. If omitted, the file is read from the beginning. When option --key-file is not specified, this option is ignored. This option is passed to the generated command(s) for LUKS2 volumes, and is only available if **zkey** has been compiled with LUKS2 support enabled.

--keyfile-size <bytes>

Specifies the number of bytes to be read from the beginning of the file specified with option --key-file. If omitted, the file is read until the end. When --keyfile-offset is also specified, reading starts at the offset. When option --key-file is not specified, this option is ignored. This option is passed to the generated command(s) for LUKS2 volumes, and is only available if **zkey** has been compiled with LUKS2 support enabled.

--tries <number>

Specifies how often the interactive input of the passphrase can be re-entered. The default is 3 times. When option --key-file is specified, this option is ignored, and the passphrase is read only once from the file. This option is passed to the generated command(s) for LUKS2 volumes, and is only available if **zkey** has been compiled with LUKS2 support enabled.

-q or --batch-mode

Optional. If specified, this option suppresses confirmation questions in the generated **cryptsetup** command(s) by assuming an implicit confirmation.

-r or --run

Optional. Executes the **cryptsetup** commands. If one command fails, execution stops.

Example

To generate **cryptsetup** commands for the volumes that use the device-mapper name enc-disk1:

zkey cryptsetup --volumes "*:enc-disk1"

Note: For LUKS2 volumes, the generated **cryptsetup luksFormat** contains option - -pbkdf pbkdf2 to set PBKDF2 as password based key derivation function. LUKS2 volumes typically default to Argon2i as password based key derivation function, but this may cause out-of-memory errors when multiple encrypted volumes are unlocked automatically at boot through /etc/crypttab. Because PAES uses

secure AES keys as volume keys, the security of the key derivation function used to encrypt the volume key in the LUKS key slots is of less relevance.

Convert an AES DATA key into an AES CIPHER key

zkey convert - convert an AES DATA key into an AES CIPHER key				
► zkey — convert -N <i><key name=""></key></i>	— -К <i><кеу</i> type>	no-apqn-check		
► ►				

where

con or convert

converts an existing secure AES DATA key that is either contained in a secure key file or is stored in the secure key repository into an AES CIPHER key. The reverse direction is not possible.

-N <name> or --name <name>

specifies the name of the secure key in the secure key repository that is to be converted.

-K <key_type> or --key-type <key_type>

specifies the type of the key to which the secure key is to be converted. You can only convert AES DATA keys into AES CIPHER keys. So the only possible value is CCA-AESCIPHER.

--no-apqn-check

Optional. Specifies that no check should be performed whether the associated APQNs are available.

-F or --force

Optional. Specifies to not prompt for a confirmation when converting a key.

Example

L

Convert an AES DATA key to an AES CIPHER key:

zkey convert --name sec_data_key -K CCA-AESCIPHER

Additional examples for zkey generate

• Generate a secure key of length 256 bit from a random clear key and store it in file securekey.bin:

zkey generate securekey.bin --keybits 256

 Generate a secure AES CIPHER key from a binary clear key input file and store it in file securekey.bin:

zkey generate securekey.bin --clearkey clearkey.bin -K CCA-AESCIPHER

If you want to securely erase the clear key, you can issue the command:

shred -u clearkey.bin

• Generate a secure key of length 256 bit for XTS cipher mode and store it in file securextskey.bin

zkey generate securextskey.bin --keybits 256 --xts

Appendix B. zkey-cryptsetup - Managing LUKS2 volume keys

Use the **zkey-cryptsetup** command to validate and re-encipher secure AES keys of volumes encrypted with LUKS2 and the PAES cipher.

These secure AES keys are enciphered with a master key of an IBM cryptographic coprocessor in CCA coprocessor mode or EP11 coprocessor mode.

Prerequisites

• The **zkey-cryptsetup reencipher** command requires the CCA host library (libcsulcca.so) and tools package to be installed for secure volume keys of type CCA-AESDATA or CCA-AESCIPHER. It requires the Linux on Z Enterprise PKCS #11 (EP11) Support Program (EP11 host library) for secure volume keys of type EP11-AES to be installed.

