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

Enabling read-only access to LUKS2 partitions on Windows and comparing the performance to other disk encryption technologies

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 e

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

Linux Unified Key Setup 2, or short LUKS2, is the second version of a disk encryption standard. It provides a specification [1] 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 differences between the original LUKS and LUKS2 are minor. According to [1], 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).

The reference implementation[2] is designed only for usage on Linux, which is why we developed a new library in Rust 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 (more precisely, the process described in section 2.1.2). Notably, it lacks the following features of the reference implementation:

- formatting new LUKS2 partitions,
- modifying or repairing existing LUKS2 partitions,
- converting a LUKS partition to a LUKS2 partition,
- actually mounting a LUKS2 partition for read/write usage (this is what our kernel driver and its userspace configuration tool is for).

Our library 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).

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 contains the metadata that is arguably 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`

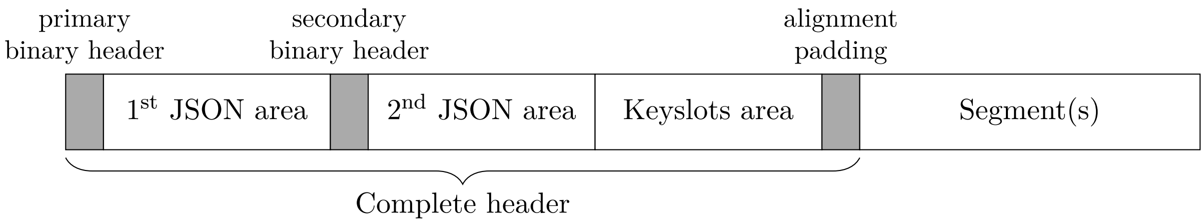

Figure 1: LUKS2 on-disk format (modified after [1]). The complete header consists of three areas: a binary header of exactly one 4096-byte sector, JSON metadata, and the binary keyslots data. A *keyslot* is an “encrypted area on disk that contains a key” [1]. For redundancy, the binary header and the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

```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 [1]. 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.

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 [1] for an example of a LUKS2 JSON area.

After the JSON area, the keyslots area is stored on the disk. This is space reserved for storing encrypted cryptographic keys. The metadata from the JSON keyslot objects describe the position of a key on the disk as well as information on how to decrypt it.

```
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  ‘¤©ƒ.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.

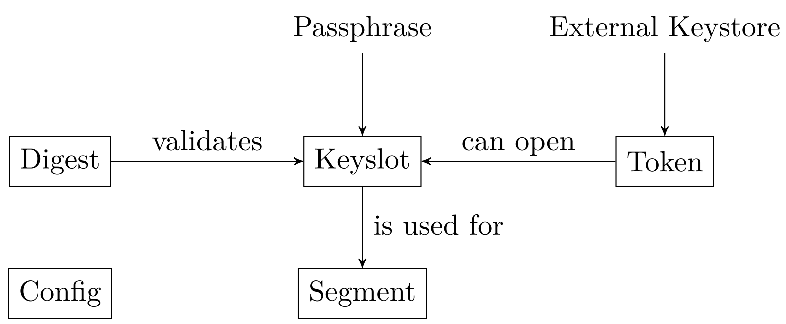

Figure 4: LUKS2 object schema from [1]. The most important objects are the following: *keyslots*, which describe the details of how cryptographic keys are stored and encrypted; *digests*, which can be used to verify that one has successfully extracted a key from a keyslot; and *segments*, which describe 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.

2.1.2 Unlocking a Partition

For simplicity, our LUKS2 Rust library does not support unlocking a keyslot using an external keystore defined by a token, even though Figure 4 mentions this possibility. 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 [1], Table 3). The default PBKDF used by LUKS2 is `argon2i` [2]. In the following paragraphs, it will become apparent what PBKDFs are used for in LUKS2[5].

The LUKS2 specification allows for multiple segments in one partition. To make things easier, our driver only supports unlocking one segment. Therefore, in this thesis we may speak of unlocking a partition and mean unlocking one of the partition's segments.

To unlock a segment means to derive the cryptographic key that is needed for reading decrypted or writing encrypted data. This key is called the segment's *master key*.

