LUKS2 On-Disk Format Specification
Version 1.0.0

Milan Brož <gmazyland@gmail.com>

Document History

Version	Date	Author
1.0.0	2018-08-02	Milan Broz <gmazyland@gmail.com>
Initial revision for LUKS2. *Not a final version, work in progress.*		
1.0.0	2018-10-23	Milan Broz <gmazyland@gmail.com>
Added segment flags.		

1 Introduction

LUKS2 is the second version of the *Linux Unified Key Setup* for disk encryption management. It is the follow-up of the LUKS1 [1, 2] format that extends capabilities of the on-disk format and removes some known problems and limitations. Most of the basic concepts of LUKS1 remain in place as designed in *New Methods in Hard Disk Encryption* [2] by Clemens Fruhwirth.

LUKS provides a generic key store on the dedicated area on a disk, with the ability to use multiple passphrases[1] to unlock a stored key. LUKS2 extends this concept for more flexible ways of storing metadata, redundant information to provide recovery in the case of corruption in a metadata area, and an interface to store externally managed metadata for integration with other tools.

While the implementation of LUKS2 is intended to be used with Linux-based dm-crypt [3] disk encryption, it is a generic format.

1.1 Design Goals

The LUKS header provides metadata for a disk encryption setup. LUKS1 version [1] contains a binary header for storing necessary metadata (like encryption algorithms parameters) and eight keyslots for independent passphrases to unlock one volume key.
The LUKS2 format is designed to provide these features:

- Cover all possibilities of LUKS1.
- Support configurable memory-hard key-derivation algorithms.
- The new header can store additional metadata for external tools and expose an interface for regular updates.
- LUKS2 uses only a small binary header that can be easily used by automatic detection tools like *blkid* in Linux. The binary header is partially compatible with LUKS1, so legacy tools still recognize a partition as the

[1] LUKS can use a passphrase or a key file, both are processed identically.

LUKS type and can see device UUID. All other metadata are stored in the non-binary header.

- The header includes a checksum mechanism that detects data corruption and unintentional header mangling.
- LUKS header can be detached from a LUKS device and can be stored on a separate device or in a file. With the detached header, the encrypted device contains no visible and detectable metadata.
- Metadata area is stored in two copies to allow for a possible recovery.[2] The recovery is transparent for most of the operations (device should recover automatically if at least one header is correct).
- Keyslot binary area is not duplicated (for security reasons), but the area is now allocated in higher device offset where a random data corruption should happen more rarely.
- A header can be upgraded in-place for most of existing LUKS1 devices.[3]
- Header store persistent flags that are used during activation.[4]
- The number of keyslots is limited only by the provided header area size.[5]
- Keyslots have priorities. Some keyslots can be marked for use only if explicitly specified (for example as a recovery keyslot).
- Metadata are stored in the JSON format that allows for future extensions without modifying binary structures. Such an extension is for example support for authenticated encryption or support for online data reencryption.
- All metadata are algorithm-agnostic and can be upgraded to new algorithms later without header structure changes.
- Volume key digest is no longer limited by 20 bytes (based on legacy SHA-1) as in the LUKS1 header.
- Keyslot can contain an unbound key (key not assigned to any encrypted data segment) that can be used for external applications. Size of an unbound key can be different from volume key size in other keyslots.
- The header contains a concept of *tokens* that are objects, assigned to keyslots, which contain metadata describing *where to get unlocking passphrase*. Tokens can be used for support of external key store mechanisms.

1.2 Reference Implementation

The LUKS2 format is currently implemented and fully supported in libcryptsetup [4] on Linux systems, together with the LUKS1 format. The implementation automatically detects version according to the binary header.

New LUKS2 devices can be created with `--type luks2` option. For example, `cryptsetup format --type luks2 <device>`.

[2] A common issue with LUKS1 is metadata corruption caused by a partitioning tool that does not recognize LUKS format.

[3] A configuration that does not use default on-disk alignment of user data offset could lack needed space for new metadata area.

[4] Example of a persistent flag is support for the TRIM operation. These flags should replace the need for flags in the external `/etc/crypttab` file.

