RZ 3806 (# Z1106-001) 06/27/11 (updated 09/06/2011)
Computer Science 16 pages

Research Report

On the Feasibility of Data Exfiltration with Storage-Device Backdoors

(Updated Version of September 06, 2011)

Anil Kurmus,[1] Moitrayee Gupta,[2] Ioannis Koltsidas[1] and Erik-Oliver Blass[3]

[1]IBM Research – Zurich
8803 Rüschlikon, Switzerland
Email: {kur,iko}@zurich.ibm.com

[2]Department of Computer Science and Engineering
UCSD
La Jolla, CA 92093, USA
Email: m5gupta@cs.ucsd.edu

[3]EURECOM
06560 Sophia Antipolis, France
Email: blass@eurecom.fr

On the Feasibility of Data Exfiltration with Storage-Device Backdoors

Anil Kurmus* Moitrayee Gupta† Ioannis Koltsidas* Erik-Oliver Blass‡

Abstract

Hardware backdoors are a substantial threat to today's information systems: they can evade today's malware detection mechanisms and survive software updates. Moreover, they are an increasingly likely threat because of extensive outsourcing of hardware manufacturing. While the feasibility of implementing backdoors in CPUs, PCI devices, and network components has been studied before, this paper investigates a new type of threat: a backdoor that leverages storage devices. We show that a remote attacker can exfiltrate data from a storage device in the absence of a direct communication channel and without a priori knowledge of the various layers (OS, applications, filesystem) between the attacker and the device. We implement such a backdoor to demonstrate the real-world feasibility of attacks. Our experiments show that `/etc/passwd` of a standard Ubuntu/Apache/PHP/MySQL installation can be remotely exfiltrated in 40 seconds. Consequently, we conclude that this attack vector should not be overlooked when assessing a system's security, and we discuss, e.g., encrypting data at rest to thwart such attacks.

1 Introduction

Backdoors allow an attacker to trigger unauthorized or unexpected operations on a system. When they operate at a low layer, e.g., within a device's firmware, they are difficult to detect and can operate covertly. Such hardware backdoors can be installed in numerous ways: for example, during the manufacturing process of the hardware by a malevolent employee or a supply-chain compromise. In addition, many devices from trusted parties have been known to contain backdoors, e.g., rootkits for copyright protection [1] or lawful interception capabilities in network devices [2, 3]. Recent reports on hard disks shipping with viruses [4] show that such threats are also realistic in the context of storage devices. Moreover, backdoors implemented in firmware can be installed on a system shortly after its OS has been compromised, e.g., by an update to the firmware of the target hardware component, to maintain access even after a clean re-installation of the OS.

Compromised hardware is typically used to compromise additional system components, e.g., by using auto-run or filesystem vulnerabilities [5] or DMA capabilities on systems lacking I/O memory management units (IOMMU). Departing from that kind of attack, our paper focuses on a storage firmware backdoor that does not modify the control flow of other components: such backdoors are therefore less intrusive and less dependent on the layers above (e.g., the OS, applications and filesystem). As these backdoors are less

*IBM Research - Zurich, 8803 Rüschlikon, Switzerland. Email: {kur,iko}@zurich.ibm.com.

†Department of Computer Science and Engineering, UCSD, La Jolla, CA 92093, USA. Email: m5gupta@cs.ucsd.edu.

‡EURECOM, 06560 Sophia Antipolis, France. Email: blass@eurecom.fr.

intrusive, they are less likely to be detected, e.g., by existing mechanisms guaranteeing OS integrity [6].

The aim of such a backdoor is to allow a remote attacker to remotely read data from the subverted storage device, i.e., perform *data exfiltration*. This requires establishing a bi-directional communication channel between the attacker and the storage device, which can be challenging: a remote attacker cannot directly communicate with the storage device. However, most Internet-based services, such as web forums, blogs, cloud services or Internet banking, will eventually store and retrieve a user's data to and from a storage device. The attacker can therefore piggy-back its communications with the subverted storage device on those existing external communication channels, provided that practical issues that we will describe later (such as the use of caches, data alignment, or compression) can be overcome.

We assume a *threat model* where an attacker initially modifies the firmware, hardware, or both, of a target storage device, e.g., by having physical access to the device in the case of hardware tampering, or by performing firmware updates through a remotely compromised OS. At that point in time, the device is not necessarily used by the victim to store data (e.g., it is still at the production facility). However, as soon as the device is used by the victim, the attacker no longer has his previous access to the device, e.g., the device is shipped to a facility with physical security preventing hardware access, or, in the case of the remote firmware update, the OS is reinstalled. As indicated by recent events and reports [2–4, 7], such a model is realistic.

In summary, we make the following major contributions. In Section 2, we present the general design of a new storage-device backdoor allowing a remote attacker to establish a covert data channel with the storage device. We also demonstrate the potential appeal of such backdoors to attackers using a concrete attack scenario for data exfiltration in a real-world setting: a web server providing a typical blogging or forum service. In Section 3, we validate this attack scenario against a web server by *implementing* a proof-of-concept storage-device *data-exfiltration backdoor* (DEB). For ethical reasons and ease of prototyping, we choose not to reverse engineer and modify the firmware of a real hard disk drive, focusing instead on modifying the source code of a QEMU-emulated block device. To the best of our knowledge, such backdoors have never been documented or implemented before. In Section 4, we report on tests we perform on the implementation to evaluate its usability and feasibility as a backdoor and its ability to remain undetected. In Section 5, we generalize our approach by considering variations and extensions to the scenario and to the implementation we present, thus showing that DEBs should be considered in a wide range of scenarios. We discuss general prevention techniques against storage-device-based backdoors in Section 6. Most importantly, we identify encryption of data at rest, which currently is not widely used, to be a very effective prevention mechanism against DEBs. We discuss related work in Section 7, and conclude this paper in Section 8.

