FIDO Device Onboard: The Device Key

December 2022

Editors:
Geoffrey Cooper, Intel
Giri Mandyam, Qualcomm
Hannes Tschofenig, Arm

Abstract

The FIDO Device Onboard (FDO) protocol permits automatic “out of box” onboarding for IoT and other devices. The protocol derives trust in the device from a device key, which is required by the specification, but not provided by FDO itself.

This document builds on the FDO specification to describe various techniques for securely storing the device key, which is used in FDO for attesting the security posture of the device. The topic of device attestation is vast, and this whitepaper is not intended to be a full treatise on this topic. Instead, the focus is on common mechanisms used to store and access device keys, and how they apply to FDO.

It is common to tie the device initialization to the device key during manufacturing. This is often called the “Root of Trust key.” We discuss how the Root of Trust relates to the FDO device key. Measurements of the hardware and firmware can then also play a part in device initialization. FDO can be used to extend this measurement to remote control of the device by the FDO “owner.”

The most important aspect of securing FDO is to assess the security threats that have to be mitigated. Based on those, security requirements for the device are derived. Appropriate solutions can then be selected to meet these requirements.

Contents

Figures

1. Introduction

FIDO Device Onboard (FDO) is an automatic onboarding protocol from the FIDO Alliance. It allows devices to be shipped from the manufacturer to their target destination, even if the ultimate onboarding target is not yet known. At the target, an installer unboxes the device and connects it to a network and power source. Then the device onboards over the network to the server of its owner, automatically and efficiently. Only limited trust and training is needed for the installer, who has no access to the device key or owner credentials.[1]

To make FDO onboarding happen, devices are configured, often while still in the factory. The first requirement of this configuration is that the device contains a securely stored "device key," which is a Public Key Infrastructure (PKI) private key. The certificate for the device key is stored in a FDO credential that is created in the factory and sent to the owner's server, called the "Ownership Voucher." The FDO protocols serve to associate the cryptographic affinity of the private key (in the device) and the public key (in the cloud) to an affinity between the device and its owner.

The fundamental trust in the device thus starts with trusting the device key. If the key's origin is the device manufacturer, then the manufacturer-consumer relationship provides the foundation of trust. Even so, the device travels through the supply chain from manufacturer to consumer, and other parties potentially have access to the device while it is in route. For this reason, the manufacturer needs to secure the device key within the device. This paper will review several common mechanisms for securing the device key. This is not intended to be an exhaustive review of this topic, which is intertwined with many fundamental aspects of computer security, but rather a guide for FDO consumers to understand how their devices perform this critical function.

First, we offer a brief overview of FDO protocol operation.

2. Automating Device Onboarding for General Purpose Devices

After factory provisioning, the device and its Ownership Voucher pass through the supply chain. The Ownership Voucher is endorsed to supply chain targets using digital signing. The final endorsement is to the device owner, the server or service that will control the device in operation.

Prior to onboarding the device, the owner registers the server IP address where it receives FDO owner requests with a Rendezvous server. The device accesses this same Rendezvous server to determine the candidate owner's server IP address, allowing the choice of the owner to be determined at onboarding time. Then the device accesses the owner server, and authentication takes place.

The owner identifies itself using a chain of endorsement signatures from the original factory provisioning, through any supply chain endorsements, to the owner itself. The device verifies the chain by starting with the factory's public key, as originally configured, and working its way through the chain. The final signature is verified when the owner proves possession of its private key using a digital signature.

The device identifies itself to the owner using the device key, which is verified using the device certificate, located within the Ownership Voucher (see Figure 1). The device key itself is bound strongly to the hardware so that it identifies the device well. The signature proves to the owner that the device has this key, without sending the key itself. This is a classic digital signature authentication:

[1] Full details of FIDO Device Onboard are described in the FIDO Alliance web site: https://fidoalliance.org/specifications/ download-iot-specifications/.

A. Owner sends a random number, called a "nonce."

B. Device signs the nonce using its private key.

Owner verifies the device signature using its copy of the nonce and the device certificate.
If the device key is stored well within the device (the topic of this paper), the owner can trust that it is communicating with the one and only such device and can proceed to send onboarding instructions.

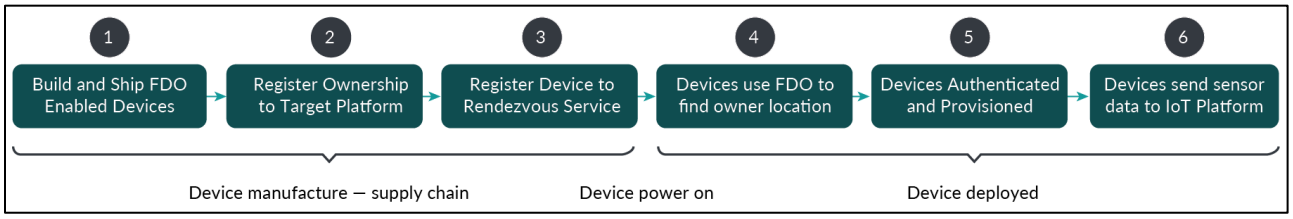

Figure 1 - Overview of FDO Operation

3. FDO Ownership Voucher

The Ownership Voucher is a digital textual message. It is cryptographically mated to the device factory credentials, so that it allows the IoT Device to distinguish the late-bound owner, even if both are in a closed network.

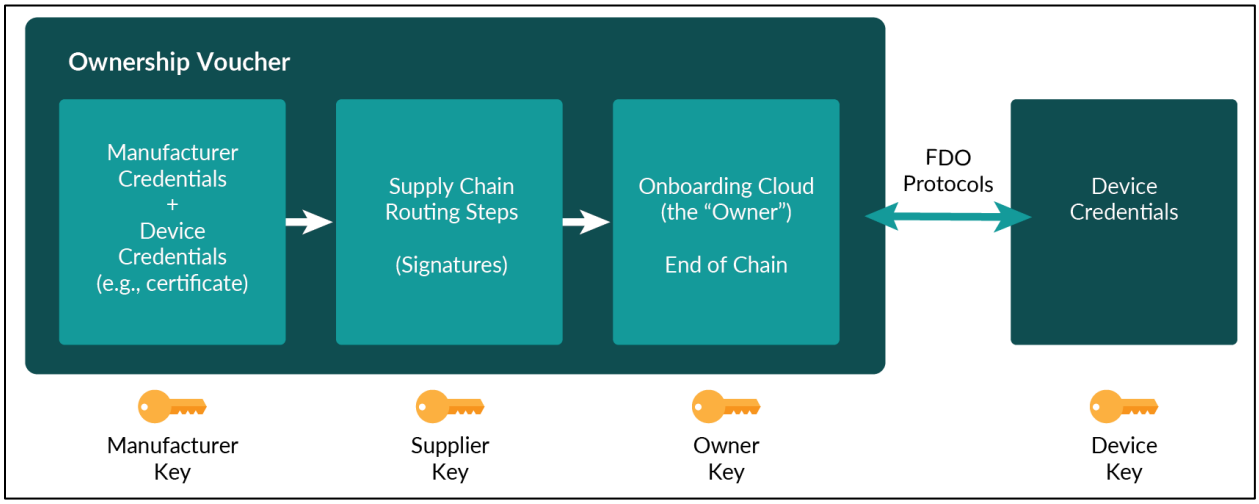

Figure 2 - Ownership Voucher

Once the device and owner server have completed two-way authentication using digital signatures, we can also apply a key exchange algorithm, such as Diffie-Hellman or ECDH (Elliptical Curve Diffie Hellman) to create a shared secret between the device and the owner server. The shared secret allows integrity and confidentiality protected communication, creating a secured channel between the device and the owner server.

The advantage of a secured channel is that it is now possible for the owner server to send arbitrary data and configuration updates, additional keys, databases, code updates, and so on to the device.

To recap, the FDO process started with the device key and owner server key. The owner server and the device exchange digital signatures to perform mutual authentication. Then a key exchange happens, and an encrypted tunnel is established. The result is a protocol where any kind of onboarding data can now be safely and privately shared between owner server and device.

4. FDO Implementation Strategies

The above presentation of how FDO works is intended to raise the questions that this paper will answer: how to store the device key, what level of protection is needed, what kinds of processor features are brought to bear, and so on.

The first step is to understand the security requirements around the device key and the FDO implementation. Based on these requirements, a device with suitable security mechanisms can be selected. We will discuss some of these security mechanisms further on. Before going directly into this discussion, it is useful to present some concrete examples of FDO configurations on common computer configurations.

4.1 Linux-based IoT Controllers

Linux is a popular operating system for higher-end controllers in many areas of application. This paper focuses on Linux due to its general adaptability and large product footprint. How would FDO and the device key be configured and protected on such a system?

In Linux, a single "super-user," called **root**, runs a single program when the system starts. This startup program runs all the other programs. The startup process is structured to enable various common peripheral devices and system services in layers, so that each layer can use the system services from the previous layer. Typically, this is a list of services to run in order, relative to other services. Before FDO can run, the networking services must start up to allow network access for FDO. However, FDO should run before more general network services, such as remote access or login, are started.

Once the FDO process is running, it needs access to the device key. One possibility is to put the device key in the Linux file system. This is not impossible, but requires extra steps in order to secure the key, which may be hard to achieve:

- The directory must be secured. Usually, keys would be stored in a subdirectory of **/etc,** so there is no later confusion about whether these need to be secured. For example, SSH keys are stored by default under the subdirectory **/etc/ssh**.
- The file system must be secure. If an attacker can remount the file system under another operating system, file system protection can be subverted.

Once all these steps have been performed, FDO can store the device key in the Linux file system and be reasonably protected against an attacker from normal OS activity. However, a root exploit on the system could still compromise the device key (and other FDO credentials), or even exfiltrate them to one or more computers, which could then masquerade as legitimate FDO devices. This is true even if the file system is encrypted against outside modification.

Physical attacks against the Linux system require further measures. If an attacker can boot a different OS on this hardware, even an encrypted file system could still be accessed. Different IoT processors have different ways of securing the boot process. A common mechanism on Intel-based processors is to use UEFI secure boot with protected access to BIOS. Of course, access to UEFI itself must then also be secured. An overview of different approaches to secure boot in IoT devices can be found in Appendix B.

For these reasons, storing the FDO key in the Linux file system, with root protection, is not a generally recommended practice.

A processor with a Trusted Execution Environment (TEE), such as Arm TrustZone or Intel CSE, can also implement a key encryption scheme based on a Hardware Unique Key (HUK) stored in One Time Programmable (OTP) memory accessible only to the TEE. FDO credentials are encrypted by the TEE and then stored encrypted in the file system. The TEE itself can then provide the encryption services needed to perform FDO or even implement the entire FDO protocol.

To better protect the FDO credentials, the FDO device key might be placed in a hardware key store. In Linux, TPMs and Secure Elements are examples of key stores that can be added to the computer outside the processor. The key store allows the FDO device key and credentials to be protected. These and device-specific key stores can also be integrated into the processor chip silicon. For example, an on-chip Replay Protected Memory Block (RPMB) can be coupled with Arm TrustZone or an on-chip security coprocessor to create an on-chip secure element.

In the best case, the key store implements both storage for the key and cryptographic algorithms to use the key. The key store can be configured to prevent the key from ever being read directly (i.e., it can only be used to encrypt/decrypt or sign/verify by a program running on the computer). In this way, even a root-level attacker cannot exfiltrate the key.

4.1.1 MCU-Based IoT Controllers

Micro Controller Unit (MCU) based IoT controllers integrate processors, RAM, and flash memory onto a single chip. These MCUs are less complex due to the absence of certain hardware features, such as the use of a memory protection unit (MPU) instead of a memory management unit (MMU) and absence of virtualization support. MCU systems also use simplified operating systems that allow a single software program to control all aspects of the MCU.

In the simplest MCU design, software, hardware, and keys all reside at the same security level. In this situation, FDO operates as part of the single body of MCU firmware. FDO can allow onboarding in this situation, by including the FDO credentials in on-chip flash. The problem here is that all the code in the MCU has access to the FDO credentials. This means that any security bug in the MCU software could compromise the onboarding process. This kind of solution may still be appropriate for simplistic applications, or applications coupled with physical device security.

Today, almost all MCUs also provide memory protection units, which allow FDO to run in a protected memory region. A correctly programmed MPU protects against many software attacks, unless the attacker can run code on the processor that reprograms the protection unit itself.

Arm Cortex v8-M with TrustZone offers an additional layer of isolation on top of the MPU capabilities by splitting the secure and non-secure processing environments.

Increasingly, MCUs provide multi-core, where different CPU cores provide truly independent operating environments, with a defined communication mechanism between them. For example, a low-end IoT device with a dual-core Arm Cortex v7-M can have one “secure” core dedicated to security functionality and the second core executing the main application program (e.g., Real Time Operating System), Internet protocol stacks, and the application.

In the case of FDO, private keys are held in the secure core and are not accessible from the application core. Any cryptographic operations executed using such key material must be performed within the context of software running on the secure processing environment MCU. This means the main FDO implementation runs in the secure core. This ensures that a flaw in regular application code cannot impact security-sensitive assets. Even so, FDO is designed to allow the FDO networking functions to run in the application core without compromising security, so a dedicated network interface (NIC) is not needed for FDO.

During FDO operation, application credentials and other data are downloaded to onboard the device. The FDO implementation transfers this information to the application core, such as via a dedicated core-to-core message handling unit.

MCU devices require special purpose manufacturing mechanisms to store FDO and other credentials. Security-sensitive values (e.g., keys or device identifiers) must only be provisioned onto the device in memory regions controlled by the secure processing environment. The requirements for a secure provisioning protocol to support FDO are provided in the FDO specification under the Device Initialize Protocol (DI). For example, the DI protocol code can access one-time programmable, non-volatile memory (also known as eFuse) to store immutable security-related information, such as the FDO Device private key. This provisioning process must take place in a secure manufacturing facility.

A device that is physically exposed to attackers may need protection against further threats, namely:

- An attacker performing a physical attack, such as utilizing exposed debug ports or chip-invasive attacks like decapsulation, depackaging, and rebonding.
- An attacker launching side-channel attacks and fault-injection attacks.

A low-end IoT device utilizing hardware crypto such as CryptoCell, in an Arm Cortex v8-M with TrustZone device, is an example of a device that may qualify for certification offering this strong level of protection.

For use with FDO, the hardware crypto is used to store keying material, and cryptographic algorithms are implemented in hardware offering side-channel and fault-injection resistance. Hardware crypto is also used to assist during the boot process and when firmware is updated by storing the trust anchor.

5. Trusting the Device Key

Earlier, we established the requirement to trust the device key for a secure FDO implementation. This starts with the need to trust how the device key is created and initially placed in the device. We shall see how the device key can be bound into the initialization of the device (the “Root of Trust” or RoT), establishing a strong connection between the device key and the device itself. A less strong connection may also be sufficient, such as protecting the key using special hardware, along with special manufacturing procedures to store the key.

A reputable manufacturer with a secured factory floor is recommended. In some situations, the key is injected into the device later in the supply chain, allowing a shorter supply chain to the key injection site. In this case, the trust requirements apply to whomever injects the key.

How the private key is stored is critical to the other trust relationships that are created using FDO. The most fundamental assumption about the device’s private key is that there is only one copy of it. This applies even if an attacker obtains access and can misuse the key.

5.1 Storing the Key in the Device

The goal for storing a private key is that the stored key remains the only copy of the key, for the lifetime of the key, which may also be the lifetime of the device. There are two basic ways in which a key can be revealed:

- There is a copy of the key stored outside the device (perhaps temporarily), and this copy is inadvertently revealed to an attacker.
- The copy on the device is inadvertently revealed to an attacker who has penetrated device security.

Of course, if there is only ever one copy of the device key, the first item can never happen. However, this is not always practical. Sometimes the device is not able to create a key internally. For some devices, they lack the algorithmic power to create a secure key. In some factory environments, powering on the device takes a long time and slows down the manufacturing process, adding expense. Some situations require that the device key can be restored in case of a hardware failure, so a second copy is required to exist. As of late, most applications are now able to transfer the trust associated with the original key to another key, so this situation is becoming less common.

Let us assume that any copies of the private key external to the device are securely stored and inviolate. Then we are left with the problem of securing the key that is on the device. There are several basic techniques:

- The key is created at a specific time in the device power cycle, such as when the device first boots. Typically, the key is then stored using one of the next two mechanisms:
    - The key is only accessible to supervisory code running on the device.
    - They key is stored in a separate hardware processor associated with the device, and access to that hardware is suitably restricted.

All these mechanisms can be used to store device keys for use with FDO, with varying security properties.

The first method, creating the key as part of the device power cycle, is particularly interesting when the key is an RoT, and we will examine it next.

5.1.1 Root of Trust Keys

One common technique is to create a key in a predictable way when a device first boots, then secure the key for later use by programs. A special case is where the key incorporates materials from the boot process (e.g., the bootstrap code itself) into its structure. This “input key material” (IKM) can come from several sources; see below. Such a key is often called the “Root of Trust” key because it is inextricably linked to the device boot process; if an attacker attempts to compromise the device by changing the startup sequence, any key obtained by the attacker will be corrupted by the attack itself.

By its nature, the Root of Trust (RoT) key is not changeable once the device is manufactured. Therefore, a strong storage mechanism is needed to keep the key secure once it is generated. Conversely, the affinity of the device to the key is maximized in the RoT—the signature of this key really defines the device. This makes the RoT key an excellent source of trust as the device key for FDO. During FDO operation, application keys can be provisioned in the device, so the Root of Trust key is not needed during normal device operation, making it harder to identify or correlate device action via its RoT key.

Creating an RoT key is only the beginning of the challenge. The key must also be *stored* within the device so that it cannot be copied or extracted. For devices with supervisory modes of operation, the key can be stored immediately into a hardware storage area that requires the highest level of access. This prevents the key from being accessed except from the most trusted code.

Examples of keys stored in memory that are only accessible to trusted code are common:

- A volatile, but protected, key store may be created out of RAM (perhaps using normal page table protections). If the RoT is computed every time the system boots, this mechanism ensures the RoT key is present only when the system is running.
- Non-Volatile RAM (NVRAM) may be divided to contain a "secure partition" that has hardware protections, so it is only visible to supervisory code.
- Special registers might be provisioned as hardware devices, only visible to supervisory code. A common technique is to use One Time Programmable (OTP) memory.

In the above, the first technique involves predictably reconstructing a key at boot time, then storing it on a volatile medium. The latter two techniques cause keys to be stored in non-volatile memory, raising the possibility of attack when the device is powered off. This might seem less secure. However, non-volatile memory can store any key, even one that comes from outside the device. This can make it easier to initialize the device with the best key possible, especially for smaller devices.

We can also mix and match techniques to get the properties we want. An RoT key can be created as the encryption key for other key storage. This protects keys when the system is powered off. Many computer systems, from small microprocessors to large servers, use this technique. Also, RoT key material can be created and re-created via carefully selected input key material. A more detailed discussion of the different types of input key material can be found in Appendix A.

6. Using a Security Processor

We have shown that we can store keys on a device and protect them from unwanted tampering, even from non-privileged software running on the device. It might seem that to do better, we would need to keep the key off the device itself. With a security processor we can do that, too.

Consider that we do not actually need to have the private key *on* the main processor. We only need to be able to *sign* with the key. If we have another processor available, with its own memory, we can put the key on that processor, and ask it to sign when we need a signature. Then, the private key is not accessible to the main processor. An attack that completely compromises every line of software on the main processor still cannot get a copy of the private key.

Is this really an advantage? Consider an attacker that can completely compromise the computer's main functions. The attacker can sign with a stored key as many times as needed. However, if the key is on a security processor, the attacker still must persist with the attack on this *specific* computer to keep on signing. If we detect the attacker's strange behavior and reset the computer (perhaps reloading its software), the attack is over. But if the attacker could exfiltrate the key from the computer and make many copies, we would never be able to stop the attacker's various computers from continuing to sign with that key. We would have to replace the key and reinitialize or even replace the original computer.

A security processor can help us achieve secure storage of keys and can prevent an attacker who compromises the main computer from exfiltrating the key. In the computer industry, we have also noticed that small security processor computers, which only have limited computing functions, can focus a great deal of design and technology on being as secure as possible. Improved resistance to hardware attack is also sometimes on the list of advantages of a security processor.

Security processors today fall into three categories, namely Trusted Platform Modules (TPMs), processors with cryptographic built-in functions (usually called "Secure Elements"), and Subscriber Identity Modules (SIM) cards. All three can deliver great security benefits. The TPM implements a standard interface, allowing computer systems to support any brand of TPM with the same drivers and high-level software. Secure Elements can offer the same cryptographic functions with a lower cost point. SIM cards have their origin in the mobile phone industry but have evolved since that time into integrated SIMs (iSIMs), which also can be used in devices other than mobile phones.

TPMs, Secure Elements, and SIMs can provide secure key storage for FDO. Also, it is common to embed these security processors inside larger processor chips (in the form of System on Chip designs), sometimes renamed as features of the processor architecture.

7. How Secure is Secure?

We have seen several techniques for securing keys. Most use special hardware or firmware, and some require special security processors. How much security is actually needed for the FDO key?

The FDO key identifies the device. If it is compromised, the attacker can pretend to be the user's device. This is probably not a desirable situation. However, the degree of damage an attacker can do, and the degree of difficulty detecting and recovering from the attack also need to be considered. This depends on the application and needs to be assessed with questions like:

- How much damage can an attack do before it is detected?
- How likely am I to detect an attack, and how long will that take?
- What is the cost of cleaning up an attack?
- Is it even possible to clean up the attack without redeploying equipment?

In practice, we believe good keying practices will yield more reliable IoT installations over the long term.

There are some special situations:

- Where devices are accompanied by personnel for other reasons, trusted personnel might be able to secure the devices by virtue of their presence, allowing less secure credential storage. However, the industry trend is increasingly to include strong key storage in these devices as well.
- Where devices have extremely low value in deployment, such as toys or demonstrations and can be easily updated, less secure credential storage might trade off with easier reinitialization.

Also, some legacy devices might not have sufficient key protections built in, but they might have an encrypted file system or other security measures. As described above, these mechanisms can be used for FDO key storage if the severity of compromise is manageable through external measures.

Of course, where personal safety or physical security considerations are a factor, the key storage should be as strong as can feasibly be deployed.

8. Conclusion

The FDO protocol provides a flexible way to onboard a device. FDO requires attestation from a device key, which can be an RoT key for the device, or a key programmed into the device specifically for FDO. A security analysis of the device and its proposed deployment can be used to choose the nature and protection of the FDO key: whether the benefits of a root of trust key are needed—perhaps including device measurement, perhaps protected by security hardware, perhaps in a security processor, and so on. In this paper we have tried to explain some of these security measures and how they work to protect the key and the computer.

While a security analysis of the IoT deployment is the best way to decide on these tradeoffs, computer processor vendors have bundled key protection mechanisms into their products and can offer convenient security suites that work well for FDO in a variety of situations.

9. Appendix A: Input Key Materials

When RoT keys or other keys are derived at boot time, the input key material (IKM) is critical in ensuring uniqueness of the resultant keys or keypairs. IKM, which is unique to the device, hard to derive and hard to read from outside the device, yields the strongest keys such as are needed for FDO, authentication, attestation encryption, or other cryptographic operations. The IKM may be derived in part or fully via a few well-known approaches common to RoTs. In this section, several methods for instantiating IKM in an RoT are described.

Since the IKM is used to derive the key, it is important that the IKM is hard to detect from outside the device. Also, the IKM needs to be repeatable on multiple boots of the device, but must also have strong guarantees that IKM differs fundamentally between otherwise identically manufactured devices.

Physically Unclonable Function (PUF)

The function of a PUF chip or circuit component is to consistently generate the same number every time it is accessed, but which number is random with respect to all other PUFs. This makes the PUF a good source of identification for a given IoT device. The PUF derives these properties by measuring the subtle deviations in the manufacturing process for semiconductors.

A PUF-derived value can be reused as IKM for derivation key material in an RoT if it is stored in some form of non-volatile memory (NVM) and is made available to the RoT derivation process. Since the IKM is the same every time the PUF is accessed, the device's RoT key is always the same every time it is regenerated (e.g., every time the device is rebooted). At the same time, the RoT key for this particular device is effectively unique with respect to all other devices.

Firmware

Many IoT RoTs consist of programmable subsystems (e.g., microcontrollers or MCUs), which execute code from read-only memories (ROMs). The ROMs contain immutable code, sometimes also referred to as firmware, in which IKM can be provisioned (e.g., through the sampling of a random number generator upon instantiation of the ROM memory). The original random number can be generated using a hardware random number generator in a manufacturing facility that is rated to provide random numbers very quickly.

One-Time Programmable (OTP) Memory

OTP memory is a type of NVM where the bits, once programmed, cannot be changed. During manufacturing, IKM can be directly provisioned into OTP memory. OTP normally takes the form of electronic fuse arrays, but it can also come in the form of specialized on-chip memory (e.g., resistive RAM).

10. Appendix B: Measured Boot and Secure Boot

The RoT key can also be bound up with the boot process of the computer. In these approaches, the RoT key's value is part of a measurement when the computer boots, which prevents normal operation if the RoT or other critical parts of the computer are changed. This provides a measurement of software and firmware integrity.

For most computing devices, software/firmware integrity is verified using three approaches: measured boot, secure boot, or a combination of the two. In this section, a brief overview of secure and measured boot is provided.

Secure Boot

Secure boot describes a boot process by which an initial subsystem (ideally an RoT) in a computing device first validates its own software and then validates a second subsystem's software before initializing and running this second subsystem. The second subsystem can in turn validate one or more additional subsystems, with those subsystems able to validate ensuing subsystems. These validations form a "trust chain." Not all software validation operations have to be executed at the same time, and they do not have to be done in sequence.

Measured Boot

Measured boot describes the process where a subsystem in a computing platform makes a measurement of its own firmware or other subsystem's firmware and stores the measurement in a persistent memory location protected by the RoT. Usually such a measurement takes the form of a cryptographic hash performed over some combination of the measurement data (e.g., software version, timestamp, cryptographic hash of the entire software image, result of a signature verification over the hash). The measurement is usually taken when the device is powered up (i.e., upon "cold" boot). An initialization process creates the first measurement against which subsequent boots are evaluated.

Boot measurements may be verified using data stored inside the device, but a common approach is to send the measurements for remote verification, where a signature of the RoT attests to the provenance of the measurement. The server receiving this "measurement root of trust attestation" verifies it against a value stored at the server location. This removes the possibility that an attacker who breaches the devices could corrupt both the system and the measurements stored for verification.

In fact, boot measurements can be transmitted during the FDO session. FDO allows the owner to negotiate messages with the Device, so the Device can be set up to send measurements over the FDO encrypted channel. Assuming the device manufacturer has published the correct measurements, the FDO owner can then verify the device is still in good stead and has not been compromised during the supply channel.

Comparing and Combining Secure and Measured Boot

A problem for remote devices, such as IoT devices, is the cost of accessing the device to correct a problem (sometimes called the "truck roll" cost). When problems are small, it is cheaper for the device owner to access and repair the device remotely.

Secure Boot implementations are prone to causing a system failure when detected during boot. This is a logical step for a user's laptop since it causes the (perhaps irate) user to bring the laptop in to be repaired. But when there is no user present it adds to cost. Modern systems implement segmented failures from secure boot or have automatic fallback to the previous firmware to prevent failures in this situation.

On the other hand, measured boot is more commonly applied without immediate comparison. Instead, the device creates an attestation that is verified when the device needs remote services. In the case of a user laptop, this prevents the user's machine from corrupting system services (e.g., the mail server), but may leave the user to unwittingly rely on a corrupted machine for local use. However, for IoT systems, which normally access remote services immediately on boot, a remote attestation allows a corrupted device to be identified quickly. Since the measurements can be taken at different device layers, it may still be possible to access the device remotely and repair it. For example, if the base operating system (and remote access server) attests OK, but the hypervisor and applications are corrupted, the system can be repaired remotely, and its subsequent attestation allows it to be trusted again.

In practice, it is common to find both some level of Secure and Measured boot active on a single device. For example, it makes sense to apply Secure Boot (with immediate failure) to the device firmware, if there is no way to recover and boot even part of the system safely. This level of device resiliency is the subject of current research, development, and standardization of IoT.

11. References

"Download IoT Specifications." FIDO Alliance. Accessed May 12, 2022. https://fidoalliance.org/specifications/download-iot-specifications .

Erdem. "What's the Difference between Secure Boot and Measured Boot?" *Jupiter Networks* (blog), September 17, 2015. https://community.juniper.net/blogs/elevate-member/2020/12/22/whats-the-difference-between-secure-boot-and-measured-boot .

"FIDO Device Onboard Specification 1.1." FIDO Alliance. April 19, 2022. https://fidoalliance.org/specs/FDO/FIDO-Device-Onboard-PS-v1.1-20220419/FIDO-Device-Onboard-PS-v1.1-20220419.html#informative.

Gao, Yansong, Said F. Al-Sarawi, and Derek Abbott. "Physical Unclonable Functions." Nature Electronics. https://www.nature.com/articles/s41928-020-0372-5.

Jung, Junyoung, Beomseok Kim, Jinsung Cho, and Ben Lee. "A Secure Platform Model Based on ARM Platform Security Architecture for IoT Devices." *IEEE Xplore* 9, no. 7 (September 1, 2021): 5548 – 5560. https://doi.org/10.1109/JIOT.2021.3109299.

Krau, Michael, and Vincent Zimmer. "Establishing the Root of Trust." Unified Extensible Firmware Interface Forum, August 2016. https://uefi.org/sites/default/files/resources/UEFI%20RoT%20white%20paper_Final%208%208%2016%20%28003%29.pdf .

POC code for FDO is available from the LF-Edge project at: https://github.com/secure-device-onboard.

Szefer, Jakub, and Margaret Martonosi. *Principles of Secure Processor Architecture Design.* Morgan & Claypool, 2018. https://doi.org/10.2200/S00864ED1V01Y201807CAC045.