TRUSTED® COMPUTING GROUP

REFERENCE

TCG Guidance for Secure Update of Software and Firmware on Embedded Systems

Version 1.0
Revision 64
July 18, 2019

Contact: admin@trustedcomputinggroup.org

PUBLIC REVIEW

Work in Progress

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

DISCLAIMERS, NOTICES, AND LICENSE TERMS

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, DOCUMENT OR SAMPLE.

Without limitation, TCG disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this document and to the implementation of this document, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this document or any information herein.

This document is copyrighted by Trusted Computing Group (TCG), and no license, express or implied, is granted herein other than as follows: You may not copy or reproduce the document or distribute it to others without written permission from TCG, except that you may freely do so for the purposes of (a) examining or implementing TCG documents or (b) developing, testing, or promoting information technology standards and best practices, so long as you distribute the document with these disclaimers, notices, and license terms.

Contact the Trusted Computing Group at www.trustedcomputinggroup.org for information on document licensing through membership agreements.

Any marks and brands contained herein are the property of their respective owners.

DRAFT

CHANGE HISTORY

REVISION	DATE	DESCRIPTION
1.0 Revision 61	July 17, 2019	• Moved to new document template • Changed DICE intro text

DRAFT

ACKNOWLEDGEMENTS

The TCG wishes to thank all those who contributed to this specification. This document builds on considerable work done in the various work groups in the TCG.

Special thanks to the members of the IoT-SG who participated in the development of this document:

NAME	AFFILIATION
Wael Ibrahim	American Express
Monty Wiseman	General Electric Company
Darren Krahn	Google Inc.
Graeme Proudler	Graeme Proudler
Steve Hanna	Infineon
Matthew Areno	Intel
Sung Lee	Intel
Ira McDonald	Ira McDonald
David Challener	JHU, Applied Physics Lab
Paul England	Microsoft
Jérôme Quévremont	Thales
Quentin Thareau	Thales
Raj Pal	U.S. Government

CONTENTS

DRAFT

DRAFT

1 PREFACE

This document, the ***TCG Guidance for Secure Update of Software and Firmware on Embedded Systems***, is part of the Trusted Computing Group's collection of Reference Documents, which are informative (non-normative) documents that provide advice and guidance. None of the text in this document is normative.

1.1 Purpose

The purpose of this document is to describe how secure software and firmware update for embedded systems can be done using Trusted Computing technologies.

This document provides assessments of the benefits of employing TPM[1], DICE[2], and other Trusted Computing technologies in securing software and firmware update for embedded systems. Those who wish to jump to the final conclusion of the report should read section 6.

1.2 Scope

The scope of this document is restricted to secure software and firmware update for embedded systems. The supporting infrastructure, such as code signing and update distribution services, are so critical to the secure and reliable update process that they cannot be excluded from consideration. Therefore, they are in scope for the purposes of this document.

Repair of damaged or broken devices is explicitly out of scope for this document. Rather, this document focuses on the ability to securely update software and firmware as part of the normal operation of embedded systems.

Recovery via firmware and software update is of increasing importance. Today's attackers increasingly aim to replace firmware and software on devices with their own malicious code to establish a permanent foothold on the device. Therefore, embedded systems designers cannot assume that their firmware and software will remain pristine. They must plan for ways to detect and recover from firmware and software compromise. Best practices for doing so are presented in this document.

Because each embedded system is different, this document cannot provide specific instructions suitable for all circumstances. The reader will need to evaluate their own situation and determine how to interpret the best practices described in this document.

Even if the reader follows all the recommendations in this document, nothing is completely secure. The state of the art in information security is advancing rapidly, and this is even more true for embedded systems security. Still, following the best practices in this document will provide a strong foundation for secure software and firmware update throughout the lifetime of the products to be shipped.

1.3 Relationship with Other Standards

Several other standards groups and consortia have issued documents on secure software and firmware update. This document complements the others by providing a Trusted Computing Group perspective on the topic. This document references existing standards in the area and may be referenced by other standards groups and consortia in the future. For a list of other standards and publications on this topic, see sections 9 and 10.

DRAFT

[1] https://trustedcomputinggroup.org/resource/tpm-library-specification
[2] https://trustedcomputinggroup.org/work-groups/dice-architectures

2 THREAT LANDSCAPE

Network-enabled embedded systems (the *Internet of Things*) are found in an ever-widening number of *smart* applications and platforms, including automobiles, household appliances, industrial systems, and medical equipment. This trend is being driven primarily by factors like functionality, convenience, and profit, for both the manufacturer and the user. Placing network connectivity into such devices allows for advanced feature sets, increased awareness and response, and faster patching and updating of system firmware and software. However, this network connectivity also results in new threats and potential issues that never previously existed in such platforms.

2.1 Examples of Attacks

Consider as an example the well-known Stuxnet[3] virus from 2010 that compromised Programmable Logic Controllers (PLCs) used in the Iranian nuclear program. A similar attack was successful against the Ukrainian[4] power grid in 2015 that resulted in a temporary loss of power for 225 thousand individuals. The power grid attack employed a variety of different techniques, including permanently disabling the serial-to-Ethernet converters connected to the circuit breakers by reflashing their firmware, whereas Stuxnet used multiple zero-day exploits to compromise computers running Microsoft Windows™ and then springboard onto PLCs running Siemens Step 7™ software. Both of these attacks (and the subsequent stream of attacks such as Xiongmai and Mirai) illustrate in a small way the potential impact of attacks against embedded systems in critical infrastructure. Although the direct impact of the Xiongmai and Mirai attacks was fairly minimal in scope, the number of susceptible devices was considered substantial and vendors worked furiously to provide patches to prohibit the spread of such attacks.

The flow of software updates has continued and in fact grown in recent years, showing no sign of abating. This is not surprising in light of the constant rhythm of security patches for stable commercial operating systems and applications. The criticality of frequent and timely deployment of updates was illustrated by the recent ransomware attack known as “WannaCry[5]”. This ransomware attack locked out file and data access on unpatched systems until the owner made a ransom payment to the attacker. Although a patch existed for many systems, the attack was possible for two primary reasons: first, systems were not updated in a timely enough manner, and second, older Operating System versions that no longer had available support were also susceptible to the attack. Although this example was not specifically targeting embedded systems, a similar attack is certainly possible, and the ransomware for unlocking critical infrastructure is likely to be substantially higher than for a common desktop system.

These examples illustrate just a few of the potential consequences and impacts of the vast array of threats targeting embedded systems. Two key questions at this point are: Why are embedded systems such a prime target? What avenues of attack are available to malicious parties who want to launch such attacks?

2.2 Attacker Motivations and Capabilities

The use-cases for embedded systems make them enticing targets for malicious attackers for a number of reasons. The market for embedded systems is huge, which in turn results in a large number of manufacturers all trying to offer the best capabilities at the cheapest price. Equally appealing is the quantity of deployments for embedded systems. The quantity provides an attacker with a much higher return on investment in terms of susceptible targets if they are able to find an associated vulnerability. Although most of the traditional motivations for hackers still exist, i.e. notoriety, personal vendettas, “because I could”, the two reasons previously listed are arguably of higher concern in embedded systems environments than in any other. And even in the case of the traditional motivations,

DRAFT

[3] http://www.bbc.com/news/technology-11388018
[4] https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf
[5] http://www.bbc.com/news/technology-39901382

hackers are likely to find much greater recognition and impact through successful attacks on critical infrastructure than any other targets.

Embedded systems are prime targets partly because of the need for manufacturers to drive down costs, thereby necessitating embedded systems developers also drive down their costs. Unfortunately, one of the first cost areas cut is often security. For instance, manufacturers are now beginning to make *smart* light-bulbs that allow user to easily adjust lighting, create schedules for usage, and even control their lights remotely. Because manufacturers realize most users are not going to pay significantly more money for a convenience feature, they will often use the cheapest security solution they can find, if they even include one at all. Yet these bulbs can mount DDoS[6] and many other attacks[7] because they are network connected, often behind the firewall in a trusted zone.

Open source software helps address this by providing capabilities at little to no cost. However, the security of open source software relies upon submission of bug reports and subsequent patches. When a bug is considered substantial enough to allow for the possibility of an attacker to modify, or even control, execution of the system, a Common Vulnerabilities and Exposure (CVE) report is generated. These CVEs are publicly available and specifically list affected software versions. It usually requires significant effort and expertise to find both a vulnerability and a way to exploit that vulnerability. Relatively few actors (such as sophisticated criminals and nation states) have such capabilities. Hence few actors mount so-called zero-day attacks by exploiting previously unpublished vulnerabilities. In contrast, the longer an exploit is public knowledge, the more the number of actors able to mount an attack using that exploit. If systems containing known vulnerabilities are not promptly patched, it becomes almost trivial for attackers to create malicious payloads and inject them onto target systems. This should not be considered a criticism of open source software, but rather a clear indication of the critical importance of frequent update deployments for any embedded system that relies upon such software. The security maintenance duration should be long enough to cover the expected lifetime of the device.