LUKS2 uses a process called *anti-forensic splitting* to store the master key on disk. This method was introduced in [3]. It is used to diffuse the key's bytes into a longer sequence of bytes that has the following property: if at least one bit of the diffused sequence is changed, the key cannot be recovered. This is achieved by a clever combination of XOR and a hash function. The motivation behind this to make it easier (or possible) to dispose of an old key in such a way that it cannot be recovered from the disk. This is because it is much more feasible to partially erase a long sequence of bytes than to completely erase a short sequence. Erasing here means to overwrite the data in such a way that it cannot be recovered, which is not as trivial as one might think.

[3] calls the operation that splits data anti-forensically AFsplit and the recovery operation AFmerge. We will adhere to this convention (with slight variations).

To necessitate the need of a password to recover the key, the data is also encrypted before it gets written to the disk. The encryption key is a hash of the password obtained by a PBKDF.

The properties of anti-forensic splitting can be used when the user wants to change the password: the master key is derived using the old password and then re-encrypted with the new password. The key as it was encrypted with the old password can then be destroyed.

[3] presents two templates for storing keys, TKS1 and TKS2. The difference is whether the key is encrypted before or after splitting it. LUKS and LUKS2 use TKS2, which is schematically explained in Figure 5. The hash function used by LUKS2 for anti-forensic splitting is SHA-256. Figure 6 shows the outline of an implementation of TKS2 in Rust.

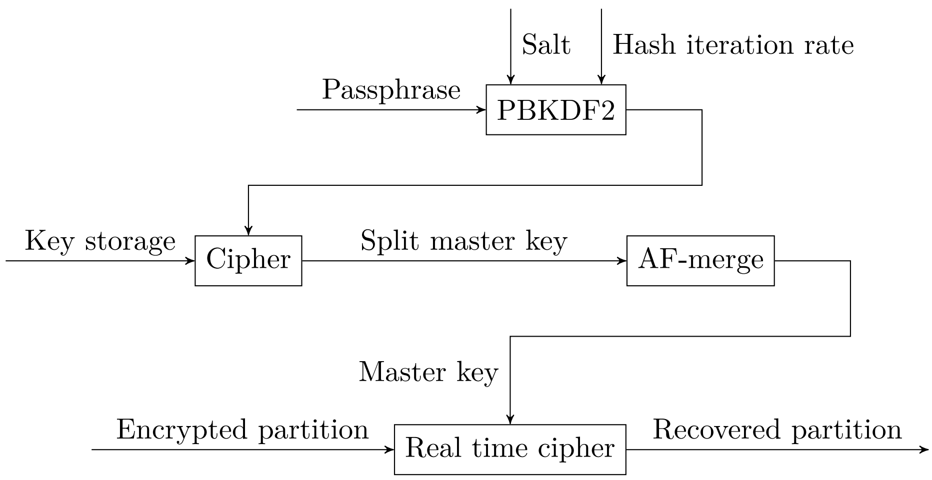

Figure 5: TKS2 scheme (modified after [3]).

[5] It will also turn out that they are quite aptly named.

```rust
1 fn decrypt_keyslot(
2     password: &[u8], keyslot: &LuksKeyslot, json: &LuksJson, /* ... */
3 ) -> Result<Vec<u8>, LuksError> {
4     let mut k = vec![
5         0; keyslot.key_size() as usize * keyslot.af.stripes() as usize
6     ];
7     // read keyslot data from disk into k...
8 
9     let mut pw_hash = vec![0; area.key_size() as usize];
10     match keyslot.kdf() {
11         // hash into pw_hash using pbkdf2, argon2i, or argon2id...
12     }
13 
14     // decrypt keyslot area using the password hash as key
15     match area.key_size() {
16         32 => {
17             let key1 = Aes128::new_varkey(&pw_hash[..16]).unwrap();
18             let key2 = Aes128::new_varkey(&pw_hash[16..]).unwrap();
19             let xts = Xts128::<Aes128>::new(key1, key2);
20             xts.decrypt_area(&mut k, sector_size, 0, get_tweak_default);
21         },
22         // 64 byte key uses AES256 instead...
23     }
24 
25     // merge and hash master key
26     let master_key = af::merge(
27         &k, keyslot.key_size() as usize, af.stripes() as usize
28     );
29     let digest_actual = base64::decode(json.digests[&0].digest())?;
30     let mut digest_computed = vec![0; digest_actual.len()];
31     let salt = base64::decode(json.digests[&0].salt())?;
32     pbkdf2::pbkdf2::<Hmac<Sha256>>(
33         &master_key, &salt, json.digests[&0].iterations(), &mut digest_computed
34     );
35 
36     // compare digests
37     if digest_computed == digest_actual {
38         Ok(master_key)
39     } else {
40         Err(LuksError::InvalidPassword)
41     }
42 }
```

Figure 6: LUKS2 master key decryption in Rust. Some values are hardcoded: only the digest with index 0 is used (lines 29, 31, 33), and it is assumed that the digest algorithm is always `pbkdf2` with SHA-256 (line 32). The latter is compliant with the specification, which lists this digest algorithm as the only option, but not optimal in the sense of input validation.