[5] Reference implementation limits the number of keyslots to 32.

2 LUKS2 On-Disk Format

The LUKS2 header is located at the beginning (sector 0) of the block device (for a detached header on a dedicated block device or in a file). The basic on-disk structure is illustrated in Figure 1.

Figure 1: LUKS2 header on-disk structure.

The LUKS2 header contains three logical areas:

- binary structured header (one 4096-byte sector, only 512-bytes are used),
- area for metadata stored in the JSON format and
- keyslot area (keyslots binary data).

The binary and JSON areas are stored twice on the device (primary and secondary header) and under normal circumstances contain same functional metadata. The binary header size ensures that the binary header is always written to only one sector (atomic write). Binary data in the keyslots area is allocated on-demand. There is no redundancy in the binary keyslots area.

2.1 Binary Header

The binary header is intended for a quick scanning by *blkid* and contains a signature to detect LUKS device, basic information (labels), header size and metadata checksum. Binary header in the C structure is described in Figure 2.

All integer values are stored in the big-endian format. All strings are in the C format, and a valid header must have all strings terminated by the zero byte.

The primary binary header must be stored in sector 0 of the device. The secondary header starts immediately after the primary header JSON area (see *hdr_size* in primary header). To allow for an easy recovery, the secondary header must start at a fixed offset listed in Table 1.

Offset (hexa) [bytes]	**JSON area** [kB]
16384 (0x004000)	12
32768 (0x008000)	28
65536 (0x010000)	60
131072 (0x020000)	124
262144 (0x040000)	252
524288 (0x080000)	508
1048576 (0x100000)	1020
2097152 (0x200000)	2044
4194304 (0x400000)	4092

Table 1: Possible LUKS2 secondary header offsets and JSON area size.

```
#define MAGIC_1ST "LUKS\xba\xbe"
#define MAGIC_2ND "SKUL\xba\xbe"
#define MAGIC_L     6
#define UUID_L     40
#define LABEL_L    48
#define SALT_L     64
#define CSUM_ALG_L 32
#define CSUM_L     64

// All integers are stored as big-endian.
// Header structure must be exactly 4096 bytes.

struct luks2_hdr_disk {
  char     magic[MAGIC_L];       // MAGIC_1ST or MAGIC_2ND
  uint16_t version;              // Version 2
  uint64_t hdr_size;             // size including JSON area [bytes]
  uint64_t seqid;                // sequence ID, increased on update
  char     label[LABEL_L];       // ASCII label or empty
  char     csum_alg[CSUM_ALG_L]; // checksum algorithm, "sha256"
  uint8_t  salt[SALT_L];         // salt, unique for every header
  char     uuid[UUID_L];         // UUID of device
  char     subsystem[LABEL_L];   // owner subsystem label or empty
  uint64_t hdr_offset;           // offset from device start [bytes]
  char    _padding[184];         // must be zeroed
  uint8_t  csum[CSUM_L];         // header checksum
  char    _padding4096[7*512];   // Padding, must be zeroed
} __attribute__ ((packed));
```

Figure 2: LUKS2 binary header on-disk structure.

The LUKS1 compatible fields (*magic*, *UUID*) are placed intentionally on the same offsets. The binary header contains these fields:

- **magic** contains the unique string (see C defines `MAGIC_1ST` for the primary header and `MAGIC_2ND` for the secondary header in Figure 2).
- **version** must be set to 2 for LUKS2.
- **hdr_size** contains the size of the header with the JSON data area. The offset and size of the secondary header must match this size. It is a prevention to rewrite of a header with a different JSON area size.
- **seqid** is a counter (sequential number) that is always increased when a new update of the header is written. The header with a higher *seqid* is more recent and is used for recovery (if there are primary and secondary headers with different seqid, the more recent one is automatically used).
- **label** is an optional label (similar to a filesystem label).
- **csum_alg** is a checksum algorithm. Metadata checksum covers both the binary data and the following JSON area and is calculated with the checksum field zeroed. By default, plain SHA-256 function is used as the checksum algorithm.
- **salt** is generated by an RNG and is different for every header (it differs on the primary and secondary header), even the backup header must contain a different salt. The salt is not used after the binary header is read, the main intention is to avoid deduplication of the header sector. The salt must be regenerated on every header repair (but not on a regular update).
- **uuid** is device UUID with the same format as in LUKS1.

