NIST Special Publication 800-193

Platform Firmware Resiliency Guidelines

Andrew Regenscheid

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-193

C O M P U T E R   S E C U R I T Y

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Special Publication 800-193

Platform Firmware Resiliency Guidelines

Andrew Regenscheid
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-193

May 2018

Authority

National Institute of Standards and Technology Special Publication 800-193
Natl. Inst. Stand. Technol. Spec. Publ. 800-193, 45 pages (May 2018)
CODEN: NSPUE2

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-193

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: sp800-193comments@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Abstract

This document provides technical guidelines and recommendations supporting resiliency of platform firmware and data against potentially destructive attacks. The platform is a collection of fundamental hardware and firmware components needed to boot and operate a system. A successful attack on platform firmware could render a system inoperable, perhaps permanently, or requiring reprogramming by the original manufacturer, resulting in significant disruptions to users. The technical guidelines in this document promote resiliency in the platform by describing security mechanisms for protecting the platform against unauthorized changes, detecting unauthorized changes that occur, and recovering from attacks rapidly and securely. Implementers, including Original Equipment Manufacturers (OEMs) and component/device suppliers, can use these guidelines to build stronger security mechanisms into platforms. System administrators, security professionals, and users can use this document to guide procurement strategies and priorities for future systems.

Keywords

BIOS; Code signing; Firmware; Option ROM; Platform Firmware

Acknowledgements

The author, Andrew Regenscheid of the National Institute of Standards and Technology (NIST), wishes to thank his colleagues who reviewed drafts of this document and contributed to its technical content. In particular, NIST appreciates the contributions from experts from industry and government who helped guide this work. These experts included Chirag Schroff from Cisco; Mukund Khatri from Dell; CJ Coppersmith, Gary Campbell, Shiva Dasari, and Tom Laffey from Hewlett Packard Enterprise; Jim Mann from HP Inc.; Charles Palmer from IBM; Bob Hale, David Riss, and Vincent Zimmer from Intel Corporation; Paul England and Rob Spiger from Microsoft, and Shane Steiger.

NIST would also like to acknowledge and thank Kevin Bingham, Cara Steib, and Mike Boyle, from the National Security Agency, who provided substantial contributions to this document, as well as Jeffrey Burke from Noblis NSP.

Audience

The intended audience for this document includes system and platform device vendors of computer systems, including manufacturers of client, servers, and networking devices. The security principles and recommendations contained in this document should be broadly applicable to other classes of systems with updatable firmware, including Internet of Things devices, embedded systems, and mobile devices. The technical guidelines assume readers have expertise in the platform architectures and are targeted primarily at developers and engineers responsible for implementing firmware-level security technologies in systems and devices.

Executive Summary

Modern computing system architectures can be thought of in layers. The top layers are *software*, composed of the operating system and applications. While these provide most of the functional capabilities employed by users, they rely on functions and services provided by the underlying layers, which this document collectively refers to as the *platform*. The platform includes the hardware and firmware components necessary to initialize components, boot the system, and provide runtime services implemented by hardware components.

Platform firmware, and its associated configuration data, is critical to the trustworthiness of a computing system. Much of this firmware is highly privileged in the system architectures, and because this firmware is necessary for the system to operate, repairing this firmware can be challenging. A successful attack on platform firmware could render a system inoperable, perhaps permanently or requiring reprogramming by the original manufacturer, resulting in significant disruptions to users. Other sophisticated malicious attacks could attempt to inject persistent malware in this firmware, modifying critical low-level services to disrupt operations, exfiltrate data, or otherwise impact the security posture of a computer system.

Earlier NIST publications have addressed the threat of attacks on one particular type of platform firmware: boot firmware, commonly known as the Basic Input/Output System (BIOS). However, the platform consists of many other devices with firmware and configuration data. These devices, including storage and network controllers, graphics processing units, and service processors, are also highly-privileged and needed for systems to behave securely and reliably.

This document provides technical guidelines intended to support resiliency of platforms against potentially destructive attacks. These guidelines are based on the following three principles:

- **Protection:** Mechanisms for ensuring that Platform Firmware code and critical data remain in a state of integrity and are protected from corruption, such as the process for ensuring the authenticity and integrity of firmware updates.
- **Detection:** Mechanisms for detecting when Platform Firmware code and critical data have been corrupted.
- **Recovery:** Mechanisms for restoring Platform Firmware code and critical data to a state of integrity in the event that any such firmware code or critical data are detected to have been corrupted, or when forced to recover through an authorized mechanism. Recovery is limited to the ability to recover firmware code and critical data.

These guidelines are intended to address platforms in personal computer (PC) clients, servers, and network devices, but should be broadly applicable to other classes of systems. Implementers, including Original Equipment Manufacturers (OEMs) and component/device suppliers, can use these guidelines to build stronger security mechanisms into platforms. System administrators, security professionals, and users can use this document to guide procurement strategies and priorities for future systems.

Table of Contents

List of Appendices

List of Figures

1 Introduction

1.1 Purpose

Modern computing and information technology systems are built upon a variety of hardware components that provide the fundamental capabilities required by the system to operate. Many of these hardware components have firmware and configuration data that drive their behavior, and which must remain in a state with integrity in order for the system to function properly. One example of such firmware is commonly referred to as the Basic Input/Output System (BIOS), which is used to facilitate the hardware initialization process and transition control to the operating system. Depending on the system, there may be tens or hundreds of microcontrollers with other kinds of programmable firmware which support the overall system architecture. That collection of hardware and firmware components is typically called the *platform*.

The devices which make up the platform are crucial to integrity and availability of the systems built upon the platform. Without these devices, systems may fail to operate correctly, or may not operate at all. Targeted attacks at certain devices within the platform could significantly impact the security posture of these systems, possibly allowing a low-level, persistent malware presence. Attacks which aim to damage or remove platform firmware have the potential to render systems permanently damaged, incurring substantial costs to the affected parties.

The purpose of this document is to provide security guidelines to support system resiliency at a platform level. As defined by the International Council of Systems Engineering (INCOSE), system resilience is “the capability of a system with specific characteristics before, during and after a disruption to absorb the disruption, recover to an acceptable level of performance, and sustain that level for an acceptable period of time.” [10] Applied to information systems, cyber resiliency is “ability to anticipate, withstand, recover from, and adapt to adverse conditions, stresses, attacks, or compromises on systems that include cyber resources.” While guidelines on cyber resiliency at a system level are described in draft NIST Special Publication 800-160, Volume 2, this publication notes that system-level resiliency should be supported by foundational security capabilities in computer platforms. The guidelines in this document support cyber resiliency by specifying mechanisms that protect firmware and configuration data from attacks, and that can detect and recover from successful attacks.

1.2 Audience

The intended audience for this document includes system and platform device vendors of computer systems, including manufacturers of clients, servers and networking devices. The technical guidelines assume readers have expertise in the platform architectures and are targeted primarily at developers and engineers responsible for implementing firmware-level security technologies in systems and devices.

The material may also be of use when developing enterprise-wide procurement strategies and deployment. The material in this document is technically oriented, and it is assumed that readers have at least a basic understanding of computer security principles and computer architectures. The document provides background information to help such readers understand the topics that are discussed.

1.3 Applicability and Scope

The goal of this document is to provide principles and guidelines that can support platform resiliency primarily against remote attacks. These principles and guidelines directly apply to the individual devices that make up a platform (see Section 2.1 for a list of examples). Specifically, they describe security mechanisms aimed at protecting each device from unauthorized changes to its firmware or critical data and restoring the platform to a state of integrity.

1.4 Document Structure

The remainder of this document is organized into the following major sections:

- Section 2 provides informative material describing platform components and architectures.
- Section 3 describes the security principles that form the basis for the guidelines in this document, and describes key concepts for applying these principles to platform resiliency.
- Section 4 contains technical security guidelines for protection of firmware code and critical data, detection of authorized changes, and recovery to a state of integrity.
- Appendix A provides an acronym and abbreviation list for the document.
- Appendix B presents a glossary of selected terms from the document.
- Appendix C contains a list of references for the document.

2 Platform Architecture

Ensuring a platform's firmware code and critical data are always in a state of integrity is critical to ensure that a computing system can be operated free from malware. Modern client and server computing systems can be considered to be separated into two high-level logical constructs, *platform* and *software*. For the purposes of this document, we will describe the combination of these two logical constructs as a system. Note that Figure 1 is merely illustrative and is not intended to represent all possible devices in a platform, nor is it intended to represent an exemplary architecture for any particular device. At a high-level, items in blue-shaded boxes are devices to be considered part of a platform.

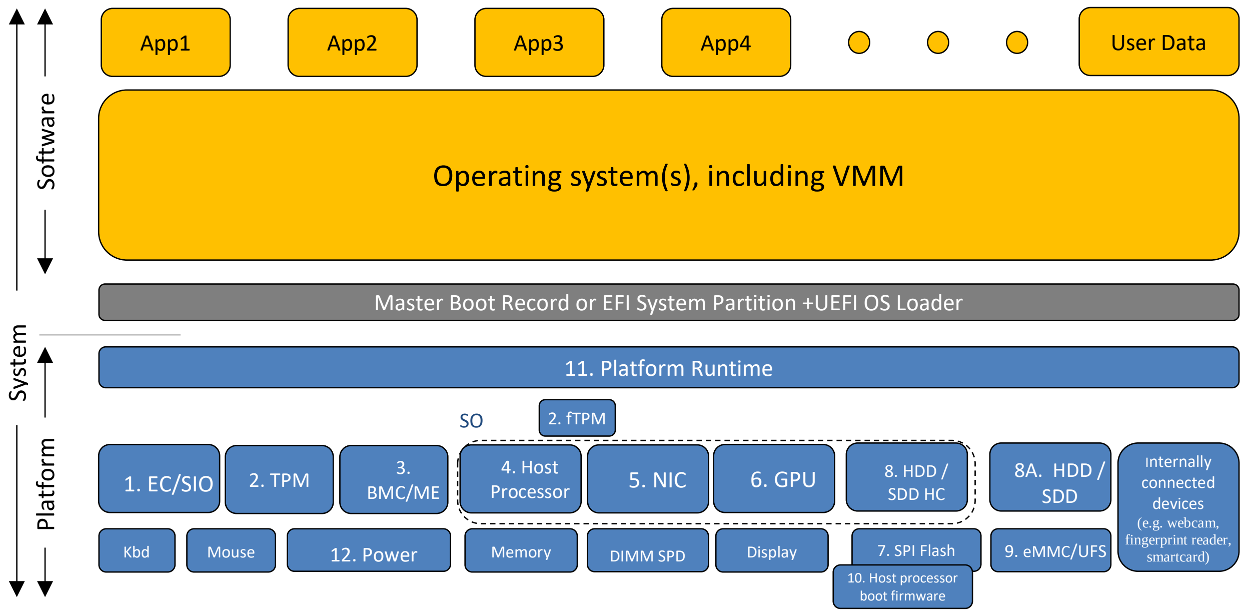

Figure 1: High-Level System Architecture

Broadly speaking, the *platform* is comprised of hardware and firmware necessary to boot the system to a point at which software, or an operating system, can be loaded; *software* is comprised of elements required to load the operating system and all applications and data subsequently handled by the operating system. Note that some firmware continues to execute once software has started. Existing industry best practice, as well as NIST publications such as NIST SP 800-147, *BIOS Protection Guidelines* [1], and NIST SP 800-147B, *BIOS Protection Guidelines for Servers* [2], already address the issue of protecting the integrity of a platform's host processor boot firmware (traditionally called BIOS, and more recently UEFI [3])[1] and its update mechanisms, but protection is only one of three key elements of cyber resiliency (the other two being detection and recovery). Additionally, the resilience of other critical firmware on the platform has not yet been addressed to the level the host processor boot firmware has been. While it is beyond the scope of this document to identify and define every category and

