

Securing hard drives with the Security Protocol and Data Model (SPDM)

Renan C. A. Alves, Bruno C. Albertini, and Marcos A. Simplicio Jr.
Dept. of Computer and Digital Systems Engineering
Universidade de São Paulo, Brazil
{renanalves, balbertini, msimplicio}@usp.br

Abstract—In modern computing systems, it is usually hard to defend against attacks made against the low level communication between hardware components. In this work, we address this issue by adapting a hard disk controller and a corresponding device driver to communicate securely using the recently proposed Secure Protocol and Data Model (SPDM). Essentially, SPDM standardizes authentication of hardware components, besides enabling the establishment of secure communication channels among them. To assess the overhead introduced by SPDM in this scenario, we tested our solution against an unmodified, unsecured hard disk on an emulated environment. Our experiments show that, while SPDM can make the task of copying a large up to 7× slower, the transference of small files is virtually unaffected.

Index Terms—hardware, security, SPDM

I. INTRODUCTION

Hardware level attacks are hard to detect, since they bypass protective measures acting at operating system level (e.g., antivirus software) and infrastructure level (e.g., firewalls). Such attacks usually involve altering the firmware controlling the hardware. For example, a USB flash drive can be programmed to mimic a keyboard, and then input commands at will [1]. Also, hard drive firmware can be modified to leak arbitrary data pieces even if partitions are encrypted [2].

The Secure protocol and Data Model (SPDM) addresses hardware-level security in multiple manners. First, it enables mutual authentication among components. Second, each endpoint may provide measurements that help verifying if the device's current state is as expected. Examples of measurements include firmware, hardware configuration, and other parameters. Finally, since its version 1.1.0, SPDM allows components to establishment of session keys for encrypting and authenticating exchanged application data [3].

Our goal in this article is to demonstrate a working prototype of an SDPM-enabled hard drive, evaluating its protected communication with the operating system. Specifically, we present our general design (Section II), implementation details (Section III) and experimental data (Section IV). Closing remarks are then presented in Sec. V.

II. PROJECT DESIGN

Securing the communication between the operating system and a peripheral (e.g., a hard drive) essentially requires altering the device's firmware and writing a matching device driver.

During system boot process, the endpoints should run the SDPM protocol to authenticate each other and establish

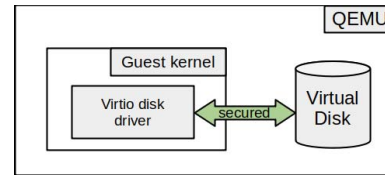


Fig. 1. Secured communication between operating system and hard drive

symmetric keys that are used to secure application data. In our scenario, application data is comprised of read/write requests issued by the kernel and the corresponding responses from the device.

Considering the goal of providing a proof-of-concept implementation, we resorted to a virtualized environment, rather than crafting a customized hardware. The main software employed to build the desired environment is QEMU, an open source emulator/virtualizer. We ran QEMU in a standard Linux box as the host OS, while we ran a customized Linux distribution (based on Buildroot) as the guest OS. Among the hard drives QEMU can emulate, we chose the virtio disk due to its widely available documentation and simplicity.

Figure 1 illustrates the system architecture, including the secured communication between OS and virtual disk. Notice that the goal is to secure in-transit data, which protects against attacks at the communication bus but, by itself, does not store encrypted data in the disk.

III. IMPLEMENTATIONS DETAILS

This section covers implementation details, including a brief introduction to the Linux block I/O system, how SPDM was thereby integrated, and specific implementation challenges.

A. Linux block I/O

Due to the layered nature of the Linux kernel, the task of reading and writing from a disk is spread across the file system, a general block layer, and the specific block device driver. Put simply, file systems deal with *where* the data is (e.g., whether it is cached, and how it is scattered throughout the disk). The block layer is a bridge between the file system and device driver, taking care of request queuing and request scheduling. Finally, the block device driver executes the requests that were organized by the block layer.

A request is represented by a complex data structure, whose definition spans across over 100 lines of code. Nonetheless, the main components used at the device driver layer are the

target disk sector, a pointer to the actual data, the data size, and the request type.

B. SPDM integration

We use `libspdm` open source library to implement SPDM functionalities. The library's build system had to be modified to provide adequate binaries to be linked with the Linux kernel. Also, since the kernel has a reduced stack space, the library had to be modified to address this limitation.

The device driver was modified to encrypt/decrypt the write/read requests payload using the symmetric keys established with SPDM. First, a header had to be included to indicate an SPDM encrypted message. Additionally, the size of a single request from the block layer would often exceed the maximum limit allowed by the `libspdm` API. This issue was addressed by repacking the original requests into smaller requests of adequate size, and encoding pure SPDM messages with a special request type. The code is available at <https://github.com/rcaalves/spdm-hd-demo>.

