Secure Firmware Development Best Practices

Cloud Security Industry Summit
Supply Chain Technical Working Group

Ver 1.1, November 2020

About

This document was produced by the Cloud Security Industry Summit (CSIS). CSIS is a group of Cloud Service Providers, with a mission to align on a vision and approach to developing best-of-breed security solutions. The group includes members from top Cloud Service Providers, partnering as an industry team and evolving a coordinated approach for improving cloud security from component to system to solution. Intel facilitates the group.

For more information about CSIS's charter and scope, visit www.cloudsecurityindustrysummit.org

We would like to recognize the following contributors to the document (listed in alphabetical order):
- Ben Stoltz, Google
- Matt King, Oracle
- Nathan House, Rackspace
- Paul McMillan, Netflix
- Rob Wood, NCC Group
- Stuart Yoder, Arm
- Sumeet Kochar, Lenovo
- Shawn Chang, HardenedLinux
- Tobias Langbein, ZKB
- Yigal Edery, Kameleon

Document Purpose & Scope

This document promotes firmware development practices that will result in improved security and reliability. This guidance will be useful to software projects in general, however, the focus here is on firmware used in large scale data centers. This document was authored by members of the CSIS Supply Chain Workgroup, in collaboration with members of the Open Compute Project (OCP) Security workgroup.

While this document is not intended to call out specific implementation requirements, many of the points cited are illustrated with best practices that should be followed in order to maintain this desired level of security. These recommendations are not meant to be proscriptive. Alternate solutions that meet the requirements can be considered and exceeding them is encouraged.

Specific firmware requirements to enable server resiliency are called out in various NIST standards (e.g. 800-147B, 800-193) and are also addressed in the CSIS position paper (download link[1]).

The scope of developing a firmware project includes, but is not limited to:

- Software stored in non-volatile memory that handles low-level hardware initialization and events (e.g. power-on, reset events). The firmware may also implement the primary runtime features of the product.
- Tools for diagnostics, flashing, etc.
- Drivers for operating systems or boot loaders to be able to use the hardware/firmware, including Option ROMs.
- Firmware embedded within peripheral hardware devices.

These best practices for secure development will apply to all the above and associated development.

Firmware requirements that pertain to maintaining integrity of the firmware and ensuring the right firmware is loaded are covered in OCP-Security and includes:

- Establishing and maintaining a system's root-of-trust (RoT).
- Signing and verification of firmware images and attestation of secure boot.
- Firmware update processes and procedures.

This document does include some commentary on these topics that is cloud-provider specific, although these aspects of firmware security are not covered here, as to avoid duplication and confusion.

Throughout this document, the best practices that should be followed are marked with a checkbox (□). This is to make it easier for the reader who “just wants to get the right things done”.

[1] https://cloudsecurityalliance.org/artifacts/firmware-integrity-in-the-cloud-data-center/

Fork This Document

This document contains recommendations across multiple use cases and pulls information from many sources, including the experiences of the authors. Please use it as a resource for your own requirements, designs, and other project specific needs.
Additional contributions to this document are most welcome. Please use the OCP-hosted Github version[2] to submit your suggestions, or to fork it and create your own customized version.

[2] https://github.com/opencomputeproject/Security/blob/master/SecureFirmwareDevelopmentBestPractices.md

Why Firmware Security Matters

Firmware represents a significant threat vector for computer systems, appliances and associated infrastructure. If the first code that executes on a device when it powers on were to become compromised, then the entire system can and should no longer be trusted as secure. Firmware can be compromised through malicious attacks or unintentionally.

Firmware attacks can be launched by either subverting existing functionality, or by replacing the intended firmware with malicious code, as is the case of injecting a system with a UEFI rootkit. These such situations are commonly addressed and minimized by adding integrity and resiliency to the platform and had been cast in our previous CSIS position paper.

Firmware can also have code vulnerabilities, which could allow an attacker to remotely gain access to the device, or gain elevated privileges by calling into exploitable run-time firmware services and executing arbitrary code in a more privileged context.

Firmware can be unintentionally overwritten when hardware or software fails to protect the firmware, for example, from an operating system’s write operations.

We must always be able to fully trust the hardware to only be running intended and secure firmware. This represents the most fundamental root of trust. Achieving this goal requires increased focus and security awareness by firmware developers, which is at the core of this document.

Threat Models

It is important that there be a common understanding of a system's security goals. A formal threat model allows for a reasoned discussion regarding protection of a system's use cases. The threat model becomes more important as the overall system complexity and number of contributors increases or new use cases are considered. Those threats that are out of scope for a particular project, must also be documented.

In this paper, we are focused on improving confidence as to what code and configuration is running on a system at a given time. Conditions or events that can subvert that confidence are considered threats.

Example

For a processing element (CPU) or other programmable logic device (e.g. CPLD, FPGA), the mask ROM that implements initial software loading and execution may be vulnerable to attack through malicious or malformed external input such as code and configuration stored on an SPI flash chip, hardware configuration pins, or manipulation via a debugging interface (e.g. JTAG).

Asset	**Threat**	**Mitigation**
Initial Program Load integrity	Malicious external input (SPI	Review and harden mask ROM

(mask ROM)	flash)	code
Initial Program Load integrity (mask ROM)	Malicious external input (HW configuration pin state)	Schematic review informed by accurate component datasheet
Initial Program Load integrity (mask ROM)	Malicious external input (JTAG)	Board logic, and other operational and software environment changes, to disable JTAG in production environments

(See OCP "Common Security Threats", a work in progress at the OCP Security Wiki[3])

Capturing Security Exceptions

The practical realities of a product(s) schedule or available resources in our industry often leads to trade-offs in the design. As such, it is common to see a production cycle of compromises with respect to security and forgetting the larger goals until the next project is underway. Signs of danger can also arise at various points in the cycle and a products team appearing inured or numb to security threats can be as concerning as even minor staff turn-over, With the latter needing to rediscover security threats, often late in the schedule when changes are costly and compromise more likely.