- **subsystem** is an optional secondary label.
- **hdr_offset** must match the physical header offset on the device (in bytes). If it does not match, the header is misplaced and must not be used. It is a prevention to partition resize or manipulation with the device start offset.
- **csum** contains a checksum calculated with the *csum_alg* algorithm. If the checksum algorithm tag is shorter than the *csum* field length, the rest of this field must be zeroed.

The rest of the binary header (including padding fields) must be zeroed. The *version*, *UUID*, *label* and *subsystem* fields are intended to be used in the *udev* database and *udev* triggered actions. For example, a system can manage all LUKS2 devices with a specific subsystem field automatically by some external tool. The *label* and *UUID* can be used the same way as a filesystem label.

2.2 JSON Area

The JSON area starts immediately after the binary header (end of JSON area must be aligned to 4096-byte sector offset). Size of JSON area is determined from binary header *hdr_size* field: $JSON\ area\ size = hdr\_size - 4096$.

The area contains metadata in JSON format [5]. The JSON metadata are stored in the area as a C string that must be terminated by the zero character. The unused remainder of the area must be empty and filled with zeroes. The header cannot store larger metadata than this fixed JSON area.[6]

2.3 Keyslots Area

Keyslots area is a reserved space on the disk that can be allocated for a binary data from keyslots. There are stored encrypted keys referenced from keyslots metadata. The structure of stored keyslot binary data depends on the keyslot type. The *luks2* keyslots type uses the same LUKS1 binary structure.

The allocated area is defined in a keyslot by an *area* object that contains *offset* (from the device beginning) and *size* fields. Both fields must be validated to point to the keyslot area. Invalid values must be rejected.

[6] For now, reference implementation uses only areas with 16 kB header (4kB binary header + 12kB JSON area).

3 LUKS2 JSON Metadata Format

The LUKS2 metadata allows defining objects that, according to the *type* field, defines a specific functionality. Objects that are not recognized byt the implementation are ignored, but metadata are still maintained inside the JSON metadata. Implementation must validate the JSON structure before updating the on-disk header.

The LUKS2 structure has 5 mandatory top-level objects (see Figure 3) as follows:

- **config** contains persistent header configuration attributes.
- **keyslots** are objects describing encrypted keys storage areas.
- **digests** are used to verify that keys decrypted from keyslots are correct.
- **segments** describe areas on disk that contain user encrypted data.
- **tokens** can optionally include additional metadata, bindings to other systems – *how to get a passphrase for the keyslot.*

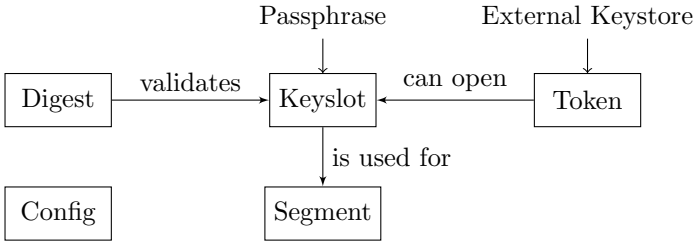

Figure 3: LUKS2 objects schema.

Except top-level objects listed above, all JSON objects must have their names formatted as a string that represents a number in the decimal notation (unsigned integer) – for example *"0"*, *"1"* and must contain attribute *type*. According to the *type*, the implementation decides how to handle (or ignore) such an object. This notation allows mapping to LUKS1 API functions that use an integer as a reference to keyslots objects.

Binary data inside JSON (for example *salt*) are stored in the Base64 [6] encoding. JSON cannot store 64-bit integers directly, a value for an object that represents unsigned 64-bit integer (*offset* or *size*) is stored as a string in the decimal notation and later converted to the 64-bit unsigned integer. Such an integer is referenced as *string-uint64* later.

