UNIVERSITY of FLORIDA

IoT Security and Privacy
Secure Bootstrapping

YIER JIN

UNIVERSITY OF FLORIDA

EMAIL: YIER.JIN@ECE.UFL.EDU

SLIDES ARE ADAPTED FROM PROF. XINWEN FU @ UCF/UMASS

Learning Outcomes

Upon completion of this unit:

- Student will be able to explain trusted boot, which can be used for IoT devices to protect the firmware/OS
- Students will be able to explain secure boot, which can be used for IoT devices to protect the firmware/OS
- Students will be able to explain TPM and its usages in secure storage and crypto accelerators
- Students will be able to explain remote attestation, which verifies the genuineness of a system
- Students will be able to explain tamper resistant/proof/response hardware and its usage

Prerequisites and Module Time

Prerequisites

- Students should have taken classes on operating system and computer architecture.
- Students must have taken crypto and know how public key crypto and symmetric key crypto work.
- Students should have mastered programming Raspberry Pi.
- Students should know basic concepts of networking.

Module time

- 4-hour lecture
- Two-hour homework

Main References

[1] Bryan Parno, Jonathan M. McCune, Adrian Perrig, Bootstrapping Trust in Modern Computers, Springer Publishing Company, Incorporated ©2011

[2] Bryan Parno, Bypassing Local Windows Authentication to Defeat Full Disk Encryption, Black Hat Europe 2015

Outline

A Challenge for System Security

How can you trust a system?

Bootstrapping Trust

How to convey information about a computer's current execution environment to an interested party?

- Enable a user to verify that her computer is free of malware or a remote web server will handle her data responsibly (?)

Requires some foundational root of trust

Trusted Computing

- Trusted Platform Module (TPM), deployed on over 350 million computers

Trust Human vs Trust Computer

Trust in a human is often based on identity

Even if a human is trusted, can his computer be trusted?

- Authentication is to authenticate the identity of Alice behind a computer
- Not that Alice's laptop is free of malware

How do we trust a computer?

Computer’s State

What is a computer’s current state?

- Hardware configuration
  - Certification from the computer’s manufacturer
- Running code
  - How to identify code? It is the code that you expect, not code that has been maliciously changed

Code Identity

Code identity can be achieved through a cryptographic hash over

- #1 Software
- #2 Inputs
- #3 Libraries
- #4 Configuration files

The calculation of the code identity is often called measurement

For bootstrapping trust, we need to record at least,

- Identity of the code currently in control of the system.
- Identity of any code that could have affected the security of the currently executing code.

When and Who Performs Measurements

When to measure a software

- before it starts

Software that starts in the order of S_1, S_2, …, S_i, …, S_n, S_{n+1}

- S_i is running and measures S_{i+1}, before S_{i+1} starts
- These measurements form chain of trust

Who (or what) measures the first software (S_1)

- Who or what measures S_1 is the root of trust, for example, hardware-based root of trust.
- Can a running code self identify/measure itself and submit the measurement?

Trusted Boot

A hardware-based root of trust initiates the chain of trust

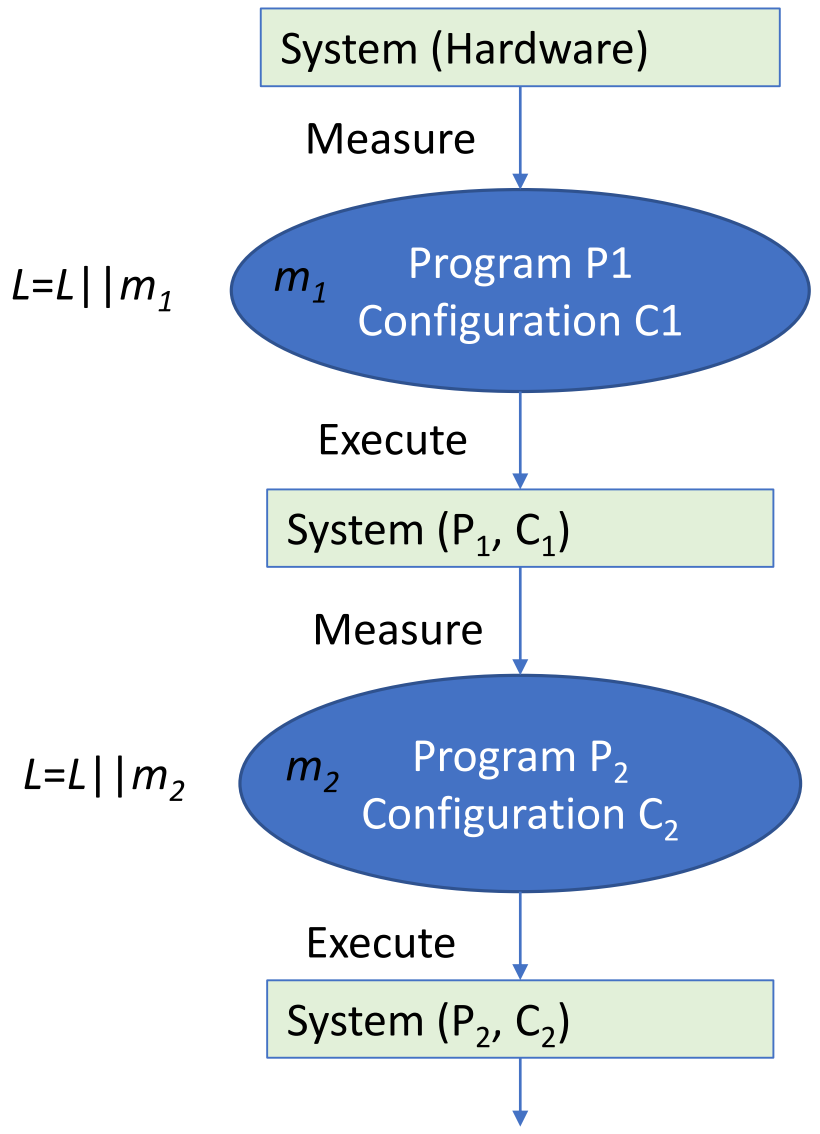
- Measures the initial BIOS code and runs it

