2019 IEEE Symposium on Security and Privacy

Self-encrypting deception: weaknesses in the encryption of solid state drives

Carlo Meijer
Institute for Computing and Information Sciences
Radboud University Nijmegen
cmeijer@cs.ru.nl

Bernard van Gastel
School of Computer Science
Open University of the Netherlands
and
Institute for Computing and Information Sciences
Radboud University Nijmegen
Bernard.vanGastel@{ou.nl,ru.nl}

***Abstract*—We have analyzed the hardware full-disk encryption of several solid state drives (SSDs) by reverse engineering their firmware. These drives were produced by three manufacturers between 2014 and 2018, and are both internal models using the SATA and NVMe interfaces (in a M.2 or 2.5" traditional form factor) and external models using the USB interface.**

In theory, the security guarantees offered by hardware encryption are similar to or better than software implementations. In reality, we found that many models using hardware encryption have critical security weaknesses due to specification, design, and implementation issues. For many models, these security weaknesses allow for complete recovery of the data without knowledge of any secret (such as the password).

BitLocker, the encryption software built into Microsoft Windows will rely exclusively on hardware full-disk encryption if the SSD advertises support for it. Thus, for these drives, data protected by BitLocker is also compromised.

We conclude that, given the state of affairs affecting roughly 60% of the market, currently one should not rely solely on hardware encryption offered by SSDs and users should take additional measures to protect their data.

I. Introduction

In recent years, the protection of sensitive data has received increased attention. Protection of digital data has become a necessity, certainly in the light of the new European Data Protection Regulation. Technically, encryption is the go-to protection mechanism; it may be implemented in software or hardware (or both). It can be applied on the level of files, or the entire drive, which is called *full-disk encryption*. Full-disk encryption is often the solution of choice as it takes away concerns of sensitive data leakage through, e.g. temporary files, page files, and caches. Several software solutions for full-disk encryption exist, and modern operating systems typically integrate it. Purely software-based encryption has inherent weaknesses, such as the encryption key being present in RAM at all times and performance drawbacks.

In an attempt to address these weaknesses, hardware full-disk encryption is often proposed; the encryption is performed within the drive itself, thereby confining the encryption key exclusively to the drive. Typically, the encryption itself is performed by a dedicated AES co-processor, whereas the software on the drive (firmware) takes care of the key management. It is often regarded as the successor of software full-disk encryption. Full-disk encryption software, especially those integrated in modern operating systems, may decide to rely solely on hardware encryption in case it detects support by the storage device. In case the decision is made to rely on hardware encryption, typically software encryption is disabled. As a primary example, BitLocker, the full-disk encryption software built into Microsoft Windows, switches off software encryption and completely relies on hardware encryption *by default* if the drive advertises support.

Contribution. This paper evaluates both internal and external storage devices, from three vendors, adhering to standards for secure storage. The vendors combined produce roughly 60% of the SSDs sold between 2014 and 2018 [1]. An overview is given of possible flaws that apply in particular to hardware-based full-disk encryption (Section IV), and a methodology is provided for the analysis (Section V). We have analyzed firmware from different SSD models offering hardware encryption, focusing on these flaws (see Section VI and Table I). The analysis uncovers a pattern of critical issues across vendors. For multiple models, it is possible to bypass the encryption entirely, allowing for a complete recovery of the data without any knowledge of passwords or keys. The situation is worsened by the delegation of encryption to the drive by BitLocker. Due to the default policy, many BitLocker users are unintentionally using hardware encryption, exposing them to the same threats. We should reconsider how we view hardware encryption: as a layered defense, or exclusively in charge of protecting data (without active software encryption).

Related work. In 2013 the possibility of debugging a hard drive through JTAG (hardware debug port which can control the processor and memory) was demonstrated and created possibly the first public hard drive firmware rootkit [2]. Domburg's work has inspired more research around anti-forensics such as [3], [4]. Background on reverse engineering embedded devices such as SSDs and PLCs can be found in [5]. Leaked documents indicate that even the NSA is using these techniques [6]. Besides, proprietary cryptographic systems have often shown to be much weaker in practice than standardized publicly available alternatives once implementation details are uncovered [7]. Within the scope of storage devices with integrated hardware encryption, serious vulnerabilities

DOI 10.1109/SP.2019.00088

have also previously been identified in external drives using proprietary protection schemes. An example is the external Secustick, which unlocks by simply sending a command (not containing a password) [8]. Another example is the Western Digital MyPassport family of external drives, which suffers from RAM leakage, weak key attacks, or even hardcoded keys [9]. However these findings are isolated incidents limited to proprietary solutions, and neither consider implementations of established standards for secure storage nor consider these issues across multiple vendors. We focus on an offline attack where an attacker has physical control of a switched off drive. Online attacks against SSDs are shown to be possible [10].

Responsible disclosure. After discovering these vulnerabilities, we followed a process of responsible disclosure. In this case, the National Cyber Security Center (NCSC) of the Netherlands was informed first, which assisted in the responsible disclosure process, by setting up the contact with the manufacturers involved. We cooperated with Microsoft, Crucial, Samsung, and Western Digital/Sandisk to fix their products and agreed not to disclose the vulnerabilities for up to six months. The parties involved were notified at the beginning of 2018 (and released to the public on November 5, 2018), except for Western Digital/Sandisk which was notified in December of 2018 due to new findings. These vendors have confirmed the reported issues. For models being supported, firmware updates are either released or in development.

II. Background

A. Software vs Hardware Encryption

To avoid negatively impacting the data throughput when encryption is switched on, SSDs with encryption support or *self-encrypting drives (SEDs)* always house a dedicated AES co-processor that provides for the encryption. Therefore, data encryption is essentially ‘free’ in terms of computational resources. These drives encrypt all data stored on them with the *disk encryption key* (DEK), even in the case when the data is not password-protected. All drives considered in this paper use this approach. This essentially transforms the problem of protecting the data into protecting the DEK, introducing two benefits: the data stored can be wiped instantly by erasing the DEK, and setting or changing the password does not require re-encryption of all user data.

B. Hardware encryption standards

ATA Security: The standard for ATA (AT Attachment, with AT being a reference to the IBM PC/AT) storage devices [11] defines the *security feature set*, which allows for locking and unlocking with a password. The goal of the ATA security feature set was limited to access control: it did not aim to stop a well-motivated attacker with physical access. At the time SEDs were first created, it made sense to re-purpose the ATA security password for encryption. However, since the feature set already existed, ATA does not standardize cryptographic primitives or even state that encryption should be used.

SED manufacturers commonly advertise that their products use strong cryptography, such as AES-256. Unfortunately, drive manufacturers typically do not provide encryption implementation details, or in case of ATA security, even state whether encryption is used at all. In our opinion, it is reasonable to assume so. However, the standard is not violated in any way in case the password is used for access control alone. From the ATA standard [11]: "*If security is enabled on the device, the use of the Master password is indicated by the* MASTER PASSWORD CAPABILITY *bit. The* MASTER PASSWORD CAPABILITY *bit represents High or Maximum as described in this subclause. The* MASTER PASSWORD CAPABILITY *bit is modified during the processing of a* SECURITY SET PASSWORD *command that specifies a User password. If the* MASTER PASSWORD CAPABILITY *bit is set to High (i.e., zero), either the User password or Master password are used interchangeably. If the* MASTER PASSWORD CAPABILITY *bit is set to Maximum (i.e., one), the Master password is not used with the* SECURITY DISABLE PASSWORD *command and* SECURITY UNLOCK *command. The* SECURITY ERASE UNIT *command, however, uses either a valid User password or Master password.*"

By default, the Master password is set by the manufacturer. In case the user sets a password, he must take care to either also change the Master password, or set the MASTER PASSWORD CAPABILITY bit to Maximum. If he fails to do so, the Master password allows anyone with knowledge of the factory-default password to access his data.

TCG Opal: TCG Opal [12] is a newer specification for SEDs. It encompasses a communication protocol that is layered on top of ATA or NVMe (Non Volatile Memory express, a recent storage interface). Furthermore, Opal mandates the use of either AES-128 or AES-256. The encryption should meet the bandwidth capability of the storage device. Opal compliant drives allow multiple passwords (*credentials* in Opal terminology) to be defined. Each can be assigned to perform various actions within the Opal subsystem. Special Admin credentials are used to perform provisioning and configuration.

A storage device can be divided into multiple *locking ranges*, that can be locked or unlocked independently. Each locking range is encrypted with a different DEK (*Media Encryption Key* in Opal terminology), and each locking range can be erased independently of the others. A range can be erased by generating a new DEK for that range. A special *global* range is defined as the range that covers all sectors of the disk not covered in other ranges. Multiple passwords can be assigned permission to unlock a particular range. Additionally, a single password can be assigned permission to unlock multiple ranges. Phrased differently: a many-to-many relation exists between passwords and locking ranges.

A scheme supporting all of the aforementioned properties, and cryptographically enforces them, is complex to implement. On top of that, no reference implementation by the Trusted Computing Group exists. Consequently, drive manufacturers all have to design and implement encryption (schemes) themselves. Finally, compliance tests do not reveal design and implementation weaknesses, as they only verify whether the drive behaves as expected given certain sequences of commands. We believe that these circumstances combined are likely to be the root cause of several implementation weaknesses.

Proprietary alternatives: Several proprietary alternatives to TCG Opal exist. Examples are Seagate DriveTrust, the Western Digital MyPassport family of drives and Samsung's portable SSDs. Manufacturers may opt for a proprietary solution for example because the standard may have been introduced before Opal came into existence, or because a simpler scheme is preferred over Opal.