3.1 LUKS2 JSON Example

The following example contains full JSON metadata from the reference implementation for a LUKS2 device that is encrypted with the AES-XTS cipher, contains two keyslots and one token. The token type is *keyring* (can be unlocked by a passphrase in the keyring) and it is bound to the second keyslot.

```
1 {
2   "keyslots":{
3     "0":{
4       "type":"luks2",
```

```
      "key_size":32,
      "af":{
        "type":"luks1",
        "stripes":4000,
        "hash":"sha256"
      },
      "area":{
        "type":"raw",
        "encryption":"aes-xts-plain64",
        "key_size":32,
        "offset":"32768",
        "size":"131072"
      },
      "kdf":{
        "type":"argon2i",
        "time":4,
        "memory":235980,
        "cpus":2,
        "salt":"z6vz4xK7cjan92rDA5JF8O6Jk2HouV0O8DMB6GlztVk="
      }
    },
    "1":{
      "type":"luks2",
      "key_size":32,
      "af":{
        "type":"luks1",
        "stripes":4000,
        "hash":"sha256"
      },
      "area":{
        "type":"raw",
        "encryption":"aes-xts-plain64",
        "key_size":32,
        "offset":"163840",
        "size":"131072"
      },
      "kdf":{
        "type":"pbkdf2",
        "hash":"sha256",
        "iterations":1774240,
        "salt":"vWcwY3rx2fKpXW2Q6oSCNf8j5bvdJyEzB6BNXECGDsI="
      }
    }
  },
  "tokens":{
    "0":{
      "type":"luks2-keyring",
      "keyslots":[
        "1"
      ],
      "key_description":"MyKeyringKeyID"
    }
  },
  "segments":{
    "0":{
      "type":"crypt",
      "offset":"4194304",
      "iv_tweak":"0",
      "size":"dynamic",
      "encryption":"aes-xts-plain64",
      "sector_size":512
    }
```

```
  },
  "digests":{
    "0":{
      "type":"pbkdf2",
      "keyslots":[
        "0",
        "1"
      ],
      "segments":[
        "0"
      ],
      "hash":"sha256",
      "iterations":110890,
      "salt":"G8gqtKhS96IbogHyJLO+t9kmjLkx+DM3HHJqQtgc2Dk=",
      "digest":"C9JWko5m+oYmjg6ROt/98cGGzLr/4UaG3hImSJMivfc="
    }
  },
  "config":{
    "json_size":"12288",
    "keyslots_size":"4161536",
    "flags":[
      "allow-discards"
    ]
  }
}
```

3.2 Keyslots Object

Keyslots object contains information about stored keys – areas, where binary keyslot data are located, encryption and anti-forensic function used, password-based key derivation function (PBKDF) and related parameters.

Every keyslot object must contain:

- **type** [string] the keyslot type (only the *luks2* is currently used).
- **key_size** [integer] the key size (in bytes) stored in keyslot.
- **area** [object] the allocated area in the binary keyslots area.
- **kdf** [object] the PBKDF type and parameters used.
- **af** [object] the anti-forensic splitter [1] (only the *luks1* type is currently used).
- **priority** [integer,optional] the keyslot priority. Here 0 means *ignore* (the slot should be used only if explicitly stated), 1 means *normal* priority and 2 means *high* priority (tried before *normal* priority).

The *luks2* keyslot type uses the same logic as LUKS1 keyslot, but allows for per-keyslot algorithms (for example different PBKDF).

The *area* object contains these fields:

- **type** [string] the area type (only the *raw* type is currently used).
- **offset** [string-uint64] the offset from the device start to the beginning of the binary area (in bytes).
- **size** [string-uint64] the area size (in bytes).
- **encryption** [string] the area encryption algorithm, in dm-crypt notation (for example `aes-xts-plain64`).
- **key_size** [integer] the area encryption key size.

The *af* (anti-forensic splitter) object contains these fields:

- **type** [string] the anti-forensic function type (only the *luks1* type compatible with LUKS1 [1] is currently used).
- **stripes** [integer] the number of stripes, for historical reasons only the 4000 value is supported.
- **hash** [string] the hash algorithm used (SHA-256).

