

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

Copyright © 2021 Max Ihlenfeldt

This document is licensed under the Creative Commons
Attribution-ShareAlike 4.0 International Public License (CC BY-SA 4.0).

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore e

Contents

1 Introduction	1
2 Background	2
2.1 LUKS2 Disk Encryption	2
2.1.1 On-Disk Format	2
2.1.2 Unlocking a Partition	4
2.1.3 Using an Unlocked Partition	7
2.2 Introduction to Windows Kernel Driver Development	8
2.2.1 Structure and Hierarchy of the Windows Operating System	8
2.2.2 Windows Kernel Drivers and the Windows Driver Model	11
2.2.3 Important Concepts for Kernel Driver Development	15
2.2.4 Debugging Kernel Drivers	21
3 Related Work	22
3.1 Measuring Filesystem Driver Performance	22
3.2 Cryptographic Aspects of LUKS2	22
4 Other Approaches	23
4.1 Linux Kernel Implementation of LUKS2	23
4.2 VeraCrypt	23
4.3 BitLocker	23
5 Design and implementation of our approach	24
5.1 Failed Attempts	24
5.2 The Final WDM Driver	24
5.2.1 Architecture	24
5.2.2 Initialization and Configuration	24
5.2.3 De-/encrypting Reads and Writes	24
5.2.4 Handling Other Request Types	25
5.3 Security Considerations	25
6 Performance of Our Driver	26
6.1 First Experiments	26
6.2 Final Experimental Setup	26
6.3 Results	26
7 Discussion	27
8 Conclusion	28
List of Figures	29
List of Tables	30
References	31

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 mandatory¹.

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 implementation² 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>

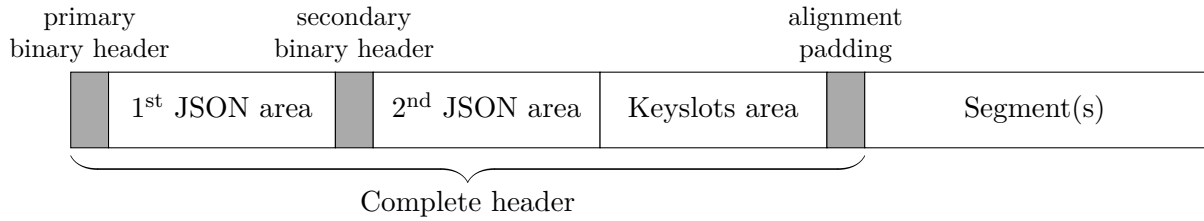


Figure 1: LUKS2 on-disk format (modified after [1]). The complete header consists of three areas: a binary header of exactly one 4096-byte sector, JSON metadata, and the binary keyslots data. A *keyslot* is an “encrypted area on disk that contains a key” [1]. For redundancy, the binary header and the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

```
#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	LUKS%.....@.
0010	00	00	00	00	00	00	00	03	54	68	69	73	20	69	73	20This 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	00sha256..
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	4Bë.ÔÆaÔ.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	3ëó.ÿÜM=ë0{7.0g]
0090	AD	A0	57	1C	0E	63	BC	D4	DD	3C	EC	F5	DE	67	F8	D8	..W...c%ÖŸ<iôPgøØ
00A0	F2	7E	82	CD	B9	DD	77	10	65	39	33	64	63	61	66	61	ô-,í'Ýw.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	'm@f.ÿûhNÃ"oLxq~
01D0	AE	1A	91	F8	E0	2C	F3	71	D5	17	CB	60	E5	2F	D6	36	@. 'øã,óqŮ.Ě 'ã/Ů6
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.

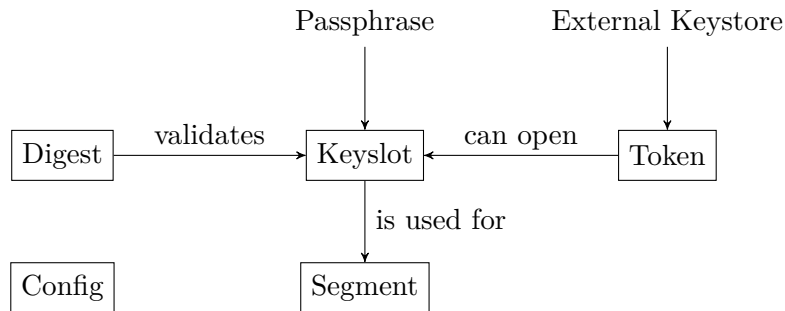


Figure 4: LUKS2 object schema from [1]. The most important objects are the following: *keyslots*, which describe the details of how cryptographic keys are stored and encrypted; *digests*, which can be used to verify that one has successfully extracted a key from a keyslot; and *segments*, which describe 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 LUKS2⁵.

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.

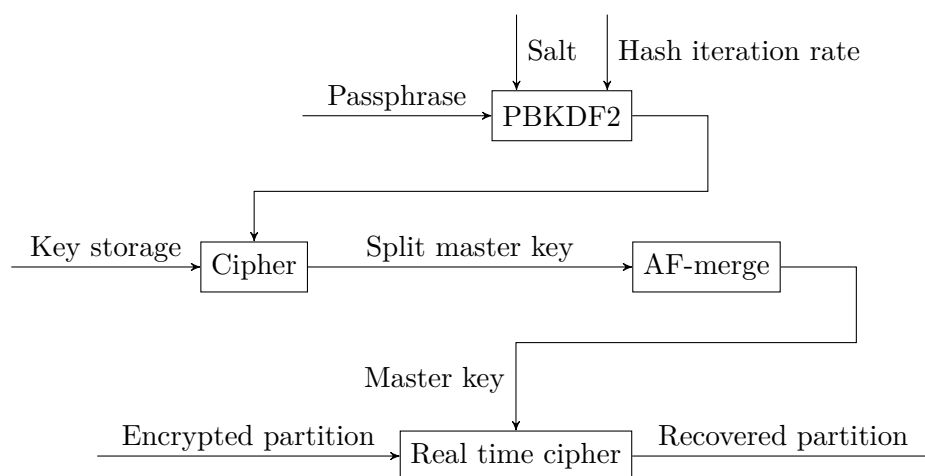


Figure 5: TKS2 scheme (modified after [3]).

⁵ It will also become apparent that they are quite aptly named.


