6

SECURE DEVICE IDENTITY

For a long time, embedded systems ran anonymously in the shadows and didn't care about remote access, digital business models, or sharing their data with other devices and cloud services. However, these days, those scenarios have changed fundamentally.

Suddenly, maintenance staff are now logging into devices remotely and can't verify that they're working with the correct device by looking at physical indicators. In addition, pay-per-use business models have become more and more popular in industrial scenarios, and devices write their own bills. Being able to prove the origin of usage data and mapping it to a specific customer is essential in this case. Moreover, devices made by different manufacturers have started talking to one another and exchanging data. All these trends have a strong requirement in common: every device needs a unique identity, and every device must be able to prove it.

The first part of this chapter investigates which properties contribute to device uniqueness and can serve as a basis for identity as well as the closely linked processes of identification and authentication. Next, we'll look at how the implementation of device identity management is regarded from two angles: the on-device storage of a cryptographic identity and the life-cycle management on the manufacturer's side. The chapter concludes with two case studies that explore identity generation and provisioning.

Every Device Is Unique

Mass production of consumer goods and industrial components might convey the impression that all the products rolling off the line are identical, right up to every bit in their firmware. However, if that were the case, how would you be able to tell one device from another? Of course, products have had stickers with serial numbers on them for a long time, but what if a sticker falls off, is removed on purpose, or even replaced with a forged version?

For modern devices, a unique identity should be an integral part of the device itself, and the component should be able to actively prove its identity to third-party devices, repair shops, and cloud services of the original manufacturer, just to name a few examples.

From a theoretical point of view, every single device—even with identical PCBs, microprocessors, and RAM—is clearly unique, because all these units are subject to (if only small) individual differences in material, timely behavior, power consumption, and so on. Academia is already working on exploiting the uniqueness of these tiny physical features to establish device identities. The corresponding research area is focused on *physical unclonable functions (PUFs)*, which have recently even found their way into the first commercial products.

The following sections explore what might be available in current devices that makes them unique from a practical point of view and how these unique identities can be proven to other parties.

Identification and Identifiers

Clearly, the term *identification* is closely related to the word *identity*. However, take a minute to think about its exact meaning.

If we want to define the process of identification, we could say it's the "claiming of a given identity." For example, if you meet someone at a conference, you could say, "Hi, my name is Joe!" You claim that you are Joe. The same happens if your device collects some usage data—let's say in the course of one month—and then connects to your backend to provide the data for customer billing. It will probably start with "Hi backend, my name is XY1337-0815!" It claims to be a device with a certain "name."

Unique Identifiers

Regarding uniqueness, telling somebody you're Joe is clearly not enough. Several Joes might exist, maybe even at the same conference. Adding your last name might narrow it down, but your name still won't be unique, at least on a global scale. If you take place and date of birth into consideration, you'll be closer to having a set of data that uniquely identifies *you*. These properties are called *identifiers*. Humans have many more of them: hair color, eye color, size, weight, and so on.

Since devices usually don't have human-like names, manufacturers have to take another path for identification. For a long time, typical identifiers

have been vendor-chosen values like model type, serial number, and date of production.

With the advent of the internet, the need for worldwide identifiers became clear. Back then, the concept of UUIDs, also known as *globally unique identifiers (GUIDs)*, was proposed. It's standardized in RFC 4122, among others, and is meant to provide 128-bit unique identifiers that don't require a central registration process. Although the probability of identifier collision is not zero, it's regarded as very close to zero in practice. The generation of UUIDs can, for example, be performed by the Linux RNG, as shown in Chapter 3.

From a cryptographic point of view, public keys generated by asymmetric crypto algorithms like RSA and ECDSA also can perfectly serve as identifiers. They might even be combined with a subject name and further attributes to obtain a unique device certificate for identification as, for example, standardized in the network authentication standard IEEE 802.1AR.

System Identities

While some devices consist of a single central component that constitutes the whole device and its identity, other product architectures are more modular and allow for partial replacements in case of defects or hardware upgrades. Discussing which components contribute to the device's identity and which don't is worthwhile for the latter cases. The physical parts of an embedded system provide a multitude of identifiers like media access control (MAC) addresses of network cards, Bluetooth chipsets, and Wi-Fi controllers, but also serial numbers and unique identifiers of CPUs, flash memories, and removable media.

Requiring a set of identifiers to be part of the system identity also means that the system identity has to be regenerated or reapproved if one of those parts changes. This requirement can be an advantage for manufacturers—for example, to force users to purchase spare parts of the same brand—because every exchange requires the acknowledgment of the manufacturer. However, system identities and forced manufacturer approval can also cause additional workload on the manufacturers' side. Further, if every little change requires a feedback loop with the original manufacturer, it could significantly limit operators' freedom to act in their daily business.