The keyslots area on the disk is large enough to store multiple split and encrypted master keys. Thus one can configure a LUKS2 partition to be unlocked by different passwords. Unlocking then works as described in Figure 7.

1. The user supplies a password.
2. One of the available keyslots is selected.
3. Using the password and keyslot, the master key is decrypted as described above.
4. The derived master key is hashed and the result compared to the corresponding digest. Which digest and what hash parameters to used is defined in the JSON section.
5. If the digests match, the master key has been successfully decrypted. Else go to step 2 and select a keyslot that has not been used yet.
6. If the master key could not be decrypted with all available keyslots, the supplied password was not correct.

Figure 7: LUKS2 master key decryption with multiple available keyslots. LUKS2 allows defining priorities that govern the order in which the available keyslots are tried. For a more detailed pseudocode see Figure 5 in [4].

2.1.3 Using an Unlocked Partition

After a partition has been unlocked, i.e. after the master key of one of its segments has been decrypted, the partition is ready to be read from and written to. This happens using what Figure 5 calls a real time cipher. LUKS2 supports different encryption algorithms for this purpose, see Figure 8 for a selection of them. Our driver only supports the default aes-xts-plain64 encryption. Therefore we will focus on that in this section.

Algorithm in `dm-crypt` notation	**Description**
aes-xts-plain64	AES in XTS mode with sequential IV
aes-cbc-essiv:sha256	AES in CBC mode with ESSIV IV
serpent-xts-plain64	Serpent cipher with sequential IV
twofish-xts-plain64	Twofish cipher with sequential IV

Figure 8: Selection of LUKS2 encryption algorithms (modified after [1]). The `dm-crypt` notation follows this format: cipher[:keycount]-chainmode-ivmode[:ivopts] [5]. See [6] for the CBC mode and the Serpent and Twofish ciphers, and [3] for the ESSIV IV mode.

The AES encryption algorithm is a block cipher for processing 128 bit data blocks [7]. This means that AES takes a key and 128 bits, or 16 bytes, of data, and generates 16 bytes of ciphertext. Using the same key, this ciphertext can be decrypted back to the original plaintext data. The key can be 128, 192, or 256 bits long. The three variants of AES, each using a different key size, are called AES-128, AES-192, and AES-256.

To encrypt data longer than one block, a *block cipher mode* is needed [6]. These are encryption functions that build on a existing block cipher. When using aes-xts-plain64, LUKS2 uses the XTS block cipher mode. The defining IEEE standard [8] describes it as follows: “XTS-AES is a tweakable block cipher that acts on data units of 128 b[its] or more and uses the AES block cipher as a subroutine. The key material for XTS-AES consists of a data encryption key (used by the AES block cipher) as well as a 'tweak key' that is used to incorporate the logical position of the data block into the encryption." This tweak key is called the *initialization vector*, or IV, in the context of LUKS2. This is what the "plain64" in aes-xts-plain64 means: "the initial vector is the 64-bit little-endian version of the sector number, padded with zeros if necessary" [5]. This sector number is relative to the first sector of the segment, i.e. the first sector uses an IV of 0.

The need for an IV arises from a critical problem that occurs when encrypting each block separately with the same key: if some unencrypted blocks are identical, then so will be their encrypted counterparts. This can lead to leaked information about structure and contents of the plaintext [6].

All this theory may sound complicated, but in section 5.2.3 we will see that the practical usage of cryptography in our driver is quite simple[6].

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

First, we will introduce some basic concepts of the Windows operating system that will be relevant in the following sections. Please note that we will focus on Windows 10 running on the x64 architecture, as that is the target of our driver. Some details may be different in other versions of Windows, though most information should still be valid.

The definitive reference for all detailed information on the inner workings of Windows is [9]. The online documentation of the Windows SDK[7] and the Windows Driver Kit[8] (WDK) both provide valuable information. It may also sometimes be useful to directly look at the data structure definitions provided by the WDK header files (`ntddk.h`, `ntifs.h`, and `wdm.h` are the most useful) [9].