The LUKS1 AF splitter is no longer much effective on modern storage devices. The functionality is here mainly for compatibility reasons. In future, it will be probably replaced.

The *kdf* object fields are:

- **type** [string] the PBKDF type (*pbkdf2*, *argon2i* or *argon2id* type).
- **salt** [base64] the salt for PBKDF (binary data).

For the *pbkdf2*[7] type (compatible with LUKS1) additional fields are:

- **hash** [string] the hash algorithm for the PBKDF2 (SHA-256).
- **iterations** [integer] the PBKDF2 iterations count.

For *argon2i* and *argon2id*[8] type fields are:

- **time** [integer] the time cost (in fact the iterations count for Argon2).
- **memory** [integer] the memory cost, in kilobytes. If not available, the keyslot cannot be unlocked.
- **cpus** [integer] the required number of threads (CPU cores number cost). If not available, unlocking will be slower.

3.3 Segments Object

Segments object contains a definition of encrypted areas on the disk containing user data (in LUKS1 mentioned as the user data payload). For a normal LUKS device, there is only one data segment present.[9]

The *segment* object contains these fields:

- **type** [string] the segment type (only the *crypt* type is currently used).
- **offset** [string-uint64] the offset from the device start to the beginning of the segment (in bytes).
- **size** [string or string-uint64] the segment size (in bytes) or *dynamic* if the size of the underlying device should be used (dynamic resize).
- **iv_tweak** [string-uint64] the starting offset for the Initialization Vector (IV tweak).
- **encryption** [string] the segment encryption algorithm, in the dm-crypt notation (for example `aes-xts-plain64`).
- **sector_size** [integer] the sector size for segment (512, 1024, 2048 or 4096 bytes).
- **integrity** [object,optional] the LUKS2 user data integrity protection type.
- **flags** [array,optional] the array of string objects marking segment with additional information.

[7] PBKDF2 contains the time cost (iterations) that describes how many times PBKDF2 must iterate to derive the candidate key.

[8] Argon2 algorithms, here used as PBKDF, are memory-hard [7] and have three costs: time, memory required and number of threads (CPUs).

[9] During the data reencryption, the data area is internally divided according to the new and the old key, but only one abstracted area should be presented to the user.

User data integrity protection is an experimental feature [8]) and requires *dm-integrity* and *dm-crypt* drivers with integrity support.

The *integrity* object contains these fields:

- **type** [string] the integrity type (in the dm-crypt notation, for example *aead* or *hmac(sha256)*).
- **journal_encryption** [string] the encryption type for the *dm-integrity* journal (not implemented yet, use *none*).
- **journal_integrity** [string] the integrity protection type for the *dm-integrity* journal (not implemented yet, use *none*).

3.4 Digests Object

The digests object is used to verify that a key decrypted from a keyslot is correct. Digests are assigned to keyslots and segments. If it is not assigned to a segment, then it is a digest for an unbound key. Every keyslot must have one assigned digest object. The key digest object also specifies the exact key size for the encryption algorithm of the segment.

The *digest* object contains these fields:

- **type** [string] the digest type (only the *pbkdf2* type compatible with LUKS1 is used).
- **keyslots** [array] the array of keyslot objects names that are assigned to the digest.
- **segments** [array] the array of segment objects names that are assigned to the digest.
- **salt** [base64] the binary salt for the digest.
- **digest** [base64] the binary digest data.

The *pbkdf2* digest (similar to a kdf object in keyslot) contains these fields:

- **hash** [string] the hash algorithm for PBKDF2 (SHA-256).
- **iterations** [integer] the PBKDF2 iterations count.

3.5 Config Object

The *config* object contains attributes that are global for the LUKS device.

It contains these fields:

- **json_size** [string-uint64] the JSON area size (in bytes). Must match the binary header.
- **keyslots_size** [string-uint64] the binary keyslot area size (in bytes). Must match the binary header.
- **flags** [array, optional] the array of string objects with persistent flags for the device.
- **requirements** [array, optional] the array of string objects with additional required features for the LUKS device.

The *flags* can contain feature and activation flags. Unknown flags are ignored.