```

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 also 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 Table 1 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 <code>dm-crypt</code> 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

Table 1: 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 encrypted data, called 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 simple⁶.

2.2 Introduction to Windows Kernel Driver Development

This section gives an introduction on the development of Windows kernel drivers and important related background information and 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 Win32 API [10] and the Windows Driver Kit (WDK) [11] 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 Sysinternals⁷ 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 [10]. In the context of registry keys, HKLM stands for HKEY_LOCAL_MACHINE.

Windows uses the physically available RAM to back its 64-bit address space of virtual memory. See Figure 8 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

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 The executables can be downloaded from <https://live.sysinternals.com> and are accompanied by documentation in [12].

is called *paging*. When a virtual address in a page that currently resides on disk is 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].

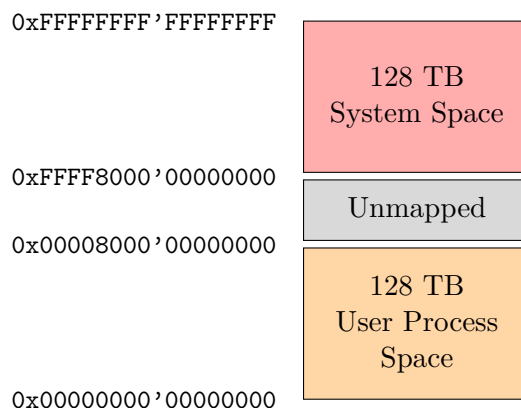


Figure 8: Address space layout (not to scale) for 64-bit Windows 10 (modified after [9]). Each process has its own copy of the user process space⁸.

Figure 9 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 execute⁹. 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) documented¹⁰ [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.

⁸ 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

⁹ These system services are also referred to as system calls or syscalls, especially in the context of Linux.

¹⁰ 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/>.

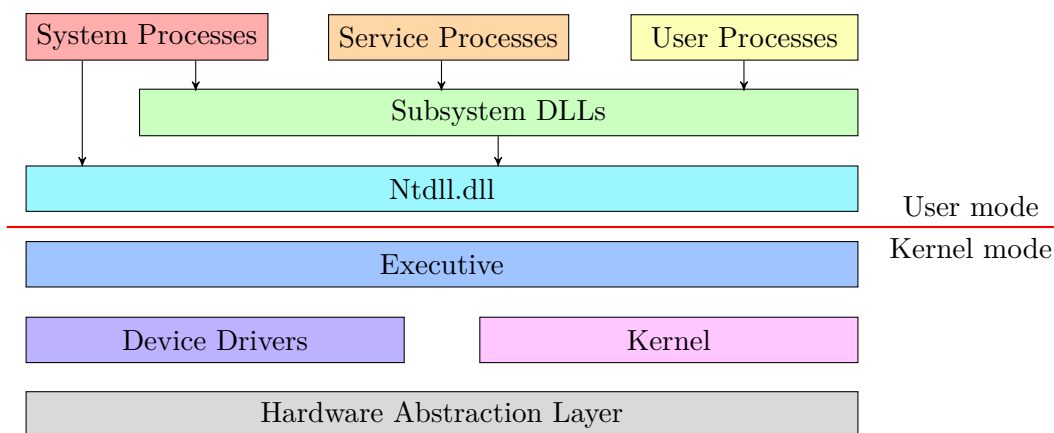


Figure 9: Simplified Windows architecture (modified after [9]). User processes are regular applications. The service process are responsible for hosting Windows services, such as the printer spooler or DNS client. System processes are fixed processes that are not started by the Service Control Manager. 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 9. 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 where 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 changeable¹¹. 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 9. 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, 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 Windows Kernel Drivers and the Windows Driver Model

Windows kernel drivers are modules that can be loaded and run kernel-mode. They are normally found in the form of files with the *.sys* extension. Drivers can be categorized as follows [9]:

- **Device drivers** As an interface between hardware and the I/O manager, these make use of the HAL to control retrieving input from or writing output to hardware.
- **File system drivers** These translate file-oriented requests into raw I/O requests.
- **File system filter drivers** These process incoming I/O requests or outgoing responses before passing them to the next driver. They can also reject an operation. The driver that this thesis is about falls into this category.

¹¹ One can see why these representations were named objects, as they follow some of the core concepts of object-oriented programming.

- **Network drivers** These either implement networking protocols, like TCP/IP, or handle the transport of file system I/O between machines on a network.
- **Streaming filter drivers** These are for data stream signal processing.
- **Software drivers** These are helper kernel modules for user-mode programs. They are used when a desired operation, e.g. retrieving internal system information, is only available in kernel mode.

Drivers can be written using different models or frameworks. Windows NT 3.1 introduced the first driver model, which lacked PnP functionality. Windows 2000 introduced PnP support as well as an extended driver model known as the *Windows Driver Model (WDM)* [9]. We will focus on WDM in this section, because that is the framework our driver uses (see section 5.2). But there also exist other models and frameworks for Windows driver development, such as the Windows Driver Foundation (WDF). It tries to simplify things and provides the Kernel-Mode Driver Framework (KMDF) and the User-Mode Driver Framework (UMDF) [9]. We will explore some of the differences between them and WDM in section 5.1.

WDM distinguishes between three types of drivers [9]:

- **Bus drivers** These are responsible for managing devices that have child devices, such as bus controllers, bridges, or adapters. They are generally provided by Microsoft because they are required for widely used buses like PCI and USB. It is nevertheless also possible for third parties to implement support for new bus types by supplying a bus driver.
- **Function drivers** “A function driver is the main device driver and provides the operational interface for its device.” [9] Every device needs a function driver, if it is not used raw. This is a mode of operation where the bus driver manages I/O together with bus filter drivers.
- **Filter drivers** These enhance an already existing driver or device by either adding functionality or modifying I/O requests and/or responses from other drivers. They are optional and the number of filter drivers for a device is not limited.