[1] For purposes of this document, host processor boot firmware will be used generically to refer to either legacy Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI).

architecture of device which contains firmware, this document is applicable to any device in the platform which contains firmware, inclusive of PCs, servers, networking devices, smartphones, tablets, etc.

2.1 Platform Devices

As noted above, the platform is a collection of devices that provide the functional capabilities and services needed by the operating system and applications. While resiliency of the platform as a whole is the ultimate objective, it is important to recognize that the platform is composed of many different devices, often developed and manufactured by different vendors. For that reason, the technical guidelines in this document are described in terms of guidelines for individual platform devices.

For the purposes of describing a resilient platform, this section provides a list of devices which are often critical to the normal and secure operation of a platform. These devices typically contain mutable firmware, and are covered by the intended scope of the security guidelines in this document.

However, this should not be considered an exhaustive list of all devices in every platform of interest. Platform vendors will need to carefully consider other devices which should be regarded as in scope for their particular platform.

In the case of a traditional x86-based platform (desktop, notebook, server, network switch), these devices are identified in Figure 1, and are defined below. Note that the numbers here reference those used to label the devices in Figure 1. The ordering is not meant to imply any priority or sequencing.

1. **Embedded Controller (EC) / Super I/O (SIO)**
   An EC is typically associated with mobile platforms (notebooks, convertibles, tablets), while a SIO is typically associated with desk-based platforms (desktops, desk-based workstations, All-in-Ones, Thin Clients). This is not universally true, but is generally true enough to establish the type of client system in which one might find an EC or SIO. An EC or SIO typically controls functions in the platform such as the keyboard, LEDs, fans, battery monitoring/charging, thermal monitoring, etc. Additionally, it is typically the first system board device in the platform to execute code, even holding the host processor in reset until the EC/SIO is ready for the host processor to fetch its first line of host processor firmware code.

2. **Trusted Platform Module (TPM)**
   A TPM [4] is a security coprocessor capable of securely storing and using cryptographic keys and measurements of the state of the platform. These capabilities can be used, among other things, to secure data stored on the system, provide a strong device identity, and to attest the state of the system. While not all platforms include or make use of a TPM, on any system in which a TPM is included and used, its firmware must be protected given its criticality in helping ensure the trustworthiness of the platform. TPMs also contain non-volatile memory storage which may contain critical data and, if so, must be protected. TPM's can be either discrete hardware devices, or may be realized in

firmware executed on a platform host controller or other microcontroller (the latter are sometimes referred to as firmware TPM's, or fTPM).

3. **Baseboard Management Controller (BMC) / Management Engine (ME)**
   A BMC is associated with server platforms while an ME is typically associated with client platforms. In both cases, a core aspect of their functionality is to serve as an out-of-band management device enabling platform administrators to manage a platform without requiring the host operating system to be running. While not always strictly necessary to the basic computing function of a server or client platform, most modern server and client platforms include a BMC/ME, making it critical that their firmware does not negatively affect the state of integrity of the host processor's security domain.

4. **Host Processor [aka Central Processing Unit (CPU), aka Application Processing Unit (APU)]**
   The host processor is the primary processing unit in a typical platform, traditionally called a CPU, and now also sometimes referred to as an APU or a System on a Chip (SoC). This is the processing unit on which the primary operating system (and/or hypervisor), as well as user applications run. This is the processor that is responsible for loading and executing the host processor firmware.

5. **Network Interface Controller (NIC)**
   Whether discrete or integrated as part of an SoC, most modern client and server platforms have at least one NIC (wired or wireless), and could have multiple, including multiple types (wired, Wi-Fi, cellular). While having a NIC is not strictly required to boot a platform, in today's connected world it's important to have some form of connectivity at some point after a system has booted. More importantly, a compromised NIC firmware image could serve as a launch pad for other exploits in the system, be used to exfiltrate data, serve as a man-in-the-middle, etc. In addition to firmware run by a microcontroller, a NIC may include expansion Read-Only Memory (ROM) firmware which is loaded during boot and executed by the host processor. It is critical that a NIC's expansion ROM firmware is also protected. The expansion ROM firmware may be stored with the host processor boot firmware (in the case of an integrated NIC), or may be stored separately with the NIC itself as in the case of an add-in card. A NIC often also contains critical data, for example a Media Access Control (MAC) address may be stored in mutable memory. An attack on this critical data could result in a denial of service (DoS) both of this platform as well as of another system with a matching MAC address.

6. **Graphics Processing Unit (GPU)**
   A GPU is a device that serves as the primary 'output' human interface device (HID) in client platforms. In some cases, GPUs may also be used as coprocessors to support high-performance computing. GPUs could serve as a launch pad for other exploits in the system. In addition to firmware run by a microcontroller, a GPU may include an expansion ROM firmware which is loaded during boot and executed by the host processor. The expansion ROM firmware may be stored with the host processor boot firmware (in the case of an integrated GPU), or may be stored separately with the GPU itself as in the case of an add-in card.

7. **Serial Peripheral Interface (SPI) Flash**
   Most modern platforms include some amount of SPI flash to store firmware, typically for host processor boot firmware, though it could be used for other purposes.

8. **A) Host Controller (HC) for mass storage devices**
   For most modern platforms, some form of local mass storage in the form of either a HDD or SSD is required in order to boot an operating system and hold a user's applications and data. In order for the data to get stored on the mass storage device, a Host Controller (HC) is used to move the data from the platform's main memory to the physical storage medium over some storage bus (e.g. SATA, SCSI, PCIe). This HC has its own microcontroller and associated firmware. The HC could either be integrated into an SoC, or could be a separate device or on an add-in card.

   B) Hard Disk Drive (HDD) / Solid State Drive (SSD)
   An HDD or SSD represents current state of the art in a traditional platform for storage of large quantities of data. These devices are coupled with the Host Controller. Within the HDD or SSD, a microcontroller and associated firmware are used to perform the actual storage operation of data sent from the platform's main memory to the mass storage device. Compromising a HDD's or SSD's firmware can also be used as a launch pad for other exploits in the system, or could be used to compromise a user and/or platform data.

9. **embedded MultiMedia Card (eMMC) / Universal Flash Storage (UFS)**
   eMMC and UFS are emerging as the standard mass storage devices for mobile systems. Each of them may include their own expansion ROM firmware and/or microcontroller with associated firmware.

10. **Host Processor Boot Firmware**
    In most modern platforms, host processor boot firmware is contained in a SPI flash device. BIOS and Unified Extensible Firmware Interface (UEFI) are examples of this type of firmware.

11. **Platform Runtime Firmware**
    In addition to boot firmware, there is platform runtime code. This is code which remains resident in memory and executable after the platform has booted. This is most typical for microcontrollers where firmware is required to execute to perform some function while the system is fully operational. An example of host processor firmware which is considered as runtime code would be System Management Mode (SMM) code.

12. **Power Supply**
    Some power supplies have their own microcontroller and associated firmware. Common battery architectures also include internal logic and firmware governing the charge and discharge behavior of the battery.

13. **Glue Logic (CPLD's, FPGA's) – *not pictured***
    Modern embedded systems use programmable logic components to provide glue logic functionality. There are two types of programmable logic components, Field

Programmable Gate Arrays known as FPGAs and Complex Programmable Logic Devices known as CPLDs. FPGAs are typically loaded with bitstream programs from attached flash devices on power up. CPLDs on the other hand are programmed with a bitstream once and then they retain the function until programmed again in the field. Typically this functionality is needed for basic operations of the system and if corrupted could result in permanent denial of service of a platform.

14. **Fans – *not pictured***
    Some fans have their own microcontroller and associated firmware.

2.2 Code and Data in Platform Devices

The devices described above will typically contain some set of firmware and data on nonvolatile storage, either resident on the device itself or on a shared storage device (e.g., the SPI flash). This section describes firmware code and data, and briefly discusses the scope of the document related to these components.

2.2.1 Code

Firmware code is the set of instructions used by any device's processing unit to perform the operations required by the device. Historically, firmware in platform devices has rarely been modified in the field, although system or component vendors may develop firmware updates which patch vulnerabilities, fix bugs, or add new functionality. As the complexity of this firmware increases, firmware updates have become more common, with hardware and operating system vendors providing tools to help administrators update their firmware.

Because firmware in large part drives the behavior of a device, it is important that it remain in a trustworthy state on the platform. Attacks on the firmware code could render a device inoperable or inject malicious functionality into a device. Firmware should only be loaded from an authorized source, typically either the manufacturer of the system or of the platform device.

The guidelines in this document describe mechanisms to protect firmware code by verifying updates using digital signatures. They also describe mechanisms to detect unauthorized changes to firmware, and secure methods of recovery.

2.2.2 Data

Data are pieces of information that Platform Firmware code uses to carry out its operation, as instructed by the code. Data can be further categorized as critical and non-critical. Critical data includes configuration settings and policies that are needed to be in a valid state for the device to maintain its security posture. Non-critical data includes all other data.