2 DEB Design

In this section, we present an informal overview of how a backdoor that allows the sending and receiving of commands and data between the attacker and a storage device, i.e., a DEB, is designed. Section 3 will present full details. Basically, a DEB has two components: (*i*) a modified firmware in the target storage device and (*ii*) a protocol to leverage the modified firmware and to establish a bi-directional communication channel between the attacker and the firmware.

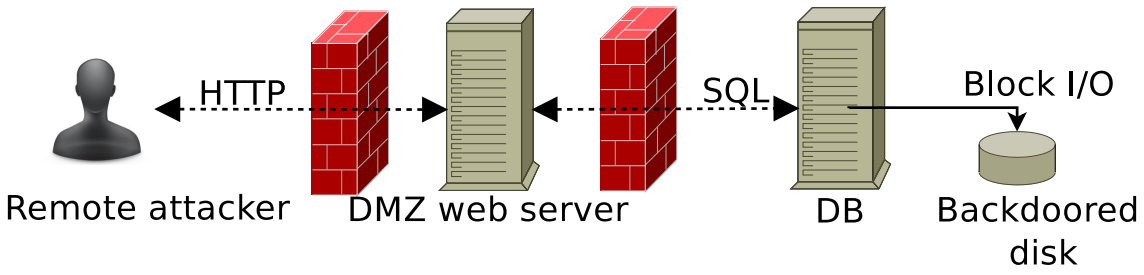

Figure 1: A server-side storage backdoor.

2.1 Firmware Design

The firmware of the storage device (also known as microcode) of a target machine is tampered with by the attacker. The main idea is that the DEB firmware intercepts and modifies *write* requests.

When a write request comprising an address (typically a sector number) X and payload buffer B is received, the firmware checks for a *magic value* in B.

If the magic value is present in B, it triggers the malicious behavior: the firmware extracts a command from the request data, such as "read data at sector Y" for data exfiltration from the storage device. The firmware reads data B' from sector Y and writes B' instead of B at sector X. At this point, a future read request at address X will return the modified content, thereby allowing unauthorized data exfiltration of the content at address Y from the device to a remote attacker.

We see in the next section how remote attackers can establish bi-directional communication channels with such backdoors in practice. We keep the example of the malicious data exfiltration "read data at sector Y" operation for the remainder of this paper because of its generality: it is an example in which the communication channel between the attacker and the storage device is bi-directional. A *data infiltration* operation, whereby the attacker remotely writes into the device, is easier to achieve because, theoretically, this operation would only require uni-directional communication from the attacker to the device.

Note that the firmware can write B' to X possibly after modifications through cryptographic and steganographic operations to prevent easy detection by the administrator of the target machine. We will discuss this extension later.

2.2 Communication Protocol

We now turn to a real-world example of a server-side DEB, where the compromised firmware runs behind a typical two-tier web server and database architecture, as shown in Figure 1. This scenario is of particular interest because the various protocols and applications between the attacker and the storage device can render the establishment of a (covert) communication channel extremely difficult. We assume that the web server provides a web service where users can write and read content, which is the case for most web services. The specific example we select here is that of a web forum or blog service where users can write and read comments.

To perform data exfiltration, the attacker proceeds in the following way:

The attacker performs an HTTP GET or POST request from his or her browser to submit a new comment to the forum of the web server. The comment contains the magic

value and a disguised "read sector X" command for the backdoor, as described above. The web server passes the comment data and other meta-data (e.g., username, timestamp) to the back-end database (e.g., through an SQL INSERT query). Using the filesystem and the operating system, the database then writes the data and meta-data to the compromised storage device as a sequence of write requests. As one of the write requests contains the magic value, some of the comment data is now replaced by the compromised firmware with the contents of sector X.

Finally, the attacker issues a GET request to simply read the exact forum comment he has just created. This causes an SQL SELECT query from the web application to the database and eventually a read request from the database to the compromised storage device. The content of the comment displayed to the attacker now contains data from sector X. The attacker has successfully exfiltrated data.

It should be stressed that the DEB allows the attacker to read arbitrary sectors and access the storage device as a (remote) block device. The attacker can thus mount filesystems and access files on this device selectively, i.e., without having to exfiltrate the storage device's contents fully.

2.3 Challenges

The various applications and protocol layers between the attacker and the backdoor increase the difficulty of establishing a communication channel. We now give an overview of these challenges.

Data encoding. The character encoding chosen by the web server or database may be different from the one the backdoor expects. The backdoor may try different character encodings on the content of incoming write requests, looking for the magic value in the data. By knowing the encoded magic value under different encodings, the backdoor can identify which encoding is being used, and encode the data to be exfiltrated such that it can be read by the application.

Magic value alignment. It is difficult to predict the alignment of the magic value at specific boundaries. This results in considerable overhead when searching for the magic value in a write buffer. Searching for a 4-byte magic value in a 512-byte sector, for instance, would require examining 509 byte sequences. As discussed later on, we mitigate the situation by *repeating* the magic value multiple times in a request, such that the overhead of searching for it becomes negligible.

Limited content size. Size restrictions when reading and writing content at the web service level, e.g., if the blog service only allows up to 140 characters long comments, may reduce the bandwidth of the covert channel.

