Using Secure Keys for Disk Encryption

Reinhard Buendgen – buendgen@de.ibm.com
Trademarks

The following are trademarks of the International Business Machines Corporation in the United States and/or other countries.

- CICS®
- Cognos®
- DataStage®
- DB2®
- GDPS
- MQ®
- Parallel Sysplex®
- QualityStage
- Rational®
- Maximo®
- SPSS®
- System Storage®
- System x®
- Tivoli®
- WebSphere®
- XIV®
- zEnterprise®
- z/OS®
- z Systems®
- z/VM®

* Registered trademarks of IBM Corporation

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries. IT Infrastructure Library is a Registered Trade Mark of AXELOS Limited. ITIL is a Registered Trade Mark of AXELOS Limited.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

UNIX is a registered trademark of The Open Group in the United States and other countries.

VMware, the VMware logo, VMware Cloud Foundation, VMware Cloud Foundation Service, VMware vCenter Server, and VMware vSphere are registered trademarks or trademarks of VMware, Inc. or its subsidiaries in the United States and/or other jurisdictions.

Other product and service names might be trademarks of IBM or other companies.

Notes:

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in your area. Regardless, our warranty terms apply. All statements regarding IBM's future direction and intent are subject to change or withdrawal notice, and represent goals and objectives only.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your area.

This information provides only general descriptions of the types and portions of workloads that are eligible for execution on Specialty Engines (e.g., zIIPs, zAAPs, and IFLs) ("SEs"). IBM authorizes customers to use IBM SE only to execute the processing of Eligible Workloads of specific Programs expressly authorized by IBM as specified in the "Authorized Use Table for IBM Machines" provided at www.ibm.com/systems/support/machine_warranties/machine_code/aut.html ("AUT"). No other workload processing is authorized for execution on an SE. IBM offers SE at a lower price than General Processors/Central Processors because customers are authorized to use SEs only to process certain types and/or amounts of workloads as specified by IBM in the AUT.

© 2017 IBM Corporation
Acknowledgements

Contributors to this project:

- Ingo Franzki  ifranzki@de.ibm.com
- Hendrik Brueckner  brueckner@de.ibm.com
- Harald Freudenberger  freude@de.ibm.com
- Martin Schwiddefsky  mschwide@de.ibm.com
Contents …

- Issues with current end-to-end disk encryption
  - Stolen keys
  - HSMs and secure keys
- Key Tokens
  - How key Tokens differ from "raw keys"
- Ciphers for key tokens
  - The PAES cipher
  - dm-crypt with PAES in plain format
- Changes suggested to cryptsetup
  - a plug-in concept for key token support
  - dm-crypt with PAES in LUKS(2) format
- Changing HSM master keys
- Two factor authentication
End-to-End Data at-rest Encryption

In Linux most prominent Example of E2E data at-rest encryption: dm-crypt/LUKS
Linux File System Stack with dm-crypt

- Application
- Virtual file system
  - File system (e.g. ext4)
    - Page cache
      - Logical block DD
        - Physical block DD
          - Disk

I/O system call (open, read, write)
Standard I/O (through page cache)
Layers of logical device drivers: logical volumes, RAID, multipath + dm-crypt

- Direct I/O (bypassing page cache)
- Direct I/O to device (e.g. swap)

encrypted
plain text
Issues with Current End-to-end Disk Encryption

- keys used to encrypt data on disks can be stolen
  - they reside in main memory in plain text format (e.g. in cryptsetup process or kernel)
  - may be inside dumps

- encrypted disks cannot be automatically unlocked
  - w/o sacrificing security
  - in particular systems using an encrypted root disk cannot be automatically booted
    - key or passphrase to protect key must be provided securely
  - not so bad for PCs but a real problem for servers
Stealing Disk Encryption Keys

Stolen keys
- allow for off-line attacks
  - to stolen disks
  - from remote servers
- little risk of being caught during data transfer
Clear Key vs. Secure Key Cryptography

Clear key crypto = plain text key

Secure key: an effective key wrapped by an HSM master key

Do not confuse secure and secret keys

Hardware security module (HSM) -- tamper proof
Key Tokens

Terminology

- **Effective key**
  - An array of bits to be used as key in the mathematical description of a cipher

- **Wrapped key**
  - A key encrypted by another key

- **Wrapping key (aka key encrypting key, KEK)**
  - A key used to encrypt another key