NOTE *Sometimes a device's reliability is the utmost goal, and if its hardware breaks, it has to be immediately replaced by an operator. In such cases, allowing a device's identity to be transferable is reasonable–for example, with a removable memory card.*

Authentication and Authenticators

In everyday language, *identification* and *authentication* are sometimes used synonymously, but authentication means much more than merely claiming an identity.

If the validity and correctness of your identity are really important—for example, if you have to apply for a passport or register to vote—and you tell

them "Hi, I'm Joe," they'll probably reply, "Hi, Joe, please show me your ID card." They'll make you *prove* your identity—the analog equivalent of a digital authentication process.

The term *authentication* means that you have to *confirm* the identity you claimed during identification some seconds before. To do so, you need to possess a valid *authenticator* corresponding to the given identity. For humans, authenticators can be ID cards, driver's licenses, and so on. For all these IDs, an authority at some point in time verified the human identity and subsequently issued a corresponding authenticator that's usually valid for a certain amount of time. During this validity period, the authority and others can use the provided authenticator to verify a specific identity.

For devices, typical authenticators are symmetric secret keys or asymmetric private keys, (temporary) authentication tokens, or passwords (in legacy cases). These authenticators were created and issued for a specific device (for example, during production), and they can be used to prove cryptographically the identity of that same device at a later time.

Authentication Protocols

Depending on the type of authenticator, the authentication process is performed in different ways. A common approach is a *challenge-response authentication protocol*. Figure 6-1 shows one form of a challenge-response handshake.

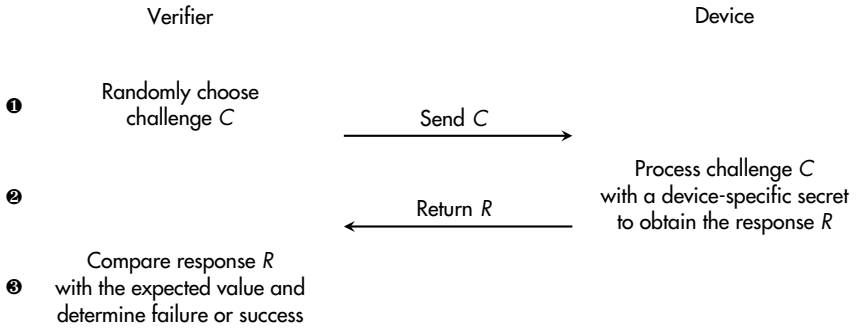

Figure 6-1: The typical steps during challenge-response authentication

The challenge-response authentication process starts with the generation of a random challenge C ❶ on the verifier side that is subsequently transmitted to the device. The device processes this unpredictable value with its secret authenticator and yields a response R that's returned to the verifier ❷. In the final step ❸, R is compared to its expected value to decide whether authentication was successful.

For symmetric secrets, the on-device algorithm processing the given challenge with the device-specific secret could be a hash function or an HMAC construction. However, the disadvantage is that the secret also has to be available in the verifier's database to compute the correct expected value and is not solely stored within the device.

In contrast, asymmetric cryptography allows for device-only authenticators that never leave the device, which is the most secure solution. Specifically, digital signatures based on RSA or ECDSA, as explained in Chapter 2, could be used to generate authentication responses from random challenges. In this case, the verifier would need only the corresponding public key in order to check the validity of the returned signature.

NOTE *In most cases, authentication is possible only with secrets. Therefore, confidentiality is a natural protection goal for all kinds of authenticators. If broken, device impersonation becomes a likely threat.*

Dedicated Authentication Chips

As introduced in Chapter 5, semiconductor manufacturers offer a variety of authentication chips that not only securely store authenticators but also provide an algorithmic means to perform a challenge-response handshake for authentication purposes.

This approach has two advantages. First, extracting the secret authenticator from the chip is a pretty difficult task for attackers. Second, since these chips usually come with integrated support for asymmetric cryptography, mainly digital signatures based on elliptic curves, the secret never has to leave the physical boundaries of the chip.

On the other hand, with this approach, you now have another component on your BOM, you need space on the PCB, and necessary software integration efforts of these devices vary among vendors. In addition, an attack vector is often overlooked—namely, the physical transfer of such an identity chip to another device. The simple 8-pin packages could be desoldered and integrated into a different original device or even into a custom attacker device. As in code-lifting attacks, adversaries might not care about the secret inside the chip if they can move the whole chip to their desired location.

Multifactor Authentication

For the authentication of human users, *multifactor authentication (MFA)* has gained a lot of attention over the last several years. Following the principle of defense in depth, MFA requires attackers to capture not only one authenticator, such as a password, but also at least a second factor, like a temporary token generated in a mobile app or by a hardware token. Since passwords are stored in human brains (or password managers) and additional authenticators often originate from an additional hardware device or at least a different communication channel, the necessary effort for successful attacks is significantly higher.