III. Attacker model

Here we list the attacker models relevant to full-disk encryption. In the rest of this article, we will only be concerned with the last one, as the implications of the first two are roughly equivalent when offsetting software against hardware encryption. We do, however, list them all here because it is in our opinion important to state why they are equivalent.

Machine off, no awareness. The adversary has momentary physical access to the powered-down machine, and the victim is unaware of this, creating an opportunity for the so-called *evil maid attack*. The encounter is used to install data exfiltration software or hardware on the victim's machine. In case of a hardware modification, e.g. a physical key logger device, to the best of our knowledge, no meaningful countermeasure exists today. For software modifications, the story is more nuanced. PCs fitted with a *Trusted Platform Module* (TPM) can take advantage of the *sealing* functionality, where cryptographic key material is bound to the software and hardware. Hardware full-disk encryption does not mitigate the evil maid scenario in a meaningful way. Hence, this attacker model is out of scope.

Machine on. The adversary has physical access to a powered-on machine while the encryption containers are unlocked. Software-based encryption solutions typically keep the cryptographic key in RAM, which is vulnerable to *cold boot attacks*, *DMA attacks*, or any other means of data exfiltration, including physical removal and readout with an external device. However, it is worth mentioning that software encryption exists that defends against such attacks, by storing the secret keys in CPU registers [13], [14].

An argument that is often put forward in favor of hardware encryption is that the secret key is not stored in RAM, and therefore is not vulnerable to the aforementioned attacks. In reality, this argument is invalid for several reasons.

First, the software running on the host PC controlling the hardware encryption, typically *does* keep a secret key in RAM, in order to support *Suspend-to-RAM* (S3), a low-power state wherein all peripheral devices are shut down. Since the SSD is powered down, it must be unlocked again once the system is resumed, and therefore either the operating system must retain a copy of the secret key at all times, or the user must enter it again. In virtually all implementations, including BitLocker, the former approach is chosen [15].

Second, the burden of keeping the secret key is moved to the SSD, not eliminated. The SSD typically keeps the key in the main memory of its controller. SSDs are not security-hardened devices by any standard. In fact, many have a debugging interface exposed on their PCB, allowing one to attach a debugging device and extract the secret key. Several means of obtaining code execution on the drive exist (see Section V-B2).

Third, a memory readout attack against software encryption requires physical access. Given this, the attacker also has the opportunity to carry out a hot-plugging attack against hardware encryption. This has been demonstrated in practice [15].

As with the previous attacker model, opportunities and subsequent impact are roughly equivalent compared to software encryption. Therefore, this attacker model is also out of scope.

Machine off, awareness. The adversary has physical access to a powered-down machine, and the victim is aware of this (such as during a border-control search in a back room). Therefore, from that point onward, the victim is unwilling to enter key information into the machine. In this scenario, given that the implementation is sound, software full-disk encryption offers full confidentiality of the data, and hardware encryption supposedly does so as well. In this paper, we focus on this attacker model.

IV. Possible security issues with hardware encryption

Properly implementing a hardware full disk encryption scheme is not trivial, as can be seen by the following list of possible pitfalls. We divide these issues in three categories: *specification*, *design* and *implementation* issues. The issues presented in the remainder of this section are used as a guideline in Section VI in order to assess how well hardware encryption is implemented.

A. Specification issues

Both lack of specification and too detailed specification can have an impact on the difficulty to implement a standard properly. An example of lack of specification and misuse can be seen in the ATA security standard, and an example of too detailed specifications is TCG Opal, both discussed below.

The purpose of the ATA security feature set is limited to access control only. It was never intended to be used for encryption. Nevertheless, manufacturers decided to use it for this purpose. The ATA standard offers no implementation guidance of any kind on how the data is stored securely.

The TCG Opal standard, being specifically designed for the purpose of encryption, addresses this issue. However, it specifies a large feature set, of which the added value is debatable (multiple passwords per range, multiple ranges, see Section II-B for details), in particular, if we take the complexity of correctly implementing such a feature set into account. On top of that, the specification offers no guidance on how the key derivation scheme should be designed that supports this feature set. This lack of guidance combined with many required features is a source of issues, listed below.

1) DEKs are not derived from the passwords: Obviously, the password should be required in order to obtain the DEK, and this requirement should be cryptographically enforced by deriving the encryption key from the password. The absence of this property results in a situation where the confidentiality of user data no longer depends on secrets. All the information required to recover the user data is stored on the drive itself and can be retrieved. We believe that the complexity of TCG Opal in combination with the absence of implementation guidance contributes to conceiving a design in which the DEK ultimately does not depend on the user password.

2) Single DEK used for the whole disk: A naive implementation of the Opal standard uses a single DEK for the entire drive, and store an encrypted variant of it for each password, whereas a proper implementation produces different DEKs for each range. On the surface, doing so may seem only a minor issue. Indeed, access to at least one range is still required. However, the (probably) most popular Opal management software, BitLocker, leaves the global range unprotected in order to allow the partition table to be accessible. Consequently, no secret (password) is needed to access the DEK and can potentially be retrieved from the device, in effect compromising the other ranges. The Opal standard is the root cause of this class of issues since it is the only standard that specifies multiple independent ranges.

3) ATA Master password re-enabling: The ATA security feature set defines both a User and Master password (see section II-B), with the possibility to revoke the Master password's permission to access the drive's contents, i.e. by setting the MASTER PASSWORD CAPABILITY bit to Maximum. Ideally, doing so would trigger the erasure of all key material allowing the DEK to be derived from the Master password. The aforementioned permission can also be reinstated using the User password. As such, the key material should also be restored. However, at that particular point in time, the drive does not have the possession of the cleartext Master password, rendering this operation nontrivial to implement. Theoretically, the issue can be addressed, e.g. by keeping an encrypted copy of the aforementioned key material using the User password as the key. However, doing so prevents the Master password from being changed independently of the User password. This can again be addressed by introducing another constant key and encrypting it using the Master password. In practice, however, all drives included in our study simply keep the key material available at all times.

B. Design issues

The issues listed in this category are design issues of which we believe do not arise from (a lack of) specification.

1) Wear leveling: SSDs use flash memory for data storage. A property of flash memory is that it can be subject to a limited number of write-erase-cycles before becoming unreliable. In order to prolong the service life of the device, *wear leveling* is applied. It works by arranging data so that erasures and rewrites are evenly distributed across the medium. Thus, multiple writes to the same logical sector typically trigger writes to different physical sectors. Older copies of a sector remain stored until overwritten (although not directly retrievable by the end user). Wear-levelling can be applied to key information as well. Suppose that the DEK is stored unprotected, after which a password is set by the end user, overwriting the unprotected DEK with an encrypted variant. Due to wear leveling, the unprotected variant may still be retrievable.

2) Power-saving mode: DEVSLP: DEVSLP is a feature that allows SATA drives to go into a low power 'device sleep' mode when sent the appropriate signal. The ATA standard is not explicit about how power consumption reduction is to be achieved. A manufacturer may freely choose, for example, to have the drive write its internal state to non-volatile storage and subsequently power down the RAM. The drive complies to the standard as long as it can become operational within 20ms of receiving the wake-up signal. Suppose that a drive indeed writes its internal state to non-volatile memory. Then care must be taken that the state from non-volatile memory is erased upon wake-up, or else an attacker may be able to extract the DEK from the last stored state.

C. Implementation issues

Lastly, we list issues that are not inherently caused by the design. Rather, they are issues that potentially occur due to implementation mistakes.

1) Lack of entropy in randomly generated DEKs: The only way for the end user to affect the DEK is by triggering randomization of it. This raises the question if sufficient random entropy is available during the DEK generation. Principally, the environment wherein SSDs are deployed allows for sufficient entropy to be acquired [16]. Example entropy sources include the drive's temperature sensor and I/O requests from the host PC. Storing and restoring the random pool upon reboots should not be an issue since we are concerned with storage devices. However, random number generators in embedded devices have seen a number of issues. [17].

2) General implementation issues: All the issues depicted above in this section apply in particular to hardware-based disk encryption. However, potential implementation issues in software-based encryption may also apply. Examples include re-use of the initialization vectors and using an insecure mode of operation. Many software-based solutions, such as VeraCrypt and later versions of Microsoft BitLocker, use the XTS mode of operation. A description of XTS is given below. The XTS, or *XEX Tweakable Block Cipher with Ciphertext Stealing* [18], mode of operation was designed for cryptographic protection of data on storage devices of fixed length data units. It is an instantiation of Rogaway's XEX (XOR Encrypt XOR) tweakable block cipher [19], extended with ciphertext stealing to support arbitrary length inputs. Furthermore, XEX mode uses a single key for both encryption and tweaking, whereas XTS mode uses two independent keys. XTS mode provides confidentiality for the protected data. Authentication is not provided, because one of the design goals is to provide encryption without data expansion. In the absence of authentication or access control, the best one can do is to ensure that any alteration of the ciphertext will completely randomize the plaintext, and rely on the application that uses this transform to include sufficient redundancy in its plaintext to detect and discard such random plaintexts. In light of this, XTS provides more protection than other confidentiality-only modes against manipulation of the encrypted data. The XTS mode of operation has received criticism [20], [21]. An important point is that the granularity to which an attacker has the ability to randomize plaintexts must equal the cipher's block size which is in the case of AES 16 bytes. Ferguson has designed a native diffuser function that addresses this problem for application in BitLocker [22]. In the same publication, XTS is not mentioned, but LRW mode with the same limitation is criticized.

