Design and Implementation of a Windows Kernel Driver for LUKS2-encrypted Volumes

I do not know yet whether I want to have a subtitle, have a placeholder for now

Max Ihlenfeldt

Universität Augsburg
Lehrstuhl für Organic Computing
Bachelorarbeit im Studiengang Informatik

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

1 Introduction

Explain use case etc.

Note that in this thesis the terms *disk*, *drive*, *volume* and *partition* are used somewhat loosely and probably mean roughly the same.

2 Background

2.1 LUKS2 Disk Encryption

also [1]

Linux Unified Key Setup 2, or short LUKS2, is the second version of a disk encryption standard. It provides a specification [2] for a on-disk format for storing the encryption metadata as well as the encrypted user data. Unlocking an encrypted disk is achieved by providing one of possibly multiple passphrases or keyfiles. The intended usage of LUKS2 is together with the Linux dm-crypt subsystem, but that is not mandatory[1].

The reference implementation[2] is designed only for usage on Linux, which is why we developed a new Rust library for interacting with LUKS2 partitions. This is not a full equivalent, but only a cross-platform helper. Its task is to take care of all the cryptographic work needed before actually decrypting and encrypting data. Notably, it lacks the following features of the reference implementation:

- creating new or modifying existing LUKS2 partitions,
- converting a LUKS to a LUKS2 partition,
- actually mounting a LUKS2 partition for read/write usage; this is accomplished by our kernel driver.

It does provide access to the raw decrypted user data, but the practical use of this is very limited: the decrypted data is in the format of a filesystem, e.g. FAT32, btrfs, or Ext4. Therefore a filesystem driver is needed to actually access the stored files. One way of exposing the decrypted data to the system's filesystem drivers is by transparently decrypting the data directly in the kernel, which is what our driver does (see section 5).

The differences between the original LUKS and LUKS2 are minor. According to [2], LUKS2 adds "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." Practically, this means that LUKS2 has a different on-disk layout and, among other things, supports more password hashing algorithms (more precisely, password-based key derivation functions).

Mention own `luks2` Rust crate

2.1.1 On-Disk Format

Figure 1 shows the high-level layout of a LUKS2-encrypted disk.

The two binary headers have a size of exactly one sector, so that they are always written atomically. Only the first 512 bytes are actually used. The header marks the disk as following the LUKS2 specification and contains metadata such as labels, a UUID, and a header checksum. The labels and UUID can be accessed using the `blkid`[3] command-line tool and also be used in the `udev`[4] Linux subsystem. For the detailed contents, see Figure 2. Figure 3 also contains an example hexdump of a binary header.

The sector containing the binary header is followed by the JSON area. This area arguably contains the metadata that is most relevant for decryption and encryption.

[1] As we show in this thesis it is possible to make the combination of LUKS2 and Windows work.
[2] `https://gitlab.com/cryptsetup/cryptsetup`
[3] `https://linux.die.net/man/8/blkid`
[4] `https://linux.die.net/man/8/udev`

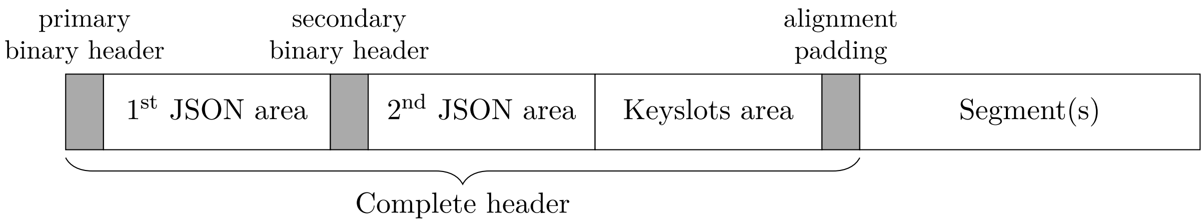

Figure 1: LUKS2 on-disk format (modified after [2]). The complete header consists of three areas: a binary header of exactly one 4096-byte sector, JSON metadata, and the binary keyslots data. A *keyslot* is an "encrypted area on disk that contains a key" [2]. For redundancy, the binary header and the JSON metadata are stored twice. After that follow one or areas containing encrypted user data. The specification calls these areas *segments*.

