Are hardware write blockers more reliable than software ones?

Maxim Suhanov

Introduction

Many forensic examiners and attorneys share the opinion that hardware write blockers are more reliable than software write blockers, this judgment is also expressed, directly and indirectly, in different publications [1][2][3]. In this article, I will try to bring internals of hardware and software write blockers to light, discussing various problems existing in these products.

Principles of operation

Hardware write blockers

All hardware write blockers can be divided into two groups based on how they handle commands received from a host:

- operating based on a whitelist;
- operating based on a blacklist.

A hardware write blocker is operating based on a whitelist when it blocks any command going to a drive if it is not included in a list of known safe commands (that are not modifying data stored on a drive). In this mode of operation, a write blocking device will block all unknown commands, including vendor-specific (e.g., used to perform low-level diagnostics of a drive) and new (not yet implemented in write blocker's firmware) ones. Such a write blocker can block newly standardized safe commands, treating them as unknown.

A hardware write blocker is operating based on a blacklist when it blocks commands included in a list of known unsafe commands (that are modifying data stored on a drive or performing other dangerous actions) and allows any other command going to a drive. In this mode of operation, a write blocking device will allow unknown (vendor-specific or newly standardized) unsafe commands going to a drive.

Also, all hardware write blockers can be divided into two groups based on their implementation details:

- acting as a command translator;
- acting as a block device sharing box.

A hardware write blocker acts as a command translator when it simply translates allowed commands received from a source interface by repeating them to a destination interface. For

example, a simple SATA-to-USB write blocker can receive SCSI commands from a USB interface (using an SCSI transparent command set for the mass storage class), and for each allowed SCSI command perform a request to a SATA controller using AHCI, forwarding any replies back to a host using the SCSI protocol.

A hardware write blocker acts as a block device sharing box when it contains a fully-featured operating system, an attached drive is recognized as a block device in this operating system, and read access to this block device is shared with a host using a special driver. Such a write blocker with a USB connection to a host will recognize an attached drive as a block device, and then use a USB gadget to enable this block device as a backing storage for an emulated USB drive. In this configuration, a hardware write blocker is not performing direct translation of commands received from a host to a target drive, it translates commands received from a host to internal requests used to read data from a backing block device instead. Thus, multiple read commands from a host can be merged into a single read request resulting in a single read command being sent to an attached drive. Also, write blocker's firmware can perform read-ahead caching, so a read command from a host may not result in a corresponding read command being immediately sent to an attached drive, because data requested was read and cached by firmware before.

Additionally, hardware write blockers can provide special features for some common and uncommon use cases:

- a read-write mode of operation;
- allowing write commands, but storing modified data on another drive;
- presenting a drive to a host as write protected (this provides an additional layer of protection, because an operating system is not expected to write to a read-only drive);
- suppressing write errors;
- exposing data areas hidden using HPA or DCO;
- allowing several unsafe commands to expose hidden data areas (removing DCO or permanently removing HPA);
- transparent data recovery from bad (faulty) sectors encountered on a drive by rereading them, transparent handling of faulty drives.

Software write blockers

Implementation details of software write blockers depend on an operating system being used. In real mode operating systems like DOS, software write blockers are intercepting the 0x13 BIOS interrupt used to read from and write to a disk, filtering out write requests, and calling the original interrupt handler for read requests. Modern operating systems like Windows and GNU/Linux rely on native drivers when communicating to a drive, the 0x13 BIOS interrupt is used by a boot loader to read a kernel and other data (like native drivers) during the early boot sequence only. So, there are many ways to implement a write blocking functionality, for example:

- a read-only driver for a specific storage class (PATA, SATA, SCSI, USB, etc.);
- a program (a driver) that is filtering out write requests going to an underlying driver for a specific storage class;
- a driver that is providing a read-only block device for a selected drive (or a partition), while a read-write block device for the same drive (partition) is still present in an operating system (an examiner is expected to use tools against a read-only block device).

Depending on the implementation, software write blockers can analyze and block the following types of requests:

- read, write, flush, and other requests in a unified format specific to an operating system (that does not depend on an underlying protocol used to communicate to a drive);
- requests in a format based on a protocol used to communicate to a drive (e.g., SCSI) or implementing that protocol directly.