In this model, the control over a device is not concentrated at one single driver, but rather shared between multiple drivers. A bus driver only informs the PnP manager about the devices connected to its bus, while the device’s function driver handles the actual device manipulation and control. The drivers responsible for a device form a hierarchy called the *device stack*. This defines the order in which the drivers process incoming and outgoing messages to or from the device: the uppermost driver receives incoming requests first and the lowermost one receives them last. For outgoing driver responses, this order is reversed [9]. See Figure 10 for an example of how such a device stack can look.

The device stack is built by the PnP manager during a process called *device enumeration*. This happens at system boot or when resuming from hibernation, but it can also be triggered manually. During enumeration, the PnP manager constructs a so-called *device tree* that captures the parent-child relationships of devices. A node in this tree represents a physical device and is appropriately called a *device node*, or short *devnode*. When a new device is discovered, its required drivers are loaded (if they have not already been

¹² Later on in this section we will talk about the registry’s role in driver management in more detail. See Figure 12 for what key to use and how the values stored under that key may look. ensure correct position

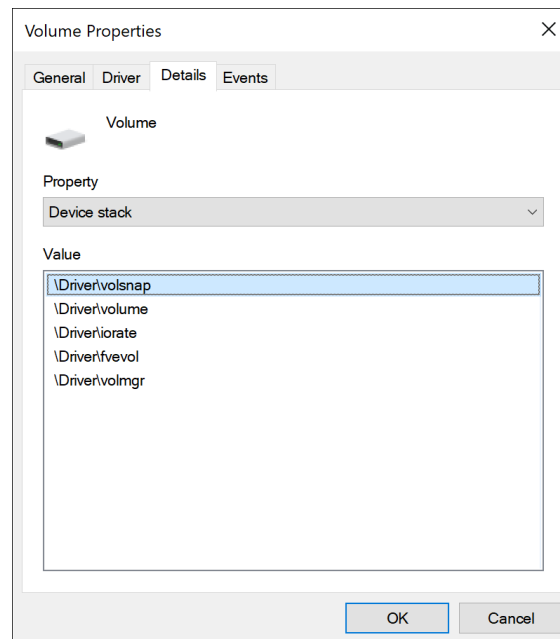


Figure 10: Example of a device stack for a storage volume (Screenshot from device properties in Windows Device Manager). The official descriptions of the drivers listed in this example are, from top to bottom: “Volume Shadow Copy driver”, “Volume driver”, “I/O rate control filter”, “BitLocker Drive Encryption Driver”, and “Volume Manager Driver”. As can be seen under the appropriate registry key¹², `fvevol` and `iorate` are lower filter drivers and `volsnap` is an upper filter driver. This leaves `volume` as the function driver and `volmgr` as the bus driver.

loaded). The drivers are then informed about the new device, and can decide whether they want to be part of the device stack or not [9].

For all the drivers of a device, the PnP manager creates objects that are used for managing and representing the relationships between the drivers. These objects form the device stack. They are also a form of device object, however they don’t represent a real, physical device, but rather a virtual one. See Figure 11 for an example device node and the objects it is made of. There are the following types of objects in a device node, ordered from bottom to top by their occurrence in a devnode [9]:

- exactly one *physical device object (PDO)*, created by the bus driver. It is responsible for the physical device interface.
- zero or more *filter device objects (FiDOs)*, created by lower filter drivers (lower being relative to the FDO).
- exactly one *functional device object (FDO)*, created by the device’s function driver. It is responsible for the logical device interface.
- zero or more FiDOs, created by upper filter drivers.

This shows that filter drivers can be placed either above the function driver (upper filter drivers) or between the function driver and the bus driver (lower filter drivers). These serve different purposes: “In most cases, lower-level filter drivers modify the behavior of device hardware. For example, if a device reports to its bus driver that it requires 4

I/O ports when it actually requires 16 I/O ports, a lower-level, device-specific function filter driver could intercept the list of hardware resources reported by the bus driver to the PnP manager and update the count of I/O ports. Upper-level filter drivers usually provide added-value features for a device. For example, an upper-level device filter driver for a disk can enforce additional security checks.” [9]

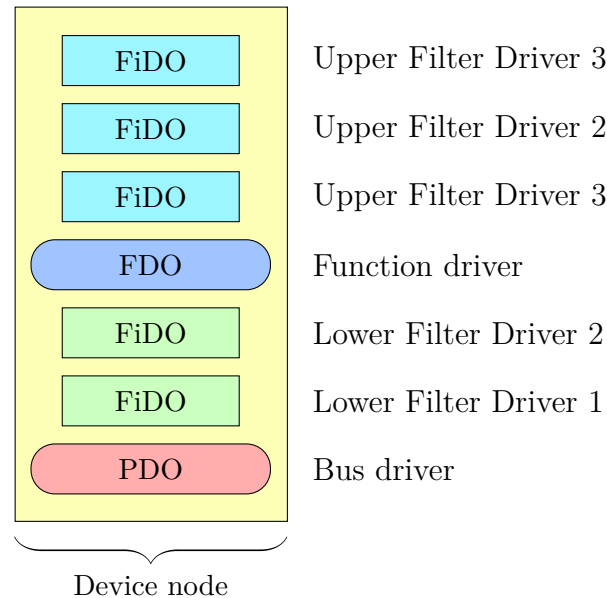


Figure 11: A device node and its device stack (modified after [9]). See Figure 10 for a concrete example of a device stack.

As already hinted at in the description of Figure 10, the registry is responsible for storing the information that governs driver loading. Both which drivers are loaded and the order they are loaded in are configured this way. Basically, it is described how exactly the device nodes should be built. Table 2 lists the registry keys that, together with their subkeys, are relevant for the configuration.

Registry Key	Short Name	Description
HKLM\System\CCS\Enum	Hardware key	Settings for known hardware devices
HKLM\System\CCS\Control\Class	Class key	Settings for device types
HKLM\System\CCS\Services	Software key	Settings for drivers

Table 2: Important registry keys for driver loading [9]. Note that **CCS** stands for **CurrentControlSet**.

The keys from Table 2 each have different roles [9]:

- The Hardware key configures driver loading for individual hardware devices. Its subkeys all follow the pattern of `<Enumerator>\<Device ID>\<Instance ID>`. The enumerator is the name of the responsible bus driver; the device ID identifies a particular type of hardware, e.g. a specific model from one manufacturer; and the instance ID distinguishes different devices with the same device ID, e.g. the device’s location on the bus or its serial number. The most important values for each device are: **Service**, which contains the name of the device’s function driver (this is used as a subkey in the Software key); **LowerFilters** and **UpperFilters**, which each

contain an ordered list of lower and upper filter driver names, respectively; and `ClassGUID`, which identifies the device's class (this is used as a subkey in the Class key).