C. Corner cases

During the test phase, we noticed that some messages encrypted by the device driver were failing to be decrypted at the disk controller. At both endpoints, `libspdm` internally maintains a monotonically increasing 64-bit counter that is used as a nonce. Hence, message decryption would fail whenever the order of messages sent/received differed from the encryption order. To address this issue, we modified the counter's behavior: we build upon the fact that the 16 least significant bits of this 64-bit counter is placed at the message header, using it directly while keeping track of 16-bit overflows (to ensure the nonce's non-repeatable nature).

Also, the strategy of repacking requests at the device driver increased the load on an internal data pool used to allocate new requests. As a result, whenever the pool is exhausted and the device driver attempts to allocate a new request, the underlying request allocation function tries to execute pending requests. However if the next pending request is actually the request that triggered the new allocation (due to the repacking strategy), a loop is created, ultimately leading to a stack overflow. As a straightforward approach to address this issue, we increased the capacity of the default data pool.

IV. EXPERIMENT RESULTS

The experiments presented in this section consist of measuring how long it takes to copy files of different sizes from/to a partition on an SPDM-enabled virtio disk partition. As baseline, the same procedures were executed on an unmodified virtio disk. Comparing both, we can estimate the performance impact of securing OS-device communication with SPDM.

The files were filled with random data, for sizes varying from 5KiB, 10KiB, 50KiB, up to 100MiB. They were then copied using the standard `cp` Linux command line tool, whereas the duration was measured by the `gettimeofday` C function. Each configuration was executed 30 times, each

time after clearing the system cache, to achieve statistical significance.

Figure 2 shows our experimental results. The impact of SPDM security is negligible for small file sizes (up to 5k), as sometimes we obtained smaller average copy times for the secured system. This implies other overheads (e.g., physically moving disk heads) are predominant in this setting.

Conversely, for medium-sized files the gap between the secured an unsecured systems begins to be noticeable, although not statistically evident. On average, copying files from 100k to 500k was 32% slower. The performance of the secured hard disk degrades as file size increases, reaching up to 7 times slower transference procedure. It is also interesting to notice that virtio to virtio transactions show the largest performance drop, since the data has to be encrypted twice in that case: from disk to OS, and then from OS to disk.

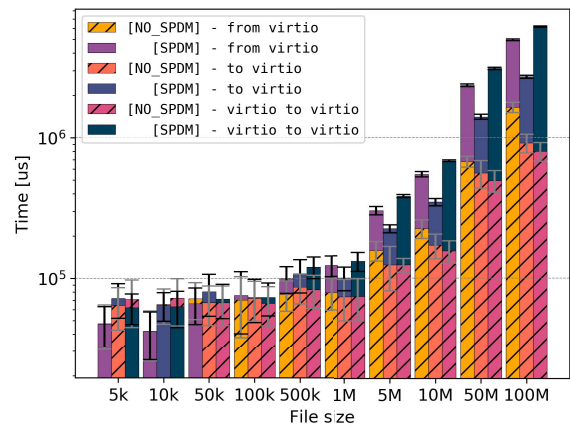


Fig. 2. Time to copy files of varying sizes in different settings

V. FINAL REMARKS

Securing the communication between internal components is a key mechanism to avoid attacks targeted at electronics supply chains. In this paper, we showcase how the SPDM standard can be used to secure the communication between an operating system and a hard drive controller firmware.

For our setup, we modify a device driver and a virtual hardware, integrating them with the `libspdm` open source library. Our experiments show that file copying performance may drop up to 7 times on large files, although this overhead remains negligible for small files.

Acknowledgment. The authors would like to thank Hewlett Packard Enterprise for supporting this work.

REFERENCES

- [1] B.-C. Choi, S.-H. Lee, J.-C. Na, and J.-H. Lee, "Secure firmware validation and update for consumer devices in home networking," *IEEE Transactions on Consumer Electronics*, vol. 62, no. 1, pp. 39–44, 2016.
- [2] J. Menn, "NSA Can Hide Spyware in Hard-Disk Firmware," <https://www.vox.com/2015/2/17/11559082/nsa-can-hide-spyware-in-hard-disk-firmware>, accessed: 2021-12-16.
- [3] DMTF, "DSP0274: Security protocol and data model (SPDM) specification, v.1.1.0," Distributed Management Task Force, Tech. Rep., Jul 2020, www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.1.0.pdf.