Compression. At the filesystem or the application level, data compression techniques might be used. This renders the magic value undetectable, because its compression will depend on surrounding data. A DEB can deal with this by controlling (or predicting) all data to be compressed or a sufficiently long sequence around (before, after, or both, depending on the compression algorithm) the magic value. For instance, compression algorithms such as Lempel-Ziv [8] will create code words depending solely on a sliding window kept on the input. To replace the compressed buffer, the backdoor must resume the compression algorithm beginning from the position where it performed the first modification on the input.

Caching. Caching at any layer between the attacker and the storage device will cause delay potentially both in the reception of the malicious command and the reply from the device. The delay corresponds to the time taken to evict the malicious command from

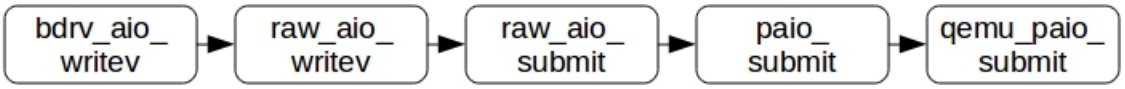

Figure 2: Call sequence of a QEMU write operation

caches above the storage device. Therefore, this delay can be reduced by putting additional load on the web server, to bring in cache more recent data.

While the above challenges are significant, because they render the exploitation of the backdoor more complicated and reduce the capacity of the communication channel between the attacker and the firmware, it is clear that they do not prevent the use of DEBs in the general case. In fact, the implementation in Section 3 copes with all the above challenges, except for compression and limited content size.

3 Implementation

In this section, we describe our proof-of-concept implementation of a DEB in QEMU [9], an open source dynamic translator which is also used by system-virtualization software such as KVM and Xen for device emulation. We modify the QEMU source code such that the QEMU-emulated IDE hard disks contain a DEB.

We now present the detailed algorithm used by the backdoor and the structure of the write buffer containing the magic value and command for an exfiltration attack, taking into account practical issues such as magic value alignment. For simplicity, we assume that the target host does not use compression for the forum application.

3.1 Write operations in QEMU

A write operation in a block device is essentially defined by a sector number to write to, the number of sectors, i.e., the length, and the buffer to be written. In QEMU hard disks, the write request sent by the OS block layer above is received in an I/O vector structure *iov*, containing a buffer and a size. QEMU aggregates multiple such write operations in a single write operation for the underlying storage layer, using a dynamically sized "QEMU I/O vector" structure *qiov*, which contains one or more *iov* structures. The size of a sector on a QEMU-emulated block device is 512 bytes. The size of an *iov* received by the block device is at least the size of a single sector, but is decided by the OS and is therefore a multiple of the filesystem block size for all operations issued by a filesystem.

Figure 2 shows the sequence of functions that are called during a write operation to a raw block device in QEMU. The backdoor is inserted into the *raw_aio_writev* function which receives a *qiov* as an argument and is called in all writes to raw files. Thus, all writes are intercepted by the backdoor function described below, which scans and possibly modifies the *qiov* before the write is submitted.

3.2 Firmware Backdoor

The backdoor function checks each *iov* buffer for a magic sequence of bytes *magic*, which is followed by a sequence of bytes *cmd* specifying the malicious command to be executed. As we focus on data exfiltration, in our case this merely contains a sector number encoded

in base64. More precisely, the attacker submits writes of length $2 \cdot bkdr\_bs$, formatted in the following way, with $\|$ being the concatenation operation:

$$\underbrace{magic\| \ldots \|magic}_{\text{repeated } count \text{ times}} \| \, cmd \, \| \, \underbrace{magic\| \ldots \|magic}_{\text{repeated } count \text{ times}} \| \, cmd$$
$$count = (bkdr\_bs - length(cmd))/length(magic)$$

Typically, there are layers between the attacker and the disk (e.g., the filesystem) that split all writes into blocks of at least *bkdr_bs* size at an arbitrary offset. Thus, the blocks created have at least one *bkdr_bs*-sized chunk exclusively containing the repeated magic sequences followed by the command (modulo a byte-level circular permutation on the chunk, i.e., a "wrap around"). This allows the backdoor (*i*) to make sure the *bkdr_bs*-sized chunk can be safely replaced by an equal-size exfiltrated data chunk, (*ii*) to check efficiently for the magic value. More precisely, the backdoor checks only the first $length(cmd) + length(magic)$ bytes of the chunk, because of the possible $length(magic)$ alignments of the magic value and the possibility of the chunk starting with *cmd*. Note that increasing the length of the magic increases the performance overhead of the backdoor.

Algorithm 1 $backdoor(qiov, magic, cmd\_size, bkdr\_bs)$

```
bkdr_count ← length(magic) + cmd_size
for iov in qiov do
  for each bkdr_bs-sized chunk chnk in iov do
    if magic present in first bkdr_count bytes of chnk then
      if chnk does not contain count successive magics then
        continue loop at next iteration
      end if
      cmd ← cmd_size bytes after last magic, wrap around if required
      sector_num ← base64_decode(cmd)
      buf ← read_sector(sector_num)
      base64_encode(buf)
      chnk ← buf
    end if
  end for
end for
```

In our implementation, we opt for a 4-byte magic value which makes for round computations while being small enough to not cause excessive performance overhead. The filesystem block size on the backdoored disk is 4 KB, thus, all *iovs* created are multiples of 4 KB in length. We can therefore set the value of *bkdr_bs* to be 4 KB as well. An attacker wanting to be independent of the filesystem block size would use 512-byte chunks, i.e., the smallest sector size on block devices. This would cause the backdoor to have a higher overhead on write operations.