- The Class key contains class GUIDs¹³ as subkeys. The configuration stored here is similar to the subkeys of the Hardware key, but applies to a whole class of devices. For example, the lower and upper filters listed under the GUID for the “Volume” class are loaded for all volumes.
- The Software key contains relevant information about drivers, most importantly the `ImagePath` value, which stores the path to the driver's `.sys` file. The subkey that contains the values for a driver is the driver's name.

Examples of subkeys of each of these keys are shown in Figure 12.

2.2.3 Important Concepts for Kernel Driver Development

The goal of this section is to explain some concepts that are relevant when developing a file system filter driver. They concern memory access, thread contexts, the kernel's I/O system, and communication between kernel and user mode. **may need updating in the future**

The code of a driver can run in different thread contexts, depending on how it was called. This is mostly relevant for accessing memory, because the translation of virtual memory addresses into physical addresses depends on the thread context. We will discuss what exactly to keep in mind when accessing memory shortly. The three different contexts that driver code can run in are the following [9]:

- when called to handle an I/O request initiated by a user thread, the driver code runs in this thread's context;
- when called by a OS component, e.g. the PnP manager, the driver code runs in the context of the calling system thread;
- when called via an interrupt, it is not possible to predict in which context the driver code will run. The context of whatever thread was currently executing when the interrupt arrived will be used. The interrupt may stem from hardware, for example when a device informs about available data, or from software, for example when a timer fires.

Note that, regardless of the context, the code always executes in kernel mode. When running in the context of a user mode thread, the CPU switches to kernel mode before running the driver code [9].

Another important parameter of code execution is the so-called *interrupt request level* (*IRQL*). As the name suggests, this has something to do with interrupts, but for this thesis it suffices to talk about one important property of this numerical value: code running at a lower IRQL cannot interrupt code running at a higher IRQL. We will discuss the implications of this in a moment. The normal IRQL of the CPU is 0, also called `PASSIVE_LEVEL`. This is also the only possible value when in user mode. Another possible

¹³ Globally Unique Identifiers, also known as Universally Unique Identifiers (UUIDs). See RFC 4122 for more information.

¹⁴ Which makes sense, because it is responsible for the ReadyBoost technology, which is designed for SD cards and USB sticks, and not hard disk volumes. See [9] for more information on this technology. **ensure correct position**