- **HSM master key**
  - A KEK used by an HSM when creating secure keys
  - HSM master keys cannot be extracted from the HSM

- **Key token**
  - A representation of an effective key

Examples of key tokens

- **Effective keys**
- **Wrapped keys**
- **Secure keys**
  - A key token that can be used for cryptographic operations only within an associated HSM
  - Typically contains an effective key wrapped by a HSM master key

- **Key label**
  - Identifying a key in a key ring / key repository

- **Key reference**
  - Pointer to a key in a key ring / key repository
Differences Between Key Tokens and Effective Keys

- **Size:**
  - key tokens may have a different size than the effective keys they describe

- **Generation:**
  - not every byte array of a specific size is a valid key token

- **Identification:**
  - a key token is not a unique representation of an effective key
Differences Between Key Tokens and Effective Keys: Size

Key tokens may have a size different from the effective keys they describe. E.g.

- An effective AES-192 key (size: 192 bits) wrapped by an AES-256 key has a size of 256 bits.
- An effective AES-256 key (size 256 bits) wrapped by a 4K RSA key has a size of 4096 bits.
- Secure keys typically contain both the (wrapped) effective key and key attributes, and are therefore larger than the effective keys.
- A key label may have an arbitrary size.
- Key references have sizes independent of the effective key they refer to.

Key token size

- Required to specify size of buffers to store the key token.

Effective key size

- Sometimes required to specify a cipher attribute (e.g. for AES, or XTS vs CBC).
Differences Between Key Tokens and Effective Keys: Generation

(Random) effective symmetric keys are often generated as (random) byte arrays of a specific size,

But not every byte array of a specific size is a valid key token

- This is not even true for all effective (symmetric) keys
  - Parity bits included: e.g. DES, 3DES
  - Weak keys should be avoided: e.g. DES, IDEA, RC4, blowfish
  - (asymmetric) keys often have structure and constraints on component values (e.g. primality)

- Wrapped keys
  - AES-192 key wrapped with AES-256 key: result of encrypting effective key + parity bits
  - RSA-PKCS #1 v1.5 / OEAP key wrapping: result of encrypting effective key + random bits + marker bit patterns

- Secure keys, PKCS #11 key objects
  - Contain key attributes (e.g. sensitivity, extractability, key usage)
  - May contain authentication tags (to ensure integrity of key object)

- Key labels, key references
  - Generating key labels or key references does not generate effective keys
  - For labels not all names may be allowed (e.g. due to a restriction of the character set allowed)
Identification: a key token is not a unique representation of an effective key. E.g.

- **Wrapped keys:**
  - An effective key may be wrapped by different KEKs.
  - Even with same RSA wrapping key, according to PKCS #1 v1.5 / OEAP:
    - Wrapped keys result from encrypting random bits + effective key.

- **Secure keys**
  - An effective key may be wrapped by different HSM master keys.
  - With secure keys the identification must not reveal the associated effective key.

- **Key labels**
  - Different key labels may denote the same effective key.
A cipher-implementation that operates on a key token must be aware of the kind of key token it supports.
- must be accessible through Crypto-API (AF_ALG)

Therefore give it a name that identifies both
- the cipher and
- the type of token

E.g.
- “tpm_aes”
  - for a hypothetical cipher implementation that operates on AES keys wrapped by a KEK stored in the local TPM
- “paes”
  - for protected key AES – see later in the presentation
The PAES Cipher

• PAES = protected key AES
  – IBM Z CPU support for symmetric ciphers to compute with secure key tokens avoiding the overhead of communicating with an HSM
    • see later in this presentation

• The PAES cipher
  – Implements AES algorithms: ecb(paes), cbc(paes), ctr(aes), xts(aes)
  – Operating on secure key tokens (size: 64 bytes) for the IBM CryptoExpress CCA-coprocessor (=HSM)

• Implemented by IBM Z specific kernel module paes_s390
  – Available since kernel 4.11
  – APIs are available
    • Inside the kernel
    • via crypto API (AF_ALG)
Changes in dm-crypt to Support Ciphers for Key Tokens

• Given a cipher for key tokens in the kernel crypto framework
  - Required changes in dm-crypt: None
Using the PAES cipher with dm-crypt – Plain Format

- **generate a file with a secure key**
  
  ```
  # zkey generate seckey.bin -xts
  - requires access to CryptoExpress adapter
  ```