Also, a software write blocker can present a drive to an operating system as a read-only device.

In modern operating systems, the following write blocking implementations were seen:

- Windows: setting up a filter driver for I/O request packets used to send SCSI commands to a low-level storage port driver (the operating system is using the SCSI protocol to communicate with storage port drivers, these request packets are translated later, if required, to a protocol used by specific hardware);
- Linux:
    - setting up a read-only loop device for a block device of a drive (or a partition), in this situation a loop device driver will filter out write requests going to a backed block device (the kernel is using its own structure to describe read, write, flush, and other requests to drives), an examiner is expected to use the read-only loop device when working with data;
    - patching the kernel in order to filter out write requests going to a block device marked as read-only.

When dealing with requests constructed using a native protocol (like SCSI in Windows), a software write blocker can operate using the blacklist and whitelist approaches mentioned before.

In general, software write blockers should intercept requests either in a common spot (e.g., before a request is sent to one of many storage drivers) or in all places possible (e.g., in all storage drivers).

It should be noted that a software write blocker cannot secure all possible ways for an unsafe command to reach the drive. For example, a kernel can provide an interface to send raw requests to a drive (e.g., the SG_IO interface in Linux), or an architecture of storage drivers allows a driver to send a request bypassing a filter driver (true for Windows), or a low-level driver can issue unsafe commands on its own, or a program can simply disable the write blocker. Although some steps can

be performed to mitigate these issues, the idea is simple – a way to write to a software write blocked drive always exists.

Quasi software write blocking

It is possible to build an operating system that does not send unsafe commands to attached drives during and after the boot, unless a user runs a program sending unsafe requests explicitly. In this case, there is no write blocking component present, but there are no commands to be blocked too (and such operating systems are often referred to as implementing a write blocking mechanism, thus the word “quasi” is used in this section).

Unfortunately, some products claim to have a software write blocker included, but, in fact, they do not, and, depending on different circumstances, such a product can send unsafe commands to an attached drive.

Comparing hardware write blockers with software ones

The following essential differences between hardware and software write blockers can be mentioned:

- Hardware write blockers introduce a break between a host and a drive, so all unsafe commands from a host should be blocked regardless of their origin. Software write blockers can be bypassed by a malicious program, as described before.
- Software write blockers for native drivers are inactive during the early boot sequence: no write blocking is present when a boot loader is using the 0x13 BIOS interrupt or EFI services to load a kernel and other components of a modern operating system. While hardware write blockers do not process any command until they are initialized.

There is an opinion that software write blockers are unreliable, because they depend on fragile environment: for example, an operating system update, or a driver update, or a bug (in hardware or software) can break the write blocking mechanism; hardware write blockers, on the other side, are considered to be reliable, because they contain stable, well-tested hardware and firmware [4].

Validating the forensic soundness

Quasi software write blockers

SUMURI PALADIN 4.01

PALADIN is a live forensic distribution (based on Linux) designed to boot a suspect computer in order to preview data or acquire it. Also, this distribution can be used on a forensic workstation as a preconfigured operating system.

According to the documentation available in PALADIN 4.01, *PALADIN has been modified to write-protect all attached media upon boot*. However, this version contains no write blocking component,

and the following modifications to attached media were observed when booting the distribution[1]:

- if an attached medium contains an unclean (not unmounted properly) Ext3/4 file system, this file system is recovered using its journal;
- if an attached medium contains an unclean NTFS file system, the journal ($LogFile) of this file system is wiped.

These issues relate to the early boot sequence of live distributions based on Ubuntu or Debian, when programs in an initial RAM file system are executed in order to find the boot medium by mounting file systems on each drive (including evidentiary drives) and searching for specific signs in these file systems (like a file containing a predefined UUID)[2].

Despite these issues being present in this and several other versions of PALADIN, the distribution was validated by NIST with no issues of this kind recorded [5].