2.2.2.1 Critical Data

Critical data may be used for various purposes, including:

- **Configuration settings:** Data which tells the code how to configure operational aspects of the device
  Example: Enabling a peripheral that is disallowed by enterprise security policies.
  Example: The table of non-functional sectors in a hard drive.
- **Policies:** Data which tells the code what path to take or how to respond
  Example: The system's boot order describes the valid devices to attempt to boot from as well as the order.
  Example: UEFI Secure Boot, a set of security configurations controlling which third party code the BIOS will hand control to.

Critical data is difficult to precisely define because data that may be critical for one device may not be critical for another. However, common characteristics of critical data include:

- It must be in a valid state for the proper booting and run-time operation of the device;
- It persists across power cycles (e.g. stored in non-volatile memory)
- It modifies the behavior or function of the device
- It must be in a valid state to support protection, detection and/or recovery of platform firmware and associated data.

Some critical data is hard-coded in code, and updated only by means of a firmware image update. For the purposes of this document, hard-coded data is considered part of the code, and protected according to the firmware code protection, detection and recovery guidelines.

Platform devices often have other data that is configurable during normal operation by Platform Administrators, hardware, firmware or software. Because corruption of critical data can interfere with the normal or secure operation of a system, it is important to protect critical data from corruption, and to be able to recover when problems are detected. However, strong protection of some forms of critical data can be architecturally difficult, due to expectations that some entities, such as operating systems and device drivers, have access to change these settings.

Some configuration data can only be changed through defined interfaces controlled by platform-level code. For example, UEFI runtime variables fall into this category. This basic level of protection guards against attackers directly modifying configuration data, and allows Platform Firmware to validate input before committing changes to storage. However, entities may be able to use these defined interfaces to make well-formed, but malicious, configuration changes.

To guard against such tampering, changes to some particularly sensitive configuration data may require authorization before being applied using the defined interfaces described above. In some cases, platform devices, such as host processor boot firmware or service processor firmware, may be capable of authenticating Platform Administrators prior to allowing them to make changes. Other authentication techniques may allow Platform Firmware to cryptographically verify the source and integrity of changes.

Some critical data is managed by the firmware with no programmatic exposure through external interfaces (e.g., wear leveling data) and if lost or damaged can result in permanent loss of service of the device. This type of state data needs to be protected at the highest level and cannot be writeable from the rest of the platform.

2.2.2.2 Non-Critical Data

Non-critical data may be used for various purposes, including:

- **Informational / UI:** Data which is merely informational or used as part of a user interface (UI) for the end user
  Example: An asset tag name of "Property of NIST" is displayed during boot
- **State:** State settings which do not affect the integrity of the platform
  Examples: The state of the Num Lock key upon system boot; whether the BIOS performs a fast boot or standard boot

Non-critical data should not be critical to the secure booting or operation of a platform. In practice, all data consumed by platform firmware may be security-sensitive, including some data that does not directly impact the correct and secure operation of the platform. Errors or malicious attacks in any data consumed by platform firmware could expose and exploit vulnerabilities in that code. As such, particular care needs to be given to any non-trusted input or data consumed by the platform.

3 Principles and Key Concepts

This section provides a brief description of the driving principles for platform resiliency which provide the foundation for the guidelines in this document. It also discusses major architectural concepts and considerations used throughout the document.

3.1 Principles Supporting Platform Resiliency

The security guidelines in this document are based on the following three principles:

- **Protection:**
  Mechanisms for ensuring that Platform Firmware code and critical data remain in a state of integrity and are protected from corruption, such as the process for ensuring the authenticity and integrity of firmware updates.

- **Detection:**
  Mechanisms for detecting when Platform Firmware code and critical data have been corrupted or otherwise changed from an authorized state.

- **Recovery:**
  Mechanisms for restoring Platform Firmware code and critical data to a state of integrity in the event that any such firmware code or critical data are detected to have been corrupted, or when forced to recover through an authorized mechanism. Recovery is limited to the ability to recover firmware code and critical data.

The technical guidelines found in Section 4 are organized around these principles. The first principle, protection, is similar in scope and purpose to the guidelines found in NIST SP 800-147, *BIOS Protection Guidelines* [1]. The basic principle of protection is expanded in this document to apply to a broader set of firmware and configuration data within the platform.

While protection mechanisms are intended to prevent destructive or malicious attacks against platform firmware and critical data, these mechanisms may be imperfect or impractical to implement on all categories of devices. In those cases, detection and recovery mechanisms are intended to discover and remediate attacks to regain normal and secure operation on the device.

3.2 Resiliency Properties

The technical guidelines in this document are written in terms of guidelines for individual platform devices in order to make them broadly applicable to a variety of devices, platforms and systems. Despite the narrow focus on devices, the intent of this document is to establish guidelines supporting overall resiliency in systems against destructive attacks by ensuring that the underlying platform is resilient.

Platforms may not be able to fully provide the protection, detection and recovery capabilities for all platform devices. A loss of functionality in even one device may be sufficient to render the complete system permanently inoperable if that particular device plays a crucial role in booting or operating the platform. For a platform as a whole to claim resiliency to destructive attacks, the set of platform devices necessary to minimally restore operation of the system, and sufficient to restore reasonable functionality, should themselves be resilient. We call this set of devices *critical platform devices*. The particular resiliency properties may vary from platform-to-platform.

Protected
For a platform to be considered *Protected*, all critical platform devices must meet the protection guidelines found in Sections 4.1 and 4.2, but may not fully offer capabilities to recover the device's firmware and/or critical data.

Recoverable
For a platform to be considered *Recoverable*, all critical platform devices must provide the means to detect corruption as described in Sections 4.1 and 4.3, and provide the means to recover from this corruption in compliance with the guidelines in Sections 4.1 and 4.4.

Resilient
For a platform to be considered *Resilient*, all critical platform devices must meet all of the guidelines in Section 4. Non-critical devices should also meet these requirements or at least be designed such that a compromise of one of these devices will not impact the security of the platform as a whole. Resilient platforms will attempt to prevent attacks capable of disrupting the correct operation of the platform, while also providing mechanisms to detect and recover from malicious of accidental problems that occur.

3.3 Roots of Trust and Chains of Trust

The security mechanisms described in this document are founded in Roots of Trust (RoT). A Root of Trust is an element that forms the basis of providing one or more security-specific functions, such as measurement, storage, reporting, recovery, verification, and update. A RoT must be designed to always behave in the expected manner because its proper functioning is essential to providing its security-specific functions and because its misbehavior cannot be detected. A RoT is typically just the first element in a Chain of Trust (CoT) and can serve as an anchor in such a chain to deliver more complex functionality. The responsibilities and capabilities of a RoT may be implemented entirely within the RoT or may be performed by a delegate or agent spawned by its RoT via a chain of trust anchored in the RoT. For example, a RoT for recovery (RTRec), when triggered, will initiate a recovery process by launching another element that determines an appropriate recovery sequence and launches a chain of successive elements that perform the recovery actions. Figure 2 provides a high-level description of how trust chains are established from an initial RoT.

Generally, successive elements are cooperative in maintaining the chain of trust started by the RoT. Components in a chain of trust are privileged to perform security critical functions like performing device updates that are not available to less trusted software. RoTs and CoTs may have mechanisms to relinquish these privileges once the security function is complete, or if it is determined that the security function is not required. A CoT may also relinquish privileges before passing control to a non-cooperative element.

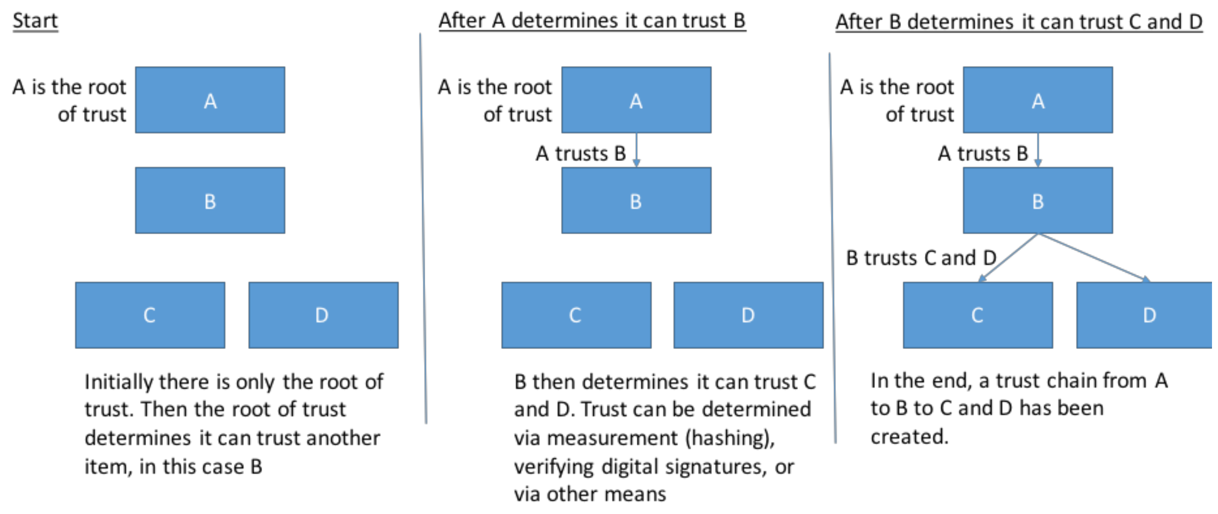
Start
A is the root of trust
A
B
C
D
Initially there is only the root of trust. Then the root of trust determines it can trust another item, in this case B
After A determines it can trust B
A is the root of trust
A
A trusts B
B
C
D
B then determines it can trust C and D. Trust can be determined via measurement (hashing), verifying digital signatures, or via other means
After B determines it can trust C and D
A is the root of trust
A
A trusts B
B
B trusts C and D
C
D
In the end, a trust chain from A to B to C and D has been created.

Figure 2: Roots of Trust

Because RoTs are essential to providing critical security functions, they need to be secure by design. Major considerations for determining confidence in RoTs are an analysis of the attack surface of a RoT and an evaluation of the mitigations used to protect that attack surface. The responsibility of ensuring the trustworthiness of a RoT is on the vendor which provides the Root of Trust. Vendors typically protect RoTs by either making them immutable, or by ensuring that the integrity and authenticity of any changes to RoTs are verified prior to performing such updates. Often, RoTs run in isolated environments, at greater privilege level than anything which could modify it, and/or complete their function before anything can modify it to ensure that other devices cannot compromise their behavior during operation.

Section 4.1 of this document provides specific guidelines on the capabilities and properties of the RoT that support platform resiliency.