BIOS then measures and runs bootloader,

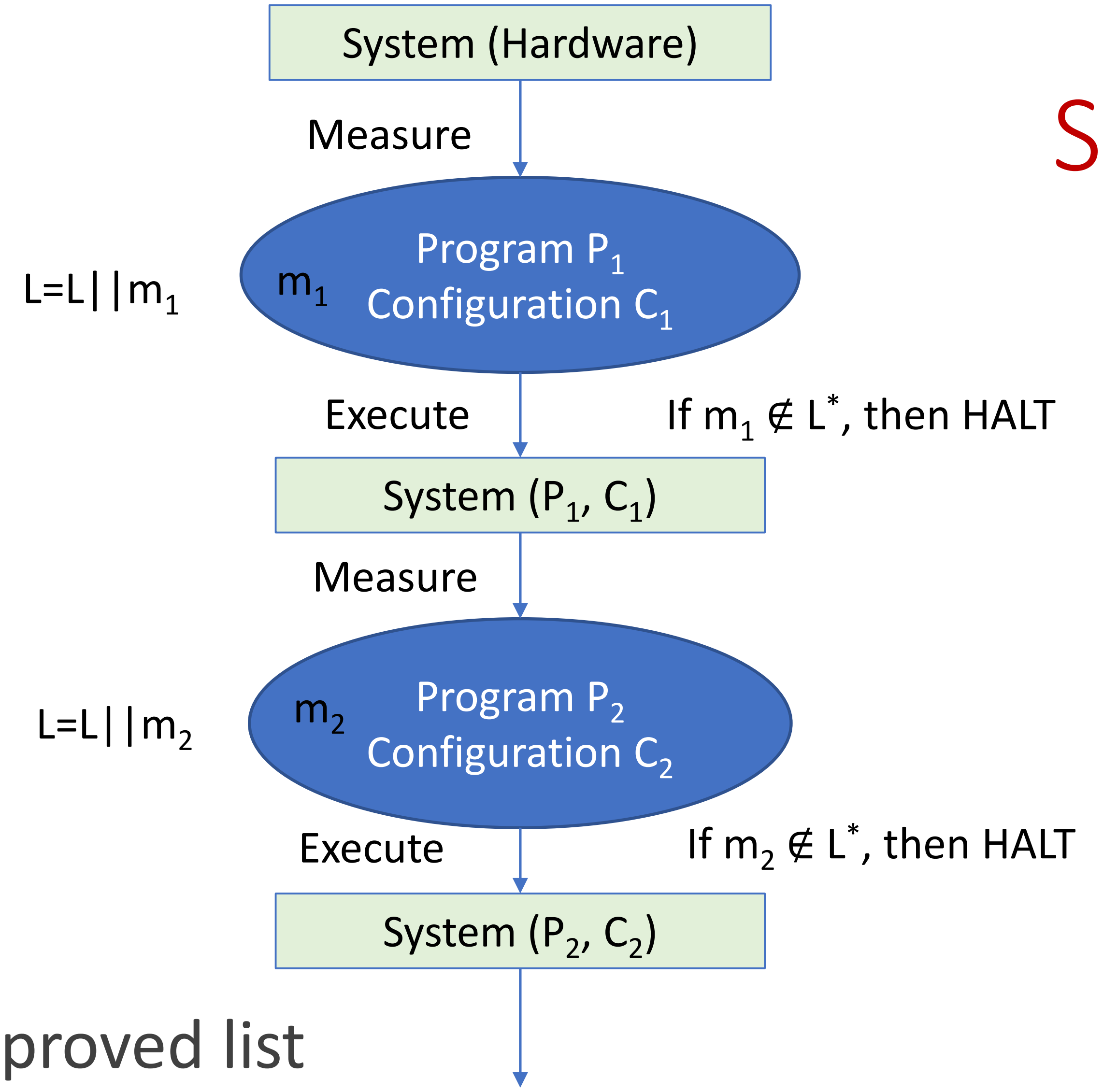
Bootloader measures and runs the operating system (OS).