Equally concerning is the placement and quantity of deployment for these devices. Attackers are constantly attempting to infiltrate corporate and government networks for a variety of reasons (sabotage, espionage, data or IP theft, etc.). Doing so typically requires breaking one or more firewalls or other network security solutions, the security of which often relies upon the fact that the attacker is launching attacks from outside the network. If an attacker is able to gain access to the network and then implant malicious code on a vulnerable embedded system that is widely used in the target network, they can create their own distributed botnet or command and control centers inside the target network. This would allow for continuous attacks against potentially sensitive systems from inside the network as opposed to outside of it. Additionally, it allows for creation of Denial of Service (DoS) "hives" from a single organization, company, or building, rather than depending upon a geographically dispersed Distributed DoS (DDoS).

The same could be applicable for an individual's home. Rather than turning a single computer into another node in their botnet network, an attacker could use that computer to launch attacks against smart appliances within the individual's home, thereby creating an entire new botnet network from what used to be a single node. And while security solutions exist for traditional desktop system which may make it easier for a user to discover that their computer is compromised, making a similar observation on their smart fridge is unlikely to be as easy. Embedded systems by their nature often have limited, or no, visual interface, and as such they make prime targets for attackers who wish to conceal their presence on networks.

These devices may also be used to capture a home owner's financial or social media information, track network traffic, gain control of connected locks, or even spy on the resident(s). This has recently been shown to be a serious problem with the advent of the website Shodan[8]. This website touts itself as, "The search engine for the Internet of Things." Potential attackers can look up an array of different devices connected to the internet, pull up IP addresses

[6] https://www.theinquirer.net/inquirer/news/3004579/university-suffers-ddos-attack-after-its-schooled-by-own-iot-devices
[7] http://fortune.com/2016/11/03/light-bulb-hacking
[8] https://www.shodan.io

DRAFT

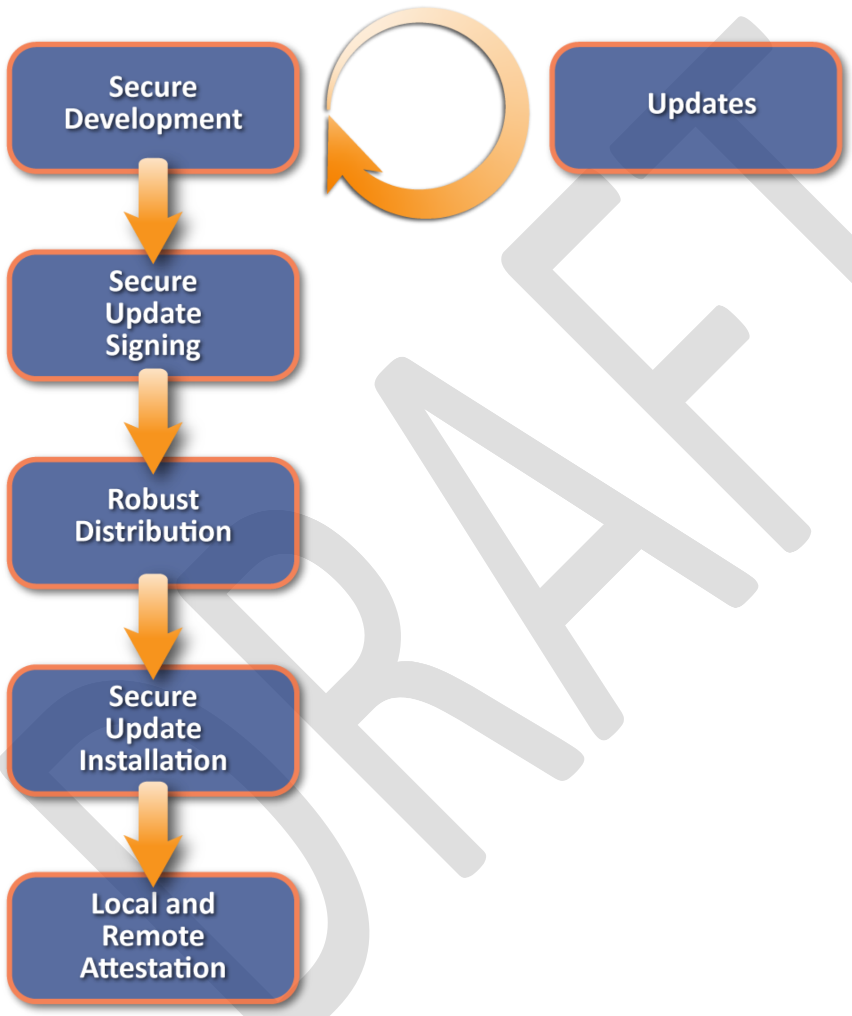
for such devices, and even find CVE reports for specific devices, thereby providing them with nearly everything they need to remotely control IoT embedded systems. Instances of cyberbullying and cyberstalking have already been reported where attackers used information gleaned from this website to find their target victim.

Detecting and mitigating such concerns are critical to the security and reliability of homes, businesses, organization, and even entire nations. The first step in this process is to ensure such systems have the means to securely and periodically perform updates that reduce vulnerabilities and attempt to validate the current execution state of the system. This document provides insight and guidance for manufacturers on making this first step.

DRAFT

3 GUIDANCE ON SOFTWARE / FIRMWARE UPDATE

As with many cybersecurity problems, no one solution can address the many attack techniques that attackers can employ. Rather, a "defense in depth" approach is needed. Devices must be capable of being updated, even when they are compromised[9]. Multiple countermeasures must be employed to ensure that attacks are prevented. Attackers try to find and exploit the weakest link, so all the steps in the update development and deployment process must be properly protected. The leftmost column in Figure 1 illustrates the main phases in the secure software and firmware update lifecycle, which are described in the subsequent sections of this document. The need for continuous updates is indicated by the Updates box on the right of the figure.



DRAFT

Figure 1 - The Software and Firmware Update Lifecycle

3.1 Secure Development

To secure the firmware and software development process, one must start by establishing and agreeing on proven software development practices. Recommended practices include:

- Building security into all steps in the development process [10]

[9] The TCG's Cyber Resilient Technology Work Group addresses these issues: https://trustedcomputinggroup.org/work-groups/cyber-resilient-technologies

[10] https://safecode.org

- Thorough threat analysis and countermeasure selection during design [11], [12], [13]
- Applying best practices for security and improving them over time to address new and emerging threats
- Agreeing on measurable security requirements that must be met before release
- Securing the development environment
- Using trustworthy tools, languages, and libraries [14]
- Careful input validation[15] and error handling[16]
- Training all participants on security
- Establishing a robust incident response process[17]

All of the practices in the preceding list should be adopted by any embedded system developer or teams. Additional practices for higher levels of security may include:

- Establishing strict physical security (e.g., mantraps) and network security (e.g., air gaps) for the development machines
- Independent security reviews of source code
- Independent security audits and certifications of the product and the development process (e.g., Common Criteria[18], FIPS 140[19])
- Setting up of an Information Security Management System (ISO 27000)
- Independent security analysis of external dependencies (tools, libraries, etc.)
- Penetration testing
- Stress testing
- Automated testing to find bugs and security vulnerabilities
- Static analysis of binary or source code to find bugs and poor coding practices
- Strong authentication for staff

Fortunately, many solid references are available on secure software development.[20] This section provides only a brief summary of the techniques described in those references.

3.2 Secure Update Signing

Creating a secure software update is more complex than creating software from scratch. In addition to the complexity that comes from updating an existing software installation (e.g., updating files created by an earlier version), the software update must be signed (known as “code signing”) so that the recipient can verify its origin and integrity before installing it.

Code signing is one of the most important steps in the secure software update process but rarely does it receive the careful attention that it deserves. For this reason, attackers have repeatedly attacked the code signing step[21] with tremendous success. A successful attack on this step permits the attacker to distribute malicious code signed with the proper key, either by exploiting defects in the signature validation process or by using the key itself. This slips the malicious code right under the nose of the defender without detection.

