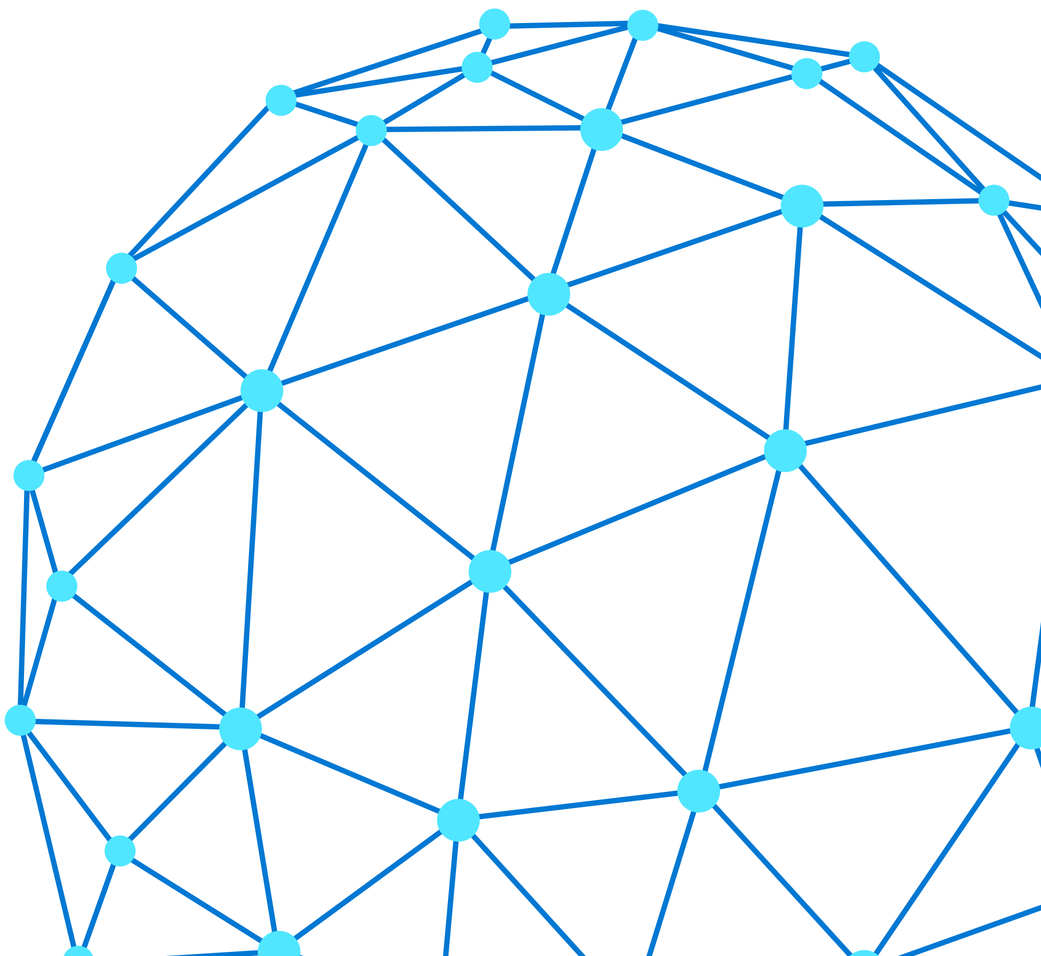
Microsoft Azure

Nineteen cybersecurity best practices used to implement the seven properties of highly secured devices in Azure Sphere

Edmund B. Nightingale and Penny Orwick
Microsoft



July 2020

Contents

Target audience

This paper is written for the technical cybersecurity professional or IT professional interested in learning more about the architecture and implementation of Azure Sphere. We hope that this paper assists in evaluating the measures used within Azure Sphere to improve the security of IoT devices. It may also act as a reference for companies that wish to compare their current and/or competing IoT offering against Azure Sphere's own offering on the market.

Given its target audience, the paper often uses terms of art that, while familiar to many, may be new to some readers of this paper. Wherever possible acronyms are explained, but the paper is not meant as a tutorial or authoritative source for the details of topics like cryptography, encryption, or the architecture of modern operating systems.

Introduction

In 2017, Microsoft introduced a new standard for IoT security by releasing the white paper, "The seven properties of highly secured devices[1]." The paper argued, based on an analysis of best-in-class devices, that seven properties must be present on every standalone device that connects to the internet. The paper also introduced the Project Sopris experiment, which demonstrated a prototype design for low-cost devices that met all seven properties. In 2020, Microsoft launched the Azure Sphere business. The first certified Azure Sphere chip is available from MediaTek in volume today and Microsoft has announced partnerships with NXP and Qualcomm to build new Azure Sphere chips. Microsoft has also announced partnerships with early-adoption customers like Starbucks, which uses Azure Sphere as part of its own digital transformation.

As we near the three-year anniversary of the original seven properties paper, we felt it was important to share our learnings with the wider community. This white paper focuses on two important areas. First, we discuss why the seven properties are always required, even when covering something as simplistic as a cactus watering sensor. Second, we realized that the wider community would benefit from a description of the best practices used to implement Azure Sphere. These best practices are the outcome of decades of experience building secured products, filtered through the lens of Azure Sphere and the MT3620—the first commercially available Azure Sphere certified chip. We hope that this discussion of the *why* and the *how* of the seven properties helps accelerate their adoption across the industry.

[1] https://aka.ms/7properties

The seven properties of highly secured devices

Figure 1 summarizes the seven properties every internet connected device must implement.

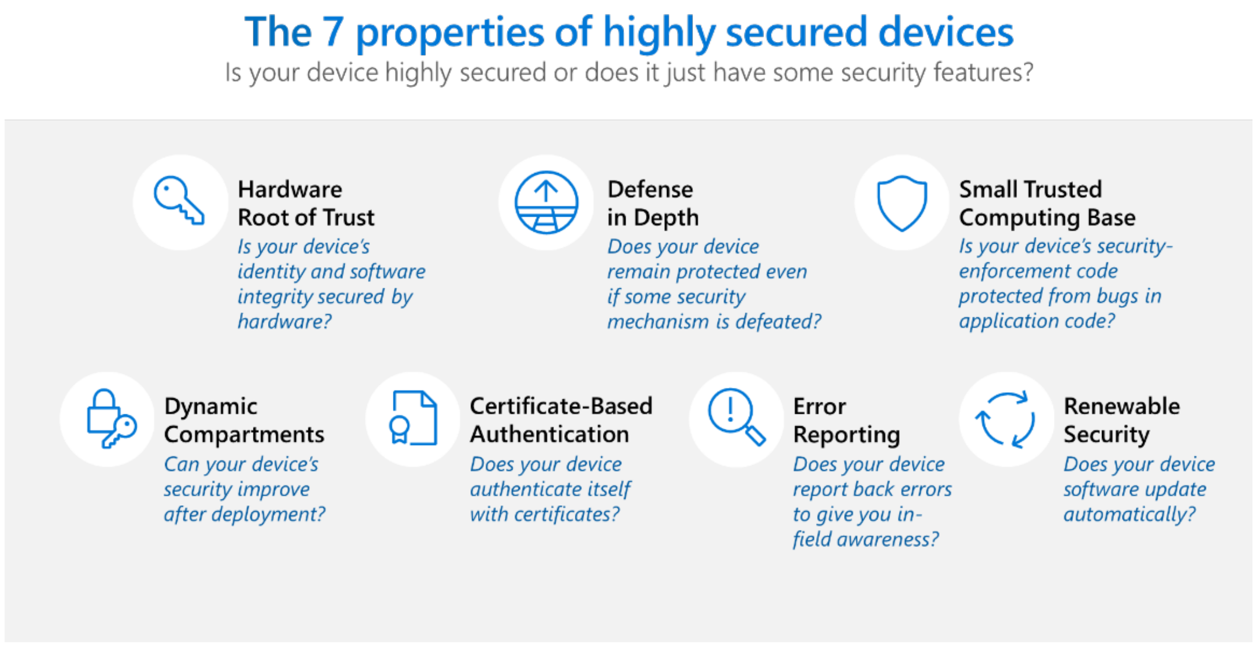

The 7 properties of highly secured devices
Is your device highly secured or does it just have some security features?
Hardware Root of Trust
Is your device's identity and software integrity secured by hardware?
Defense in Depth
Does your device remain protected even if some security mechanism is defeated?
Small Trusted Computing Base
Is your device's security-enforcement code protected from bugs in application code?
Dynamic Compartments
Can your device's security improve after deployment?
Certificate-Based Authentication
Does your device authenticate itself with certificates?
Error Reporting
Does your device report back errors to give you in-field awareness?
Renewable Security
Does your device software update automatically?

Figure 1. The seven properties of highly secured devices.

Some of these properties, like the presence of a hardware-based root of trust or compartmentalization, require certain silicon features. Others, like defense in depth, require a certain software architecture, as well as the presence of other properties, like a hardware-based root of trust. Finally, other properties, such as renewable security, certificate-based authentication, and failure reporting, require not only silicon features and certain software architecture choices within the operating system, but also deep integration with cloud services. Piecing these critical parts of infrastructure together is difficult and prone to errors. Ensuring that a device meets these properties could therefore increase its cost. This led us to believe that the seven properties also introduced an opportunity for security-minded companies to implement these properties as a platform, freeing device manufacturers to focus on product features, rather than security.