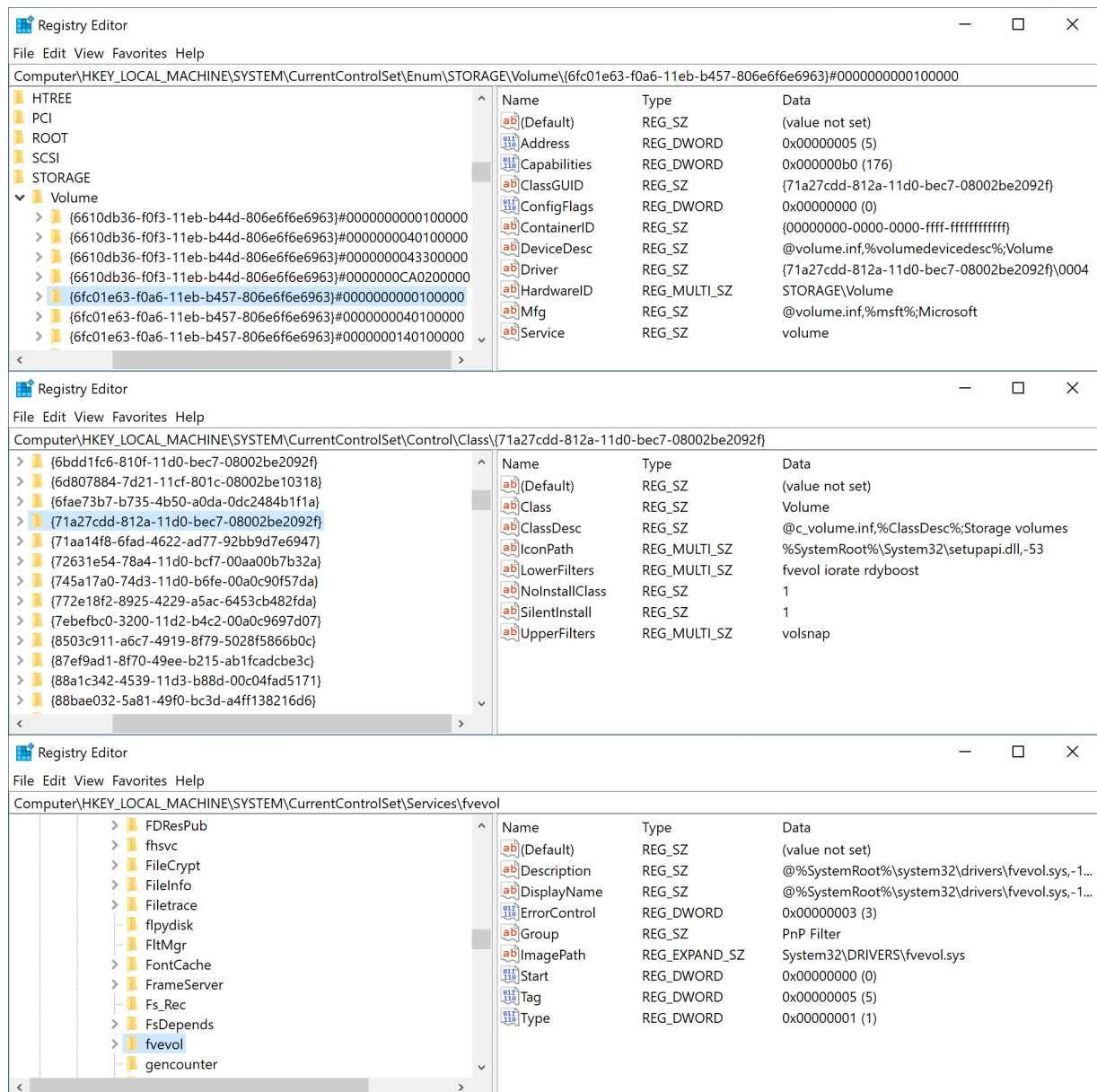


Figure 12: Example contents of a Hardware, Class, and Software subkey (Screenshots from the Windows Registry Editor). These are all relevant for the device whose device stack is shown in Figure 10. As already partly discussed there, the configured values match the final device stack: the **Service** value under the Hardware subkey designates **volume** as the function driver; the drivers listed in the **LowerFilters** and **UpperFilters** values of the Class subkey are indeed loaded before or after **volume**, respectively (also note that they are loaded, from bottom to top, in the order listed; the **rdyboost** driver does not appear in the device stack because it probably decided not to be part of it¹⁴); the **ImagePath** under the Software subkey shows the path to one of the driver's **.sys** file.

value for kernel mode code, more or less the only other one non-device drivers (such as filter drivers) will come in contact with, is an IRQL of 2. This is known as **DISPATCH_LEVEL**. We will present a concrete example of code that can run at **DISPATCH_LEVEL** when talking about the I/O system. The important thing is that the kernel's thread scheduler runs

at this IRQL. Because of the property mentioned earlier, this means that the scheduler cannot interrupt code running at `DISPATCH_LEVEL`. The immediate consequences of this are [9]:

- Synchronization objects that rely on the scheduler’s support, such as semaphores or mutexes, no longer work. The problem is that when waiting on one of these objects, the thread goes to sleep. But the scheduler can’t run, because that would mean interrupting the thread, and thus it can never wake up the thread again. Windows prevents this by checking the IRQL and crashing if the current IRQL has a value of 2.
- Page faults must not occur. Handling them requires a context switch, but for that the scheduler would need to work. Therefore, code at `DISPATCH_LEVEL` can only access memory that is not currently paged to disk. This can be ensured by allocating the memory in a special pool which, by definition, always is resident in memory: the *non-paged pool*¹⁵.

In general, when running at an IRQL of 2, every called function needs to be checked for the ability to be called at this level. The supported IRQL is documented for all WDK functions.

As mentioned before, access to memory in userspace generally requires special care. To be precise, the following problems can occur [9]:

- Recall that the translation of user mode virtual addresses to physical addresses depends on the thread context. Referencing an address from a different context than it belongs to either leads to an access violation or accesses random data from another process.
- Userspace memory always has the risk of being paged out. Code running at an IRQL of 2 must ensure that this is not the case before accessing memory. As discussed before, page faults are illegal in this state.

One situation where a driver needs to access user memory is doing I/O, i.e. read or write operations. Because these are so common, the I/O manager presents a solution, namely the *Buffered I/O* and the *Direct I/O* modes [9]:

- **Buffered I/O** When using this mode, the I/O manager copies the data from the user buffer to a newly allocated kernel buffer from the non-paged pool (or vice versa, depending on whether it is a read or write operation). This means the driver does not have to care about accessing the user buffer at all and can just work with the kernel buffer. Because it was allocated from non-paged pool, no page fault can occur, and as a system space address, the kernel buffer address is valid in any thread context. See Figure 13 for how this works when reading data. Buffered I/O is commonly used for small buffers (up to one page, or 4KB).
- **Direct I/O** In this mode, the I/O manager locks the user buffer in RAM, so that it cannot be paged out to disk. The driver can then create a mapping of the buffer in kernel memory and just use that, without any copying. To do so, the I/O manager provides the driver with a MDL, which stands for *memory descriptor list*. It holds information about the physical memory occupied by the user buffer,

¹⁵ There also exists an allocation pool without this special guarantee, called the *paged pool*.

and it is needed to map the buffer into system space. At this point, the buffer's physical memory is mapped twice: once into user memory, and once into kernel memory. Both mappings can be accessed at any IRQL, because the buffer cannot be paged out. However, while the user mapping is only valid in one thread context, the kernel mapping is always valid. After the operation has completed, the I/O manager removes the kernel mapping and unlocks the buffer. Figure 14 illustrates this process. Direct I/O is useful when using large buffers (larger than one page, or 4KB), because no copying is needed¹⁶.