Figure 4 contains an overview of the objects stored in JSON and their relationships. For this thesis' brevity's sake, please refer to Chapter 3.1 in [2] for an example of a LUKS2 JSON area.

For simplicity, our LUKS2 Rust library does not support unlocking a keyslot using an external keystore defined by a token. Only unlocking via password is implemented. The library does however include support for different *password-based key derivation functions (PBKDFs)*, namely `pbkdf2` with SHA-256, `argon2i`, and `argon2id`. These are all the PBKDF algorithms that are listed in the LUKS2 specification (see [2], Table 3).

2.2 Introduction to Windows Kernel Driver Development

This section gives an introduction on the development of Windows kernel drivers and related important concepts.

2.2.1 Structure and Hierarchy of the Windows Operating System

Roughly summarize important concepts from chapters 1 and 2 of [3]

2.2.2 The Windows Driver Model for Kernel Drivers

Also explain how it gets loaded (if not done already)

2.2.3 Communication Between Kernel and Userspace

Via ports

```c
#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

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 structure from [2]. Integers are stored in big-endian format, and all strings have to be null-terminated. The `magic`, `version`, and `uuid` fields are also present in the LUKS1 binary header and were placed at the same offsets as there.

3 Related Work

3.1 Measuring Filesystem Driver Performance

3.2 Cryptographic Aspects of LUKS2

[4]

Search for more papers, e.g. attacks against LUKS?

0000	4C	55	4B	53	BA	BE	00	02	00	00	00	00	00	00	40	00	LUKSº¾........@.
0010	00	00	00	00	00	00	00	03	54	68	69	73	20	69	73	20	........This is
0020	61	6E	20	41	53	43	49	49	20	6C	61	62	65	6C	00	00	an ASCII label..
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0040	00	00	00	00	00	00	00	00	73	68	61	32	35	36	00	00	........sha256..
0050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0060	00	00	00	00	00	00	00	00	EB	0F	D2	C6	E3	D2	8D	4B	........ë.ÒÆãÒ.K
0070	BB	2B	8A	49	E6	2E	4E	B7	04	2F	A9	39	76	71	8F	8A	»+ŠIæ.N·./©9vq.Š
0080	33	E8	F3	90	FF	DC	4D	3D	E8	30	7B	37	01	30	E7	5D	3èó.ÿÜM=è0{7.0ç]
0090	AD	A0	57	1C	0E	63	BC	D4	DD	3C	EC	F5	DE	67	F8	D8	..W..c¼ÔÝ<ìõÞgøØ
00A0	F2	7E	82	CD	B9	DD	77	10	65	39	33	64	63	61	66	61	ò~‚Í¹Ýw.e93dcafa
00B0	2D	65	65	30	62	2D	34	31	36	38	2D	61	61	37	63	2D	-ee0b-4168-aa7c-
00C0	66	33	30	34	37	34	38	38	36	61	32	65	00	00	00	00	f30474886a2e....
00D0	54	68	69	73	20	69	73	20	61	6E	20	6F	70	74	69	6F	This is an optio
00E0	6E	61	6C	20	73	65	63	6F	6E	64	61	72	79	20	6C	61	nal secondary la
00F0	62	65	6C	00	00	00	00	00	00	00	00	00	00	00	00	00	bel.............
0100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
01A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
01B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
01C0	91	A4	A9	83	03	FF	FB	68	4E	C2	94	6F	4C	78	71	AF	‘¤©f.ÿûhNÂ"oLxq¯
01D0	AE	1A	91	F8	E0	2C	F3	71	D5	17	CB	60	E5	2F	D6	36	®.‘øà,óqÕ.Ë‘å/Ö6
01E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
01F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Figure 3: LUKS2 binary header example. The fields, as described in Figure 2, were coloured differently to be easily distinguishable. A similar header, although with different salt and hash, can be generated by executing `fallocate -l 16M luks2.img && cryptsetup luksFormat --label 'This is an ASCII label' --subsystem 'This is an optional secondary label' --uuid e93dcafa-ee0b-4168-aa7c-f30474886a2e luks2.img` in a Linux shell.

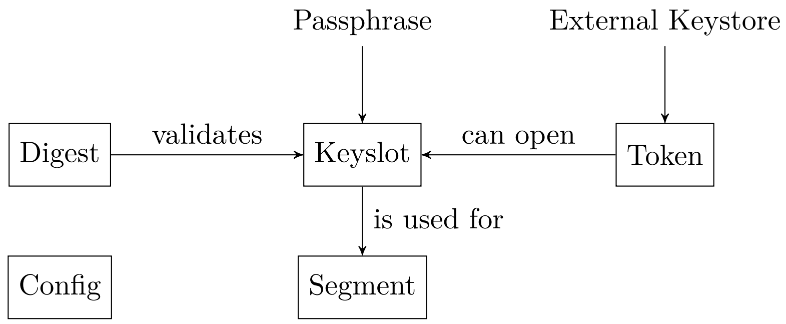

Figure 4: LUKS2 object schema from [2]. The most important objects are the following: *keyslots*, which describe the details of how cryptographic keys are stored and encrypted; *digests*, which can be used to verify that one has successfully extracted a key from a keyslot; and *segments*, which describe the disk areas where the encrypted user data is stored. Figure 1 shows where the areas described by the keyslot and segment objects actually lie on disk.

4 Other Approaches

4.1 Linux Kernel Implementation of LUKS2

4.2 Other Implementations of Encrypted Filesystems

4.2.1 VeraCrypt

4.2.2 BitLocker

[5] and [6] and [7] and [8]

5 Design and implementation of our approach

5.1 Failed Attempts

FilterManager framework

Mention KMDF / UMDF and why we didn't use that if not already done in earlier section

5.2 The Final WDM Driver

Why WDM?

5.2.1 Architecture

5.2.2 Initialization and Configuration

`luks2filterstart.exe`

5.2.3 De-/encrypting Reads and Writes

custom AES implementation

LUKS2 supports many encryption algorithms (see [2], Table 4), but `luks2flt` only supports `aes-xts-plain64`.

