Implementation and Implications of a Stealth Hard-Drive Backdoor

Jonas Zaddach[†*] Anil Kurmus[‡*] Davide Balzarotti[†] Erik-Oliver Blass[§] Aurélien Francillon[†]
Travis Goodspeed[¶] Moitrayee Gupta[‖] Ioannis Koltsidas[‡]

ABSTRACT

Modern workstations and servers implicitly trust hard disks to act as well-behaved block devices. This paper analyzes the catastrophic loss of security that occurs when hard disks are not trustworthy. First, we show that it is possible to compromise the firmware of a commercial off-the-shelf hard drive, by resorting only to public information and reverse engineering. Using such a compromised firmware, we present a stealth rootkit that replaces arbitrary blocks from the disk while they are written, providing a *data replacement backdoor*. The measured performance overhead of the compromised disk drive is less than 1% compared with a normal, non-malicious disk drive. We then demonstrate that a remote attacker can even establish a communication channel with a compromised disk to infiltrate commands and to exfiltrate data. In our example, this channel is established over the Internet to an unmodified web server that relies on the compromised drive for its storage, passing through the original webserver, database server, database storage engine, filesystem driver, and block device driver. Additional experiments, performed in an emulated disk-drive environment, could automatically extract sensitive data such as `/etc/shadow` (or a secret key file) in less than a minute. This paper claims that the difficulty of implementing such an attack is not limited to the area of government cyber-warfare; rather, it is well within the reach of moderately funded criminals, botnet herders and academic researchers.

[*]Both authors are first authors.
[†]EURECOM, 06560 Sophia Antipolis, France. Email: {jonas.zaddach,davide.balzarotti,aurelien.francillon}@eurecom.fr.
[‡]IBM Research – Zurich, 8803 Rüschlikon, Switzerland. Email: {kur,iko}@zurich.ibm.com.
[§]College of Computer and Information Science, Northeastern University, Boston, MA, USA. Email: blass@ccs.neu.edu.
[¶]travis@radiantmachines.com
[‖]Department of Computer Science and Engineering, UCSD, La Jolla, CA, USA. Email: m5gupta@cs.ucsd.edu.

1. INTRODUCTION

Rootkits and backdoors are popular examples of malicious code that allow attackers to maintain control over compromised machines. They are used by simple botnets as well as by sophisticated targeted attacks, and they are often part of cyber-espionage tools designed to remain undetected and collect information for a long period of time.

Traditionally, malicious code targets system utilities, popular network services or components of the operating system. However, in a continuous effort to become more persistent and avoid detection, the target of the infection has shifted from software components towards more low-level elements, such as bootloaders, virtual-machine hypervisors, computer BIOS, and recently even the hardware itself.

The typical hardware-based threat scenario involves a malevolent employee in the manufacturing process or a compromised supply chain. In addition, many devices from trusted parties have been known to contain rootkits for copyright protection [14] or lawful interception capabilities in network devices [5, 10]. Recent reports of hard drives shipping with viruses [23] show that such threats are also realistic in the context of storage devices. In this paper, we will demonstrate that it is not even necessary to have access to the manufacturer or to the supply chain in order to compromise a hard drive's firmware. Instead, a firmware backdoor can be installed by, e.g., traditional malware after the operating system has been compromised.

From the attacker's point of view, a drawback of hardware backdoors is the fact that they are highly hardware dependent, requiring customization for each targeted device. This has made hardware backdoors less generic and less attractive than more traditional operating-system backdoors. However, the hard-drive market has now shrunk to only three major manufacturers, with Seagate and Western Digital accounting for almost 90% of all drives manufactured [4]. While drive firmwares may vary across product lines, porting a backdoor from one model to another of the same manufacturer should require only a limited amount of work, making backdoors on hard drives an attractive attack vector.

So far, malicious hardware has typically been used as a stepping stone to compromise other system components: for example, by exploiting the auto-run functionality, filesystem vulnerabilities [21], or DMA capabilities on systems lacking properly configured I/O Memory Management Units (IOMMU). In such cases, malicious code on the operating system is simply *bootstrapped* from the hardware device. Then, to perform its operation, the malware propagates and

infects the OS kernel, using the compromised hardware only as a way to survive re-installation and software updates. However, as soon as malicious code "leaves" the firmware and moves to the system memory, it breaks cover. Therefore, such malware can be detected and prevented by kernel- or hardware-supported integrity mechanisms, such as Copilot [25].

In this paper, we describe how an attacker can overcome the above limitations by leveraging a storage firmware backdoor. Such a firmware backdoor does not require any modification to the operating system. The backdoor is, therefore, less intrusive and less dependent on other layers (e.g., OS, applications, and filesystem). As a consequence, it cannot be detected by existing mechanisms that guarantee OS integrity [17, 25].

As a proof of concept, we present a Data Exfiltration Backdoor ("DEB") that allows an attacker to remotely retrieve and modify any data stored in the device. A DEB allows a bi-directional communication channel to be established between the attacker and the storage device that potentially resides in a data center well outside the attacker's reach. As most Internet-based services, such as web forums, blogs, cloud services or Internet banking, eventually need to read and write data to disk, a DEB can be used to remotely exfiltrate data from such services. The rationale of this *data-replacement backdoor* is that the attacker can piggy-back its communications with the infected storage device on disk reads and writes. Indeed, the attacker can issue a specific command by encapsulating it in normal data which is to be written to a block on a compromised hard drive. This command makes the malicious firmware replace the data to be written with the data of an *arbitrary* block specified by the attacker. In a second step, the attacker can then request the block that was just written and therewith, effectively, retrieve the content of any block on the hard-drive. We also discuss a number of challenges that arise with this technique, and show how the attacker can overcome them (e.g., data alignment and cache issues).