To download the required packages, go to <u>www.ibm.com/security/cryptocards</u> and proceed to the software download page for your IBM cryptographic coprocessor version.

- The **zkey-cryptsetup** command requires the libcryptsetup library that comes as part of the cryptsetup package. You require cryptsetup version 2.0.3 or newer available from https://gitlab.com/cryptsetup/cryptsetup/.
- The **zkey-cryptsetup** command also requires the pkey kernel module. For more information, see chapter *Protected key device driver* in *Device Drivers, Features, and Commands*, SC33-8411 available on the IBM Knowledge Center at

www.ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_lib.html.

• You must have at least one IBM Crypto Express adapter configured as a CCA coprocessor or configured as an EP11 coprocessor with an AES master key properly set up.

When you open a key slot contained in the LUKS2 header of the volume using **zkey-cryptsetup**, a passphrase is required. You are prompted for the passphrase, unless option --key-file is specified. Option --tries specifies how often a passphrase can be re-entered. When option --key-file is specified, the passphrase is read from the specified file. You can specify options --keyfile-offset and --keyfile-size to control which part of the key file is used as passphrase. These options behave in the same way as with **cryptsetup**.

zkey-cryptsetup syntax



where:

val or validate

validates a secure AES key of a volume encrypted with LUKS2 and the PAES cipher. It checks if the LUKS2 header of the volume contains a valid secure key. It also displays the attributes of the secure key, such as key sizes, whether it is a secure key that can be used for the XTS cipher mode, and the master key register (CURRENT or OLD) with which the secure key is enciphered.

re or reencipher

re-enciphers an existing secure key with a new master key. A secure key must be re-enciphered when the master key of the CCA CCA coprocessor changes.

setv or setvp

sets a verification pattern of the secure AES key of a volume encrypted with LUKS2 and the PAES cipher. The verification pattern identifies the effective key used to encrypt the data on the volume. The verification pattern is stored in a token in the LUKS2 header.

setk or setkey

sets a new secure AES key for a volume encrypted with LUKS2 and the PAES cipher. Use this command to recover from an invalid secure AES key contained in the LUKS2 header. A secure AES key contained in the LUKS2 header can become invalid when the CCA master key is changed without reenciphering the secure volume key.

<volume>

specifies the name of the volume that you want to work with.

-N or --to-new

The NEW register of a cryptographic coprocessor contains a new master key to be set. The master key in the NEW register cannot be used until it is made the current master key. Use the --to-new option to pro-actively re-encipher a secure key with the NEW master key before this key is made the CURRENT key.

-O or --from-old

Use the --from-old option to re-encipher a secure volume key that is currently enciphered with the master key in the OLD register with the master key in the CURRENT register. This option is only available for secure keys of type CCA-AESDATA or CCA-AESCIPHER.

If both options --from-old and --to-new are specified, a secure volume key that is currently enciphered with the master key in the OLD register is re-enciphered with the master key in the NEW register.

If both options are omitted, **zkey-cryptsetup** automatically detects whether the secure volume key is currently enciphered with the master key in the OLD register or with the master key in the CURRENT register. If currently enciphered with the master key in the OLD register, it is re-enciphered with the master key in the CURRENT register. If it is currently enciphered with the master key in the CURRENT register, it is re-enciphered with the master key in the NEW register. If for this case the NEW register does not contain a valid master key, then the re-encipher operation fails.

-i or --in-place

forces an in-place re-enciphering. This is the default for OLD to CURRENT.

-s or --staged

stores the key in a file <*key-name*>.renc in the repository. The key in <*key-name*>.skey is still valid. Once a new CCA master key has been set, you need to rerun the **reencipher** command with option --complete. This copies the file <*key-name*>.renc to <*key-name*>.skey and thus completes the staged re-enciphering. Re-enciphering from CURRENT to NEW is by default done in staged mode.

-c or --complete

completes a staged re-enciphering.