```
VOID
EncryptWriteBuffer(
    PUINT8 Buffer,
    PLUKS2_VOLUME_INFO VolInfo,
    PLUKS2_VOLUME_CRYPTO CryptoInfo,
    UINT64 OrigByteOffset,
    UINT64 Length
)
{
    UINT64 Sector = OrigByteOffset / VolInfo->SectorSize;
    UINT64 Offset = 0;
    UINT8 Tweak[16];

    while (Offset < Length) {
        ToLeBytes(Sector, Tweak);
        CryptoInfo->Encrypt(
            &CryptoInfo->Xts, Buffer + Offset,
            VolInfo->SectorSize, Tweak
        );
        Offset += VolInfo->SectorSize;
        Sector += 1;
    }
}
```

5.2.4 Handling Other Request Types

5.3 Security Considerations

How does `cryptsetup` send the master key to `dm-crypt`?

6 Performance of Our Driver

6.1 Experimental Setup

6.2 Results

7 Discussion

8 Conclusion

List of Figures

List of Tables

References

[1] C. Fruwirth, *LUKS1 On-Disk Format Specification Version 1.2.3*, 2018, visited on 2021-07-31. [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/on-disk-format.pdf

[2] M. Broz, *LUKS2 On-Disk Format Specification Version 1.0.0*, 2018, visited on 2021-07-31. [Online]. Available: https://gitlab.com/cryptsetup/LUKS2-docs/-/raw/861197a9de9cba9cc3231ad15da858c9f88b0252/luks2_doc_wip.pdf

[3] P. Yosifovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more.* Microsoft Press, 2017.

[4] C. Fruhwirth, "New methods in hard disk encryption," Ph.D. dissertation, Vienna University of Technology, 2005.

[5] J. D. Kornblum, "Implementing bitlocker drive encryption for forensic analysis," *Digital Investigation*, vol. 5, no. 3-4, pp. 75–84, 2009.

[6] S. G. Lewis and T. Palumbo, "Bitlocker full-disk encryption: Four years later," in *Proceedings of the 2018 ACM SIGUCCS Annual Conference*, 2018, pp. 147–150.

[7] S. Türpe, A. Poller, J. Steffan, J.-P. Stotz, and J. Trukenmüller, "Attacking the bitlocker boot process," in *International Conference on Trusted Computing.* Springer, 2009, pp. 183–196.

[8] C. Tan, L. Zhang, and L. Bao, "A deep exploration of bitlocker encryption and security analysis," in *2020 IEEE 20th International Conference on Communication Technology (ICCT).* IEEE, 2020, pp. 1070–1074.