Threat Model.

In our threat model, an attacker has compromised an off-the-shelf computer. This machine may have been initially infected with a malware by a common attack such as a drive-by-download or a malicious email attachment. Then the malware infects the machine's hard drive firmware by abusing its firmware update mechanisms. Finally, the OS part of the malware removes itself from the machine, and future malicious behavior becomes completely "invisible" to the OS, anti-virus or forensics tools. Following such an infection, the malware can keep control of the machine without being detected even if the drive is formatted and the system re-installed.

We show in this paper that, surprisingly, the above attack requires the same amount of effort and expertise as the development of many existing forms of professional malware (e.g., large scale botnets). Moreover, we claim that this attack is well within the capabilities of current cyber-espionage tools. Finally, we note that this threat model applies to dedicated hosting providers, since an attacker could temporarily lease a dedicated server and infect an attached hard drive via a malicious firmware update. A subsequent customer leasing a server with this infected drive would then be a victim of this attack.

Contributions.

We make the following major contributions:

- We report on our reverse-engineering of a real-world, off-the-shelf hard-disk drive, its code update mechanism, and how one could infect its firmware with a backdoor that can (generally) modify blocks written to disk. We measured our backdoor's worst-case performance impact to be less than 1% on disk operations (Section 2).
- We present the design of a novel exfiltration mechanism, a *data replacement backdoor*, allowing a remote attacker to establish a covert data channel with the storage device (Section 3).
- We evaluate the impact of our compromised drive in a realistic attack scenario involving the communications between the attacker and the disk drive storing the database of a typical Linux/Apache/PHP web forum. As our prototype modified disk firmware was not stable enough for such complete experiments, we have evaluated this scenario in a QEMU simulation (Section 3.4).
- We discuss possible countermeasures and defense strategies against our attack, such as encrypting data at rest. We also discuss explorative defense techniques, e.g., a page-cache-based probabilistic detection mechanism (Section 4).

2. BACKDOORING A COMMERCIAL OFF-THE-SHELF HARD DRIVE

In this section we describe how we inserted a backdoor into the firmware of a stock hard drive.

2.1 Modern Hard-Drive Architecture

The software and system architecture described here are specific to the drive we analyzed. However, we observed that it is almost identical for two distinct drives from one product family of the same manufacturer, and a brief look at one drive from another major manufacturer revealed a very similar architecture.

Physical Device.

A hard disk is a set of rigid magnetic disks aligned on a spindle, which is rotated by a motor. A rotary actuator structure moves a stack of heads relative to concentric tracks on the surface of the disks. The entire apparatus is contained in a tightly sealed case. A micro-controller takes care of steering the motors and translating the higher-level protocol that a computer uses to communicate with the disk to and from a bitstream, which is processed by specialized hardware (a DSP or FPGA) and fed to the heads [8]. Today, hard disks interface with other systems mostly through Serial ATA (SATA) and Small Computer Systems Interface (SCSI) buses, although bridge chips might translate to other buses, such as USB. Parts of those protocols are typically handled directly in hardware.

Execution Environment.

Like many embedded systems, this hard drive is based on a custom System on Chip (SoC) design. This SoC is built

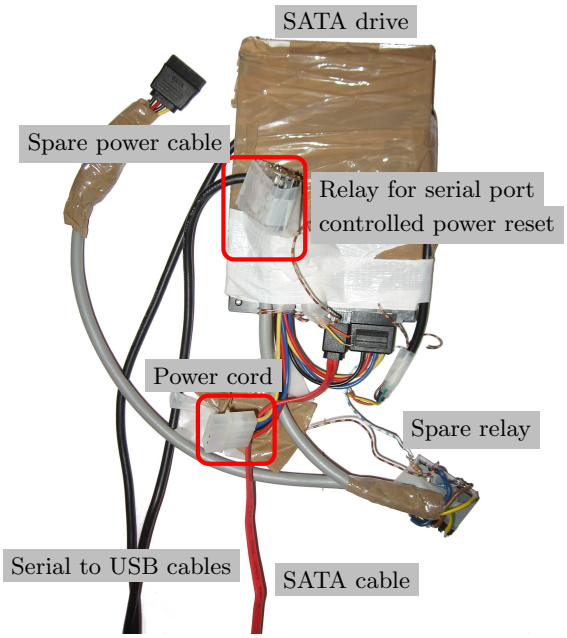

Figure 1: Custom backdoor development kit. This apparatus was built to reset the drive, allowing easy scripting and automated tasks. One USB to serial cable controls the relay, the second is connected to the serial port of the drive. The SATA cable is connected through a USB-SATA adapter for backdoor development. It is then directly connected to a computer motherboard for the field tests.

around an ARM966 CPU core, a read-only memory (ROM) containing a "mask ROM" bootloader, internal SRAM memories, an external serial FLASH (accessed via an SPI bus), and an external DRAM memory. This DRAM is the largest memory and is used to cache data blocks read from or written to disk as well as a part of the firmware code that does not fit into the SRAM.

Interestingly, this hard drive also provides a serial console accessible through a physical serial port on the drive's Master/Slave jumper pins.

Software Architecture and Boot Sequence.

The bootloader in mask ROM is executed immediately after the CPU resets and loads a reduced boot firmware from the serial FLASH chip. The boot firmware has the capability to initialize the hardware to a sufficient degree to load the main firmware from the magnetic disks. However, it does not implement the full SATA protocol that this hard drive uses to talk to the computer.

Finally, the main firmware is loaded into memory from a reserved area of the disk (not user accessible) and then executed. Additional *overlays*, providing non-default functionality, can be loaded on demand from the reserved area. For example, a diagnostic menu available through the serial console is in overlays "4" and "5". The memory layout at run-time is depicted in Figure 2.