Software write blockers

SUMURI PALADIN 6.01

PALADIN 6.01 includes a write blocking component for the kernel (Linux), which was silently implemented without any mention in the documentation or in the change log. This component was silently removed from later versions of the distribution (e.g., PALADIN 6.07).

During the validation, the following bug was identified in the write blocking implementation: during the early boot sequence, when programs in the initial RAM file system are executed, the write blocking component is denying write and discard requests to all block devices with the major number 8 only. In this implementation, only PATA/SATA/SCSI/USB drives are write blocked during the boot stage mentioned, while, for example, media in card readers (MMC) are not write blocked, being a subject to issues described before.

Hardware write blockers

Tableau TD3 forensic imager

The Tableau TD3 forensic imager (firmware version: 2.0.0) can be used as a network-based write blocker, allowing access to an attached evidentiary drive over the iSCSI protocol. The iSCSI driver is blocking write requests going to the evidentiary drive, but no write blocking component is present in the kernel (Linux) nor in hardware. It was observed that attaching a drive containing an Ext4 file system with an I/O error recorded in the journal will result in several write commands (modifying the file system) being issued by the imager through a “write blocked” port, because an operating system of the imager (based on Linux) is automatically mounting file systems on a drive attached to

1 This and subsequent sections do not contain an exhaustive list of issues found. Only some indicative issues are mentioned. For the same reason, similar issues found in other forensic products based on Ubuntu, Debian, or other distributions are not listed.

2 These actions are required to complete the transition from the 0x13 BIOS interrupt or EFI services used by a boot loader to native drivers used by a kernel when reading from a boot medium.

a “write blocked” port.

Tableau T356789iu forensic bridge

The Tableau T356789iu forensic bridge (firmware version: 1.3.0) is blocking a host from reading sectors adjacent to a bad one. It was observed that a single bad sector present on an attached drive will result in 128 bad sectors being reported as read failure to a host. It was found that the bridge is using the read-ahead and cache functionality of the kernel (Linux) when reading data from an attached drive (the kernel can read and cache sectors before they were requested by a program, read requests from programs are served through cache), and the kernel has poor error granularity (it does not reread individual sectors within a large block to be cached after read failure for that block).

Tableau T35es forensic bridge

According to Guidance Software [6], Tableau T35es forensic bridges with old firmware were allowing WRITE(16) SCSI commands received from a host (over a USB connection) to reach an attached drive. This issue is an example of blacklist failure.

Tableau T8-R2 forensic bridge

According to Guidance Software [7], Tableau T8-R2 forensic bridges with old firmware were writing to an attached drive when connected to a host over a USB connection under unspecified conditions. No details were provided, except that data written to the drive is random.

Conclusions

Definitely, hardware write blockers are not more reliable than software ones, there could be critical issues with both of them (especially when hardware write blockers contain general-purpose operating systems as firmware). But this answer is not satisfiable for the computer forensics community, because it does not give us a solution to resolve the problem.

In my opinion, the following statements should be taken into account and implemented:

- the community needs better computer forensics tool validation methodologies and methods;
- these methods should cover all common use cases;
- these methods must not rely solely on black-box testing;
- computer forensics tool developers should not hide issues they confirmed nor deploy bug fixes silently.

Why do we need better computer forensics tool validation methodologies and methods?

The answer is simple: because current methodologies and methods do not satisfy the current needs of the community.

Software write blocking tool testing methodologies and methods published by NIST [8][9] do not cover write blocking mechanisms in operating systems based on Linux, as well as they do not cover any additional issues existing in live Windows and Linux distributions (like automatic code execution from evidentiary media when booting from a USB drive by selecting an evidentiary medium as the boot one, or treating a file system on evidentiary media as containing updates to be applied automatically to a live environment).

Testing methodology and accompanying documents for hardware write blockers published by NIST [10][11] do not focus on a situation in which a hardware write blocker is writing to an attached drive on its own, without a corresponding command from a host. Issues with error granularity are not covered by these documents too. In general, the hardware write blocking tool testing methods mentioned before imply that a hardware write blocker is a command translator.

Why do we need methods that cover all common use cases?

Because common use cases of computer forensic tools are broader than a typical validation effort.