As shown in Algorithm 1, the DEB proceeds by decoding the sector number from the command and reads that sector. The data returned by the read function is then encoded using base64, which increases its size by 1/3. To ensure that the encoded data fits into the 4 KB chunk, the backdoor performs read requests of size 3 KB, i.e., 6 consecutive sectors. The resulting 4 KB chunk replaces the original chunk in the *iov*.

Once all *iovs* in the *qiov* have been examined by the backdoor function, control is returned to the calling function *raw_aio_writev*, which proceeds to submit the write with

the modified *qiov*. A subsequent read request for the data submitted by the attacker will result in the extracted data being returned.

Note that valid magic sequences can occur during normal operation, i.e., non-malicious use of the storage device. This false-positive would result in the storage device detecting the magic sequence and writing faulty data to a sector – risking the stability of the system. However, such a false-positive occurs (not taking into account the possible range restrictions on the 8-byte *cmd* value, which can be checked for in Algorithm 1) with a negligible probability of $2^{-count \cdot bitlength(magic)}$, which equals 2^{-32704} for our values, assuming a 4 KB storage block where each bit is uniformly randomly chosen. In reality, the byte values on the underlying storage device are not uniformly distributed, e.g., alpha-numeric values are more likely, and the magic value also happens to be an alpha-numeric string in our case. Taking this into account and denoting by h the entropy measure function, we obtain a probability of $p = 2^{-count \cdot length(magic) \cdot h(alphanum\_byte)}$, or approximately 2^{-24528} with our parameters, given that the entropy of an alpha-numeric byte is about 6 bits. Even on a 1 PB storage device, the latter event still occurs with negligible probability (it occurs at least once with a probability of $1 - (1-p)^{2^{38}} \approx 2^{38}p \approx 2^{-24490}$).

In contrast, under malicious inputs, it is possible that attacker-supplied data is written in a block next to a few non-attacker-supplied bytes which happen to correspond to the format expected by the backdoor, resulting also in false-positives. For instance, the attacker's *bkdr_bs*-sized first block could be written starting at a 8-byte offset inside a chunk. These first 8 bytes would correspond to non-attacker-supplied bytes, which the backdoor should not replace. It is possible that these 8 bytes happen to be a valid sector number, which would result in the entire chunk (including these first 8 bytes) being replaced. Although we did not observe this case in our tests, the backdoor can be written to ignore requests which result in two blocks being simultaneously exfiltrated, to further prevent non-attacker-supplied bytes to be overwritten. The probability of occurence of this event also highly depends on the offset at which chunks are written. This offset is not uniformly distributed, and depends on the higher-level application writing to the device. The attacker could therefore also leverage his knowledge of the application to elimitate this stability risk.

4 Evaluation

In this section, we report on the tests that we carried out to evaluate whether the attacker can expect a data exfiltration bandwidth high enough for practical use (taking into account potential caching issues), and whether the backdoor affects the disk's common-case performance (which would render the backdoor less stealthy). We base this evaluation on the scenario described in Section 2.2.

4.1 Experimental Setup

We conduct our experiments on a virtual machine with 1 GB of memory running on a QEMU code base containing the backdoor. This is the attacker's target host. The physical machine running QEMU is a system equipped with an Intel Core 2 Duo 2.10 GHz processor, 3 GB of physical memory, and a Seagate ST9320325AS SATA disk. Our tests are performed on the emulated IDE disk with writeback caching. On the target host, we install Ubuntu 9.04 and set up an Apache web server with two simple PHP scripts emulating a web forum (or blog) functionality. The forum shows all (recently) made comments (or "posts") using the first PHP script, and also allows the submission of new

comments, using the second script. These comments are written to and read from a table in an installed MySQL 5.0.75 database, which runs atop an ext3 filesystem.

4.2 Tests

We perform the following three tests to evaluate the performance of the backdoor.

Insert and latency test. We measure the time needed to insert a series of malicious comments into the database via the PHP form. The time taken to use the PHP form to submit the commands, measured at the client side, is the *insert* time.

Now, because of caching, the inserted comments are not immediately updated with the exfiltrated data: the malicious comments need to be evicted from the cache. Therefore, after inserting the sequence of malicious commands, we additionally insert non-malicious ("dummy") comments — thereby simulating the use of the website by other non-malicious users, for example — and record the time taken to insert these comments until the exfiltrated data eventually shows up. The time taken until the exfiltrated data is retrieved by the client is the *latency* time. Note that this also includes the time to download the extracted data.

So, the total time that passes for the attacker starting from submitting his or her commands until he or she can see the extracted data is the sum of *insert* and *latency*.

The insert and latency tests are performed using *wget* to submit comments and to check the contents of the PHP forum.

Overhead test. To put the overhead imposed by the backdoor on the target machine into perspective, we compare it with the disk performance during normal (i.e., non-malicious) write operations.

We use IOZone [10] to measure the throughput of writes on the target machine. As the backdoor functionality is only activated during writes, we use the IOZone *write-rewrite* test, and compare the write throughputs obtained on guests running on the unmodified QEMU code base and the backdoored QEMU code base. We additionally perform the test with the IOZone O_DIRECT option set to compare the results when the filesystem buffer cache is bypassed. Although most applications will use the filesystem buffer cache when accessing the disk, this test verifies that the backdoor's overhead is not simply hidden because of caching.