As our hard drive has a SATA bus, read and write requests

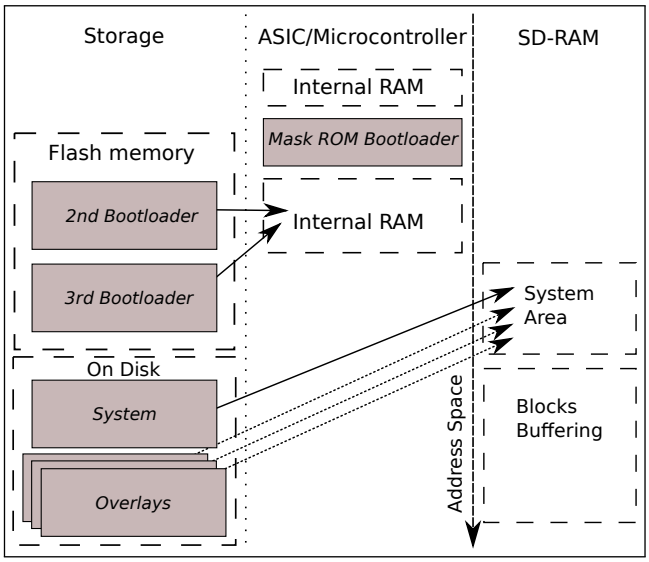

Figure 2: Overview of a hard drive's architecture.

to it are encapsulated in the ATA protocol. This is a simple master-slave protocol where the computer will always send a request, to which the hard drive replies with a response.

Inside the hard drive's firmware, five components take care of processing data: the interrupt handlers process hardware events, the SATA task processes data from the SATA port, the cache task manages the cache memory and evicts blocks from the cache, the read-write task transfers data to and from the disk platters, and the management task handles diagnostic menu commands and background activities.

Analysis Techniques.

Knowledge about the system was acquired from publicly available information (e.g., [1]) and by reverse-engineering a hard drive in our lab. While the firmware (except for the mask ROM bootloader) is contained in update files, their format is not obvious and the header format was not documented. Fortunately, the diagnostic menu allows parts of memory to be dumped while the system is loaded. Thus, it proved easier to dump the firmware of the running hard drive through this menu than recovering it from the firmware update binary.

The mask ROM bootloader contains another menu, which can be accessed at boot time on the serial console. This menu provides a means to read and write the memory contents before the boot firmware is loaded. We therefore designed a small GNU Debugger (GDB) stub that we injected into the hard drive's memory.

Inconveniently, our target hard drive's ARM 966 [3] core lacks hardware debug support. Therefore, we relied purely on software breakpoints, rather than on hardware breakpoints or single-stepping. In this context, software breakpoints are essentially instructions that trigger a data abort interruption. By hooking into that interrupt vector's handler and replacing instruction by a breakpoint, one can implement a debugger stub fully in software.

If a software breakpoint is overwritten prior to it being reached, e.g., because the firmware loads new code, the breakpoint will never be triggered. In addition, we have

observed that interrupt vectors or the debugger code itself can be overwritten by the firmware. To work around these problems, because of the lack of hardware breakpoints and watch-points, we manually identified all sections of code that load new code and hooked these functions to keep our debugger from disconnecting.

Finally, because setting a software breakpoint requires to modify instructions, it was not possible to put breakpoints on the ROM memory that contains the first bootloader and many other important library functions.

Our debugger stub itself requires only 3.4 kB of memory, and it can be easily relocated to a new address. It communicates with a GDB instance over the serial port while still allowing the firmware's debug messages to be printed on the serial port. As the stub is stateless, it does not require any permanent storage of information. Complex debugging features, such as the bookkeeping required for breakpoints, are managed on the reverse engineer's workstation by GDB.

2.2 Developing Malicious Payloads

Our main goal in designing a proof-of-concept compromised hard-drive firmware is to be able to modify blocks as they are read from or written to the disk. Hooking into *write* requests allows the backdoor to read and tamper with data blocks in the write buffer before they are written to the disk. In particular, we use a sequence of bytes in the first few bytes of a block, as a *magic value*. When this *magic value* is detected by the backdoored firmware, predefined actions of the backdoor will be triggered.

Hooking Writes in the Firmware.

A write operation in a modern hard drive specifies the logical block number to write to (LBA), the number of blocks to write, and the data to be written. This information is encoded in ATA commands and transmitted to the hard drive through the Serial ATA connection.

On the hard drive we reverse engineered, specialized hardware is responsible for receiving the ATA messages and notifying the firmware by raising an interrupt. The firmware then performs the action corresponding to the *opcode* field of the ATA message. In a *write DMA extended* ATA command, the data is then passed to the cache management task. This task keeps the received data blocks in volatile low-latency memory. When contiguous blocks are received, the firmware aggregates these blocks in memory. Eventually, the blocks will be evicted from cache memory, either because the cache is filled with newer data, or because a task commits them to the hard drive. Finally, the blocks will be passed to the read/write task, which takes care of positioning the head on the right track of the platter, and writes the data to the magnetic storage.

Figure 3 shows the sequence of the operations inside the hard drive. Our backdoor inserts itself in the call chain between the cache manager and the read/write task. By hooking writes after the cache, we ensure that the performance overhead remains low. At this point ATA commands have already been acknowledged, thus, the overhead of searching for the magic command in a block is less apparent to the user.

Reading Blocks from inside the Firmware.

Reading blocks inside the firmware proved to be harder than modifying writes. In order to read an arbitrary block, the modified firmware has to invoke a function providing several structured parameters. In our prototype implementation, this operation seems to trigger some internal side effect that makes the firmware unstable when multiple consecutive read operations are performed by our code.

Update Packaging and Final Payload.