DRAFT

[11] ***Threat Modeling: Designing for Security by Adam Shostack***
[12] ***Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis by Tony UcedaVelez***
[13] ***Securing Systems: Applied Security Architecture and Threat Models by Brook S. E. Schoenfield***
[14] https://www.securecoding.cert.org
[15] https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet
[16] https://www.owasp.org/index.php/Error_Handling
[17] http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-61r2.pdf
[18] https://www.iso.org/standard/72891.html
[19] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf
[20] https://www.microsoft.com/en-us/sdl
[21] https://www.zdnet.com/article/security-flaw-lets-attackers-recover-private-keys-from-qualcomm-chips

When building a secure update signing system, recommended practices include:

- Thorough threat analysis and countermeasure selection during design
- Applying best practices for security
- Using separate keys and certificates for signing production code vs. development code
- Using reliable, well-vetted cryptographic algorithms and tools
- Selecting a signing key with adequate strength[22]
- Ensuring that customer products only accept code signed with the production key
- Applying extra security measures to the production code signing process
- Carefully vetting any CAs and other parties trusted in the signing process
- Using a patch format that includes version information to prevent rollback
- Designing in a revocation process for patches (in case they turn out to be bad) and keys (in case they are compromised)
- Planning for key expiration and rollover (because no key lasts forever)
- Designing in cryptographic algorithm agility (because no algorithm lasts forever)

All of the practices in the preceding list should be adopted by any embedded system developer. Additional practices for higher levels of security may include:

- Placing the production signing key in a Hardware Security Module (HSM) to prevent extraction
- Using a dedicated and air-gapped computer for production code signing
- Strictly controlling physical and logical access to the production code signing system
- Requiring multiple parties to authorize production code signing or to gain access to the production code signing system[23]
- Carefully examining code for signs of compromise before signing it

While these measures may seem extreme, any flaws in the security of code signing can give attackers free rein over the systems being updated, so these measures are justified. A best practices document is available at: https://www.thawte.com/code-signing/whitepaper/best-practices-for-code-signing-certificates.pdf

3.3 Robust Distribution

After a software update has been signed, it must be distributed to the machines that will be updated. If the distribution network is intermittent, devices may need to assemble updates from data received at irregular intervals whenever the communication link becomes available. If networks have restricted bandwidth and/or high latency, updates may need to be just the differences between the device's current state and the upgraded state, instead of an entire upgraded state. In addition to the fundamental challenge of sending large files out to many endpoints, other essential challenges arise such as ensuring the identity and trustworthiness of the distribution service.

Many organizations are concerned that updates may cause unexpected problems so they insist on performing their own testing before distributing updates. This adds an element of delay and another trusted party to the distribution system. If patching is delayed too long, endpoints may be placed at greater risk of compromise.

Weaknesses in the distribution step can be exploited, even if all other security measures are properly implemented. For example, needed updates can be blocked or malicious commands injected into the update stream.[24]

When building a robust update distribution system, recommended practices include:

[22] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf
[23] Multi-party authorization for production code signing could be implemented with physical locks and keys, passwords, biometrics, electronic ledgers, or a variety of other mechanisms.
[24] https://www.scmagazineuk.com/windows-server-update-services-open-to-attack/article/535511

DRAFT

- Thorough threat analysis and countermeasure selection during design
- Applying best practices for security
- Securing communications with thoroughly vetted security protocols
- Establishing the identity and trustworthiness of the distribution service, generally through a trusted third party
- Automating the distribution and installation of software updates to minimize the risk that updates are not widely distributed and installed, subject to administrative control for organizations that need to control and manage software updating
- Designing software update distribution mechanisms in a manner that avoids overloading networks or servers
- Securing administrative access to update servers to ensure that only authorized updates are distributed
- Using established software update distribution mechanisms[25] to reduce administrative workload
- Permitting administrators to schedule updates on their organization's systems
- Detecting and stopping denial of service attacks on update servers or on the distribution mechanism

All of the practices in the preceding list should be adopted by any embedded system developer. Additional practices for higher levels of security may include:

- Placing the communications security keys in a Hardware Security Module (HSM) to prevent extraction
- Authenticating endpoints to determine which updates they are authorized to receive
- Tracking installation of updates to ensure that endpoints have been updated
- Alerting administrators if updates cannot be installed on some endpoints
- Permitting administrators to override user efforts to stop updates or to install updates without administrative approval
- Assuming that the update servers may be compromised at some point. For this reason, endpoints should verify any updates downloaded from the servers and other mechanisms should be included to verify that updates are being properly downloaded and installed.

3.4 Secure Update Installation

Installing a software or firmware update on an embedded system may seem to be a simple and easy matter but actually there are many tricky operational and security aspects.

Downtime is a primary concern for some embedded systems, especially those that control critical processes. Although steps can be taken to reduce downtime due to an update, some downtime is generally involved. For this reason (and to manage risks), device owners must be able to schedule updates, at least for critical devices. Alternatively, devices may need to accept upgrades whilst the devices are working. This typically requires redundancy in the device, so that part of the device can be upgraded while the rest of the device continues to operate, plus a way to determine when it is safe for the device to switch from existing functionality to upgraded functionality. Performing updates in the background may enable encrypted data to be immediately overwritten with updated encrypted data, avoiding the need to decrypt during the update. Yet another alternative is to employ one or more backup devices while the primary device is being updated.

Software and firmware updates sometimes fail to complete successfully, for various reasons (power failure during update, inability to convert data or to reach external resources, etc.). Devices should be able to recover from such failures, if only by reverting to a recovery mode. Another way to do this is to use an A/B mechanism, as described in section 8.

[25] http://www.embedded-computing.com/dev-tools-and-os/identifying-secure-firmware-update-mechanisms-for-embedded-linux-devices-and-open-source-options

DRAFT

Infection of the update with malicious software and firmware is always a concern. Every possible step must be taken to prevent such infections, as they may result in device compromise or bricking. Mutable Roots of Trust[26] must be especially well protected against compromise.

Updates often require changes to data stored on the embedded system. For example, the format of a database or configuration file may need to change when a new version of software is installed. With restricted on-device storage or a desire to accommodate graceful recovery if an update fails, any complex data conversions may best be achieved by sending the data to the cloud or an external server for backup and conversion.

- Recommended practices for secure update installation include:
- Thorough threat analysis and countermeasure selection during design
- Applying best practices for security[27]
- Restricting update installation privileges and activities to a minimally sized, carefully coded, and tightly controlled Root Update Engine (RUE, as defined in section 5.2), including protections for keys and code that are critical to the update process
- Verifying updates before installation for: source authentication and authorization to make sure they come from a trusted update signer, integrity, appropriateness for this device, and any other necessary authorizations (e.g. device owner)
- Avoiding TOCTTOU (Time Of Check To Time Of Use) attacks and similar race conditions, for example by ensuring that no other operations can take place while an update is occurring
- Including a recovery process for detecting and recovering from failed or malicious updates of software and firmware
- Permitting administrators to schedule updates on their organization's systems

All of the practices in the preceding list should be adopted by any embedded system developer. Additional practices for higher levels of security may include:

- Using encrypted updates to increase the difficulty of reverse engineering updates to discover and exploit vulnerabilities in old code. In extreme cases (e.g., for secure processors), unencrypted code may only be present in tamper-resistant hardware.
- Using physically tamper-resistant storage and/or execution for keys and code that are critical to the update process (e.g., the RUE and RTM)
- Carefully designing, testing, and protecting the RUE to ensure that it can handle maliciously crafted inputs, thus preventing OS compromise from escalating to firmware compromise
- Countersigning updates, so even genuine updates (signed by a manufacturer) cannot be applied without permission from entities (such as a platform's Administrator) with a legitimate interest in determining whether and when an update should be applied.
- Ensuring that the recovery process cannot be used as a way to roll back to vulnerable firmware (e.g., by requiring installation of fresh firmware not just reverting to the factory configuration)

3.5 Post-Update Verification and Attestation

Distribution systems need to determine whether an upgraded device has been properly upgraded, is functional, provides all its APIs, and can access networks. This verification process may include manual or automated testing of functionality and integrity.

As part of this process or on an ongoing basis, device owners and manufacturers (and other parties) may wish to monitor the version and integrity of firmware and software installed on a device. Trusted Computing enables this

DRAFT

[26] https://trustedcomputinggroup.org/resource/tcg-glossary

[27] ETSI TS 103 645 Cyber Security for Consumer Internet of Things, https://www.etsi.org/deliver/etsi_ts/103600_103699/103645/01.01.01_60/ts_103645v010101p.pdf

securely through the remote attestation capability. This practice may be necessary for cases where device monitoring or management is needed.

To enable attestation, a measured boot must be performed. During such a measured boot, native measurements (cryptographic hashes) of firmware and software are taken during the boot sequence, initially by a Root of Trust for Measurement (RTM), and sent to a Root of Trust for Storage (RTS). Alternatively, native measurements of firmware and software taken by the device are compared by the device with signed versions of measurements stored in the device: if they match, the device verifies the signatures using public keys stored in the device, and sends the public keys to the Root of Trust for Storage (instead of sending the native measurements). This option increases device complexity but simplifies attestation infrastructure. (See later in this section or [D-RTM].)

After the measured boot, remote attestation can be performed. The measurements taken during boot can be securely sent to an attestation server via a cryptographic handshake with a Root of Trust for Reporting (RTR). The attestation server can compare the measurements against a set of “golden measurements” to determine which firmware or software is running on the device. Devices with old firmware or software can be isolated to protect them from infection and upgraded to the newest versions. Devices with malicious or unknown firmware or software can be isolated for forensic inspection.