V. Methodology

In order to assess how well hardware encrpytion in SSDs performs in practice, we argue that we should analyze its implementations. This is, in our opinion, the most realistic measure. Such an analysis is inherently a somewhat ad-hoc process, since implementations vary wildly among manufacturers and models. However, to the extent possible, we document a generic approach that is applied to every device subject to analysis. We will describe each step:

A. Obtaining a firmware image

The difficulty of obtaining a firmware image from an SSD varies greatly among manufacturers and models. Below, we list a few examples.

1) Downloading a firmware update: Most manufacturers distribute firmware updates for their SSDs, by making them available for download from their website or through their SSD management utility. For all the drives we studied, firmware updates consist of the entire firmware image. Firmware updates downloaded from a manufacturer's website often comprise of a bootable ISO image, containing an operating system, firmware update utility, and the firmware image itself. A special command is used by the update utility to apply the update, for ATA it is the 0x92 DOWNLOAD MICROCODE command. Extracting the firmware from the ISO image is typically straightforward. Obtaining a firmware image distributed through SSD management utility typically requires more effort, but is certainly not impossible. For example, the utility may apply obfuscation on its communication channels and/or firmware images that require some reverse engineering in order to remove. Some manufacturers use encrypted firmware images; the image is transferred to the drive, and subsequently decrypted by the drive itself. We can let the drive decrypt these images, and retrieve the decrypted image from the drive (see V-A2).

2) Extract the running firmware: Sometimes, a copy of the running firmware can be extracted. This can be achieved e.g. by using the device's debugging capabilities (see below), or by exploiting a vulnerability in the handling of storage interface commands. In effect, this allows one to extract the currently running firmware from the device's RAM.

B. Gaining low level control over the device

A firmware image allows for static analysis. However, the possibility of dynamic analysis through e.g. JTAG is a significant advantage. It allows us to quickly confirm (or refute) assumptions and findings resulting from static analysis. Furthermore, in case weaknesses are found in the cryptographic scheme, a means of low level control is often required in order to exploit them.

1) JTAG: JTAG allows full control over a device. We can halt/resume the CPU, read/modify registers, place breakpoints, read/write arbitrarily within the address space, and execute arbitrary code. Some SSDs expose a JTAG debugging interface on their PCBs. Standardized pin layouts exist, though, manufacturers may opt for a proprietary one. The JTAGulator [23] allows us to automatically determine whether a set of pins speak the JTAG protocol.

2) Unsigned code execution: Some SSD manufacturers disable the JTAG feature of the storage controller. In the absence of JTAG, a suitable alternative is the ability to execute arbitrary code on the storage controller, as it allows for essentially the same capabilities. However, all drives in our study have countermeasures in place to prevent this, such as cryptographic signature verification of firmware updates.

Still, various means of gaining code execution exist, such as *vendor-specific commands*, *memory corruption*, *storage chip access*, or a *fault-injection attack*. These are described below.

Vendor-specific commands Most manufacturers implement vendor-specific commands for diagnostic purposes. Through static analysis of firmware images, we found examples in which a command exists that allows for arbitrary values to be written to a memory address of choice. This can be leveraged into code execution, e.g. by overwriting a function pointer.

Memory corruption Memory corruption vulnerabilities can in many situations be leveraged into unsigned code execution, a stack-based buffer overflow is an example of this.

Storage chip access A more invasive technique for gaining unsigned code execution is by using an external reader device to make modifications to the currently installed firmware.

The NAND flash chips usually contain the user-accessible storage and firmware. They typically come as BGA packages, requiring them to be desoldered from the PCB in order to attach them to a reader. Alternatively, many SSDs also contain NOR flash, connected through SPI (Serial Peripheral Interface). SPI flash chips usually expose their pins on the outside, therefore not requiring them to be desoldered. Typically, the NOR flash contains the drive's capacity, serial number, error logs, and more. In some occasions, it contains the boot loader. Unsigned code execution becomes possible by making modifications to it.

Fault injection attack Finally, although beyond the scope of this paper, a fault injection attack may be used to achieve unsigned code execution. E.g., a clock glitch during a cryptographic signature check can be introduced by physical interference (power or electro-magnetic), tricking the drive into accepting a firmware update with an invalid signature. In order to successfully mitigate such an attack, both hardware and software countermeasures are necessary. To the best of our knowledge, no SSD on the market has these countermeasures.

C. Analyzing the firmware

Once a firmware image for a particular drive is acquired, we analyze it. The file format used for firmware images differs between manufacturers. The images are usually divided in sections. Section information, such as the size, memory address, and offset in the file, is usually contained within the image header. In some cases, the section information is immediately apparent by inspection. In other cases, some reverse engineering is needed. Once the sector information is uncovered, the firmware image can be loaded into a disassembler and analysis tool, such as IDA Pro.

When reverse engineering SSD firmwares, a good starting point is identifying the *ATA dispatch table*, i.e. an array of data structures containing at least the ATA opcode and the address of the function that implements it. All drives in our study implement the ATA standard in a way similar to this. Once the table is identified, the implementation of any desired command can be studied by analyzing the code located at the respective address.

For each of the possible issues given in Section IV, we attempt to find out whether the drive is susceptible to it by studying the relevant code.

VI. Case studies

A. Crucial MX100

The Crucial (Micron) MX100 is a SATA SSD released in 2014. It supports ATA security, as well as TCG Opal, both version 1 and 2. The controller used is the Marvell 88SS9189, which houses a dual-core ARM CPU. Firmware updates are available for download through Micron's website. They come as a Linux-based bootable ISO image. The firmware image is stored within the ISO image, and is sent unmodified to the drive. The firmware image is cryptographically signed using 2048-bit RSA and SHA256. The signature verification is based on mbedTLS's `rsa_pkcs1_verify` function. The MX100 has a JTAG interface that can be used to connect a debugging device. The standardized ARM14 JTAG pin layout is used.

Findings: In this section, we present our findings with respect to both ATA security and TCG Opal.

ATA security. We found that the implementation of the ATA `F2h` SECURITY UNLOCK command passes the incoming password to the SHA256 hash function, and compares the output to another buffer. If they match, the drive unlocks. However, the original password buffer remains unused during this process. Hence, the DEK is not derived from the password.

TCG Opal. The TCG Opal implementation works in a similar fashion; i.e. no derivation of the DEK from the password. Each locking range is encrypted with a unique key. These keys are stored encrypted using a single key, effectively negating the potential advantages of unique per-range keys.

Other findings All key material is generated by entropy generating hardware, which the firmware refers to as a TRNG. All information related to full-disk encryption is stored in SPI flash, with no wear leveling applied. Several vendor-specific commands were encountered that allow engineers to diagnose the device. A non-exhaustive list is given in Appendix A.

Security evaluation: The MX100 has critical security issues in both the ATA security and TCG Opal implementation. Namely, the DEK is not derived from the password. We demonstrated in practice that, by modifying the password validation routine in RAM through JTAG, the MX100 unlocks with any password. This applies to both ATA security and TCG Opal. With the current key derivation scheme, per-range keys are encrypted using a single key. Thus, introducing a cryptographic dependency on the password would still allow any credential to unlock all ranges. No random entropy issues nor wear leveling issues were identifed.

Furthermore, a vendor-specific command allows for arbitrary modifications within the address space. This enables malware with remote access to the host PC to infect the drive's firmware, allowing it to hide itself and/or to survive re-installation of the host PC's OS.

Attack strategy: In order to to recover the data from a locked MX100 drive, we connect a JTAG debugging device. Then, we use it to modify the password validation routine in RAM so that it always validates successfully. Finally, we unlock the drive as normal, with an arbitrary password. The strategy is the same for both ATA security and TCG Opal.

B. Crucial MX200

The Crucial MX200 is a SATA SSD released in 2015. It is essentially an MX100 with some write performance advantages. The MX200 is built around the same 88SS9189 controller. The firmware is very similar to that of the MX100. In terms of the issues listed in section IV, the MX200 performs identical to the MX100. It too suffers from the lack of derivation of the DEK from the password. Furthermore, the vendor-specific commands found in the MX100 are also present in the MX200 (see Appendix A).

Attack strategy: The attack strategy is identical to that of the MX100. See Section VI-A.

C. Crucial MX300

The Crucial MX300 is a SATA SSD released in 2016. Similar to both its predecessors, it supports the ATA security feature set, as well as TCG Opal version 1 and 2. The MX300 is fitted with a Marvell 88SS1074 controller, the successor to the 88SS9189. The MX300 differs from its predecessors in some aspects, including the controller's JTAG feature being switched off, and the code related to cryptography being subject to a major revision.

Debugging: A firmware image can be obtained through Micron's website. Hence, it can be analyzed. As stated in Section V-A2, JTAG allows for low level monitoring and control of the storage controller's CPU. It significantly aids the analysis, as it allows for verification of assumptions and findings, and possibly exploitation of weaknesses. Hence, absence of this feature is problematic. Therefore, we used the strategies listed in Section V-B2 in order to gain arbitrary code execution on the device. We found that the vendor-specific commands present in the MX100 and MX200 are still present. However, since the MX300, the unlock mechanism

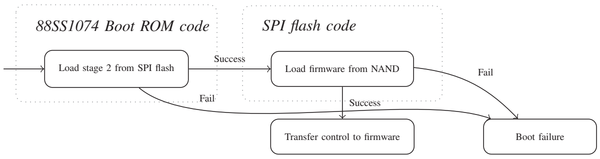

Fig. 1. Crucial MX300 boot process.