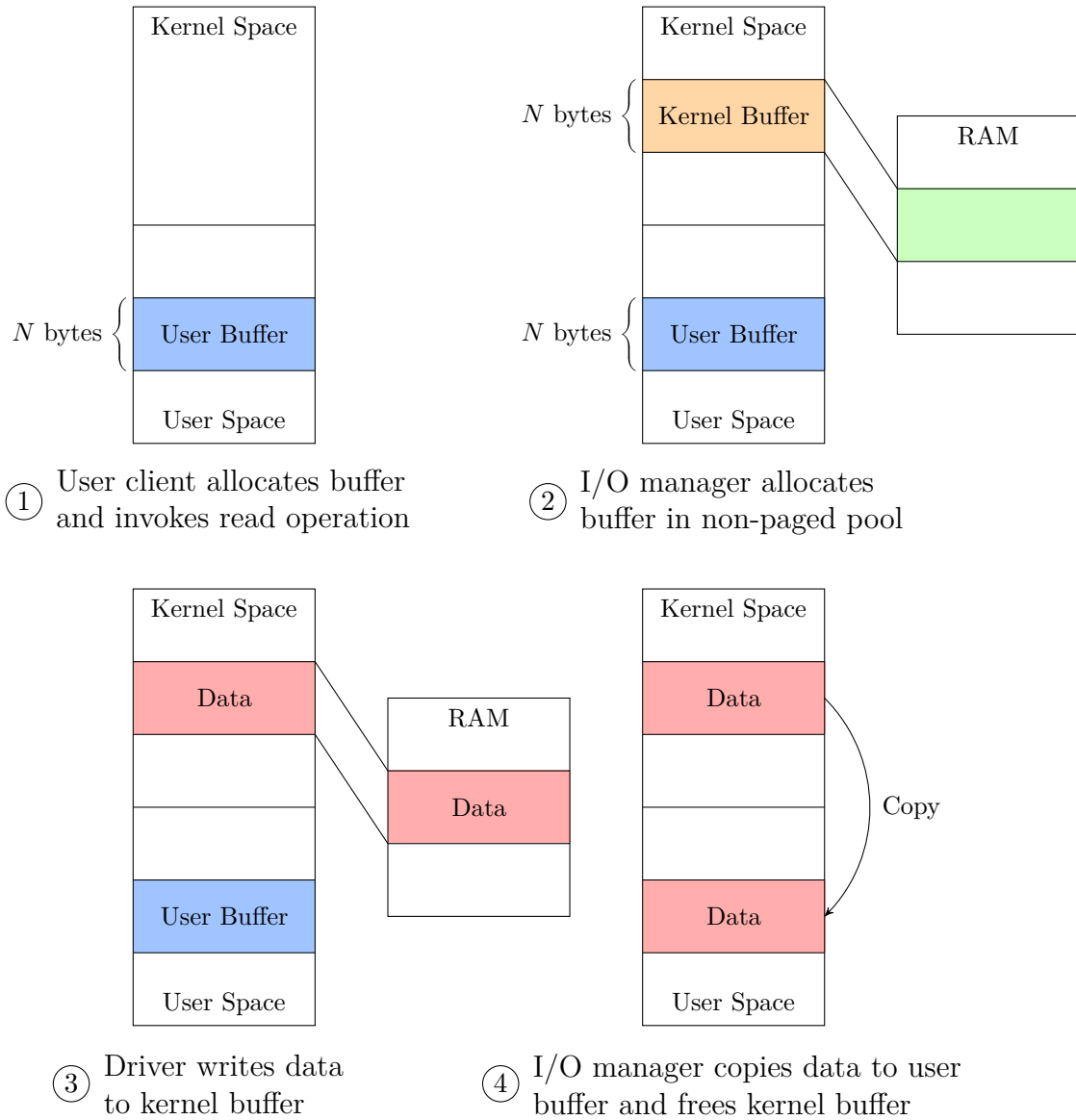


Figure 13: Buffered I/O for a read operation (modified after [9]). For write operations, the copying is done before the driver is called to handle the request.

It is up to the driver to decide which I/O mode to use. Their choice is signalled to the I/O manager using a flag of their virtual device's object (the one that is part of the device node). This applies to all I/O operations, the exception being so-called *device I/O*

¹⁶ It is also more suitable than buffered I/O for devices with direct memory access (DMA), “because DMA is used to transfer data from a device to RAM or vice versa without CPU intervention – but with buffered I/O, there is always copying done with the CPU, which makes DMA pointless.” [9]

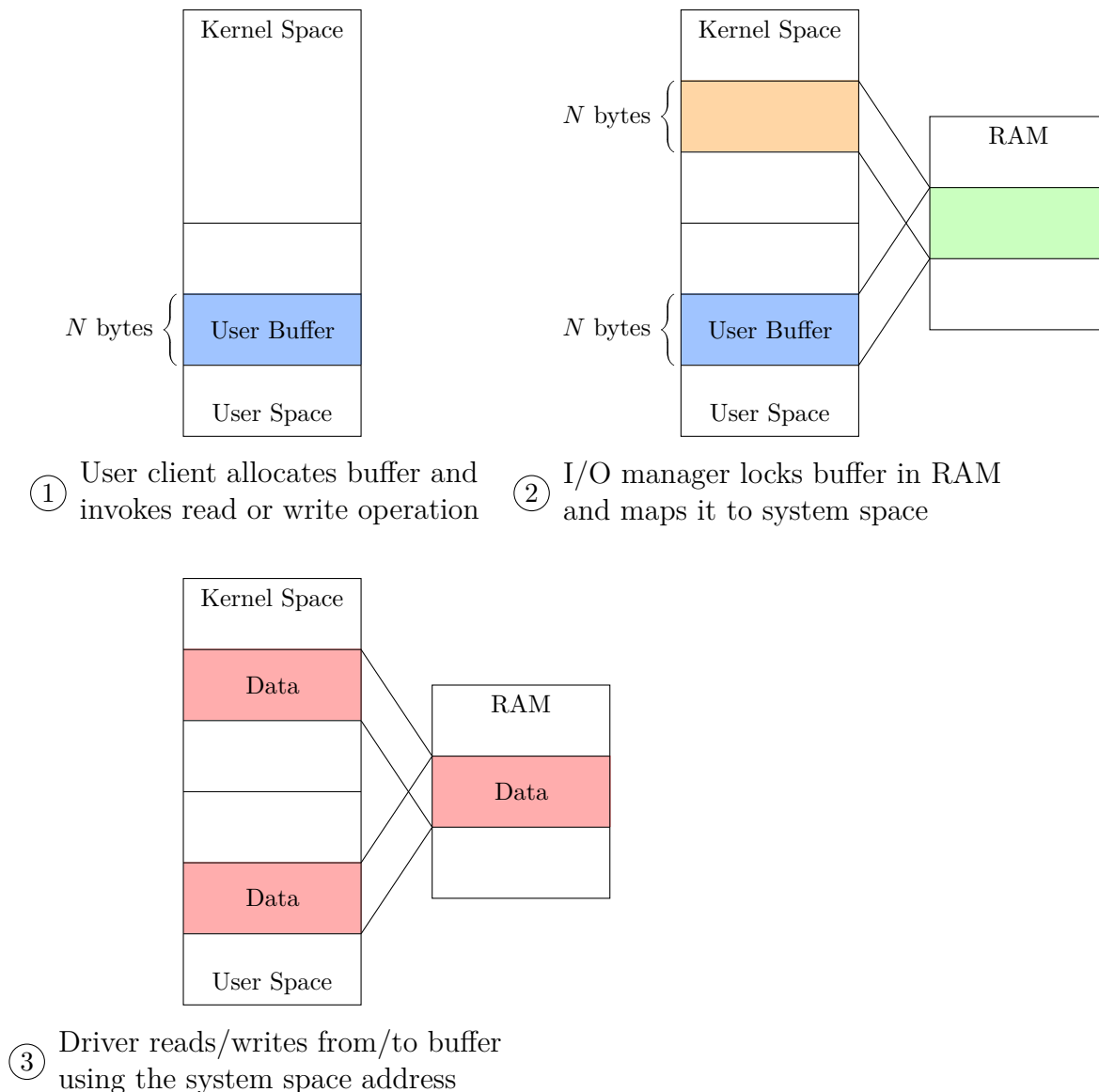


Figure 14: Direct I/O for a read or write operation (modified after [9]).

control requests. These are the Windows equivalent to Unix' `ioctl` syscall and are used to communicate with a device from user mode¹⁷. A device control code, specifying what exactly the user wants from the device, also defines the used I/O mode, and drivers must adhere to this [9].