There are also some invaluable tools to directly take a look at OS internals. Most of them are from the Sysinternals[9] Suite, others are included with Windows. See table 1-4 in [9] for a more detailed compilation.

The Windows *registry* is a database for storing system-wide and per-user configuration. It can also be used to query the current state of the system, e.g. performance counters or information on loaded device drivers [9]. Each data entry in the registry, called a *value*, has a path, also known as its *key*, and a name. This name is used to distinguish different entries stored under the same key. The keys are organized hierarchically in a tree, similar to file paths [11].

Windows uses the physically available RAM to back its 64-bit address space of virtual memory. See Figure 9 for how the address range is used. The addresses are grouped into so-called *pages*, which are typically 4KB large. Most systems have less physical RAM available than the sum of virtual memory used by all processes. The memory manager solves this problem by transferring pages currently not in use to disk. This is called *paging*. When a virtual address in a page that currently resides on disk is

[6] The implementation of cryptographic algorithms like the XTS mode of course remains non-trivial. Implementing AES itself has been made much easier using the AES-NI CPU instruction set, though.

[7] `https://docs.microsoft.com/en-us/windows/apps/desktop`

[8] `https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi`

[9] The executables can be downloaded from `https://live.sysinternals.com` and are accompanied by documentation in [10].

accessed, a *page fault* occurs. In that case, the needed page is loaded back into memory. This process is completely transparent to applications, aside from latency introduced by paging. However, this will become relevant when writing drivers, as driver code may be called in situations where page faults are not allowed (see section 2.2.2 for more) [9].

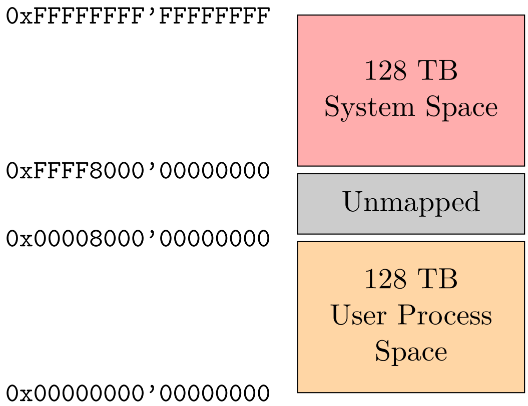

Figure 9: Address space layout (not to scale) for 64-bit Windows 10 (modified after [9]). Each process has its own copy of the user process space[10].

Figure 10 shows a simplified view of the architecture of Windows. It also differentiates between *user mode* and *kernel mode*. The difference is that when a process is running in kernel mode, it has access to all CPU instructions, the whole system memory. Kernel mode also allows direct access to hardware. User mode, on the other hand, only allows access to a limited subset of all that. To protect the OS from user applications and to also isolate different programs from each other, user applications run in user mode. Kernel mode is reserved for OS code, such as drivers and system services. This separation ensures that an incorrectly programmed or malicious program cannot jeopardize the whole system's stability and/or data integrity [9].

The permission checks for user mode are enforced by the processor. Switching between user and kernel mode is achieved by executing a special CPU instruction called `syscall`. Prior to that, the user code specifies which system service it wants the kernel to execute[11]. When kernel mode is activated, control is transferred from the application to a special part of the operating system. Its purpose is to dispatch control to the part of the OS that implements the requested system service. When that has been completed, the OS orders the processor to switch back into user mode and hands control back to the application [9].

Developers of regular applications will not come into contact with syscalls. Normally, these are the responsibility of Windows API subroutines. For example, the `CreateProcess` function will instruct the OS to create a new process by executing the `NtCreateUserProcess` syscall. Because they are not meant to be used by non-OS code, these system calls are not (officially) documented[12] [9].

As mentioned before, each user process has its own private address space. In contrast to that, all kernel-mode OS components and device drivers share one area of memory.

[10] Sven Peter has a nice quote on this topic on his blog: "One of the kernel's tasks is to lie to each application running in userland and to tell them that they're the only one in the address space." ensure correct position

[11] These system services are also referred to as system calls or syscalls, especially in the context of Linux.

[12] There are unofficial resources that attempt to map out the space of Windows syscalls, even across different versions, e.g. `https://j00ru.vexillium.org/syscalls/nt/64/`.