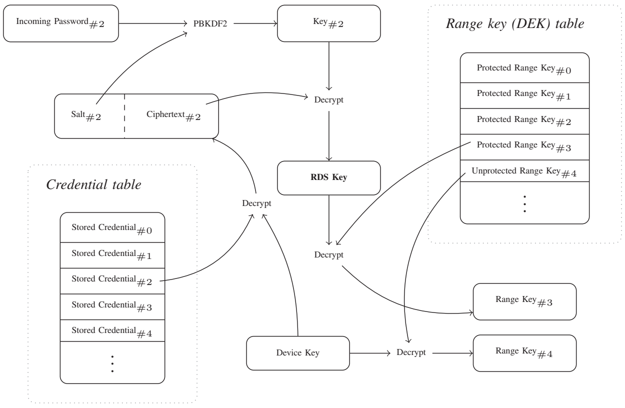

Fig. 2. Scheme used to obtain a range key (DEK) from the user-supplied password. In this example, credential #2 is used to unlock range #3.

is replaced by one that relies on asymmetric cryptographic signatures. Hence, the vendor commands no longer serve as a vehicle for unsigned code execution. Furthermore, we identified several memory corruption vulnerabilities, however we could not exploit them to gain control over the execution. We have acquired arbitrary code execution by manipulating the device's SPI flash, which is described below.

Findings: We used an external SPI communication device to communicate with the drive's SPI flash chip, allowing its contents to be retrieved and manipulated. In order to leverage this into unsigned code execution, we must first understand the drive's boot process, which we reverse engineered. A diagram depicting the boot process is given in Figure 1. Once the storage controller is powered on, the first instructions executed by its CPU are located in a ROM, embedded within the controller. The ROM code loads its next stage, which we refer to as *stage 2*, from the SPI flash, located on the drive's PCB. Stage 2 is responsible for retrieving the drive's firmware from NAND, and then transfering control to it. Hence, by modifying the stage 2 code, one can (indirectly) change the behavior of the drive's firmware. I.e., by injecting code at a particular location so that it runs after the firmware is retrieved from NAND, but before transfering control to it. This way, one can, for example, remove cryptographic signature checks applied during a firmware update.

Ideally, we would like to have the capability of reading, writing and executing arbitrarily within the drive's address space. We crafted a modified firmware image, which includes these capabilities, and installed it by means of the aforementioned cryptographic signature circumvention technique. Once the process is completed, we have these capabilities.

Key derivation scheme We have reverse engineered the key derivation scheme of the MX300. It is depicted in Figure 2. Unlike its predecessors, the MX300 derives the DEK from the password.

Each MX300 drive has a per-device unique key, which we refer to as the *device key*. It is stored in one-time-programmable memory contained within the controller. As such, an attacker is unable to obtain it, unless he has the ability to execute arbitrary code on the controller's CPU.

As is mandated by Opal, the scheme allows for multiple credentials and ranges. Each credential has a data structure associated stored within the SPI flash. This is what we refer to as the *credential table*. Entries within this table are encrypted using the device key. Each entry contains a *salt* and a *ciphertext*. The random salt and the user-supplied password are fed to PBKDF2. The result is then used as a key in an attempt to decrypt the ciphertext. If the password is correct, the decrypted result is the *RDS key* (referred to by the firmware as such). All stored credentials yield the same RDS key.

Each locking range is protected with its own unique DEK. The DEKs are stored in the, what we refer to as, *range key table*. All keys corresponding to protected ranges (i.e. requiring a password before becoming accessible) are encrypted using the RDS key. All other DEKs are encrypted using the device key and are therefore always accessible.

From the description given above, we can already see that the RDS key can be obtained once only a single password is known. Subsequently, the RDS key allows access to all protected ranges. The drive will refuse to unlock a range for a user who does not have permission to access it. However, this check is not cryptographically enforced. This is already a weakness in the design of the key derivation scheme. However, we found that even a single password need not be known, which we explain in detail below.

Opal Setup During the set-up phase of TCG Opal, the credential table and range key table are populated. In order to better understand this process, we used our arbitrary write capabilities to inject tracing functionality at various places in the firmware. The execution trace generated during the BitLocker set-up phase is given in Appendix D. If `sedutil` is used instead of BitLocker, the result is similar. Pseudocode for some of the routines captured is given in Appendix C. From the execution trace, we can clearly see that once the BitLocker set-up phase is completed, the RDS key is protected (encrypted) with a zero buffer as a password, and stored in all credential table slots between 11 and 29, with the exception of slot 15. Hence, the RDS key can be recovered from any of these slots, by invoking the VerifyPasswd on one of them. We use our unsigned code execution capability to do so. As such, this allows any DEK to be decrypted without a password.

ATA security As stated in Section II-B, the MASTER PASSWORD CAPABILITY bit determines whether the factory-set Master password may unlock the drive. In order for the end user to prevent using the Master password, (s)he either has to set the MASTER PASSWORD CAPABILITY bit to Maximum, or change the Master password. In the case of the MX300, the former approach is insufficient. We found that the Master password allows for successful decryption of the RDS key, regardless of the MASTER PASSWORD CAPABILITY bit. Hence, in case the end user has set it to Maximum, but has not changed the Master password, the drive's contents is still accessible to anyone in possession of the default Master password. In the case of the MX300, this is an empty string.

Other findings Similar to the MX100 and MX200, keys are randomly generated by a hardware RNG. All information related to full-disk encryption is stored in SPI flash, with no wear leveling applied.

Attack strategy: In order to recover the data from a locked MX300 drive, we first install a modified firmware that includes arbitrary read/write/execute capabilities. The process is described in detail in Section VI-C. The following steps describe how to recover the data from a drive that is set up through TCG Opal, or ATA security, respectively.

TCG Opal We use our custom firmware's arbitrary write and execute capability in order to write executable code in the device's address space and execute it. Our code invokes the VerifyPasswd function with a zero buffer as password, using credential slot 11 and with bExtractRdsKey set to **true**. At this point, the RDS key is recovered, allowing for all DEKs to be decrypted. By using the arbitrary write capability once more, we modify the VerifyPasswd function such that it always returns SUCCESS. Note that by doing so, the function will no longer affect the global RDS key buffer, which is desired behavior since it already contains the correct key. At this point, any password can be used to 'authenticate' successfully. Note that permission checks are still enforced. However, we can impersonate any desired user.

ATA security We use the arbitrary write capability in order to change the MASTER PASSWORD CAPABILITY bit in RAM from Max (1) to High (0). Then, we authenticate to the drive as normal, using an empty string as the Master password, and unlock the drive. Note that this approach will not work in case ATA security is used instead of Opal, with the Master password changed rather than disabled. However, we believe only a small minority of full-disk encryption users will fall under this category.

D. Sandisk X600

The Sandisk X600 is a SATA SSD released in December 2017. It is built around the same Marvell 88SS1074 controller as the Crucial MX300. The X600 supports both ATA security and TCG Opal version 2. Firmware updates are distributed through Sandisk's *SSD Dashboard* tool. By reverse engineering this tool we gained the opportunity to download firmware images for all Sandisk SSDs, including the X600. However, the X600's firmware image is encrypted and decryption is performed within the drive itself. The controller's JTAG feature is switched off by Sandisk, thus recovery of the encryption keys is not straightforward.

Since the X600 is built around the same controller as the Crucial MX300, its boot process is similar: a bootstrap image located in ROM retrieves stage 2 from an SPI flash chip (see Figure 1). By manipulating the stage 2 code with an external reader, the drive can be re-programmed to expose its firmware encryption keys, or its main firmware as a whole. Contrary to the MX300, however, the X600 has the controller's cryptographic signature validation feature enabled over stage 2. Therefore, modifying it will invalidate its signature, and the controller will refuse to execute it.

We obtained a copy of the 88SS1074 boot ROM by extracting it from our Crucial MX300. Subsequent analysis revealed a weakness, which exists within all 88SS1074 controllers, that allows us to bypass this cryptographic signature validation. As such, unsigned code execution can be obtained on all drives based on this controller, including the X600. No further details about this weakness will be disclosed in this paper, as they are under responsible disclosure embargo at time of publication.

We extracted the firmware from the drive by exploiting the abovementioned weakness (combined with injecting additional code) so that, once the firmware has been retrieved from flash and decrypted, it is copied into the SPI flash. The firmware can then be retrieved with an external reader.

Findings: As stated above, the boot-time cryptographic signature validation feature can be bypassed. This can not only be leveraged into obtaining a copy of the firmware, but also into full control over the device. We used the opportunity of manipulating stage 2 to insert modifications into the firmware, so that the device accepts an ATA command allowing for reading, writing, and executing within the device's address space. This functionality greatly benefits our analysis, as it allows the device's memory to be inspected and manipulated at runtime. We found that the X600 derives the DEK from the user password. However, other severe issues exist allowing for the full-disk encryption to be compromised in many situations.

TCG Opal. Although the X600 shares no code with the Crucial MX300, the design pattern behind its Opal implementation is similar: all passwords allow for a single key (*RDS key* in Crucial terminology, we continue to use this term here) to be obtained, which allows the DEK associated with any range to be decrypted (see Figure 2). Contrary to the MX300, the code does not follow alternative paths for protected versus unprotected ranges. Thus, in case one or more unprotected ranges exists, the RDS key must be available in order to support this use case. The drive does this by keeping a so-called *anonymous key-encrypting-key (KEK)*, which is essentially the RDS key encrypted with a zero buffer. We found that the anonymous KEK is absent only in case a the global range is the only range defined, and it is password protected. Any other configuration causes the anonymous KEK to be present, effectively reducing the security to the equivalent of no encryption.