OS can measure each application

No enforcement of what to do if a measurement is not right



Trusted Boot



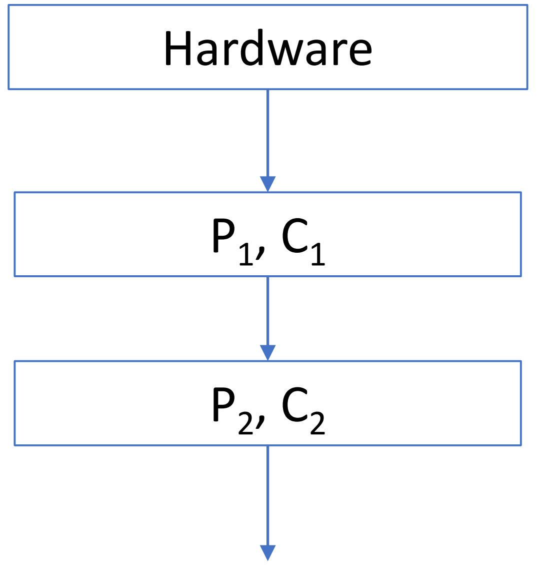
Secure Boot

L* - an approved list

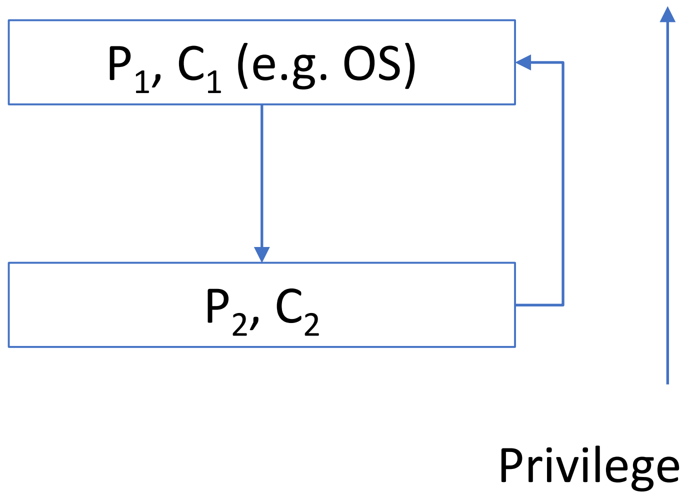
Secure Measurements

What if measurements be erased by malicious software?

Execution handoffs



a. One-way handoff



b. Temporary handoff

Two Attacks against the Transfer

Handoff attack - Trusted software unintentionally executes malicious software

Privilege escalation attack - less privileged code exploits privileged code to

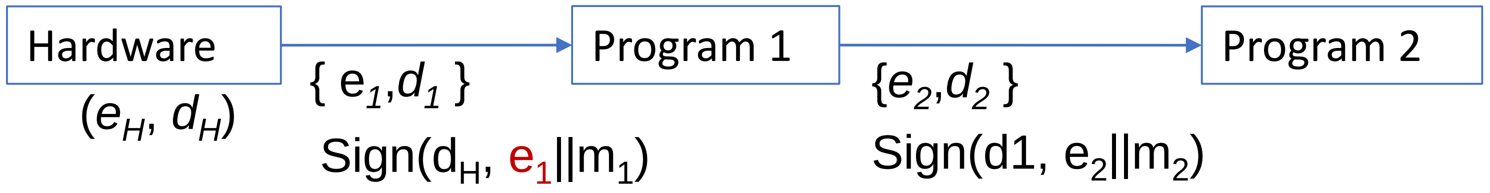
- access secrets of privileged code,
- erase the record of the malicious code’s presence, or
- create fake records of other software.

Certificate Chains Securing Measurements

Before P_{i+1} runs P_i

- Generates a public key pair for P_{i+1} - (e_{i+1}, d_{i+1})
- Creates a certificate containing P_{i+1} and a measurement m_{i+1}.
- Erases its own secrets d_i
- Loads the new software, providing the new keypair and certificate as inputs.

The hardware's public key can be used to verify the certificate chain



What Certificate Chains Can Do?

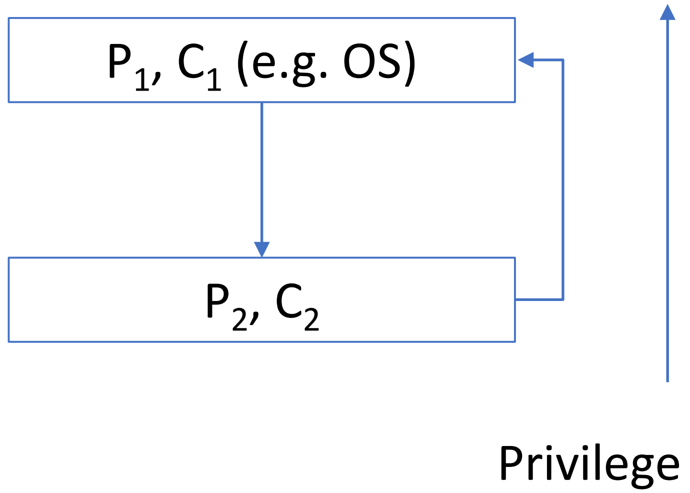
Certificate chains prevent handoff attacks in the trusted boot (not secure boot)

- Trusted boot is necessary since we do not want to halt the system sometimes
- So the question is when the malicious software runs, can it fabricate its certificate?

What Certificate Chains Can Do? (Cont'd)

A certificate chain cannot prevent a privilege escalation attack

- To maintain the certificate chain, privileged code has to keep its private key
- What if P_2 exploits P_1 and takes the privilege of P_1?



Hash Chain Securing Measurement

Hash chains can be more efficient recording software measurements.

- Only a constant amount of secure memory to record an arbitrarily long, append-only list of code identities.

Hash chain is stored in **secure memory** can defeat both privilege escalation and handoff attacks

- What if the certificate chain is stored in secure memory? Too large?

Trusted Platform Module (TPM) uses hash chain to store measurements

How Hash Chain Works?

Hardware-backed hash chain

- Use protected memory register initialized to a known value (e.g., 0) when the computer first boots.
- A hardware API is used to extend identity of a code*, I,* into the log

Calculate new hash: $V \leftarrow \text{Hash}(V \,||\, I)$

- *I*: the identity record of the code and can be saved for later reference
- *V*: the current register value *V*

Given collision-resistant hash, V guarantees integrity of the append-only log

Malicious software cannot erase its identity from the log without rebooting the platform and losing control of the machine.

TPM-Based Measurement Example

BIOS (B), bootloader (L), and operating system (O) should support measurement collection.