For device authentication, the situation is a little bit different because devices don't use brains and mobile apps for authenticator storage and generation. However, you could still consider a multifactor approach—for example, using one authenticator stored in firmware and a second one that originates from a dedicated authentication chip. The authentication process would then consist of two handshakes, one with the hardware component

and one based on the device software, forcing attackers to compromise two different parts of your device if they want to get hold of its identity.

Besides additional explicit authenticators, you can use implicit, environmental parameters to strengthen device authentication. A common example is geographical limitations, also known as *geo-fencing*. In that scenario, device authentication (or general operation) succeeds only if the device's location matches a predefined area. One way to determine this parameter is the public IP address the device uses for internet communication. Of course, the security gain by these implicit authentication properties is maximized if an attacker who has compromised a device can't forge the parameters. They should be observable from the outside and not just claimed by the device itself.

Trusted Third Parties

In the past, the main verifier of a device's identity was the manufacturer of the same device. Proprietary (and eventually insecure) authentication processes did their job. However, in a multilateral digital ecosystem in IoT and IIoT scenarios, the need for cross-manufacturer device authentication becomes obvious.

This requirement means manufacturers have to trust authenticators of other devices, including competitors. Since one-to-one trust relations between manufacturers would lead to enormous management overhead, the concept of a *trusted third party (TTP)* is necessary, as shown in Figure 6-2.

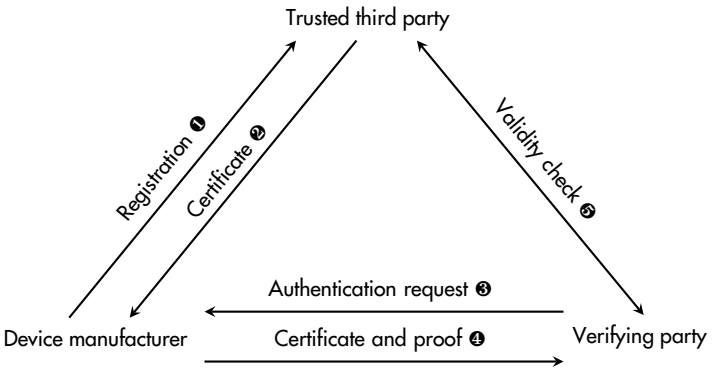

Figure 6-2: The role of a trusted third party in device authentication

In this approach, manufacturers register their device identities with the TTP ❶. After verification, the TTP certifies the given identity and returns a device-specific certificate ❷. Upon an authentication request in the field ❸, the device can provide the issued certificate and cryptographically prove that it is in possession of the corresponding authenticators ❹. However, at this point, the verifying party can't be sure that the given cryptographic data corresponds to the actual device identity. The verifier finally has to check the validity of the authentication ❺, either in direct communication with

the TTP or by using data like public keys provided by the TTP. Afterward, a reliable trust relation with a previously unknown device can be established.

Certificates and Certificate Authorities

I've been using the term *certificate* to describe a digital document that's issued by a TTP to confirm a device's identity. Technically speaking, the most common implementations of this concept are X.509 v3 certificates based on asymmetric cryptography as specified in RFC 5280.

The purpose of these certificates is to bind a given public key to its corresponding subject, such as a device, and to a set of attributes including a validity period and a certificate serial number. A *certificate authority (CA)* digitally signs these values with its own private key. This CA is also included in the certificate, in the `Issuer` field. The result is the smallest version of a *certificate chain*, which means that a device certificate and its public key can be cryptographically verified, and if successful, the next certificate (namely, that of the CA) has to be verified. Authentication is trusted only if both verifications succeed.

In practice, a manufacturer might have its own product CA, which is certified by an intermediate CA of a TTP, which is again certified by an internationally recognized root CA. With that process, the rather complex hierarchical certificate chains are established that have to be verified up to their root, whenever a device needs to authenticate itself.

The *root certificates* aren't certified by anybody; they're self-signed and have to be available in some kind of root store on the verifier's side. This means that verifying parties also must unconditionally trust all their root certificates. Therefore, the root store requires strong integrity protection; otherwise, attackers can inject new trust relations by manipulating the stored certificates.

In several cases, a certificate can't be trusted until its actual end of validity—for example, because of a private-key compromise, device theft, or similar issue. For such situations, CAs maintain a *certificate revocation list (CRL)* that lists all certificates no longer trusted even though their validity period is not yet over. The *Online Certificate Status Protocol (OCSP)* is a common protocol for checking the revocation status of a certificate during authentication, as standardized in RFC 6960.

The whole architecture of verification, certification, and revocation, and the corresponding processes and services, are often referred to as the *public-key infrastructure (PKI)*. Since such a system demands significant maintenance and documentation efforts, small and medium-sized companies often hesitate to implement it themselves and rely on PKI service providers instead, which means TTPs.

