Proceedings of the Eleventh SIGOPS European Workshop, ACM SIGOPS, Leuven, Belgium, September 2004.

Secure Bootstrap is Not Enough: Shoring up the Trusted Computing Base

James Hendricks
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA
James.Hendricks@cs.cmu.edu

Leendert van Doorn
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY
leendert@watson.ibm.com

Abstract

We propose augmenting secure boot with a mechanism to protect against compromises to field-upgradeable devices. In particular, secure boot standards should verify the firmware of all devices in the computer, not just devices that are accessible by the host CPU. Modern computers contain many autonomous processing elements, such as disk controllers, disks, network adapters, and coprocessors, that all have field-upgradeable firmware and are an essential component of the computer system's trust model. Ignoring these devices opens the system to attacks similar to those secure boot was engineered to defeat.

1 Introduction

As computers continually integrate into our business and personal lives, corporate and home users are storing more sensitive data on their personal computers. However, widespread Internet usage has exposed more computers to attack and provided would-be attackers with the information needed to scale such attacks. To protect this increasingly sensitive data from these increasingly prolific attacks, next-generation personal computers will be equipped with special hardware and software to make computing more worthy of trust. Such *trustworthy computing* will provide security guarantees never before seen on personal computers.

Trustworthy computing requires a Trusted Computing Base (TCB)—a core set of functionality that is assumed secure—to implement the primitives that provide security guarantees. The TCB typically consists of hardware, firmware, and a basic set of OS services that allow each application to protect and secure its data and execution. Security of the bootstrap mechanism is essential. Modeling the bootstrap process as a set of discrete steps, if an adversary manages to gain control over any particular step, no subsequent step can be trusted. For example, consider a personal computer with a compromised BIOS. The BIOS can modify the bootstrap loader before it is executed, which can then insert a backdoor into the OS before the OS gains control.

This secure bootstrap problem is well-known and various solutions have been proposed to deal with it. For example, Arbaugh et al. [1] propose a mechanism whereby the first step in the bootstrap process is immutable and therefore trustworthy. This *trust* is then bootstrapped all the way up to the operating system by checking a digital signature for each bootstrap step before it is executed. For example, the BIOS could verify a public-key signature of the disk's boot sector to ensure its authenticity; the boot sector could then verify the public-key signature of the OS bootstrap code, which could likewise verify the privileged OS processes and drivers. Though such an approach would obviously not guarantee the security of the OS code, it would at least guarantee the authenticity.

A weakness to this approach is that the BIOS in most personal computers is writable. One solution is to store the BIOS on a ROM. However, a ROM-based approach is by definition inflexible, preventing BIOS updates that may be required to support maintenance applications, network booting, special devices, or CPU microcode updates. Furthermore, the use of digital signatures introduces a key management problem that is amplified by the requirement to store the initial public key in ROM. To ameliorate these problems, a secure hardware device can be used both to verify a programmable BIOS and to authenticate this verification. This is the approach taken by the Trusted Computing Group (TCG)[13], described in Section 2.

Both the Arbaugh et al. and TCG based approaches share a CPU-centric view of the system that is inadequate for establishing a trustworthy system. In Section 3, we argue that, though the current specification goes to much trouble to defend against attacks utilizing the CPU, it fails to defend against similar attacks utilizing peripherals, and in Section 4 we argue that such attacks are not much more difficult. Section 5 describes how the current specification could be improved with a minor augmentation.

2 The Current Approach

The Trusted Computing Group advocates using a secure hardware device to verify the boot sequence and authenticate this verification. Such a device could provide assurance even to a remote user or administrator that the OS at least started from a trustworthy state. If an OS security hole is found in the future, the OS can be updated, restarted, and re-verified to start from this trustworthy state. An example of this kind of device is the Trusted Platform Module (TPM) [14]. Such a device has been shown to enable a remote observer to verify many aspects of the integrity of a computing environment [8], which in turn enables many of the security guarantees provided by more complex systems, such as Microsoft's NGSCB (formerly Palladium) [4].