Platforms are often composed of numerous devices, often with isolation boundaries between devices and different manufacturers. A platform may need multiple independent RoTs and CoTs to provide comprehensive coverage for resiliency. For example, a hard disk controller may have a separate microcontroller and firmware than the host platform. Both the hard disk controller and the host platform may need their own independent chain of trust for recovery if their individual critical data become corrupted.

3.4 Device Relationships

Due to lack of capability or functionality, some platform devices may not have their own root(s) of trust to perform an update, detection, or recovery. We refer to devices needing assistance as symbiont devices and those lending assistance as host devices. A dependency may be established whereby a host device and a symbiont device jointly fulfill the guidelines for protection, detection and/or recovery that the symbiont device cannot fulfill independently. Such dependencies might leverage a secure communication channel or other techniques. To be effective at lending assistance, the host device needs to meet the guidelines itself for the

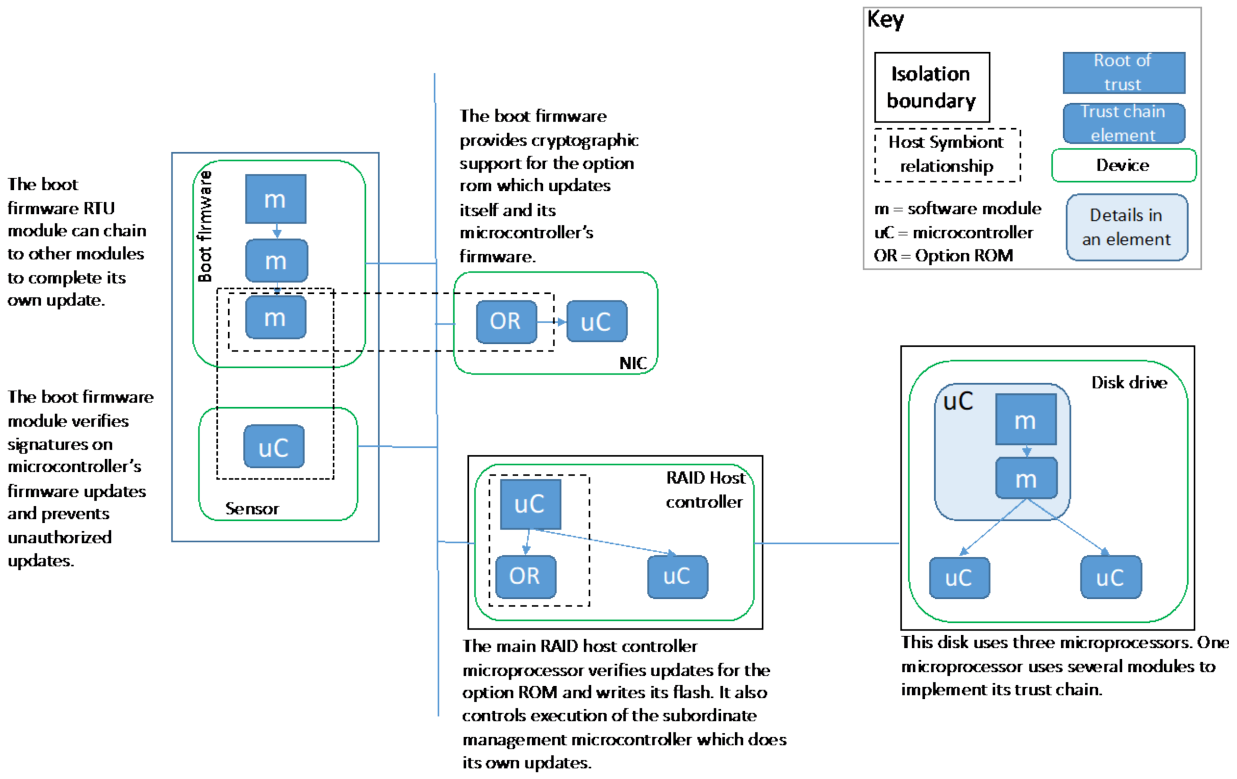
Key
Isolation boundary
Root of trust
Trust chain element
Host Symbiont relationship
Device
m = software module
uC = microcontroller
OR = Option ROM
Details in an element
The boot firmware RTU module can chain to other modules to complete its own update.
Boot firmware
m
uC
Sensor
The boot firmware module verifies signatures on microcontroller's firmware updates and prevents unauthorized updates.
The boot firmware provides cryptographic support for the option rom which updates itself and its microcontroller's firmware.
OR
NIC
RAID Host controller
The main RAID host controller microprocessor verifies updates for the option ROM and writes its flash. It also controls execution of the subordinate management microcontroller which does its own updates.
Disk drive
This disk uses three microprocessors. One microprocessor uses several modules to implement its trust chain.

Figure 3: Trust Chains

mechanisms it helps convey to the symbiont device. Together, the host and symbiont device provide a CoT that implements the security guidelines for protection, detection and/or recovery.

There may be relationships between devices where the trust is implicit— that is, where trust is provided by the architecture of the system. A device may receive indication of unambiguous physical presence from a device where an implicit trust relationship already exists. The fact that the other device sent the message through a trusted path means that the device can trust the request.

The diagram in Figure 3 shows different aspects of the relationship between symbiont and host; this relationship can be within an isolation boundary or across isolation boundaries across devices. It also shows how different devices co-exist together with several roots of trust, several chains of trust and communication paths.

There may also be other relationships which do not imply nor require any level of trust. Consider a device responsible for receiving updates. That device may then propagate those updates to other devices. Since each device (or the symbiont device along with its host device) is responsible for verifying its own updates, there is no requirement for trust between the device distributing updates and the devices those updates are provided to.

3.5 Firmware Update Mechanisms

A central tenet to the firmware protection guidelines is ensuring that only authentic and authorized firmware update images may be applied to platform devices. An update image is authentic if the source (e.g., the device, system manufacturer, or another authorized entity) and integrity can be successfully verified. Technical processes to verify images before applying updates are called *authenticated update mechanisms*.

Authorization, however, is the permission to perform an update. While authentication is typically rooted in the device or system manufacturer, authorization to perform updates is typically rooted in the device or system owner.

3.5.1 Authenticated Update Mechanism

An authenticated update mechanism employs digital signatures to ensure the authenticity of the firmware update image. An update of the firmware image using an authenticated update mechanism relies on a Root of Trust for Update (RTU) that contains a signature verification algorithm and a key store that includes the public key needed to verify the signature on the firmware update image. The key store and the signature verification algorithm are stored in a protected fashion on the computer system and are modifiable only through use of an authenticated update mechanism or a secure local update mechanism.

The key store in the RTU includes a public key used to verify the signature [7] on a firmware update image or includes a hash [6] of the public key if a copy of the public key is provided with the firmware update image. In the latter case, the update mechanism hashes the public key provided with the firmware update image and ensures that it matches a hash which appears in the key store before using the provided public key to verify the signature on the firmware update image.

It is possible that the private key corresponding to the public key in the key store may become “compromised”, for example, by the private key being stolen and exposed. An attacker that gains access to this key could sign invalid firmware that might damage platform devices or inject malware into the platform. Proper use of signatures thus necessitates provisions to recover from a key compromise. A variety of techniques may be used to recover from these situations. Examples range from the complex, including key hierarchies, to simpler, including updating the key store when recovering (or updating) the rest of an image.

3.5.2 Authorized Update Mechanism

A system and its supporting management software and firmware may provide several authorized mechanisms for legitimately updating a firmware image. These include:

1) **User-Initiated Updates**: Vendors typically supply end users with utilities capable of updating a firmware image. This could be from external media to perform these updates, or via utilities that can update the firmware image from the user’s normal operating system. Depending on the security mechanisms implemented on the system, these utilities might directly update the firmware image or they may schedule an update for the next system reboot. The updated code will encounter critical data written by a different

revision of the code. The updated code should ensure that the platform continues to function by remaining compatible with the critical data, by updating the critical data to be compatible with the updated code, or, at least by resetting the critical data values to their defaults.

2) **Managed Updates**: A given computer system may have hardware and software-based agents that allow a system administrator to remotely update the firmware image without direct involvement from the user.

3) **Rollback**: Implementations that authenticate updates before applying them may also check version numbers during the update process. In these cases, the firmware image may have a special update process for rolling back the installed firmware to an earlier version. For instance, the rollback process might require the physical presence of the user. This mechanism guards against attackers installing old firmware with known vulnerabilities.

4) **Manual Recovery**: To recover from corrupt or malfunctioning firmware, computer systems may provide mechanisms to allow a user with physical presence during the boot process to replace a firmware image with a known good version and configuration.

5) **Automatic Recovery**: Some computer systems are able to detect when a firmware image has been corrupted and recover from a backup firmware image stored in a location separate from the corrupted image (e.g., a second flash memory chip, a protected region of a storage device).

3.5.3 Secure Local Update

While this document recommends firmware updates be done through an authenticated update mechanism as described in Section 3.5.1, some devices may also support a secure local update mechanism. These mechanisms instead authorize firmware updates through a process demonstrating unambiguous physical presence. A secure local update mechanism can be used, for example, to recover a corrupted firmware image that cannot be updated using an authenticated update or an automated recovery mechanism. The secure local update mechanism could also be used by a physically-present administrator to update to an earlier firmware image on a device that does not allow rollback.

To protect against remote attacks exploiting secure local update mechanisms, it is important that these mechanisms verify that a user has physically authorized the update. Remote mechanisms, such as interacting with a device or system via a remote console, do not satisfy this requirement for physical presence. Similarly, mechanisms that can be spoofed by malware running on the system or device do not satisfy this requirement. See Section 3.6.2 for more details.

However, note that devices that implement the secure local update mechanism are potentially vulnerable to attacks by rogue administrators or other attacks with physical access to the device or system. Additional physical, environmental and technical security measures are essential to protecting these devices, but they are beyond the scope of this document.

3.6 Other Considerations for Platform Resiliency

This document does not address certain other considerations that a purchaser, user, or IT administrator may take into account pertaining to platform cyber resiliency. A non-exhaustive list and discussion of these other considerations follows.

3.6.1 Management

Vendors should carefully consider their target customers when designing resilient platforms to ensure proper management and control of policies and configuration settings can be administered in the way which best serve customer needs. Management of policies and configuration settings can be performed either locally or remotely. Depending on platform type, customers may expect the capability to fully administer a platform securely from a remote location. Some customers may expect to require a physically present user to approve a change in policy. Other customers may expect to be able to remotely extract any log data, or they may wish to prevent the exfiltration of log data except through authorized local mechanisms.