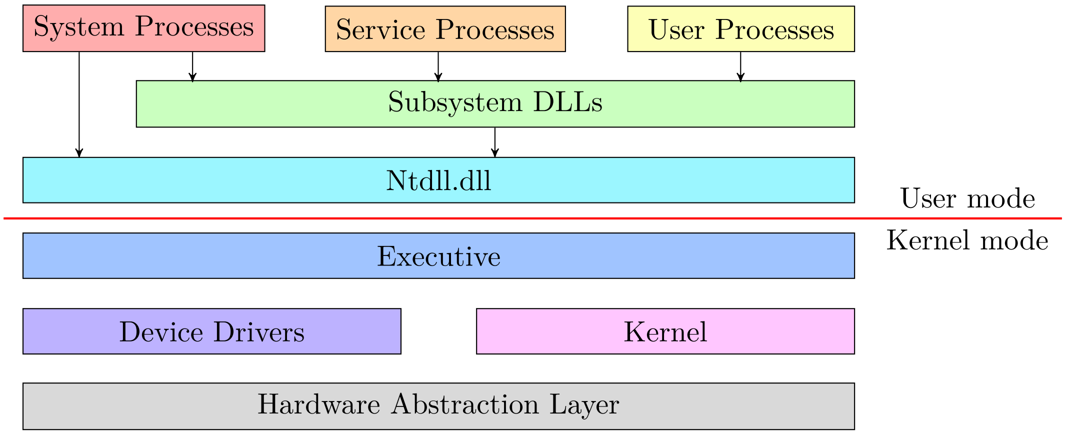

Figure 10: Simplified Windows architecture (modified after [9]). User processes are regular applications. The service process are responsible for hosting Windows services, such as the printer spooler or DNS client. System processes are fixed processes that are not started by the Service Control Manager. The Session Manager and the logon process are examples for system processes. The subsystem dynamically linked libraries (DLLs) are responsible for translating public API functions into their corresponding internal system service calls. The latter are mostly implemented in Ntdll.dll. The executive and kernel are explained in detail later on. Finally, the purpose of the Hardware Abstraction Layer (HAL) is to separate the kernel and device drivers from hardware-specific differences. It is intended to be used by device drivers and therefore largely documented in the WDK.

Furthermore, all user mode addresses can also be accessed from kernel mode. Pages can however be marked as read-only, a restriction which not even the kernel can circumvent [9].

The fact that there exists only one global kernel address space means that special care must be taken when writing code that runs in kernel mode. A single malfunctioning driver can destabilize the complete system or even introduce critical security vulnerabilities. To aid in discovering bugs in drivers, Windows provides a *Driver Verifier* tool [9].

It is also necessary to carefully choose which code is allowed to run in kernel mode. This is why Microsoft mandates that all third-party drivers for Windows 10 must be signed by one of two accepted certification authorities. They also have to be reviewed and signed by Microsoft. For testing purposes, Windows can be configured to load self-signed drivers. This is known as *test mode* [9].

One part of the OS that runs in kernel mode is the Windows executive, which was already mentioned in Figure 10. It is responsible for multiple things [9]:

- The implementations of the aforementioned system services reside in the executive.

- `DeviceIoControl`: not sure what the book means here, this just calls the NtDeviceIoControlFile syscall? look in VM with WinDbg what exactly happens here? screenshots? if screenshots mention it when explaining syscalls above, may be a nice example

- The executive also provides implementations of functions that can be used by other kernel mode components. Some of them are documented, others are not. They generally fall into one of four categories: managing Windows executive objects that represent OS resources; message passing between client and server processes on the same machine; run-time helpers for e.g. processing strings or converting datatypes; and general support routines for allocating and accessing memory as well as some synchronization mechanisms, e.g. fast mutexes.

– The executive can be categorized into different components, of which the following are most relevant when developing a device driver: the I/O manager, the Plug and Play (PnP) manager, and the Power manager. The functions exported by these components generally follow a naming scheme were their names start with `Io`, `Pp`, and `Po`, respectively. Please refer to table 2-5 in [9] for a more exhaustive list of commonly used prefixes of function names.

– Shareable resources, e.g. files, processes, and events, are represented as objects by the executive to user mode code. The state of the represented resource is captured in data fields called the object's attributes. Objects are *opaque*, i.e. their attributes can only be accessed through object methods. This makes the internal implementations of objects easily changeable[13]. The object methods may also implement certain security and access checks. A reference to a object is called a *handle*. Keeping track of these references allows the system to automatically recognize and deallocate objects that are no longer in use.