- **open block device as device mapper volume**
  
  ```
  # cryptsetup open --type plain --key-file seckey.bin \n  --key-size 1024 --cipher paes-xts-plain64 /dev/dasdb1 \n  plain_enc
  
  - new virtual device mapper volume /dev/mapper/plain_enc will be created
  - requires access to CryptoExpress adapter
  ```

- **use new device mapper volume**
  
  ```
  - (only once) create file system:
    
    # mkfs.ext4 /dev/mapper/plain_enc
  
  - mount:
    
    # mount /dev/mapper/plain_enc /mount_point
  
  - access:
    
    # echo "hello world" > /mount_point/myfile
Required changes to LUKS(2) format

• None!
  - just allow for cipher names known to kernel crypto framework that operate on key tokens.
  - key size – is size needed to store key (i.e. key token size)

• Optional
  - interpret key digest as digest of key identification
    • allows to deal with re-enciphering key due to HSM masker key changes
Managing LUKS(2) Volumes for Key Token Based Ciphers

- Management is supported by cryptsetup tool functions, e.g.
  - LuksFormat
  - LuksOpen
- Cryptsetup uses the cipher
  - To encrypt data on volume (passed to dm-crypt)
  - To wrap/unwrap “LUKS master key” in LUKS header
- Cryptsetup needs key lengths
  - Effective key size: to describe what key tokens to generate, input to password based key derivation
  - key token size: to pass key token to dm-crypt, to allocate buffers when generating key tokens
- Cryptsetup generates key
  - to obtain a new random effective “LUKS master key”
  - When deriving a KEK from the passphrase to wrap/unwrap effective “LUKS master key”
- Cryptsetup must identify keys
  - when generating initial key digest
  - when determining the right key slot to use
Methods for Cipher “Objects” Supporting Key Tokens

Cipher handling

- `int is_cipher_and_mode_valid_func(...);`
  - check whether cipher, IV processing and key size are compatible

- `unsigned int get_cipher_blocksize_func(...);`
  - return the block size for a cipher

Key length handling

- `unsigned int get_key_token_size_func(...);`
  - return the size in bytes of a key token

- `unsigned int get_key_token_key_bits_func(...);`
  - return the cryptographic key length (in bits) of the key contained in the key token

- `int get_derived_key_bits_func(...);`
  - derive a key size (in bits) dependent on the mode of operation (e.g. XTS key size vs CBC key size)

Key token generation

- `int generate_key_token_func(...);`
  - generate (valid) random key token

- `int generate_key_token_from_effective_func(...);`
  - generate key token from effective key

Computing unique identifier of key token

- `int generate_key_token_identifier_func(...);`
  - compute unique effective key identifier for the key token.

Optional

- `int key_token_discloses_secret_func(...);`
  - indication whether a key token must be erased after use
Default Methods for Cipher “Objects” supporting Key Tokens

**Cipher handling**

```c
int is_cipher_and_mode_valid(...);
```
- whatever consistency checks are done today

```c
unsigned int get_cipher_blocksize(...);
```
- return the block size for a cipher

**Key length handling**

```c
unsigned int get_key_token_size(...);
```
- return the size of a key (token)

```c
unsigned int get_key_token_key_bits(...);
```
- return the size of the key (token)

```c
int get_derived_key_size(...);
```
- e.g. 128 for AES128-CBC or 256 for AES128-XTS

**Key token generation**

```c
int generate_key_token(...);
```
- generate random byte array

```c
int generate_key_token_from_effective(...);
```
- Return (copy of) input buffer

**Computing unique identifier of key token**

```c
int generate_key_token_identifier(...);
```
- return key (token)

**Optional**

```c
int key_token_discloses_secret(...);
```
- return true
PAES Methods for Cipher “Objects” supporting Key Tokens

Cipher handling

```c
int is_cipher_and_mode_valid(...);
```
- e.g. disallow essiv mode

```c
unsigned int get_cipher_blocksize(...);
```
- Return 128

Key length handling

```c
unsigned int get_key_token_size(...);
```
- return 64 bytes for CBC mode 128 bytes for XTS mode

```c
unsigned int get_key_token_key_bits(...);
```
- 128 or 256 depending on key token

```c
int get_derived_key_size(...);
```
- e.g. 256 for AES-CBC or 512 for AES-XTS

Key token generation