Golden native measurements of firmware and software are short-lived, because native measurements change whenever firmware or software changes. However, golden measurements are long-lived if they comprise public keys used by the device to verify signatures of native measurements: the public keys stay the same, irrespective of which version of firmware or software is installed on the device, if the same private key is always used to sign updates of the same firmware or software. Local policies can also be implemented (e.g., using TPM sealing or PCR-based policies) to enable local detection of old or malicious firmware. Appropriate responses to local detection of such problems could include triggering a recovery mode and/or alerting the administrator via a display indication.

When remote attestation is employed, recommended practices include:

- Thorough threat analysis and countermeasure selection during design
- Follow Secure Software Development Life Cycle (SSDLC)[28] practices
- Using measured boot and remote attestation to enable remote management servers to detect and manage device firmware and software versions

All of the practices in the preceding list should be adopted by any embedded system developer who employs remote attestation. Additional practices for higher levels of security may include:

- Implementing network access controls to isolate devices with old, malicious, or unknown firmware or software versions
- Using local policies as described above to detect improper firmware and software versions and trigger device recovery or at least prevent harmful behavior

3.6 Threats and Countermeasures

	Countermeasures				
Threats	**Secure Development**	**Secure Update Signing**	**Robust Distribution**	**Secure Update Installation**	**Remote Attestation**
Compromise of Supply Chain (e.g. Compiler)	X				
Compromise of Signing Keys		X			

[28] https://safecode.org

DRAFT

	Countermeasures				
Threats	**Secure Development**	**Secure Update Signing**	**Robust Distribution**	**Secure Update Installation**	**Remote Attestation**
Obsolescence of Crypto Algorithm		X			
Denial of Service on Update Distribution			X		
Compromise of Update Distribution		X		X	X
Update without Local Approval				X	X
Failed Update Leaving Unsafe Device				X	X
Exploit Vulnerability in Update Process	X			X	

Table 1 - Threats and Countermeasures

DRAFT

4 HANDLING CONSTRAINED DEVICES AND OTHER CONSTRAINTS

Embedded systems are very diverse and impose different constraints that drive a need for different solutions to be chosen for the secure update of software and firmware.

Examples of such constraints are enumerated below. Section 5 describes several solutions suitable for use in such constrained environments.

4.1 Device Constraints

Depending upon the application domain and the architectural choices, the devices can have reduced computing, isolation, Root-of-Trust, connectivity and/or energy resources. Examples of such resources include:

- Separation/Isolation Mechanism
    - Separate memory for secure processing
        - Physically separate memory
        - MMU separation
        - Cryptographic separation
    - Separate execution environments
        - Processes – separated by kernel
        - Containers – separated by kernel
        - Sandboxes – separated by application code
        - Virtual machines – separated by hypervisor
        - Trusted execution environment – separated by processor security
        - Physically separate – separate chips or cores
- Restricted Power
    - Battery power with long life needed
    - Scavenged power
    - Intermittent power
- Different Communications protocols and media
    - Restricted bandwidth
    - High latency
    - Broadcast/multicast (satellite)
    - One-way
    - Multi-hop or mesh
    - Intermittent or poor connectivity
    - Small message length
- Memory size
    - Restricted Non-Volatile Storage
    - Restricted Volatile Storage
    - Subject to wear-out

DRAFT

- Processor
  - Low speed
  - Low word length
  - Lack of cryptographic acceleration primitives (see the note at the end of this section)
- Availability of Device Security Mechanisms
  - TPM
  - DICE
  - Firmware Update Latch
  - Crypto accelerator
  - Secure storage
  - RUE (Root of Trust for Update)
  - RTM (Root of Trust for Measurement)

The design process for these devices can also involve constraints:

- OS and Language properties
  - Strong types
  - Automatic memory management
  - ASLR
  - Protection against buffer overflows

The production process for the devices is a source of additional constraints:

- Manufacturing and logistics constraints
  - Per-device customization
  - Per-device identity
  - Untrusted manufacturing line
  - Untrusted shipping and logistics
  - Untrusted component suppliers
- Cost constraints
  - Restricted budget

Note: In the specific context of Trusted Computing, the ability to perform computing-intensive asymmetric cryptography has a tremendous impact on the ability to deploy PKI and the use of X.509 certificates:

- A device that cannot support asymmetric cryptography imposes the storage of secret keys in the managing entity (usually a server) and therefore the need to increase its security.
- A device that supports asymmetric cryptography can lack the full support of X.509 certificate management. In such case, the managing entity will include some middleware to translate X.509 certificates into a simpler protocol supported by the embedded device.
- A device that comprises the resources to fully support X.509 certificates will remove the need to offload PKI-related functions.

4.2 System Constraints

The environment and the system can also set constraints on the secure update solution:

- Environmental constraints
  - Subject to tampering

DRAFT

- Administrative issues
    - Ability to schedule downtime
    - Availability of skilled administrative staff
    - Availability of physical access to device
    - Need to retest or recertify after update
- Lifecycle issues
    - Availability of manufacturer support

4.3 Application Requirements

Finally, the application domain can also involve specific constraints.

- Long product lifetime
    - Implies need for high MTBF
    - Implies increased maintenance and obsolescence management
- Real-time responsiveness
    - Need to respond to interrupts or events within a certain period of time

DRAFT

5 SOLUTIONS FOR CONSTRAINED DEVICES

This section presents several solutions to support recommendations identified in section 3 for embedded devices that have some of the constraints enumerated in session 4. For further reading, section 7 evaluates these solutions and section 8 presents a few alternative solutions.

5.1 Isolation

Secure computing depends on engines (software and/or hardware components) that perform dedicated and critical functions. These engines are protected from interference by isolation. An engine may be isolated via any combination of physical and logical mechanisms that provide the necessary level of protection.

Physical isolation operates all the time. The construction of a physically isolated engine, and the construction of the mechanisms that provide a logically isolated engine, must be sufficient to repel the physical attacks that the host platform is specified to resist. Physical isolation of an engine may also be sufficient to repel logical interference with that engine, provided the engine cannot be subverted via its interface. Depending on the degree of isolation that is required, physical isolation may entail physical constructs that hinder and detect interference, and/or physical protection provided by the platform's environment (a guarded and locked room, say).

Logical isolation operates only when the host platform is operating. Logical isolation relies upon physical isolation, because logical mechanisms are ultimately provided by a physical construct with a physical boundary. Logical isolation mechanisms must be properly implemented, and sufficient to repel the logical attacks that the host platform is specified to resist. Logical isolation mechanisms include (in no particular order) processor execution modes, micro kernels, hypervisors, virtualization, sandboxing, and trusted execution environments.

5.2 Generic Root Update Engine (RUE)

This section describes properties of an Update Engine that affect the trustworthiness of a platform.

An Update Engine is an engine in a platform that, because of the platform's architecture, can modify the behavior of one or more engines in the platform. Typically, an Update Engine modifies behavior by modifying software executing on a processing engine.

If a platform has a hierarchy of Update Engines, a subordinate Update Engine can be updated by a superior Update Engine. The root of a hierarchy of Update Engines, or the sole Update Engine in a platform that doesn't have a hierarchy of Update Engines, is a Root Update Engine. An updateable Root Update Engine must be able to update itself, as well as updating other engines.

There are three classes of Root Update Engine.

1. If the host platform is not a Trusted Platform[29], the Root Update Engine must be a Root-of-Trust (for update) because any misbehavior of the Root Update Engine cannot be detected. This class of Root Update Engine is a Root-of-Trust-for-Update.
2. If the host platform is a Trusted Platform and a Root Update Engine updates a TCG Root-of-Trust, the Root Update Engine must be either an integral part of a TCG Root-of-Trust, or a TCG Root-of-Trust in its own right, because any misbehavior of the Root Update Engine cannot be detected. This class of Root Update Engine is a Root-of-Trust-for-Update.
3. If the host platform is a Trusted Platform and a Root Update Engine updates an engine that is not a TCG Root-of-Trust, the Root Update Engine is not a Root-of-Trust, because misbehavior of that Root Update Engine can be detected via PCRs, sealing and attestation. This class of Root Update Engine is called a Trusted Updater.

[29] A Trusted Platform is "a platform that uses Roots of Trust to provide reliable reporting of the characteristics that determine its trustworthiness". See TCG's Glossary http://trustedcomputinggroup.org/resource/tcg-glossary

Figure 2 illustrates an instance of the first class of Root Update Engine, updating itself and updating software in a platform that is not a Trusted Platform.