-q or --batch-mode

Optional. This option is passed to the generated commands. It suppresses all confirmation questions. These are assumed to be answered with YES. This enables a better automation of **zkey-cryptsetup** commands.

-m <secure_key_file> or --master-key-file <secure_key_file>

Specifies the name of a file containing the secure AES key that is set as the new volume key.

-d <key_ file_name> or --key-file <key_ file_name>

Optional. Reads the passphrase from the specified file. If this option is not specified, or if the filename is "-" you are prompted for the passphrase.

-o <bytes> or --keyfile-offset <bytes>

Optional. Specifies the number of bytes to skip in the file specified with the --key-file option. When not specified, the file is read from the beginning. If the --key-file option is not specified, this option is ignored.

-l <bytes> or --keyfile-size <bytes>

Optional. Specifies the number of bytes to read from the file specified with option --key-file. When not specified, the file is read until the end. When option --key-file is not specified, this option is ignored.

-T <number> or --tries <number>

Optional. Specifies how often the interactive input of the passphrase can be retried. The default is 3 times. When option --key-file is specified, this option is ignored, and the passphrase is read only once from the file.

-D or --debug

Displays additional debugging messages during processing. This option implies --verbose.

-V or --verbose

Displays additional information during processing.

Examples

• To re-encipher the secure key of the encrypted volume /dev/mapper/disk1:

zkey-cryptsetup reencipher /dev/mapper/disk1

• To re-encipher the secure key of the encrypted volume /dev/mapper/disk1 in staged mode:

zkey-cryptsetup reencipher /dev/mapper/disk1 --staged

• To complete re-enciphering the secure key of the encrypted volume /dev/mapper/disk1:

zkey-cryptsetup reencipher /dev/mapper/disk1 --complete

• To re-encipher the secure key of the encrypted volume /dev/mapper/disk1 in in-place mode:

zkey-cryptsetup reencipher /dev/mapper/disk1 --in-place

• To re-encipher the secure key of the encrypted volume /dev/mapper/disk1 that is currently enciphered with the master key in the OLD register with the master key in the NEW register:

zkey-cryptsetup reencipher /dev/mapper/disk1 --from-old --to-new

• To validate the secure key of the encrypted volume /dev/mapper/disk1 and display its attributes:

zkey-cryptsetup validate /dev/mapper/disk1

• To set the verification pattern of the secure key of the encrypted volume /dev/mapper/disk1:

zkey-cryptsetup setvp /dev/mapper/disk1

• To set the secure key contained in file seckey.key as the new key for the encrypted volume /dev/ mapper/disk1:

zkey-cryptsetup setkey /dev/mapper/disk1 --master-key-file seckey.key

88 Linux on Z and LinuxONE: Pervasive Encryption for Data VolumesSeptember 2020

Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Documentation accessibility

The Linux on Z and LinuxONE publications are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF file and want to request a Web-based format for this publication send an email to eservdoc@de.ibm.com or write to:

IBM Deutschland Research & Development GmbH Information Development Department 3282 Schoenaicher Strasse 220 71032 Boeblingen Germany

In the request, be sure to include the publication number and title.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility at

www.ibm.com/able

90 Linux on Z and LinuxONE: Pervasive Encryption for Data VolumesSeptember 2020

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Index

A

accessibility 89 adjunct processor queue number APQN 6 **AES CIPHER 12** AES DATA 12 AES keys managing with zkey 71 AES secure key change in the secure key repository 40 copy/duplicate in the secure key repository 41 import into secure key repository 38 list from secure key repository 39 re-encipher 37 rename from secure key repository 40 validate 36, 46 AP queue 6 APQN adjunct processor queue number 6 assumptions 9 auto-detection function zkey 48, 51 automatically opening volumes 22 automatically unlocking partition at Linux startup 66 in plain mode 66 automatically unlocking volumes 22

В

block cipher modes, See cipher modes block devices in Linux on Z $\underline{3}$ block size combinations $\underline{56}$