- If not, they have to be changed
- #1 When the computer first boots, the TPM's Platform Configuration Registers (PCR) are initialized to a known value (e.g., 0).
- #2 ROM code measures BIOS (B), invokes PCRExtend with a valid PCR index, and then runs BIOS
  - PCRExtend(5, B) means $PCR_5 \leftarrow H(0||B)$
- #3 BIOS measures bootloader, updates PCR, and runs bootlaoder
  - PCRExtend(5, L) : $PCR_5 \leftarrow H(\ PCR5||L) = H(\ H(0||B)||L)$
- #4 Bootloader will extend a measurement of the OS (O) into the TPM before running OS

TPM-Based Measurement Example

- #5 measurement of the application (A) into the TPM and runs the application.
  - As a result, PCR5 <= h = H(H(H(H(0||B)||L)||O)||A)
  - Notice that the entire boot sequence is captured in a single hash value

Recording Dynamic Properties

Is measuring code identity is sufficient to guarantee security?

What if the system accepts user inputs, which exploit system vulnerabilities?

- Buffer overflow?

In addition to code identity, we also want to know dynamic properties of code

- Integrity of the code's control flow (e.g. function call)
- Integrity of its data structures (e.g., the stack), or
- Information-flow control.

Load-time Dynamics Analysis

Transform the program and record identity of transformed program

- Inserting inline reference monitors that enforce a variety of properties, such as stack and control-flow integrity

Recording code identity of the transformed code ensures that a program with the appropriate built-in dynamic property enforcements was loaded and executed.

Load-time dynamics analysis does not protect against attacks that do not tamper with valid control flows.

- For example, a buffer overflow attack might overwrite the Boolean variable *isAdministrator* to give the attacker unexpected privileges.

Run-time Dynamics Analysis

A trust code (e.g. Java Virtual Machine) dynamically enforces a given security property on less-privileged code.

“semantic” attestation - a language runtime (e.g., the Java or .NET virtual machine) monitors and records information about the programs it runs.

- Information includes the class hierarchy or a particular security policy.

Which Property is Necessary?

Many code properties are relevant to security

Which one is fundamentally needed?

- Many dynamic properties can be achieved via code identity
- The identity of the code conveys the expected dynamic properties
- Or dynamic properties enforced on other pieces of software

Other properties may not be that fundamental

Additional hardware support for monitoring (or enforcing) dynamic properties are also necessary

- Code identity should be implemented as a higher priority

Outline

Secure Boot

Measurement of next code to run is compared against a list of measurements for authorized software

- Authorized software - signed software by a CA
- Where to save CA's certificate? Firmware? Can be saved anywhere since it is just a certificate?

Secure boot halts the boot process if unauthorized code is detected

Example system - AEGIS.

- Measure next code to run, check its identity against a certificate from the platform's owner.
- The certificate identifies authorized software.
- No certificate, no running

Storage Access Control Based on Code Identity

How to protect the secrets that applications generate for a long term?

Examples secrets include

- keys used for full disk encryption or email signatures, and
- a list of stored passwords for a web browser.

Solution: access control mechanism to protect cryptographic keys

- where access policies consist of sets of allowed platform configurations
- only if the policy is met, access is granted

IBM 4758 – Secure Storage

IBM 4758 family of cryptographic co-processors

- Tamper-responding storage in battery-backed RAM (BBRAM).
- Additional flash memory *encrypted with keys maintained in BBRAM*.

Physical tampering with the device will result in it actively erasing secrets.

Root cryptographic keys for protected storage can be kept in BBRAM

IBM 4758 Storage Access Restrictions

Storage access restrictions based on the concept of software privilege layers.

- Layer 0: read-only firmware.
- Layer 1: the IBM-provided CP/Q++ OS by default.
- Layers 2 and 3: applications.

Each layer can store secrets either in BBRAM or in flash.

A hardware ratcheting lock prevents a lower privilege layer from accessing the state of a higher-privilege layer.

- Secrets of layer 1 cannot be accessed by applications at layer 2 or 3

TPM-Based Sealed Storage

Software on the platform's main CPU *seals* or *binds* secrets to a set of measurements representing some future platform state

Both operations (seal and bind) essentially encrypt the secret value provided by the software.

TPM will refuse to perform a decryption, unless the current values in its *Platform Configuration Registers* (***PCR***) match those specified during the seal or bind operation.

Full Disk Encryption

The disk encryption keys can be sealed to measurements representing the user's operating system

(*Specific OS, disk encryption keys*)

- e.g. Microsoft BitLocker does it [2]
- The disk can only be decrypted if the intended OS kernel has booted?

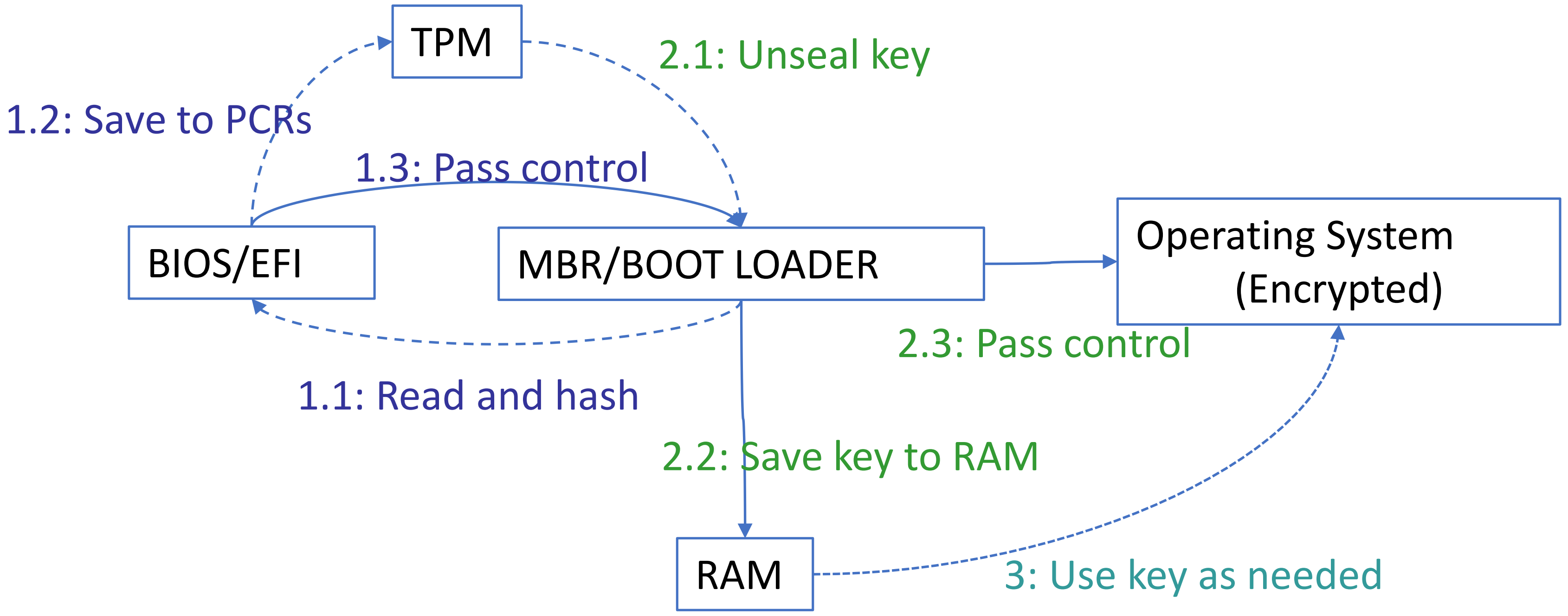
Connecting disk encryption with code identity ensures that