ATA security. We found that, in case the drive's full-disk encryption is configured by means of ATA security, the user-supplied password (be it a user or master password) is used in a scheme that yields the RDS key. However, we also found that the anonymous KEK remains present within file 97 (see below) at all times. As such, no password is required in order to access the drive's contents. Moreover, in case the user upgrades from high to max mode, cryptographic key information allowing the RDS key to be derived from the master password is not erased. Therefore, the master password still serves as a means to access the drive's contents.

Wear leveling The X600 stores settings and other internal data in a small internal file system. The file system uses file numbers rather than names, but apart from this pecularity, functionality seems to be similar to any other file system. The file system has a wear leveling feature built-in. File number 97 contains all information related to cryptography.

We used our arbitrary write primitive to alter the behavior of a function issued by the file system driver that retrieves data from the raw flash. We did so such that, besides its normal behavior, its arguments (block number, die number, plane number, etc.) are stored in a buffer, so that they can later be inspected. Through this we learned that each time file 97 is written to, its physical location within flash changes. However, in case the anonymous KEK is removed from file 97, the drive immediately issues a routine referred to as *file system compaction*, which erases previous copies of any file from its flash chips. We confirmed that no previous copy of file 97 exist once the process is completed. Other operations causing file 97 to be changed (i.e. changing a password) do not trigger the file system compaction, and hence a previous version can be recovered. However, this is only useful to an attacker in case a previous password is either known or easy to guess. We found that previous copies of file 97 can typically be recovered, until a file system compaction is issued. Besides explicitly (as seen previously), this also happens in case the file system runs out of free space. However, to the best of our knowledge, no files are modified under normal behavior limited to unlocking and reading/writing. Hence, these copies are likely to linger for a significant amount of time.

Cryptographically signed firmware updates In addition to information related to drive encryption, file 97 also contains two secret keys for the encryption and signature verification of firmware updates. Both schemes are based on symmetric cryptography. Thus, once these keys are obtained, they can be used by an attacker to encrypt and sign custom firmware updates, which will subsequently be accepted by all X600 drives. Sandisk seems to rely on the premise that it is infeasible to obtain these keys and therefore sacrifices the security benefits of asymmetric cryptographic signatures in exchange for performance.

Other findings We found that all encryption key material is generated by means of a hardware RNG.

Attack strategy: In many cases, the contents stored on a Sandisk X600 can be retrieved without a password. We make a distinction between the case wherein only the global range is defined and password protected, and otherwise. In both cases, we assume the drive runs a modified firmware that allows for arbitrary modifications in the device's address space, achieved either by bypassing the signature verification in the boot ROM, or by any other means listed in Section V-B2.

Suppose that drive encryption is configured through ATA security, or at least a single range is defined besides the global range. Then the RDS key, and hence all DEKs, are recoverable by decrypting the *anonymous KEK*, located in file 97, with a zero buffer as the key. This is already done during the drive's startup procedure. Thus, in order to access any protected range, only the password validation routine need be modified so that it accepts any password. Furthermore, some adjustments are required in order to prevent the correct RDS key to become overwritten with the result of a decryption with an incorrect password. Finally, the protected range can be unlocked with an arbitrary password.

In the other case, the anonymous KEK is absent from file 97, and all previous copies of it are erased from flash. In this case, the data on the drive is likely secure. However, in case the user changed the password because it was compromised, the data can likely be recovered using the compromised password. This can be done by scanning through the raw flash, looking for previous copies of file 97. Once found, the current version can be replaced with the previous one, and subsequently, the drive can be unlocked using the compromised password.

E. Samsung 840 EVO

The Samsung 840 EVO is a SATA SSD released in 2013. It supports ATA security, as well as TCG Opal version 2. At its core is Samsung's own MEX controller, built around a triple-core Cortex R4 (ARM).

Firmware updates are downloadable through Samsung's website. They come as bootable ISO images. The firmware image can be found within the ISO image, albeit in an obfuscated form. De-obfuscation is performed by the update utility itself. Hence, recovery of the obfuscation algorithm is straightforward. The obfuscation algorithm has been previously reverse engineered [24]. Once de-obfuscated, the image is transferred to the drive using the ATA `92h` DOWNLOAD MICROCODE opcode. From this point onward, the firmware update process takes place on the drive itself. The firmware image is cryptographically signed with ECDSA. The curve and its exact parameters are yet to be determined. The hash function used is SHA256.

The 840 EVO has a JTAG interface with a proprietary pin layout. It was found with help of the JTAGulator [23]. It was independently found by [25].

Findings: **Key derivation scheme** Firstly, a data structure is used by the firmware that provides for both password validation and key derivation. It contains two salts and a hash result. Entries in the password storage table shown in Figure 3 are of this structure. Password validation is performed by computing PBKDF2, with the user-supplied password as key, over the first salt, $\text{Salt}_{\text{verif}}$. If the output matches the hash result contained within the data structure, validation succeeds. Subsequently, the derived key is obtained by computing another PBKDF2 using the same password, over the second salt, $\text{Salt}_{\text{deriv}}$.

TCG Opal Samsung's Opal implementation allows for a total number of 9 ranges and 14 credentials to be specified. For all 14 credentials, a table entry exists containing the aforementioned password validation/derivation data structure. Once the user-supplied password is validated against one of the entries, the derived key is then used to decrypt an entry in

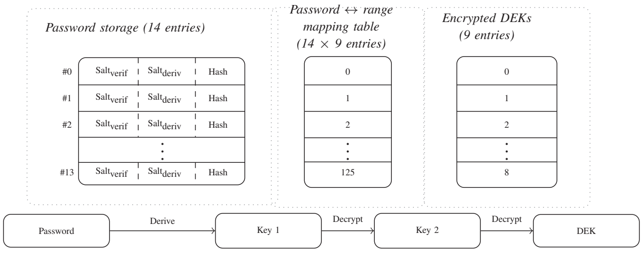
Password storage (14 entries)
Password ↔ range mapping table (14 × 9 entries)
Encrypted DEKs (9 entries)
#0
#1
#2
#13
Salt$_{verif}$
Salt$_{deriv}$
Hash
0
1
2
125
8
Password
Derive
Key 1
Decrypt
Key 2
Decrypt
DEK

Fig. 3. Relation between password and DEK on the Samsung 840 EVO

a table that maps credentials to ranges, i.e. this table is 9×14 entries wide. Permission of a certain credential to access a particular range is determined by the existence of an entry in this table. Finally, the decrypted result is then used as a key to decrypt an entry in the DEK table. This final step is required in order to support erasure of independent ranges by re-generating its corresponding DEK, without requiring knowledge of all passwords that unlock it. As such, all Opal properties are cryptographically enforced. All the data required in order to support this scheme is contained within a 64 KB binary blob, which we refer to as, the *crypto blob*.

Vendor-unique commands The 840 EVO features several vendor-specific commands. They are listed in Appendix B.

ATA security The DEK *may* be cryptographically tied to the ATA password. This depends on the value of the MASTER PASSWORD CAPABILITY bit during the ATA security setup. In case it is set to Maximum, the DEK cryptographically depends on the User password. In High mode, however, there is no dependency. Thus, this allows the encryption to be bypassed.

TCG Opal After reverse engineering and carefully studying the design of the key derivation scheme used in the drive TCG Opal implementation, we have not identified any weaknesses.

Random entropy The 840 EVO has a hardware RNG. However, in many situations, a pseudo RNG is used, which works by encrypting an incrementing counter using the AES co-processor. The pseudo RNG is seeded with data supplied by the hardware RNG. All key material related to full-disk encryption is also generated by the hardware RNG. We assume that the output generated by the hardware RNG is cryptographically secure.

Wear leveling The Samsung 840 EVO stores its crypto blob within the device's NAND flash, within a region designated for internal data structures. Despite this, the crypto blob storage is wear-leveled. Suppose that at time t_0, the drive is in an unprotected state, i.e. neither ATA security nor TCG Opal is set up. In this state, the drive has a single locking range defined that covers the entire user-accessible storage. The DEK for this range is contained unprotected within the crypto blob. At time t_0, the crypto blob is stored at physical sector s_0 in flash. Subsequently, at time t_1, a password is set, either through ATA security, with the MASTER PASSWORD CAPABILITY bit set to Maximum, or through TCG Opal. As such, the password is required in order to obtain the DEK from the crypto blob. The updated crypto blob is stored at sector s_1 in flash.

Due to the wear leveling mechanism, $s_0 = s_1$ is not guaranteed. Therefore, from time t_1 onward, the DEK can be recovered by retrieving the crypto blob from physical sector s_0. This is mitigated again as soon as s_0 is overwritten. We have successfully demonstrated this attack in practice. Once a previous revision of the crypto-blob has been recovered, it can be made active through a vendor-specific command (see Appendix B).

Empirical measurements indicate that $s_0 \neq s_1$ occurs approximately 1 in every 20 times the crypto-blob is stored (i.e. every time crypto related information is updated). Furthermore, s_0 is overwritten within roughly one week of casual office use. As such, the attack vector is mostly theoretical, as finding previous copies of the crypto blob at an arbitrary point in time is very unlikely.

Attack strategy: Suppose that we want to recover the data from a locked 840 EVO drive. The approach taken depends on whether the drive is protected with the ATA security feature set, with the MASTER PASSWORD CAPABILITY bit set to High. If this is the case, then the DEK does not cryptographically depend on the password. Hence, the only barrier we have to overcome is the password validation routine. We connect a JTAG debugging device and modify the password validation routine such that it always validates successfully. Finally, we unlock the drive as normal, with an arbitrary password.