3.6.2 Authorization Mechanisms

Some recovery and administrative actions can make significant changes to either the Platform Firmware or software. For example, firmware settings may control the boot order, and a software recovery agent may restore a backup erasing recently created data. Modifying these settings could require Platform-level Authorization to demonstrate that the entity requesting a change is authorized to do so. For some environments, like large organizations or data centers, a professional Platform Administrator may authorize actions remotely using credentials provisioned to manage the platform. In other environments, e.g., consumers or smaller enterprises, there may not be a remote Platform Administrator. Some systems, however, may have Platform Administrator credentials that can be used locally. Alternatively, some systems may allow users to assert platform-level authorization by ensuring that a physically-present user has issued a command or requested a change. On these systems, the platform must unambiguously verify that a physically present user has authorized the action. If done correctly, malware cannot impersonate an authorization check that involves confirmation from a physically present user. We use the term Unambiguous Physical Presence to indicate a local user that cannot be impersonated by malware.

Unambiguous Physical Presence allows for assertion of Platform-level Authorization (or a portion of Platform-level Authorization) by demonstrating that a person is physically interacting with a device or platform. By ensuring that recovery actions or critical data changes are authorized by a physically present person, Unambiguous Physical Presence provides a management path that is intended to be protected from influence by malware.

Creating platforms and devices that properly and reliably verify confirmation of a physically present person is complex. Dedicated physical buttons or hardware jumpers could provide a relatively direct and explicit method by which to demonstrate physical presence. Platform design or deployment considerations may prevent a person from having direct physical mechanisms to interact with each device supporting a function that relies on Unambiguous Physical Presence. In these cases, there will need to be a trusted path between the mechanisms used to verify Unambiguous Physical Presence, and the device which will perform an action on behalf of the

Administrator. To satisfy the non-bypassability guidelines found later in this document, this trusted path, which could include I/O devices (e.g., human interface device, graphics card, etc.) and internal buses, needs to be protected from manipulation by malware.

There are a variety of techniques that could provide a trusted path between a physical mechanism that verifies Unambiguous Physical Presence and a platform or device. One example could be to accept or confirm commands from a physically-present person only when the platform can be trusted to be in a state with integrity, before malware could disturb these processes, such as early in the boot process. In other cases, system architectures may provide trusted paths between a Service Processor (e.g., EC, BMC) and other platform devices.

Devices which rely on Unambiguous Physical Presence in place of Platform Administrator credentials to authorize administrative actions may be vulnerable to attacks by individuals with physical access to the device. As such, its usage may not be suitable in applications or environments that lack strong physical security.

3.6.3 Network-Assisted vs. Local Recovery

In most cases, the ability to recover locally from corruption will be the most expedient, provide the highest level of customer satisfaction, and may be necessary if there is no network connectivity. But it is recognized that this is not always possible, particularly given storage limitations that many devices will have. In instances where local recovery is not possible, network-assisted recovery can be implemented if done in a secure and trustworthy manner, which may include the use of encryption, digital signatures, secure transport methods, etc. While either local or network-assisted recovery are acceptable implementation mechanisms, the ability for a device to support both provides for an even higher level of resiliency and is therefore recommended.

3.6.4 Automated vs. Manual Recovery

Recovery can proceed in one of three ways:

1. ***Fully automated*** -- no user interaction required to initiate recovery or during recovery process
2. ***Partially automated*** -- recovery initiated automatically but requires user interaction at some point during the recovery process
3. ***Manual*** -- user interaction required to initiate recovery

Fully automated recovery mechanisms may be preferred by some users, as this can allow for faster recovery at scale in the event of a widespread attack.

Fully automated recovery may not be supported by all systems or desired by all users. For example, systems may require administrative credentials or authorization to continue with the recovery process.

Manual recovery may be preferred by some users so that a Platform Administrator is informed that something is wrong, and then wait for that administrator to decide what steps will be taken next. This can also be useful in the event that a Platform Administrator wishes to capture information in order to help with forensic analysis.

Administrator-defined policies typically define the behavior and privilege requirements for manual recovery. Such policies may also affect automatic recovery. For example, an administrator-defined policy may constrain the versions of firmware that may be installed during the recovery. Those who set recovery policies must do so with care. The firmware version set in the policy might well be the version that was successfully attacked necessitating the recovery. Simply rewriting the vulnerable version may lead to an attack/recovery cycle.

Policy itself may be a target of attack so the design of the recovery implementation must account for the possibility that policy is not available. Also, since recovery can be a multi-step process, a policy requirement that will be met by the end of the recovery process might not be met during intermediate steps.

Recovery schemes which write over firmware images might, in the process, destroy evidence that would be useful in the analysis of the attack. Recovery schemes should, where practical, provide methods to retain or record attacked images and other information in cases where recovering the firmware may lose that information.

3.6.5 Event Logging

Logging firmware and recovery-related events can often be useful for multiple purposes, including but not limited to:

- Forensic analysis which allows for a Platform Administrator of a system to capture information which might have led to an attack on a platform or actual platform compromise. This can be useful in determining if the platform might contain an unknown security vulnerability, or understanding if there might be a widespread attack of a similar nature.
- Providing an audit trail to know when an event has occurred and if an update or recovery was authorized, who authorized it and when.

Platform and device manufacturers need to determine what level of event logging might be required for their systems, taking into account the intended users' environments for those systems. Events which are logged should be recorded in a manner which provides assurances of their integrity and allows for the secure recovery and transmission of logged events. Care must be taken to ensure event log access is controlled. Unauthorized personnel can use event log data analysis to broaden the attack surface.

4 Firmware Security Guidelines for Platform Devices

This section details the technical security guidelines for devices in a platform for each of the three elements of resiliency: protection, detection, and recovery. Devices may implement the requirements in one or more of these sections based on the firmware resiliency properties, as defined in Section 3.2, they aim to support. Section 4.1 provides foundational security guidelines for the Roots of Trust that support those properties. Section 4.2 provides security guidelines for the protection of firmware code and critical data. Guidelines for mechanisms to detect unauthorized changes to firmware and data are described in Section 4.3. Finally, Section 4.4 specifies security guidelines for firmware and data recovery mechanisms.

While the guidelines are written in terms of applying to individual devices, a device may implement these guidelines with assistance from another device. Security functionality may be done by the device itself (self-contained) or it may rely upon a security architecture whereby another platform device provides some or all those security functions for that device. The reliance on another device to provide necessary security functionality demands a critical trust relationship between these devices, as described in Section 3.4. These guidelines refer to the device relying on security functionality from another device as a *symbiont*, and the device providing that functionality for the symbiont as a *host device*. In these cases, the symbiont and host together form the Root of Trust or Chain of Trust responsible for implementing the security functions. The host device must additionally meet all of the requirements for a self-contained device.

The use of **shall**, **should**, and **may** are used as defined in RFC 2119 [5].

4.1 Roots of Trust

This section provides foundational guidelines on the Roots of Trust (RoT) and Chains of Trust (CoT) that support the subsequent guidelines for Protection, Detection, and Recovery. These guidelines are organized based on the logical component responsible for each of those security properties:

- The ***Root of Trust for Update (RTU)*** is responsible for authenticating firmware updates and critical data changes to support platform protection capabilities.
- The ***Root of Trust for Detection (RTD)*** is responsible for firmware and critical data corruption detection capabilities.
- The ***Root of Trust for Recovery (RTRec)*** is responsible for recovery of firmware and critical data when corruption is detected, or when instructed by an administrator.

Note that these are logical components which need not be distinct. In many cases, RoTs will be part of low-level platform firmware, and will share many components with one another. Furthermore, while each RoT is responsible for the functions necessary to support a given resiliency property, in most cases it will not implement all of those functions within the RoT itself. As described in Section 3.3, most of those functions will be implemented in a CoT anchored in the RoT. The RoT is the inherently trusted component within that chain, and extends trust to other components in a secure manner.

4.1.1 Roots of Trust (RoT) and Chains of Trust (CoT)

1) The security mechanisms **shall** be founded in Roots of Trust (RoT).
2) If Chains of Trust (CoT) are used, a RoT **shall** serve as the anchor for the CoT.
3) All RoTs and CoTs **shall** either be immutable or protected using mechanisms which ensure all RoTs and CoTs remain in a state of integrity.
4) All elements of the Chains of Trust for Update, Detection and Recovery in non-volatile storage **shall** be implemented in platform firmware.

   ***Note:** This guideline that RoTs and CoTs be implemented as part of platform firmware applies only to elements that implement the platform resiliency functions described in this paper. Platform vendors are encouraged to maintain a chain of trust from boot firmware through the Operating System to provide resiliency against various forms of attacks.*

5) The functions of the RoTs or CoTs **shall** be resistant to any tampering attempted by software running under, or as part of, the operating system on the host processor.
6) Information transferred from the software on the host processor to the platform firmware **shall** be treated as untrusted.
7) CoTs **may** be extended to include elements that are not from non-volatile storage. Before use, those elements **shall** be cryptographically verified by an earlier element of the CoT.
8) RoTs and CoTs that cross device boundaries, or that provide services to a symbiont device, **shall** use a secure communication channel between devices.

4.1.2 Root of Trust for Update (RTU) and Chain of Trust for Update (CTU)

1) Each platform device with mutable firmware **shall** rely on either a Root of Trust for Update (RTU), or a Chain of Trust for Update (CTU) which is anchored by an RTU, to authenticate firmware updates.
2) If the RTU or CTU is mutable, then the RTU or CTU elements **shall** be updated using an authenticated update mechanism, absent physical intervention through a secure local update. During such an update, the RTU or CTU **shall** always be operational or recoverable upon a subsequent reboot even in the event of an unexpected, catastrophic event (e.g., power loss in the middle of a flash write operation).
3) The RTU or CTU **shall** include a key store and an approved digital signature algorithm implementation from FIPS 186-4 [7] to verify the digital signature of firmware update images.
4) If the key store is updateable, then the key store **shall** be updated using an authenticated update mechanism, absent unambiguous physical presence through a secure local update.

   ***Note:** Updatable key stores provide a means to recover from compromise of the signing key, but may make the device's key store more vulnerable to tampering. Implementers that use a non-updatable key store are encouraged to design* *mitigations and recovery mechanisms that address the threat of potential disclosure of the firmware signing keys.*

5) An authenticated update mechanism anchored in the RTU **shall** be the exclusive means for updating device firmware, absent unambiguous physical presence through a secure local update.