In order to break this cycle, it is proposed that a system of formal "Security Exceptions" be instituted. A security exception clearly documents the gap between a particular implementation and the desired implementation. In some cases, a security exception will recognize flaws that must be corrected before a product is put into a production environment. Others may have mitigations that can be implemented. However the scenario, be sure to capture the gap as input to aid future product development.

The Meaning of a Signature

This section provides some background as to the meaning and value of signing firmware and related artifacts. It does not go into the details of signing operations or provide a checklist approach. The selection of signing algorithm, key strength, and the particulars of boot-time signature checking on various devices are out of scope.

At the time of writing, a complementary work is ongoing in the Open Compute Project's Security Working Group. In their paper, they mention their out-of-scope items, some of which are addressed here:

OpenCompute Project Contributions, IBM White Paper, "Best Practices for Firmware Code Signing"[4]

[3] https://www.opencompute.org/wiki/Security
[4] http://files.opencompute.org/oc/public.php?service=files&t=f4171bae8c7a32f05b0401378ee08483&download

> "Out of scope for this paper are chain of trust requirements, firmware development practices, supply chain and manufacturing processes, as well as usage of certificates, certificate authorities, and trusted application stores. Also out of scope for this paper but needing attention, is establishing trust in audit logs. Are the audit logs protected against tampering? Can the audit logs be falsified from the start? Are automated procedures in place to detect anomalous activity related to signing server audit logs?"

One needs to understand what a particular signature means. At the most lax end of the trust spectrum, a signature is applied only to satisfy a policy requirement (e.g. system will not boot without a signed primary boot image) and functions only as an integrity check prior to use. In the degenerate case, where a signing key is well known, a signature has the same meaning as an equivalent strength hash algorithm (e.g. SHA256) ensuring that the bits were read correctly, but saying nothing about them being trustworthy.

A signature should be an attestation to all of the qualities one looks for in trustworthy software. That is to say, the signature is tied to evidence that:

- access to signing related secrets and the ability to sign is tightly controlled and well understood.
- the mapping from inputs, through the build process, to the outputs must be deterministic and reproducible.
- the build and signing processes are auditable; the environments, tool chains, and other inputs that themselves have known provenance, and the people or role accounts used to execute the build are recorded as part of that process.
- The persons and processes applying signatures understand the provenance of the signed artifacts.
- The signed artifacts are easily mapped back to their provenance when that information is needed.

Audit the signing activities

Signing logs should be audited against known builds and releases. Any artifacts signed with valuable keys that do not correspond to known releases may be a cause to rotate keys and other actions.

Reproducible Builds

Reproducible builds indicate that the supplier has sufficient control over the build process and understands what the inputs to that process are. A source control system that allows bodies of source to be referenced and retrieved by a unique identifier is usually required, e.g. Git tags[5]. Snapshots or a sufficiently flexible change management system may be used to keep track of the tool chains, binary blobs like pre-compiled 3rd party libraries or microcode, and other elements of the build environment. Note that binary blobs demand significant leaps of faith and are discouraged.

Reproducible builds improve quality by assuring developers, testers, and others that the build products they are working on are actually derived from the indicated source code. When it comes time to reproduce and instrument hard-to-find bugs, this attribute is very valuable in that the developer can

[5] https://git-scm.com/book/en/v2/Git-Basics-Tagging

trust that the instrumentation changes they are making are the only variable in their test builds. For example, a developer should be able to bisect the build (see https://git-scm.com/docs/git-bisect) to isolate the change that introduced a bug.

Different Purposes of Signing

There are several points in the life-cycle of build products (firmware, configuration, release notes, etc.) where trust needs to be established or renewed. Often, a single signing operation covers all of the cases, but sometimes it is useful to separate them. An example might be a hyper-scale cloud service provider that keys systems to only respect that owner/operator's own signatures. Firmware arrives from manufacturers signed with that manufacturers key. Before that firmware can be used in the owner/operator's data centers, the manufacturer's signature must be replaced with the owner/operator's signature.

Useful times to sign, and the meaning of that signature:

- **Build artifacts during firmware development** - All of the building and signing processes are exercised on a regular basis but with keys accessible to the development team. The keys used are not accepted by end-user systems. The signature binds the inputs and artifacts.
- **Build artifacts for release** - The provenance and testing/validation of the generated products can be attested to and have been deemed suitable for release beyond the development team. In addition to any artifacts containing a boot-time, install-time, or other signature, the entire collection of artifacts, or a manifest describing those artifacts, needs to be signed. As an example of a threat; can an attacker insert a README file that directs the owner/operator to install in a way that is not secure or substitutes a script that ultimately subverts the system?
- **Delivery** - The owner/operator will accept delivery through some mechanism (ftp server, web server, emailed, full reproduction from retrieved sources and specified build environment) and must be able to check the supplier's signature on each element of a release and/or the release as a whole.
- **Deployment** - The owner/operator, especially in any large-scale data center, will have automated procedures for deploying updates to firmware, configuration, FPGA bitstreams, etc. Those procedures are likely to have their own signing process or make reference to the suppliers' signatures. It is often the case that an owner/operator will not want to deploy updates as they arrive from the supplier without going through their own quality control process. New firmware for example, can have negative interactions with other parts of a system or data center operations that do not manifest until they are tested in production. A separate installation authentication mechanism, or equivalent, can act to prohibit updates that are not yet approved.
- **Diagnostics** - When supporting deployed devices and diagnosing field issues, firmware with custom instrumentation may be required if the issues cannot be reproduced on development hardware. When signing such privileged builds, care should be taken to prevent them from being used on a wider set of systems than intended in the event that these builds are leaked. This may be accomplished by tying the build to the specific hardware instance(s) being debugged (e.g. add a check that an expected serial number is burned into the CPU).
- **Monitoring** - The owner/operator wants to know that an update has been applied and that previous versions have been expunged from the data center. Devices that can cryptographically attest to their current firmware and configuration improve the level of trust.