The reference implementation uses these flags:

- **allow-discards** allows TRIM (discards) on the active device.
- **same-cpu-crypt** compatibility performance flag for dm-crypt [3].
- **submit-from-crypt-cpus** compatibility flag for dm-crypt [3].

- **no-journal** disable data journalling for dm-integrity [9].

The *requirements* array can contain an array of additional features that are mandatory when manipulating with a LUKS device and metadata or that are required for proper device activation. If an implementation detects a string that it does not recognize, it must treat the whole metadata as read-only and must avoid device activation. These requirements markers are used for future extensions to mark the header to be not backward compatible. Currently, only the *offline-reencrypt* flag is used that marks a device during offline reencryption to prevent an activation until the reencryption is finished.

3.6 Tokens Object

A token is an object that can describe *how to get a passphrase* to unlock a particular keyslot. It can also contain additional user-defined JSON metadata.

The mandatory fields for every token are:

- **type** [string] the token type (tokens with *luks2-* prefix are reserved for the implementation internal use).
- **keyslots** [array] the array of keyslot objects names that are assigned to the token.

The rest of the JSON content is the particular token implementation and can contain arbitrary JSON structured data (implementation should provide an interface to the JSON metadata directly).

For example, the reference implementation of *luks2-keyring* token allows automatic activation of the device if the passphrase is preloaded into a keyring with the specified ID.

The *luks2-keyring* token type contains these fields:

- **type** [string] is set to the *luks2-keyring*.
- **keyslots** [array] is assigned to the specific keyslot(s).
- **key_description** [string] contains the ID of the keyring entry with a passphrase.

4 LUKS2 Operations

Basic operations of a LUKS2 device are the same as specified in LUKS1 [1]. Header update operations must be synchronized due to the redundancy of metadata and operations must be serialized to prevent concurrent processes from updating the metadata at the same time. The metadata update must be implemented in such a way that at least of one header (primary or secndary) is always valid to allow for a proper recovery in the case of a failure. These steps require an implementation of some high-level locking of metadata access.

4.1 Device Formatting

Initialization (formatting) of a LUKS2 device starts with generating basic metadata parameters, like *UUID* and writing both binary headers and basic metadata structures. The JSON area must always contain valid LUKS2 top-level objects. The config object must be initialized to include proper area size parameters that match the binary header. If the LUKS2 header references a user data segment,

that segment must be initialized with all mandatory parameters. The keyslots, digests and tokens can be empty in this step.

4.2 Keyslot Initialization

The next step is allocation of new keyslot and metadata and assignment to the key digests and segments. The key stored in the keyslot and salt for the keyslot should be generated using a cryptographically secure RNG.

The PBKDF cost parameters (iterations, memory, CPU cores) that are used to derive the keyslot unlocking key from a user passphrase must be either specified by the user, or it can be benchmarked according to user needs. See the reference cryptsetup implementation [4] as an example of this approach.

Once the key is generated, a new key digest is created, and a new keyslot object is allocated and assigned to the digest (and segment). The new keyslot contains the binary keyslot area allocated according to the stored key size.

The size of the binary allocated area is determined according to the key size and the anti-forensic (AF) splitter output (see section 2.4 in LUKS1 [1]).

LUKS2 keyslots can store different keys with different key sizes. The allocation of binary keyslot data depends on the order of creation. Keyslot positions are no longer fixed as in LUKS1.

The last step of keyslot initialization writes the encrypted key to the allocated binary keyslot area. A user passphrase and a salt are processed by the configured PBKDF. The PBKDF output key is used for the keyslot binary area encryption algorithm. The key is split using AF splitter and encrypted by the keyslot encryption algorithm.

4.3 Keyslot Content Retrieval

The user provided passphrase with the salt and parameters from the header metadata are processed through the PBKDF. The derived key is used to decrypt the binary keyslot area. The decrypted content is processed (merged) in the AF splitter. The assigned key digest is calculated with the recovered candidate key. If the calculated digest and the digest in metadata match, the recovered key is valid. If the digest does not match, the provided passphrase must be rejected.