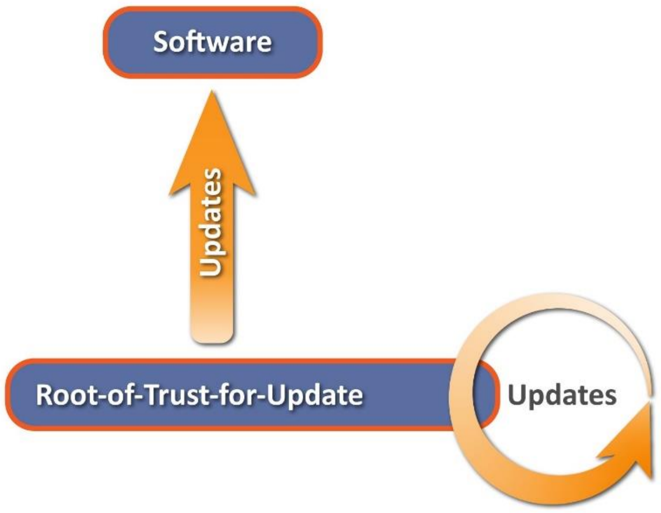


Figure 2 - Updating a platform that is not a Trusted Platform

Figure 3 illustrates a typical instance of a Trusted Platform where the second class of Root Update Engine is updating itself and updating a Root-of-Trust, and the third class of Root Update Engine is updating itself and updating software.

DRAFT

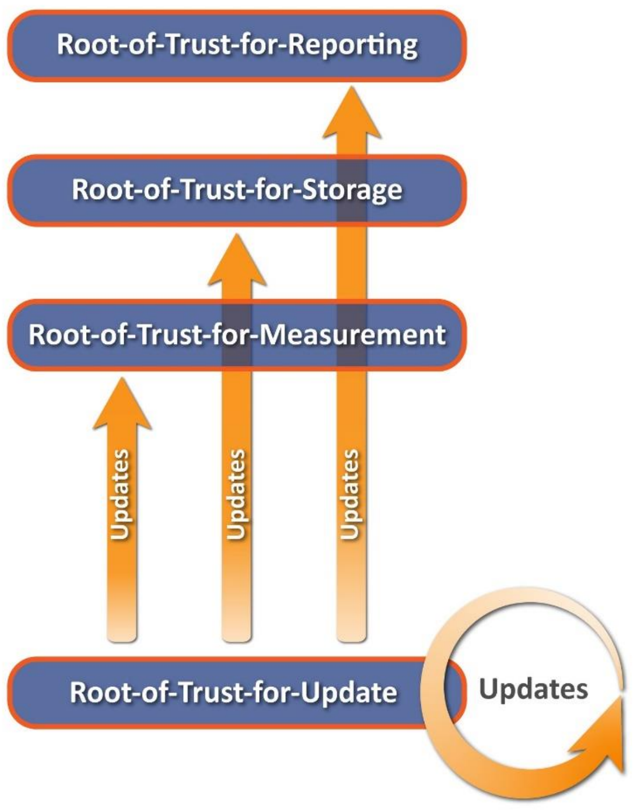


Figure 3 - Updating a Trusted Platform

It is essential that an Update Engine is sufficiently isolated from interference, so the Update Engine can always be relied upon to correctly update itself and update other engines. As such, it is often advantageous to perform updates as soon as a platform boots. This is because nothing can interfere logically with an Update Engine if updates are done before any other engine executes on the platform, provided the Update Engine finishes updating and is disabled before other engines start executing on the platform.

An update to an engine must itself be signed by the manufacturer (or proxy) of the engine that is about to be updated. This is to ensure that an update is not an attack. Each engine that is capable of being updated should be provisioned with its manufacturer's public key, or (more likely) a hash of the manufacturer's public key, to minimize storage space. The manufacturer's public key (or hash) must be protected from unauthorized modification to prevent subversion of the update process. If an engine cannot provide that protection, the Update Engine must protect the engine manufacturer's public key (or hash) on behalf of the engine.

During an update, the Update Engine is loaded with:

- a manufacturer's public key
- an identifier of the engine to be updated
- the update package
- a signature over the identifier and update package that is signed with the manufacturer's private key

The Update Engine hashes the manufacturer's public key and verifies that the hash of the manufacturer's public key[30] matches the hash stored in the engine that is about to be updated. Then the Update Engine uses the

[30] the public key supplied with the update

manufacturer's public key to verify the signature on the update package. Finally, the Update Engine applies the update package to the engine that is being updated.

Provisioning engines with their manufacturer's public key eliminates the necessity for Update Engines to parse a list of certificates back to a root certificate. This is advantageous because certificate parsing is very complex and implementations can be vulnerable to maliciously crafted input data intended to trick an Update Engine into accepting a rogue public key and hence applying a rogue update package.

The manufacturer's public key (or hash) in an updateable engine must be replaced when a manufacturer's private key is compromised. If a platform can be updated before rogues have the opportunity to exploit the compromise, it may be safe to use the compromised key to authorize installation of a replacement manufacturer's public key (or hash). It is safer, however, to install an ordered list or hierarchy of public keys (or their hashes) in an engine, instead of just a single manufacturer's public key (or hash): then a superior key can authorize replacement of compromised subordinate keys. One simple implementation of such a mechanism is a list of public keys (or their hashes) in an engine, where compromised keys are removed from the list when a subsequent key is used to sign an update package.

An update to an engine may additionally be signed by entities such as the host platform's Administrator. This enables an Administrator to control if and when a genuine update should be applied, i.e. the Update Engine must refuse to update an engine unless all signatures are present and valid. An alternative implementation may be preferred when a host platform has a Trusted Computing Base[31] (TCB) that can gate access to engines. If a host platform can implement a secure wrapper (such as a TCB) around Update Engines, the wrapper could verify additional signatures on an update before submitting the update to the Update Engine, which verifies just the manufacturer's signature on the update.

While these previous descriptions of the update process assume asymmetric cryptographic signatures, other techniques can also be used to verify approved updates. All these techniques have their normal advantages and disadvantages, which are not specific to approving an update. Table 2 illustrates some of the usual advantages and disadvantages of some techniques for approving an update.

Technique used to verify permission to apply an update	**Advantages**	**Disadvantages**
Verify asymmetric signatures on the update	Easy remote approval, because a signature doesn't need to be protected in transit. Less complex infrastructure, because just one secret per signer must be protected.	More resources are required because asymmetric cryptography engines are more complex than symmetric cryptographic engines
Verify symmetric signatures (HMAC) on the update	Easy remote approval because a signature doesn't need to be protected in transit. Fewer resources are required because symmetric cryptography engines are less complex than asymmetric cryptographic engines	More complex infrastructure because one secret per signer/verifier pair must be protected. Greater risk because all copies of the symmetric key must be protected.

[31] Lampson et. al [LAMPSON] define the TCB as "a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security."
The Orange Book [ORANGE-BOOK] defines the TCB as "the totality of protection mechanisms within it, including hardware, firmware, and software, the combination of which is responsible for enforcing a computer security policy."

DRAFT

Technique used to verify permission to apply an update	Advantages	Disadvantages
Use Physical Presence to approve an update (verify that a switch has been operated, say)	Minimal resources are required because no cryptographic engines are involved	No remote approval because reliably verifying a distant action is usually impractical More complex infrastructure because each update must be protected in transit to the platform, to ensure that the applied update is genuine and relevant

Table 2 - Assessment of Update Techniques

Update packages should include a version number, in order to prevent engines being updated with genuine-but-old update packages that have known weaknesses. The version number of updated instructions must be greater than the version number of current instructions in an engine, so Update Engines should compare the version number of the updated instructions against the version number of the current instructions in an engine. If an update is found to cause problems, the update should be reversed by creating and applying a (remedial) update package that is the same as a prior update package, with the exception that this remedial update package has a version number greater than the problematic update. It is critical that version numbers for the update be included in a portion of the update package that is signed and not located in a header or some other potentially unverified location.

Update Engines do not necessarily enforce increasing version numbers in updates to critical engines, and might be able to apply update packages with smaller version numbers to critical engines. This exception to the general rule might be necessary for critical engines that cannot remain unavailable for the time required to create a remedial update package, even though allowing smaller version numbers exposes engines to rollback attacks.

In some cases, an update should not be applied to an engine unless the host platform has obtained permission from the platform's administrator. This is because the act of applying the update might cause problems:

- unacceptable (albeit temporary) loss of service, which could cause danger if the platform performs a safety-critical function
- the functionality of the resultant updated platform might be unacceptable to the owner/administrator

Also, the additional functionality or benefits of the update to the platform might be of no interest to the owner/administrator. If an administrator decides not to apply an update, the administrator may need to accept liability for the consequences.

The method of obtaining permission for an update depends on individual circumstances. In some cases, but not in cases such as M2M (machine-to-machine) IoT, physical interaction with a platform may be practical and sufficient. In other cases, a platform with an existing ability to recognize administrator commands may require such a command before initiating an update, or may require update packages to be counter-signed by the administrator. Permission may also involve a legal agreement between the manufacturer and the platform's owner/administrator.