For example, NIST did not notice data modification issues with PALADIN 4.0, because tests performed did not include unclean file systems. However, unclean file systems are common when performing forensic examinations (e.g., after pulling the plug).

Two recent reports published by NIST [12][13] state that live forensic distributions were tested together with hardware write blockers, but common use cases for live forensic distributions include data acquisitions without a hardware write blocker (for example, when dealing with drives that either cannot be removed from a computer or cannot be attached to a hardware write blocker). For sure, such results cannot be extended to data acquisitions without a hardware write blocker.

Why do we need methods that do not rely solely on black-box testing?

Because a number of common use cases is beyond real capabilities.

File systems contain different flags that can be set or unset in their superblocks or similar structures, a journaling file system can be used without a journal, and so on. File systems can be stored on hard disk drives, solid state drives, or other media, and an operating system can behave differently depending on this. It is infeasible to take all possible states of a file system (and, thus, all possible code execution paths that can lead into write requests being sent) into account.

Based on this fact, it is feasible to identify weak spots using a white-box approach first, and then develop test cases for black-box tests.

Why do we want developers to uncover issues they confirmed and bug fixes they deployed?

Of course, any modifications made to original evidence do not automatically mean that this evidence becomes inadmissible. For example, data acquisitions from mobile devices and running

systems (live forensics) imply modifications to an original medium in order to access the data. But these acquisition types are not an excuse for modifications made by tools that are not expected to do so (although such modifications still do not make evidence inadmissible in all cases).

Depending on legal requirements, a forensic examiner might want to know what modifications to original evidence, if any, are performed by a tool in question, the reasons and prerequisites for this behavior, and how evidence is affected in each situation. Without information from developers (like a detailed changelog for issues related to the forensic soundness), it is not possible to satisfy these requirements in full.

References

1. Nikkel B. Practical forensic imaging. 2016.
2. Morton T. Introduction to Digital Forensics. 2011. Available from: https://en.wikibooks.org/wiki/Introduction_to_Digital_Forensics
3. Al Falayleh M. and Al-Karaki J. On the Selection of Write Blockers for Disk Acquisition: A Comparative Practical Study. 2013.
4. Menz M. and Bress S. The Fallacy of Software Write Protection in Computer Forensics. 2004. Available from: http://mykeytech.com/softwarewriteblocking2-4.pdf
5. NIST. Test Results for Digital Data Acquisition Tool: Paladin 4.0. 2014. Available from: https://www.dhs.gov/sites/default/files/publications/508_Test%20Report_NIST_Paladin%204%200_August%202015_Final_0.pdf
6. Guidance Software. Tableau Firmware Update: TFU v6.80. 2010. Available from: https://tableau.guidancesoftware.com/index.php?pageid=firmware&releaseID=28&model=T35ES&view=overview
7. Guidance Software. Tableau Firmware Update: TFU v6.84. 2011. Available from: https://tableau.guidancesoftware.com/index.php?pageid=firmware&releaseID=24&view=overview
8. NIST. Software Write Block Tool Specification & Test Plan. 2003. Available from: http://www.cftt.nist.gov/documents/SWB-STP-V3_1a.pdf
9. NIST. ACES Test Suite User's Guide. 2008.
10. NIST. Hardware Write Blocker (HWB) Assertions and Test Plan. 2005. Available from: http://www.cftt.nist.gov/HWB-ATP-19.pdf
11. NIST. Hardware Write Blocker Device (HWB) Specification. 2004. Available from: http://www.cftt.nist.gov/HWB-v2-post-19-may-04.pdf
12. NIST. Test Results for Disk Imaging Tool: Paladin v6.09. 2016. Available from: https://www.dhs.gov/sites/default/files/publications/1496_508_Test%20Report_NIST_Disk%20Imaging_Paladin%20v6.09_October_14_2016.pdf

13. NIST. Test Results for Disk Imaging Tool: Paladin v6.08. 2016. Available from: https://www.dhs.gov/sites/default/files/publications/1495_508_Test%20Report_NIST_Disk%20Imaging_Paladin%20v6.08%201.0.5_October_14_2016.pdf