4.1.3 Root of Trust for Detection (RTD) and Chain of Trust for Detection (CTD)

1) Each platform device which implements a detection capability **shall** rely on either a Root of Trust for Detection (RTD), or a Chain of Trust for Detection (CTD) which is anchored by an RTD, for its detection.
2) The RTD or CTD **shall** include or have access to information necessary to detect corruption of firmware code and critical data.
3) Detection mechanisms anchored in the RTD **shall** provide the detection capabilities specified in Section 4.3.

Note: *This document provides minimum requirements for detection capabilities rooted in low-level hardware and firmware to provide resiliency against destructive attacks. However, nothing in this document should be construed as disallowing other detection capabilities that are outside this trust chain.*

4.1.4 Root of Trust for Recovery (RTRec) and Chain of Trust for Recovery (CTRec)

1) Each platform device which implements a recovery capability **shall** rely on either a Root of Trust for Recovery (RTRec), or a Chain of Trust for Recovery (CTRec) which is anchored by an RTRec, for its recovery.
2) The RTRec or the CTRec **shall** perform the recovery.

Note: *RTR was not chosen as the acronym for Root of Trust for Recovery because RTR is typically used to denote Root of Trust for Reporting. As such, throughout this document we will disambiguate RTR by using RTRec to denote Root of Trust for Recovery.*

4.2 Protection

While previous efforts have addressed protection of BIOS (e.g., NIST SP 800-147 [1], NIST SP 800-147B [2]), there remains other security-critical firmware in the platform that has not been addressed. This includes firmware resident in management controllers, service processors, storage devices, network controllers, and graphics processing units. Protection must also extend to critical data associated with the firmware being protected, as some of this data could be a vector of attack which can compromise the integrity of the platform.

All platform devices which provide protection of firmware code and critical data must meet the requirements which follow.

4.2.1 Protection and Update of Mutable Code

This section specifies guidelines for firmware protection based on the principles of authenticated firmware updates, integrity protection, and non-bypassability of security mechanisms. Authenticated update mechanisms use digital signatures to verify the integrity and authenticity of firmware update images. Firmware integrity protections prevent unintended or malicious modification of firmware outside the authenticated firmware update process. The final principle, non-bypassability, ensures that there are no means for an attacker to bypass the protective mechanisms.

4.2.1.1 Authenticated Update Mechanism

One or more authenticated update mechanisms anchored in the RTU **shall** be the exclusive means for updating device firmware, absent unambiguous physical presence through a secure local update, as defined in Section 3.5.3. Authenticated update mechanisms **shall** meet the following authentication guidelines:

1) Firmware update images shall be signed using an approved digital signature algorithm as specified in FIPS 186-4 [7], *Digital Signature Standard*, with security strength of at least 112 bits in compliance with SP 800-57, *Recommendation for Key Management – Part 1: General* [8].
2) Each firmware update image shall be signed by an authorized entity – usually the device manufacturer, the platform manufacturer or a trusted third party - in conformance with SP 800-89, *Recommendation for Obtaining Assurances for Digital Signature Applications* [9].
3) The digital signature of a new or recovery firmware update image shall be verified by an RTU or a CTU prior to the non-volatile storage completion of the update process. For example, this might be accomplished by verifying the contents of the update in RAM and then performing an update to the active flash. In another example, it could also be accomplished by loading the update into a region of flash, verifying it, and then selecting that region of flash as the active region.

4.2.1.2 Integrity Protection

To prevent unintended or malicious modification of the firmware, nonvolatile storage regions containing device firmware need to be protected from such modifications outside of an authorized update mechanism.

1) The flash regions that contain device firmware **shall** be protected so that it is modifiable only through an authenticated update mechanism or by a secure local update mechanism that ensures the authenticity and integrity of the firmware update image by requiring that an authorized user physically touch the system itself to conduct the update.

   Note: *To ensure integrity protections cannot be bypassed, integrity protections must either always be enabled, or must be engaged prior to execution of code outside of the CTU. Hardware integrity mechanisms may provide higher assurance than software or firmware-based mechanisms. These integrity* *protection mechanisms must ensure that firmware can only be modified as part of an authenticated update, or a secure local update.*

4.2.1.3 Non-Bypassability

The principle of non-bypassability is that it should not be possible for an attacker to modify device firmware outside of the authenticated update mechanism or, if supported, a secure local update. Any intended or unintended mechanisms capable of bypassing the authenticated update mechanism could create a vulnerability allowing malicious software to modify device firmware with a malicious or invalid image. These could include development or diagnostic interfaces that allow access to flash regions, architectural features that allow direct memory access, or low-level vulnerabilities that allow manipulation of memory (e.g., rowhammer attacks).

To satisfy the principle of non-bypassability, these potential vulnerabilities need to be considered in overall system design. This could include efforts to limit the attack surface of devices, careful analysis of interfaces to devices and non-standard command sets, and disabling development and diagnostic interfaces in production devices.

1) The protection mechanisms **shall** ensure that authenticated update mechanisms are not bypassed.
2) The authenticated update mechanism **shall** be capable of preventing unauthorized updates of the device firmware to an earlier authentic version that has a security weakness or would enable updates to a version with a known security weakness.

   Note: *Updates to earlier firmware versions, sometimes called "rollback," may provide a means to recover from a firmware update that is not functioning correctly. However, unauthorized rollback could allow an attacker to restore a vulnerable firmware image, which in turn could allow the attacker to damage the device or inject malware. As such, devices that support rollback should include appropriate security controls to ensure it cannot be exploited by an unauthorized entity in an attack.*

4.2.2 Protection of Immutable Code

Code could be stored in field non-upgradable memory, such as Read Only Memory (ROM). While the protections for this type of storage are strong, the trade-off is the inability to update the code to fix bugs and patch vulnerabilities. Manufacturers of systems and devices should carefully weigh the advantages and disadvantages of using nonvolatile storage that is not field upgradable.

1) If used, the write protection of field non-upgradable memory **shall not** be modifiable.

4.2.3 Runtime Protection of Critical Platform Firmware

To satisfy the principle of non-bypassability described in Section 4.2.1.3, it is important that software or bus-mastering hardware under the control of software not be capable of interfering with the intended function of *Critical Platform Firmware*. Critical Platform Firmware is the collection of all Platform Firmware that either (a) performs the functions of protection, detection, recovery and update of any Platform Firmware, (b) maintains the security of critical data or (c) implements interfaces for critical data that are non-bypassable.

Devices that claim conformance with the Protection requirements, and rely on critical platform firmware to protect the firmware image and/or critical data at OS runtime, must meet the guidelines in this subsection. The goal of these guidelines is to establish an environment for critical platform firmware to execute in which it is isolated (protected) from software. Such isolation (protection) may be provided either logically (e.g., use of System Management Mode in x86-based platforms, or TrustZone in ARM-based platforms), or physically (e.g. in RAM attached to a non-host processor which is physically or logically isolated from the host processor).

This subsection does not necessarily apply to firmware that is classified as non-critical (for instance, the majority of the BIOS on a PC-style platform is typically non-critical).

1) If Critical Platform Firmware code in non-volatile memory is copied into RAM to be executed (for performance, or for other reasons) then the firmware program in RAM **shall** be protected from modification by software or **shall** complete its function before software starts.
2) If Critical Platform Firmware uses RAM for temporary data storage, then this memory **shall** be protected from software running on the Platform until the data's use is complete.
3) Software **shall not** be able to interfere with the intended function of Critical Platform Firmware. For example, by denying execution, modifying the processor mode, or polluting caches.

> ***Note:*** *These guidelines do not preclude the use of RAM that is writable by software specifically for communication with Firmware or device hardware, including using memory as a staging area for updates. The guidelines are intended to prevent the unauthorized modification of executing code or private state used by Critical Platform Firmware.*

4.2.4 Protection of Critical Data

Unauthorized changes to critical data stored and used by devices could also seriously impact the security posture of a device. Such changes could modify or disable important security-relevant functions provided by the platform, or prevent the device from functioning at all. While critical data may need to be modifiable by operating systems and other components, the guidelines in this section aim to provide a controlled interface for these changes and guard against changes that would put the device in an invalid state.

1) Critical data **shall** be modifiable only through the device itself or defined interfaces provided by device firmware. Examples of defined interfaces include proprietary or public application programming interfaces (APIs) used by the device's firmware, or standards-based interfaces. Symbiont devices may rely on their host devices to meet this requirement.

2) Critical data updates **shall** be validated either by the device or a symbiont's host device prior to committing changes to critical data to ensure that the new data is well-formed. Examples of validation can include range or bounds checking, format checking, etc.
3) Critical data updates **shall** be authorized by a Platform Administrator or part of an authorized firmware update mechanism.
4) Critical data updates **may** employ mechanisms to authenticate the critical data before it is used.
5) The device **shall** protect its factory defaults at least as well as it protects its code. The factory defaults **shall** be able to be updated in the same manner as the code.

4.3 Detection

The detection guidelines in this section describe mechanisms which can detect unauthorized changes to device firmware and critical data before it is executed or consumed by the device. When unauthorized changes are detected, a device could initiate a recovery process, as described in Section 4.4. Detection mechanisms are particularly important for devices that lack strong protections on their firmware or critical data. However, these mechanisms can also provide a means to detect failures in firmware or critical data protection for devices that attempt to implement the guidelines in Section 4.2.

All devices which provide detection of corruption of their firmware code and critical data must meet the guidelines which follow.

4.3.1 Detection of Corrupted Code

Execution of unauthorized or corrupted firmware on a device could damage the device, inject malware in the system, or otherwise impact the security functions and capacities of a device or encompassing system. The following guidelines describe mechanisms to verify the integrity of firmware during the boot process using the Root of Trust for Detection (RTD), specified in Section 4.1.3. While cryptographic integrity checks, either by the device itself or a host device, are preferred, some hardware device (e.g., FPGAs or CPLDs) may use other mechanisms to detect corruption in their code and programmable logic.

For these detection mechanisms to be effective, the design of the device needs to ensure that the RTD remains trustworthy in the event of a successful attack on the firmware itself.

1) A successful attack which corrupts the active critical data or the firmware image, or subverts their protection mechanisms, **shall not** in and of itself result in a successful attack on the RTD or the information necessary to detect corruption of the firmware image.
2) One or more of the following techniques **shall** be used by the RTD or CTD to validate firmware code:
    a) Integrity verification, using an approved digital signature algorithm or cryptographic hash, of device firmware code prior to execution of code outside the RTD.