If ATA security (with the MASTER PASSWORD CAPABILITY bit set to Maximum) or TCG Opal is used, then the DEK is cryptographically tied to the password. However, due to the wear-leveling issue pointed out in Section VI-E, there is a slight chance that the data on the drive can still be recovered by reverting to a previous version of the crypto blob that was used while the drive was in an unprotected state.

In order to do this, first, we craft code that searches the raw NAND flash for crypto blobs, at the region designated for internal data structures. Through JTAG, we load the code into the device's address space and execute it. For all crypto blobs found, we determine whether it contains the unprotected DEK. In case we find a crypto blob with this property, we have all the cryptographic secrets needed for a full recovery. Having the previous version of the crypto blob at our disposal, the next step is to instantiate it. A vendor-specific command exists (see Appendix B) that conveniently allows us to do so. At this point, in case the drive was protected through ATA security, the contents are accessible. In the case of TCG Opal, the drive still demands a password. However, this can be overcome by, once more, crippling the password validation routine. Finally, the drive can be unlocked with any password.

F. Samsung 850 EVO

The Samsung 850 EVO is a SATA SSD released in 2014. Similar to the 840 EVO, it supports TCG Opal version 2. It is based around Samsung's MGX controller, which, contrary to the 840 EVO, is a dual-core Cortex R4.

Similar to the 840 EVO, downloadable firmware images are obfuscated. Although the obfuscation function is different, de-obfuscation is still performed on the host PC. The firmware image is again cryptographically signed with ECDSA. The implementation is likely a copy of that of the 840 EVO. The 850 EVO has the same JTAG pin layout as the 840 EVO.

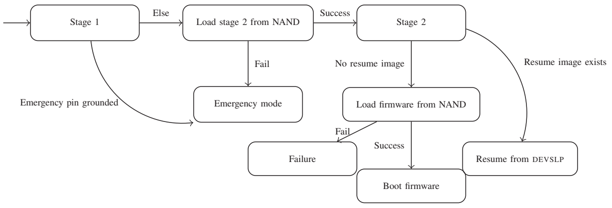

Fig. 4. Samsung 850 EVO boot process.

Findings: The motivation for analyzing the 850 EVO internals is twofold. Firstly, it is valuable to verify whether the weaknesses identified in the 840 EVO are also present in its successor. Secondly, the 850 EVO supports DEVSLP, and other drives of the same family likely use the same or a very similar implementation. In case DEVSLP is not implemented carefully, it may compromise the encryption (see section IV-B2).

Key derivation scheme The Opal key derivation scheme has not changed significantly since the 840 EVO. The implementation is still based around a crypto blob, although its size has doubled, resulting in a 128 KB crypto blob. The exact reason for this remains to be researched. The Opal key derivation scheme is identical. Furthermore, the vendor-unique commands listed in Appendix B have remained unaltered.

DEVSLP mode In case the DEVSLP signal is received, all secret key information present in SRAM is encrypted using a hardcoded key. The result is copied to DRAM. Subsequently, four 'magic' numbers are written to DRAM, and finally, the cores and SRAM are powered down.

In order to determine whether portions of secret key information reach non-volatile storage, we reverse engineered the boot process of the drive. A diagram picturing the code flow during the boot process is given in Figure 4.

The first portion of code is, what we refer to as, the Stage 1 boot loader. Essentially, its purpose is to retrieve Stage 2 from NAND and execute it. However, in case the emergency pin is grounded, or in case the firmware cannot be retrieved, the drive goes into an emergency state. In this state, the drive accepts firmware images through a proprietary protocol layered over UART. The protocol was reverse engineered by [25].

Once Stage 2 is reached, the DRAM is initialized. Shortly after, the decision is made to either resume from a previous state, or to perform a normal startup procedure. The decision is made based on hardware I/O address `0x10050040`, bit 3. Before reverting to the previous state, a check is performed on whether the magic numbers written to RAM previously have remained unaltered.

No I/O addresses related to NAND are interacted with, indicating that the DRAM is kept powered during DEVSLP. We devised the following steps in order to confirm it:

(i) Modify a firmware image, such that within the Stage 2 boot loader, all references to `0x10050040` are replaced so that a DEVSLP resumption scenario is simulated. Furthermore, at the point in the code where the magic numbers are checked, an infinite loop is inserted.
(ii) Modify the currently running firmware in RAM such that it accepts firmware updates with invalid signatures.
(iii) Flash the modified firmware image through the ATA 0X92 DOWNLOAD MICROCODE command. The drive will not reboot.
(iv) Send the DEVSLP signal. The drive goes into DEVSLP mode.
(v) Power up the drive by sending the DEVSLP signal again.
(vi) The execution is stuck at the point where the infinite loop is inserted. Halt the execution and verify that the magic numbers in DRAM are present.
(vii) Power down the drive by removing the power plug.
(viii) Power it up again. The execution is stuck at the same point. In case the magic numbers still exist in DRAM, they must have originated from non-volatile storage. If absent, either the non-volatile storage device is erased during (v), or volatile storage is used.
(ix) Use the emergency mode to flash an unmodified version of the firmware, repeat all previous steps and omit (v) and (vi). Absence of the magic values in DRAM confirms that volatile storage is used.

By pursuing the above steps, we confirmed that the secret key information is indeed kept in volatile storage. The reason for encrypting it with a hardcoded key remains unclear.

Security evaluation: The implementations of full-disk encryption in the 850 EVO and the 840 EVO are very similar. Using the ATA security mode the drive can be tricked into granting access to its contents, in case the MASTER PASSWORD CAPABILITY bit is set to High, just as with the 840 EVO.

Since TCG Opal implementation is mostly identical to its predecessor, no weaknesses have been identified. As is the case with the 840 EVO, the 850 EVO features a hardware RNG, with the added possibility to use a PRNG based on AES.

Wear leveling Unfortunately, despite numerous efforts, we were unable to identify the routines responsible for storing/retrieving the crypto blob from NAND flash. However, during the responsible disclosure trajectory, a contact at Samsung informed us that from the 850 EVO series onward, the crypto blob storage is no longer wear leveled. Instead, a fixed physical address in NAND is used for the crypto blob storage. Therefore, contrary to its predecessor, the 850 EVO is not vulnerable to the crypto blob recovery attack (see 840 EVO).

Attack strategy: The attack strategy is identical to that of the 840 EVO, with the exception of the wear leveling issue not being present. See Section VI-E for further details.

G. Samsung 950 PRO

The Samsung 950 PRO is an NVMe SSD released in late 2015. It supports TCG Opal version 2. The controller is, again, developed in-house: the Samsung UBX. Except for the switch from SATA to NVMe, the controller and firmware share many commonalities with the 850 EVO. As such, the move to NVMe does not seem to provoke any other major revision to the architecture of an SSD.

The firmware image file format is also similar to that of the 850 EVO, however, besides the host-side obfuscation, an additional layer of encryption is applied, which is removed by the drive itself during the firmware update process. The drive has a JTAG debugging interface, thus the firmware can be extracted from RAM.

Findings and Attack strategy: We found that the implementation of the cryptography is very similar, if not identical, to that of the 850 EVO. This entails all the points listed in section IV, including the vulnerable implementation of ATA security. As such, the attack strategy is also the same. Interestingly, ATA security can be used through NVMe, even though it is an extension of ATA. However, we believe it is unlikely that a substantial share of 950 PRO users are affected, since for full disk encryption, TCG Opal is likely the preferred solution, and legacy implementations do not exist since NVMe was standardized later than Opal.

H. Samsung T3 portable

The Samsung T3 portable SSD is an external drive connected through USB-3.1 Gen 1. It offers optional password protection through a proprietary command set. The drive comes with a tool that allows the user to set or remove a password, lock and unlock.

Physically opening the drive uncovers that it is essentially an 850 EVO mSATA behind a USB to mSATA bridge, albeit fitted with a special firmware supporting the proprietary command set. No firmware update for this drive is available. Fortunately, the firmware can be extracted from RAM through JTAG.

Capturing USB packets with the help of Wireshark during locking and unlocking of the drive reveals that the ATA opcode `8Eh` (vendor-specific) is used for both operations. Analysis of the firmware reveals that the implementation of the operations is built upon the ATA security functionality of the 850 EVO. However, it resembles the behavior observed when the MASTER PASSWORD CAPABILITY bit is set to High. Thus, the password is not cryptographically linked to the DEK.

Attack strategy: Similar to an 850 EVO set up using ATA security with the MASTER PASSWORD CAPABILITY bit set to High. The password validation routine can be crippled through JTAG, allowing one to unlock the drive with any password.

I. Samsung T5 portable

The Samsung T5 portable SSD is the successor of the T3. It uses the same MGX controller found in the 850 EVO and the T3. A notable difference between the T5 and its predecessor is that its USB to mSATA converter support for USB-3.1 Gen 2.

Another important difference is that the JTAG feature is disabled. Additionally, the emergency pin is also no longer functional. Finally, no firmware updates for the T5 are available for download. Hence, for this drive, we do not have a firmware image at our disposal.

The T5 features the same vendor-specific commands found in all other Samsung SSDs (Appendix B). Thus, despite the lack of a firmware image and debugging capabilities, the crypto blob can still be transferred from/to the device.

We retrieved a copy of the crypto blob by means of the vendor command both before and after setting a password, and inspected the differences. We refer to these blobs as B_0 and B_1, respectively. The crypto blobs are encrypted (obfuscated) with a per-device one-time-programmable key stored within the controller. As such, it can not be extracted without JTAG or unsigned code execution, both of which we do not have. However, since XTS mode is used, we can observe whether or not the two blobs differ on a per-block (16 bytes) granularity. By studying the T3 firmware, and assuming the implementation is broadly the same, we found that the differences between B_0 and B_1 are explained by the following modifications to the plain-text crypto blob:

(i) The crypto blob revision number.
(ii) A data allocation bitmap determining for each slot whether or not it is in use.
(iii) The key storage data structure (Fig. 3).
(iv) The so-called ‘security state’ byte (referred to in the firmware as such).