Azure Sphere is Microsoft's entry into the market with a seven-properties-compliant, end-to-end, product offering. We are excited to see so much discussion in the marketplace around the seven properties and we look forward to more products entering the market that meet this high bar for securing IoT products. With this summary in mind, let us discuss why all seven properties are required when connecting directly to the internet.

Are there devices that use the internet and do not need the seven properties?

Are there internet-connected devices that are non-critical, and therefore do not require all seven properties? In other words, are the seven properties required only when a device might cause harm if it is hacked? This line of reasoning suggests that there are two types of internet-connected devices: one type, for which security is important, and another type, for which fewer (or no) security features are required because a compromised device is harmless. This paper answers this question by analyzing the least safety-critical IoT device: a cactus water sensor.

In this hypothetical scenario, a company is selling a cactus water sensor. The device senses the amount of water present in the soil and sends a notification via a service, such as Azure IoT, to notify someone when the cactus needs additional water to survive. You might wonder, rightly, what possible harm comes from a hacker taking over this device? If the hacker stops the notifications, surely the cactus will survive a long period without water! Alternatively, if an attacker continues to send the "water needed" messages, the person watering the cactus would note that the cactus does not need water and ignore the notification. What possible need is there for the seven properties? Why require an advanced CPU, a security subsystem, a hardware root of trust, and a set of services to secure this simple device?

Weaponizing MCUs

The answer to these questions is found by examining the Mirai botnet[2], which is a real-world example of IoT gone wrong. The Mirai botnet was composed of approximately 150,000 internet-enabled security cameras. The cameras were hacked and then turned into a botnet that launched a distributed denial of service (DDoS) attack that took down internet access for a large portion of the eastern United States. Note that the botnet that launched this attack was composed of only 150,000 devices; any significant market for an IoT device could be larger.

For security experts analyzing this hack, the Mirai botnet was distressingly unsophisticated: it scanned IP addresses for telnet (yes, telnet!) servers that would accept connections using default credentials like "username: root password: root." After discovering the servers, the botnet would install itself into the memory of the device and then contact a command and control, or "C&C" server, for instructions. Each compromised device added itself to this botnet army by contacting the C&C server and waiting for instructions. The owner of this online army sent a command from the C&C server to "flood" a target with internet traffic to take it offline. A target could be any online presence anywhere in the world. The device could be used as an entry into a private network to attempt to gain access to valuable, private data or the device could be targeted at some other, external online service like an online business, a government agency, or a public utility. These attacks are called distributed denial of service attacks because many

[2] https://www.csoonline.com/article/3124344/armies-of-hacked-iot-devices-launch-unprecedented-ddos-attacks.html

distributed, infected hosts from around the world attack a target to deny it the ability to function online.

Botnets are not rare. They are rented to criminals, used to extort money from businesses by threatening DDoS attacks, and can even become money-making enterprises by using an infected computer to mine bitcoin or other crypto currencies[3, 4, 5]. The Mirai botnet is a great example of internet connectivity as an afterthought. Closed-circuit TV camera (CCTV) systems could be secured if they were never connected to the internet. Only someone physically present could attempt to hack a CCTV, and then they could only hack a single system. Adding internet connectivity means a remote attack *scales* to hundreds of thousands or millions of devices.

The ability to scale a single exploit to hundreds of thousands of devices is cause for reflection on the upheaval that IoT brings to the marketplace. Before internet connectivity, an MCU performed as a single-purpose device whose manufacturer wrote code to do a single thing and nothing more. We refer to this as a "fire-and-forget" device. A traditional microwave oven, or a simple alarm clock, or a non-connected thermostat with a digital display are all examples of fire-and-forget devices. The device does one thing, and, since the device does not have internet connectivity, features like defense in depth, secure software coding practices, or a hardware root of trust are not important. However, once the decision is made to connect an MCU to the internet, that device has the potential to go from a single-purpose device to a general-purpose computer capable of launching a denial of service attack against any target in the world. Further, the Mirai botnet demonstrated that a manufacturer does not need to sell many of these devices to create the potential for a "weaponized" device.

A weaponized cactus water sensor

Where does this leave the canonical cactus water sensor? If these sensors sell even at relatively low volume, and they contain a remotely exploitable security vulnerability, then a hacker can turn that population of devices into a botnet and wreak havoc on an internal network or the internet. Therefore, IoT security is not only about safety-critical deployments. *Any* deployment of a connected device at scale requires the seven properties. In other words, a device's function, purpose, and cost are not the only considerations as to whether security is important. The sheer number of devices also factors into whether a hacker targets a product. To further complicate matters, vulnerable software may be in *many different devices*. If thousands of different devices leverage the same library with a zero-day exploit, then all those devices could be exploited. All software has unknown bugs. Error reporting and renewable security are critical tools to learn about *and fix those bugs* after a product is deployed. If vulnerable software cannot be updated

[3] https://www.fastcompany.com/90417865/new-botnet-nabbed-victims-by-sending-30000-sextortion-emails-per-hour
[4] https://securityintelligence.com/the-many-faces-of-necurs-how-the-botnet-spewed-millions-of-spam-emails-for-cyber-extortion/
[5] https://cointelegraph.com/news/sophisticated-mining-botnet-identified-after-2-years

remotely, or errors detected remotely, then the only way to get those devices off the internet is to disconnect them and recall them.

The seven properties do not guarantee that a device will not be hacked. The seven properties minimize certain classes of threats and make it possible to detect and respond when a hacker gains a toehold in a device ecosystem. Error reporting combined with renewable security would allow remote patching of devices to eliminate the vulnerability. Likewise, defense in depth would make it harder for attackers to take ownership of a device, but vulnerabilities will continue to exist and be exploited by attackers.

If the seven properties are not implemented in a device, then processes not implemented in software must be replaced with human practices. For example, without software update, a security incident requires disconnecting devices from the internet and then recalling those devices or sending people to all locations to manually patch the devices that were attacked.

The first part of this paper addressed the *why* of the seven properties: why they are important, and why they are applicable to every device that connects to the internet. The remainder of this paper addresses the *how* of the seven properties by describing nineteen best practices followed when designing and implementing Azure Sphere.

Azure Sphere background

Before getting into the details, we present an overview of Azure Sphere hardware, software, and terminology to provide context for each best practice. This overview is not meant to be exhaustive. You can find further detail at the Azure Sphere website[6].

Chip architecture overview

Figure 2 shows a representative Azure Sphere chip architecture at a high level. In the upper-left corner of the diagram is a block that shows the Pluton security subsystem. Pluton is at the heart of every Azure Sphere chip and is its hardware root of trust. It stores the keys critical for Secure Boot and remote attestation, it accelerates common cryptographic operations on behalf of the operating system, and it implements the mechanism required to provide access to resources contained within the chip. Each Azure Sphere chip has RAM and flash. As an example, the MT3620 uses on-die SRAM and in-package flash. These design decisions make it more difficult for an attacker to execute a physical attack on the chip by tightly integrating those resources either onto the silicon die, or within the “package” that contains it.

[6] https://aka.ms/azuresphere

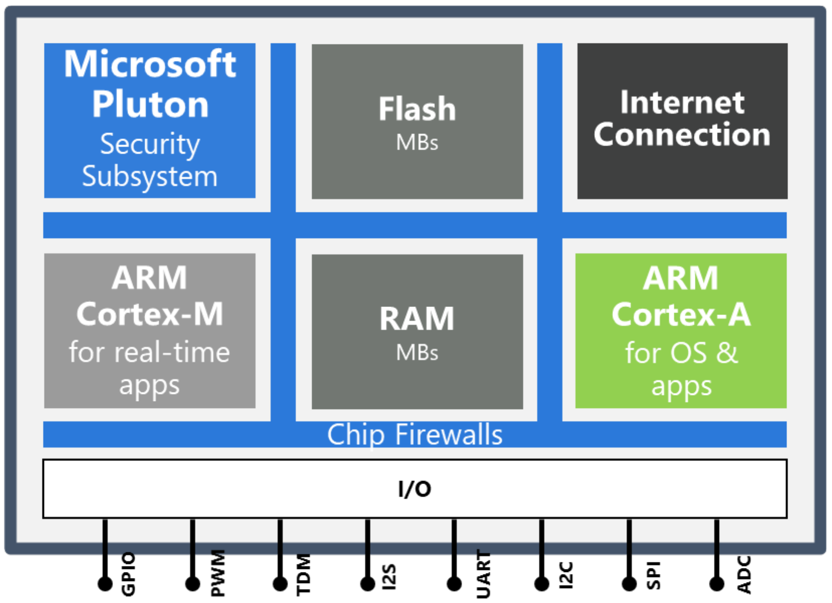


Figure 2. An example of Azure Sphere chip architecture.

In the upper-right corner of Figure 2 is the portion of the chip that puts the "I" in IoT. The MT3620 has connectivity on die, which means the Azure Sphere operating system has tight control over the code that executes on the private core that implements low-level Wi-Fi functionality. Many Azure Sphere chips also contain one or more real-time cores, where an RTOS or other "bare metal" application can execute. In this example, the real-time core is represented by a Cortex-M. A chip may have different types of real-time cores, like a RISC-V core or a specialized machine learning ASIC, to support whatever special functionality the chip requires. These cores are often used where tight timing or deterministic execution is required. Finally, the lower right-hand corner shows a Cortex-A, where the Azure Sphere operating system and applications execute. The chip also features a collection of peripheral controllers, which allow the chip to control various features (e.g., a button or a sensor) within a finished device. The MT3620 can map each of these peripherals to either the Cortex-A or to a real-time core.

Software architecture overview

With the overview of the chip architecture in mind, Figure 3 provides some detail on the software that runs within three different execution domains on the MT3620. Azure Sphere relies on ARM's TrustZone technology, which provides two independent execution environments on a single chip. The execution environments are named Secure World and Normal World. Each execution environment can run its own operating system and applications. Higher privileged software runs in Secure World; less privileged software runs in Normal World. The Azure Sphere trusted computing base, or TCB, is composed of the silicon and software that runs in Secure World. On the MT3620, the TCB is limited to the Security Monitor, which executes on the Cortex-A7, and the Pluton Runtime, which executes on a separate, private core.

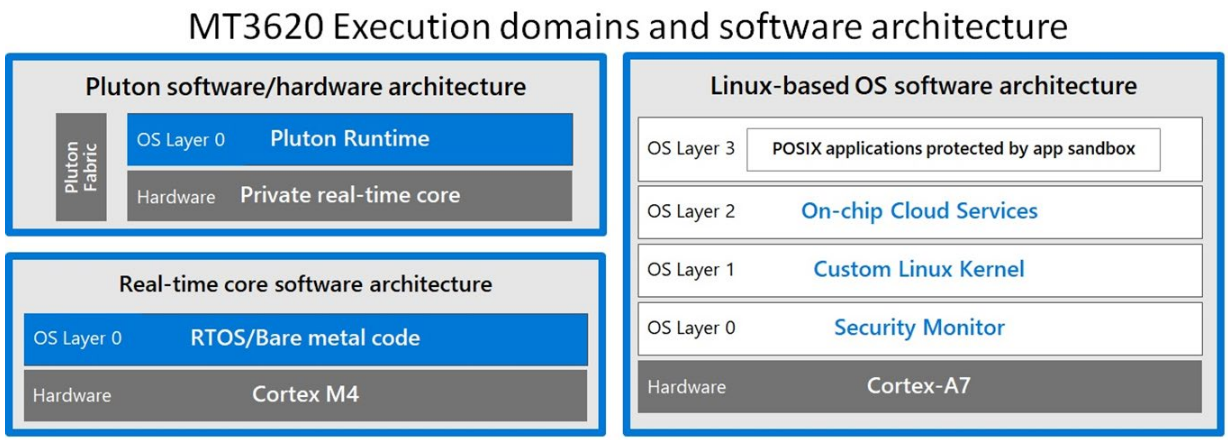


Figure 3. Azure Sphere software architecture on the MT3620.

Figure 3 shows, in clockwise order, the Pluton security subsystem, the software that runs on the Cortex-A7 core, and the software that can run on either of the two real-time cores present on the chip. The Pluton security subsystem is composed of three parts. First, the Pluton Fabric refers to a set of security features implemented directly in silicon. Second, the Pluton Runtime is a software runtime that initializes operations within the Pluton Fabric. Third, a private real-time core is dedicated to the execution of the Pluton runtime. The Pluton Fabric is where the actual execution of protocols like ECDSA signature checks, measured boot, and the signing of attestation values takes place. Only the Pluton Fabric has access to the fuses that contain public/private key pairs. When this document describes the Pluton Runtime executing an algorithm, the Pluton software runtime only *initializes* the execution of these algorithms by the Pluton Fabric, and then checks for a result. The runtime software itself is not privileged in any way, and therefore has neither special access to keys nor the ability to influence the output of the Pluton Fabric. Its only privilege is that it is the only software that can initialize the specialized functionality within the Pluton Fabric.

The right side of Figure 3 shows a high-level overview of the software architecture executing on the A7 core. The Security Monitor runs in Secure World. The Security Monitor is responsible for evaluating and granting certain access control policies, implementing the software update algorithm, and auditing. It is the only software allowed to write data to flash. Normal World is composed of the Linux kernel, a set of Microsoft authored services (on-chip cloud services) that are used to communicate with the Azure Sphere Security Service, and applications written by the device manufacturer.

Finally, the lower-left corner of Figure 3 shows the software architecture of the real-time cores. These cores are completely dedicated to customer applications; applications that run on these cores may run on bare metal or within an RTOS. Silicon features isolate the resources available to each real-time core, and all software that runs on a real-time core is run with the same privileges as a Normal World application that executes on the A7 core. Peripherals may be mapped to a real-time core to guarantee predictable timing or deterministic execution.

Other Azure Sphere chips will have a different composition of cores, features, and peripherals. The MT3620 has an additional core, not shown, that runs the Wi-Fi physical and data link layer (OSI layers 1 and 2). This core is outside the TCB and is isolated in its own execution environment. The exact composition of cores depends on the design of each chip. The Azure Sphere operating system will continue to evolve as the number of chip offerings increases.

Public key infrastructure primer: keys, certificates, and the Azure Sphere ecosystem

One of the most confusing and complex parts of any security discussion revolves around the use of symmetric and asymmetric cryptography, as well as the public key infrastructure (PKI) used to determine whether a key is legitimate and whether to trust it. We begin with a brief primer on technologies and then provide an overview of how those technologies are used within Azure Sphere. This discussion is not meant to be a complete tutorial. If terms like public/private key pairs, symmetric encryption, or asymmetric encryption are not familiar, please look to other sources that provide better descriptions than those found in this document. However, Azure Sphere's unique usage of some of these technologies means it is worth reviewing a few terms and use cases.

Certificates, often using the X.509 standard, are the de facto standard for confirming the legitimacy of public keys on the internet. For the purposes of this discussion we will ignore protocols like Secure Shell (SSH) in favor of discussing Transport Level Security (TLS), which is the protocol currently used by Azure Sphere devices and is the underlying protocol for https. Public keys are part of a *public/private* key pair that is used to establish a private, confidential connection between two computers. Often this occurs over a TLS connection. The existence of a public key allows someone to prove that another party holds the public key's associated *private key*. Certificates depend on placing trust in a *root* certificate issued by a trusted party (e.g., Verisign). Other certificates are derived from this via a *certificate chain,* which allows anyone to distribute their own public key within a certificate. The certificate issuer, in turn, signs the certificate using its own private key. When trusted certificates are available on a PC, they are used to confirm that a certificate from a company like Microsoft was issued by a trusted party and therefore contains the legitimate public key used for communication with that company.

A small number of trusted certificate issuers maintain the infrastructure to issue certificates only to legitimate parties. This PKI is only as secure as the certificate issuer. A compromised, or rogue, certificate issuer can issue malicious public keys in anyone's name. A PC comes installed with a set of trusted root certificates, which allows certificates that are derived from those roots to be verified.

For example, when a web browser connects via TLS to https://www.microsoft.com, it uses a certificate chain that is signed by a third party to confirm that the public key used to negotiate the TLS connection to Microsoft is actually Microsoft's key. This effectively allows a PC to verify the identity of a remote server. For the remainder of this document, this process will be called *server authentication,* because it is used to prove that the remote server is authentic and not an imposter.

Device authentication reverses the roles. A remote server wants to determine whether a device's identity is genuine. It is possible to use a certificate that the device provides to prove the device's identity. However, the certificate must be placed on the device when it is manufactured, and the certificate's private key must be secured. Both bootstrap problems are difficult with IoT-based devices because device security is typically poor.

Device authentication is not used when your PC connects to the internet or to a secure site on it (like a bank). Instead, remote servers verify the identity of the person using the computer via credentials, tokens, keys, or a similar mechanism. IoT devices may not have interactive users, but the devices still need authentication and an unforgeable identity. A fake device might acquire or steal sensitive data if the fake device successfully connects to a remote server.

The problem is that devices need a scalable authentication mechanism that does not rely on a single secret for a population of devices. Usernames and passwords are used outside of IoT, but any *hardcoded* information on a device could be stolen by an attacker and then used to pretend to be another device. Because this constitutes a major security risk, a single, hardcoded secret for all devices is an approach to avoid. Having a single password is an example of a "break one, break all" policy where compromising a single device compromises the entire ecosystem of devices. Alternatively, requiring that all devices have a device certificate that is presented to a server is a sound approach. However, the device must store private keys securely in a hardware root of trust, so they are not easily stolen. If every private key is unique, compromising a single device does not compromise the entire ecosystem of devices. In a traditional PKI model, this requires issuing billions of long-lived device certificates. The result is that renewal and revocation are problematic. If a device is offline, missing a renewal window could mean the device cannot connect to online services. Further, if billions of devices use long-lived certificates, some entity must track whether any of them are compromised so that their certificates can be revoked. If a large-scale attack occurred, the Certificate Revocation Lists could be larger than the amount of memory any IoT device has available. These are thorny problems, and Azure Sphere implements a scalable solution by using Measured Boot and issuing very short-lived certificates to eliminate the renewal and revocation problems typically associated with certificate-based PKI.

Mutual authentication refers to establishing a private, confidential connection where both the remote server and the device are authenticated via an exchange of keys whose identities are verified via (in the case of TLS) X.509 certificates.

Besides keys and certificates, it is worthwhile to review a couple of terms used throughout the remainder of this document. *Secure Boot* refers to the process of using digital signature verification to prove that a piece of signed software is legitimate before it is run during the initial chip boot process. That is, is a piece of software legitimately signed by Microsoft, or is it an imposter? Like certificate authentication, digital signature verification also involves keys, but instead of establishing a confidential communication channel, they are used to verify that software was distributed by a trusted party. In this case, Microsoft uses a private key to sign software, and a device uses the associated public key to prove that software was really signed using the associated private key. *Distribution* of these keys is critical to keeping malware out of an ecosystem and it is common to use certificates as part of key distribution. Azure Sphere's approach in this case avoids the use of certificates, and is discussed in the subsection titled *Best practice #4: use secure boot everywhere and always*, which describes the use of Secure Boot. Azure Sphere uses Secure Boot to guarantee that all software executed on the device is genuine.

Measured Boot is part of the remote attestation protocol, which proves to a remote server that Secure Boot completed successfully. In other words, remote attestation allows a remote server to verify which software was used to boot a device. The remote server then implements policy decisions based on the software used by the device. For example, the remote server might realize that Secure Boot was hacked if the device is running unknown software. The device could then be denied a connection to critical resources. Alternatively, the remote server might realize the device is running legitimate but out-of-date software that is no longer trusted. This would happen if a zero-day exploit were found in a release that was so dangerous that Microsoft does not trust any chip running that release. Therefore, although the release is legitimate, it is not trusted. The device might then be forced to update to newer software before being allowed to connect to critical resources.

Azure Sphere uses all these technologies. Table 1 provides a summary, which will help to clarify the discussion in the remainder of the document.

Technology	Purpose	Protocol	Notes
Server authentication	Verify remote server identity.	TLS	Verifies Microsoft's identity. Certificate chain is put on chips during manufacturing.
Secure Boot	Verify software executed is genuine.	ECDSA using ECC public keys on device	Uses chain of trust. First public key burned into fuses on device.
Measured Boot/ Remote Attestation	Proves to the Azure Sphere Security Service that the chip is genuine and running trusted software.	Custom remote attestation protocol	Depends on ECC public/private key pair generated within Pluton and burned into fuses. Only private key on-device used by Azure Sphere.
Device authentication	Proves to any service on the internet that the Azure Sphere device completed attestation successfully.	TLS	Generates a special, short-lived device certificate, via remote attestation, which is used for TLS device authentication.

Table 1. Summary of different cryptographic protocols used by Azure Sphere.

With this background in place, the remainder of the paper discusses some best practices that Microsoft uses to implement Azure Sphere.

Best practices for implementing the seven properties

The remainder of the paper collects the best practices used by Microsoft, based on decades of experience implementing secured products, to implement Azure Sphere. We hope that these best practices provide insight into why Azure Sphere sets such a high standard for security and that they lead to questions or guidelines that can be used to evaluate other IoT security products.

Best practice #1: treat boot ROM as non-updatable software and minimize its size

ROM is the first code that executes when a chip is powered on. While ROM is code, it is unique because it is part of the chip when it is manufactured. Therefore, it might be tempting to assume that ROM is somehow more secure than software loaded from durable storage and thus to treat it as “hardware.” This assumption that ROM is hardware stems from the fact that it is not possible to substitute “bad” ROM into the chip. Only the code that was etched into silicon (usually a metal layer) at manufacturing time will begin executing. Someone cannot substitute other ROM, or a hacked ROM, at boot. However, it is difficult to guarantee that an attacker will not skip portions of the ROM and only run ROM code that assists an attack. Although the code resides in silicon, it is executed by an MCU or CPU just like any other software.

One common class of attack glitches the CPU to skip some instructions. Attackers do this to try to skip critical ROM code, like software signature checks, so that they can load arbitrary code. Loading arbitrary code allows an attacker to take control of the MCU and use it for any purpose. If the MCU does not have countermeasures to protect against these types of attacks, any assumptions around ROM security are invalid. Therefore, two important outcomes stem from this analysis. First, build countermeasures to make it more difficult to successfully skip critical code paths in ROM. Second, put as little functionality as possible into ROM.

Both of these best practices were followed in implementing the MT3620. Azure Sphere and MediaTek worked to minimize the amount of ROM code while also implementing basic countermeasures to make it harder for an attacker to glitch the CPU and execute arbitrary code.

To summarize this first best practice: Treat ROM like software that is difficult to update. Build countermeasures into the chip itself to make it more difficult for an attacker to skip critical sections of code and ensure that all code that is not ROM is updatable.

Best practice #2: never expose private device keys to software

Azure Sphere chips use elliptic-curve cryptography (ECC) public/private key pairs to implement both Measured Boot and Secure Boot. Measured Boot stores its private portion of the public/private key pair in write-once storage, which is called a one-time programmable (OTP) fuse or sometimes by the specific technology used, like "e-fuse[7]." Writing an e-fuse is often called "blowing" or "burning" a fuse, since it is a one-way operation that cannot be undone without making the entire value unreadable. The fuses that store the public and private keys are readable only by the Pluton Fabric during an operation that uses them. Software, even the most trusted software in the Pluton Runtime, cannot read the fuse bank or gain access to a private key stored in fuses.

The Measured Boot key pairs are generated when the chip is first manufactured, and the public portion of each key pair is collected by Microsoft at the same time. This eliminates the need for other PKI, like a certificate, to prove that a public key is legitimate since the key is collected directly from the chip long before the chip is put into a device. Therefore, rather than trusting a third party via certificate generation, Microsoft instead trusts the public key collection mechanisms used in a secure facility when the chip is manufactured. The presence of that public key is critical to knowing whether a chip that claims to be an Azure Sphere chip really exists. Therefore, Azure Sphere's key generation mechanism must secure and trust the public key collection process to make sure that an attacker cannot inject illegitimate public keys into it. Such keys would allow counterfeit chips into the marketplace. Microsoft and its silicon partners work carefully together to secure and protect this collection process.

The Pluton security subsystem goes beyond storing keys securely; it also securely generates each Measured Boot key pair in hardware during the chip manufacturing process. Often, secure elements, or a hardware root of trust, securely store keys. However, those keys are generated elsewhere. One common and accepted scenario is to use a hardware security module (HSM) to generate the key pair and then transfer that key pair for storage on a device. The HSM is designed to be tamper resistant and is often stored in a facility that is secured against physical tampering. However, using an HSM means the key must travel from the HSM to the chip. Transferring a key electronically requires securing the key transfer operation against man-in-the-middle attacks and ensuring that the keys are not leaked or copied by any software involved in the process of burning them to fuses within the chip. Therefore, depending on the threat model used, using an HSM to generate keys and transfer them to a chip may be less secure than generating the keys internal to the chip itself. The private portion of an Azure Sphere key pair is never exposed to software during the lifetime of the chip. This practice removes another attack surface that could otherwise be used against a supply chain.

[7] We note that other technologies, like anti-fuses, also exist. For the purposes of this discussion, we elide the distinction and simply use the term e-fuse to refer to all write-once fuse technology on a chip.

Best practice #2 is all about PKI—how keys are generated, stored, and managed. Azure Sphere eliminates the risk of exposing any private device-specific keys to software by generating them internally, in silicon, during the manufacturing phase of the chip. The tradeoff is that the public key collection process must be secured against tampering.

Best practice #3: in IoT, choose ECC, not RSA, for device-specific keys

The two most common public/private encryption algorithms are ECC and RSA. Azure Sphere chose ECC keys for its internal device keys for two reasons. First, ECC keys of equivalent resistance to cracking through brute force are significantly shorter than RSA keys, which is important when working with resource-constrained devices like those in IoT. Larger keys require more RAM and larger e-fuses to store them. The result is a more expensive chip, and thus a more expensive device. Smaller keys lessen these overheads without sacrificing encryption strength. We expect that ECC will become more common as IoT devices look to bolster strength while minimizing cost.

Second, generating an ECC key pair requires only that a source of randomness, or entropy, be available. The Pluton Fabric contains a true hardware random number generator to generate the random numbers required for ECC key generation. The random number generator monitors its own output, and if the output deviates from an expected model of random output—either due to a malfunction or an attacker influencing the generation of random numbers—the random number generator shuts itself down.

Self-monitoring protects the integrity of key generation. Besides longer key length, RSA key pair generation requires the creation of two large prime numbers; otherwise the keys may be vulnerable to attack. Generating and checking for prime numbers is computationally expensive, which would require Pluton to be larger and therefore more expensive to manufacture to implement correctly. Alternatively, it would require sacrificing internal, silicon-driven key generation. Selecting ECC makes Pluton smaller in terms of die area, and it allows the Pluton Fabric to generate its own Measured Boot keys.

These steps allow Azure Sphere customers, silicon partners, and device manufacturers to trust less of their supply chain and its infrastructure. Customers do not need to reinvent these security features for their own IoT products; they come as part of the package for purchasing an Azure Sphere chip and a license to its OS and its Azure-based security service.

Best practice #3, in summary: stronger security does not always equate to higher costs. Azure Sphere approached the choice of asymmetric encryption algorithms based on the expected resources in IoT chips. This exploration of the design space lowered costs by limiting chip size without sacrificing security.

Best practice #4: use Secure Boot everywhere and always

Secure Boot ensures that software is executed during the OS boot process only if it comes from a trusted provider. Every executable binary that runs on an Azure Sphere device is signed by Microsoft. When boot begins, the ROM verifies the signature of the bootloader, which is the first piece of software loaded from flash. Once the bootloader is verified, it begins execution. The bootloader is then responsible for verifying the next piece of software to run, which on the MT3620 is the Pluton Runtime. Once its signature is verified, the Pluton Runtime begins execution, and it verifies the next piece of software. This sequential execution builds a *chain of trust* during boot, with each legitimate piece of software ensuring the next piece of software to execute is also legitimate. Azure Sphere uses Secure Boot on every piece of software that runs on the device, from the bootloader to third-party applications. Unsigned software never executes on the device. This is one of the primary protections against supply chain attacks that load malware on a device before it is delivered to the field.

During Secure Boot, Azure Sphere again uses ECC keys, via a digital signature algorithm called ECDSA, but with a different key pair than those generated by the device. Instead, Secure Boot relies on a set of public/private key pairs generated by Microsoft. The private keys are stored securely within Microsoft's own secure key storage facilities—the same facilities that guard the private signing keys that sign Microsoft's own products like Microsoft Windows. Securing these private keys is critically important. If an attacker ever acquired a private signing key, malware could be distributed as trusted software, eliminating the benefits of Secure Boot. Likewise, the public keys used to verify digital signatures must also be trusted. If a device has the wrong public keys on it, attackers could sign their own binaries and distribute them, and the device would trust those binaries instead of binaries distributed by Microsoft. Therefore, Azure Sphere needs infrastructure to ensure that only the right set of public keys is trusted by the device and used for Secure Boot.

At the same point that the chip generates its own ECC public/private key pairs, Microsoft also burns a single, additional public ECC key to Pluton's fuses. This key is used for signature verification, and because it is written into fuses it is the only key that cannot be revoked. Consider this the "bootstrap" key, which is used by the ROM to verify the bootloader. It is burned into e-fuses so that the ROM does not need to trust anything outside the Pluton Fabric to execute signature verification. This public key is the lynchpin for PKI in Azure Sphere. Its private counterpart is therefore not only stored securely with other signing keys, but the number of employees who can access that key within Microsoft is extremely limited, and the accounts used to access that key are restricted. These additional steps limit the possibility that an attacker could gain access to that key.

The bootstrap key is used in only two cases. First, it is used to verify the signature of the bootloader by the ROM. Second, it is used to verify a binary blob of public keys and certificate chains. The public keys are then loaded and used by Pluton to verify the remaining binaries that run on the chip. Different public keys are used to verify different types of software on the device, and since those keys are stored in a binary blob, they can be updated at some future date. The certificates are neither parsed nor used within the TCB but instead are used later during TLS negotiation to connect securely to Microsoft for the purposes of software update and remote attestation.

To implement best practice #4, make sure that every piece of software in the field is signed, and that those signatures are verified. Further, develop processes that guarantee that public keys used in secure boot cannot be forged, introduced later in the manufacturing process, or require complex certificate parsing to verify.

Best practice #5: use silicon-based Measured Boot to attest Secure Boot

Measured Boot is used to prove to a remote server that Secure Boot was successful by providing a signed list of the software used during boot. *Remote attestation* integrates Measured Boot into a client/server protocol. The Measured Boot algorithm is straightforward, and the Pluton Fabric provides a hardware implementation that provides additional defense-in-depth security.

Measured Boot in the Pluton Fabric uses two hardware registers: a digest accumulation register[8] and a nonce register. During Secure Boot, a hash value of each loaded binary is appended[9] into the digest accumulation register, which changes its value. Importantly, this register cannot be reset, and measured values cannot be reversed, without resetting the entire chip. Therefore, even if an attacker gained control of the Pluton Runtime, it could not forge a new attestation value. This is another example of the value of a hardware root of trust with silicon-based attestation.

When remote attestation takes place, an Azure Sphere chip connects, via TLS, to the Azure Sphere Security Service. The Azure Sphere Security Service responds with a nonce and asks that Azure Sphere provide its attestation value, along with a set of "measurements" used to generate

[8] In a Trusted Platform Module (TPM), this is referred to as a Platform Configuration Register (PCR).
[9] In this case, *append* means cryptographically merged using a one-way hash function.

the Measured Boot value. The Pluton Fabric combines the nonce with the digest accumulation register, and signs that value with the private ECC attestation key stored in e-fuses. Other measurements are also signed, and this bundle of values is sent back to the Azure Sphere Security Service.

The private ECC attestation key is used only for attestation. Hardware disallows any other use, thus preventing an attacker from trying to sign arbitrary payloads that match an expected attestation value. Furthermore, only the Pluton Fabric uses the key. No software ever has access to this key, and the algorithm is invoked only by an API call to the Pluton Fabric. The Azure Sphere Security Service already has the public ECC attestation key, which was collected when the chip was manufactured. The service therefore can verify that the device is authentic, and, by using the nonce, that the value of the attestation register is current (i.e., that it was not an old value replayed by an attacker). The server reproduces this value by using a software implementation of the attestation algorithm.

Executing the remote attestation protocol results in three possible outcomes: a successful attestation, an attestation that requires a software update before allowing the device to connect to other online services, and a failed attestation attempt.

First, if attestation completes successfully and the device is running known and trusted software, then the device receives two X.509 certificates. The first certificate, called the update certificate, is used to connect to the Azure Sphere update service to acquire and install new software and to upload error reports. The second certificate, called the customer certificate, allows the device to connect to the customer's own Azure-based services (e.g., Azure IoT Hub) or to other third-party services using TLS. Customers acquire their own public certificate chains from Azure Sphere and then use those certificate chains to allow mutual authentication to complete correctly with any service that supports TLS-based authentication. Each certificate contains the identity of the Azure Sphere chip embedded within it, as well as the public portion of a public/private key pair generated by the Azure Sphere chip during attestation. This public key and the certificate that encapsulates it are used by the Azure Sphere Security Service and by other services to verify a device's identity through TLS-based mutual authentication. This key pair is thrown away when the certificate acquired during attestation expires.

The update and customer certificates are valid for only 24 hours, which eliminates the need to manage the revocation of either certificate since they expire quickly. Further, the private key generated by the chip to create these certificates is kept only in secured SRAM and therefore attestation must be completed either after 24 hours or after the chip reboots. These two certificates, which are issued as a result of attestation, are used to prove that a chip successfully completed attestation within the last 24 hours. The certificate infrastructure is a stand-in to allow any service to confirm that attestation recently completed, without requiring each service to implement attestation verification itself.

Second, it is possible that the attestation value is valid, but the software is not trusted. This is a critical distinction: a correctly signed binary means the binary is authentic, but it does not necessarily mean the binary is trusted. Consider a Linux kernel binary that is signed by Microsoft but is discovered to contain a critical, zero-day exploit. Microsoft can “stop trusting” this binary during the attestation process and ensure that any device that attempts to complete attestation will fail. Azure Sphere uses this distinction to mark binaries as untrusted. Microsoft then tells the device to update by issuing the update certificate, but it withholds the ability for a device to connect to a customer’s web service by withholding the customer certificate until the update completes and the device successfully completes attestation with trusted software. Note that being out of date does not prevent attestation from completing successfully. Only if a device is out of date and running a binary that Microsoft deems untrusted is the device forced to update before being allowed to connect to a customer’s infrastructure.

Finally, attestation may fail altogether if the device sends an attestation value that is simply invalid or whose signature is incorrect. These devices fail attestation, and do not receive any certificate, thus preventing them from connecting to either Azure Sphere’s or the customer’s own web services.

In conclusion, best practice #5 is about the nitty gritty details of device attestation, authentication, and health. Azure Sphere uses attestation to generate an authentication token using the X.509 certificate format, which allows any service that executes TLS-based mutual authentication to verify that Microsoft is vouching for the health of a device. In other words, rather than require all customers to reimplement attestation correctly, TLS and an X.509 certificate act as a proxy to prove that attestation succeeded. Finally, limiting the lifetime of a certificate to 24 hours eliminates the need for certificate revocation.

Best practice #6: do not use (or parse) certificates in the trusted computing base

In reading the last three sections, you might notice the absence of certificates when discussing Measured Boot and Secure Boot key provisioning and storage. Their absence is no accident. Parsing certificates has been the source of numerous security vulnerabilities[10]. In general, code that parses an input value must be written and tested carefully, as parsing can often lead to vulnerabilities that allow arbitrary code execution. Some well-known examples of exploits that bypassed Secure Boot mechanisms rely on certificate parsing[11]. As a format becomes more complex, parsing becomes more complex and therefore the code is more likely to contain bugs.

[10] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ASN.1+parsing contains one list. There are many others.
[11] https://blog.quarkslab.com/vulnerabilities-in-high-assurance-boot-of-nxp-imx-microprocessors.html

The X.509 format is complicated and was historically a source of vulnerabilities across platforms. Azure Sphere adopted a strategy that eliminates the need to parse certificates to implement Secure Boot or Measured Boot. In turn, certificates are not used within the trusted computing base (TCB), which removes a common attack surface from the TCB.

Access to two different types of keys is required within the TCB. First, public ECC keys are used during Secure Boot. Second, a private ECC key is used during the completion of the remote attestation protocol. Earlier, we discussed how the keys used in Measured Boot are generated within the Pluton Fabric and are collected early in the manufacturing process. Likewise, keys used during Secure Boot also are incorporated during the manufacturing process to avoid the need to represent those keys in certificates. Therefore, certificates are never parsed during boot—or at any other time—within the TCB to verify keys, as is common on some other platforms. This eliminates certificate parsing from the TCB.

What about situations, such as setting up a TLS connection, that require certificate parsing? Azure Sphere licenses and uses wolfSSL, an open source, proven TLS library, for its TLS connections. Certificate parsing is limited to user-mode (i.e., application) code. Any bug within that code does not immediately put the TCB at risk. User-mode code is the least trusted, and therefore the least capable, code that executes on the chip, thus limiting the damage from an attack that takes advantage of a vulnerability in certificate parsing.

To summarize best practice #6: by generating keys on device and collecting those keys during the chip manufacturing process, there is no need for additional PKI to prove those keys are authentic. Therefore, certificates are not used within the TCB, which eliminates an additional attack surface from the TCB.

Best practice #7: handle server certificate expiration gracefully

Server authentication requires that a device use a trusted certificate to verify the identity of a remote server. Azure Sphere puts a small number of certificate chains into a binary blob whose authenticity is verified during the secure boot process. These certificates are used to verify Microsoft's identity when connecting to perform attestation or when downloading software updates. However, certificates expire, so what happens if one of Microsoft's certificates is no longer valid? An expired server certificate used to authenticate Microsoft's identity prevents the device from connecting to Microsoft to complete attestation. Failing attestation prevents the device from connecting securely to customer cloud services like Azure IoT. Therefore, connecting to the Azure Sphere Security Service using TLS to execute remote attestation is the *second* step taken by any Azure Sphere device when it comes online. The first step manages server certificate expiration.

Before diving into the details, it is worth noting how this problem is solved in other environments. Home PCs receive new certificates through software updates, and enterprise environments have IT teams that update certificates on devices as certificate expiration approaches. IoT presents unique challenges since a device may sit in inventory for months or years or be disconnected for weeks or months at a time. Therefore, it is important that a device can automatically update its certificates when it comes online and discovers that its server certificates are expired.

Azure Sphere manages this scenario with a step that might surprise the reader: the first step in connecting to the Azure Sphere Security Service is to connect to a service managed by an insecure http, and not https, endpoint. This endpoint is, of course, vulnerable to man-in-the-middle attacks where an imposter pretends to be the intended server. The Azure Sphere device connects to this endpoint with the full knowledge that the endpoint might not be authentic. This endpoint hosts the most up-to-date version of the binary blob, which contains signing keys and critical certificate chains. The blob is signed with the bootstrap key, whose public counterpart is burned into fuses on each chip. When an Azure Sphere device first connects to the internet, it checks whether this blob differs from what it is currently using. If so, it downloads the blob, and, if the signature is correct, it updates its store of keys and/or certificates and then reboots. It is possible that an attacker could replay a valid but older binary blob. However, this would only serve to provide an older set of possibly expired certificates.

Since these binary blobs are rarely updated, this additional step normally involves only a few messages to verify that the current blob of keys and certificates is up to date. This "bootstrap" step means that devices can be offline for an arbitrary amount of time and still renew critical certificates to connect to the Azure Sphere Security Service without human intervention.

Best practice #7 in summary: IoT devices will operate in environments where connectivity is spotty or rare, and those devices may sit in warehouses for months or years without connectivity. When a device comes online, it must always be able to securely connect to online services, which means the device must have a means of automatically renewing server certificates when they expire. Otherwise, a device manufacturer may be forced to issue a recall or send a technician to service each device.

Best practice #8: connectivity is optional

The discussion around certificates, connectivity, Secure Boot, and Measured Boot often leads to a logical question: what happens when a device is disconnected from the internet and its update and customer certificates expire? The device continues to operate just as it had before. Remember that Secure Boot does not use certificates. Therefore, a device that is disconnected from the internet boots and runs its applications whether it has attestation-derived certificates. If its certificates are expired, it can connect to the Azure Sphere Security Service when connectivity is once again available. This ability to operate while disconnected for an arbitrary length of time is critical to success in many IoT scenarios. For example, battery-operated devices, or devices that are in a mobile environment, cannot depend on continuous connectivity. Further, if a home device, such as an appliance, suddenly stopped working because a Wi-Fi router failed, end users would be unhappy!

Best practice #8 in summary: do not require connectivity to operate a device. Instead, require connectivity only for features or functions that absolutely require it.

Best practice #9: make it harder to build botnets out of zero-day vulnerabilities

What happens if an internet worm or malware manages to compromise an application? How does that malware turn the device into a botnet capable of a coordinated distributed denial of service attack? As discussed earlier when describing the Mirai botnet, malware reaches out and connects to a *command and control* server to ask for instructions. Then the device may be issued commands as part of a larger group of compromised devices that form a botnet. These instructions might ask the device to download a payload of additional malware, to attack an IP address or web address, or execute some code like bitcoin mining or a myriad of other scenarios.

Azure Sphere counters this risk through a firewall that is programmed when an application is *compiled*, rather than when it is run. Each Azure Sphere application contains a manifest in which the application developer sets access control policies for local resources, peripherals, and outside IP addresses. The manifest is included in the signed application package that is verified before the application runs. Among other things, the manifest is used to program the device's network firewall. By default, it denies all outgoing and incoming connection attempts. This differs from a firewall on a typical PC, which denies all incoming connections and allows all outgoing connections. If an Azure Sphere application is compromised, there is no way for an attacker to reprogram the firewall to contact any host that is not already programmed into the device. This feature is key to protecting the Azure Sphere ecosystem; we expect that applications will be compromised, and this "static" firewall is critically important to protecting devices. This firewall approach is not only used by customer applications. Each Microsoft-authored application or background service also specifies its allowed endpoints during application development. Microsoft-authored applications follow the same procedures and live under the same restrictions as third-party applications.

This leads to a practical question: how does a device connect to a service where the address is not known *a priori*? Azure Sphere supports two protocols to allow dynamic discovery of services at runtime: mDNS and DNS-SD. More detail about these features is available in the Azure Sphere documentation[12].

Best practice #9 in summary: specify required network firewall addresses during application development and prevent an application from modifying that firewall at runtime. This approach limits the ability of an attacker to add a device to a botnet when an attacker discovers an exploit that compromises an application.

Best practice #10: use a policy of "deny by default" and enforce it in silicon

Azure Sphere chips like the MT3620 ground resource access control mechanisms in silicon. Every resource that is accessible from software is capable of silicon-based lockdown. Azure Sphere takes advantage of ARM's TrustZone technology (discussed on page 7) to implement these access control policies. As an example, the 4 megabytes of SRAM attached to the Cortex-A7 is divided into segments that are accessible from Secure World and segments that are accessible from Normal World. The Normal World memory can be further subdivided into memory accessible from the Cortex-A7 and memory accessible from one of the real-time cores. This additional sharing is used to facilitate inter-core data transfer and communication. Azure Sphere resource access policies are described both by which core is requesting a resource, and whether code executes on that core in Secure World or Normal World.

Azure Sphere follows a "deny by default" approach to resource acquisition. Consider the example of access to SRAM. When the chip first boots, SRAM is only accessible to Secure World. To make SRAM available to code executing on the A7 in Normal World (e.g., the Linux kernel), an associated access control policy change must be executed by software running in Secure World. In contrast, in a common strategy, everything is "open" by default, and software must choose which resources to lock down. Deny by default is less error-prone since software will not run if a required resource is unavailable. However, proving that access to a resource is properly disabled is far more difficult.

[12] https://docs.microsoft.com/ azure-sphere/app-development/service-discovery

A second example of this policy is the mapping of peripherals to cores, which we call a “chip firewall” or a “core mapper.” The core mapper is responsible for allowing access to a peripheral from a core or in some cases from Normal World versus Secure World. By default, no peripherals are available on any core, and each peripheral must be explicitly mapped to a core. Application developers specify which peripherals they require in the same application manifest that configures the network firewall. This manifest is then included as part of the signed binary loaded onto an Azure Sphere chip.

Finally, the core mapper contains a “sticky” bit. Once the bit is set, the core mapper cannot be modified until the entire chip is reset and all software reboots. Making settings sticky means that even if an attacker compromised the code that maps peripherals to cores, an attacker would not be able to remap a critical peripheral to another core. Making settings sticky is another example of a deep defense-in-depth measure provided by Azure Sphere chips.

Best practice #10 in summary: a policy of deny by default makes it far harder to reconfigure access to critical resources by a compromised or buggy application. Azure Sphere uses application manifests to allow access only to those resources specified by an application and allowed by operating system policy. Deny by default ensures that no other resources can be accessed unless explicit action is taken.

Best practice #11: eliminate the concept of users on IoT devices

IoT devices do not have users and they therefore do not need user accounts. The data they generate might be accessed by a user. The cloud services to which they upload data may be managed by a user. The device itself, however, should not have a concept of users once it is deployed to the field. One of the defining features of the Mirai botnet was the existence of user accounts on the device with hardcoded, permanent, default passwords. Once a device is deployed to the field, including the concept of a user (or a super user) account on the device simply opens an attack surface for a malicious user to exploit. Further, including users and passwords means they must be changed, reset, and managed.

The Azure Sphere operating system does not have user accounts, logins, nor their associated passwords. This attack surface is removed from the operating system, eliminating the need to determine how to secure it. All IoT devices should remove these attack surfaces.

Taking this step requires changing portions of Linux, because the process of executing applications on behalf of a user (i.e., a user account that logged into the operating system) is deeply ingrained into the operating system. Two specific best practices were followed in designing access control policies for the Azure Sphere operating system[13]. First, application identities (Linux user and group designations) are assigned at runtime and are unique to each application. Unlike a desktop environment, where an application executes with the privileges of the user that invokes it, Azure Sphere applications execute with their own identity that has its own privileges assigned based on the type of work it does, its required privilege level (e.g., an application versus a system service), and its unique application ID.

Azure Sphere depends on existing Linux access control mechanisms combined with extensions in a custom Linux Security Module (LSM) to enforce its access control policies, and they apply across resource acquisition in the operating system. As an example, Azure Sphere applications are contained within their own application package, which has an internal file system. The dynamic assignment of a unique user/group ID per application means that the application automatically has permission to access the files in its own application package, but no inherent permission to access the files in another application package, or in data written to mutable storage by any other application. Using unique user/group IDs is a key part of locking down applications into an application "sandbox" that limits the actions each of application on the operating system.

When an application on a typical Linux desktop temporarily takes an action that is privileged, it is granted escalated privileges that are normally reserved for a "super user account" or "root" identity that has greater control and broader permissions within the operating system. The problem is that an attacker can exploit privilege escalation to change its identity to the super user account and take control of the system. Azure Sphere eliminates the ability for any application to change its identity (the associated system calls fail), even when an application is granted increased privilege to accomplish some task. These policies are enforced by the Azure Sphere LSM.

If users are eliminated and access control is strictly limited, how do software developers debug or examine their code? The answer lies not in user accounts, but in device capabilities secured by the Pluton hardware root of trust. When customers (or Microsoft's own internal Azure Sphere developers) need to run code during the software development process (referred to as "test-signed" code), or when they need to debug code or access resources that are normally restricted, they are issued a "device capability" that allows access to special privileges. These capabilities are issued by the Azure Sphere Security Service, to which Microsoft strictly controls access. Stealing a capability from a development kit and putting it on another Azure Sphere-based device does no good, because each capability is tied to a specific chip and will not work on any other Azure Sphere chip. Using device-specific capabilities enables some flexibility during the software development process without granting privileges that span devices or adding the need for user accounts.

[13] Please see this video for an in-depth treatment: https://www.youtube.com/watch?v=ISU4-yG68x0

Best practice #11 in summary: IoT devices should not contain user accounts, login prompts, or user identities that are used to invoke applications. Further, these devices should not have shells, logins, or support for remote, interactive computing through a protocol like SSH. Eliminating these concepts from a device eliminates a broad set of attack surfaces. Eliminating an attack surface eliminates the need to secure it.

Best practice #12: physically separate real-time execution from internet communication

Azure Sphere chips contain two different types of cores. Applications that execute on a Cortex-A core execute in a familiar POSIX environment on top of the Linux kernel. These applications connect to remote services and have access to on-chip peripherals. However, the applications have no guarantees on the latency of execution. Other services and applications execute in parallel, and critical operations like software update will also take time away from application execution.

The other type of core is a real-time core, which is dedicated to customer software. No other software competes with it for cycles on the real-time core. The physical separation between cores also means that timing-sensitive code can execute on the real-time core without exposing that code to the unpredictable timing of internet communication. The real-time core can send and receive messages to and from applications executing on the A-core, but it may do so at its convenience during its execution cycle. Applications that run on the real-time core can implement algorithms that require deterministic timing to an external peripheral.

Best practice #12 revolves around separation of concerns. Separating execution domains into different physical cores is the best way to guarantee that one core cannot interfere with another.

Best practice #13: divide code into user-mode code and kernel code

One common practice in existing MCUs is to use an RTOS, which is a high-performance library that provides functionality like scheduling or peripheral access. For MCUs not connected to the internet, an RTOS provides a way to accelerate time to market by eliminating the need to reimplement common functionality. However, an RTOS is often executed at the same privilege level as the application itself. Therefore, if an attacker exploits a vulnerability in an RTOS, there is little standing in the way of compromising all functionality in the system.

In a modern operating system like Windows or Linux, execution is divided between a *kernel* and *applications*. Each application has the illusion of a private, dedicated execution environment. The kernel multiplexes these applications across hardware. The kernel/application boundary also adds additional defense-in-depth measures. The kernel usually has more privileges and can grant or deny an application access to system resources. If an application is compromised, all is not lost: an attacker must also compromise the kernel to gain complete control of the computer.

Azure Sphere adds additional defense in depth measures by putting some functionality in the Security Monitor, the Pluton Runtime, and the Pluton Fabric. Other operating systems add additional defense-in-depth measures by running software in hypervisors. Each of these system architectures adds new defense-in-depth layers, increasing the difficulty for an attacker to completely compromise any system.

Therefore, best practice #13 is all about modern operating system design and the separation of work between an operating system kernel and applications. Azure Sphere uses a Cortex-A for its Linux-kernel-based operating system because it supports the required underlying functionality to have virtualized address spaces, an isolated kernel, and hardware isolated applications.

Best practice #14: ensure all software is updatable

In "The seven properties of highly secured devices," the final property is renewable security. The heart of this property is the ability to remotely update the software on an IoT device. This property is familiar to everyone who uses a PC or a smart phone. The ability to update a device remotely is the key to fixing stability bugs, responding to newly discovered exploits, and introducing new features to market long after a device is purchased. However, this familiar experience is the exception, and not the rule, in IoT. Setting up the infrastructure to update millions, or billions, of devices worldwide requires a massive foundational investment. Further, writing a software update implementation on a device is not a simple endeavor. The next three best practices pertain to the device-side implementation of software update.

First, every piece of software must be updatable. On the surface, this seems obvious, but the lowest level code (i.e., a first-stage bootloader) may be structured in a way that makes it difficult to update. Software that is difficult to update is an obvious target for attacks and exploits. If the software cannot be updated easily, any discovered exploit is much more likely to be abused again and again before the software is fixed. In Azure Sphere, every piece of software, including the bootloader, can be updated remotely. Therefore, Microsoft retains the ability to patch software throughout the software stack.

Best practice #14 in summary: when building your IoT device, make sure that all software is remotely updatable, and that the infrastructure is in place to do so regularly and at scale.

Best practice #15: make software update fault tolerant

A second update-related best practice is to make sure software update is fault tolerant. A fault[14] may occur for a variety of reasons that will cause software update to fail. For example, new software may have a defect that causes it to crash, or the storage mechanism (e.g., NOR flash) could have a flipped bit that corrupts the instructions within the program. IoT devices will often be deployed in a way that makes them difficult to service or update in person, and therefore software update must function properly even when faults occur during update or after the update process completes.

Azure Sphere implements several features to improve fault tolerance. First, Azure Sphere updates all Microsoft-authored software atomically. This means that an update either succeeds or fails, and all Microsoft software is updated at once. Before beginning to update the operating system, Azure Sphere first ensures that it has a complete, compressed, backup copy of each piece of software currently running on the device. This is called the Last Known Good, or LKG, version since it is the last known version of the software to run successfully on the device. Software update may fail for a variety of reasons: signature verification checks might fail, or the new software may fail to boot or might continuously crash during execution. If update fails, the device atomically rolls back by reinstalling the LKG and restoring the device to a state where it was known to run successfully. The device then lets the Azure Sphere Security Service know that the update failed and retries the software update later.

Second, once the OS update is downloaded, the chip reboots using the new Pluton Runtime and the new Security Monitor to execute the details of the update. The new software evaluates the downloaded software, updates the OS, and preserves the previous version. Booting into the new software before installing it provides a tremendous amount of flexibility. It allows wholesale changes in the way software update is executed because the update is always executed in the new, rather than the old, software. For example, an update could introduce new components or new update rules that the old software never knew about. Since software update is executed by the new software, the old software need not know about the new logic for the new software to contain it.

[14] David A. Patterson. An introduction to dependability. https://www.usenix.org/system/files/login/articles/1818-patterson.pdf

Finally, Azure Sphere implements erasure coding to guard against flash corruption due to bit flips or software bugs. Erasure coding is a technique that can repair data on flash if a portion of it is lost or corrupted without keeping an entire second copy of the protected data. The Azure Sphere Security Monitor creates erasure coding data during each software update, and it monitors the data protected by erasure coding for corruption. As an example, the erasure coding algorithm implemented on the MT3620 can detect and recover from up to a megabyte of continuous corruption of the operating system or application binaries.

Best practice #15 in summary: software update must be reliable. Azure Sphere uses several techniques to ensure that software update succeeds. It makes updates atomic, it retains the ability to roll back to the last known good version of the software if update fails, it boots into the new version of the software to conduct the installation, and it uses erasure coding to guard against corruption in flash storage.

Best practice #16: isolate applications to make upgrading easy

An application often has two sets of dependencies. Some dependencies are provided by the Azure Sphere operating system and are maintained and updated by Microsoft. Others are application-specific dependencies like hardware resources, libraries, media, or data unique to the application. Azure Sphere gathers all the application dependencies into a read-only application package format.

The application package makes update easier by putting all the updateable resources in a single binary object. Updating an application means replacing the existing package with a new one. In addition, Azure Sphere uses application packages to decouple software update for third-party applications from updates for the operating system. This feature is common on PCs and phones, but less common on MCUs. A device manufacturer can update its applications without restarting the device and without updating the operating system. Together, these features make it easier to update third-party applications more frequently and make it more likely that an application update will succeed.

In best practice #16, isolate applications and bundle their unique dependencies into a single package to make it possible to update applications independently of each other and the operating system.

Best practice #17: do not allow the system to dynamically change code execution

Dynamically introducing new functionality into a program or operating system is a common way to introduce tailor-made features from third parties at run time, rather than when a system is developed. However, these new features also introduce new attack surfaces that must be defended. One example of dynamic functionality is just-in-time (JIT) code execution. JIT compiles code at run time, literally generating new assembly language code during program execution. Some languages, like Java and C#, depend on this functionality as part of their overall application execution model. This model requires determining when it is okay for a program to generate new code, and when such code generation represents an attack. Azure Sphere eliminates dynamic code execution from its application programming model to remove this attack surface.

A second example of dynamic functionality is seen in Linux kernel modules, which enable new functionality at run time by inserting a precompiled binary into the kernel. This presents a security challenge because the kernel now supports dynamically linking and executing new functionality. Any kernel module that is loaded immediately bypasses all the defense-in-depth features that prevent applications from misbehaving. This attack surface must be secured, because any attack that compromises kernel module loading would allow an attacker to install arbitrary code into the kernel. Rather than attempt to secure this attack surface, Azure Sphere disables it. All functionality must be compiled into the kernel statically and exist in the single, signed binary that ships with every chip.

Best practice #17 in summary: dynamic code execution at run time introduces attack surfaces that are difficult to secure. Azure Sphere disables these attack surfaces, so that attackers cannot exploit them.

Best practice #18: defend against downgrade attacks

Signature verification ensures that only legitimate software is executed on the chip. However, another type of attack exists in which an egregious vulnerability is discovered and exploited. After devices are patched and updated, an attacker tricks a device into installing an old, legitimate, but vulnerable piece of software to launch an attack against it. This is called a downgrade attack and it is another way that attackers work around protections like secure boot to exploit a vulnerable device.

Azure Sphere can stop trusting (and running) all previous versions of the operating system. If an attacker discovers a way to trick a device into downgrading to a version of the operating system that is no longer trusted, the device will fail to boot and the OS will attempt to roll back to the Last

Known Good, which may in fact be a newer version of the OS. This downgrade protection is implemented by using additional e-fuses in the Pluton Fabric. The number of e-fuses available in the Pluton Fabric is limited, and therefore blowing fuses to revoke previous versions of the OS occurs only when an exploit or vulnerability is particularly egregious.

Best practice #18 in summary: make sure your chip can limit the effectiveness of downgrade attacks by using silicon to stop trusting vulnerable, but legitimate, software.

Best practice #19: use tools and processes to make software more secure

Automated tooling and processes are important in developing secured software. In this best practice, we describe four different techniques to make software development more secure: automated common vulnerabilities and exposures (CVE) checks, software fuzz testing, static analysis, and red team exercises.

The Common Vulnerability Exposure dataset[15] is a clearinghouse for identifying, describing, and publishing vulnerabilities in software. A CVE is one of the most important ways to know whether the software shipping on any device has a vulnerability. CVEs often end up patched in new versions of software or require protective countermeasures. Each CVE is assigned a rating to help software developers assess the risk of successful attacks against an unpatched system. Azure Sphere depends on open source software that may have CVEs. Azure Sphere's build system checks for CVEs in the operating system build process. If a CVE has been filed against a piece of software within the operating system, it is automatically flagged and a software developer who is monitoring the system is notified. This automatic flagging of CVE exposure eliminates the need to manually check whether a CVE is filed and allows the team to quickly evaluate the CVE and determine when and how to deploy a patch to fix it.

Software fuzz testing is a technique that attempts to generate random input to software that reads or parses data. The goal is to find inputs that crash the target software or fail in a way that creates an opportunity for an attacker to take control of the software. The software component runs within a special fuzz testing framework and can run millions and millions of iterations to try to find bugs in the component's implementation. The Azure Sphere software development processes integrate several different fuzz testing tools to look for bugs in data processing and parsing before the software ships to customers. Fuzz testing is done above and beyond any unit tests, integration tests, or stress tests written for the product.

[15] https://cve.mitre.org/

Static analysis uses a set of rules to find bugs in code as it is compiled. Static analysis looks for code patterns that typically indicate security vulnerabilities and flags that code for human analysis. While fuzz testing uses random inputs to find bugs, static analysis compares the code against a known set of rules and looks for patterns that result in buggy code or code that might be exploited. Azure Sphere runs several types of static analysis tools. A lightweight static analysis tool runs before any code commit completes. This tool checks against a small set of rules that the developer must not break when writing code. Additionally, the team regularly runs a much more complex, high-powered static analysis tool to look for more complex or subtle problems across the codebase. Bugs flagged by this tool are triaged and fixed regularly by the software development team.

The last process used by Azure Sphere is a red team exercise. In a red team exercise, a team of talented software engineers with a background in software security and vulnerability research are given detailed information about a product and are asked to find vulnerabilities in the system and exploit them. Unlike the processes already described in this section, red team exercises involve human ingenuity to find problems or vulnerabilities in the codebase. In other words, red team members are paid to think like attackers and to push the product to find problems in its implementation. Azure Sphere regularly hosts red team exercises against both the operating system and the Azure Sphere Security Service.

Writing secure software is difficult. Software always has unknown bugs. Best practice #19 is focused on implementing process and automation that make it harder for different classes of bugs to reach the field, and to find vulnerabilities before an attacker finds one and exploits it. Make sure your software development team implements automated CVE checks, fuzz testing, static analysis, and regular red team exercises to build more confidence in the software written and deployed to the field.

Conclusion

In 2017 Microsoft published a white paper on “The seven properties of highly secured devices.” This paper discusses why those properties are required for every internet-connected device—not only those that operate critical machinery. The cactus watering sensor demonstrates why the scale of a deployment matters just as much, if not more, than the type of device under attack. Any device that is deployed at scale can be weaponized to cause havoc as part of a botnet and therefore those devices should contain all seven properties.

Azure Sphere is Microsoft's own product offering that is seven-properties compliant. After years of implementation experience, we share our own best practices that were put into place as part of the Azure Sphere product. Those best practices are briefly summarized below.

1. **Treat ROM as non-updatable software and minimize its size.**
   Treating ROM as silicon leads to dangerous assumptions that attackers will exploit.

2. **Never expose private device keys to software.**
   Software, even the most trusted and privileged software on a device, should never be able to read or access a private key. This eliminates the possibility that an attacker could read that key by exploiting a software vulnerability.

3. **In IoT, choose ECC, not RSA, for device-specific keys.**
   ECC keys are smaller than their RSA counterparts, which allows IoT chips to be less expensive without sacrificing security. In addition, ECC keys can more easily be generated in silicon, without using software.

4. **Use Secure Boot everywhere and always.**
   All software should be signed and verified before execution.

5. **Use silicon-based Measured Boot to attest remotely that Secure Boot completed successfully.** Measured boot enables remote attestation, which allows remote servers to evaluate the software loaded by a device during boot. Implementing it in silicon guarantees that an attacker cannot forge its value.

6. **Do not use (or parse) certificates in the Trusted Computing Base (TCB).**
   Certificates are complicated and parsing them is error prone. Eliminate the need for certificate parsing by using other methods to ensure that key ownership is legitimate, such as generating keys in the hardware root of trust and collecting keys during chip manufacturing.

7. **Handle server certificate expiration gracefully.**
   IoT devices will be disconnected or turned off for hours, months, or days. An IoT device should manage certificate expiration automatically, without human intervention.

8. **Connectivity is optional.**
   Never require network connectivity to operate an IoT device.

9. **Make it harder to build botnets out of zero-day vulnerabilities.**
   Network firewalls should not be modifiable by an application at run time. Instead, make firewall configuration a part of application development to limit the ability of an exploit to contact a command and control server.

10. **Use a policy of "deny by default" and enforce it in silicon.**
   When a chip is first powered on, all resources should deny access by default. It is far easier to reason about granting access than to lock down resources later if the chip starts in a state where resources are accessible by default.

11. **Eliminate the concept of users on IoT devices.**
   Users need accounts and passwords, but IoT devices do not have users in the traditional sense. Eliminating the concept from an IoT device eliminates the attack surface.

12. **Physically separate real-time execution from internet communication.**
   Using separate cores within a single chip for real-time execution and internet connectivity isolates and protects the most critical code execution on a device.

13. **Divide code into user-mode and kernel-mode code.**
   Dividing code between user mode and kernel mode adds an additional layer of defense against attackers. Compromising an application does not automatically compromise the security of the entire system.

14. **Ensure that all software is updatable.**
   All software, even the lowest layers such as boot loaders, must be updatable. Non-updatable software presents a huge risk to device security when a vulnerability is found within that software.

15. **Make software update-fault tolerant.**
   Software update will fail. Build software update so that when a fault occurs, software update can recover and ensure the device continues to function.

16. **Isolate applications to make update easier.**
   Put an application's unique dependencies in a single package. This reduces the complexity of updating an application to the act of updating a single object.

17. **Do not allow the system to dynamically change code execution.**
   Any system that allows new code execution paths at run time must secure those mechanisms. Eliminating them for IoT devices eliminates one more area of attack.

18. **Defend against downgrade attacks.**
    Attackers will try to trick a device into running buggy, but legitimate, software. Build defenses into the device to prevent untrusted software from running.

19. **Use tools and processes to make software more secure.**
    Use some of the many excellent tools and processes to discover vulnerable software before it reaches your customers. Automatically check for CVEs, use fuzz testing and static analysis, and regularly hold red team exercises to improve the quality of software.

We hope that the discussion of these best practices sheds some additional light on the large number of features the Azure Sphere team implemented to protect IoT devices. We also hope that this provides a new set of questions to consider in evaluating your own IoT solution. Azure Sphere will continue to innovate and build upon this foundation with more features that raise the bar in IoT security.

Microsoft Azure