- Boot sequence cannot be changed to load malware or other OS

Seal and bind operations use 2048-bit asymmetric RSA keys

- RSA keys are generated on TPM
- Private keys are never released in the clear.
- Encryption scheme provides both secrecy and integrity

Simplified BitLocker boot process [2]



TPM Key Hierarchy

Because of limited storage space, TPM uses a key hierarchy to protect storage keys.

- Private portion of its *Storage Root Keypair* saved in TPM's protected storage.
- P*ublic portion of the Storage Root Keypair* to *encrypt private portion of other keys*
- Computer software manages encrypted *keys used for seal and bind operations*

Before performing a seal/bind operation,

- The software loads encrypted keys into TPM.
- TPM decrypts the ciphertext to obtain a key, checks its integrity, and then uses it to perform the seal/bind operation.

TPM Key Hierarchy (Cont’d)

Benefits of TPM key hierarchy

- Arbitrary number of storage keys
- Constant amount of protected storage.

Software can generate symmetric keys and use the TPM to protect the symmetric keys.

Sealing Data

Seal operation inside TPM:

(encrypting a secret value, setting access policy)

- *Access policy*: controls the decryption of encrypted secret.
- *Access policy format*: a set of PCR indices and the right values for decryption;
- PCR values in the access policy may be different from the current PCR values.

Unseal - when software wishes to decrypt the sealed secret in the future, it asks the TPM to unseal it.

- TPM decrypts the ciphertext internally;
- If access policy is met, the TPM releases the plaintext

Example of Sealing Data: Secure software updates

Trusted software module P decides to upgrade to a new version P’.

- P has secret data that P’ should inherit
  - #1 P validates the upgrade e.g. through digital signature
  - #2 P passes that data to a seal operation, specifying PCR values corresponding to P’.
  - #3 When P’ runs, P’ asks TPM to unseal the data
- How? Hint: refer to the definition of unsealing

Binding Data

Binding – Encrypt data with the bind key, which is derived from a storage key

- Encryption need not take place on the TPM.
- Flexible when *performing data encryption* because there is too much data?

Decryption on the TPM

- Data is bound to a specific TPM?

TPM-Based Sealed Storage Example

Goal: Only application A can access its secret

BIOS (B), bootloader (L) and the operating system (O)

- Support TPM.

Application (A) is running and PCR_5 is :

- h = H(H(H(H(0||B)||L)||O)||A)

A can generate secret data D_{secret} and seal it under the current value of PCR_5 by invoking:

- Seal((5), D_{secret}) → *C* = Enc_K((5,h)||D_{secret})
- where K is a storage key generated by the TPM.

C is returned to the software that invoked the seal operation.

If the same boot sequence occurs later, A can unseal the secret

- Otherwise, no

Outline

Attestation

Attestation - conveying measurement chains to an external entity in an authentic manner, called *outbound authentication* too

- A remote party (verifier) would like to learn the security status of a local system (attestor)

Verifier needs an authentic measurement chain represents the software state of attestor

- A verifier's trust in an *attestor*'s measurement chain builds from a hardware root of trust.

Prerequisites for attestation: the verifier

- #1 Understands the attestor's hardware configuration and
- #2 Has an authentic public key bound to the hardware root of trust.

General Purpose Coprocessor-Based Attestation

A coprocessor is a computer processor used to supplement the functions of the primary processor (the CPU).

- General-purpose cryptographic co-processors

Coprocessor applications authenticate themselves to remote parties

- Generate and maintain authenticated key pairs
- Communicate securely with remote party.
- Keep a private key in tamper-protected memory
- Have certificates signed by CA

TPM-Based Attestation

Less flexible than general coprocessor-based attestation, without general-purpose computation.

For attestation, attestor’s software relays information between the remote verifier and the TPM

Attestor’s TPM has generated an Attestation Identity Keypair (AIK),

- An **asymmetric** keypair
- **Public key known to the verifier in advance**, and
- Private on the TPM.

TPM-Based Attestation Protocol

#1 The verifier sends a nonce to the attestor

- Prevent replay of old attestations

#2 The attestor asks the TPM to generate a Quote.

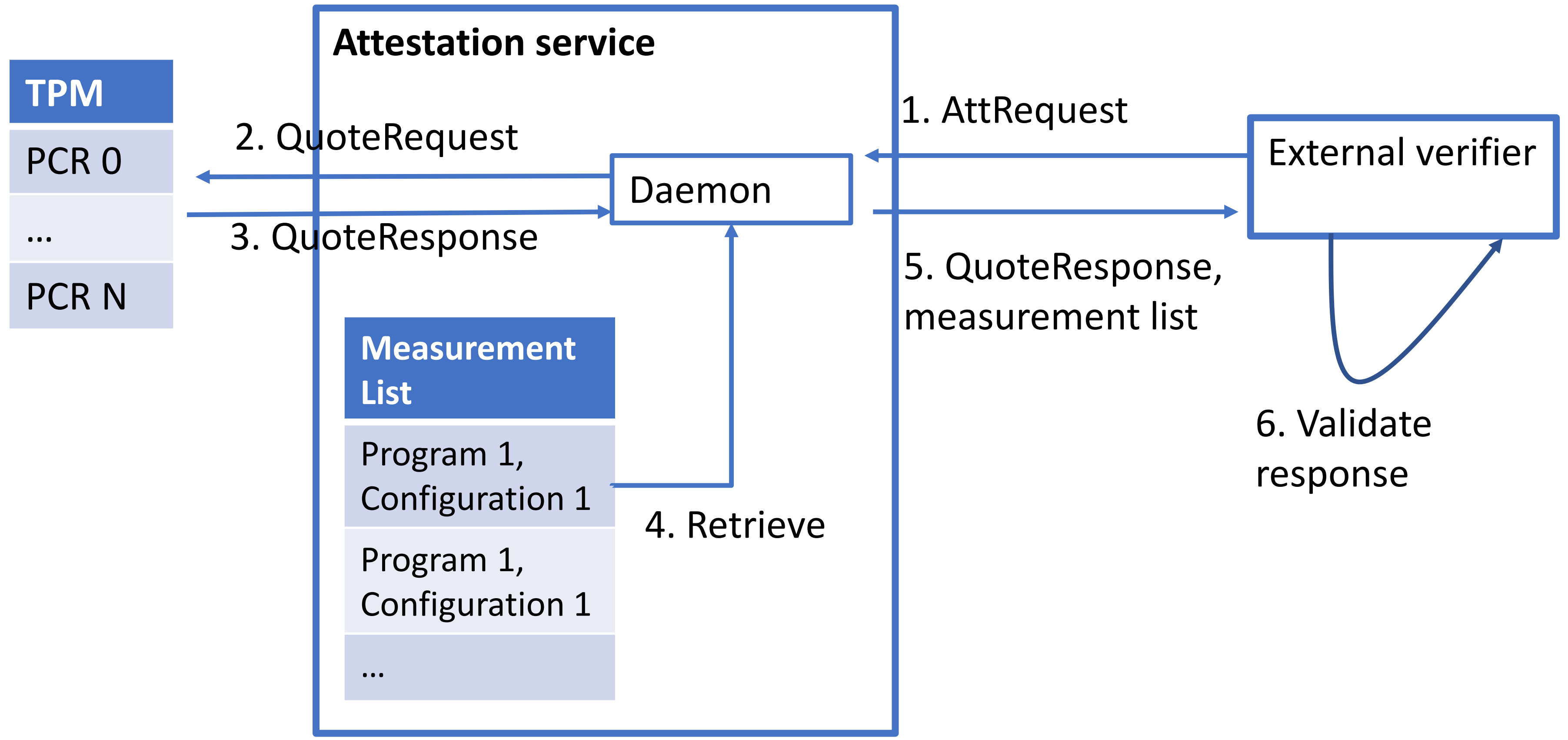
- **Quote**: digital signature(verifier's nonce, the current measurement in the TPM's Platform Configuration Registers (PCRs)).

#3 The attestor sends both the quote and an accumulated measurement list to the verifier. This measurement list

- Contains enough information about measured entities so that the verifier will know what are being measured.
- Implementation-specific, e.g. the hash and full path to a loaded executable

#4 To verify the measurement list, the verifier performs basically signature verification

TPM-Based Attestation Protocol



Outline

Coping With Information Overload

Can we build perfect secure software?

- If yes, a computer running such software can be trusted.
- Unfortunately, no.

Most computers run buggy and unverified code.

State-space explosion

- So many versions, libraries, drivers, dependable applications

How to deal with state-space explosion?

Focusing on Security-Relevant Code

Only record the identity of code that will affect security.

Reduce the amount of security-relevant code

- Simplifies the verifier’s workload for attestation too

To achieve this reduction, the platform must support multiple privilege layers, preventing privilege-escalation attacks

- The more-privileged code enforces isolation between itself and less-privileged code modules.
- Layering simplifies or interprets information given to a verifer

Privilege Layering via the Operating System

Marchesini *et al*. introduce privilege layering OS

A long-term core (an SELinux kernel in their case)

- loads and verifies a **policy file** supplied by an administrator.

An *Enforcer* module in the core ensures that only applications matching the policy can execute.

- Kind of the secure boot model.
- Secrets bound to the long-term core, not applications

Remote attestation by verifier

- The long-term core is trustworthy,
- Enforcer is configured with an appropriate policy (i.e., one that satisfies the external party's requirements).

Privilege Layering via Virtualization

Microsoft's Next-Generation Secure Computing Base (NGSCB)

- Security-sensitive operations confined to one virtual machine (VM)
- General purpose computing in another VM
- Virtual machine monitor (VMM), i.e., hypervisor, provides strong isolation between VMs

Remote attestation to verifier

- The identity of the VMM
- A particular VM, rather than applications in the other VMs.

Challenge: privilege-escalation attacks

The impact of NGSCB architecture

- TPM
- DRTM (dynamic root of trust for measurement)
- How to build secure applications on a bootstrapped foundation of trust

Hardware-Supported Isolation: System-Management Mode

Azab *et al*. uses *System-Management Mode* (*SMM*) to monitor and attest to the integrity of a hypervisor

- No need of software “underneath” the hypervisor.

SMM: a special x86 execution mode invoked via System-Management Interrupts (SMIs) – hardware based

- Often used to control fan speed
- React to special keys on laptop keyboards such as the volume control buttons

Code in an SMI handler operates independently of normal system code, such as the OS or hypervisor

- Can introspect on CPU state and the contents of memory.

Hardware-Supported Isolation: Dynamic Root of Trust for Measurement

VMM-based isolation includes a lot of non-security-relevant code,

- BIOS, the boot loader, and various option ROMs.

To address these shortcomings, AMD and Intel extended the x86 instruction set to support a dynamic root of trust for measurement (DRTM – also known as late launch) operation

- Secure Virtual Machine (SVM) by AMD
- Trusted eXecution Technology (TXT) (LaGrande Technology) by Intel

A DRTM operation

- Resets the CPU and memory controller to a known state
- Atomically measures a piece of code into the TPM
- Begins executing the code in a hardware-protected environment.

Slicing and Privilege Separation

A persistent challenge: most software was written without meticulous attention to the principle of least privilege.

Solution: privilege separation to a legacy codebase, via tools such as PrivTrans

PrivTrans leverages programmer annotations to automatically separate an existing program into privileged and unprivileged components.

The authors compare their results against the manual privilege separation performed for the OpenSSH utility.

Conveying Higher-Level Information

Convert the information into a set of higher-level properties that facilitate trust judgements

Convey information about software

- Static methods such as type checking
- Inline reference monitors
- Dynamic methods such as hypervisors
- Security kernels
- Language runtimes

Attesting to code identity assures verifier the transformed code runs

Outsourcing

Outsource the problem of interpreting code identity to a third party.

One strategy

- Clients obtain certificates from their software providers that map hash values to software names and/or versions.
- A client sends these certificates for their attestation

Benefit

- No need of database for mapping hash values to software packages at verifier

Outline

Different Roots of Trust

Trust in any system needs

- a foundation or
- a root of trust

The root of trust can be built upon

- Secret private key embedded in hardware;
- Corresponding public key certified by the hardware’s manufacturer.
- The first piece of code that make measurements (e.g. code in the early boot process)
- Physical hardware itself.

General-Purpose Tamper-Resistant and Tamper-Responding Devices

Manufacturing tamper-resistant devices is expensive

- No much commercial success

Commercial solutions do use research ideas

- Hardware stores a secret private key,
- Manufacturer digitally signs a certificate of the corresponding public key.
- The verifier uses the certificate to establish trust in the platform.

Commercial Solutions – Cryptographic co-processors

General-purpose cryptographic co-processors such as IBM PCI-based 4758

- Tamper resistant and tamper-responding properties

Packaging for resisting and responding to physical penetration and fluctuations in power and temperature.

Batteries powering active response to detected tampering

- Immediately erasing internal secrets
- Permanently disabling the device

Commercial Solutions – Smart Card

Many higher-end smart cards and SIM cards can fight against physical attack, e.g. Infineon line of SLE88 chips

- Sensors to detect voltage fluctuations, glitches, and light,
- Filters designed to smooth power usage and resist power analysis

How it works?

- A private key for authentication is stored in the microprocessor,
- All private key operations within the microprocessor
- The card can interact with untrusted terminals without leaking the key

Low-end smart cards and SIM cards may not have any active tamper response mechanisms, but hardware obfuscation

General-Purpose Devices Without Dedicated Physical Defenses

Goal: increase the software security of software systems without explicit physical defense measures.

- Achieve different degrees of resilience to physical compromise.

Different physical attacks

- #1 on a daughter card that can be readily unplugged and interposed on,
- #2 soldered to the motherboard,
- #3 integrated with the “super-IO” chip, and
- #4 on the same silicon as the main CPU cores.

Example commodity platforms today

- Trusted Platform Module (TPM), or its mobile counterpart
- Mobile Trusted Module (MTM), or
- a smart card.

TPM-equipped Platforms

The TPM chip is a hardware device

- No specific tamper resistance.

Trust in the TPM stems from three roots of trust, for Storage, Reporting, and Measurement.

- *Trusted storage* by an encryption key in TPM's *nonvolatile RAM*.
- *The root for reporting* (remote attestation) by the TPM's storage facilities.
- *TPM measurement* by firmware called the Core Root of Trust for Measurement, which initializes the TPM when the system boots.

MTM-equipped Platforms

The TCG Mobile Phone Working Group (MPWG) has specified a Mobile Trusted Module (MTM)

The MTM specification interleaves two different profiles, depending on the device's owner: a Mobile Local Owner Trusted Module (MLTM) and a Mobile Remote Owner Trusted Module (MRTM).

- The local owner has physical control over the device, i.e., its user.
- The remote owner is a stakeholder without physical access to the deployed device, e.g., a device manufacturer or a network service provider.

The root of trust for execution typically makes use of the isolated execution features of the platform's main CPU, e.g., ARM TrustZone or TI M-Shield

Boot integrity is provided using a secure boot model.

Special-Purpose Minimal Devices

Specifically designed for some applications,

What functionality is needed in secure hardware for various classes of applications?

- An open area of research.

Research Solutions Without Hardware Support

That is software-based attestation

- #1 Code computes a checksum over itself
- #2 A verifier checks the checksum for integrity and also measures the computation time.
- #3 Adversarial interference with the checksum computation will either slow the computation (detectable by the verifier) or will result in an incorrect checksum.

Many other schemes …

Outline

What we have learned so far?

How to use secure hardware mechanisms for secure bootstrap

- Secure hardware to monitor and report on the software state

We can then decide whether to trust the platform

- And what to do else

Secure hardware such as cryptographic co-processors with tamper resistant/proof/response properties is expensive

- Cheaper TPM is sued and against software attacks
- TPM can be defeated by an adversary who possesses it
- TPM can be used for secure boot, measurements and remote attestation
- Recent CPU advances such SGX make measurements simpler

Remaining Question

How do we bootstrap trust in the TPM itself?

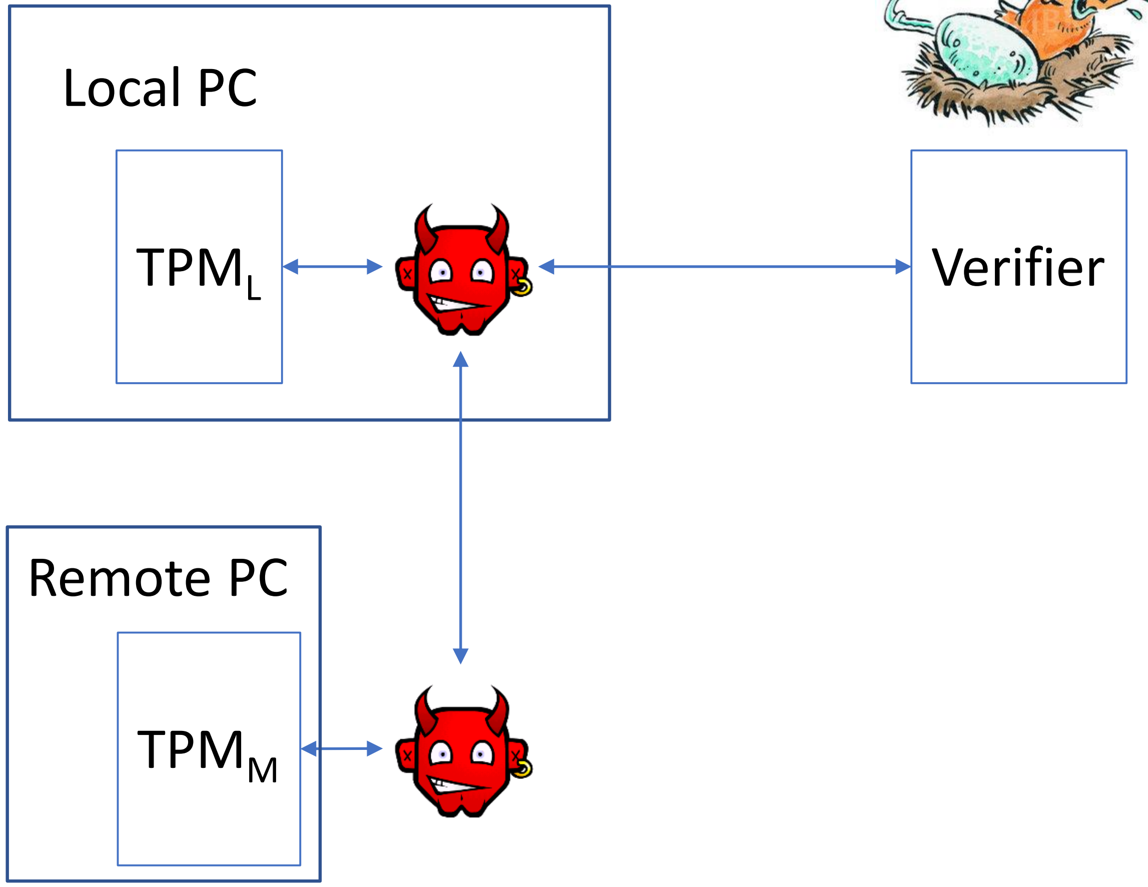
How can a user get the TPM’s public key?

- An authentic public key is needed to verify if we are communicating with the right TPM

Otherwise, cuckoo attack

- Malware on the local machine forwards the user’s messages to a remote adversarial TPM

Cuckoo Attack



Potential Solutions

How to securely deliver the TPM’s public key in case of malware attacks?

Establishing a Secure Channel

Hardware-Based Secure Channels

- **Special-Purpose Interface**. Add a new hardware interface on a computer for an external device to talk directly to the TPM of the computer.
- **Existing Interface**. Use an existing external interface (such as Firewire or USB) to talk directly to the TPM.
- **External Late Launch Data**. New CPU features from AMD and Intel.
- **Special-Purpose Button** that can execute an authenticated code module establishing a secure channel between the verifier (connected via USB, for example) and the TPM

Cryptographic Secure Channels

Camera-based Channel. *Seeing-is-Believing (SiB)*, an approach - a hash of the platform's identity can be put in a 2-D barcode on the platform's case.

Human-based Channel. The hash as an alpha-numeric string that can be entered into a smartphone, or into a dedicated trusted device.

Trusted BIOS outputs the platform's identity (either visually or via an external interface, such as USB).

Trusted Third Party. A certificate the TPM with a particular machine.

Outline

Validating

How to validate involved hardware, software, and protocols?

From a hardware perspective, Smith and Austel

- discuss efforts to apply formal methods to the design of secure coprocessors
- state formal security goals for such processors.

Bruschi et al. use a model checker to

- find a replay attack in the TPM's Object Independent Authorization Protocol (OIAP)
- propose a countermeasure to address their attack, though it requires a TPM design change.

Etc.

References

[1] Bryan Parno, Jonathan M. McCune, Adrian Perrig, Bootstrapping Trust in Modern Computers, Springer Publishing Company, Incorporated ©2011

[2] Bryan Parno, Bypassing Local Windows Authentication to Defeat Full Disk Encryption, Black Hat Europe 2015

[3] Secure the Windows 10 boot process, 06/23/2017