Identity Life Cycle and Management

Now that we've covered the basic concepts of device authentication, this section establishes the need for reliable strategies for managing the *life cycle* of

device identities. Life-cycle management has four main steps, as depicted in Figure 6-3: identity generation, its provisioning in manufacturer systems as well as within the device, everyday usage in the field, and the often-forgotten exchange or destruction of the same identity.

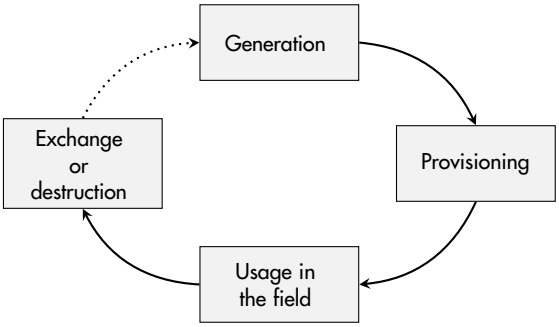

Figure 6-3: The life cycle of device identities

WARNING

Don't treat life-cycle management as optional. Even if you've solved all the technical challenges regarding identifiers, cryptography, and secure memory, make sure your organization has answers regarding the organizational challenges ahead.

Generation

A device's identity can be generated in various places and at various times. The place and time you choose affect the security requirements and procedures of your production process. If you use electronics manufacturing services (EMS) to manufacture your product, a trustful and close cooperation with your service provider is essential.

Generating the identity on the device itself during production can be the most secure option of all, but only if the corresponding authentication secrets *never* leave the device. Asymmetric cryptography enables this use case because the generated private key might stay on the device, and its public counterpart can be made available to potential verifiers. Of course, you might also generate a symmetric secret on the device during the production process, but in that case, the key has to be exported later to enable identity verification.

While on-device generation has security advantages, it comes with operational and practical challenges. Imagine that a device "loses" its identity because the memory that stores it gets broken. If that was the single storage location, a new identity has to be generated after repair, which might lead to a conflict because a new entry is generated in your product database, but the hardware is actually old. Also, your customers would have to replace the old device identity with the new one in their asset management systems. If you as a manufacturer have an identity backup, this case could be handled easily, but at the cost of security.

A second disadvantage of on-device generation can be the late availability of product identities, because they are available only after a certain manufacturing step is completed. Sometimes that might be exactly what you want,

but if your device identities have to be populated in your own IT systems to enable smooth operation from day one, you might want to prepare those processes with your device identities even before actual production.

WARNING *On-device identity generation based on RSA keys is a nondeterministic process and takes a variable amount of time. This limitation has to be considered when planning production processes, especially for low-performance devices.*

Generating an identity outside the device provides more flexibility in managing the device identity before production and in cases of repair. Authentication secrets are prepared in advance within an identity management system and provided to production in a second step. However, this means that these identities exist before the real device is even assembled and, of course, they already carry the protection goals of confidentiality and integrity. Information disclosure or data manipulation before production could have severe consequences for the security of your product.

A last point that might influence your on-/off-device generation decision is the involvement of a TTP. If you generate identities during production, they have to be registered, verified, and certified with a third party within a tight schedule. Of course, that's possible and desirable, and it's already implemented by leaders in this field, but it requires a significant amount of infrastructure and process management efforts.

Provisioning

Depending on the identity generation phase, the following provisioning step comes in two flavors with their own pros and cons. In both cases, the end result should be that the identity is provisioned on the device itself and within the product-tracking and identity management system of the manufacturer and the eventually involved EMS provider.

After on-device generation, all manufacturer systems have to be provisioned with the new identity, which requires a read-out step during production. For asymmetric crypto, only the public key or a corresponding certificate from a TTP has to be stored in the manufacturer's identity database. However, if backups of authentication secrets are desired, you can create them by extracting the private key at this point.

The offline generation of identities requires information flow in the other direction—namely, from an identity management system to the device to be produced. Clearly, a programming step is necessary, in which the secret and the attributes of the pregenerated identity are written to specific memory locations or hardware resources within the product. This step might be integrated in existing firmware programming procedures or similar processes.

In all these cases, when sensitive data is transferred to or from a device during provisioning, at least the integrity and often also the confidentiality of this communication should be guaranteed. Otherwise, authentication secrets might be disclosed, devices might end up with a manipulated identity, or the manufacturer's identity data might be corrupted.

NOTE *If your device identities are generated before production and are then sent to your EMS provider by email or on a USB stick by snail mail, consider carefully whether this meets your protection goals. If you're honest, it probably doesn't.*

Usage in the Field

The previously generated and provisioned identities are used for authentication in the field. So far, so good. Can we take any other precautions during everyday usage? Absolutely. An identity management system allows us to perform sanity and plausibility checks whenever devices authenticate with our systems.