```c
int generate_key_token(...);
```
- generate random secure key

```c
int generate_key_token_from_effective(...);
```
- generate secure key from byte array

Computing unique identifier of key token

```c
int generate_key_token_identifier(...);
```
- return the encryption of an array of zero bytes

Optional

```c
int key_token_discloses_secret(...);
```
- return false
Plug-in for cipher objects

- Library to implement cipher objects functions
- Requires architected interface for such plug-ins in cryptsetup
- Processing ("LUKS2 style plug-in") using wrapper functions
  - if wrapper function is called with "common" cipher name
    - Use default function
  - Else search for cipher in cipher handler array calling into plug-in library loaded at first use
    - After linking cipher plug-in initialize cipher handler with functions implementations for the new cipher
- Options to locate plug-in libraries:
  - as determined by new cryptsetup option
  - *path derived from cipher name*
  - path taken from environment variable
  - path taken from config file
CPACF Protected Keys

- IBM Z function of the CPU (CPACF)
- each virtual server (LPAR or guest) has a *hidden* master key
  - hidden master key is not accessible from operating system in LPAR or guest
- a key wrapped by the hidden master key is called a *protected* key
- protected key tokens can be generated
  - from clear keys (insecure)
  - from secure keys using CEX Adapter (secure)
- IBM Z CPU can compute symmetric encryption and decryption functions for protected keys
  - **pro**
    - no clear keys in operating system memory
    - fast, encryption/decryption at CPU speed, no I/O needed
  - **cons**
    - "hiding" of master key not as good and tamper proof as that of an HSM
    - not certified as HSM
    - the LPAR/guest master key is not persistent
      - protected keys are volatile
      - need to save (secure) key to derive protected key from
- Similar Schemes
  - Intel provides some kind of asymmetric protected keys
  - Could be implemented using technology like Intel SGX enclaves
Managing Protected Keys: the pkey Kernel Module

- **pkey kernel module**
  - since kernel version 4.11
  - implements misc device: /dev/pkey
  - provides functions (IOCTLS) to
    - PKEY_GENSEC: generate a random CCA secure key
    - PKEY_CLR2SEC: generate a CCA secure key from a clear key
    - PKEY_SEC2PROT: generate a protected key from a CCA secure key
    - PKEY_FINDCARD: find an adapter and domain associated with a given secure key
    - PKEY_SECK2PROTK: first PKEY_FINDCARD then PKEY_SEC2PROT
    - PKEY_GENSEC, PKEY_CLR2SEC, PKEY_SEC2PROT have target adapter&domain as well as key type as arguments
  - Secure keys are CCA secure keys of type AESDATA
  - protected keys are of type:
    ```c
    struct pkey_protkey {
        __u32 type;           /* 1: AES 128, 2: AES-192, 3: AES-256 */
        __u32 len;            /* bytes actually stored in protkey[]: 48 – 64 */
        __u8 protkey[MAXPROTKEYSIZE]; /* wrapped key + verification pattern */
    };
    ```
The PAES in-kernel Cipher (cont’d)

- paes ciphers take CCA AES secure keys as key arguments
  - transforms secure key into protected key
    - using pkey module
      - discovers matching HSM
    - caches protected key (into encryption context aka transform)
    - uses protected key for cryptographic operations
  - Each protected key operation may
    - either succeed
    - or return a verification pattern miss-match result
      - Then the secure key must be transformed into a protected key again.
      - E.g., this will happen after a system migration.
Secure Key Handling: the zkey Tool

- part of s390tools
- requires pkey module
- generate, validate, re-encipher secure AES keys to be transformed into protected keys
  * generate
    - generates file with AES secure key (AESDATA)
    - random key or from clear key
    - single key (for CBC) or two keys (for XTS)
    - size of secure keys: 64 bytes (single key), 128 bytes (XTS key) regardless of AES key size
  * validate
    - checks if input file contains valid AES secure key
    - if yes displays key attributes
  * re-encipher
    - support master key change on CryptoExpress adapter
    - transforms a valid secure key wrapped by an current (or old) HSM master key into a secure
      key wrapped by a new (or current) master key
    - requires installation of CCA package from
Using the PAES with dm-crypt – LUKS Format

0: insert new kernel modules during boot
  - requires access to CEX5C or CEX6C adapter

1: format raw volume as dm-crypt volume
  - cryptsetup luksFormat ...
    - cipher, effective key length, hash, password
    - use “paes” instead of “aes”