Thanks to the debugger and the full firmware image, we were able to understand the firmware update format. We then generated a modified firmware update file that includes the original firmware infected with our proof-of-concept malicious code. Such a firmware update file can then be programmed to the disk with the manufacturer's firmware update tool, which could be done by a malware with administrator rights. The backdoor will then be permanently installed on the drive.

With the current state of our reverse engineering of the hard drive, we can reliably hook write commands received by the hard drive and modify the data to be written to the magnetic platter. The backdoor can also read and exfiltrate arbitrary blocks, but it is not stable enough to retrieve multiple blocks from the disk. A more stable implementation would allow the full port of the Data Exfiltration Backdoor that we will present in Section 3. We could invest more time to try to solve the bug in our code, but there are few incentives to do so as our aim is to demonstrate the feasibility of such attacks rather than to develop a weaponized exploit for the hard drive.

However, the current state is sufficient to fully implement more straightforward attacks. For example, we can reimplement the famous backdoor presented by Ken Thompson in *Reflections on Trusting Trust* [29]. In this lecture Thompson presented a compiler that inserts a backdoor while compiling the UNIX `login` command, allowing the password check to be bypassed. Similarly a compiler would transmit such a functionality when compiling a compiler. A malicious drive version of the `login` program backdoor simply detects a write to the disk of a critical part of the `login` binary and replace the code by a malicious version of the `login` binary.

2.3 Evaluation of the backdoor

We performed an *overhead test* to measure the impact of the backdoor under worst-case hard-drive operation. Indeed, if the backdoored firmware introduced significant overhead, this may alert a user of an anomaly.

This experiment is performed on the hard drive with the firmware backdoor described in Section 2.2, on an Intel Pentium E5200 2.5 GHz desktop computer equipped with 8 GB of physical memory. The hard drive was connected over internal SATA controller (Intel 82801JD/DO (ICH10 Family) 4-port SATA/IDE Controller).

Overhead Test.

We measured the write throughput on the test machine using IOZone [18]. As the backdoor functionality is only activated during writes, we use the IOZone *write-rewrite* test. We compare the write throughput obtained on the system running the unmodified hard drive firmware with the one running the backdoored firmware.

We perform the test with the IOZone `o_direct` option set to compare the results when the filesystem cache is not present. Most applications make use of the filesystem buffer cache to optimize access to the hard drive. However, with

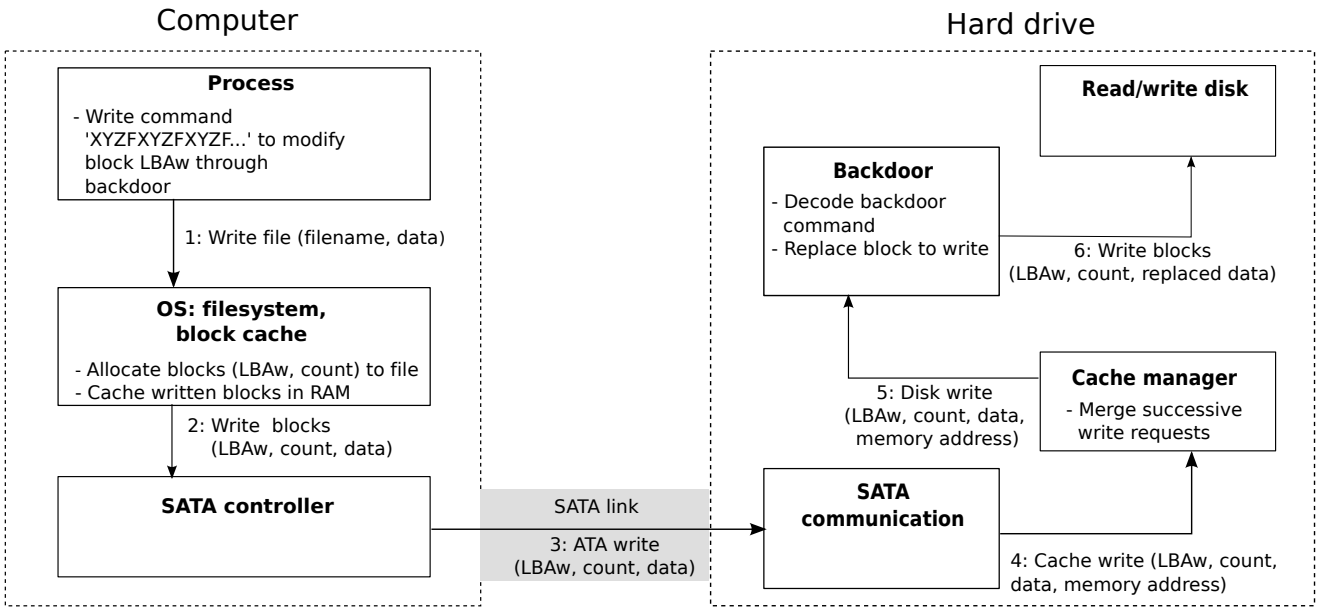

Figure 3: Call sequence of a write operation on the hard drive.

the cache enabled, our experiments showed it was impossible to distinguish the performance of the modified firmware from the original one. Hence, we emulate, as best as we can, a suspicious user attempting to detect hard-drive anomalies by testing the direct throughput.

Table 1: Filesystem-level write-throughput

	Write test	
	Mean (MB/s)	95% CI
With backdoor	37.57	[37.56; 37.59]
Without backdoor	37.91	[37.89; 37.94]

We perform 30 iterations of the experiment, with a 30 second pause between successive iterations. For each set of values measured, we compute 95%-confidence intervals using the t-distribution. Table 1 shows the comparison of the write throughputs of the hard drive with the unmodified and the backdoored firmware. In both cases, we executed the IOZone write/rewrite test to create a 100 MB file with a record length of 512 KB.