It may be important for the host platform to be able to prove that it contains a Root Update Engine. For example, to know whether a platform can be reliably updated and what entity is able to update a platform. If a Root Update Engine is part of an existing Root-of-Trust and updates that RoT, the Root Update Engine is implicitly included in and endorsed by the RoT's endorsement, whatever that might be. Otherwise, a Root Update Engine could be endorsed via its own Endorsement Certificate, or a Root Update Engine in a Trusted Platform could be measured and reported via a RTM and TPM. A Root Update Engine never needs to parse its own Endorsement Certificate: only third parties need to parse a Root Update Engine's Endorsement Certificate to decide whether the Root Update Engine can be trusted.

DRAFT

Update Engines provide several of the platform capabilities that are recommended in this reference document. As explained in this section, Update Engines verify updates before applying updates and prevent rollback attacks. In addition, Update Engines may also help by decrypting update packages, detecting failed updates, and initiating recovery from failed updates. Update Engines therefore can implement many of the recommended practices described in section 3.4, such as restricting update installation privileges and activities to a minimally sized, carefully coded, and tightly controlled Engine, verifying updates before installation, permitting administrators to schedule updates, using encrypted updates, and ensuring that the recovery process cannot be used as a way to roll back to vulnerable firmware. Update Engines can also address the threat "Update without Local Approval" described in section 3.6.

Update Engines in conjunction with other technological building blocks can implement recommended practices. Some such aspects are described in section 5.3, section 5.4, and section 5.5. The merits of these combinations are summarized in section 7.

5.3 Firmware Update Latch with Root Update Engine

The firmware update latch is a mechanism that prevents the firmware from being modified outside the RUE, thus minimizing the risk of unauthorized changes to firmware. Such a mechanism is described in section 3.1 of [NISTSP800-147]. Figure 4 is a diagram illustrating a typical RUE and firmware update latch:

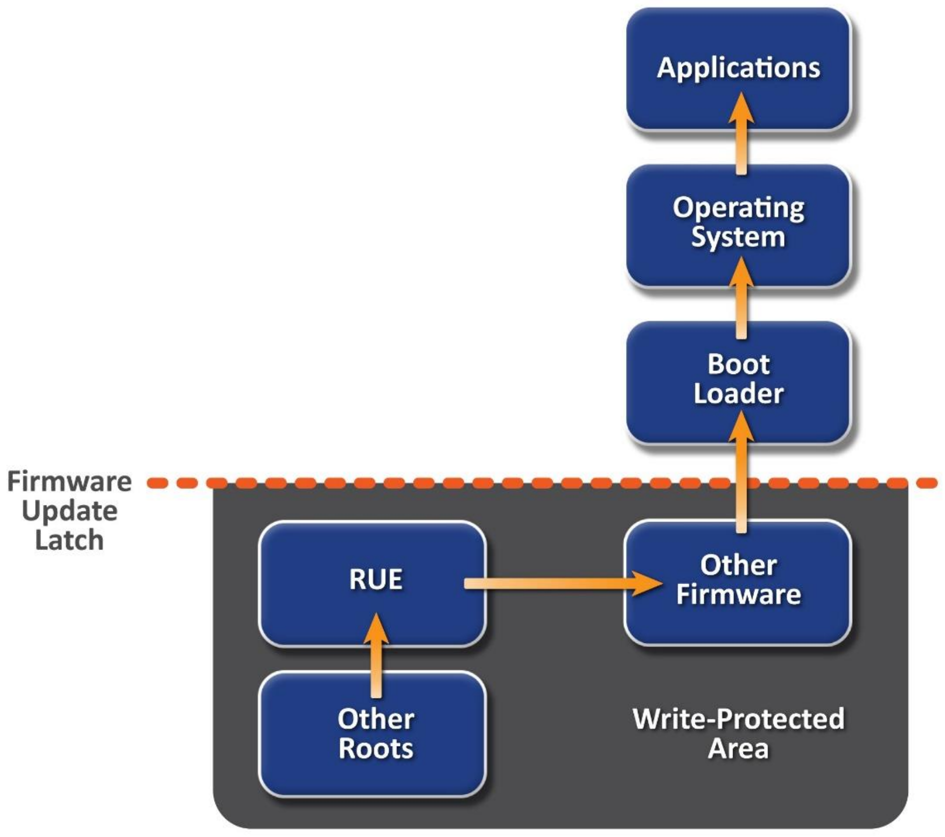


Figure 4 - Firmware Update Latch with RUE

While there are many ways to implement a firmware update latch, a simple method is to have the RUE run early in the boot sequence. When the RUE has completed its work, it write-protects the firmware, typically by flipping a write-protection bit in a manner that cannot be reversed without a reboot. This mechanism ensures that only the RUE or firmware that runs earlier in the boot sequence can modify the firmware. Software and firmware that runs after the RUE may be able to request updates to the firmware by placing the updates into some special location and rebooting so that the RUE will find, verify, and install the updates. But code that runs after the RUE cannot directly update the firmware.

More sophisticated implementation methods may involve an RUE that can be invoked without needing to reboot, perhaps involving entry into a trusted execution mode. Those methods require a more sophisticated method of unlocking the firmware update latch so that the RUE can perform a firmware update while minimizing the risk of interference from attackers who want to inject malicious updates. At this time, several proprietary mechanisms exist that can be used for this purpose.

Following sections will evaluate the Firmware Update Latch with RUE relative to the capabilities recommended in sections 3.4 and 3.5.

5.3.1 Capability "Restrict updates to minimal RUE"

By definition, the firmware update latch provides this capability. However, it's a simple tool that must be used properly by the system designer. The system designer must ensure that the RUE is carefully coded and minimized to prevent attackers from exploiting bugs and vulnerabilities in the RUE to maliciously modify firmware.

In order to protect keys and code that are critical to the update process from unauthorized modification, those keys and code will need to be included in the memory that is protected by the firmware update latch.

5.3.2 Other Capabilities

Most of the other capabilities listed in sections 3.4 and 3.5 will depend on the capabilities provided by the RUE and can be implemented by a carefully written RUE with supporting services. For example, the RUE may or may not include the ability to decrypt updates.

However, several of the capabilities listed will require more than just a well-written piece of RUE code. In particular, specialized hardware support will generally be required to implement physically tamper-resistant storage or execution for critical keys and code or to implement remote attestation.

5.4 Device Identifier Composition Engine with Root Update Engine

The goal of DICE technology, specified by the TCG, is to develop approaches to enhancing security and privacy with minimal silicon requirements. It combines simple silicon capabilities and software techniques to establish a cryptographically strong device identity, attest software and security policy, and assist in safely deploying and verifying software updates. A DICE is an engine that is relied upon to HMAC or hash a device's secret (the Unique Device Secret) with the measurement of the first mutable code that executes on the device when the device boots. The resultant signature or digest is called a Compound Device Identifier (CDI). The measured first mutable code can use the CDI to create cryptographic keys via a Key Derivation Function. As long as the first mutable code properly restricts access to the CDI, one such cryptographic key (statistically) uniquely identifies both the specific device and the first mutable code that booted on the device. Another cryptographic key derived with the CDI can unseal data belonging to the first mutable code on that particular device.

Usually, the RUE associated with a DICE is mutable software that is measured by the DICE and incorporated into the CDI. A DICE alone cannot provide a full secure update mechanism, but rather the DICE can provide fundamental primitives that a RUE can leverage to implement secure update.

5.4.1 Capability "Verify updates"

The measurement and identity provided by DICE can be used in several ways to enhance the security of the RUE. First, if the manufacturer's public key is part of the RUE, the CDI derived from measuring the RUE will include and reflect the correctness of that public key. Second, the CDI can be used to derive a secret used to calculate an HMAC across the update package. This can be used for online verification of the update package and the device identity.

DRAFT

5.4.2 Capabilities "Failed update detection" and "Recovery from failed update"

If an update of the first mutable software fails, the resultant CDI will not match the expected value. This CDI mismatch can provide evidence that the update has failed. This CDI mismatch can be detected locally via sealed values (which can no longer be unsealed) or detected remotely via failed authentication. Recovery from failed update can be aided by the detection mechanisms just described, but it will be dependent on other mechanisms to trigger recovery.

5.4.3 Capability "Decrypt updates"

The DICE engine provides several fundamental capabilities that can be used to implement update decryption. For example, the key used to decrypt the update can be derived from the CDI.

5.4.4 Capability "Measure and attest"

The DICE natively supports the capabilities needed to measure and attest to the integrity of the software and firmware on the platform. Measurements are performed during the boot sequence, and the device's CDI and secret keys are derived.

Attestation may be performed in several ways. The document "Implicit Identity Based Device Attestation"[32] describes one mechanism, whereby an extension within an X.509 certificate provides verification of device integrity. Because this mechanism uses X.509 certificates, it's highly compatible and interoperable with existing infrastructure.