File exfiltration test. To further evaluate the usability of DEBs, we measure the time to exfiltrate the `/etc/passwd` file on the target host. We do so by writing a python script that successively (a) retrieves the partition table in the MBR of the disk, (b) retrieves the superblock of the ext3 partition, (c) retrieves the first block group descriptor, (d) retrieves the inode contents of the root directory `/` (always at inode number 2) in the inode table and (e) retrieves the block corresponding to the root directory, therefore finding the inode number of `/etc`. By repeating the last two steps for `/etc`, the attacker retrieves the `/etc/passwd` file on the target host.

4.3 Results

We perform 30 iterations for all three tests, with a 30 s pause between successive iterations. For each set of values measured, we compute 95%-confidence intervals using the t-distribution.

Row 1 in Table 1 shows the time taken to insert 500 8-KB comments into the table, using the PHP form. As described in Section 3, the backdoor replaces each of these comments with 3 KB of exfiltrated data starting at the sector number included in the comment. Row 2 in Table 1 shows the update latency in seconds for the 500 comments

Table 1: Time to exfiltrate 3000 disk sectors

	Mean (s)	95% CI
Insert	10.7	[10.65; 10.71]
Latency	9.7	[9.55; 9.82]

Table 2: Filesystem-level write-throughput overhead of the backdoor

	Without Buffer Cache		With Buffer Cache	
	Mean (MB/s)	95% CI	Mean (MB/s)	95% CI
With Backdoor	5.18	[5.16; 5.20]	15.61	[15.25; 15.96]
Without Backdoor	5.27	[5.23; 5.32]	16.20	[16.07; 16.34]

inserted during the insert test. It follows that an attacker is able to exfiltrate 3000 sectors in $10.7 + 9.7 = 20.4$ s on our setup, achieving a read bandwidth of 74 KB/s. In practice, an attacker may not want such a high bandwidth in order to remain more stealthy. Additionally, those values will differ depending on the characteristics of the system (e.g., more physical memory will cause the comments to persist longer in cache, although more load on the server will cause the opposite). However, these results show that the bandwidth is likely to be sufficiently high, and the attacker can realistically use steganographic transformations for instance (see next section), which will further reduce the data exfiltration bandwidth, and increase stealthiness.

Table 2 shows the comparison of the write throughputs achieved on guest virtual machines running on both an unmodified and backdoored QEMU code base. In both cases, we executed the IOZone write/rewrite test to create a 100 MB file with a record length of 4 KB.

Comparing the mean values and taking the 95%-confidence intervals into account, we conclude that the backdoor does not add significant overhead to write operations, both with direct I/O and the buffer cache. The same write throughput test performed directly on the host's disk yields a comparable direct I/O throughput of 15 MB/s. Assuming that the few comparisons for magic values on each chunk will cause the same latency, we can expect similar overhead results on a physical disk firmware implementation of the backdoor.

Table 3 shows that `/etc/passwd` can be exfiltrated in a very reasonable amount of time. Because the process of retrieving the file requires 9 queries for a few sectors, each of them depending on the results returned by the preceding query, this figure is mainly dominated by the time taken to evict comments from the cache. This means that the actual latency for a single sector is around 4 s (for a comparison, note that the latency figure in Table 1 also includes the retrieval time of the 3000 sectors).

Table 3: Time to exfiltrate `/etc/passwd`

	Mean (s)	95% CI
File exfiltration	40.0	[39.6; 40.4]

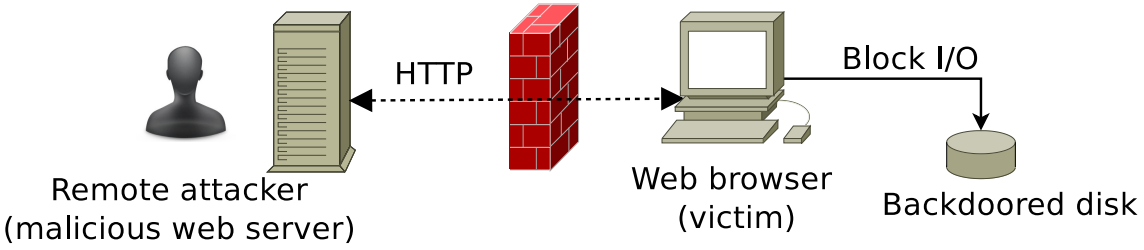

Figure 3: A client-side storage backdoor.

5 Variations and Extensions

The preceding sections study the feasibility of DEBs in a very specific, but quite typical, scenario. The purpose of this section is to widen our perspective by considering a wider range of use cases and improvements to DEBs.

5.1 A client-side remote storage backdoor

In general, a DEB can be adapted to be used in different settings. Often, the adaptation is not straightforward, and may require innovative uses of application-level features. We present our thought process on how the attacker can leverage bi-directional communication with a storage backdoor in a client-side storage backdoor setting, as depicted in Figure 3.

Establishing a Uni-directional Channel. Assume that the user of a target computer is lured into visiting an attacker-controlled website.

Now, the attacker needs to provoke a write to the compromised storage device with controlled content. We remark that web browsers typically write webserver-provided content to persistent storage, for instance (a) when using cookies (browser cookies, but also cookies associated with plugins such as Flash local shared objects), (b) when caching previously visited web pages for performance (browser cache) by storing them as files on the filesystem, (c) when using the HTML5 local storage feature, or (d) when downloading content with user interaction. All four solutions allow an attacker to easily establish a uni-directional channel to the compromised storage device.