In situations where it is not too complicated, drivers can choose to handle the user buffer access completely by themselves. This mode is known as *Neither I/O*. It has the advantage of zero overhead, because the I/O manager does no extra work. It does however increase the risk of bugs and possible security risks. See [9] for more information.

I/O operations have been mentioned a lot in this section. In the following paragraphs, the important details of the kernel's I/O system will be laid out.

Firstly, all I/O has a *virtual file* as its target. This may be backed by a "real" file or by something different, like a device¹⁸. This abstraction allows the I/O manager to only

¹⁷ An example is `IOCTL_DISK_GET_DRIVE_GEOMETRY`, which is used to get the physical layout of a disk. See www.ioctls.net for a comprehensive (unofficial) list of almost all relevant control codes and the documentation for `DeviceIoControl` for information on how to use them.

¹⁸ This concept will be familiar to Linux users.

care about files, leaving the translation of file commands to device-specific operations to drivers [9].

Secondly, Windows' internal I/O system is packet-driven. Almost all I/O requests¹⁹ travel through the different relevant parts of the system in the form of an *I/O request packet (IRP)*, which contains all relevant information about the request. IRPs are allocated in non-paged pool, usually by the I/O manager²⁰, but it is also possible for drivers to initiate I/O by creating an IRP. After an IRP's allocation and initialization, a pointer to it is passed to the driver that shall handle the request. The driver then performs its operation and returns the IRP to the I/O manager. It also reports whether it was able to complete the requested operation or whether further work by another driver needs to be done [9].

An IRP holds a lot of information. For this thesis, it suffices to know about the following fields²¹ [9]:

- a pointer to a MDL, if the driver uses the direct I/O method;
- a pointer to a system buffer, if the driver uses the buffered I/O method;
- information about an IRP's final status, if it has been completed;
- information about the IRP's stack locations, which will be explained momentarily.

In memory, the IRP is followed by one or more *stack locations*, their count matching the number of drivers this IRP will pass through²². These stack locations are represented by `IO_STACK_LOCATION` structures, whose content is shown and explained in Figure 15. Because the information about the request is split between the IRP and its stack location, drivers can modify the stack location parameters for the next driver, while keeping the current set of parameters. The original parameters may for example be useful in a completion routine [9].

The I/O manager only initializes the first stack location before sending the IRP over to the driver at the top of the device stack. Each driver, except the lowest in the stack, is then responsible for initializing the stack location for the next driver. If a driver can complete an IRP and no other drivers need to be called, initializing the next stack location is of course not necessary. This initialization can be done in two ways [9]:

- if no changes to the stack location are needed, the driver can “skip” the current stack location. This means that the next driver will receive the same parameters in its stack location as the current driver.
- if changes to the stack location need to be made, the driver can copy its stack location and modify the copy. Registering a completion routine (explained momentarily) counts as a change, and a driver must copy the stack location instead of skipping it in that case [11].

¹⁹ The exception is so-called *Fast I/O*, which works differently [9]. It is not supported by our driver and will therefore not be covered.

²⁰ The PnP or power manager also sometimes allocate IRPs [9]. For simplicity, we will write “the I/O manager” instead of “the I/O, PnP, or power manager” in the following paragraphs.

²¹ See [9] and its documentation in the WDK for more details.

²² This value is known when allocating the IRP, because it is equal to the size of the device stack. In some scenarios, most of which involve the filter manager, an IRP might be sent over to a new device stack. This might necessitate readjusting the count of stack locations, if the new device stack has a different size than the one before [9]. We will discuss the filter manager in section 5.1.

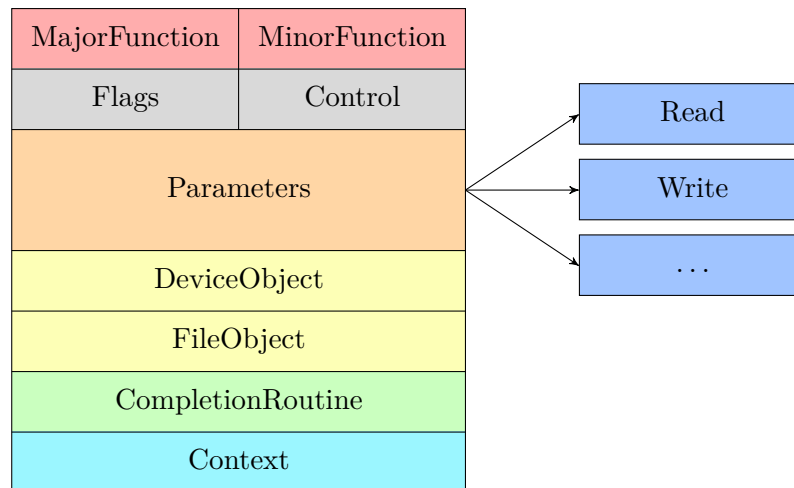


Figure 15: Layout of the `IO_STACK_LOCATION` structure (modified after [9] using information from [11]). The `MajorFunction` field has one of 28 possible values, denoting the request type, e.g. `IRP_MJ_READ` or `IRP_MJ_WRITE`. `MinorFunction` is used to further specify the type for some of the major functions. We will not discuss `Flags` and `Control`, because they are only rarely relevant. The `Parameters` are “a monstrous union of structures, each of which valid for a particular major function code or a combination of major/minor codes.” They contain information further specifying the request, e.g. the byte offset when reading from a file. `DeviceObject` and `FileObject` are pointers to the related device and file objects. `CompletionRoutine` holds a pointer to a completion routine, which will be explained separately. `Context` is a driver-defined parameter passed to the completion routine.