Secure Boot and Secure Firmware Updates

SecureBoot and Secure Updates are important from a platform security perspective because they provide high assurance in the integrity of firmware on the device. However, these are beyond the scope of writing secure firmware. They are covered by OCP-Security.

This is usually achieved through several mechanisms:

- A secure boot mechanism, which validates the signature of the intended image before it is booted.
- A secure mechanism for firmware updates, including functionality such as having update policies which verify each update before taking effect, ability to fully wipe-out old firmware in favor of a new update, and rollback prevention mechanisms to prevent an attacker from reverting to known-bad firmware.
- A mechanism for securely maintaining and attesting for the chain of trust inside the device, including implementing measurements of the firmware on a device. A measurement should consist of a cryptographically secure hash of the firmware and non-unique configuration data (device unique configuration values like serial numbers can, and in some cases should, be excluded).

All of these are within the charter of the OCP Security workgroup and are covered there. CSIS would like to avoid duplication and endorse that work. The OCP-Security documents can be found here: https://www.opencompute.org/wiki/Security

Firmware Development Best Practices

This section outlines a subset of software development best practices focused on issues relevant to firmware development.

Design

Security is best when it's integrated into the earliest phases of firmware inception, starting from the design phase. It is cheaper and easier "catch" design issues that will lead to security issues, and correct them during design, before a single line of code is written.

- ❏ Threat modelling should be performed on all components, as a whole system, at design time and as changes are made, to understand expected trust boundaries and how to mitigate potential exploits.
- ❏ Trust boundary assumptions should be documented.
- ❏ Least-privilege, de-privileging.
    - ❏ Lock critical SPI regions against updates until next reboot.
    - ❏ Block access to key material not needed by subsequent stages of firmware.
    - ❏ Limit the use of SMM / SMI Handlers to the bare-minimum necessary.
- ❏ Limit the pre-signature validation attack surfaces.
    - ❏ Avoid the need for complex parsing of image or metadata prior to signature validation (eg. encrypt/compress before signing, instead of signing before encrypt/compress operations)
    - ❏ Do not expose privileged debug, manufacturing, or diagnostic functionality until after successful authentication (when enabled).
- ❏ Understand the hardware design sufficiently to properly program and configure a secure boot sequence.
- ❏ Implement event logs for security related events & including common failures (e.g. hardware initialization fails).

Input Validation

"Input" is used to describe any commands or data that are directly or indirectly supplied by any entity not controlled by the firmware itself. This includes all commands and data supplied to the device by vendor drivers and utilities which can be modified or replaced and transactions from other hardware or firmware system components which can be malicious or spoofed, including SPI flash and other persistent memories and file systems. That is in addition to requests directly initiated by an end-user.

- ❏ External input, including configuration data, must always be sanitized or validated before use.
- ❏ Normalize strings and characters.
- ❏ Filter or escape special characters that could allow for directory traversal attacks, etc.
- ❏ Validation of metadata in updates and on flash storage to ensure malformed metadata cannot be used as an attack vector.

Memory Safety

Probably the most common sources of exploitable vulnerabilities are bugs that allow an attacker to corrupt memory and use that to execute arbitrary code. Following some basic rules can significantly reduce the likelihood of such incidents.

- ❏ Use memory safe practices.
  - ❏ Initialize all variables.
  - ❏ Always bounds check[6] buffers and arrays.
  - ❏ Check for integer overflows and underflows.
  - ❏ Use safe string and buffer functions.
  - ❏ Memory should never be both writable and executable (W^X[7]).
- ❏ Consider using memory safe languages.
  - ❏ E.g: Rust, or a safer user-space runtime for embedded systems, such as https://github.com/u-root/u-root.
- ❏ Enable applicable memory corruption exploit-mitigations, such as:
  - ❏ ASLR - Address Space Layout Randomization[8].
  - ❏ Stack protections[9].
  - ❏ MRR - Memory Range Registers.
  - ❏ Consider using fault-inducing guard pages between different memory regions, such as stacks, heaps, and code.
  - ❏ Consider the use of control flow integrity[10] protections (e.g.: shadow stack)

Concurrency

Modern devices contain multiple processing cores, either in a symmetric multiprocessing capacity, or in a distributed asymmetric model, and most commonly both. Not all processing cores will be running firmware in the same security context. Firmware running on such systems that interacts with any shared resources, such as shared memory, shared hardware registers, or shared peripheral devices and co-processors, must take care to prevent various race conditions. This can occur where a low privileged subsystem can interact with a shared resource while a high privileged subsystem is using it. Some best practices include:

- ❏ Leverage hardware interlocks where possible to prevent multiple concurrent accesses to a shared resource.
- ❏ Make local copies of externally provided data before validating and using it to avoid Time-of-Check-Time-of-Use (TOCTOU) issues. Deep copies of any indirect data structures may be required as well.