The following is a simplified description of how the

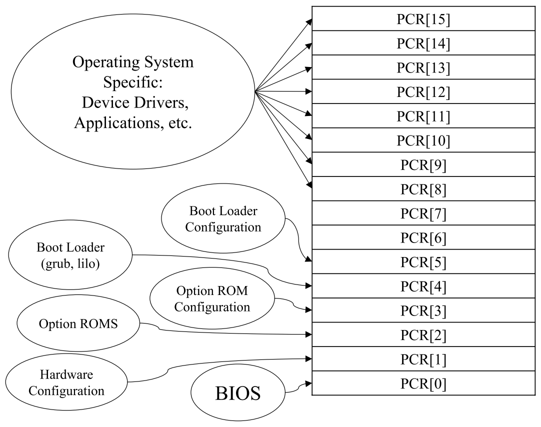

Figure 1: Hashes of the bootstrap code, operating system, and applications are stored in the Platform Configuration Registers, which can later be queried to verify what was executed.

TPM can be used to verify the integrity of a computing system (see the specification for details [15]). The TPM *measures* data by hashing the data. It *extends* a measurement to a Platform Configuration Register (PCR) by hashing together the current value of the PCR and the hash of the data and storing the result in the PCR. To *measure to a PCR*, the TPM measures data and extends it to a PCR. All code must be measured before control is transferred to it.

When the computer is reset, a small and immutable code segment (the Core Root of Trust for Measurement, CRTM) must be given control immediately. The CRTM measures all executable firmware physically connected to the motherboard, including the BIOS, to PCR[0] (PCR[0] is the first of sixteen PCRs). The CRTM then transfers control to the BIOS, which proceeds to measure the hardware configuration to PCR[1] and option ROM code to PCR[2] before executing option ROMs. Each option ROM must measure configuration and data to PCR[3]. The BIOS then measures the Initial Program Loader (IPL) to PCR[4] before transferring control to it (the IPL is typically stored in the first 512 bytes of a bootable device, called the Master Boot Record). The IPL measures its configuration and data to PCR[5]. PCR[6] is used during power state transitions (sleep, suspend, etc.), and PCR[7] is reserved. The remaining eight PCRs can be used to measure the kernel, device drivers, and applications in a similar fashion (the post-boot environment), as Figure 1 depicts.

At this point, the bootstrap code, operating system, and perhaps a few applications have been loaded. A remote observer can verify precisely which bootstrap code or operating system has been loaded by asking the TPM to sign a message with each PCR (the TPM_QUOTE command); this operation is called *attestation.* If the TPM, operating system, bootstrap code, and hardware are loaded correctly, the remote observer can trust the integrity of the system. The TPM should be able to meet FIPS 140-2 requirements [14]; hence, it is reasonably safe to assume the TPM is trustworthy (see FIPS 140-2 requirements for details [16]). The integrity of the operating system and bootstrap code is verified by the remote observer; hence, the operating system and bootstrap can be trusted to be what the remote observer expects. The hardware, however, is not verified; fortunately, hardware is more difficult to spoof than software.

From this, we can describe attacks that are and are not defended against. Attacks that exploit a known hole in the OS can be detected at attestation. Attacks that modify the BIOS, option ROMs, or IPL are detected at boot. Similarly, upgrades and repairs to these components are verifiable. However, physical attacks on the TPM (such as invasive micro-probing or EM attacks [7]) or other components (such as RAM bus analysis) are not detected. Furthermore, malicious hardware may provide an avenue of attack; a malicious processor would not be detected by attestation, yet it could circumvent most security policies.

For Microsoft's NGSCB, an alternate secure boot method is proposed [15]. This method requires the addition of a new operation to the CPU instruction set architecture that resets the CPU and ensures the execution of a secure loader without reseting the I/O bus. This method allows the secure loader to gain full control of the CPU without the need to reinitialize the I/O subsystem. While this method reduces its reliance on the BIOS, it still assumes that the CPU is in control of all executable content in the system, which, we argue, is a flawed assumption.

3 A Security Vulnerability in This System

Though it is relatively safe to trust hardware circuits (because mask sets are expensive to develop, etc.), there is less sense in trusting firmware. Firmware is dangerous because it can be changed by viruses or malicious distributors. Though current attestation methods detect attacks on the OS, BIOS, and option ROMs, attacks on other firmware may be no more difficult. Firmware with direct access to memory is no less dangerous than the BIOS or the kernel, and even firmware without direct memory access may require trust. Hence, though peripherals and memory are implicitly proposed to be a part of the TCB, we do not believe they are currently adequately verified.