In the absence of proper derivation of the disk encryption key, the security state byte alone likely determines the locking state of the drive, and reverting it to its previous state will result in the drive being unlocked. We create a new crypto blob B_1', which is constructed by taking B_1 and selectively reverting the 16-byte block containing the security state byte by taking its ciphertext value from B_0. Subsequently we upload the B_1' crypto blob to the drive through the designated vendor-specific command. We found that the drive successfully unlocks after pursuing the steps above, confirming that the password is indeed not used to derive the DEK.

Attack strategy: Although the steps given above confirm that the T5 lacks derivation of the DEK from the password, the steps themselves do not serve as an attack strategy, as (a portion of) the crypto blob from a previous state, B_0, is needed. However, as we confirmed, protection of the user data is not cryptographically enforced. Hence, a means of low level control over the device, e.g. unsigned code execution, will allow us to bypass it.

Acquiring unsigned code execution on the device is considerably time-consuming and labor-intensive. Given that we exploited the issue in practice on the T5's predecessor, the T3, and given that the exact same issue is confirmed to exist in the T5, it is in our opinion justified to skip the act of acquiring code execution on the T5, solely for the purpose of developing an exploit for this issue.

For completeness: unsigned code execution may by accomplished via one of the methods described in Section V-B2. Once accomplished, one can deploy the same strategy as with the T3 (Section VI-H), i.e. modifying the password validation routine in RAM so that it accepts any password, and subsequently unlocking the drive as normal with any password.

VII. Discussion

A non-exhaustive overview of possible flaws in hardware-based full-disk encryption was given, categorized in specification, design and implementation issues (Section IV). We have analyzed the hardware full-disk encryption of several SSDs by reverse engineering their firmware, focussing on finding these possible flaws. These drives were produced by three manufacturers and sold between 2014 and 2018, have a SATA,

Drive	1	2	3	4	5	6	7	8	9	Impact
Crucial MX100 (all)	✗	✗	✗		✗		✓	✓		Compromised
Crucial MX200 (all)	✗	✗	✗		✗		✓	✓		Compromised
Crucial MX300 (all)	✓	✓	✓		✗	✗	✓	✓		Compromised
Sandisk X600 (SATA)	✓	✓	✓		✗	✗	✓	✗		Probably compromised
Samsung 840 EVO (SATA)	✗	✓	✓		✓		✓		✓	Depends
Samsung 850 EVO (SATA)	✗	✓	✓		✓		✓	✓	✓	Depends
Samsung 950 PRO (NVMe)	✗	✓	✓		✓		✓	✓	✓	Probably safe
Samsung T3 (USB)				✗			✓	✓		Compromised
Samsung T5 (USB)				✗			✓	✓		Compromised

[1] Derivation of the DEK from the password in ATA Security (High mode)
[2] Derivation of the DEK from the password in ATA Security (Max mode)
[3] Derivation of the DEK from the password in TCG Opal
[4] Derivation of the DEK from the password in proprietary standard
[5] No single key for entire disk
[6] Not vulnerable to ATA Master password re-enabling (only if derivation is present)
[7] Randomized DEK on sanitize and sufficient random entropy
[8] No wear leveling related issues
[9] No DEVSLP related issues

TABLE I
OVERVIEW OF CASE STUDY FINDINGS.

NVMe, or USB interface, and have a M.2, traditional 2.5", or external form factor. The analysis uncovers a pattern of critical issues across vendors, due to problems in all three categories. For multiple models, it is possible to bypass the encryption entirely, allowing for a complete recovery of the data without any knowledge of passwords or keys. Table I gives an overview of the models studied, and the flaws found.

The situation is worsened by software solutions delegating encryption to the drive. As a primary example, BitLocker does this delegation for supported drives and disables its software encryption, relying entirely on the hardware implementation. As this is the default policy, many BitLocker users are unintentionally using hardware encryption, exposing them to the same threats as when using a hardware encryption only setup.

The results presented in this paper show that one should not rely solely on hardware encryption as offered by SSDs for confidentiality. Since the encryption in these drives is always performed, disabling hardware encryption (equaling to storing the DEK unprotected) offers no performance benefits. Thus, users currently relying on these features may continue using them. *However, we strongly encourage users that depend on hardware encryption implemented in SSDs to install an open source, preferably audited, disk encryption software solution as soon as possible*. In particular, VeraCrypt allows for in-place encryption while the operating system is running, and can co-exist with hardware encryption. Based on our vulnerability disclosure, additional information has been released by Microsoft [26] and Samsung [27], however Crucial did not. Sandisk (Western Digital) may release information in a later stage (when their disclosure period ends).

As for recommendations, we have structured them according to the categories of issues we have found. Based on early feedback we have received, we want to make clear these issues are not exhaustive but are exemplary of the underlying issues.

The *specification issues* found can be addressed by making simpler standards, with a clearer guide on how to implement them correctly. From a security perspective, standards should favor simplicity over a high number of features. The complexity of storage standards such as TCG Opal contributes vastly to the difficulty of implementing the cryptography in SEDs. A modern standard for self-encrypting drives that is simpler to implement is highly preferable over Opal. In particular, the requirement of multiple ranges is a needlessly complex feature and should be removed. Doing so implies that a scheme supporting a many-to-many relation is no longer necessary. In fact, such a standard already exists today. Opalite [28] defines a subset of Opal's features and it is also authored by the Trusted Computing Group. Unfortunately, to the best of our knowledge, drives that support it are extremely rare. Standard organizations such as TCG should publish a reference implementation of their standards (such as Opal) to aid implementors of the standard, which should be made available for public scrutiny. If additional requirements are needed (such as multiple ranges), they should be implemented as an additional layer.

We found several *design issues*. There is not much public information available on how to design crypto schemes with the requirements as set out in the standards. Therefore, hardware encryption currently comes with the drawback of having to rely on proprietary, non-public, hard-to-audit crypto schemes designed by their manufacturers. Designs should be audited and subject to as much public scrutiny as possible. Manufacturers that take data confidentiality and security seriously want to publish their crypto schemes (and corresponding code) so that security claims can be verified. Any design should take into account that wear-leveling is applied to the storage.

We did not find any *implementation issues* in the analyzed drives. However, we do note that any compliance tests that are made available for standards should also cover the implementation of the cryptography, to keep avoiding these problems in the future. These tests too should be independently assessed.

In general, we can ask ourselves what problem SEDs are trying to address. SEDs do not offer any meaningful mitigations in situations where software encryption falls short (see Section III). However, as demonstrated, in situations where software encryption offers full data confidentiality, hardware encryption often does not. Hence, at best, the data confidentiality guarantees of SEDs are similar to that of software encryption, and often much less. The traditional advantage of SEDs was performance, but this is no longer the case as the AES-NI extension on x86 CPUs has become mainstream. Therefore, the industry should reevaluate any preference for hardware encryption, as software encryption has the benefit that its workings are easier to verify and audit. This hold especially for open-source software solutions, but also proprietary ones as reverse engineering software-only solutions take less effort as opposed to reverse engineering the works of SEDs. A start of this reevaluation has been made, as since our public release Microsoft has made available a new preview version (build 18317) of the next Windows 10 version (19H1). In that version, the default behavior of BitLocker is to not delegate (and trust) the encryption of data to the drives.

References

[1] Statista. (2018) Global market share of solid state drive suppliers. [Online]. Available: https://www.statista.com/statistics/412158/global-market-share-solid-state-drive-suppliers

[2] J. Domburg, "Hard disk hacking," 2013, see http://spritesmods.com/?art=hddhack.

[3] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, "Implementation and implications of a stealth hard-drive backdoor," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 279–288.

[4] T. Goodspeed, "Active disk antiforensics and hard disk backdoors," in *Talk at 0x07 Sec-T Conference (video: https://www.youtube.com/watch?v=8Zpb34Qf0NY)*, vol. 8, 2014.

[5] L. Cojocar, K. Razavi, and H. Bos, "Off-the-shelf embedded devices as platforms for security research," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec'17. New York, NY, USA: ACM, 2017, pp. 1:1–1:6. [Online]. Available: http://doi.acm.org/10.1145/3065913.3065919

[6] J. Horchert, J. Appelbaum, and C. Stöocker, "Shopping for spy gear: catalog advertises NSA toolbox," *Der Spiegel*, 2013.

[7] R. Verdult, "The (in) security of proprietary cryptography," Ph.D. dissertation, Radboud University, Nijmegen, 2015.

[8] J. Domburg and Tweakers.net, "Secustick gives false sense of security," 2007, see https://tweakers.net/reviews/683/secustick-gives-false-sense-of-security.html.

[9] G. Alendal, C. Kison, and modg, "Got HW crypto? on the (in) security of a Self-Encrypting Drive series." *IACR Cryptology ePrint Archive*, vol. 2015, p. 1002, 2015.

[10] D. Boteanu and K. Fowler, "Bypassing self-encrypting drives (SED) in enterprise environments," in *BlackHat Europe*, 2015.

[11] C. Stevens, "AT Attachment 8-ATA/ATAPI Command Set – 4 (ACS-4)," *Working Draft, American National Standard, Revision 14*, 2016. [Online]. Available: http://www.t13.org/documents/UploadedDocuments/docs2016/di529r14-ATAATAPI_Command_Set_-_4.pdf

[12] Trusted Computing Group, "TCG storage security subsystem class: Opal specification version 2.01," 2015.