С

CBC 13 CCA AES CIPHER 12 CCA AES DATA 12 CCA coprocessor 3, 11 CCA coprocessor mode 11 CCA cryptographic coprocessor 11 CCA master key changing 48 re-enciphering secure keys 48 CCAcoprocessor mode 11 change zkey command 40 change an AES secure key in the secure key repository 40 changing CCA master key 48 changing master key in plain mode 65 checking cryptographic coprocessors 55 kernel modules 55

cipher modes CBC 13 CTR 13 ECB 13 XTS 13 clean room environment 59 clear key available 59 clear key encrypted volume re-encrypting onto new secure key encrypted volume 32 transform into secure key encrypted volume 32 clear key to secure key on new volume 32 clear key to secure key on same disk 32 clear key, definition 6 configuration of the infrastructure for protected volume encryption 9 configuration verification 55 considerations cipher mode 13 cryptographic coprocessor 11 device 11 secure key 12 convert clear key to secure key encrypted 32 LUKS1 to LUKS2 32 coprocessor mode 11 copy zkey command 41 copy an AES secure key in the secure key repository 41 copying data to new encrypted volume 31 create a file system 66 creating encrypted volumes 17 Crypto Express cryptographic coprocessor 3 cryptographic adapter 3 cryptographic algorithm paes 13 selecting 13 cryptographic card 3 cryptographic coprocessor considerations 11 definition 6 replacing 52 replacing with different master key 53 replacing with same master key 52 cryptographic coprocessors sharing master keys 52 cryptsetup debugging 58 encryption in plain mode 63 zkey command 42 cryptsetup commands generate 42 cryptsetup luksFormat 17 cryptsetup plain mode 63 cryptsetup plainOpen 63 crypttab zkey command 41

crypttab entry generate from secure key repository <u>41</u> CTR 13

D

data copying to new encrypted volume <u>31</u> migrating to new encrypted volume <u>31</u> debugging cryptsetup <u>57</u> pkey <u>57</u> zkey-cryptsetup <u>57</u> debugging pkey <u>58</u> device considerations <u>11</u> disk password <u>66</u> distribution information v dm-crypt volume <u>3</u> duplicate an AES secure key in the secure key repository <u>41</u>

E

ECB 13 effective key, definition 6 encrypted volume creating accessing 17 mounting at user login 23 opening at user login 23 transform into secure key encryption 31 encrypted volumes managing 17 encrypting an unencrypted volume in plain mode 65 environment variable **ZKEY REPOSITORY 71** environment variable ZKEY_REPOSITORY 35 EP11 AES secure key 12 error handling out-of-memory 58 etc/crypttab editing 25 etc/fstab 17 examples zkey 71 zkey-cryptsetup 85 export zkey command 39 export an AES secure key from secure key repository 39

F

file system create <u>66</u> firmware master key, definition <u>6</u>

G

generate cryptsetup commands <u>42</u> zkey command 36 generate a secure key in the secure key repository <u>36</u> generate secure key in a file <u>63</u> in plain mode <u>63</u> generating a secure key <u>12</u>

Н

hardware prerequisites <u>9</u> hardware security module HSM <u>6</u> HSM hardware security module <u>6</u> HSM master key, definition <u>6</u>

I

IBM Z cryptographic hardware 3 import zkey command 38 import a secure key into secure key repository 38 In-place option 44 infrastructure for protected volume encryption components 3 concepts 1, 3 configuration 9 planning 11 prerequisites 9 required modules and components 13 sample system 15 secure key considerations 12 setting up 9 setup 3

Κ

key management 35

L

libcryptsetup 85 Linux distribution v Linux startup automatically opening encrypted volumes 22 automatically unlocking encrypted volumes 22 unlocking partition automatically 66 list zkey command 39 list an AES secure key from secure key repository 39 loading modules 13 logical volume group 15 loss of master key 59 loss of secure key recovering encrypted volumes with key from secure key repository 60 LUKS LUKS1 and LUKS2 3 LUKS volume encryption key LVEK 3