Consider a compromised disk. For example, assume the delivery person is bribed to allow an attacker to "borrow" the disk for a few hours to be returned in "perfect" condition. This disk could collect sensitive data; modern disks are large enough that the compromised firmware could remap writes so as to never overwrite data (similar to CVFS [10]). On a pre-specified date, or when the disk starts to run low on storage, the disk can report disk errors. The disk could ignore commands to perform a low-level format or otherwise erase its data while being prepared for warranty service. Once again the bribed delivery person could allow the attacker physical access, literally delivering gigabytes of sensitive data to the attacker's doorstep.

The attacker could then reset the firmware to act normal for a few months, leading the disk vendor to send the disk to another customer because it believes this customer misdiagnosed the problem.

Generalized, the above attack takes place in three phases: first, the device is compromised; second, the device compromises the integrity of data; third, the device delivers data to the attacker. There are many techniques to perform each of these steps, and security is violated even if the third step does not occur.

3.1 Compromising a Device

The first step is to compromise the device. We consider only attacks on firmware for autonomous computing engines that are not under control of the main CPU. These include the operating systems found on disks [2] and some network cards [6]. We rule out attacks that replace parts of the hardware for several reasons: replacement requires physical access; unlike overwriting firmware, replacement costs money; the cost of fabricating a custom device is likely much greater than the cost of modifying the firmware; etc. Furthermore, we assume the manufacturer is not malicious.

The most direct attack is to provide a firmware update to the user and use social engineering to convince the user to install this update. Or consider the man-in-the-middle attack, where the device is compromised after it leaves the trusted manufacturer but before it arrives at the victim. For example, the manufacturer may outsource the actual manufacturing to a plant in an adversarial country, where the firmware could easily be replaced. The delivery person, the installation crew, or the maintainance team could similarly compromise the firmware. A less glamorous (but more likely) attack would be to embed the update in a virus or worm that scans infected systems for vulnerable devices.

Essentially, any attack that can compromise an unattested operating system could likely compromise unattested firmware. Furthermore, note that once a device is compromised, future firmware updates may not guarantee that the device is safe (the malicious firmware could modify the update utility or ignore update commands); also, reinstalling the computer software won't reinstall the firmware. Hence, compromising firmware is potentially more damaging than compromising the operating system.

3.2 Compromising Data

Once the firmware has been replaced with malicious firmware, there are two ways in which the device can compromise the integrity of data. If the device can directly issue a DMA request, or if it can solicit a device to issue a DMA request on its behalf, it can overwrite valid data or read confidential data in host RAM. But even if DMA is not an option, the device can still store unencrypted and manipulate unauthenticated data that is fed to it, or simply discard data.

3.3 Delivering Data to the Attacker

If the compromised device is a network device, it can deliver confidential data over the network. If the device has direct or indirect DMA access, it can bus master a DMA request to the network device's ring buffer, which the network device will then transmit over the network. But even if there is no reachable network connection to the outside world, a device may still be able to breach confidentiality; for example, the device can store data and then misbehave, causing the user to send the device in for warranty. Once again, a man-in-the-middle attack can be used, this time to extract the data and hide the tracks of the malicious firmware (other attacks used to compromise the device may be similarly adapted). Note that storing data is not unique to storage devices; this works for any device with an EEPROM, and every device vulnerable to an attack on its firmware has some EEPROM.

3.4 Summary

All DMA-capable peripherals are trusted, and must either be verifiable or not have firmware. Furthermore, many devices without DMA capabilities are trusted to some degree. If these devices may have firmware that is not verified, data sent to them must be either encrypted and authenticated or insensitive to security violations. There remains a question of feasibility: even if it is feasible to replace the firmware, read or modify sensitive data, and deliver sensitive data, how difficult is it to generate the malicious firmware?

4 Is Writing Malicious Firmware Feasible?

Security is about risk management; hence, it is appropriate to ask which attacks are most likely. Attacks on software have been shown to be quite popular; attacks on firmware and hardware have been less prolific. We argue that attacks on firmware are only incrementally more difficult than attacks on software, and that, once attacks on software become more difficult, attacks on firmware will become common. We further argue that attacks on hardware are more difficult because hardware is not malleable; hence, circuits and ROMs are relatively trustworthy.

Because security is about risk management, there is a natural tendency for conflicts to escalate to slightly more sophisticated variants. Defenders plug the easiest holes, and attackers ratchet attacks up to the next level. For example, the simplest buffer-overrun relies on jumping to executable code on the stack. The direct solution, non-executable stacks, led to slightly more elaborate attacks [17]. Perhaps the greatest vulnerability of firmware attacks is that modifying firmware may be no harder than modifying OS code. We believe attacks have been limited up to this point because firmware has been less homogeneous than software and most programmers have less experience with firmware. Both of these factors are changing: device vendors are consolidating, and programmers are being exposed to firmware. The LinuxBIOS project [5] has successfully replaced the BIOS of several commodity PCs to provide flexibility. Also, hacked firmware is becoming more common: many DVD players have hacked firmware to support DVDs from any region [9], and game stations such as the X-Box have hacked versions of firmware [3] that convert them into cheap computers.

As discussed above, any device that can DMA and any device that is fed unencrypted or unauthenticated data is a threat. Unless these devices are verified, one of two options must be taken to ensure security: either DMA must be disabled and all accesses to devices must be encrypted and authenticated, or memory must not be trusted (as in AEGIS [11] or XOM [12]). Both options are severe and would limit performance.

5 The Technical Solution

This paper contributes two complimentary technical solutions: 1) Each compliant device must be included in the TCB. It must ensure that its firmware is signed and verified at startup just like the rest of the executable code, and it must verify its children. Such recursive verification will form a tree of trust. 2) Every other device must be recognized as explicitly external to the TCB. Applications must be aware that it is unsafe, and its I/O must be sandboxed.

5.1 An Example: A Trustworthy Disk

A trustworthy disk would have a firmware signing mechanism: for example, a cheap processor and ROM for some immutable root of trust. On power-on, this system would work in much the same manner as the TPM; all security sensitive code would be measured to a local PCR, which would then be signed with a key embedded in the disk's TPM and returned to the host CPU on request. Of crucial importance is that this mechanism is not necessary for basic operation of the device; it is an optional feature. The disk can be manufactured and the additional firmware signing hardware can be installed optionally. The signing hardware could read the firmware directly and send the measurement through a vendor specific command to the host CPU. Such a solution would have a marginal cost for systems without the security hardware, and likely less than a dollar for systems with the hardware, which both keeps costs down and provides disk vendors with a "value add."

5.2 The Generalized Solution: A Verification Mechanism for Trusted Peripherals

A generalized version of the above solution is to descend the device chain and recursively verify the trustworthiness of all devices. On system reset, the BIOS and option ROMs are currently measured, as well as the current hardware configuration. When the hardware configuration is measured, each device should measure its firmware. For example, when the PCI bus is configured and measured, each device on the PCI bus should attest its firmware, if it is field-upgradeable. During PCI configuration, the SCSI host adapter will be queried; the SCSI host adapter will measure its firmware then query each disk; finally, each disk will measure its firmware and return this measurement. This creates a tree of trusted devices, as depicted in Figure 2.

The host can determine the trustworthiness of a device by assuming that the device was initially secure and therefore verify the initial attestation statement against future ones, or the host can compare the firmware attestation statement against a trust certificate provided by the device vendor. If the device is unable to provide an attestation statement or the vendor is unable to provide a trust certificate, we have to assume the firmware and therefore the device cannot be trusted.

5.3 Untrustworthy Devices

Because there may exist some devices whose trustworthiness is unknown, there must be a compatibility mode. One solution is to tag such devices as untrustworthy, and restrict their DMA access to a memory address range sandbox using mechanisms similar to an I/O-MMU or machine partitioning [4]. Furthermore, the operating system and sensitive applications must understand that they cannot rely on unencrypted or unauthenticated data sent or received from an untrustworthy device. All devices bridged by an untrustworthy device are untrustworthy; for example, a trustworthy disk attached to an untrustworthy SCSI controller is untrustworthy.

5.4 Guarantees Provided

If all critical software and firmware are verifiable, then only attacks on hardware can go undetected. For example, consider a system where the OS is verifiable, boot firmware is verifiable, field upgradable firmware for trusted devices is verifiable, and all other devices are sandboxed as in Section 5.3. Then all remotely malleable components are verifiable, and, for the first time ever, strong guarantees can be provided: all remote attacks on PCs are remotely detectable as soon as the method of attack is known, patches can be verifyably installed, and attacks cannot survive across reboot. A remote observer can verify that a PC is not vulnerable to any known remote attacks; attacks can no longer hide in unverified storage. Known attacks on software are likely to be fixed with a patch that can be verifyably installed. Likewise for firmware; furthermore, if no patch is provided, the firmware can be isolated as untrustworthy. Hence, assuming that all vulnerabilities are eventually discovered—and many vulnerabilities are discovered before attacks surface—attackers are limited to hardware attacks. Hardware attacks either requires physical access or buggy hardware; the former is hard to come by and the latter can be isolated.

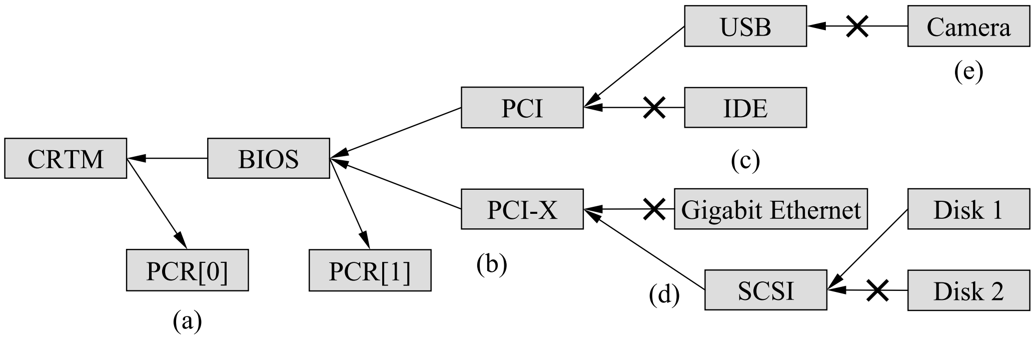

Figure 2: a) On reset, the CRTM measures the BIOS to PCR[0] before transferring control to it. b) The BIOS recursively measures devices on the PCI bus and PCI-X bus. c) The IDE controller and Gigabit Ethernet controller do not support firmware measurements—they cannot be trusted—and hence their DMA must be sandboxed (the Gigabit Ethernet sandbox is its entire ring buffer). d) The SCSI controller reports that one of its disks cannot be trusted with unencrypted or unauthenticated sensitive data. e) The USB controller reports that the Camera cannot be trusted; however, the USB controller itself can still utilize DMA.

6 Conclusion

The added complexity of any security facility is worthwhile only if the additional security provided justifies its cost. But the additional security of current secure bootstrap facilities is minimal, because they are vulnerable to attacks on firmware. These attacks are at least as damaging as their software counterparts, as deployable, and nearly as straight forward. Fortunately, a simple extension to secure bootstrap prevents such attacks on firmware. This extension utilizes the current framework, allows device vendors to cheaply add the required functionality, and accounts for legacy hardware. It makes known remote attacks detectable and forces attackers to focus on hardware attacks, which—though possible—are difficult enough to justify the cost of secure bootstrap.

7 Acknowledgments

We would like to thank Greg Ganger, James Hoe, Adrian Perrig, and the anonymous reviewers for their comments. James is supported in part by a NDSEG Fellowship, which is sponsored by the Department of Defense.

References

[1] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[2] Arm storage: Seagate-Cheetah family of disk drives. http://www.arm.com/markets/armpp/462.html.

[3] J. Davidson. Chips to crack Xbox released on internet. *Australian Financial Review*, page 16 (Computers), 21 Jun 2003.

[4] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[5] LinuxBIOS. http://www.linuxbios.org.

[6] Myricom home page. http://www.myrinet.com.

[7] J. R. Rao and P. Rohatgi. EMpowering side-channel attacks. Technical Report 2001/037, IBM, 2001.

[8] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th Usenix Security Symposium*, August 2004.

[9] T. Smith. Warner attempts to out-hack DVD hackers. http://www.theregister.co.uk/content/2/13834.html, Sep 2000.

[10] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies*, San Francisco, CA, Mar 2003.

[11] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM Press, 2003.

[12] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 168–177. ACM Press, 2000.

[13] The Trusted Computing Group: Home. http://www.trustedcomputinggroup.org.

[14] The Trusted Computing Group. *TPM Main: Part 1 Design Principles*, Oct 2003.

[15] The Trusted Gomputing Group. *TCG PC Specific Implementation Specification*, Aug 2003.

[16] U.S. National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules*, Jan 1994. FIPS PUB 140-2.

[17] R. Wojtczuk. Defeating solar designer's non-executable stack patch. http://www.insecure.org/sploits/non-executable.stack.problems.html, Jan 1998.