[13] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *USENIX Security Symposium*, vol. 17, 2011.

[14] T. Müller, B. Taubmann, and F. C. Freiling, "Trevisor," in *International Conference on Applied Cryptography and Network Security*. Springer, 2012, pp. 66–83.

[15] T. Müller, T. Latzo, and F. C. Freiling, "Self-encrypting disks pose self-decrypting risks," in *the 29th Chaos Communinication Congress*, 2012, pp. 1–10.

[16] K. Mowery, M. Wei, D. Kohlbrenner, H. Shacham, and S. Swanson, "Welcome to the entropics: boot-time entropy in embedded devices," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 589–603.

[17] J. Wetzels and A. Abbasi, "Wheel of fortune: Analyzing embedded OS random number generators," 2016.

[18] "IEEE standard for cryptographic protection of data on block-oriented storage devices," *IEEE Std 1619-2007*, pp. c1–32, April 2008.

[19] P. Rogaway, "Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2004, pp. 16–31.

[20] T. Ptacek and E. Ptacek, "You don't want XTS," 2014, see https://sockpuppet.org/blog/2014/04/30/you-dont-want-xts/.

[21] D. Clunie, R. Shroeppel, P. Rogaway, V. Bharadwaj, and N. Ferguson, "Public comments on the XTS-AES mode," *Collected email comments released by NIST, available from their web page*, 2008.

[22] N. Ferguson, "AES-CBC+ Elephant diffuser: A disk encryption algorithm for Windows Vista," Tech. Rep., 2006.

[23] J. Grand, "JTAGulator: assisted discovery of on-chip debug interfaces," in *21st DefCon Conference, Las Vegas*, 2013.

[24] D. Chen. (2015) Firmware deobfuscation utilities. [Online]. Available: https://github.com/ddcc/drive_firmware

[25] P. Gühring, "The missing Samsung EVO 840 - 250 GB SSD repair manual," 2016-2018, see http://www2.futureware.at/~philipp/ssd/TheMissingManual.pdf.

[26] Microsoft. (2018) Guidance for configuring BitLocker to enforce software encryption. [Online]. Available: https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/ADV180028

[27] Samsung. (2018) Consumer notice regarding Samsung SSDs. [Online]. Available: https://www.samsung.com/semiconductor/minisite/ssd/support/consumer-notice/

[28] Trusted Computing Group, "TCG storage security subsystem class: Opalite specification version 1.00," 2015.

Algorithm 1 ProtectPasswd

```
Require: abRdsKey, abDeviceKey, aabCredentialTable
Ensure: Credential szPasswd is stored in aabCredentialTable at dwSlotNo
  procedure PROTECTPASSWD(szPasswd, bStoreRdsKey, dwSlotNo)
      if bStoreRdsKey then
          abPlaintext ← abRdsKey
      else
          abPlaintext ← [0x00 × 32]              ▷ abPlaintext is a zero buffer
      abSalt ← RANDOM(32 bytes)
      abKey ← PBKDF2(szPasswd, abSalt)
      abCiphertext ← ENCRYPT(abKey, abPlaintext)
      stProtectedPasswd ← (abSalt, abCiphertext)
      abOutput ← ENCRYPT(abDeviceKey, stProtectedPasswd)
      aabCredentialTable[dwSlotNo] ← abOutput
```

APPENDIX A
VENDOR COMMANDS AVAILABLE ON THE CRUCIAL MX100 AND MX200

The Crucial MX100 and MX200 feature several vendor-specific commands that allow engineers to diagnose the device. The commands must be unlocked before they can be used. The list of commands presented below is far from exhaustive.

Unlocking. Unlocking the vendor-specific features is done by issuing a `FDh` (vendor-specific) ATA command, with feature code `55h`. Setting the LBA to `306775h`, and the *block count* to `65h` will unlock the vendor-specific commands.

SPI flash functions. Both the MX100 and the MX200 have a NOR flash connected through the SPI bus. It contains, among other things, the drive's capacity, serial number, error logs, and boot loader (see Section VI-C). Vendor-specific ATA command `FAh`, with feature code `D2h` reads a page from the SPI flash and returns the result. The LBA represents the page number that is to be retrieved. Likewise, command `FCh`, feature `E2h` erases a page. Command `FBh`, feature `D2h` writes data to the SPI flash.

Arbitrary memory write. The MX100 allows one to write arbitrary data to any desired address within the address space. The opcode is `FBh`, feature code `23h`. The command expects a concatenated list of address-value tuples.

APPENDIX B
VENDOR COMMANDS AVAILABLE ON THE SAMSUNG 840EVO

The 840 EVO features several vendor-specific commands. As is the case with the Crucial drives, these commands require unlocking. Only a small subset of commands are analyzed, since the vast majority are not security related.

Unlocking the vendor-specific features is done by issuing a `85h` (vendor-specific) ATA command with feature code `46h`. The payload is a single block (512 bytes) with the last 16 bytes set to

`C7D0B1B3C1BEC0CCB6AFB6AFBEEEBCAD`

The crypto blob can be retrieved by issuing a `83h` (vendor-specific) ATA command, with feature code `12h`. Likewise, the crypto blob can be overwritten with command `83h`, feature code `13h`.

Security evaluation: We managed to identify several implementation mistakes, two of which, depending on the circumstances, can be leveraged into full recovery of the data.

APPENDIX C
PSEUDOCODE OF VARIOUS ROUTINES IN THE CRUCIAL MX300 FIRMWARE.

The ProtectPasswd function (Algorithm 1) takes a password and stores it in the credential table so that an incoming password can be checked for validity at a later point in time. The bStoreRdsKey parameter determines whether the stored credential should encapsulate the RDS key. In that case, the credential allows access to protected ranges (see Figure 2).

The function VerifyPasswd (Algorithm 2) is the inverse of ProtectPasswd. It has two purposes: checking the validity of a password, and, in case the bExtractRdsKey parameter is set, using the password to decrypt the RDS key and copying it to the global RDS key buffer, allowing other functions to use it.

Algorithm 2 VerifyPasswd

```
Require: abRdsKey, abDeviceKey, aabCredentialTable
Ensure: Verify szPasswd and set global RDS key if bExtractRdsKey = true
  function VERIFYPASSWD(szPasswd, bExtractRdsKey, dwSlotNo)
      abInput ← aabCredentialTable[dwSlotNo]
      stProtectedPasswd ← DECRYPT(abDeviceKey, abInput)
      if decrypt failed then
          return ERROR
      (abSalt, abCiphertext) ← stProtectedPasswd
      abKey ← PBKDF2(szPasswd, abSalt)
      abPlaintext ← DECRYPT(abKey, abCiphertext)
      if decrypt failed then
          return ERROR
      if bExtractRdsKey then
          abRdsKey ← abPlaintext
      return SUCCESS
```

Furthermore, the UnwrapDek function (Algorithm 3) takes an entry from the range key table, and decrypts it using either the RDS key, or the device key (for protected and unprotected ranges, respectively), as determined by the bIsProtectedRange parameter. Obviously, for protected ranges, the RDS key must be decrypted, prior to invoking UnwrapDek.

Algorithm 3 UnwrapDek

```
Require: abRdsKey, abDeviceKey, aabRangeKeyTable,
          aabUnwrappedRangeKeyTable
Ensure: Range key dwRangeNo is unwrapped
  function UNWRAPDEK(dwRangeNo, bIsProtectedRange)
      if bIsProtectedRange then
          abKey ← abRdsKey
      else
          abKey ← abDeviceKey
      abCiphertext ← aabRangeKeyTable[dwSlotNo]
      abPlaintext ← DECRYPT(abKey, abCiphertext)
      if decrypt failed then
          return ERROR
      aabUnwrappedRangeKeyTable[dwSlotNo] ← abPlaintext
      return SUCCESS
```

Finally, we implicitly define the functions WrapDek, CopyCredential, GenerateRandomDekAndWrap, and StoreCryptoContextInSpiFlash as their functionality is clear from their names.

APPENDIX D
EXECUTION TRACE CAPTURED ON A CRUCIAL MX300 DRIVE DURING THE BITLOCKER SET-UP PHASE.

```
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
CopyCredential(dwSourceSlot=2, dwDestinationSlot=10)
ProtectPasswd(szPasswd=[0x00 × 32], bStoreRdsKey=true, dwSlotNo=11)                    ▷ szPasswd is zero buffer
CopyCredential(dwSourceSlot=11, dwDestinationSlot=12)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=13)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=14)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=15)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=16)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=17)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=18)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=19)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=20)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=21)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=22)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=23)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=24)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=25)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=26)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=27)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=28)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=29)
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=10)
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
ProtectPasswd(szPasswd=«BitLocker SID password», bStoreRdsKey=true, dwSlotNo=2))
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=10)
ProtectPasswd(szPasswd=«BitLocker SID password», bStoreRdsKey=true, dwSlotNo=10)
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
GenerateRandomDekAndWrap(dwRangeNo=1, bIsProtectedRange=false)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
StoreCryptoContextInSpiFlash()
UnwrapDek(dwRangeNo=1, bIsProtectedRange=false)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
UnwrapDek(dwRangeNo=1, bIsProtectedRange=false)
WrapDek(dwRangeNo=1, bIsProtectedRange=true)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
ProtectPasswd(szPasswd=«BitLocker user password», bStoreRdsKey=true, dwSlotNo=15)
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd=«BitLocker user password», bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=«BitLocker user password», bExtractRdsKey=true, dwSlotNo=15)
```