  - writes dm-crypt header to disk

2: open dm-crypt volume and assign it a virtual volume name
  - cryptsetup luksOpen ...
    - dm-crypt volume, virtual volume, passphrase
      (needs not be protected)
  - creates virtual volume in /dev/mapper

3: use virtual volume
  - mkfs (or mkswap)
  - mount (or swapon)
  - any kind of standard I/O
  - do not use raw volume directly

header contains wrapped secure key

kernel translates I/O to virtual volume into I/O to raw volume and performs decryption/encryption

kernel stores raw and encryption info for virtual volume
LUKS/dm-crypt with Protected Keys - Components

Components

- **pkey** kernel module for protected key management
  - generate secure key
  - transform secure key into protected key

- **paes** kernel module to perform protected key encryption-decryption
  - implements paes cipher

- **dm-crypt** kernel module (unchanged)

- extended dm-crypt management tool **cryptsetup**
  - supports cipher plugins
  - e.g. paes cipher plugin
    - stores secure key into LUKS header

- **zkey** tool to
  - generate and manage secure keys
  - re-encipher secure key
Changing the HSM Master Key

- Simply changing a HSM master key invalidates all existing secure keys.
- A careful procedure is required to transform (aka “re-encipher”) secure keys derived from old MK to secure keys derived from new MK.
- It’s a rare procedure (typically less frequent than once a year)
  - Sometimes enforced by security regulations

**HSM Option 1**

- HSM has a storage for old MKs.
- When replacing the current MK by a new MK.
- The former current MK becomes the old MK.
- It is then possible to re-encipher secure keys from old MK to new MK.
- Possibly the HSM can still use secure keys derived from old MKs to encrypt/decrypt data.

**HSM Option 2**

- HSM has a storage for new MKs.
- After storing a new MK,
  - it is possible to re-encipher secure keys from current MK to new MK.
- Only after committing the new MK the new MK is written to the current MK storage.

Some HSMs combine both options.
Requirements to Support MK Changes with LUKS(2)

- Hash the return value of the identity function of the key token rather than the key (token) to determine which key slot matches a passphrase.

- Add a “force” option for luksAddKey
  - to allow to store secure keys that cannot yet be used into a new key slot.

Note, the same requirement helps to support accessing a shared volume from different systems each equipped with different HSM (with different MKs):

- put key tokens containing same effective key wrapped by different HSM MKs into different key slots.
Two Factor Authentication

The support of secure keys provides 2FA

- Factor 1: knowledge of passphrase
  - to generate KEK to wrap secure key

- Factor 2: possession of HSM with right HSM master key
  - to use secure key

Neglecting Factor 1 allows automatic opening of LUKS devices secured by Factor 2 only
  ➔ Automatic booting of servers.
Summary

- Ciphers to support key tokens
  - allow the usage of multiple key representations
  - including key labels, wrapped keys and HSM secure keys
- Kernel support for ciphers based on key tokens can be used by dm-crypt today
- Key token based ciphers can be specified in existing LUKS or LUKS2 format
- The management of LUKS(2) formatted disks
  - requires some functions to be cipher specific
  - a plug-in scheme is suggested to deal with such cipher specific functions
Questions?
1. format volume
   - write secure key and paes cipher in volume header

2. open volume
   - passphrase not security relevant
   - fetch secure key from volume header
   - pass secure key to dm-crypt
   - let paes transform secure key into protected key

3. read/write data
   - dm-crypt uses paes with protected key to decrypt/encrypt data
LUKS/dm-crypt with Protected Keys - Opening

1. format volume
   - write secure key and paes cipher in volume header

2. open volume
   - passphrase not security relevant
   - fetch secure key from volume header
   - pass secure key to dm-crypt
   - let paes transform secure key into protected key

3. read/write data
   - dm-crypt uses paes with protected key to decrypt/encrypt data

Cryptsetup with plugin API

Paes plugin

Zkey

Dm-crypt

Paes

Pkey

Machine

Crypto

LUKS

"paes"

CEX5C

CPU
LUKS/dm-crypt with Protected Keys - Processing

1. format volume
   - write secure key and paes cipher in volume header

2. open volume
   - passphrase not security relevant
   - fetch secure key from volume header
   - pass secure key to dm-crypt
   - let paes transform secure key into protected key

3. read/write data
   - dm-crypt uses paes with protected key to decrypt/encrypt data