Imagine your authentication logs show that the same device connects from two locations within a short time. This might be an indicator that somebody has stolen a device's identity and is using it for their own purposes. If such cases can be identified early and specific investigations follow in a timely manner, damage can be significantly limited.

Exchange or Destruction

Even if some devices (especially in industrial, military, or space applications) are meant to last forever physically, their authentication secrets usually don't. On the internet, a common validity period for web server certificates is 90 days (as, for example, implemented by Let's Encrypt at *https://letsencrypt.org*), which means that these identities have to be regenerated at least every three months.

Clearly, identity renewal in IoT and IIoT scenarios is still far away from such high frequencies. However, at least if X.509 certificates are used for authentication purposes, a validity period is a mandatory parameter that has to be specified, either by your company or by the TTP of your choice. Some manufacturers issue device certificates with a validity period of 20 years or more, but even if the chosen crypto is future-proof, it's hard to estimate whether such an identity will still be trustworthy after 15 years or more.

Some network products (for example, those from Cisco) support certificate management protocols like the *Simple Certificate Enrollment Protocol (SCEP)* or its more recent alternative, *Enrollment over Secure Transport (EST)*. Since this is new ground for IoT and IIoT devices, no common standard has been established as of this writing, but it's pretty clear that automation is key to continuous and reliable identity and certificate management.

NOTE *In 2022, manufacturers of security gateways for accessing the German health telematics infrastructure claimed that devices had to be physically replaced because the validity of their five-year cryptographic identities came to an end. Subsequently, the Chaos Computer Club (CCC) proved the opposite and, by its own account, saved the German healthcare system €400 million. This is just one example that emphasizes the importance of robust identity-renewal processes.*

The final step of an identity's life cycle is literally its destruction. While physical removal is not always possible, a manufacturer should at least be prepared to revoke the trust relation for a specific device if it reaches its end of life before the defined end of its validity period. A typical measure for this purpose is a CRL maintained by a CA or a trust status flag in your own manufacturer database.

Case Study: Identity Generation and Provisioning

In this case study, I investigate the availability of identifiers for an STM32-MP157F-DK2 and how to extract them in order to derive a system identity. Further, we'll see how to prepare a certificate signing request on this device that can subsequently be provided to a TTP, which in turn, is able to issue a valid device certificate.

Identifiers and System Identity

The STM32MP157F-DK2 evaluation kit is an embedded system that consists of several components. Many of these components come with their own identifiers that engineers might capture and use to create a comprehensive device identity.

A common identifier is the serial number of a device's main CPU. In this regard, ST's *Reference Manual RM0436* for STM32MP157F devices states: "The 96-bit unique device identifier provides a reference number, unique for a given device and in any context. These bits cannot be altered by the user." This unique ID (UID) is immutably stored in the OTP memory of the STM32-MP157F chip. Listing 6-1 shows that this UID is split into three 32-bit words that can be read from specific memory addresses.

```
Base address: 0x5C00 5000 (BSEC base address on APB5)
Address offset: 0x234 = UID[31:0]
Address offset: 0x238 = UID[63:32]
Address offset: 0x23C = UID[95:64]
```

Listing 6-1: The physical addresses of the UID in STM32MP157F devices

We can use the `devmem2` command line tool to read physical memory addresses. As shown in Listing 6-2, the application outputs three 32-bit words representing the chip's identity, given a combination of base address and UID offsets.

```
# devmem2 0x5c005234
...
Read at address  0x5C005234 (0xb6fb0234): 0x0038003D
# devmem2 0x5c005238
...
Read at address  0x5C005238 (0xb6fb9238): 0x34385114
```

```
# devmem2 0x5c00523c
...
Read at address  0x5C00523C (0xb6f1423c): 0x36383238
```

Listing 6-2: Reading the CPU UID of my STM32MP157F device from physical addresses

On Linux systems, the serial number is also available from */proc/cpuinfo*. The output shown in Listing 6-3 confirms that the serial number is the same as that extracted from the raw memory locations before.

```
# cat /proc/cpuinfo | grep Serial
Serial : 0038003D3438511436383238
```

Listing 6-3: Capturing the CPU serial number available in Linux

However, the STM32MP157F chip is not the only one on the PCB. ST's *User Manual UM2637* describes a multitude of implemented communication interfaces. Besides classic Ethernet networking, the device includes an IC that provides Wi-Fi and Bluetooth capabilities. All these interfaces have unique MAC addresses that might be used to derive system identities. Listing 6-4 shows how to extract those values when running on Linux.

```
# cat /sys/class/net/eth0/address
10:e7:7a:e1:81:65
# cat /sys/class/net/wlan0/address
48:eb:62:c4:0a:08
# cat /sys/kernel/debug/bluetooth/hci0/identity
43:43:a1:12:1f:ac (type 0) 00000000000000000000000000000000 00:00:00:00:00:00
```