LUKS1 3.5 LUKS1 clear key convert to LUKS2 secure key 32 LUKS2 5 LUKS2 header paes-verification-pattern 45 LUKS2 secure key convert from LUKS1 clear key 32 validate 47 LUKS2 volume key re-encipher 44, 49 set verification pattern 45 setting a new 59 validate 44 LUKS2 volume key, definition 6 LUKS2 volume keys managing with zkey-cryptsetup 43,85 luksFormat 17 luksFormat command generate 42 lvcreate 17 LVEK LUKS volume encryption key 3 LVM physical volume transforming unencrypted to encrypted 25 unencrypted to encrypted 25 LVM volume group 25

Μ

managing encrypted disks or partitions 17 encrypted volumes 17 LUKS2 volume keys 43 secure key repository 35 managing keys 35 master key changing 48 re-encipher LUKS2 volume key with zkey-cryptsetup 49 re-encipher secure key with zkey 48, 51 re-enciphering secure keys 48 master key change in plain mode 65 master key loss 59 master key saving 59 master key, definition 6 master keys sharing 52 MIGR VOLGROUP 25 migrate data from unencrypted volume into encrypted 65 in plain mode 65 migrating data to new encrypted volume 31 migrating unencrypted to encrypted LVM physical volume 25 modes of operation, See cipher modes modprobe 13 mounting at user login encrypted volume 23

Ν

new encrypted volume migrating data <u>31</u> new secure LUKS2 volume key setting $\underline{45}$

0

opening at user login encrypted volume 23 opening encrypted volume 22 opening encrypted volumes at Linux startup 22 out-of-memory error handling 58

Ρ

PAES 13 PAES cipher 3 paes-verification-pattern 45 PAM module 66 pam mount 66 pam_mount configuration 23 pam_mountconfiguration file 66 persistent unlocking during system startup 63 pervasive encryption 17 physical block size combinations 56 pkey debugging 58 protected AES key 69 plain mode automatically unlocking partition 66 changing master key 65 cryptsetup 63 encrypting an unencrypted volume 65 encryption with cryptsetup 63 generate secure key in a file 63 transform unencrypted partition into encrypted 65 unlocking encrypted volume 66 unlocking partition at user login 66 plainOpen cryptsetup 63 plainOpen command generate 42 planning 11 prerequisites assumptions 9 hardware prerequisites 9 software prerequisites 9 problem resolution 55 protected key swap disk 69 protected key, definition 6 protected keys encrypting swap disks 69 pvcreate 17 pvextend 25 pvmove 25

R

pvremove 25

re-encipher secure LUKS2 volume key 44 re-encipher an AES secure key 37 re-encipher LUKS1 clear key to LUKS2 secure key 32 re-encipher secure key from a file 51 re-encipher secure key from the secure key repository 48 re-enciphering secure keys 48 re-encryption from clear key to secure key 31 recovering from invalid secure key 59 recovering encrypted volumes with a secure key from the secure key repository 60 recovery 55 reencipher zkey command 37 reencipher command (zkey-cryptsetup) 44 remove zkey command 39 remove an AES secure key from secure key repository 39 rename zkey command 40 rename an AES secure key from secure key repository 40 replacing a cryptographic coprocessor 52 replacing cryptographic coprocessor with different master key 53 replacing with same master keycryptographic coprocessor 52 required modules 13 revision history v

S

same volume convert LUKS1clear key to LUKS2 secure key 32 sample system configuration 15 hardware components 15 software components 15 saving the master key 59 sector size combinations 56 secure AES key export from secure key repository 39 remove from secure key repository 39 secure AES keys managing with zkey 71 secure key generate 36 generate in a file 63 import into secure key repository 38 in plain mode 63 re-encipher 37 re-encipher from a file 51 re-encipher from the secure key repository 48 re-encipher LUKS2 volume key 49 recovering from invalid 59 validate 36 validate from file 47 secure key (SK) 12 secure key considerations 12 secure key file 12 secure key from the repository recovering encrypted volumes 60 secure key generation 12