Comparing the results, we can conclude that the backdoor adds an almost unnoticeable overhead to write operations. For instance, to put those results into context, we measured larger disk throughput fluctuations by changing the cable that connects the hard drive to the computer than in the case of our backdoor.

3. DATA EXFILTRATION BACKDOOR

In this section, we present the design overview of a backdoor that allows to send and receive commands and data between the attacker and a malicious storage device, i.e., a Data Exfiltration Backdoor (DEB).

Basically, a DEB has two components: (*i*) a modified firmware in the target storage device and (*ii*) a protocol

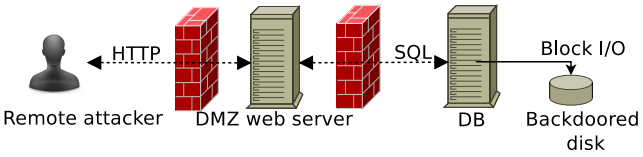

Figure 4: A server-side storage backdoor.

to leverage the modified firmware and to establish a bi-directional communication channel between the attacker and the firmware.

First we describe a concrete scenario in which the data exfiltration attack is performed, and then proceed to describe the challenges and our solution in detail.

3.1 Data Exfiltration Overview

We start with a real-world example of a server-side DEB, where the compromised drive runs behind a typical two-tier web server and database architecture, see Figure 4. 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 then read back content. This is the case for many web services. The specific example we select here is that of a web forum or blog service where users can post and browse comments.

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

First, 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 a trigger value, or *magic value*, and a disguised "read sector X" command for the backdoor. The web server passes this

comment data and other meta-data—such as the user name and timestamp—to the back-end database 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 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 just created. This causes an SQL SELECT query from the web application to the database, which triggers 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.

We stress that this DEB allows the attacker to read arbitrary sectors and access the storage device as a (remote) block device. The attacker can thus *remotely mount* filesystems and access files from the device selectively, without having to exfiltrate the storage device's contents fully.

For example, by extracting the first couple of sectors, the attacker can read the device's partition table, inferring the filesystem types in use. He or she can then follow the fileystem meta-data either locally inside the disk or remotely on his or her client machine to request individual files. We have automated this process and present results in Section 3.4. In conclusion, the attacker has a complete remote read access to the hard disk.

3.2 Challenges in Implementing a DEB

While modern operating systems and disks do little to actively prevent this type of attack, we have observed some challenges that we address next.

Data Encoding.

The character encoding chosen by the application should match 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 back without conflicts by the application.

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 caches above the storage device. Therefore, this delay depends on the load of the web server and can be influenced by the attacker.

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 in the next section, we mitigate this by *repeating* the magic value multiple times in a request, such that the overhead of searching for it becomes negligible. At the same time, these repeated sequences form a suitable space for the exfiltrated data to be written to by the firmware backdoor.

While the above challenges are certainly significant and render the exploitation of the backdoor more complicated, they do not *prevent* the use of DEBs in the general case. Our implementation provides adequate solutions to all the above complications.

3.3 Solutions Implemented

When a write request at a block number Y with a to-be-written buffer B is received, the backdoor checks for a *magic value* in buffer B. In our implementation the magic value is a sequence of bytes (*magic*), and followed by a sequence of bytes (*cmd*) specifying the malicious command to be executed. As we now focus on data exfiltration, *cmd* contains only the hex-encoded block number to be read. It would be easy to extend this encoding, for example, to support other operations, such as appending data to existing blocks, tampering with stored data, or injecting malicious code into executables. Here, 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 (such as the filesystem) between the attacker and the disk 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, and (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 value increases the performance overhead of the backdoor. We chose a 4-byte magic value which results in a low performance overhead.

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

```
bkdr_count ← length(magic) + cmd_size
for blk in blocks do
  if magic present in first bkdr_count bytes of blk then
    if blk 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
    block_num ← hex_decode(cmd)
    buf ← read_block(block_num)
    base64_encode(buf)
    blk ← buf
  end if
end for
```

If the magic value is present in B, the malicious behavior of the DEB is triggered: The backdoor extracts the command from the request data, such as "read data at sector X" for data exfiltration from the storage device, as shown in Algorithm 1. The backdoor reads data buffer B' from

sector X, encodes it using base64, which increases its size by $\frac{1}{3}$, and writes B'. To ensure that the encoded data can be successfully exfiltrated, the backdoor checks for the presence of at least $bkdr\_bs * \frac{4}{3}$ bytes of consecutive magic values in a sequence of blocks and then replaces these by the base64-encoded data. At this point, a future read request at address Y will return the modified content, allowing unauthorized data exfiltration of the contents at address X from the device to a remote attacker.

Valid magic sequences could occur during normal, non-malicious use of the storage device. Such a false-positive would result in the storage device to detect the magic sequence and write faulty data to a sector, possibly undermining the stability of the system. However, such a false positive can only occur with negligible probability, as the backdoor always checks for about two blocks of successive magic values before attempting a replacement.

Also note that the firmware can write B' to Y possibly after modifications through cryptographic and steganographic operations to prevent easy detection by the administrator of the target machine.

3.4 DEB Evaluation

As we mentioned in the previous section, our backdoor in the off-the-shelf disk drive it is not stable enough to perform multiple arbitrary reading operations from the disk, which is required for implementing the complete DEB. In this section, we therefore report on experiments performed on a QEMU-based prototype.

We implemented the DEB inside QEMU's storage device functionality, which is used when using virtual IDE drives in system-virtualization software such as KVM and Xen. This provided us with an easy-to-use platform to develop, test, debug, and evaluate the backdoor.