Listing 6-4: Extracting Ethernet, Wi-Fi, and Bluetooth MAC addresses in Linux

Finally, one part of the system can be removed and replaced easily: the removable media card. In my case, it's a microSD card that contains a *card identification (CID)*. This 128-bit value uniquely identifies an SD card. Among other information, it contains a manufacturer ID, a product serial number, and the date of production. Again, Linux provides a corresponding entry in its `sysfs` that can be read out as illustrated in Listing 6-5.

```
# cat /sys/block/mmcblk0/device/cid
275048534431364760dad3df9a013780
# cat /sys/block/mmcblk0/device/serial
0xdad3df9a
```

Listing 6-5: Reading the unique CID of an SD card

Besides the `cid` value, Linux provides the `serial` value for an SD card, which solely contains the memory card's serial number.

For this case study, let's assume your team has chosen to use the central CPU ID and the Wi-Fi MAC address as the two relevant system identifiers. They can be combined by a hash function, as shown in the next section.

Certificate Signing Request

A *certificate signing request (CSR)* is a data structure that requests a CA to certify that a given public key is bound to a specific identity, a device identity in this case. Linux offers several ways to generate a CSR and provide the necessary information. Listing 6-6 shows the imports necessary to accomplish RSA key generation and CSR creation with the help of the `cryptography` Python module. Also, the `subprocess` module is included to get system identifiers by using the available command line tools.

```
import subprocess
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes
```

Listing 6-6: The necessary imports from the `cryptography` and `subprocess` modules

The first part of the on-device identity generation is usually based on asymmetric cryptography (RSA, in this case). As shown in Listing 6-7, a random key pair can be created with a single line.

```
# Generate RSA key
key = rsa.generate_private_key(public_exponent=65537, key_size=4096 ❶)

# Write key to disk
with open('dev.key', 'wb') as f:
    f.write(key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=
        serialization.BestAvailableEncryption(❷ b'PrivateKeyPassphrase'),
    ))
```

Listing 6-7: An on-device generation of an RSA key pair

For this case study, I decided to use an RSA key length of 4,096 bits ❶ to account for an (I)IoT device's lifetime of several years. To simplify this example, the generated private key is stored in the *dev.key* file and is protected by a standard passphrase ❷. In an actual production environment, the key should be stored in a secure way, as discussed in Chapter 5.

Listing 6-8 shows an example procedure of identifier collection and processing.

```
# Collect system data
❶ output = subprocess.Popen('cat /proc/cpuinfo | grep Serial',
          shell=True, stdout=subprocess.PIPE)
cpu_serial = output.stdout.read().split()[2]
```

```
❷ output = subprocess.Popen('cat /sys/class/net/wlan0/address',
          shell=True, stdout=subprocess.PIPE)
  wifi_mac = output.stdout.read().split()[0]

  # Hash collected system data
❸ digest = hashes.Hash(hashes.SHA256())
  digest.update(cpu_serial)
  digest.update(wifi_mac)
  system_id = digest.finalize()
❹ system_id = system_id[:4].hex()
```

Listing 6-8: Collection and processing of an on-device identifier

In the first step, the CPU serial number ❶ and the Wi-Fi MAC address ❷ of the produced system are read by the means Linux provides. Subsequently, the hash function SHA-256 ❸ is used to process those values and to derive a 4-byte system identifier ❹ that would change if the CPU or the Wi-Fi chip is replaced in the future. The SD card ID is neglected on purpose, because SD cards break every now and then, which would lead to an unnecessarily high demand for identity regeneration.

For a device certificate and a CSR, respectively, you need to specify a *common name* for the device, as shown in Listing 6-9.

```
# Manufacturer data
manufacturer = 'IoT Devices Corp'
manufacturer_device_serial_no = 'IOTDEV-1337-08151234'

# System name for CSR and certificate
❶ cert_common_name = manufacturer_device_serial_no + '-' + system_id

# Generate CSR and sign with private key
csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
    x509.NameAttribute(NameOID.ORGANIZATION_NAME, manufacturer),
  ❷ x509.NameAttribute(NameOID.COMMON_NAME, cert_common_name),
❸ ])).sign(key, hashes.SHA256())

# Write CSR to disk
with open('dev.csr', 'wb') as f:
    f.write(csr.public_bytes(serialization.Encoding.PEM))
```

Listing 6-9: An on-device CSR preparation

In this case study, the unique device name is the combination of the serial number given by the manufacturer and the hardware-dependent system identifier ❶. This string is used as an input to the CSR generation ❷, together with the manufacturer's name in the CSR's *organization* field. Finally, the device signs the CSR with its unique confidential private key ❸. Afterward, the CSR is stored in the *dev.csr* file.