4.4 Keyslot Revocation

To discard a keyslot, the binary area for the keyslot must be physically overwritten (to discard the stored data). After this step, the keyslot metadata object must be removed with all bindings to digests and segments.

Note that the key digest and its binding to the segment can remain in metadata (not assigned to any keyslots). If a user has the copy of the encryption key, the validity of the key can still be verified with this digest and the device can be later still activated.

4.5 Metadata Recovery

The replicated metadata allows for a full LUKS2 header recovery (except binary keyslot areas) if some part of headers become corrupted. A part of the recovery

can be automated, but because this process can revert some intentional changes, a user interaction is suggested.

The automatic recovery should always update both copies to the more recent version (with higher *seqid*). The metadata handler should first try to load the primary header, then the secondary header. If one of the headers is more recent, the older header is updated. If the primary header is corrupted, a scan on several known offsets for the secondary header can be performed.

4.6 Mandatory Requirements

While the LUKS2 format is algorithm-agnostic, some algorithm implementations are crucial for internal function.

The cryptographic backend for LUKS2 must support these algorithms:

- **SHA-1** hash algorithm (for compatibility with old LUKS1 devices).
- **SHA-256** hash algorithm, used as the default checksum for the binary header and in the PBKDF2 digest.
- **PBKDF2** password-based key derivation (for digest and backward compatibility with LUKS1).
- **Argon2i and Argon2id** memory-hard key derivation functions for new LUKS2 keyslots.
- **AES-XTS** symmetric cipher for the default keyslot encryption and the default user data encryption.

4.7 Conversion from LUKS1

If an existing LUKS1 device header contains enough space for the LUKS2 metadata, then it can be converted in-place to the LUKS2 format. Reference implementation provides the `cryptsetup convert --type luks2` command.

LUKS1 [name]	**128-bit key** [sectors]	**256-bit key** [sectors]	**512-bit key** [sectors]
Header	0	0	0
Keyslot 0	8	8	8
Keyslot 1	136	264	512
Keyslot 2	264	520	1016
Keyslot 3	392	776	1520
Keyslot 4	520	1032	2024
Keyslot 5	648	1288	2528
Keyslot 6	776	1544	3032
Keyslot 7	904	1800	3536
Padding	1032	2056	4040
Data offset	2048	4096	4096
Unused sectors	1016	2040	56

Table 2: Offsets (in 512-byte sectors) of common LUKS1 headers.

For reference, Table 2 contains offsets of LUKS1 keyslots that can be converted to LUKS2 in-place. The resulting LUKS2 header has 12kB JSON area in

all these cases. Note that the binary keyslot area is directly copied to the proper position, there is no recovery possible if the convert operation fails. Schema of area locations during conversion is illustrated in Figure 4.

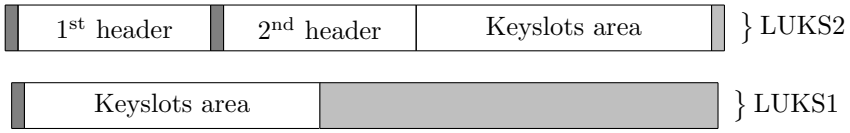

Figure 4: LUKS1/LUKS2 areas placement.

If a LUKS2 header uses compatible options with LUKS1 (PBKDF2, no integrity protection, no tokens, no unbound keys) then it can also be converted back to the LUKS1 header in-place with the `cryptsetup convert --type luks1` command.

4.8 Algorithm Definition Examples

The LUKS2 specification supports all algorithms that are provided by the cryptographic backend (in the Linux case by kernel dm-crypt and userspace cryptographic library). Figures 3 and 4 list few examples of symmetric ciphers for data encryption and PBKDF algorithms.

Algorithm in LUKS2 notation	**Description**
pbkdf2	PBKDF2 with HMAC-SHA256 [10]
argon2i	Argon2i as PBKDF (data independent) [7]
argon2id	Argon2id as PBKDF (combined mode) [7]

Table 3: LUKS2 PBKDF algorithms.