secure key loss recovering encrypted volumes with key from secure key repository 60 secure key repository change a secure key 40 copy a secure key 41 default location 35 export a secure key 39 generate a secure key 36 generate cryptsetup commands for volumes 42 generate crypttab entries for volumes 41 generate secure key 17 import a secure key 38 list secure keys 39 re-encipher a secure key 37 re-encipher secure key 48 recovering encrypted volumes 60 remove a secure key 39 rename a secure key 40 setting location 35 validate a secure key 36, 46 secure key, definition 6 secure keys managing 71 validating 46 secure LUKS2 volume key re-encipher 44 set verification pattern 45 validate 44 selecting a cryptographic algorithm 13 service information 57 set verification pattern of secure LUKS2 volume key 45 setting new secure LUKS2 volume key 45 setup of sample system 15 setup of the infrastructure for protected volume encryption 9 setvp set the verification pattern 25 setvp command (zkey-cryptsetup) 45 sharing master keys on different cryptographic coprocessors 52 shred 12 shred command 31, 32 SK secure key 12 software prerequisites 9 Staged mode option 44 Summary of changes v swap disk pkey-generated protected key 69 swap disks encrypting with protected keys 69 syntax zkey 71 zkey-cryptsetup 85 system startup time persistent unlocking 63

Т

TKE (Trusted Key Entry workstation) <u>5</u> transform clear key encrypted volume into new secure key encrypted volume <u>32</u> transform encrypted volume transform encrypted volume (continued) into secure key encrypted volume 31 transform unencrypted volume into encrypted in plain mode 65 transforming unencrypted into encrypted volume 25 transforming unencrypted to encrypted LVM physical volume 25 troubleshooting 57 Trusted Key Entry workstation (TKE) 5

U

unattended reboots <u>66</u> unencrypted into encrypted volume <u>25</u> unlocking encrypted volume in plain mode <u>66</u> unlocking encrypted volumes at Linux startup <u>22</u> unlocking partition at user login <u>66</u> automatically at Linux startup <u>66</u> in plain mode <u>66</u> user login unlocking partition in plain mode <u>66</u> user password <u>66</u> useradd <u>66</u>

V

validate secure LUKS2 volume key 44 zkey command 46, 47 zkey-cryptsetup command 47 validate a LUKS2 secure key 47 validate a secure key from a file 47 validate an AES secure key 36, 46 validate command (zkey-cryptsetup) 44 validating secure keys 46 verification pattern set for secure LUKS2 volume key 45 setting 25 verifying the configuration 55 vgcreate 17 vgextend 25 vgreduce 25 volume opening encrypted 22 transforming unencrypted into encrypted 25 unlocking encrypted 22 volume, definition 6 volumes generate cryptsetup commands luksFormat 42 plainOpen 42

W

working with encrypted volumes <u>17</u> wrapping key, definition 6

Χ

XTS 13

Ζ

zkev auto-detection function 48, 51 changing master key 48, 51 command reference 71 debugging 57 examples 71 managing secure AES keys 71 re-encipher secure key 48 re-encipher secure key from file 51 subcommands 71 syntax 71 zkey commands change 40 copy 41 cryptsetup 42 crypttab 41 export 39 generate secure key 36 import 38 list 39 reencipher 37 remove 39 rename 40 validate 46, 47 zkey utility 12 ZKEY_REPOSITORY 35, 71 zkey-cryptsetup command reference 85 debugging 57 examples 85 In-place 44 managing LUKS2 volume keys 43, 85 re-enciphering LUKS2 volume key 49 reencipher 44 setting new secure LUKS2 volume key 45 setvp 45 Staged mode 44 syntax 85 validate 44 zkey-cryptsetup commands validate 47 zkey-cryptsetup convert 32 zkey-cryptsetup setvp 25 zkey-cryptsetupvalidate 47 zkeychange 40 zkeycopy 41 zkeycryptsetup 42 zkeycrypttab 41 zkeyexport 39 zkeygenerate 36 zkeyimport 38 zkeylist 39 zkeyreencipher 37 zkeyremove 39 zkeyrename 40 zkeyvalidate 46, 47

98 Linux on Z and LinuxONE: Pervasive Encryption for Data VolumesSeptember 2020



SC34-2782-04