Upgrading to a Bi-directional Channel. The attacker will furthermore try to establish a bi-directional channel. Requiring user interaction (file download) is not realistic and does not allow the establishment of a bi-directional channel unless the user uploads the file back. The HTML5 local storage feature provides an easy way to establish a bi-directional channel (by design, it allows web-server-controlled read and writes to the disk), but is not yet widely implemented. We focus therefore on the first two options for writing to the storage device.

HTTP cookies for a given domain will be included in all requests by the web browser according to the same origin policy. A back-channel can therefore be established by the compromised web server by simply forcing a page reload after the cookie has been accepted by the victim's browser. However, browser cookies are limited in size (up to 4 KB), and sometimes disabled — especially on untrusted web sites — for privacy reasons. For this reason, we also explore the last option: using the browser cache. The remaining challenge here is to send the exfiltrated data, which will be in the cached version of the web page, back to the web server. This can typically be done by using JavaScript to send content asynchronously with *XmlHttpRequest* or synchronously with an HTTP GET or

POST request; the web page the victim initially received can contain, for example, an *XmlHttpRequest* command to send back a request to the web server containing the magic value and the command. The contents of the magic value and command will be modified by the backdoor once the browser caches the content to the disk. As soon as the page has been reloaded from the cache, the modified web page will send the data to be exfiltrated. Note that the fact that the page has changed on the client's side will not cause the web browser to issue a new request to the web server for new content, because the *Etag* value associated with the page — which is sent by the client to the web server when repeating a previous request to validate the cached response — has not changed.

Discussion. Nowadays, visiting an untrusted web site can be considered a significant threat by itself, e.g., because of drive-by-download vulnerabilities [11]. However, in the example of the browser cache-based storage backdoor, the attacker does not necessarily need to lure the target into visiting an untrusted web page. To write to the disk, the attacker solely needs to modify content on a web site often visited by the victim that does not disable browser caching. The attacker can easily achieve this, for example, by sending an email to a web-based email service, posting comments on a blog, sharing information on a social network, or even by posting ads to communicate with a large number of backdoored storage devices simultaneously. For the back-channel, the attacker would, however, need to modify the backdoor to include JavaScript code that would post the exfiltrated data when replacing the magic value and command, either through a messaging functionality of the website or directly by sending a request to a web server under his or her control. However, it can be argued that this renders the backdoor less stealthy to some extent, as the control flow of the browser is now modified by the backdoor because of the addition of the JavaScript.

5.2 Variations

Other storage devices. So far, we used the term *storage device* to refer to persistent storage devices, such as hard disks and solid-state disks. However, DEBs can also be implemented in volatile memory, such as RAM, CPU registers or even buffers inside network interface controllers. One significant difference is that these storage devices (perhaps with the exception of network controllers) must operate at very low latencies. Continuously checking for the presence of magic values is therefore not realistic, considering the performance and cost requirements of those devices. Similarly, storage devices that are mainly used for archival purposes, such as tapes or optical media, will have very high data exfiltration latencies for the attacker. Note that such storage backdoors can also be included at higher levels in the storage hierarchy, such as databases, file systems, storage-area-network or network-attached-storage devices, although this renders the backdoor less stealthy and more prone to removal. An administrator detecting a breach will proceed to perform a clean re-installation of all machines, but is unlikely to do so for the firmware of hard disks.

RAID. Systems using RAID-configured disk arrays [12] can be interesting for the attacker in the scenario where attached disks are compromised but not the RAID controller. Clearly, if the firmware of the RAID controller has been backdoored, the attacks described herein are still relevant, independently of the disks being compromised. The goal of RAID schemes is to allow the array to tolerate faults on the disks. However, the RAID controller typically only guards the system against non-malicious faults, such as hardware component failures; thus, it comes as no surprise that malicious attacks can succeed even when the controller firmware is intact, as long as some of the disks in the array have been compromised. For instance, in schemes with no redundancy and no parity (such as RAID0), the

attack will succeed when the target data reside on a backdoored disk. For schemes that use redundancy (mirroring), such as RAID1, the attack will succeed when all disks have been backdoored; otherwise, only reads served from the backdoored mirrors will return exfiltrated data. For RAID schemes that use parity, such as RAID5/6, the attack will succeed when all disks in the array have been compromised. If the attacker knows the RAID segment size for each one of the disks, then he or she can repeat the magic value in each RAID segment, with the disguised request's target LBA translated from the adapter address space to the appropriate hard-disk LBA address for each compromised hard disks. Thereby, a whole stripe can be exfiltrated with a single request. However, even when the RAID segment size is unknown, the attacker can still exfiltrate data from compromised disks, albeit without knowing to which RAID segment a chunk of data belongs. This would make it harder, but not impossible, for the attacker to reconstruct the entire RAID stripe.

Steganography. To prevent the exfiltrated data from attracting the attention of administrators or other users, simple steganographic transformations can be applied to conceal the exfiltrated data. In the scenario presented in Section 2.2, the attacker tries to hide exfiltrated data from a administrator monitoring forum behavior. For example, administrators or "legitimate" users reading the blog comments might see an excessive amount of base64-encoded messages appearing in the forum. To conceal and cover exfiltrated data, the attacker might use text or linguistic steganography, both of which have been studied extensively [13–15]. For example, with text steganography, the backdoor can use white spaces in the comments to encode information. Clearly, the use of steganography further reduces the exfiltration bandwidth of the attacker. We note that the attacker can also leverage formats other than text for which steganography exists, such as pictures and videos, to perform the same exfiltration attack. Moreover, the need of using steganography does not necessarily exist in scenarios where the data is meant to be private to the user, for example a web-based email service.