In this case, we evaluate the data exfiltration *latency* from an attacker's point of view. In addition, we perform a file *exfiltration test* to show the feasibility of retrieving sensible remote files without needing to exfiltrate the entire disk. We base this evaluation on the scenario described in Section 3.1. We have conducted experiments on a virtual machine with 1 GB of memory running on a modified QEMU containing the backdoor. This is the attacker's target host. Our tests were performed on the emulated IDE disk with write-back caching. The target host runs Ubuntu and an Apache web server with two PHP scripts providing 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 a MySQL database which runs atop an ext3 filesystem.

We emphasize here that the results of this second set of experiments highly depend on the application, the workload on the machine, and the total available system memory – and do not depend much on the caracteristics of the disk or firmware backdoor. Indeed, because the Linux page cache[1] is essentially an LRU-like cache, forcing the eviction of pages from main memory requires generating accesses for about as much data as there is free available memory for buffers and caches on the system (and the more eager the operating system is to swap pages, the higher the memory that is available). For a single block, the time to generate that workload largely dominates the transfer time from and to the disk for a single block (even in our setup where relatively little memory is available).

[1]The page cache caches blocks read from and written to block devices, and is integrated with the filesystem cache (or buffer cache).

Table 2: Data exfiltration performance

	Mean (s)	95% CI
Insert	10.7	[10.65; 10.71]
Latency	9.7	[9.55; 9.82]
File exfiltration	40.0	[39.6; 40.4]

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

Latency Test.

Because of caching, the inserted comments are not immediately updated with the exfiltrated data. In fact, the malicious blocks are temporarily stored in the page cache — from where they are retrieved when they are immediately accessed by the attacker. Therefore, the presence of a cache forces the attacker to wait until the blocks are evicted from the cache. In our scenario, this can be forced by the attacker as well, namely by inserting dummy comments to quickly fill up the cache and thus force eviction of least recently accessed data.

The *insert time* in Table 2 shows the time taken to insert 500 8-KB comments sequentially, using the PHP form. As described in Section 3.3, the backdoor replaces each of these comments with 3 KB of exfiltrated data starting at the sector number included in the comment. The *latency time* in Table 2 shows the update latency in seconds for the 500 comments inserted during the insert test — during this time, the attacker sends many other dummy comments to speed up cache eviction. It follows that an attacker is able to exfiltrate 3000 sectors in $10.7 + 9.7 = 20.4$ seconds in our setup, achieving a read bandwidth of 74 KB/s. In practice, an attacker may limit bandwidth to avoid detection. In addition, those values will differ depending on the characteristics of the system (mainly, more physical memory will cause the comments to persist longer in cache, and more load on the server will cause the opposite). Hence, these results show that the latency is likely to be sufficiently low, and that an attacker can realistically use this technique.

Exfiltration Test.

Let's now consider a typical case in which an attacker attempts to exfiltrate the `/etc/shadow` file on the target host.

To that end, we created a python program 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/shadow` file on the target host.

Table 2, row 3, shows that `/etc/shadow` can be exfiltrated in less than a minute. Because the process of retrieving the file requires nine queries for a few sectors, each of them de-

pending 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 about 4 seconds (for a comparison, note that the latency figure in row 2 also includes the retrieval time of the 3000 sectors).

4. DETECTION AND PREVENTION

We first discuss the applicability of existing standard techniques for defeating or mitigating DEBs, including encryption of data at rest, signed firmware updates, and intrusion detection systems. Subsequently, we propose two new techniques specifically targeting the detection of DEBs: OS page cache integrity checks and firmware integrity verification.

4.1 Encryption of Data at Rest

The use of encryption of data at rest is still an exception, both on servers and desktop computers. When used, it is often for the purpose of regulatory compliance or to provide easy storage-device disposal and theft protection (by securely deleting the encryption key associated with a lost disk). *Under some conditions*, encryption of data at rest mitigates the possibility of data-exfiltration backdoors on storage devices: it renders establishing a covert communication channel more difficult for remote attackers and prevents the untrusted storage device from accessing the data in the first place.

Hardware-Based Disk Encryption.

Hardware-based disk-encryption mechanisms commonly rely on the hard disk drive to encrypt data itself. Decryption is only possible after a correct password has been provided to the drive. In such a setup, as data is encrypted and decrypted within the drive, a backdoor would only have to hook into the firmware before the encryption component. Thereafter, the hard-disk will encrypt and decrypt data for the backdoor.

Software-Based (Filesystem and Partition) Encryption.

Other hard-disk encryption systems, among them BitLocker, FileVault, and TrueCrypt, encrypt full partitions over arbitrary storage devices. Such mechanisms often rely on a minimal system to be loaded from a non-encrypted partition whose integrity is verified by a trusted boot mechanism. A trusted boot mechanism relies on a TPM to prevent a modified system, e.g., modified by the drive itself, to access a protected key sealed by the TPM. However, without an IOMMU, the backdoor on a hard drive can launch a DMA attack [13] to read arbitrary locations from the main memory. This allows the backdoored hard disk to obtain the encryption key. Recently, it has been shown that even mechanisms to protect encryption keys against DMA attacks [24] can be circumvented [7].

In conclusion, neither hardware-based nor software-based encryption offer full protection against DEBs in all cases. Disk encryption *can* prevent DEBs as presented in this paper when keys are not managed by the disk itself and when the disk is not able to use DMA to access main memory. This corresponds to setups in which:

- system-level encryption is used *and* disks are attached to the computer (e.g., desktops or laptops) *and* an IOMMU (e.g., Intel VT-d or AMD SVM) is present and properly configured;

- system-level encryption *and* remote storage are used, for example, servers with a Network Attached Storage (NAS) or Storage Area Network (SAN). Such a remote storage must not support remote DMA capabilities, like Infiniband or Myrinet protocols does.