The saved CSR file has to be transmitted to the CA responsible for certifying the identity of produced devices. Also, the manufacturer or EMS

provider might extract the collected and generated device data in a database. As an example, Listing 6-10 shows the data from an STM32MP157F device.

```
Collected CPU serial number:  0038003D3438511436383238
Collected Wi-Fi MAC address:  48:eb:62:c4:0a:08
Derived system identifier:    f30cf858
Given device serial number:   IOTDEV-1337-08151234
Common name in certificate:   IOTDEV-1337-08151234-f30cf858
```

Listing 6-10: Example output of identifier data from my STM32MP157F device

As you can see, a 4-byte system identifier is generated from the listed individual identifiers and appended to the device serial number. This string is subsequently used as the common name for the generated CSR.

Certificate Authority

Before we issue the final certificate, let's look at what the CSR contains. Listing 6-11 shows how to display CSR contents with the `openssl req` command line tool.

```
$ openssl req -in dev.csr -noout -text
Certificate Request:
    Data:
        Version: 1 (0x0)
     ❶ Subject: O = IoT Devices Corp, CN = IOTDEV-1337-08151234-f30cf858
     ❷ Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:d3:a0:14:fb:e1:0e:d0:74:3d:26:d4:ef:a1:ed:
                    ...
                    c9:2a:f5:46:e4:b2:ad:a9:5e:ee:cb:79:85:d9:1e:
                    9f:3e:57
                Exponent: 65537 (0x10001)
        Attributes:
            (none)
            Requested Extensions:
    Signature Algorithm: sha256WithRSAEncryption
 ❸ Signature Value:
        81:98:b1:e8:c2:fe:3a:55:32:39:2e:27:ce:2c:a8:54:bd:04:
        ...
        17:77:6c:a1:5b:4a:a7:ed:22:55:33:23:26:55:05:90:26:d2:
        90:7a:5e:34:65:80:32:4e
```

Listing 6-11: Example CSR for my specific STM32MP157F device

The subject ❶ is represented by an organization string (`O`) and a common name (`CN`) as specified in our CSR preparation script in Listing 6-9,

followed by its corresponding RSA public key ❷. The device's digital signature ❸ can be clearly identified at the end of the given request. The CA can use it to verify whether the requesting subject actually has access to the private key corresponding to the given public key in the CSR.

CA and PKI infrastructures usually consist of complex processes with a variety of organizational and technical measures to ensure proper and trustworthy functioning. As shown in Listing 6-12, we create a test CA that's far from production-ready but okay for educational purposes. The term *quick and dirty* might be applicable here.

```
$ openssl genrsa -out ca.key 4096
$ openssl req -new -x509 -key ca.key \
              -subj "/C=DE/L=Augsburg/O=Super Trusted Party/CN=CA 123" \
              -out ca.crt
```

Listing 6-12: A quick generation of a test CA with openss1 tools

We can generate the test CA with the help of the `openssl genrsa` tool. The first command in Listing 6-12 generates a 4,096-bit RSA key pair for the CA and stores it as *ca.key*. Since, in this case study, this is the root of the CA, the corresponding certificate has to be self-signed. The *ca.crt* certificate can be obtained by using the `openssl req` tool and telling it the CA's attributes—for example, the country (`DE` for Germany) and city it's located in (`Augsburg`), its organization's name (`Super Trusted Party`), and its common name (`CA 123`).

After the CA has registered and successfully verified the certificate request at hand, it takes the CSR data and adds attributes like the validity period. In Listing 6-13, you can see that the `-days` parameter is set to `3650`, which means that the issued certificate is valid for 10 years.

```
$ openssl x509 -req -in dev.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
               -days 3650 -out dev.crt
```

Listing 6-13: Generating a certificate from a CSR with the openss1 tool

In the device certificate generation process, the CA decides on the length of the validity period, but of course that has an influence on your device identity life cycle. Make sure to choose this value deliberately.

Let's look at the final result of this demanding process. The `openssl x509` tool is able to output the device certificate contents, as shown in Listing 6-14.

```
$ openssl x509 -in dev.crt -noout -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
          ❶ 45:3c:c3:30:c1:e3:c2:a9:49:5c:14:d6:16:5d:79:69:24:6c:31:66
        Signature Algorithm: sha256WithRSAEncryption
      ❷ Issuer: C = DE, L = Augsburg, O = Super Trusted Party, CN = CA 123
        Validity
            Not Before: Apr  5 11:18:13 2024 GMT
```

```
        ❸ Not After : Apr  2 11:18:13 2034 GMT
       Subject: O = IoT Devices Corp, CN = IOTDEV-1337-08151234-f30cf858
       Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
               RSA Public-Key: (4096 bit)
               Modulus:
                   00:d3:a0:14:fb:e1:0e:d0:74:3d:26:d4:ef:a1:ed:
                   ...
                   c9:2a:f5:46:e4:b2:ad:a9:5e:ee:cb:79:85:d9:1e:
                   9f:3e:57
               Exponent: 65537 (0x10001)
   Signature Algorithm: sha256WithRSAEncryption
❹ Signature Value:
       75:d5:07:71:ec:fe:c6:27:fd:e2:a7:1c:fa:b9:89:b3:9c:0f:
       ...
       8d:fa:f6:f1:53:79:32:1e:a8:ec:6f:f7:03:57:2f:7b:f4:fb:
       45:77:6a:f8:c6:70:72:41
```