The already mentioned completion routines are a way for drivers to intercept the flow of an IRP after its completion by another driver. Completion routines are called in the opposite order that they were registered in, i.e. those registered by lower drivers first. See Figure 16 for an example. In such a routine, a driver can check the status of the completed IRP and do necessary post-processing. It is even possible to mark the IRP as uncompleted again and resend it down the stack [9]. Completion routines may run at an IRQL equal to `DISPATCH_LEVEL` [11].

2.2.4 Debugging Kernel Drivers

WinDbg (page 38), screenshots? don't take away content from 5.2.1

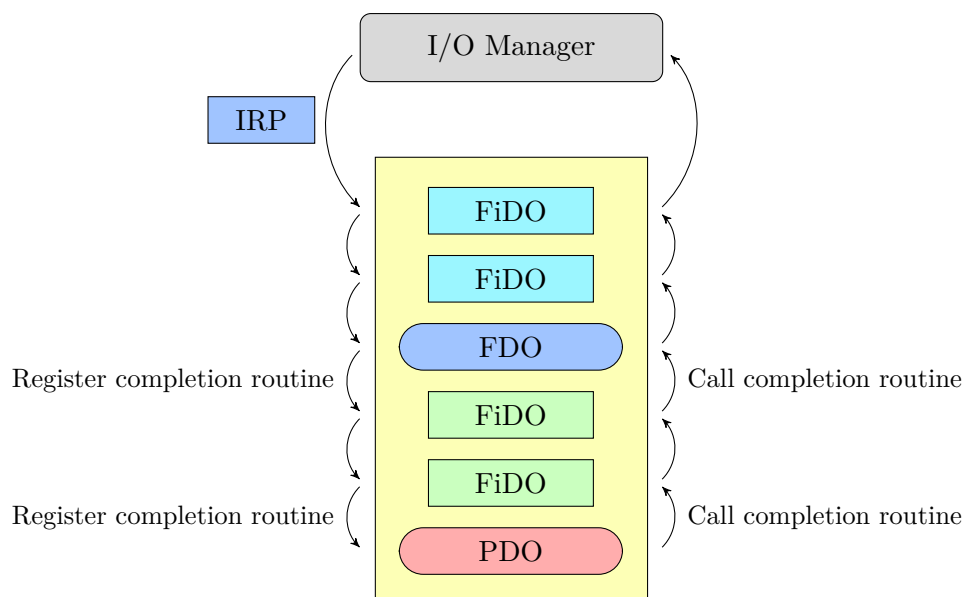


Figure 16: Example IRP flow (modified after [9]). The I/O manager sends an IRP down the device stack of a devnode. It first passes through two upper filter drivers and the function driver, with the latter registering a completion routine. It then passes through two lower filter drivers, with the second of those also registering a completion routine. After the bus driver completes the IRP, it travels back up the device stack. Because only the second lower filter driver and the function driver registered a completion routine, they are the only drivers that the IRP passes through on its way in this direction. The real path of the IRP is slightly more complex than shown in this figure, because normally the I/O manager takes care of routing the IRP from one driver to another [9]. It is also responsible for calling the completion routines [11].

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. [13]

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

[14] and [15] and [16] and [17]

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 fsrec driver and debugging

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

here and/or in other sections: screenshots from WinDbg showing e.g. a device stack and luks2flt's device object

Also screenshots from the registry showing the modified volume class' lower filters?
explain "by the way"? or move to section 2.2.3

- INF files (page 485) (mention <https://docs.microsoft.com/en-us/windows-hardware/drivers/device>)
- look-aside lists (page 331)
- communication between kernel and userspace via ports

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 kernel
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

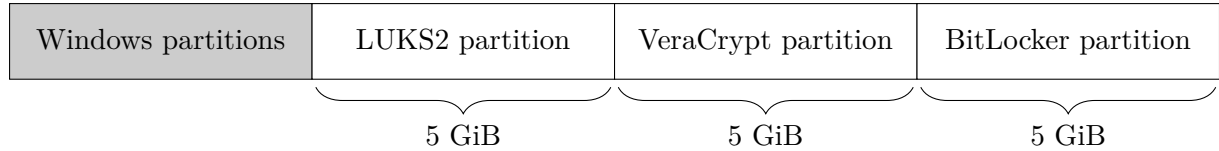


Figure 17: Disk layout of the real hardware SSD.

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.

The default compiler optimization settings in Visual Studio were a little bit conservative and also optimized for smaller code size than more speed. After tuning these settings to enable more aggressive optimizations and also focus on speed rather than code size, we found that this is a little bit complicated.....

7 Discussion

8 Conclusion

List of Figures

1	LUKS2 on-disk format	3
2	LUKS2 binary header structure	3
3	LUKS2 binary header example	4
4	LUKS2 object schema	4
5	TKS2 scheme	5
6	LUKS2 master key decryption in Rust	6
7	LUKS2 master key decryption with multiple available keyslots	7
8	Address space layout for 64-bit Windows 10.	9
9	Simplified Windows architecture	10
10	Example of a device stack for a storage volume	13
11	A device node and its device stack	14
12	Example contents of a Hardware, Class, and Software subkey	16
13	Buffered I/O for a read operation	18
14	Direct I/O for a read or write operation	19
15	Layout of the <code>IO_STACK_LOCATION</code> structure	21
16	Example IRP flow	22
17	Disk layout of the real hardware SSD	26

List of Tables

1	Selection of LUKS2 encryption algorithms	7
2	Important registry keys for driver loading	14

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. Fruhwirth, *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] *Programming reference for the Win32 API*, visited on 2021-08-19. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api>
- [11] *API reference docs for Windows Driver Kit (WDK)*, visited on 2021-08-19. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi>
- [12] M. E. Russinovich and A. Margosis, *Troubleshooting with the Windows Sysinternals Tools*. Microsoft Press, 2016.
- [13] V. Polášek, “Argon2 security margin for disk encryption passwords,” Master’s thesis, Masaryk University, 2019.
- [14] J. D. Kornblum, “Implementing BitLocker drive encryption for forensic analysis,” *Digital Investigation*, vol. 5, no. 3-4, pp. 75–84, 2009.
- [15] 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.

-
- [16] S. Törpe, 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.
 - [17] 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.