We believe that both setups are uncommon. While IOMMUs are present in many computers, they are rarely activated because of their significant performance overhead [6]. On the other hand, servers that rely on a SAN or NAS are typically not using software disk encryption because of its significant performance impact.

4.2 Signed Firmware Updates

To protect a device from malicious firmware updates, cryptographic integrity checks can be used. The use of asymmetric signatures is preferable in this case, and each device would be manufactured with the public key of the entity performing the firmware updates. Although the idea of signing the firmware is widely known, we have not been able to assess how widespread its use is for hard-disks and storage devices in general. We have found evidence that some RAID controllers [26] and USB flash storage sticks [20] have digitally signed firmware, but these appear to be exceptions rather than the rule.

Nevertheless, signed firmwares do not prevent an attacker with physical access to the device from replacing it with an apparently similar, but in reality backdoored, device. Also note that the recent compromise of certification authorities, software vendors' certificates, and hash collisions has demonstrated real-world limitations of signature mechanisms.

Finally, firmware signatures merely check code integrity at load time and do not prevent modifications at run time. A vulnerability in the firmware that is exploitable from the ATA bus[2] would allow infection of the drive, bypassing the signed update mechanism. In addition, such vulnerabilities are likely to be easily exploitable, because no modern exploit-mitigation techniques are present in the disk firmwares we analyzed.

4.3 Intrusion Detection Systems

Current network-based intrusion detection systems and antivirus software products use, to a large extent, simple pattern matching to detect known malicious content. The DEBs presented in this paper could be detected by such tools if the magic value is known to the latter. This can be the case if the attacker targets a large number of machines with the same magic value, but is inadequate for targeted attacks. For instance, an attacker could change the magic value for each target machine or it would be possible to make the magic values a time-dependent function to evade detection. Finally, the attacker's channel used for communication with the firmware may be encrypted. We conclude that today's intrusion detection systems do not offer a strong protection mechanism against DEBs.

[2]Or an insecure functionality that could be abused without physical access.

4.4 Page-cache-driven Integrity Checks

In addition to the standard mechanisms presented above, one could also envision detection technique that relies on the page cache. Most filesystems leverage the page cache to significantly speed up workloads by caching most recently accessed blocks. We propose to modify the page cache to also perform probabilistic detection of DEBs. As the cache contains recently written data, it can be used to check the integrity of disk-provided data.

More precisely, the cache would allocate a new entry on write misses, and, after the data has been written to the disk (immediately for write-through caches, and after laundering for writeback caches), subsequent reads from the cache would be randomly subject to asynchronous integrity checks. The checks would simply read back data from the disk and check for a match.

However, with deterministic cache-eviction algorithms such as *least recently used* (LRU), both the disk and the remote attacker could estimate the size of the cache in use, and the attacker could adjust queries to guarantee that the data has been evicted from the cache by the time it is read back. Therefore, we suggest to partially randomize the cache eviction policy. For instance, a good candidate would be a *randomization-modified LRU-2* algorithm, whereby the eviction from the first-level cache to the second-level cache would remain LRU, but the eviction from the second level cache would be uniformly random. This technique would introduce a performance overhead, but we conjecture that this could be an acceptable trade-off for detecting such backdoors in the wild.

4.5 Detection Using Firmware Integrity Verification

Recent research in device attestation [22] could be applied to detect malicious firmwares. However, we note that device attestation is controversial [9], especially in the specific context of this work: the firmware is typically stored in different regions of the drive (such as disk platters and serial flash), and accessing those different regions is slow and subject to various time delays. Delays are difficult to predict, and this questions standard assumptions made by existing software-based attestation techniques, rendering them ineffective in our scenario.

However, one could leverage the fact that the disk always starts executing from the ROM code, essentially providing a hardware root of trust. By interfacing with the ROM bootloader and using it to control execution and verify code loading one could guarantee that only correct code was loaded.

5. RELATED WORK

Backdoors have a long history of creative implementations: Thompson [29] 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. King et al. [19] 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 [15, 16] that insert rootkits into the kernel at boot time. However, with the exception of Triulzi [30], who presented a NIC backdoor that provides a shell running on the GPU, and contrary to our approach, those previous backdoors were only *bootstrapped* from hardware devices, from where they tried to compromise the host machine's kernel. Therefore, those kinds of backdoors can be detected and prevented by kernel integrity protection mechanisms, such as Copilot [25], which is implemented as a PCI device.

Cui et al. [11] present a firmware modification attack on HP LaserJet printers. The authors remark that, in the case of most printers, firmware updates could be performed by sending specially-crafted printing jobs. Cui et al. also state that firmware updates were not signed and that signing would not prove sufficient in the presence of exploitable vulnerabilities, which is in line with our observations. In addition, they create, as payload, a VxWorks rootkit capable of print job exfiltration using the network link the printer is connected to. However, all such communications can easily be prevented by following network-level best practices (segregating printers into their own VLANs without direct internet connectivity). In contrast, in this work, we focus on hard drives and present a data-exfiltration payload that works without direct internet connectivity.

Concurrently and independently from our work, Domburg (a.k.a. sprite_tm) reverse-engineered a hard drive from another manufacturer and also demonstrated that modifying a hard disk firmware to insert a backdoor is feasible [2], albeit without demonstrating data exfiltration. This confirms our findings that reverse engineering of a hard-drive firmware is possible and within reach of moderately funded attackers.

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 equally easily be identified and blocked by a firewall at the network boundary. Thus, a covert channel is needed to communicate with the backdoor, as mentioned in [12] 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 easily distinguished from legitimate traffic at the network level.