Other channels. In this paper, we make use of storage channels as opposed to, for example, timing channels, which can be a stealthier, although harder to realize, alternative to establish a communication channel (see for example the creative use of timing channels in [16]). However, we believe that the extensive number of layers (applications, caches) between the attacker and the storage device will create jitter that would render such attacks impractical (very low channel capacity). Other covert channels could be envisioned: for example, the alternation of read and write requests can be used to convey information; although this channel would also be highly dependent on concurrent requests to the devices as well as requiring a priori knowledge, by the attacker, of the various components using the storage device. We leave the evaluation of the practical potency of such channels to future work.

6 Prevention

6.1 Encryption of data at rest

We believe encryption of data at rest is too rarely used nowadays, both by enterprises and end users. When used, it is for the purpose of regulatory compliance or providing easy storage-device disposal and theft protection (by securely deleting the encryption key associated with a lost disk). We show here that encryption of data at rest also prevents data-exfiltration backdoors on storage devices, for two reasons: it eliminates the covert storage channel for remote attackers and prevents the untrusted storage device from

accessing the data in the first place.

Integrity. The data exfiltration process in Section 2 requires modification of the data provided by the write requester. Cryptographic integrity checks would detect and prevent this specific attack, provided that they are performed above the backdoored storage device and that the cryptographic materials (e.g., the hash value if a cryptographic hash used, and authentication key if a message authentication code is employed) are not stored on the compromised device. This also prevents *data infiltration*, i.e., a malicious write operation from the backdoor on the disk. However, the uni-directional channel still remains operational, and the compromised storage device could use another covert channel such as a timing channel to return the data to the attacker: data integrity alone may therefore be insufficient to deal with DEBs.

Confidentiality. To prevent the data stored on the storage device from being read and subsequently forwarded to an external attacker, the data can simply be encrypted. This prevents data exfiltration completely, but we also need integrity to guarantee that the backdoor does not modify the content of the disk (e.g., flip bits in a block, or restore a previous version of a block).

Key management. It may seem that the use of cryptography will only shift the problem because cryptographic material needs to be stored on a (small) trusted storage device, which in turn needs to be protected from DEBs. However, dedicated key management servers [17, 18] can be used to prevent such attacks. As such servers are not meant to store attacker supplied input, DEBs cannot be used. Additionally, as a general principle, it is easier to secure smaller systems with dedicated uses (principle of least privilege and economy of mechanism).

6.2 Signed firmware updates

A straightforward way to protect a device from malicious firmware updates is to require all firmware images to be cryptographically signed. The use of asymmetric signatures is preferable in this case: each device would be manufactured with the public key of the entity performing the firmware updates. Although signed firmware is a widely known approach, we have not been able to assess how widespread its use is for hard-disks and storage devices in general. We did find evidence that some RAID controllers (e.g., [19]) and USB flash storage sticks (e.g., [20]) have digitally signed firmware but these appear to be a minority.

Note that this does not prevent an attacker with physical access to the device from replacing it with an apparently similar, but in reality backdoored, device. Furthermore, signing does not prevent attacks from a malicious or compromised manufacturer, although it can help trace the origin of a malicious firmware (e.g., if each manufacturing plant has a distinct key).

6.3 Intrusion detection systems

Current network-based intrusion detection systems and anti-virus systems use, to a large extent, simple pattern-matching to detect known malicious content. Clearly, the DEBs presented in this paper could be detected by such tools if the magic value is publicly known. This could be the case, for example, if the attacker targeted a large number of machines with the same magic value. However, these detection systems are clearly inadequate for targeted or more sophisticated attacks, e.g., an attacker could change the magic value for each different machine he targets, and make the magic values a time dependent function to evade detection.

7 Related work

Backdoors have a long history of creative implementations: In [21] Thompson describes how to write a compiler backdoor that would compile backdoors into other programs, such as the login program, and persist when compiling future compilers.

Many papers describe the design and implementation of hardware backdoors. In [22], the authors present the design and implementation of a malicious processor with a circuit-level backdoor allowing, for example, a local attacker to bypass MMU memory protection. Heasman presents implementations of PCI and ACPI backdoors in [23, 24] that insert rootkits into the kernel at boot time. However, we note that, contrary to [7], where Triulzi presents a NIC backdoor that provides a shell running on the GPU (which is achieved via PCI-to-PCI transfers between the two devices), and contrary to our approach, those backdoors are only *bootstrapped* from hardware devices: their target is to compromise the host machine's kernel. Therefore, they can be detected and prevented by any kernel integrity protection mechanism, such as Copilot [6], which is implemented as a PCI device.

Concerning counter-measures, software-based attestation may also be used to detect firmware tampering [25, 26]. This is achieved by implementing a self-integrity checking mechanism in the firmware of the target device, which acts on a nonce sent by the OS (for replay protection). To prevent the device from manipulating results, the delay of this computation is also measured, as well as other measures. Although such techniques can be used to detect firmware DEBs, they cannot be used for DEBs implemented directly in hardware (which is out of the scope of software-based attestation, but is in the scope of this paper).

Other examples of data-exfiltration attacks involving NICs include [27], where the authors use IOAPIC redirection to an unused IDT entry that they modify to perform data exfiltration. More generally, remote-DMA-capable NICs (such as InfiniBand and iWARP) can be used to perform data exfiltration [28]. However, such traffic can be equally easily identified and blocked by a firewall at the network boundary. Thus, a covert channel is needed to communicate with the backdoor, as mentioned in [29] for ICMP echo packets (independently of any hardware backdoor). In comparison, our approach leverages an existing channel on the backdoored system (e.g., HTTP) and therefore cannot be distinguished from legitimate traffic at the network level.

8 Conclusion

This paper studied a new kind of firmware backdoor for storage devices and showed it to be a realistic and powerful threat. An attacker may introduce such a backdoor by tampering with the hard-disk firmware of a storage supplier of entities, such as cloud providers, banks, or even end users. The attacker may then exfiltrate data from the storage device remotely through covert storage channels that bypass network-based or host-based intrusion detection systems. To evaluate its real-world potential, we implemented such a backdoor prototype, and showed how some of the practical difficulties identified can be overcome, e.g., by ensuring that the backdoor does not cause noticeable performance overhead to the victim. Based on our evaluations that reveal the ease of such attacks for a determined attacker, we conclude that this attack vector must not be overlooked when assessing the security of future systems. To mitigate the threat, we recommend encrypting data at rest to prevent such attacks and to reduce the overall trusted computing base.

References

[1] J. Halderman and E. Felten, "Lessons from the Sony CD DRM episode," in *Proceedings of the 15th USENIX Security Symposium*, pp. 77–92, 2006.

[2] F. Baker, B. Foster, and C. Sharp, "Cisco Architecture for Lawful Intercept in IP Networks." RFC 3924 (Informational), Oct. 2004.

[3] T. Cross, "Exploiting Lawful Intercept to Wiretap the Internet." `http://www.blackhat.com/presentations/bh-dc-10/Cross_Tom/BlackHat-DC-2010-Cross-Attacking-LawfulI-Intercept-wp.pdf`, 2010. Black Hat DC.

[4] "Maxtor Basics Personal Storage 3200 virus." `http://seagate.custkb.com/seagate/crm/selfservice/search.jsp?DocId=205131&NewLang=en`.

[5] J. Larimer, "Beyond Autorun: Exploiting vulnerabilities with removable storage." `https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf`, 2011. Black Hat DC.

[6] N. Petroni Jr, T. Fraser, J. Molina, and W. Arbaugh, "Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proceedings of the 13th USENIX Security Symposium*, pp. 13–13, 2004.

[7] A. Triulzi, "Project Maux Mk.II, I own the NIC, now I want a Shell!," in *the 8th annual PacSec conference*, 2008.

[8] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977. ISSN 0018-9448.

[9] "QEMU." `http://wiki.qemu.org/Main_Page/`.

[10] "IOZone." `http://www.iozone.org/`.

[11] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code," in *Proceedings of the 19th International Conference on World Wide Web*, pp. 281–290, 2010. ISBN 978-1-60558-799-8.

[12] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pp. 109–116, 1988.

[13] L. Y. Por, T. F. Ang, and B. Delina, "WhiteSteg: A new scheme in information hiding using text steganography," *WSEAS Transactions on Computers*, vol. 7, pp. 735–745, June 2008. ISSN 1109-2750.

[14] R. Bergmair and S. Katzenbeisser, "Content-Aware Steganography: About Lazy Prisoners and Narrow-Minded Wardens," in *Proceedings of the 8th International Conference on Information Hiding*, pp. 109–123, 2007. ISBN 978-3-540-74123-7.

[15] K. Bennett, "Linguistic Steganography: Survey, Analysis, and Robustness Concerns for Hiding Information in Text," 2004. Cerias Technical Report, Purdue University, `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.158.8602&rep=rep1&type=pdf`.

[16] G. Shah, A. Molina, and M. Blaze, "Keyboards and Covert Channels," in *Proceedings of the 15th USENIX Security Symposium*, pp. 59–75, 2006.

[17] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for Key Management," NIST special publication 800-57, National Institute of Standards and Technology (NIST), 2007. `http://csrc.nist.gov/publications/PubsSPs.html`.

[18] M. Björkqvist, C. Cachin, R. Haas, X. Hu, A. Kurmus, R. Pawlitzek, and M. Vukolić, "Design and implementation of a key-lifecycle management system," *Financial Cryptography and Data Security*, pp. 160–174, 2010. ISBN 978-3-642-14576-6.

[19] "Configuring the RSA II adapter." `http://www.scribd.com/doc/3507950/RSAII-Card-Installation`.

[20] "Kingston DataTraveler Vault." `http://www.cheapmemory.com.au/Cheap-USB-Business-Flash-Drive-C41/2GB-Ultra-Secure-USB-FIPS-140-2-CMDT5000/2GB.html`.

[21] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984. ISSN 0001-0782.

[22] S. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and Implementing Malicious Hardware," in *Proceedings of the 1st USENIX Workshop on Large-scale Exploits and Emergent Threats*, pp. 1–8, 2008.

[23] J. Heasman, "Implementing and Detecting an ACPI BIOS Rootkit." Black Hat 2006.

[24] J. Heasman, "Implementing and Detecting a PCI Rootkit," `http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf`.

[25] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[26] Y. Li, J. McCune, and A. Perrig, "VIPER: Verifying the Integrity of PERipherals' Firmware," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2011. To appear.

[27] S. Sparks, S. Embleton, and C. C. Zou, "A Chipset Level Network Backdoor: Bypassing Host-Based Firewall & IDS," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pp. 125–134, 2009. ISBN 978-1-60558-394-5.

[28] F. Sultan and A. Bohra, "Nonintrusive Remote Healing Using Backdoors," in *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.130.4429&rep=rep1&type=pdf`.

[29] daemon9, "Project Loki." Phrack 49, `http://www.phrack.org/issues.html?id=6&issue=49`, 1996.