Trusted Disk Loading in the Emulab Network Testbed

Cody Cutler

University of Utah, School of Computing
ccutler@cs.utah.edu www.emulab.net

Abstract

Network testbeds are critical for systems research but can be problematic due to their complex nature. Testbeds like Emulab allocate physical computers to users for the duration of an experiment. During an experiment, a user has nearly unfettered access to the devices under his or her control. Thus, at the end of an experiment, an allocated computer can be in an arbitrary state. A testbed must reclaim devices and ensure they are properly configured for future experiments. This is particularly important for security-related experiments: for example, a testbed must ensure that malware cannot persist on a device from one experiment to another.

Physical testbed nodes can be securely reconditioned in a scalable, maintainable way by making use of the Trusted Platform Module (TPM) and through adherence to a strict network boot protocol. This thesis presents the TDLS that we have implemented for Emulab. When Emulab allocates a PC to an experiment, the TDLS ensures that if experiment set-up succeeds, the PC is configured to boot the operating system specified by the user. The TDLS uses the TPM of an allocated PC to securely communicate with Emulab's control infrastructure and attest about the PC's configuration. The TDLS prevents state from surviving from one experiment to another, and it prevents devices in the testbed from impersonating one another. The TDLS addresses the challenges of providing a scalable and flexible service, which allows large testbeds to support a wide range of systems research. We describe these challenges, detail our TDLS for Emulab, and present the lessons we have learned from its construction.

1 Introduction

Like other fields, experimentation is critical to systems research. Since the ideal testing environment for some experiments is hundreds of computers which must be connected in a specific network topology, experiment creation is hard. The cost of creating such an experimental environment is restrictively high, both in terms of hardware and man power. Furthermore, it is not clear that investing in the creation of this system would be worthwhile since early experimental results may advise against pursuing the current work further.

To aid us in this problem of complex and costly experiment creation, researchers use what are known as "network testbeds." A testbed is a network of computers (also called "nodes") with supporting software that enables the experimenter to easily create arbitrary virtual network topologies with physical nodes. The experimenter is then given full access to the nodes to do as they see fit.

Network testbeds are generally designed to help users create test environments that are *realistic*. That is, in order to produce practically useful results, experimenters often want to create test environments that are like true deployment environments with respect to the properties of interest to a test. In addition, for systems-level research, testbed users often need a great deal of *freedom* within the environments they create. A systems researcher may need to install custom operating systems, use nonstandard network protocols, or perform other administrator-like tasks.

To support both realism and freedom, testbeds like Emulab are designed to allocate actual physical devices as well as virtual devices to users [21]. For the duration of an experiment, a user has exclusive and essentially complete control over the devices that are allocated to him or her. At the end of an experiment, the user releases the devices back to the testbed, and the testbed must reclaim them. Virtual devices are straightforward: they can simply be destroyed by the testbed. Physical devices, however, must be recovered so that they can be usefully and safely allocated to another user in the future. Since physical devices are composed of many pieces of hardware and those pieces often have their own persistent storage of some kind, it is difficult to be confident that a device's state has been completely reset.

In this thesis, we describe the system we designed, implemented, and use in production at Emulab which securely recovers and "reconditions" physical devices that support secure remote attestation. This includes most modern PCs, which contain Trusted Platform Module (TPM) hardware [19]. Emulab regains control over these devices in a trustworthy manner through a protocol rooted in the TPM of each device. Once a node is brought under Emulab's control, Emulab prepares the node for a new experiment by loading the node's disk with contents chosen by the experiment's creator. Emulab's *trusted disk-loading system* (TDLS) is responsible for both of these steps. In addition to preventing unwanted state from persisting on a device from one experiment to the next, the TDLS protects Emulab against other attacks that would misconfigure the devices allocated to an experiment.

When Emulab creates an experiment environment for a user, the TDLS ensures that the allocated PCs are configured to boot the user-specified disk images. The TDLS does *not* guarantee that the software within a disk image is "secure" by any standard, nor does it protect a PC from attacks during the execution of an experiment. Indeed, testbed users often want to install and study insecure software. The job of the TDLS is to set the *initial states*—disk contents and boot data—of the PCs allocated to an experiment as specified by the experiment's creator. If the PCs cannot be so configured, the TDLS will cause the creation of the experiment to fail.

Emulab's TDLS addresses several challenges in providing a secure node-configuration service for a large testbed that supports systems-level experiments on physical devices. The first is merely to regain control over physical nodes as they are released from experiments. An experiment may leave a device in an arbitrary—or even dangerous—state. The TDLS regains control through a combination of power control, remote attestation, and cryptographically secure network protocols. A second challenge comes from the size and diversity of the testbed. The Utah Emulab site contains hundreds of PCs. In addition, the site provides dozens of standard disk images that may be loaded onto those nodes, and users are free to create their own. The TDLS is therefore designed to require little or no administrator action when new devices or disk images are added to a testbed. A third challenge is to repel network-based attacks against the TDLS. This includes preventing a device from pretending to be Emulab's trusted control server, and preventing devices in untrusted states from initiating or rejoining the trusted disk-loading protocol. These issues are addressed through the design of the TDLS protocol and careful handling of the TPM.

The first contribution of this thesis is the identification of two design principles for building distributed, scalable, and secure boot-chains. These principles are: (1) *Fine-grained measurement* and *frequent reporting* of the boot chain allows for early error detection and attack surface minimization; (2) *Partitioning the static components from the frequently updated components* of the boot chain minimizes maintenance costs due to code evolution. The second contribution is the design and implementation details of the trusted disk-loading system we created for Emulab. In addition, this thesis summarizes the "lessons learned" from the development of our TDLS.

We will first cover the necessary background by describing Emulab and the capabilities of the Trusted Platform Module in section §2. In section §3, the challenges and shortcomings of Emulab's standard disk reloading system are analyzed and our threat model is given. The design of the Trusted Disk Loading System, how it is able to remotely verify a node's integrity, and how most the initial setup can be automated are detailed in section §4. Section §5 argues that similar providers can benefit from the TDLS and it's design principles and section §6 covers our lessons learned.

2 Background

Our Trusted Disk Loading System was designed for Emulab [21] with the Trusted Platform Module [19] being a core component. The properties of the Emulab environment greatly influence the design decisions of the TDLS and set this work apart from others.

2.1 Emulab

Emulab is a well-known network and distributed-systems testbed that emphasizes realistic experimentation by providing its users with physical hardware. Users do not have physical access to the testbed; instead, users communicate with Emulab and its resources remotely via the Web, SSH, and other Internet protocols. The testbed is managed by a trusted control server, called *boss*.

When a user creates an *experiment*, Emulab allocates physical resources—real PCs running real OSes and connected by physical switching infrastructure—to the experiment. When a machine (*node*) is allocated to an experiment, the creator of the experiment controls all aspects of that node: e.g., what operating system is running, what applications are installed, and how the nodes are interconnected with each other. This control extends beyond experiment set-up. Because a user has "root" access to his or her nodes, he or she can install and remove software at any time during an experiment.

Emulab is also a shared facility. Its physical cluster is space-shared: at any given time, multiple independent experiments may be taking place, each in its own allocation of physical resources. Emulab isolates experiments so that they cannot observe or interfere with each other. Emulab's resources are also time-shared. A node that is allocated to one user's experiment at a particular point in time will be dedicated to a different user's experiment for an entirely different purpose at a future time.

2.2 The Trusted Platform Module

A TPM is a stand-alone tamper-resistant microcontroller that is a standard component of many current desktop

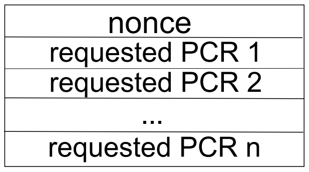

Figure 1: A high-level layout of quotes. Quotes are used in *remote attestation* and are signed by the TPM after they are filled in.

and server machines. In general, it improves security by making secret encryption keys available for use without ever divulging the keys. It can also guarantee that specific pre-determined keys are only usable when the PC is judged to be in a certain state. The capabilities we rely on are (1) its ability to provide secure key storage and (2) its ability to securely attest to the measurements of software.

Every TPM contains a *Storage Root Key* (SRK) that is created before the TPM can be used and never leaves the chip. The TPM cannot be used before the SRK is created because all other generated keys are encrypted with the SRK ensuring that no key ever leaves the chip in an unencrypted form. A TPM can generate other asymmetric keys and encrypt them with the SRK so that they can only be used by the TPM that generated them. The encrypted keys are then exported from the TPM and can be stored anywhere. For safety, it is not possible to use the SRK directly. A TPM also includes some number of *Platform Configuration Registers* (PCRs). A PCR cannot be written with arbitrary values (it would provide no security if it were directly writable as its current value would be meaningless); instead, a PCR can only be *extended*, an operation that stores a SHA-1 hash of the current PCR value concatenated with a new value. A PCR value is called a *measurement* and is a secure hash of some piece of state on the machine.

An *attested boot* of a machine causes each stage of the boot process, starting with the BIOS, to measure the next stage of the boot chain into a PCR. The effect is that, at any stage of the bootstrap, there is a unique set of PCR values that attest to the current state of the machine. By remotely comparing this set of values against a precomputed set of correct values, one can be assured that the machine is in a particular state.

To securely transfer a set of PCR values to a remote machine, the TPM supports a *quote* operation. The quote operation requires a TPM-created *Attestation Identity Key* (AIK), the indices of the PCR registers whose values are wanted, and a nonce (shown in Figure 1). The TPM creates a list of the desired PCR values combined with the nonce, hashes it, and signs the hash with the AIK. This signed hash is called a quote. The quote is then returned to the remote machine, which verifies the signature and checks the PCR values.

TPM-supported remote attestation in no way *prevents* tampering with the boot path: it only makes it possible for an outside party (Emulab) to reliably *detect* any tampering that does occur. As such, it is just one element of providing the TDLS.

2.3 Related Work

Emulab's TDLS establishes *only* the initial condition of a physical PC that has been allocated to an experiment. This purpose distinguishes our TDLS from prior work that uses the TPM to implement secure bootloading, integrity guarantees, and execution services.

Emulab's TDLS is not "just" a secure bootloader; it is a large system that embeds a secure bootloader to implement a particular, staged, disk-loading protocol. The goal of a typical secure bootloader is to ensure properties of the "user-visible" OS being booted. In contrast, the task of the TDLS is only to ensure that a machine is *ready* to boot the user-visible OS, and is independent of the properties of that OS. The security (or insecurity) of the OS contained within an Emulab disk image is explicitly up to the experimenter. This is necessarily so: for security-related experiments, users must be allowed to install whatever OSes they choose.

This difference in purpose leads to differences in design. TrustedGRUB [17], for instance, uses the TPM to measure not only the binary of the kernel being loaded, but also individual files that are important to the system. In contrast, Emulab's TDLS is designed to load entire disk images, so measuring the kernel or individual files within a disk image is unnecessary.

Our TDLS uses a static root of trust for measurement (SRTM): i.e., measurement of a system's BIOS at boot time. The OSLO bootloader by Kauer [11] establishes a *dynamic* root of trust by using the "late-launch" features of AMD processors. This technique removes the BIOS from the trusted computing base, which is useful in principle and practice. We chose to use the SRTM in our TDLS. Unlike OSLO, TDLS incorporates an agent (boss) that can be trusted to demand quotes, verify them, and take corrective actions.

There is much work that aims to provide integrity guarantees to running operating systems. Examples include SecVisor [18], a hypervisor that protects a kernel against code-injection attacks; Livewire [8], an intrusion-detection system based on VM introspection; Terra [7], a trusted virtual machine monitor that protects virtual machines from each other and from the underlying platform; and rootkit-resistant disks [2], which prevent system files from being modified on a node's persistent store. Unlike these systems, our TDLS guarantees only the initial state of a user-chosen operating system. Our TDLS provides integrity at the start of a testbed-based experiment, not *within* a running experiment. Also, because Emulab provides users with unmediated access to physical devices, our TDLS avoids hypervisor- and VM-based approaches to ensuring the integrity of nodes.

Flicker [12] uses TPM and late-launch features to create trustworthy environments for code execution. Flicker allows code to be executed securely at essentially any time, whereas our TDLS is concerned with executing code only at node-configuration time. A possible future project would be to use Flicker to allow a testbed to securely monitor experiments over their full lifetimes.

Dunn et al. [4] designed and implemented a protocol which uses late-launch and the TPM to distribute and execute malware while never revealing the malware in an unencrypted form. The malware's behavior can be kept secret by encrypting the malware payload with a key that only the TPM has access to and by using late-launch to make execution observation infeasible. Their work uses the TPM *against* the owners of the hardware and their goal is secrecy while ours is primarily system integrity with secrecy being secondary.

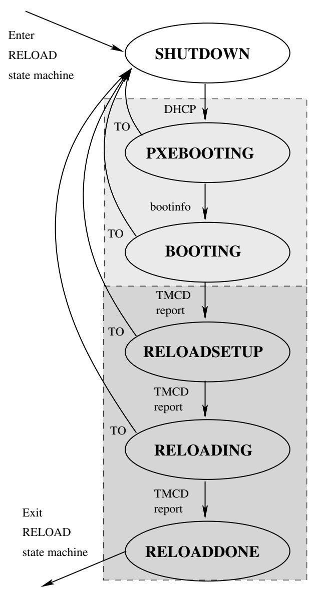

Figure 2: The Emulab node-imaging state machine.

3 Disk-Loading Challenges

Because devices are time-shared, Emulab must "wash" every physical device between the time it is released by one experiment and the time it is allocated to another. When a device is released from an experiment, however, it can be in a nearly arbitrary state. That state might contain malware that was the subject of the just-completed experiment, for example, and which must not be transferred to subsequent experiments. More egregiously, a malicious user might leave a device in a state that attempts to subvert Emulab's ability to recondition the node for subsequent uses. Consider a user who wants to transfer malware to subsequent experiments, or who wants to spy on future experiments. Such a user might install software that participates in Emulab's disk-loading protocol, but which does not actually reload the disk—thus allowing the malware or spyware to persist.

Because the testbed is space-shared, other experiments will be running while Emulab prepares nodes for a new experiment. These can also threaten Emulab's ability to recondition devices as they pass from use in one experiment to use in another. Malware within an experiment may try to migrate onto a reloading node, or a malicious user might try to hijack the disk-reloading process to propagate malware or spyware.

Emulab must ensure that the node-reloading process happens completely and correctly, without interference from (1) software on the node being reloaded or (2) any current experiments running on the testbed. Below, we describe Emulab's standard node re-imaging mechanism and how it addresses—or falls short of addressing—the threats outlined above. In Section 4, we describe how our new trusted disk-loading system addresses the shortcomings of the current, standard mechanism.

3.1 Standard Disk Loading

Emulab's standard disk-loading process involves two devices: the device being reloaded and Emulab's *boss* control server. As detailed below, the reloading device is expected to contact various services running on boss to obtain the data that it needs to configure itself for use in a new experiment. Testbed devices communicate with boss through a dedicated *control network*. At Utah's Emulab site, the control network connects all the PCs in the testbed to Emulab's central servers, including boss.

All PCs in Emulab are connected to power controllers (accessible to boss) and configured to boot only from the control network. These are the primary elements that ensure that Emulab can always gain control of a node, regardless of the state it is in. The boss server can power-cycle a node, and the node will then load a known first-stage boot program that allows boss to manage the node. Typically, boss directs the node to boot from its hard drive or to reload its hard drive.

Emulab uses a set of state machines [13] to track the states of nodes under its control. Events, including some that are "self-reported" by the monitored nodes, cause nodes to transition between states, which are tracked by boss. Trigger actions can be invoked when nodes enter states, and timeouts associated with each state allow corrective actions to be performed when a node misbehaves.

Figure 2 shows the state machine that controls disk imaging. In the first part (the upper shaded region), Emulab regains control of the node and boots it into an environment from which the node can be re-imaged.

The node is placed in the SHUTDOWN state and Emulab uses its power-cycling capability to force the node to reboot. The system or network card BIOS on the power-cycled node ensures that the control network interface will perform a PXE [10] boot. The PXE BIOS uses DHCP to obtain IP information for the node along with the name of the next-stage bootloader, which it downloads via TFTP and then executes. The DHCP request to boss causes a state transition to PXEBOOTING.

The next-stage loader is *pxeboot*, a custom Emulab boot program. The pxeboot loader talks to the Emulab *bootinfo* service to determine what the node should do next: boot from a partition on disk, or download an OS kernel and memory-based filesystem image (collectively known as an *MFS*) via TFTP. In the node re-imaging case, boss tells the node to download the *disk-loading MFS*. The bootinfo request to boss causes the second state transition into BOOTING.

After downloading the MFS, pxeboot hands off to the MFS kernel. A successful boot of the MFS is marked by the node explicitly reporting the RELOADSETUP state to boss via Emulab's *tmcd* service. This causes the third state transition and marks the start of the second stage of re-imaging (the lower shaded box): downloading and installing a disk image.

The node contacts tmcd to learn what image to load onto its disk and where to get that image. The node then reports its state as RELOADING, causing the fourth transition, and starts the *frisbee* disk-imaging client [9]. On successful completion of the reload, the node reports RELOADDONE, making the fifth state transition. Emulab reboots the node into a WAITING state for the next experiment and moves the node to a different state machine.

Notice that there are no explicit failure reports during any of the steps. Instead, each state has an associated timeout value. If the node fails to transition from the state in a timely manner, Emulab makes a timeout (TO) transition for the node—resulting in the node being rebooted and starting the process over again.

These mechanisms provide a robust process for rebooting and re-imaging nodes, but rely on certain properties of Emulab's infrastructure and assume a certain level of trust to function correctly. A breach of that trust during re-imaging could result in information leakage between experiments, or worse, propagation of malware.

3.2 Threats

Emulab's re-imaging system can potentially be subverted, leading to a node that is not re-imaged correctly. Consider, for example, the following threats to the disk-loading process.

1. Avoiding the network boot. Because Emulab provides "raw" access to nodes, the preceding experiment may have done anything to a node. A malicious experiment may have modified the BIOS configuration, for example, and arranged for the node to boot from its hard drive, not from the network. When Emulab power-cycles the node, the node will boot from disk. Software on the disk can then emulate a network boot, performing the necessary actions to force the state transitions shown in Figure 2. Emulab will believe it is talking to a trusted disk-imaging environment, when in fact it is not.

2. Hijacking the network boot. Even without BIOS modifications, a malicious node may still spoof Emulab. Emulab places all nodes being re-imaged in a common VLAN, to take advantage of the disk imager's multicast features. Unless all nodes in this VLAN are rebooted simultaneously, a lagging node could use a man-in-the-middle attack to interpose itself between boss and another node, and thereby subvert the imaging of that node. While a hijack is unlikely once a node starts using authenticated communication with boss, the initial stages (DHCP, TFTP, and bootinfo) are vulnerable, offering a window of opportunity to infect a node before it resumes communicating with boss.

3. Aborting the network boot. Even without a hijack, a malicious node might be able to interfere with the network-boot process of another device, so that the other device falls back to booting from its hard drive.

Even if a node successfully boots into the disk-loading MFS, there are attacks against the Emulab disk-imaging subsystem during the second part of the process (the lower portion of Figure 2). These attacks are detailed in earlier work [16] and are summarized below.

4. Modifying the transferred image. Because frisbee uses an unauthenticated, IP-based multicast protocol to distribute disk images, an adversary could transparently replace portions of an image: e.g., replace a section that contains the password file.

5. Corrupting the transferred image. Even if an adversary lacks knowledge about the content of an image, he or she can still inject data into the transfer, corrupting the resulting disk and preventing it from being used.

6. Observing the transferred image. IP multicast does not have any built-in limitations on group membership. Thus, any node may join a frisbee group and obtain a copy of an image being loaded by another user, which may contain sensitive data.

Emulab currently uses a variety of techniques to mitigate, but in most cases not eliminate, these threats. When a node enters the re-imaging path, all access to that node by previous users is revoked. This includes access via the serial console and network, so interactive attacks are eliminated. To help ensure that nodes reach the disk-loading

MFS, Emulab site operators typically password-protect the BIOS of their testbed PCs. Emulab's software enforces timeouts on a node's state as it transitions to the reloading MFS. Coupled with a method for client and server authentication, and judicious use of per-experiment infrastructure and firewalls, these strategies have proven adequate to protect Emulab sites from the actions of non-malicious users.

However, the overall strategy is ad hoc and provides insufficient guarantees for many types of security experimentation. It also does not take advantage of advances in technology—in particular, the increasing availability of TPM-enabled platforms.

4 Trusted Disk Loading

In this section, we describe how we make use of TPM technology to implement a scalable, *trusted disk-loading system* (TDLS) for Emulab. The full protocol is shown in Figure 3.

4.1 Using the TPM to Verify Node Boot

To support a secure boot path, we added the notion of *secure states* to Emulab's state-machine mechanism. Certain operations, such as fetching image-decryption keys, are allowed only while a node is in an appropriate secure state. A secure state can only be entered by providing a TPM-attested quote to Emulab's boss server, as described below. If the quote is incorrect, or if a timeout period passes, the offending node is placed into a special SECVIOLATION state.[1] When a node enters this state, an email notification is sent to the testbed operators and the node is powered off. Thus, any node that strays from the trusted boot path will be handled by the operators, and the amount of damage it can do is limited.

Figure 4 shows the state machine that drives the new TDLS. It is similar to the state machine shown in Figure 2, but has three additional states (described below) and new, secure states (indicated by double ellipses). Note that a timeout or error from *any* state results in a security violation.

Since we are using a static root of trust, a reasonable strategy would be to measure each component one after another and attest to boss using one quote at the end of the protocol. A benefit of doing it this way is that no boot-chain component other than the final stage needs to have the capability of remote attestation—a capability that requires a significant amount of code. We, however, chose to report quotes more frequently (not only during

[1]There is currently a single SECVIOLATION state in the TDLS. We are planning to implement multiple violation states as this will help operators distinguish between certain security violations (bad quotes) and potential ones (timeouts).

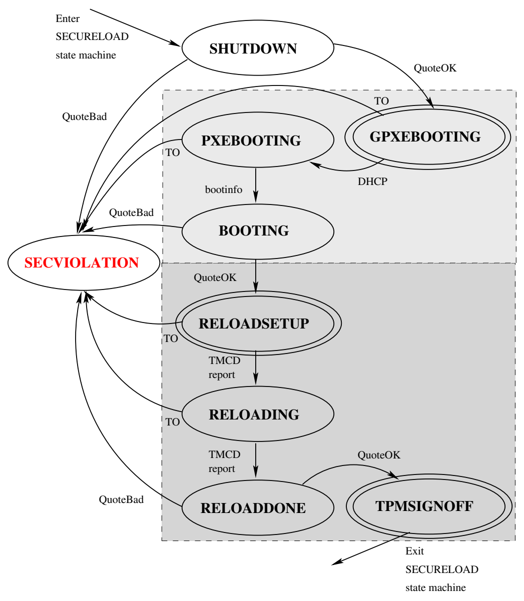

Figure 4: The Emulab TDLS state machine has three new states (GPXEBOOTING, TPMSIGNOFF, and SECVIOLATION). A quote is required to enter the secure states shown with double ellipses.

the final stage) since this way we can detect exactly when any particular piece of the chain is not what we expect. This early detection allows us to minimize the time that unexpected (and possibly malicious) code runs on the nodes.

Another possible design simplification would be to measure all the boot-chain components into a single PCR. Measuring all the components into a single PCR not only tracks the contents of the components but also ensures that they are measured in a particular order. The downside of this design is that it is not conducive to code evolution—if any stage of the boot chain is ever modified, the acceptable PCR values known to boss must be updated not only for the modified stage itself but also for every stage after the modified stage.

4.1.1 Step 1: BIOS Boot

The first stage in the boot process is the BIOS boot. At power-on, immutable BIOS code measures the rest of the BIOS and its configuration parameters with the TPM. The first sector of the boot device is then measured before control is passed to it.

To support trusted booting, we modified the way that Emulab uses PXE. PXE typically is loaded from ROM on a host's network interface. However, current implementations do not allow the measurement of this ROM by the BIOS, and we cannot modify the ROM to measure the next boot stage. This creates a break in the secure boot chain. To avoid this problem, rather than use the

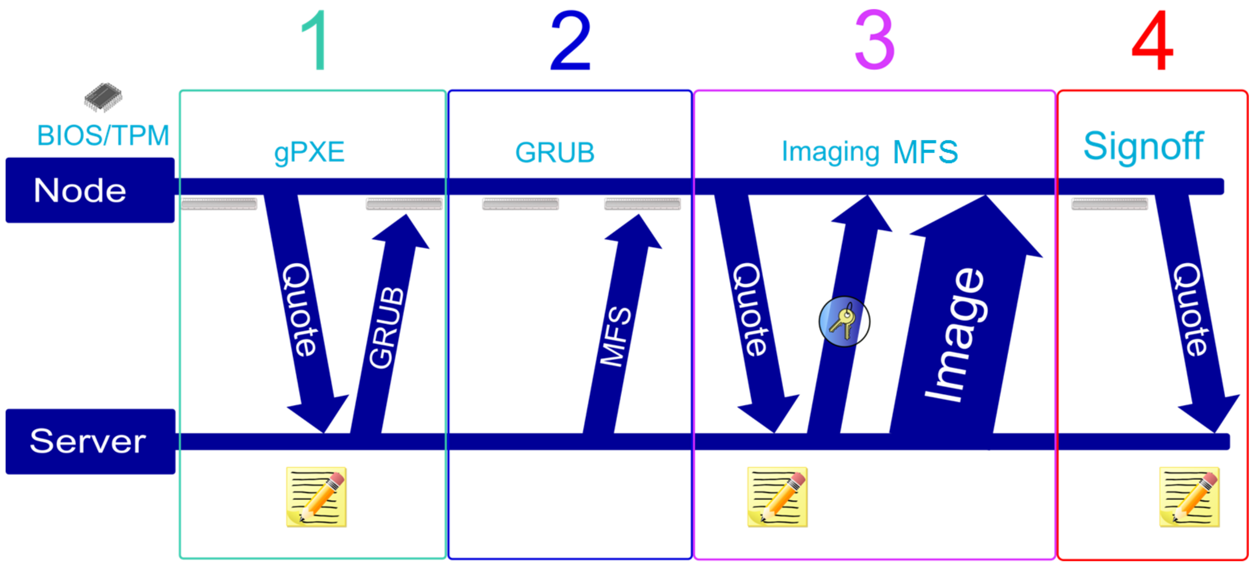

Figure 3: The full trusted disk loading protocol. Arrows represent network communication, grey bars represent PCR extensions, and pencils with paper represent quote verifications.

NIC's PXE ROM, we perform the first-level boot from a write-protected USB flash device, which the BIOS can measure. That code then performs the PXE boot.

The previous user of a machine may still modify the BIOS (Section 3.2, Threat 1) or otherwise interfere (Threat 3) so that the machine does not boot from the USB device. In these cases, the BIOS and first-level bootloader measurements will be incorrect. Emulab's boss server will detect this when the node attempts to check in, or when a timeout period has passed.

Enabling the TPM is all that is necessary for the BIOS to be measured on boot as the TPM standards require it [19]. On our systems, this is done by enabling the TPM in the BIOS configuration utility. The early boot code that measures the BIOS is also guaranteed to be immutable by the specifications.

BIOS configuration is problematic for a testbed with many nodes since it is usually machine dependent and not easily done in batch. We discuss how we deal these limitations in §4.3.

4.1.2 Step 2: Attested Emulab Network Boot

The flash device contains a modified version of the gPXE [5] bootloader. Our version of gPXE is TPM-aware and can establish TLS sessions with network-based services. We can thus use gPXE to perform a *measured network boot* as described below.

gPXE uses the PXE and DHCP protocols to acquire an IP address from the Emulab boss server. It then uses TFTP to download the next stage as specified in the DHCP information. The downloaded next stage is measured, and the result extended into a PCR. There is no assurance at this point that the DHCP and TFTP packets are not spoofed or tampered with (Threat 2). If they are, however, this will be discovered in the next step.

After measuring the next boot stage, gPXE attempts to perform the first secure state transition (to GPXEBOOTING, shown in section 1 of Figure 3). This involves the node sending a TPM-attested quote to boss. To make this quote, the node requests from boss its AIK, the PCRs to return, and a nonce. Using the AIK and nonce, gPXE requests a quote from the TPM. The resulting quote is unforgeable, and the nonce prevents replay of previously generated quotes (Threat 2).

To ensure that the node is talking to the real boss (i.e., that it was boss that responded to the initial DHCP request), this request is sent via a TLS session. The boss server is authenticated via a certificate embedded in the boot image—in this case, gPXE. Because the certificate is part of the gPXE image measured by the BIOS, we can be assured that any tampering with the certificate will be discovered. Note that boss needs no strong authentication of the reloading node during this exchange, since the AIK being returned is only usable by the node that generated it.

Emulab's boss server verifies the quote, comparing the returned PCR values with a set of measurements that have been precomputed and stored in Emulab's database. The precomputed measurements describe a correct boot of a particular node, through the BIOS and gPXE, and having measured gPXE's next stage. A correct quote allows the node to continue booting. An incorrect quote causes Emulab to place the node into the SECVIOLATION state and power it off.

The implementation is complicated by the fact that only the first sector of the boot device is measured by the TPM since the first sector is all that is loaded by the BIOS on x86 bootup. It is up to this single sector of code to load and measure the rest of gPXE from the boot device. Therefore this code must be written carefully to ensure that we measure gPXE in its entirety, otherwise there will be a break in the secure boot chain.

Because our version of gPXE needs to be capable of many various TPM operations, we use libtpm [1] instead of the BIOS TPM interrupts directly. libtpm gives low-level access to the TPM but also depends on the OS to provide a driver for the TPM and to make it available to userspace via device nodes. To resolve this dependency, we took the TPM driver code from Linux and adapted it to gPXE with significant modifications.

We use the PolarSSL library [1] for our TLS and cryptographic needs. It is well suited for gPXE since it is designed specifically for use in embedded systems. There were some complications, however. In order to set up a secure TLS session, it is necessary to know the current time. We used the real-time clock code from Linux and assimilated it into gPXE for use. PolarSSL also assumes the existence of the BSD constructs *socket()* and *connect()* and POSIX constructs *read()* and *write()* which do not exist in gPXE. Though these constructs are awkward to implement since gPXE is a cooperative multitasking system with one stack, they were added and easily interfaced with PolarSSL. We were also concerned that the random numbers generated by PolarSSL would turn out to be very predictable since PolarSSL is generally used in a more mature environment with device interrupts. To ease our minds, we setup an experiment to harvest the entropy generated by PolarSSL: a node was configured to boot our version of gPXE, connect to boss securely over TLS while dumping the entropy generated for the TLS handshake, and reboot to restart the process. We let the experiment run over night—enough for 245 TLS handshakes (our server class test node takes about 4 minutes to POST). In the end we found that not only were the randomly generated numbers for every TLS handshake unique, there were 7.9999 bits of entropy per byte over all gathered entropy (as reported by ent [20]). Had we found the generated entropy to be insufficient, we would have investigated the possibility of using the TPM's built-in random number generator.

An important design point is that the gPXE flash device contains no node-specific data, and is thus amenable to large-scale duplication. This is essential for managing a large set of machines. Every time the operators of the testbed change the version of gPXE in use, the flash devices must be physically replaced and new, known-good measurements of it must be taken. However, by design, the functionality of gPXE is limited. We expect that modifications to it will be extremely infrequent. The same is not true of later stages such as the Linux MFS - we separated out and measure these stages individually in order to easily adopt updated boot-chain components.

4.1.3 Step 3: Booting the Disk-Loading MFS

gPXE downloads and executes a version of the GRUB 2 bootloader [6] that we have enhanced to support communication with Emulab's boss and the TPM (section 2 of Figure 3. (In the secure boot path, GRUB replaces pxeboot.) Similar to the standard boot path, GRUB makes DHCP and bootinfo requests, triggering transitions to the PXEBOOTING and BOOTING states. Unlike the standard path, a failure or timeout results in a transition to SECVIOLATION.

We use the BIOS TPM interrupts in GRUB instead of libtpm because, unlike gPXE, GRUB doesn't need to perform any TPM operations besides measuring. There is a subtle difference in the measurements taken by GRUB compared to those taken by gPXE though: GRUB's disk subsystem fetches the given file's data on-demand while gPXE waits until the image is successfully loaded. This behavior makes it difficult to measure properly; reads covering the same file but of varying lengths will yield different PCR results.

Our solution to this problem is to read the entire file into a buffer when it is opened, measure it, and then return data from that buffer on future reads. This ensures that the given file will always have the same PCR value regardless of a change in storage media or the length of the reads.

As shown in section 3 of Figure 3, GRUB proceeds to download and measure the *secure disk-loading MFS*: a minimal Linux system that we have configured to disable all network listeners. The measurement is extended into a PCR, but not immediately reported to boss. After measuring and extending, GRUB transfers control to the Linux kernel in the MFS.

Before starting the disk-loading subsystem, the secure disk-loading MFS produces another quote for boss to check. This includes the updated PCR values that cover GRUB's measurement of the MFS. The quote process is identical to that performed by gPXE previously. It causes a secure state transition to RELOADSETUP if successful, or SECVIOLATION otherwise (Threat 2).

4.1.4 Step 4: Securely Reloading the Disk

Once a node has performed an attested boot into the disk-imaging MFS, we make use of prior work [16] that extends the Emulab disk-imaging system [9] to provide confidentiality, integrity protection, and authentication for images and their distribution (Threats 4, 5, and 6).

In our new TDLS, we improve upon that previous work by providing a trusted platform on which to run the disk-imaging client. The TDLS also uses the TPM to implement cleaner ways of (1) assuring node identities to the server and (2) distributing image-decryption keys.

To provide better node identification, for every node, we create a per-node certificate that is associated with a key pair created by each node's TPM. The certificate (including the public key) and TPM-encrypted private key are stored in the database on boss. When a node is running in secure disk-loading MFS, it acquires the certificate and encrypted key over an insecure channel. The key is loaded into the TPM, and whenever a TLS session is started with boss, the client certificate associated with the key is given to boss during the handshake.

This authenticated channel is used to pass the image-decryption key from boss to the node. It is important that this key not be released to a node until we are certain that (1) the node is the one it claims to be and (2) that it is running in a trusted environment. We now have both those assurances. At this point the node enters the RELOADING state and invokes frisbee to obtain the actual disk image.

4.1.5 Step 5: Signing Off

After the disk has been loaded but *before* handing off to the OS on the disk (RELOADDONE), the TDLS "invalidates" the PCR state so that the soon-to-be-booted OS cannot produce a quote using the state of those registers (section 4 of Figure 3). This prevents the loaded OS, which is *not* trusted, from participating in the TDLS protocol.

The TDLS makes an explicit hand-off using PCR 15 and a mandatory final quote to Emulab's boss server. This PCR is set to zero by a reboot (and only by a reboot), and the TDLS includes its value in all quotes to boss. A zero in PCR 15 is the boss-visible indicator that the node is executing the trusted disk-loading protocol.

To invalidate the PCR state, the secure disk-loading MFS extends a known value into PCR 15, which sets the PCR to a non-zero value. The MFS then produces and transmits a final quote, including this non-zero measurement, to signal that the trusted image load has completed (TPMSIGNOFF).[2]

4.1.6 Epilogue: Booting the Loaded Disk

The secure disk-loading protocol ends when the disk-loading MFS signs off. It still remains, however, to boot the node from the downloaded disk.

[2]Conceivably, the final quote could include a measurement of the final disk contents, as further protection against disk failures and malicious activity. We consider this impractical, due to the time needed to hash entire—and potentially very large—disks.

After sending its final quote, the MFS immediately reboots the node. This re-engages the BIOS boot (§4.1.1), which runs our modified gPXE (§4.1.2), which securely contacts the bootinfo service. In the TDLS protocol, bootinfo would tell the node to download the disk-loading MFS. Now, however, bootinfo tells the node to boot from its local disk—i.e., the image that was just downloaded. gPXE immediately invalidates the node's PCR state, transmits the "sign-off" quote to boss, passes control to GRUB (whose measurement was included in the "sign-off" quote) and proceeds to boot the on-disk OS.

Because disk loading and disk booting are separate, an attacker might try to subvert correct booting by modifying the node between these steps. We believe that such attacks are infeasible without physical access to the PC or compromising the Emulab software. Even when not in the TDLS, TPM-enabled nodes use the secure boot procedure up to the GRUB stage, which mutually authenticates the node and the true Emulab boss. Failure to boot through the secure path will be discovered through an incorrect quote or timeout.

4.2 Establishment and Maintenance

Because of the large size and dynamic nature of Emulab, the scalability and maintainability of new features must be analyzed. It is therefore essential to consider the tasks required to set up and maintain the TDLS.

The most labor-intensive one-time task is installing USB flash devices on all machines. Because the USB-based gPXE image contains no node-specific information, this is reduced to a task of physical replication.

Another task is the creation of the two per-node TPM-encrypted keys: the AIK used to produce quotes, and the TLS key (and certificate) used to authenticate a node to boss. These steps must be performed on the nodes themselves, when they are in a secure state. This can be automated, running a script in the gPXE environment and taking advantage of its ability to securely identify and communicate with boss.

Likewise, the correct values of the PCRs used in quotes must be produced for each node type and stored in Emulab's database. Assuming all nodes of one node type have the same BIOS version (as our nodes in Emulab do), the BIOS measurements will be identical across all nodes of the same node type. Additional values covering gPXE and later stages can be computed offline using the SHA-1 hash function [19].

Changes to the TDLS, or the addition of other trusted boot paths, require the collection of additional PCR values. As long as all are based on booting through gPXE, we can compute these values offline.

4.3 From Prototype to Production

The TDLS was designed to keep maintenance as low as possible. That being said, the one-time setup procedure of adding a node to the TDLS has multiple sensitive steps. The following steps are required to add a node to the TDLS: (1) Generate a TPM AIK; (2) Generate a TPM TLS key; (3) Generate a signed TLS certificate using the TLS key; (4) Insert both keys and the certificate into Emulab's node database; (5) Configure the node's BIOS to boot the gPXE dongle; (6) Configure Emulab to send TPM-aware GRUB2 as the second stage; (7) And finally, attach the gPXE dongle to the node.

Most of these steps are not trivial. Furthermore, they must be done on every node that is to be added to the TDLS. We have over 100 nodes that will be added to the TDLS at Emulab Utah; manually going through this setup process would be extremely tedious and error-prone. To remedy these concerns, we wrote a helper system which automatically acquires an Emulab node equipped with a TPM, configures the BIOS, initializes the TPM for use, generates all the necessary keys, and then releases the node. This system does not touch the Emulab database though; the database updates are left up to us humans so we can carefully verify the keys and certificates before inserting them into the database.

Using standard Emulab tools, the helper system first gets exclusive access to the node by creating and placing it in a new Emulab experiment. Once the experiment is created, it reboots the node and runs a machine-dependent Python script which uses Expect to enter the BIOS, enable the TPM, and change the first boot device to the USB dongle. The helper system then forces the node to boot a Linux kernel with a special memory file-system. This memory file-system contains a suite of TPM helper tools (which we also wrote) that make the creation of TPM keys easy. The node executes an embedded script, runs they key creation tools, and copies the generated keys to an FTP server where they are later harvested. Finally, the helper system destroys the experiment and releases the node.

This software was well worth the implementation time; it successfully configured and harvested the keys from 92% of our nodes. The remaining nine were done ”by hand.”

To simplify key harvesting in the future, we are considering the following improvement: because our version of gPXE already has the capability to securely communicate with boss and libtpm support, gPXE could be modified so that, on first boot, it would automatically discover that it needed to generate the prerequisite keys, generate them, pass them to boss to store them, and continue on with the TDLS protocol. This would be done when a new node is being installed into the TDLS before any users are given access to it, reducing the chances that an attacker could interfere. With these modifications, adding a new node to the TDLS would be reduced to configuring the BIOS and simply plugging in the gPXE USB dongle.

5 Applicability

The TDLS is important for containing the effects of experiments, and thus important for a testbed that seeks to provide isolated, reliable, and secure services to its users. Our TDLS design can inform the creation of node-configuration services for network testbeds, businesses that loan physical nodes (such as Rackspace [15]), and other, similarly managed networks. The xCAT 2 toolkit [22], for example, supports user-provisioning of physical PCs in clusters. Although our TDLS was built for Emulab, we believe our implementation could be adapted for use in other testbeds that provide users with unmediated access to TPM-enabled devices.

The TDLS design can be beneficial to systems that have the following properties: (1) users have "root" access to physical PCs; (2) PCs are serially reused over time for many experiments and users; (3) and PCs are reconditioned for experiments using a network server (boss);

Although many cloud and grid platforms allow users to allocate virtual machines only, our TDLS could nevertheless be useful to the *providers* of cloud and grid services who must manage the underlying physical resources.

6 Lessons Learned

Our trusted disk-loading system for Emulab has recently been enabled for production use. During the design and implementation of the prototype and the subsequent push to production, we identified these principles that we believe are general. These "lessons learned" could be usefully applied to similar, trustworthy node-configuration systems for security-conscious testbeds.

Separate changing from unchanging components. For a large testbed like Emulab, scalability is critical. The known-good measurements must be reestablished for every component that is added or modified. Stages of the boot chain are also stored on per-node media making updates to those stages even more costly. Total maintenance due to boot-chain updates can be reduced by dividing the static parts from the often-evolving parts making the TDLS as a whole easier to maintain and more readily extensible.

Check in frequently to minimize damage. While it would be possible to check boot state only at the end of the device "reconditioning" process, the timeout in some states may be very long. If attestation is performed only at the end of the node-configuration process, it is not possible for a testbed to detect nodes that have left the trusted boot path until all timeouts have expired. Performing attestations frequently, at a number of boot stages, helps to minimize the window of opportunity for attackers.

Make an explicit transition to untrusted code. The TDLS relies on a trusted boot path and disk loader to load and boot *untrusted* user code. TPM-based attestation can make the transition point visible and prevent user code from impersonating trusted code.

Write-protect boot-chain stages when possible. While it provides no additional security since modifications to the USB dongle would be detected by the PCR measurements, write-protecting the media makes the TDLS more *maintainable*. The USB dongles are left plugged into the testbed nodes which are constantly being imaged with various operating systems and are passed from one user to the next; dongle modifications are likely, whether accidental or malicious. Preventing these modifications frees us from the task of finding, re-imaging, and replacing modified dongles.

7 Future Work

One nice feature of using a state machine to track node progress is that it provides enough flexibility to control exactly how to respond to different node behaviors. For example, it is possible to create a separate and distinct state in which to place all nodes that timeout during one specific transition in the TDLS protocol. This variable granularity allows fine-grained node management. In the future, we would like to have two separate states for the problematic nodes instead of one (SECVIOLATION). In one state we would put all nodes that timeout and, in the other, all nodes that produce incorrect quotes. This distinction is useful because nodes that timeout probably need administrator inspection less urgently than a node that has a corrupt BIOS or is receiving modified software.

We have also considered the possibility of using OpenFlow [14] in conjunction with a node's state machine in order to manipulate the surrounding network paths or to give/revoke access to network resources.

Other future work includes the possibility of allowing Emulab users to upload and securely boot their own MFSes. Sometimes imaging the entire disk is not necessary and an MFS provides enough functionality for an experiment.

8 Conclusion

We have presented a new *trusted disk-loading system* for Emulab. The task of the TDLS is to gain control over a physical PC, which may be in a nearly arbitrary state, and set its state as directed by Emulab. In particular, the TDLS is responsible for establishing the initial condition of a PC that has been allocated to an experiment. We have identified ways in which Emulab's standard disk-loading system can be subverted by an attacker, thus causing initial conditioning to fail. Using the TPM of modern PCs, our TDLS addresses these threats to reliable disk loading. Our early experience has confirmed that our system is practical to deploy and maintain at the scale of hundreds of physical PCs, and in the face of constant testbed evolution.

Acknowledgments

Thanks to Mike Hibler, Eric Eide, and Robert Ricci for co-authoring the original paper [3], "Trused Disk Loading in the Emulab Network Testbed", Proceedings of the *Third Workshop on Cyber Security Experimentation and Test* (CSET'10), Washington, DC, August 2010. We also thank Ryan Jackson for his assistance with gPXE and the secure disk-loading MFS.

This material is based upon work supported by the National Science Foundation under Grant No. 0709427.

References

[1] Paul Baker. PolarSSL. `http://www.polarssl.org/`.

[2] Kevin R. B. Butler, Stephen McLaughlin, and Patrick D. McDaniel. Rootkit-resistant disks. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 403–415, October 2008.

[3] Cody Cutler, Mike Hibler, Eric Eide, and Robert Ricci. Trusted disk loading in the Emulab network testbed. In *Proceedings of the 3rd Workshop on Cyber Security Experimentation and Test (CSET)*, August 2010.

[4] Alan M. Dunn, Owen S. Hofmann, Brent Waters, and Emmett Witchel. Cloaking malware with the Trusted Platform Module. In *Proceedings of the 20th USENIX Security Symposium*.

[5] Etherboot Project. Etherboot/gPXE Wiki. `http://etherboot.org/`.

[6] Free Software Foundation, Inc. GNU GRUB. `http://www.gnu.org/software/grub/`.

[7] Tal Garfinkel, Ben Pfaff, Jom Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 193–206, October 2003.

[8] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, February 2003.

[9] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with Frisbee. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 283–296, June 2003.

[10] Intel Corporation. Preboot execution environment (PXE) specification version 2.1, September 1999. `http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf`.

[11] Berhhard Kauer. OSLO: Improving the security of trusted computing. In *Proceedings of the 16th USENIX Security Symposium*, August 2007.

[12] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozak. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 315–328, April 2008.

[13] Mac G. Newbold. Reliability and state machines in an advanced network testbed. Master's thesis, Utah, May 2005.

[14] Open Networking Foundation. OpenFlow. `http://www.openflow.org`.

[15] Rackspace Hosting. Rackspace homepage. `http://www.rackspace.com/`.

[16] Robert Ricci and Jonathon Duerig. Securing the Frisbee multicast disk loader. In *Proceedings of the Workshop on Cyber Security Experimentation and Test (CSET)*, July 2008.

[17] Marcel Selhorst, Christian Stüble, and Oliver Altmeyer. TrustedGRUB, version 1.1.4, November 2009. `http://sourceforge.net/projects/trustedgrub/`.

[18] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 335–350, October 2007.

[19] Trusted Computing Group. Trusted platform module. `http://www.trustedcomputinggroup.org/developers/trusted_platform_module`.

[20] John Walker. ENT - a pseudorandom number sequence test program. `http://www.fourmilab.ch/random/`.

[21] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, December 2002.

[22] xCAT – Extreme Cloud Administration Toolkit. `http://xcat.sourceforge.net/`.