6. CONCLUSION

This paper presents a practical, real-world implementation of a data exfiltration backdoor for a SATA hard disk. The backdoor is self-contained, requiring no cooperation from the host. It is stealthy, in that it only hooks legitimate reads and writes, with no reliance on DMA or other advanced features. We compromised a common off-the-shelf disk drive with a backdoor that is able to intercept and perform read and write operations, with an almost impossible to detect overhead. This backdoor can be installed by software in very little time. We have also demonstrated that it is feasible to build such a backdoor with an investment of roughly ten man-months, despite difficulties in debugging and reverse engineering a disk's firmware. To mitigate the threat, we recommend encrypting data at rest to reduce the trust placed in the disk. We also present a number of forensic techniques which can help to identify a similar backdoor, although we emphasize that further research is needed to catch such a backdoor in the wild.

7. ACKNOWLEDGEMENTS

The research leading to these results was partially funded by the European Union Seventh Framework Programme (contract N 257007) and the US National Science Foundation (grant N 1218197).

References

[1] HDD Guru Forums, 2013. URL http://forum.hddguru.com/.

[2] Jeroen Domburg (a.k.a. Sprite_tm). HDD Hacking. URL http://spritesmods.com/?art=hddhack. Talk given at OHM 2013.

[3] ARM. ARM966E-S, Revision: r2p1, Technical Reference Manual, 2004. URL http://infocenter.arm.com.

[4] backupworks.com. HDD Market Share - Rankings in 2Q12, 2012. URL http://www.backupworks.com/hdd-market-share-western-digital-seagate.aspx.

[5] F. Baker, B. Foster, and C. Sharp. Cisco Architecture for Lawful Intercept in IP Networks. RFC 3924 (Informational), October 2004. URL http://www.ietf.org/rfc/rfc3924.txt.

[6] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn. The Price of Safety: Evaluating IOMMU Performance. In *The Ottawa Linux Symposium*, pages 9–20, Ottawa, Canada, 2007.

[7] E.-O. Blass and W. Robertson. TRESOR-HUNT: Attacking CPU-Bound Encryption. In *Annual Computer Security Applications Conference*, pages 71–78, Orlando, USA, 2012. ISBN 978-1-4503-1312-4.

[8] Robert A. Caldeira, John C. Fravel, Richard G. Ramsdell, and Romeo N. Nolasco. Hard disk drive architecture, July 1995. URL http://www.freepatentsonline.com/5396384.html.

[9] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Conference on Computer and Communications Security*, pages 400–409, Chicago, USA, 2009. ISBN 978-1-60558-894-0.

[10] T. Cross. Exploiting Lawful Intercept to Wiretap the Internet. Black Hat, 2010. URL http://www.blackhat.com/.

[11] A. Cui, M. Costello, and S.J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Network and Distributed System Security Symposium*, 2013 (to appear).

[12] Daemon9. Project Loki. Phrack 49. URL http://www.phrack.org/issues.html?id=6&issue=49.

[13] M. Dornseif. Owned by an iPod: Firewire/1394 Issues. PacSec, http://md.hudora.de/presentations/firewire/PacSec2004.pdf, 2004.

[14] J.A. Halderman and E.W. Felten. Lessons from the Sony CD DRM episode. In *USENIX Security Symposium*, pages 77–92, 2006.

[15] J. Heasman. Implementing and Detecting an ACPI BIOS Rootkit. Black Hat, 2006. URL www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf.

[16] J. Heasman. Implementing and Detecting a PCI Rootkit. Black Hat, 2007. URL http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf.

[17] O.S. Hofmann, A.M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, Newport Beach, USA, 2011. ISBN 978-1-4503-0266-1.

[18] IOZone. IOZone, 2013. URL http://www.iozone.org/.

[19] S.T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and Implementing Malicious Hardware. In *Workshop on Large-scale Exploits and Emergent Threats*, San Francisco, USA, 2008.

[20] Kingston. Secure USB Flash Drives, 2013. URL http://www.kingston.com/us/usb/encrypted_security.

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

[22] Y. Li, J.M. McCune, and A. Perrig. VIPER: Verifying the Integrity of PERipherals' Firmware. In *Conference on Computer and Communications Security*, pages 3–16, Chicago, USA, 2011. ISBN 978-1-4503-0948-6.

[23] Maxtor. Maxtor Basics Personal Storage 3200 virus, 2013. URL http://knowledge.seagate.com/articles/en_US/FAQ/205131en?language=en_GB.

[24] T. Müller, F.C. Freiling, and A. Dewald. TRESOR runs encryption securely outside RAM. In *USENIX Security Symposium*, pages 17–17, San Francisco, USA, 2011. URL http://www.usenix.org/event/sec11/tech/full_papers/Muller.pdf.

[25] N.L. Petroni Jr, T. Fraser, J. Molina, and W.A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, San Diego, USA, 2004. URL http://usenix.org/publications/library/proceedings/sec04/tech/full_papers/petroni/petroni.pdf.

[26] RSA. Configuring the RSA II adapter, 2013. URL http://www.scribd.com/doc/3507950/RSAII-Card-Installation.

[27] S. Sparks, S. Embleton, and C.C. Zou. A Chipset Level Network Backdoor: Bypassing Host-Based Firewall & IDS. In *Symposium on Information, Computer, and Communications Security*, pages 125–134, Sydney, Australia, 2009. ISBN 978-1-60558-394-5.

[28] F. Sultan and A. Bohra. Nonintrusive Remote Healing Using Backdoors. In *Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, USA, 2003. URL http://www.cs.rutgers.edu/~bohra/pubs/sm03.pdf.

[29] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, 1984. ISSN 0001-0782.

[30] A. Triulzi. Project Maux Mk.II, I 0wn the NIC, now I want a Shell! In *PacSec Conference*, 2008. URL http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf.