Problem

• My computer is running a process
• It makes a request to your computer
  ‣ Asks for some secret data to process
  ‣ Provides an input you depend on
• How do you know it is executing correctly?
• **Example**
  ‣ ATM machine is uploading a transaction to the bank
  ‣ How does the bank know that this ATM is running correctly, so the transaction can be considered legal?
What would you do?

- Nothing
- Proof by authority (**Certificates**)
  - Tells you who, but not what
- Constrain the system (**Secure Boot**)
  - Effective, limiting, but proof is implied
- Inspect the runtime state (**Authenticated Boot**)
  - Flexible, attestable, but difficult to prove semantics
Secure Boot

• Check each stage in the boot process
  ‣ Is code that you are going to load acceptable?
  ‣ If not, terminate the boot process

• Must establish a **Root-of-Trust**
  ‣ A component trusted to speak for the correctness of others
  ‣ Assumed to be correct because errors are **undetectable**
AEGIS

• AEGIS architecture (1997)
  ‣ ROM checks the BIOS
  ‣ BIOS checks expansion ROMs and boot block
  ‣ Boot Loader checks the OS

• Why not just boot from a floppy (DVD now)?
• Is this a root of trust?
• What can it verify?
• How do we know it booted securely?
Authenticated Boot

- Secure boot enforces requirements and uses special hardware to ensure a specific system is booted
  - Implied verification (Good because it is)

- By contrast, we can measure each stage and have a verifier authenticate the correctness of the stage
  - Verifier must know how to verify correctness
  - Behavior is uncertain until verification
  - Can you verify yourself?

- What is our root of trust?
Remote Attestation

- Attestation enables verifier to establish trust in untrusted device
  - Attestation tells verifier what code is executing on device
  - If intended code is executing on untrusted device, verifier can trust its operation
Outbound Authentication


- **Goals**: Securely boot the 4758 and prove to remote parties (combines secure boot and attestation)

- More specifically, relying party $P$ wants to prove that only entity $E$ holds key $K$
  - $E$ is high integrity despite depending on several integrity-relevant events (e.g., boot and upgrade)

- **Defines a precise logic for reasoning about such properties**
  - But, the 4758 is a very limited system (one application)
• “A relying party needs to conclude that a particular key pair really belongs to a particular software entity within a particular untampered coprocessor.”

• Why does this prove the integrity of the processing environment?

• What else is needed to make this connection between the key and entity’s correctness?

• IBM 4758 Secure coprocessor contains various hardware protections to isolated memory, manage keys, and perform updates.
Configurations and Epochs

- A *Layer N configuration* is the maximal period in which that layer is run-able, with an unchanging software environment in Layers 1, ..., N.

- A *Layer N epoch* is the maximal period in which the layer can run and accumulate state.

- Software runs for an epoch, but any change to the software (integrity-relevant event) results in a new configuration.
  - Hardware constrains these events.

- What are other integrity-relevant events in conventional systems?
Execution History

- $E$ wishes to prove it “owns” $K$ by presenting a $\text{Chain}(E, K, H)$ of certificates
  - $H$ shows the chain of entities that certify $E$’s $K$ before the current run $R$.
  - The chain speaks for the correctness of $K$, which the relying party $P$ should trust.

- Implications: Only these entities should have access to the secrets and configuration of $E$
  - Hardware limits the set of integrity-relevant operations that can affect $E$
  - General purpose systems allow more operations
Dependency (Integrity)

• Each entity $E$ has a dependency set $D(E)$
  ‣ An entity $E$ depends on entities that have read/write access to its secrets and write access to its code
    • In general purpose systems, it is primarily dependence on untrusted data that leads to integrity problems

• $\text{TrustSet}(P)$ – set of entities that $P$ trusted
  ‣ $\text{TrustSet}(P)$ should be a superset of the measured dependencies

• Implications
  ‣ Dependency must be comprehensively defined
    • Initialization, Code Load, Subsequent Reads
Validation

• P wants to verify E depends only on its TrustSet(P)
  ‣ A run R, prefixed by H, defines an entity’s $D_R(E)$

$\text{Validate}(P, \text{Chain}(E, K, H)) \Rightarrow D_R(E) \subseteq \text{TrustSet}(P)$

• Hardware protections imply $D_R(E)$

• If $D_R(E)$ is in P’s trust set, then the chain is valid

$D_R(E) \subseteq \text{TrustSet}(P) \Rightarrow \text{Validate}(P, \text{Chain}(E, K, H))$

• OA requires an entity E’s dependencies satisfy the trust set of P to validate that E owns K
Validation Implications

• Difficult for P to verify all entities, integrity relevant events, and dependencies
  ‣ We just want a green light (iTurtle)

• Enforcement simplifies the protocol
  ‣ OA makes this seem easy, but has a lot of constraints to simplify the problem
Trusted(?) Computing

- The Trusted Platform Module (TPM) brought authenticated boot into the main stream
- Essentially, the TPM offers few primitives
  - Measurement, cryptography, key generation, PRNG
  - Controlled by physical presence of the machine
  - BIOS is Core Root of Trust for Measurement (CRTM)
- Spec only discussed how to measure early boot phases and general userspace measurements
• The **Trusted Platform Module** is a tamper-resistant secure microcontroller.