Another major kernel mode OS component is the Windows kernel, also already presented in Figure 10. Its tasks include the following [9]:

– It takes over the task of thread scheduling and (multi-processor) synchronization and also dispatches interrupts and exceptions. Aside from that, it avoids making policy decisions, and leaves this to the executive.

– In general, the one of the kernel's jobs is to conceal some of the differences of the different hardware architectures that Windows supports. It is assisted in this task by the HAL.

– Intended for usage by other, higher-level components, including the executive, it provides low-level routines and basic data structures. These are partly documented and may be used in device drivers. The function names all start with the `Ke` prefix.

– The exposed data structures are called *kernel objects*. These are simpler than the executive's objects and therefore have no overhead for e.g. manipulation via handles or security checks. They are the building blocks for most executive-level objects.

2.2.2 The Windows Driver Model for Kernel Drivers

Also explain how it gets loaded (if not done already) page 83

– IRPs and other IO stuff

– non-pageable paths and locking user memory, MDLs?

2.2.3 Communication Between Kernel and Userspace

Via ports

[13] One can see why these representations were named objects, as they follow some of the core concepts of object-oriented programming.

3 Related Work

3.1 Measuring Filesystem Driver Performance

3.2 Cryptographic Aspects of LUKS2

[3]

Search for more papers, e.g. attacks against LUKS? e.g. [12]

4 Other Approaches

4.1 Linux Kernel Implementation of LUKS2

https://web.archive.org/web/20210728082434/https://blog.cloudflare.com/speeding-up-li

4.2 VeraCrypt

4.3 BitLocker

[13] and [14] and [15] and [16]

Use Ghidra and look at some of the code of `fvevol.sys`?

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, mention failed attempts of making existing crypto libraries work in kern

make sure this chapter and section 2.1.3 are not too similar

LUKS2 supports many encryption algorithms (see [1], 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 First Experiments

6.2 Final Experimental Setup

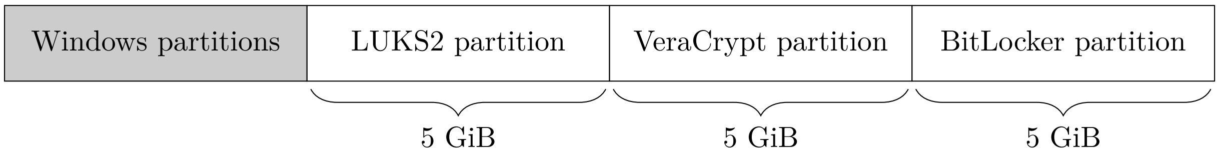

Figure 11: Disk layout of the real hardware SSD.

6.3 Results

For testing purposes, we tried optimizing the performance by restricting support to AES256-XTS. This enabled removing some if-else constructs that dispatched de-/encryption functions based on whether AES128 or AES256 was used. Even though these conditionals were located in a performance critical section, we saw no speed improvements. Our theory for why this was the case is the following: our driver was only ever used for one LUKS2 partition and therefore always took the same path through the if-else (either always AES128 or always AES256). This trained the CPU's branch prediction on this one specific path. Thus, after a short training phase, the CPU always speculatively executed the correct path, resulting in the same performance as without the if-else.

7 Discussion

8 Conclusion

List of Figures

List of Tables

References

[1] 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

[2] *Frequently Asked Questions Cryptsetup/LUKS*, 2020, visited on 2021-08-10. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/FrequentlyAskedQuestions?version_id=6297db166f8ae73c6a616ba874a12f5d43d37fd9

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

[4] 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

[5] *dm-crypt: Linux device-mapper crypto target*, 2020, visited on 2021-08-06. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt?version_id=ee336a2c1c7ce51d1b09c10472bc777fa1aa18cd

[6] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering – Design Principles and Practical Applications*. Wiley, 2010.

[7] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," U.S. Department of Commerce, Tech. Rep., 2001.

[8] *IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*, IEEE Std. 1619-2018 (Revision of IEEE Std. 1619-2007), 2019.

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

[10] M. E. Russinovich and A. Margosis, *Troubleshooting with the Windows Sysinternals Tools*. Microsoft Press, 2016.

[11] Microsoft, *Structure of the registry*, 2018, visited on 2021-08-14. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/sysinfo/structure-of-the-registry

[12] V. Polášek, "Argon2 security margin for disk encryption passwords," Master's thesis, Masaryk University, 2019.

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

[14] 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.

[15] 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.

[16] 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.