---

[6] https://en.wikipedia.org/wiki/Bounds_checking
[7] https://en.wikipedia.org/wiki/W%5EX
[8] https://en.wikipedia.org/wiki/Address_space_layout_randomization
[9] https://en.wikipedia.org/wiki/Stack_buffer_overflow#Protection_schemes
[10] https://en.wikipedia.org/wiki/Control-flow_integrity

Source Control

A robust source control system is critical for enabling reproducible builds and auditing of changes. Acceptable source control systems will include the following features or enable the following activities:

- ❏ Maintain commit history that includes the identity of the committers and intent of the commit. (e.g. by requiring commit signing[11])
- ❏ Policy enforcement, including:
  - ❏ Code reviews prior to check-ins.
  - ❏ Triggering automated testing hooks and generate testing reports.
  - ❏ Consider CI/CD - Continuous Integration / Continuous Deployment.
- ❏ Integration with issue tracking.
  - ❏ A bug database should link bugs to associated source code changes.
- ❏ Ability to reproduce any externally-facing build with the purpose of issuing targeted security fixes.

Security Code Reviews

Performing a security review of the firmware source code periodically and on check-in by security teams and/or external parties is an important part of firmware development.

- ❏ Firmware must be routinely reviewed for security issues at least in the following events:
  - ❏ Prior to release.
  - ❏ Prior to updates.
  - ❏ Retroactively for previous versions when issues are discovered. (e.g. variant hunting)
- ❏ Consider occasionally inviting staff from other departments, customers, or other third parties to code reviews to get a fresh set of eyes and to force the development team to revisit the assumptions and decisions made during code development.

3rd-Party Libraries

Use of 3rd-party libraries, including open source, is often required as part of the firmware being developed. And could become a source of security issues, so it's important to keep track of these and to follow some basic rules when using them.

- ❏ Review secure development practices of 3rd-party providers to ensure they meet or exceed the security practices outlined in this document.
- ❏ Document all 3rd-party code usage. Include references to libraries and versions at a minimum, with best practice to document specific functions used, especially for larger libraries.
- ❏ When possible, avoid using pre-compiled binaries whose source code is not available. When such binaries must be used, document them and verify their trustworthy-ness by tracking the origin of these binaries and verify their included hashes.

[11] https://help.github.com/en/articles/about-required-commit-signing

- ❏ Where possible, implement privilege separation or isolation of 3rd party binary components to limit their interaction with the rest of the software (e.g. kernel-enforced privilege separation, etc)
- ❏ Maintain an updated list of included libraries and associated versions to aid in tracking vulnerabilities, security updating and proper timely mitigation.
- ❏ When issues are discovered, and originate from 3rd-party sources, require a follow-up to understand impact and patch production.

Testing

An important part of secure development includes testing for security issues. The following practices will help find security issues before shipping firmware.

- ❏ Run static analysis[12] tools to identify issues in the code.
- ❏ Perform fuzz testing[13] of the code.
- ❏ Measure the code coverage[14] of the test suite and aim for an acceptable level.
    - ❏ E.g. Discussion on acceptable metrics here[15].
- ❏ Consider using formal methods for security-critical code sections (more details here[16])
- ❏ Automated security testing as part of check-in
- ❏ Test for all the threat vectors identified by your threat model and include external dependencies.

Capture Security Exceptions

Context is provided in the previous section. Here are the recommendations:

- ❏ Document all security exceptions and consider them in the next versions development cycle.
- ❏ Institute processes to audit security progress across versions.

Build & Compilation

Naturally, the code that actually runs in production is the compiled code. Therefore, it is important to ensure integrity of builds, and optimize compilation for security.

- ❏ Reproducible Builds (see explanation in section Different Purposes of Signing).
    - ❏ Run the build process in a secured and trusted environment.
    - ❏ Document all build dependencies, including compiler toolchains, libraries, and build scripts.
    - ❏ Ensure binaries can build identically by multiple parties (see https://reproducible-builds.org/docs/ and https://wiki.debian.org/ReproducibleBuilds).
- ❏ Compiler settings.

---

[12] https://www.owasp.org/index.php/Static_Code_Analysis
[13] https://www.owasp.org/index.php/Fuzzing
[14] https://en.wikipedia.org/wiki/Code_coverage
[15] https://en.wikipedia.org/wiki/Code_coverage#In_practice
[16] https://en.wikipedia.org/wiki/Formal_methods

    - ❏ Should be set at high if not maximum warning level, with warning treated as errors. Errors should be corrected as necessary once these settings are in place. (e.g. if using Microsoft Visual Studio, /W4 /WX are appropriate compiler switches to use).
    - ❏ Compiler and assembler options associated with firmware builds should enable sufficiently strong source code hash emission in debug files. (e.g. if using Microsoft Visual Studio compiler and assembler, use /ZH:SHA_256).
- ❏ Archive build artifacts for official releases.
    - ❏ All firmware build artifacts corresponding to firmware builds that are made available publicly, must be retained. This should include debug symbol files and map files to enable investigation of issues found in the field, change lists, build log files, etc.

Debug Hooks

Debugs hooks are necessary and the use cases for some debug mechanisms can become very complex with regard to maintaining system trust. There are no best practices that will cover all use cases.

Some design choices that may be applicable include:

- ❏ Disable debug hooks in production builds.
- ❏ Ensure debug hooks cannot be turned back on in ways that violate device specific security. (e.g. don't create an exploitable high permission "debug mode" susceptible to potential attacks.)
- ❏ Require a device identifier and nonce be signed by a secure, logged, authorization service to enable a debug feature. The nonce becomes invalid if the device is reset.
- ❏ Require that a device's chain of trust be invalidated as part of enabling debug features. That may imply that a device cannot participate in normal production activity from that point on which may also include purging of sensitive state and keys.

Source Code Access

Granting firmware users access to the source code helps raise the security level of that code, and increases trust in the firmware security posture.

- ❏ Whenever possible, make source code available for review by 3rd parties. This helps improve the security of the code.
- ❏ Making code available could be done through open-sourcing or through targeted source code disclosure to key customers. For cloud providers specifically, not sharing source code would often mean the firmware would not be trusted by the cloud providers.
- ❏ In the case where there is a need for code confidentiality (IP, Intellectual Property), consider sharing key parts of the firmware, such as code related to secure boot, firmware updates, and crypto implementations.
- ❏ If code cannot be made available, then consider audits by a trusted third party that can provide assurance on the trust level of the firmware.

- ❏ To ensure that source code is available when needed to develop security patches for the entire lifetime of the product, consider placing the source code into escrow to hedge against business continuity risks.

Verification/Compliance

In order to be able to prove that the firmware development followed the right security best practices, it is often required to produce evidence of the following artifacts for the firmware builds that were publicly released:

- ❏ An independent (not firmware development team) security review / pen-test.
- ❏ Reports of testing & reviews.
- ❏ Build logs.
- ❏ Test logs.
- ❏ Revision history / change log.

Secure Configuration

Many firmware implementations support runtime configuration. Configuration settings may change the security parameters of the firmware environment, disable hardware and software security features, enable debug functionality and legacy features that are known to be less secure, etc. As such, the configuration is as important to firmware security as the firmware itself.

Best practices for secure configuration include:

- ❏ Where hardware-based configuration is used, prefer components that support fuse/One-Time-Programmable (OTP) configuration, as external configuration pins with strap resistors are more easily attacked/modified.
    - ❏ When using fuses to store configuration data, ensure the fuses/OTP capabilities allow for changes to settings which may need to be updated over the life cycle of the device (eg: device ownership, failure analysis/debug state)
- ❏ For flash-based configuration, configuration should be signed and verified before use. Consider also including device-identity to prevent the configuration from being copied and reused in other devices (i.e.: non-signed flags should be avoided)
- ❏ Default configuration states should be secure, wherever possible (i.e.: excluding manufacturing initial provisioning)
- ❏ Avoid fail-open semantics such as where specific values are used to indicate more secure states[17], or where blank settings indicate “not secure”[18]

---

[17] https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-Breaking_CRP_on_NXP_LPC_Microcontrollers_slides.pdf

[18] http://faq.riffbox.org/content/3/71/en/how-to-erase-frp-factory-reset-protection-using-riff-box-generic-instrucions.html

- ❏ Implement multiple configuration checks (e.g. at each time of use, and periodically) to ensure configuration values do not change at runtime for example, by fault injection attacks

Enable Deprecation of Legacy Standards

Many legacy standards were not designed with security in mind. These standards often allow for unauthenticated firmware loading. Updates often contain a history of known vulnerabilities (CVEs) and generally pose a risk to the platform.

However, these standards are often not ready for full deprecation, and are still used in production. There needs to be a transition where over time, the industry moves to a more secure foundation.

From a cloud provider's perspective, it's important to be able to completely disable, and ideally not even include code for legacy standards. If support must be included, these systems, when disabled, should reject requests made to these subsystems, and terminate the processes handling these inputs. The following lists such known to be risky ones.

Legacy BIOS Boot

Legacy BIOS does not support secure boot, which means firmware is not authenticated during load.

The following requirements addresses this threat:

- ❏ Support a new boot mechanism that has secure boot support (such as UEFI SecureBoot, Linux Boot, Verified Uboot)
- ❏ Sign all firmware so that it can be verified for authenticity.
- ❏ Enable the owner/operator to disable support for legacy boot.

IPMI

The group responsible for the IPMI standard has asked[19] that the industry find suitable alternatives and deprecate IPMI.

> *"No further updates to the IPMI specification are planned or should be expected. The IPMI promoters encourage equipment vendors and IT managers to consider a more modern systems management interface which can provide better security, scalability and features for existing datacenters and be supported on the requisite platforms and devices. DMTF's Redfish standard (from dmtf.org/redfish) is an example of one such interface."*

IPMI has a known history of exploitable vulnerabilities (See Intel IPMI CVE Reports[20], http://fish2.com/ipmi/), as well as relies on authentication based on a shared secret, which, if leaked, leaves the systems exposed.

The following mitigations are recommended:

[19] https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html
[20] https://www.cvedetails.com/product/30635/Intel-Ipmi.html?vendor_id=238

- ❏ Enable customers to remove/disable IPMI.
- ❏ For systems that must support IPMI, enable an additional layer of authentication on top (e.g. SSH)
- ❏ Always use the most updated IPMI implementation with the latest security fixes

Insecure Network Protocols

In many cases, firmware has to support capabilities such as boot or updates via network. This exposes it to risks such as unverified image download and execution, unverified option ROMs loaded to access the network, or denial of service from malicious servers on the same layer two network (DNS, DHCP, TFTP).

The following mitigations are recommended:

- ❏ When network boot or network-based updates are required, implement the latest secure protocols (e.g. use latest TLS versions, use HTTPS instead of TFTP in PXE boot)
- ❏ Consider supporting an alternative local boot path (e.g. BMCs, if trustworthy, offer methods for installing to a host's local boot block storage device, instead of relying on network boot.)
- ❏ Support disabling any such network-based functionality in case it’s not needed by the owner/operator.

Support Tooling

Tools, utilities and drivers that are developed to support the associated component, firmware or platform should also conform to security best practices. They should be developed maintaining security and similar quality consistency across each.

Software Development Best Practices

- ❏ Firmware update and diagnostic utilities should follow Secure Development Lifecycle (SDL) best practices because they often end up being used on production environments.
  - ❏ For Windows, look at the Microsoft Driver Security Checklist[21]
- ❏ Should adhere to the principle of least-privilege[22] wherever possible (e.g. drop root after mapping IO resources)
- ❏ Should only access memory or IO resources associated with a target device, and not arbitrary memory or IO resources (excessive access results in local privilege escalation vulnerabilities[23]).
- ❏ Utilities intended for use in production environments
  - ❏ Should check that responses from devices are well-formed before processing. Devices exhibiting erroneous behavior should not be able to cause utility programs to crash or fail by providing unexpected or out-of-spec responses to commands.
  - ❏ Should be able to run offline or in isolated environments.
  - ❏ Utilities delivered in binary form must not require disabling of kernel protections against running unsigned code. This implies device drivers must be signed.
    - ❏ Windows utilities should be WHQL certified.
    - ❏ EFI utilities should be signed by the UEFI Certificate Authority.
    - ❏ For Linux, follow kernel module signing guidelines[24]
      - ❏ Follow package signing procedures outlined per distro package manager guidelines for utility packages.
      - ❏ If 3rd party distributed, provide alternative means of cryptographic package validation such as GNU Privacy Guard detached signatures.
- ❏ Debug, diagnostic, manufacturing, and testing utilities
  - ❏ Should not contain any secrets that enable additional debug capabilities.
  - ❏ Should require authorization for privileged debug capabilities that:
    - ❏ Must be tied to a specific instance of a device.
    - ❏ Should be revocable.
    - ❏ Should produce an audit trail.

[21] https://docs.microsoft.com/en-us/windows-hardware/drivers/driversecurity/driver-security-checklist
[22] https://en.wikipedia.org/wiki/Principle_of_least_privilege
[23] https://eclypsium.com/2019/08/10/screwed-drivers-signed-sealed-delivered/
[24] https://www.kernel.org/doc/html/v5.0/admin-guide/module-signing.html

Documentation

- ❏ Utilities implementing industry specified firmware update commands (eg: NVMe, ATA or SCSI) should document any deviations or additional actions necessary to apply the firmware update.
- ❏ It is recommended to document all memory and IO resources accessed by device drivers.
- ❏ All utility sub-commands and options should be properly documented.
- ❏ Non-standard interface protocols and extensions should be documented sufficiently to permit security testing to be conducted
- ❏ Recommended to provide utility source code including:
    - ❏ Debug symbols
    - ❏ Code Headers
    - ❏ Partial source code better than nothing

Environment-Specific Requirements

- ❏ UEFI
    - ❏ Tools must function normally if installed in a location other than FS0:\ (First filesystem listed in UEFI mappings table).
- ❏ DOS
    - ❏ Legacy mode boot will not be supported by future platforms, making DOS based tools unusable. Firmware update and diagnosis tools must not depend on a DOS environment.
- ❏ Linux
    - ❏ Non-EOL kernel version following your products expected support timeline.
    - ❏ Driver module support, either init of thereafter. Space restrictions or security could be a factor in only supporting kernel compiled drivers too.
    - ❏ Add appropriate access control policies to kernel driver interfaces where supported (e.g.: no world-accessible files, SELinux policies, etc)
    - ❏ Kernel drivers must make use of all appropriate uaccess[25] methods to move data in and out of the kernel, making local copies wherever required to avoid race conditions.
    - ❏ Filesystem interfaces to kernel drivers must expose a minimal attack surface. Audit all exposed interfaces (/proc, /sys, /debugfs, /dev, etc)
    - ❏ When implementing kernel drivers, ioctl[26] interfaces are error-prone and should be avoided (*file_operations* read/write are preferred).
    - ❏ Interfaces exposed in debugfs[27] should be removed for production builds.
    - ❏ If driver is not in upstream Linux, include source, kernel headers and safe examples for usage.

[25] https://github.com/torvalds/linux/blob/master/include/linux/uaccess.h
[26] https://lwn.net/Articles/428140/
[27] https://lwn.net/Articles/429321/

Post-Release Processes

Product release is just a milestone in the lifecycle of a product, moving from being an internal delivery, to having actual customers use your product. From a security perspective, you have the responsibility to continue looking for vulnerabilities, and for managing a process that allows customers to stay secure even when new vulnerabilities are discovered. ISO/SEC standards 29147 and 30111 provide a detailed and exhaustive set of requirements for how to manage these processes. The following checklist should help you get started in the right direction.

Proactively Looking for Vulnerabilities and Exploits

- ❏ Regularly look for issues in any new release/patch, as per the practices described in the test section
- ❏ Establish a secure process for customers, security researchers, or other such external sources to report vulnerabilities they discover; the process must provide a means for reporting issues without mandating participation in a bug bounty program or which otherwise places any preconditions on the terms of the disclosure. Ensure that this process is easily discoverable (eg. security.txt[28]).
- ❏ Consider offering a bug bounty program[29] to provide additional encouragement for security researchers and the user community to help find and fix security issues. This includes publishing enough required pre-requisites and information required to successfully find, assess, and report risks.

Issue Classification & Risk Assessment

- ❏ When issues are reported, implement investigation and severity classification with discovery for required fixes.
- ❏ Have the ability to issue targeted patches for specific release versions.
- ❏ It is best to use one of the publically available frameworks to classify the level of threat for issues found, based on factors such as the attack vector, ease of exploitation, possible implications, etc.
    - ❏ E.g. CVSS[30] can help calculate risk level per threat.
    - ❏ E.g. STRIDE[31] and DREAD[32] models can help with assessing threats and risk levels.
- ❏ Assume urgency until proven otherwise.

Disclosure of Vulnerabilities and Availability of Patches

- ❏ Follow the responsible disclosure[33] practices and timelines the disclosee will consent to, and provide well-defined SLA's for patch releases.

---

[28] https://securitytxt.org/
[29] https://en.wikipedia.org/wiki/Bug_bounty_program
[30] https://www.first.org/cvss/
[31] https://en.wikipedia.org/wiki/STRIDE_(security)
[32] https://en.wikipedia.org/wiki/DREAD_(risk_assessment_model)
[33] https://en.wikipedia.org/wiki/Responsible_disclosure

- ❏ When issues are identified as exploitable – customers must be notified in a timely manner, and given enough time to prepare/patch themselves.
- ❏ As soon as you're made aware of an active incident (i.e. an issue that is known to be exploited 'in the wild') – customers using your product must be notified immediately.
- ❏ Prioritizing early disclosure to cloud providers is essential as they are often targeted due to high-value.
- ❏ When security issues are made public (by you or by someone else), ensure to report/open CVE's (https://cve.mitre.org/). This helps the customers you couldn't notify keep track of incidents applicable to them.
- ❏ Ensure disclosures are tied back to the specific published versions they are applicable to, and that fixes are issued specifically to address the vulnerability. Whenever possible, avoid bundling security fixes with other features and enhancements.
- ❏ Make the firmware updates discoverable through an industry standard mechanism and packaging, and consider integrating into distribution mechanisms such as LVFS or Windows Update.

Timeline & SLA's

- ❏ Establish a committed timeline for disclosure upon vulnerability discovery and upon active incident. This timeline should give customers enough time to prepare/patch. Exact timelines can vary, but think hours and days, not weeks and months (e.g. https://cyber.dhs.gov/bod/19-02/ and https://www.cyberessentials.ncsc.gov.uk/requirements-for-it-infrastructure)

Component Specific Requirements

Most every hardware platform will contain many hardware devices or components that it needs to interface with to provide the additional hardware functionality. This interface is enabled through the components firmware which can either provide direct access to the hardware functionality or means to load additional interfaces to access the additional functionality. This section outlines details for these interfaces and components.

UEFI

The Unified Extensible Firmware Interface is a boot interface accessed via BIOS or boot ROM that provides a standard for accessing lower level devices and their associated interfaces. UEFI provides a great deal of underlying hardware access and if properly developed against, can be secured to ensure firmware integrity. Guidelines for this include:

- ❏ Systems must pass configuration security checks such as:
  - ❏ CHIPSEC[34] on Intel x86 platforms
  - ❏ Relevant FWTS tests[35]
- ❏ And include support for:
  - ❏ Secure Boot (or equivalent)
    - ❏ Hardware SRTM or DRTM
    - ❏ DBx (Key Blacklist for known vulns)
  - ❏ BIOSGuard (or equivalent)
  - ❏ Non-bypassable authenticated update mechanism (NIST 800-147)[36]
  - ❏ Option-ROM disabling capability.
  - ❏ PKI creation and management.
  - ❏ Disabling Secure Boot with visual notification.
  - ❏ Credential auth with privileged roles and secure access.

coreboot

coreboot[37] is an extended firmware platform that delivers a fast and secure boot experience on multiple hardware architectures(x86, RISC-V, ARM, PPC, etc). coreboot implements most security features supported by the chipset but it doesn't enable them by default. Verified boot and measured boot need to be implemented by the payload, e.g: heads.
Platforms that support coreboot should follow these guidelines:

- ❏ Systems must pass configuration security checks such as:

[34] https://github.com/chipsec/chipsec
[35] https://wiki.ubuntu.com/FirmwareTestSuite/Reference
[36] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-147.pdf
[37] https://doc.coreboot.org/

        - ❏ For Intel x86 platforms, CHIPSEC recommended security settings and features[38]
        - ❏ Relevant FWTS tests
- ❏ Measured boot support:
    - ❏ Heads[39] as coreboot payload implementation for systems that support TPMv1.2
    - ❏ Vboot based implementation[40] for systems that support TPMv2
- ❏ DRTM (Intel TXT on x86 platform)
    - ❏ Memory clearing[41]
    - ❏ Measurement[42] in BIOS/SINIT ACMs

Option-ROMs

Option ROMs contain firmware-level drivers needed to support pre-OS functions. They are provided by peripheral cards or incorporated into the mainboard's boot flash. These include network drivers needed to PXE boot, video display drivers, and storage drivers.

Use of legacy BIOS option ROMs is not advisable.

- ❏ Signature trusted by UEFI DB. (see UEFI Specification v2.8[43], section 32.4.1)
- ❏ Can be disabled, locked via credentials or set read-only as a configurable security measure.
    - ❏ e.g. See Microsoft's "UEFI Validation Option ROM Guidance[44]". Keep in mind that only the strictest "Secure Boot" configurations may be useful in mitigating supply chain attacks.

If rebuilding a UEFI firmware image is not desirable, one may be able to inject additional known good option ROMs using tools such as "Fiano"[45]. If embedding an option ROM to improve security, one must also re-sign the UEFI image for secure boot.

BMC

Baseboard Management Controllers (BMCs) provide out-of-band management services for servers.

BMCs have a poor history with respect to security given that their original threat model relied heavily on isolated management networks and well behaved system administrators. The model was that a BMC offered access equivalent to a system administrator's physical presence to the extent that was possible.

---

[38] https://github.com/hardenedlinux/Debian-GNU-Linux-Profiles/blob/master/docs/harbian_fw/harbian_chipsec.md
[39] https://github.com/osresearch/heads
[40] https://github.com/coreboot/coreboot/blob/master/Documentation/security/vboot/measured_boot.md
[41] https://github.com/coreboot/coreboot/blob/master/Documentation/security/memory_clearing.md
[42] https://github.com/coreboot/coreboot/blob/master/Documentation/security/intel/txt.md
[43] https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf
[44] https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/uefi-validation-option-rom-validation-guidance
[45] https://github.com/linuxboot/fiano

As services like bare-metal hosting have come into being, it is necessary to support a threat model where the host OS is potentially hostile and the BMC offers the only scalable means for the service provider to restore the host to a trustworthy state. For example, management through a BMC must be able to enforce the boot environment of the host and subsequent restoration of a good known state before handing the host off to the next customer. Models where the host OS is implicitly trusted by a BMC are not compatible with a bare-metal hosting service.

Almost all BMCs run some variant of Linux with a fixed software stack, i.e. the BMC only runs the code contained in its firmware image. The outline below provides general guidance on key points of interest.

- ❏ The BMC should be running a well supported OS release (e.g. a Long Term Support version[46])
- ❏ Use hardened Kernel settings (e.g.: KSPP guidance[47])
- ❏ Document all available interfaces:
  - ❏ IPMI[48] interface (note the recommendation in this section, i.e. do not use)
  - ❏ SMASH CLP[49] over ssh
  - ❏ Redfish[50] (Security details[51])
  - ❏ Web interface over SSL (443) with capability of disabling or rerouting from 80
- ❏ Provide a capability of disabling unwanted interfaces:
  - ❏ Telnet
  - ❏ SSH
  - ❏ Insecure Web (80)
  - ❏ Media Management Services used to attach and detach host media (platform specific)
  - ❏ IPMI
- ❏ Provide a capability of disabling or restricting features on services:
  - ❏ Prefer role-based access restrictions.
  - ❏ Prefer remote authentication via Active Directory, Open Directory, LDAP, EDir, Radius etc.

Peripheral Firmware

Firmware that is not directly executed by a host processor (e.g. firmware embedded in the peripheral hardware and responsible for its initialization), and firmware that is supplied by a peripheral to the host for execution (e.g. Option ROMs mentioned in previous section) must also abide by best practices to ensure a secure system. This firmware and and associated host-run commands are typically needed for host OS driver development. General guidance is as follows:

- ❏ Peripheral firmware should follow SecureBoot signature validation flows, just like platform firmware

[46] https://www.kernel.org/
[47] https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project/Recommended_Settings
[48] https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-technical-resources.html
[49] https://www.dmtf.org/standards/smash
[50] https://www.dmtf.org/standards/redfish
[51] http://redfish.dmtf.org/schemas/DSP0266_1.7.0.html#security-details-a-id-security-details-a-

- ❏ Implement peripheral firmware attestation, as defined by OCP-Security, to provide the host platform with visibility into peripheral firmware integrity state.
- ❏ Secure boot policies applied to peripheral firmware may include:
    - ❏ Do not transfer control to code that has not passed a strong signature check, thereby "bricking the system/sub-system", *or*
    - ❏ Drop privileges, or access to keys, and log exceptions before transferring control to code that failed signature checks. This allows unattended recovery procedures to attempt system recovery, then reboot to a trusted state. Or,
    - ❏ Design hardware that has host or BMC unilateral access to audit and program a peripheral. This punts the problem to earlier stages in the chain of trust but may complicate system operation if the peripheral is needed as a boot device.
- ❏ Consider the possibility of a malicious PCIe device in your threat model.
    - ❏ Does an IOMMU needs to be configured? When and how?
- ❏ All available commands must be listed, including unsupported commands. Supported commands must be fully documented.
- ❏ All available interfaces used to access peripheral firmware functionality need to be documented.
- ❏ Provide the capability of disabling individual interfaces if not needed for core functionality.

References

- CII Badges , https://github.com/coreinfrastructure/best-practices-badge
- NIST 800-193 , https://csrc.nist.gov/publications/detail/sp/800-193/draft
- CSIS position paper response to 800-193, https://www.cloudsecurityindustrysummit.org/document/firmware-integrity-in-the-cloud-data-center.pdf
- SAFECode , https://safecode.org/publications/
- Code Review Guidelines for Boot Firmware, https://edk2-docs.gitbooks.io/edk-ii-secure-code-review-guide/code_review_guidelines_for_boot_firmware/
- ISO 27000 , http://www.iso27001security.com/html/iso27000.html
- OWASP Secure Software Development Lifecycle Project , https://www.owasp.org/index.php/OWASP_Secure_Software_Development_Lifecycle_Project
- IOMMU protection against I/O attacks: a vulnerability and a proof of concept , https://link.springer.com/article/10.1186/s13173-017-0066-7
- ISO/SEC 29147 , https://www.iso.org/obp/ui/#iso:std:iso-iec:29147:ed-2:v1:en
- ISO/SEC 30111 , https://www.iso.org/standard/53231.html
- Best Practices for Firmware Code Signing https://www.opencompute.org/documents/ibm-white-paper-best-practices-for-firmware-code-signing

Revision History

Version	Date	Description
1	2019-06-18	Initial version out of CSIS SCWG
1.1	2019-10-27	Added community feedback, including the following: - requirements related to pre-signature validation checks - requirements for memory safety - requirements for code concurrency in multi-proc systems - requirements for secure configuration - component-specific requirements for coreboot support - improved requirements for post-release processes - additional useful references for further reading - many smaller edits and enhancements across the document - recognized new community contributors. Document is now published on OCP GitHub. Additional contributions are most welcome there.