5.5 Trusted Platform Module with Root Update Engine

The Trusted Platform Module (TPM) includes a set of Roots of Trust specified by the TCG. This section presents the benefits of using a TPM to achieve some capabilities identified in sections 3.4 and 3.5, as well as the main TPM commands involved. Some capabilities presented herein, such as the decryption of a payload, cannot be handled by a TPM prior to the 2.0 family.

Software TPMs support the TPM 2.0 library of commands [TPM2L] like hardware TPMs. Software TPMs are presumed to be executed on the same processor as the device main functionality. The security features presented below assume that TPM sensitive elements, like secret and private keys, are not accessible by the main software running on the processor. This implies that the TPM is isolated as presented in section 5.1.

Hardware TPMs assure such isolation by design as they are physically distinct from the main processor. They also typically include crypto-accelerators and hardware random number generators. The acceleration factor may however be decreased by the transfers between the TPM and the main processor.

A TCG Software Stack (TSS)[33], some of which are open-source, makes it easier to develop an application that includes a TPM.

5.5.1 Capabilities "Decrypt updates" and "Tamper protect critical code & keys"

These capabilities can be achieved by leveraging the TPM's ability to safeguard cryptographic keys and decrypt payloads. The update images are symmetrically encrypted for their transport between the update distribution server and the device. The symmetric encryption key is asymmetrically encrypted for its distribution to the TPM. The critical keys are protected by the TPM and are not available as plaintext in the processor memory space.

For the avoidance of doubt, the TPM does not protect the confidentiality or integrity of the application code at rest in the device memory. Firmware update latches (cf. section 5.3) can be used in addition to protect this integrity. Such protection can optionally be controlled by the use of PCRs (Platform Configuration Registers) that record the measurements of previously loaded configurations and firmware.

[32] https://trustedcomputinggroup.org/resource/implicit-identity-based-device-attestation
[33] https://trustedcomputinggroup.org/work-groups/software-stack

DRAFT

The main steps here are:

- The genuine TPM is first enrolled with the distribution server (refer to section “13.8.1 Taking Ownership” of [TPM2L] part 1).
- The symmetric decryption key is sent to the TPM and recorded in its protected storage with the TPM2_Load() command. It can later be updated the same way.
- The payload can then be decrypted with the symmetric key using the TPM2_EncryptDecrypt() command.
- Additionally, a policy based on a PCR can disable the use of the symmetric key if the measurement of previously loaded firmware, such as the RUE, does not match the expected value. Such a policy is defined using the TPM2_PolicyPCR() command.
- Several PCR values can be declared via the TPM2_PolicyOR() command that allows the combination of policies.

The level of tamper protection of the keys loaded in a software TPM will be directly bound to the assurance level of the isolation of the software TPM. The level of tamper protection of the keys loaded in a hardware TPM will generally be attested by a Common Criteria certificate[34].

5.5.2 Capability “Verify updates”

The TPM can verify signatures using the TPM2_VerifySignature() command. Both asymmetric signatures and symmetric authentication codes (HMAC) can be verified via this command.

The transport of the secret key used for HMAC verification can be encrypted all along its transfer from the update server to the TPM and be loaded using the TPM2_Load() command into the TPM. The secret key will never appear in cleartext in the main processor address space.

In the case of asymmetric signature verification, the TPM does not need to provide additional security as public keys are used. However, hardware TPMs can provide acceleration for this operation.

5.5.3 Capabilities “Measure and attest” and “Detect failed updates”

The TPM natively supports the measurement and attestation capabilities, provided the platform includes the supporting infrastructure, such as a CRTM (code root-of-trust for measurement). The DICE technology (cf. section 5.4) can offer such a service.

To perform measurements, the object to be measured is hashed (either by the main processor or via the TPM2_Hash() command). This hash and subsequent hashes are then extended into a PCR (Platform Configuration Register) using the TPM2_PCR_Extend() command.

The use of TPM objects can be bound to a specific PCR value with a policy using the TPM2_PolicyPCR() command. Several PCR values, representing several authorized versions, can be declared via the TPM2_PolicyOR() command.

The remote attestation capability based on PCR is achieved using the TPM2_Quote() command. The attestation can be used to detect failed updates.

Software TPMs can support this provided they have been loaded and activated before the objects that they need to measure. As an illustration, this will presumably be applicable to applicative firmware/software but not to the boot loader.

5.5.4 Capability “Restrict updates to minimal RUE”

This section mostly applies to hardware TPMs.

[34] The protection profiles of PC Client and Automotive-Thin hardware TPMs target the EAL4+ evaluation assurance level.

DRAFT

In this capability, the intention is to ensure that only the genuine and trusted RUE can write a new image into the processor memory. This is achieved by disabling write access at start up, and then enabling write access when the RUE has been measured and is in its expected state. Write access is then disabled in the final stages of the RUE before passing the control to the application.

To achieve this write control, a GPIO line controlled by the TPM can be used to enable or disable write accesses to the areas of the external memory that contains the firmware or software. External hardware support is required.

GPIO lines are controlled by the TPM as NV indices. The primary command is TPM2_NV_Write(). To perform the desired functionality, a policy based on the PCR that measures the RUE is asserted via the TPM2_PolicyPCR() command. Several PCR values, representing several authorized versions, can be declared using the TPM2_PolicyOR() command that allows the combination of policies.

After the RUE has written the image to its memory space, the write access is deactivated. Further activation of this write access is prevented via the TPM2_NV_WriteLock() command. Proper attributes should be set so that this lock persists only until the new start up.

Software TPMs are not expected to manage GPIOs. Alternative solutions using isolation techniques described in section 5.1 are usually preferred.

DRAFT

6 CONCLUSION

The designer of each embedded system must assess the attacks that they must resist and the consequences of a successful attack. This assessment will help them to weigh the costs and benefits of the various secure update approaches, finding one that provides adequate protection for their system.

A system is only as secure as its weakest link.

A firmware update latch with RUE provides a large amount of value while adding only a small amount of complexity. Adding a DICE provides additional benefits for secure updates, such as local and remote attestation. And using a TPM goes beyond the benefits of DICE, adding secured storage, random number generation, etc. The protections provided by a hardware TPM considerably exceed those that can be provided by a software TPM due to the lack of tamper-resistance and generally late start time for the software TPM.

DRAFT

7 SOLUTION EVALUATION

This section illustrates the degrees to which building blocks described in section 5 satisfy the constraints and requirements of section 4. These assessments are subjective, not definitive, so individual readers' own assessments may differ from these assessments.

7.1 Capabilities

7.1.1 Critical capabilities for platform updates

The capabilities in Table 3 are critical capabilities for platform software and firmware updates.

Critical Capability	FW Update Latch with RUE	DICE with RUE	SW TPM with RUE	HW TPM with RUE
Restrict updates to small RUE	Yes	No, unless another mechanism is provided	No, unless another mechanism is provided	Yes, can create latch w GPIO & policy based on PCRs
Verify updates	Depends on RUE	Depends on RUE	Depends on RUE	Yes, can use TPM to verify signature
Detect failed update	Depends on RUE	Yes	RUE may detect failed update but SW TPM can't help because not available early in boot	Yes, with attestation
Recover from failed update	Depends on RUE	Depends on RUE	RUE may recover from failed update but SW TPM can't help because not available early in boot	Depends on RUE

Table 3 - Critical Capabilities for Updates

7.1.2 Desirable capabilities for platform updates

The capabilities in Table 4 are desirable for platform software and firmware updates.

Desirable Capability	FW Update Latch with RUE	DICE with RUE	SW TPM with RUE	HW TPM with RUE
Decrypt updates	Depends on RUE	Depends on RUE	Yes, with TPM2_EncryptDecrypt	Yes, with TPM2_EncryptDecrypt. Also, TPM can protect symmetric decryption key.
Tamper protect critical code & keys	No	Can't prevent tamper but may detect	Depends on the isolation solution	Yes, for keys. Code (RUE) protection depends on external mechanism such as a latch.
Prevent rollback attacks	Depends on RUE	Depends on RUE	Depends on RUE	Depends on RUE
Measure & attest	No	Yes, a particular strength	Yes, but depends on security of isolation solution and underlying code	Yes, assuming the supporting infrastructure (RTM, etc.) is in place

DRAFT

Table 4 - Desirable Capabilities for Updates

DRAFT

8 OTHER SOLUTIONS

This section describes and evaluates some other software and firmware update mechanisms that are popular in embedded systems. Further, it describes how these mechanisms can be used with TCG technologies.

8.1 A/B Updates

A/B updates (e.g., [Android A/B]) use two sets of partitions to store the software in non-volatile storage, referred to as slots A and B. At any point in time, the system is only running code from one of these slots: the “current” slot, as illustrated in Figure 5. This approach makes updates more fault resistant. If an update fails, the older code is available in the other slot as a fallback. This is especially attractive when the code being updated is so fundamental that no underlying component can reinstall it if the update fails.

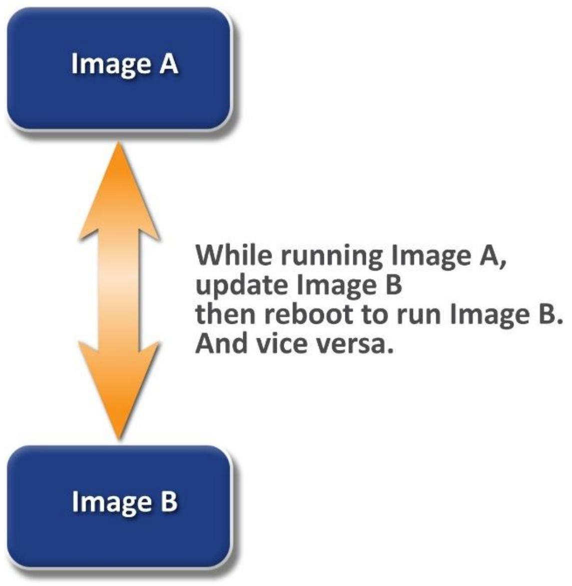


Figure 5 - A/B Updates

Updates can run either in full or streamed fashion. Full updates must be completely downloaded into another storage area (e.g., a data directory). Then they can be run and will write a new system image to the unused slot. Streaming updates write blocks directly to the unused slot as they are downloaded, without having to store the blocks in the data directory. Thus, streaming A/B updates only need a small amount of temporary storage for metadata.

One of the primary advantages of A/B updates, also known as seamless updates, is that they ensure a workable booting system remains on the disk during an over-the-air (OTA) update. This approach reduces the likelihood of a bricked device after an update. A second advantage is that OTA updates can occur while the system is running and functioning. In cases where the update operation fails, the system will fall back to the current slot.

One of the primary disadvantages of A/B updates is that they require enough storage to hold two copies of the system image. The storage used by the inactive image is “wasted” most of the time. Another disadvantage of this technique is that rebooting after an update may take longer than normal if special validation procedures must be performed on the new image. Optimizations may reduce this slowdown, for example by performing the validation steps before the reboot.

Using TPM and DICE with A/B updates can provide several additional benefits. TPM can be used to protect the keys used to decrypt or verify updates and to verify signatures. TPMs are an excellent source of random numbers.

DICE can be used to measure boot code and to restrict access to data to certain software if that data was encrypted with a derived key. A TPM can do this also using sealing.

8.1.1 Capability "Verify updates"

Verification of updates is commonly done in A/B update systems. However, if this verification is done by application or operating system code instead of by the RUE or some other highly trusted code, the verification checks may be bypassed by malware that infects the OS or update application. To relieve these substantial security issues, update verification should be done by highly trusted code. This can happen after the update has been installed and a reboot triggered but this will slow down boot time after an update. Alternatively, some systems may be built to perform update verification using highly trusted and protected mechanisms but in parallel with normal operation before the update code is activated.

8.1.2 Capabilities "Failed update detection" and "Recovery from failed update"

A/B updates excel at recovery from failed updates because fallback to the inactive system image is relatively simple compared to in-place updates.

Detecting failed updates can be done via a variety of mechanisms such as post-update checks of integrity or functionality. If the updated system fails these checks, recovery can be triggered.

8.1.3 Capability "Decrypt updates"

For A/B updates, the advantages and challenges of decrypting updates are not substantially different from other update mechanisms.

8.1.4 Capability "Measure and attest"

With A/B updates, the advantages and challenges of update measurement and attestation are not substantially different from other update mechanisms.

8.2 Verified Boot

Verified Boot (e.g., [Android AVB]) strives to ensure all executed code comes from a trusted source, rather than from an attacker or corruption. It establishes a full chain of trust, starting from a hardware-protected root of trust to the bootloader, to the boot partition and other verified partitions including system, vendor, and optionally OEM custom partitions. During the device bootstrap sequence, each stage verifies the integrity and authenticity of the next stage before it hands off execution and control to that stage. Verified boot may also include anti roll-back protection mechanisms.

While verified boot is not an update mechanism per se, it is a technique that can be used in conjunction with many update mechanisms. As noted in section 3.4, update verification is an essential step in the update process. Even if the update was verified before installation, there is great value in verifying the integrity and authenticity of the system image during every boot sequence. This mechanism can detect unauthorized changes to firmware or software and also as a way to verify that an update was successful and resulted in authorized code.

In some cases, device manufacturers can use verified boot to restrict which software can run on a particular device. However, this is not the only way that it can be used. If the device owner controls the policies and trust anchor certificates used to verify the software and firmware, they can use verified boot to ensure that their own preferences are complied with.

DRAFT

9 RELATED DOCUMENTS AND STANDARDS

Several standards and guidelines have been published in recent years pertaining to secure software and firmware update for embedded systems and more are in development. NISTIR 8200 provides a good summary of relevant standards in this area.

9.1 NIST SP 800-147

NIST SP 800-147 ***BIOS Protection Guidelines*** focused on PCs whereas this document focuses on embedded systems. Some of the differences between PCs and embedded systems are:

- Limits on power consumption in some embedded systems
- Lower compute and storage capabilities in some embedded systems
- Stricter uptime requirements in some embedded systems
- Stricter real-time requirements in some embedded systems
- Strictly controlled software stack on most embedded systems
- Strict controls on boot time (which applies to both PCs and embedded systems)

9.2 SOG-IS

https://www.sogis.eu/documents/cc/domains/sc/JIL-Application-note-on-security-requirements-on-code-loading-v1.0.pdf

SOG-IS is the European recognition agreement for CC certification. SOG-IS is publishing mandatory or guidance documents in order to support CC evaluation; some of them are adopted at CCRA level which is the international recognition agreement (not this reference). This reference is CC oriented but it starts from general security principles that could be a good source of inspiration.

DRAFT

10 REFERENCES

[Android A/B]	***A/B (Seamless) System Updates***, https://source.android.com/devices/tech/ota/ab
[Android AVB]	***Android Verified Boot***, https://android.googlesource.com/platform/external/avb
[D-RTM]	***TCG D-RTM Architecture*** Version 1.0.0, https://trustedcomputinggroup.org/resource/d-rtm-architecture-specification
[GLOSSARY]	***TCG Glossary*** Version 1.1, Revision 1.00, http://trustedcomputinggroup.org/resource/tcg-glossary/
[LAMPSON]	B. Lampson, M. Abadi, M. Burrows and E. Wobber, ***Authentication in Distributed Systems: Theory and Practice***, ACM Transactions on Computer Systems, 1992.
[NISTSP800-147]	***BIOS Protection Guidelines***, April 2011 – NIST SP 800-147: https://csrc.nist.gov/publications/detail/sp/800-147/final
[NISTIR 8200]	***Interagency Report on the Status of International Cybersecurity Standardization for the Internet of Things (IoT)***, November 2018 – NISTIR 8200: https://doi.org/10.6028/NIST.IR.8200
[ORANGE-BOOK]	***Department of Defense trusted computer system evaluation criteria***, DoD 5200.28-STD, 1985: http://csrc.nist.gov/publications/history/dod85.pdf
[TPM2L]	***Trusted Platform Module Library Specification***, Family “2.0”, Level 00, Revision 01.38 – September 2016, parts 1-4: https://trustedcomputinggroup.org/resource/tpm-library-specification/

DRAFT

11 TERMINOLOGY

Here is a list of acronyms and terminology used in this document. See the TCG Glossary [GLOSSARY] for definitions of TCG terms.

ASLR	Address Space Layout Randomization
CA	Certificate Authority
CDI	Compound Device Identifier
CVE	Common Vulnerabilities and Exposures
DDoS	Distributed Denial of Service
DoS	Denial of Service
DICE	Device Identifier Composition Engine
D-RTM	Dynamic Root of Trust for Measurement
GPIO	General Purpose Input Output
HMAC	Hash-based Message Authentication Code
MMU	Memory Management Unit
MTBF	Mean Time Between Failure
PCR	Platform Configuration Register
PKI	Public Key Infrastructure
PLC	Programmable Logic Controller
RoT	Root of Trust
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
RUE	Root Update Engine
SSDLC	Secure Software Development Life Cycle
TPM	Trusted Platform Module
TOCTTOU	Time Of Check To Time of Use
Trusted Computing Base	Lampson et. al [LAMPSON] define the TCB as “a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.” The Orange Book [ORANGE-BOOK] defines the TCB as “the totality of protection mechanisms within it, including hardware, firmware, and software, the combination of which is responsible for enforcing a computer security policy.”
Trusted Platform	See TCG’s Glossary [GLOSSARY]
Update Engine	An engine in a platform that can modify the behavior of one or more engines in the platform