Algorithm in dm-crypt [3] notation	**Description**
aes-xts-plain64	AES in XTS mode with sequential IV [11, 12]
aes-cbc:essiv:sha256	AES in CBC mode with ESSIV IV [3, 11]
serpent-xts-plain64	Serpent cipher with sequential IV [13]
twofish-xts-plain64	Twofish cipher with sequential IV [14]
aegis128-random	AEGIS (128-bit) with random IV, AEAD [15]
aegis256-random	AEGIS (256-bit) with random IV, AEAD [15]
morus640-random	MORUS640 with random IV, AEAD [16]
morus1280-random	MORUS1280 with random IV, AEAD [16]

Table 4: LUKS2 encryption algorithms examples.
AEAD algorithms are experimental and require dm-integrity [9] support.

Glossary

AEAD Authenticated Encryption with Additional Data.

AF Anti-Forensic splitter defined for LUKS1. [1, 17]

Base64 Binary to text encoding scheme. [6]

blkid Utility to locate and print block device attributes.

dm-crypt Linux device-mapper crypto target. [3]

dm-integrity Linux device-mapper integrity target [9].

IV Initialization Vector for an encryption mode that tweaks encryption.

JSON JavaScript Object Notation (data-interchange format). [5]

Keyslot Encrypted area on disk that contains a key.

Length-preserving encryption Symmetric encryption where plaintext and ciphertext have the same size.

libcryptsetup Library implementing LUKS1 and LUKS2. [4]

Metadata locking A way how to serialize access to on-disk metadata updates.

PBKDF Password-Based Key Derivation Function.

RNG Cryptographically strong Random Number Generator.

Sector Atomic unit for block device (disk). Typical sector size is 4096 bytes.

TRIM Command that informs a block device that area of the disk is unused and can be discarded.

udev Device manager for Linux kernel implemented in userspace.

UUID Universally Unique IDentifier (of a block device).

Volume Key The key used for data encryption on disk. Sometimes called as Media Encryption Key (MEK).

References

[1] LUKS1 On-Disk Format Specification, Version 1.2.3, 2018. `https://gitlab.com/cryptsetup/cryptsetup/wikis/Specification`.

[2] Clemens Fruhwirth. *New methods in hard disk encryption.* PhD thesis, Institute for Computer Languages Theory and Logic Group, Vienna University of Technology, 2005. `http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf`.

[3] dm-crypt: Linux device-mapper crypto target, 2018. `https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt`.

[4] Cryptsetup and LUKS, 2018. `https://gitlab.com/cryptsetup/cryptsetup`.

[5] The JSON Data Interchange Format. Technical Report Standard ECMA-404, 1st edition, ECMA, 2013. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[6] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, 2006. `https://www.ietf.org/rfc/rfc4648.txt`.

[7] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications, 2017. `https://www.cryptolux.org/index.php/Argon2`.

[8] Milan Brož, Mikuláš Patočka, and Vashek Matyáš. Practical Cryptographic Data Integrity Protection with Full Disk Encryption Extended Version, 2018. `https://arxiv.org/abs/1807.00309`.

[9] dm-integrity: Linux device-mapper integrity target, 2018. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMIntegrity.

[10] Burt Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), 2000. https://www.ietf.org/rfc/rfc2898.txt.

[11] FIPS Publication 197, The Advanced Encryption Standard (AES), 2001. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf.

[12] Morris J. Dworkin. SP 800-38E. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices, 2010. NIST, https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38e.pdf.

[13] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. https://www.cl.cam.ac.uk/~rja14/serpent.html.

[14] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-bit Block Cipher*. John Wiley & Sons, Inc., 1999. https://www.schneier.com/academic/twofish/.

[15] Hongjun Wu and Bart Preneel. AEGIS, A Fast Authenticated Encryption Algorithm (v1.1). Technical report, 2016. https://competitions.cr.yp.to/round3/aegisv11.pdf.

[16] Hongjun Wu and Tao Huang. The Authenticated Cipher MORUS (v2). Technical report, 2016. https://competitions.cr.yp.to/round3/morusv2.pdf.

[17] Clemens Fruhwirth. TKS1 - An anti-forensic, two level, and iterated key setup scheme, 2004. https://www.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/TKS1-draft.pdf.