  ‣ Manages cryptographic keys and functionality it uses to support security relevant operations.

  ‣ Measures the code loaded by the system (firmware, BIOS, OS kernel, device drives, application processes, ...)

• Measurements are hashes of loaded code (PCRs)
Authenticated Boot

- A lot of FUD was generated around what it does & doesn’t do
- Paladium/NGSCB architecture (Microsoft, 2002)
  - Use virtualization to split system
  - Measure the “trusted” part to prove its integrity before responding
- “Meet the emerging requirements of an interconnected world” – Microsoft
- Take over the world – Ross Anderson and others
Linux Integrity Measurement

- **Problem**: How can we verify the software environment of networked systems?
- **Solution**: Extend TPM measurement architecture to measure system’s runtime (Software Stack)

- **Additional Goals**
  - Load-time integrity
  - Unobtrusive
  - Tamper-evident
  - Usability
Integrity Measurement

• Means used to determine the state of the host
• Relies on *measurement* (i.e., hash fingerprinting of the code)
  ‣ Hardware support emanates from the core root of trust for measurement (CRTM), secured on the host
  ‣ subsequent measured steps: BIOS, bootloader (stage 1 & 2), OS

• Attestation of the code is performed with TPM *Quote* operation
• System integrity depends on several components
  ‣ Executables
    • Programs, libraries, kernel modules
  ‣ Configuration Files
    • httpd.conf, /etc/shadow
  ‣ Unstructured Input
    • Network data, keystrokes, basically everything else
IMA Implementation

- Place hooks throughout Linux kernel
  - Later added as an LSM and then special LIM hooks
- Extend TPM PCR at file load-time
  - PCR = SHA1(File || PCR)
- Applications instrumented to measure inputs
  - Bash scripts, interpreters…
- Verifying all events is difficult
  - Need known “good” values to validate measurements
  - Leverage OS distribution definitions
Limitations

- What does IMA prove?
- Can a system with a valid IMA attestation be malicious?
- What else can be done to improve attestations?
Limitations

- **Static** root of trust for measurement *(reboot)*
  - Coarse-grained, measures entire system
    - Requires hundreds of integrity measurements just to boot
    - Every host is different
      - firmware versions, drivers, patches, apps, spyware, …
    - What does a PCR mean in this context?
    - TCB includes entire system!
  - Integrity measurements are done at **load-time**, not at run-time
    - Time-of-check-time-of-use (TOCTOU) problem
    - Cannot detect any dynamic attacks
    - No guarantee of execution
DRTM (Late Launch)

• *Dynamic* Root of Trust for Measurement
• Involves both CPU and TPM v1.2
• Security properties similar to reboot... without a reboot
• Removes many things from TCB
  • BIOS, boot loader, DMA-enabled devices, …
  • Long-running OS and Apps if done right
• When combined with virtualization
  ‣ VMM can be measured (MVMM), potentially lengthy uptimes
  ‣ Integrity of loaded code can be attested
  ‣ Untrusted legacy OS can coexist with trusted software
• Allows introduction of new, higher-assurance software without breaking existing systems
AMD/Intel Late Launch Extensions

• AMD: Secure Virtual Machine (SVM)
• Intel: Trusted eXecution Technology (TXT)
  ‣ Formerly LaGrande Technology (LT)

• Similarities:
  – Late launch of a measured block of code
    ‣ Hardware support for virtualization

• Differences:
  ‣ AMD provides measured environment only
  ‣ Intel adds authenticated code capabilities
    • The system’s chipset contains a public key to verify signed code
Secure Kernel Init

• AMD SKINIT instruction - allows for DRTM

• Atomic execution:
  ‣ sets CPU to INIT, disables interrupts, enables DMA over 64 KB Secure Loader Block
  ‣ resets dynamic PCRs in TPM to zero (new in TPM v1.2)

• Verifier receives attestation after SKINIT is run
  ‣ knows SKINIT was used and that software TCB includes only the SLB, as well as what was executed
Take Away

- Programs on systems may be security-critical
  - How do we determine if they are up to the task?

- Secure and authenticated boot processes enable a party to prove a system’s integrity satisfies some requirements
  - Secure boot proves to local parties
  - Authenticated boot for remote parties

- OA provides secure boot and authenticate boot for comprehensive control – of a simple device

- IMA provides authenticated boot for Linux