***Note:** Integrity verification could also be performed at runtime. These mechanisms may or may not be anchored in the RTD.*

b) Symbiont devices **may** rely on a host device to perform detection. If the symbiont device boots independently from the host device, integrity verification of the symbiont's device firmware **shall** be performed prior to execution of code outside the host CTD. In such cases, the following additional requirements apply:

   i) The symbiont's firmware **shall** be protected according to the requirements in Section 4.2.1.

   ii) The host device **should** be capable of immediately triggering a recovery of the symbiont's firmware, followed by a restart of the device, in cases where corruption is detected.

c) Certain hardware devices (e.g., FPGAs, CPLDs) may have field-upgradable logic rather than firmware code, often referred to as configuration bitstream. If these devices do not have the ability to support cryptographic verification or the ability to measure and report in conformance with a) or b), they **shall** use hardware-based mechanisms to detect device load failures.

d) Other techniques (e.g., watchdog timers) **may** be used in combination with cryptographic integrity checks to detect other problems with the initialization process of platform devices.

3) If corruption is detected, the RTD or CTD **should** be capable of starting a recovery process to restore the device firmware code back to an authentic version.

4) The detection mechanism **should** be capable of creating notifications of corruption.

5) The detection mechanism **should** be capable of logging events when corruption is detected.

6) The detection mechanisms **may** be capable of using policies set by the Platform Administrator which define the actions taken by the RTD/CTD in the above guidelines.

4.3.2 Detection of Corrupted Critical Data

This section describes mechanisms to detect invalid or corrupted critical data in platform devices. As noted above, invalid critical data could render a device inoperable or disable certain critical security functionality. Verifying critical data is challenging, because data is often intended to be user-configurable. The guidelines in this section recommend either directly verifying critical data contents or implementing other mechanisms to look for symptoms of data corruption.

1) The RTD or CTD **shall** perform integrity checks on the critical data prior to use. Integrity checks may take the form, for example, of validating the data against known valid values or verifying the hash of the data storage.

2) Either as an alternative to integrity checks (for devices that cannot support such a capability) or in addition to those checks, the RTD or CTD **may** use watchdog timers to detect potential corruption of critical data.

3) If corruption of critical data is detected, the RTD or CTD **shall** be capable of starting a recovery process to restore the device's critical data.
4) The detection mechanism **should** be capable of logging events when corruption is detected.
5) The RTD or CTD **should** be capable of creating notifications of corruption.
6) The RTD or CTD **may** be capable of forwarding notifications of corruption.

4.4 Recovery

This section describes mechanisms for restoring platform firmware and critical data to a valid and authorized state in the event that any such firmware or critical data are detected to have been corrupted, or when an administrator initiates a manual recovery process.

4.4.1 Recovery of Mutable Code

The firmware recovery guidelines in this section specify mechanisms to recover firmware to a locally-stored backup or to a recovery image downloaded from another source. In either case, these guidelines specify using an Authenticated Update Mechanism (Section 4.2.1.1) to verify the integrity and authenticity of the image prior to recovery.

1) Firmware recovery mechanisms **shall** resist attacks which corrupt the active critical data or the primary firmware image, or subverts their protection mechanisms.
   a) The RTRec, CTRec and authentic recovery firmware image **should** be protected independently of the running firmware.
2) The RTRec or CTRec **shall** be capable of obtaining an authentic device firmware image.
   a) If the authentic firmware image is stored locally in non-volatile memory, the image **shall** be protected from unauthorized modification.
3) Updates to a locally stored authentic firmware image **shall** be by way of an Authenticated Update Mechanism (Section 4.2.1.1) or a secure local update (Section 3.5.3).
4) Non-local recovery mechanisms **shall** use an Authenticated Update Mechanism (Section 4.2.1.1) to verify the integrity and authenticity of recovery images prior to restoring them.
5) If the authentic firmware image is stored remotely, the recovery policies **shall** be configurable with the location of this image.
6) If the device (a *symbiont device*) relies upon another platform device (a *host device*) to provide its RTRec or CTRec, then the host device's RTRec or CTRec **shall** invoke the host/symbiont Authenticated Update Mechanism during recovery operations.
7) The device **shall** either implement its own recovery capability, or that device (a *symbiont device*) and another platform device (a *host device*) **shall** together implement the symbiont device's recovery capability.
8) The recovery mechanism **should** be capable of logging and reporting events when recovery is performed.

9) The recovery mechanism **should** be capable of providing notifications of recovery events and actions.
10) The recovery mechanism **may** be capable of performing the recovery action without notification or intervention by the user or system administrator.
11) The recovery mechanism **may** request approval from the user or system administrator to perform a recovery action.
12) The platform administrator **should** be able to initiate recovery of mutable code. Devices **should** provide a method for Platform Administrators to force recovery. Devices **may** receive platform-level authorization to force recovery through a chain of one or more trusted devices, the first of which **shall** verify platform-level authorization prior to instructing downstream devices to recover.
13) The recovery process **should** protect against unauthorized recovery to an earlier firmware version that contains a security weakness. The overall recovery process **should** facilitate recovery to a recent firmware version. These **may** be implemented as a multi-stage recovery process.

4.4.2 Recovery of Critical Data

This section describes mechanisms to recover critical data in platform devices in the event that the device or administrator has reason to believe it has been corrupted. Because critical data can be user-configurable, recovery requires the availability of trusted, known-good backup copies of critical data. These backup copies may be stored on the device itself or by some other host device. Because these backups may also be vulnerable to attack, these guidelines specify that devices also provide a means to restore to known-good factory defaults.

1) Mechanisms to recover critical data **shall** resist attacks which corrupt the active critical data or the Primary Firmware Image, or subvert their protection mechanisms.
2) The device **should** provide a method to backup a known good copy (or copies) of the active critical data to another location or locations. The protections on those locations **shall** be at least as good as that for the active critical data. The protections **should** be better than those for the active critical data.
    a) A symbiont device that cannot backup its own critical data **should** make its critical data available to its host device. In this case, the host device shall backup the symbiont's data.
    b) If a symbiont provides its critical data to a host device so the host device may backup the data, then the symbiont **should** be able to consume the recovered critical data.
3) The device **may** determine that its critical data is "known good" by using that data as part of a successful reboot.
4) The device **shall** backup the critical data either automatically or when instructed by the user or when instructed to do so by another trusted device.
5) The RTRec or CTRec **shall** be capable of recovering critical data to factory defaults.
6) The RTRec or CTRec **should** be capable of recovering to last known good critical data.

7) A device **shall not** use policies stored as critical data by that device to recover its own critical data. However, a symbiont **may** rely on policies which are provided by a host device.
8) If multiple back-ups are available, the RTRec or CTRec **may** allow a choice of which back up to use.
9) If detection of corruption of critical data is automatic, the RTRec or CTRec **may** gain approval from a host device or the user before replacing the current critical data.
10) In the absence of the RTD or CTD triggering recovery actions, the platform administrator **should** be able to initiate recovery of critical data. Devices **should** provide a method for Platform Administrators to force recovery. Devices which receive authorized requests to force recovery **may** then instruct other devices with which there are established trust relationships to force recovery, either directly or through chains of trusted devices.

While it is outside the scope of this document to define what constitutes an appropriate state for a platform to recover to (other than a state of integrity), examples include any of the following:

- Recover to a last known good state
- Reset to factory defaults
- Update to the newest firmware image
- Perform a partial ‘repair’ operation
- Recover to an enterprise-defined ‘starting point’
- Any combination of the above

Note that recovery processes may require multiple stages before normal operation is restored. For example, devices may initially restore factory-default configuration data prior to recovering last-known-good configuration data.

When considering how to determine which of the above states of integrity to recover a device to, using a policy-based approach necessitates the use of critical data in order to store/maintain those policies. However, if that critical data becomes corrupted, then the recovery process may either not be able to happen or it may happen incorrectly. As such, vendors should carefully consider the algorithm used to recover. As an example, a straightforward mechanism might be the use of a simple Most Recently Used (MRU) algorithm, e.g., the algorithm might first try to recover to the last known good state; if that data is not valid, then it might next try recovering to a prior saved state; if that is not available, then it might try recovering from a remote enterprise storage location; if that is not available, then it might try resetting to factory defaults. Using an algorithmic approach in this manner eliminates the need to rely on critical data being in a state of integrity during the recovery operation.

Appendix A—Acronyms

Selected acronyms and abbreviations used in this paper are defined below.

BIOS	Basic Input/Output System
CoT	Chain of Trust
CPLD	Complex Programmable Logic Device
CTD	Chain of Trust for Detection
CTRec	Chain of Trust for Recovery
CTU	Chain of Trust for Update
FPGA	Field-Programmable Gate Array
ROM	Read Only Memory
RoT	Root of Trust
RTD	Root of Trust for Detection
RTRec	Root of Trust for Recovery
RTU	Root of Trust for Update
UEFI	Unified Extensible Firmware Interface

Appendix B—Glossary

Active Critical Data	The copy of critical data that is used to initialize or configure a device. *Note: active critical data does not include back-ups of critical data.*
Add-in Card	A generic term used to refer to any device which can be inserted or removed from a platform through a connection bus, such as PCI. Add-in cards are typically inserted within a platform's physical enclosure, rather than residing physically external to a platform. An add-in card will have its own devices and associated firmware, and may have its own Expansion ROM Firmware.
Authenticated Update	An update which uses an authenticated update mechanism.
Authenticated Update Mechanism	An update mechanism which ensures that an update firmware image has been digitally signed and that the digital signature can be verified using one of the keys in the key store of the Root of Trust for Update (RTU) before updating the firmware image.
Authorized Update	An update which uses an authorized update mechanism.
Authorized Update Mechanism	An update mechanism that checks for approval before installation of an update. Approval could consist of checking possession of a credential, confirmation by a physically present person or similar means.
Boot Firmware	Generic term to describe any firmware on a platform executed to boot (start-up, initialize) a device. This may include, but is not limited to, initialization of device memory, initialization of device registers, initialization of connectivity interfaces, health checks of the device, etc. The general purpose of boot firmware is to prepare a device or platform for normal operational use.
Chain of Trust (CoT)	A Chain of Trust (CoT) is a sequence of cooperative elements which are anchored in a Root of Trust (RoT) that extend the trust boundary of the current element by conveying the same trust properties to the next element when it passes it control. The result is both elements are equally able to fulfill the trusted function as though they were a single trusted element. This process can be continued, further extending the chain of trust. Once control is passed to code which is not, or cannot be, verified then the Chain of Trust has ended. This is also referred to as passing control to a non-cooperative element.
Code	Instructions directly executed by a processor or like device (FPGA, CPLD, etc.). Also included are instructions that are interpreted by a program. *Source code* is the human-readable instructions that are translated into code and then executed.

Corruption	Corruption is a loss of integrity of firmware code, or an error or unexpected value in critical data, which could be the result of any one of a number of different causes, including but not limited to, malicious activity (e.g. an attacker), poorly written code (e.g. buffer overflows, algorithmic error), accidental events (e.g. inadvertent user action), failure to install a security patch, unauthorized changes, or hardware-induced (e.g. signal integrity, alpha particles, power failures).
Critical Data	Critical data is mutable data which persists across power cycles and must be in a valid state that has been authorized by the Platform Administrator in order for the recovery and/or booting of the platform to securely and correctly proceed.
Critical Platform Firmware	The collection of all Platform Firmware that either (a) performs the functions of protection, detection, recovery and update of any Platform Firmware, (b) maintains the security of Critical Data or (c) implements interfaces for Critical Data that are non-bypassable.
Device	A generic term used to refer to any computing or storage element or collection of computing or storage elements on a platform. Examples of devices include a central processing unit (CPU), applications processing unit (APU), embedded controller (EC), baseboard management controller (BMC), Trusted Platform Module (TPM), graphics processing unit (GPU), network interface controller (NIC), hard disk drive (HDD), solid state drive (SSD), Read Only Memory (ROM), flash ROM, etc.
Device Firmware	The collection of non-host processor firmware and Expansion ROM firmware that is only used by a specific device. This firmware is typically provided by the device manufacturer.
Expansion ROM Firmware	Peripheral Component Interconnect (PCI) term for firmware executed on a host processor which is used by an add-in device during the boot process. This includes Option ROM Firmware, UEFI applications, and UEFI drivers. Expansion ROM Firmware may be bundled as part of the Host Processor Boot Firmware, or may be separate (e.g., from an add-in card). In this document, we will use the term Expansion ROM Firmware when referring to either Option ROM Firmware or UEFI Drivers and Applications generically.
Firmware	Generic term to describe any code stored in a chip that either resides at the reset vector (or equivalent) of the corresponding processor or which is provided as extensions to other firmware (such as Expansion ROM Firmware). General purpose operating systems that

	are stored in chips are generally not considered firmware in the scope of this document.
Host Device	A device that assists a symbiont device to establish the roots and chains of trust required to meet the protection, detection, and/or recovery guidelines in this document. The allocation of functionality between the host and symbiont devices is implementation dependent. The Host Device must, itself, meet the guidelines for its own firmware on its own. Note that this should not be confused with a Host Processor. A Host Processor may serve as a Host Device, but a Host Device is not necessarily a Host Processor.
Host Processor	A host processor is the primary processing unit in a platform, traditionally called a Central Processing Unit (CPU), now also sometimes referred to as an Application Processing Unit (APU), or a System on Chip (SoC). This is the processing unit on which the primary operating system (and/or hypervisor), as well as user applications run. This is the processor that is responsible for loading and executing the Host Processor Boot Firmware.
Host Processor Boot Firmware	Generic term used to describe firmware loaded and executed by the Host Processor which provides basic boot capabilities for a platform. This class of firmware includes Legacy BIOS, System BIOS and UEFI, as well as other implementations. Where the distinction between Legacy BIOS and UEFI is not important, the term Host Processor Boot Firmware will be used. Where the distinction is important, it will be referenced accordingly. Expansion ROM firmware may also be considered as part of the Host Processor Boot Firmware. Expansion ROM Firmware may be embedded as part of the Host Processor Boot Firmware, or may be separate from the Host Processor Boot Firmware (e.g., loaded from an add-in card). Host Processor Boot Firmware includes firmware which may be available during runtime.
Immutable	Unchangeable. In the context of this document, this refers only to the inability to make changes in the field through manufacturer intended mechanisms and/or defined interfaces. Note that a platform or device manufacturer may still be able to make changes through manufacturing or service tools directly connected to a locally (physically) present platform or device.
Legacy BIOS (Basic Input/Output System)	One form of Host Processor Boot Firmware used on x86 platforms which uses a legacy x86 BIOS structure. This form of host processor boot firmware has been or is being replaced by UEFI.
Mutable	Changeable. In the context of this document, this refers only to the ability to make changes in the field through manufacturer intended

	mechanisms and/or defined interfaces. Such mechanisms may require cryptographic mechanisms or unambiguous physical presence.
Non-critical Data	Non-critical data is mutable data which persists across power cycles but is not critical to the booting, operating, or recovery of the device.
Non-Host Processor	A non-host processor is a generic term used to describe any processing unit on a platform which is not a host processor (e.g. a microcontroller, co-processor, etc.).
Non-Host Processor Firmware	Non-host processor firmware is a generic term used to describe firmware used by any processing unit on a platform which is not a host processor.
Option ROM Firmware	Legacy term for boot firmware typically executed on a host processor which is used by a device during the boot process. Option ROM firmware may be included with the host processor boot firmware or may be carried separately by a device (such as an add-in card).
Peripheral (aka external device)	A peripheral (also known as an external device) is a device which resides physically external to a platform and is connected to a platform, either wired or wirelessly. A peripheral is comprised of its own devices which may have their own firmware. While conceptually the principles and guidelines in this document could apply equally to peripherals, they are outside the scope of this document.
Platform	A platform is comprised of one or more devices assembled and working together to deliver a specific computing function, but does not include any other software other than the firmware as part of the devices in the platform. Examples of platforms include a notebook, a desktop, a server, a network switch, a blade, etc.
Platform Administrator Privilege	Privileges required to manage the firmware and critical data on platform devices. In particular, this privilege may be needed to authorize firmware updates, change firmware configuration settings, and firmware recovery operations.
Platform Administrator	An entity with Platform Administrator Privileges.
Platform Firmware	The collection of all device firmware on a platform. In this document, the term Platform Firmware may be used when making references to the collection of all firmware used by devices on a platform.

Primary Firmware Image	The executable code stored on the device. Different parts of the primary firmware image may be protected differently.
Read Only Memory (ROM)	A memory device that once it has been initially set, cannot be overwritten through any mechanism, making that memory immutable (unchangeable).
Resilient	The capability of a system with specific characteristics before, during and after a disruption to absorb the disruption, recover to an acceptable level of performance, and sustain that level for an acceptable period of time.
Root of Trust (RoT)	An element that forms the basis of providing one or more security-specific functions, such as measurement, storage, reporting, recovery, verification, update, etc. A RoT is trusted to always behave in the expected manner because its misbehavior cannot be detected and because its proper functioning is essential to providing its security-specific functions.
Runtime Firmware	Generic term to describe any firmware on a platform active/functional or available for use during runtime (after boot has completed).
Software	Software is comprised of elements required to load the operating system, the operating system, and all user applications and user data subsequently handled by the operating system. Refer to Figure 1 for a graphical depiction.
Symbiont Device	A symbiont device is a device that relies wholly or partially on another device (a host device) to establish the roots and chains of trust required to meet the protection, detection, and recovery guidelines in this document. The allocation of functionality between host device and symbiont device is implementation dependent. For example, the host device verifies updates and backs up critical data while the symbiont device is responsible for meeting all other guidelines. The symbiont property can be transitive: A device may be a symbiont device to a host device while the two may then serve as the host to another symbiont device and so on.
System	A system is the entirety of a computing entity, including all elements in a *platform* (hardware, firmware) and *software* (operating system, user applications, user data). A system can be thought of both as a logical construct (e.g. a software stack) or physical construct (e.g. a notebook, a desktop, a server, a network switch, etc.). Refer to Figure 1 for a graphical depiction.

UEFI (Unified Extensible Firmware Interface)	One form of Host Processor Boot Firmware which uses a Unified Extensible Firmware Interface (UEFI) structure (as defined by the UEFI Forum).
UEFI Drivers	Standalone binary executables which are loaded during the boot process to handle specific pieces of hardware.
Unambiguous Physical Presence	Indicates authorization by a local person that cannot be impersonated by malware.

Appendix C—References

[1] D. Cooper, W. Polk, A. Regenscheid, and M. Souppaya, *BIOS Protection Guidelines*, NIST Special Publication (SP) 800-147, National Institute of Standards and Technology, Gaithersburg, Maryland, April 2011, 26pp. https://doi.org/10.6028/NIST.SP.800-147

[2] A. Regenscheid., *BIOS Protection Guidelines for Servers*, NIST Special Publication (SP) 800-147B, National Institute of Standards and Technology, Gaithersburg, Maryland, August 2014, 32pp. https://doi.org/10.6028/NIST.SP.800-147B

[3] Specifications, Unified Extensible Firmware Interface Forum [Web site], http://www.uefi.org/specifications [accessed 5/2/18]

[4] TPM Library Specification, Trusted Computing Group [Web site], https://trustedcomputinggroup.org/tpm-library-specification/ [accessed 5/2/18]

[5] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, Internet Engineering Task Force, March 1997, 2pp, https://doi.org/10.17487/RFC2119

[6] U.S. Department of Commerce. *Secure Hash Standard*, Federal Information Processing Standards (FIPS) Publication 180-4, August 2015, 36pp. https://doi.org/10.6028/NIST.FIPS.180-4

[7] U.S. Department of Commerce. *Digital Signature Standard*, Federal Information Processing Standards (FIPS) Publication 186-4, July 2013, 130pp. https://doi.org/10.6028/NIST.FIPS.186-4

[8] E. Barker, *Recommendation for Key Management, Part 1: General*, NIST Special Publication (SP) 800-57 Part 1 Revision 4, National Institute of Standards and Technology, Gaithersburg, Maryland, January 2016, 160pp. https://doi.org/10.6028/NIST.SP.800-57pt1r4

[9] E. Barker, *Recommendation for Obtaining Assurances for Digital Signature Applications*, NIST Special Publication (SP) 800-89, National Institute of Standards and Technology, Gaithersburg, Maryland, November 2006, 38pp. https://doi.org/10.6028/NIST.SP.800-89

[10] International Council for Systems Engineering, "Resilient Systems Working Group Charter," November 2011.

[11] R. Ross, R. Graubart, D. Bodeau, and R. McQuaid, *Systems Security Engineering: Cyber Resiliency Considerations for the Engineering of Trustworthy Secure Systems*, NIST Special Publication 800-160 Volume 2 (DRAFT), National Institute of Standards and Technology, Gaithersburg, Maryland, March 2018, 158pp. https://csrc.nist.gov/CSRC/media/Publications/sp/800-160/vol-2/draft/documents/sp800-160-vol2-draft.pdf