Listing 6-14: The certificate contents of a sample device

In comparison to the original CSR, you can see that the CA added a certificate serial number ❶ and its own data at the `Issuer` field ❷. The validity period ❸ is set to be 10 years from the moment of issuance. And, finally, all these attributes are signed by the CA ❹ together with the device's information and its public key. Now every entity that trusts the used CA is able to authenticate the produced device.

After issuing the certificate, it has to be provided to the device itself, but also to the manufacturer's identity management system. During production, this whole process of generation, certificate issuance, and provisioning should run with a high degree of automation and with precautions taken to minimize threats to confidentiality and integrity.

Case Study: RSA Key Generation in Production

Although ECDSA has some advantages over RSA, as discussed in Chapter 2, it's still widely used in certificates. However, if you work with RSA, be aware that RSA key generation is a nondeterministic process and might take varying amounts of time.

This second, brief case study investigates how much time is required during the production process to generate RSA keys of a given length. Listing 6-15 shows a simple way to analyze RSA key-generation times.

```
import time
from cryptography.hazmat.primitives.asymmetric import rsa

time_data = []
for n in range(16):
    start_time   = time.time()
    key = rsa.generate_private_key(public_exponent=65537, key_size=4096)
```

```
    elapsed_time = time.time() - start_time
    print('Try', n, ': RSA 4096-bit key generation took',
          '{:.3f}'.format(elapsed_time), 'seconds!')
    time_data.append(elapsed_time)
print('MIN:', '{:.3f}'.format(min(time_data)), 'seconds')
print('MAX:', '{:.3f}'.format(max(time_data)), 'seconds')
print('AVG:', '{:.3f}'.format(sum(time_data)/len(time_data)), 'seconds')
```

Listing 6-15: An RSA key-generation timing analysis

This example uses the `cryptography` Python module and the parameters from the previous case study. It performs 16 tries for simplicity, but a sound statistical analysis would require a larger number of test runs. Listing 6-16 shows exemplary results of RSA 4,096-bit key-generation times obtained by running the code from Listing 6-15 on my STM32MP157F device.

```
# python3 rsa_key_gen_time.py
Try 0 : RSA 4096-bit key generation took 59.920 seconds!
Try 1 : RSA 4096-bit key generation took 28.696 seconds!
Try 2 : RSA 4096-bit key generation took 72.872 seconds!
Try 3 : RSA 4096-bit key generation took 109.765 seconds!
...
Try 12 : RSA 4096-bit key generation took 48.925 seconds!
Try 13 : RSA 4096-bit key generation took 50.885 seconds!
Try 14 : RSA 4096-bit key generation took 90.907 seconds!
Try 15 : RSA 4096-bit key generation took 40.634 seconds!
MIN: 28.696 seconds
MAX: 109.765 seconds
AVG: 62.768 seconds
```

Listing 6-16: RSA key-generation timing results on my STM32MP157F device

The variation of generation times is not negligible. The RSA key generation may finish within 30 seconds but might also take 110 seconds or even more. This variation has to be considered in production scheduling, and since an upper bound doesn't exist for the generation time, you have to expect outliers that might take significantly longer.

Summary

There's no doubt that every single device is a physically unique object. With the help of identifiers like CPU serial numbers, MAC addresses, and values chosen by the manufacturer, we're able to represent this uniqueness in the digital space and provide a base for device identities.

However, merely claiming an identity isn't enough for most applications. Devices have to be able to cryptographically prove their identity with the help of unique and confidential authenticators like cryptographic keys. This process is called *authentication*. The secure storage of those authentication secrets is essential to prevent impersonation attacks. Chapter 5 provided some ideas for confidential data storage in hardware or software.

A common concept to establish trust in device identities is the registration of devices at third parties that verify their identities and issue digital device certificates. These can be used by anybody trusting the issuer to authenticate a device.

Besides the technical challenges of binding a digital identity to a device, a much broader field of organizational processes have to be specified to provide secure and reliable identity life-cycle management. These processes often involve EMS providers, TTPs, and your custom process specifics, which leads to a complexity that should never be underestimated.

The more you dive into this topic, the more "interesting" problems you will discover. For several years, researchers have been working on PUF implementations to exploit manufacturing process variations in order to derive implicit chip identities, and the first products on the market already contain such circuits. Further, identity management automation in on- and offline scenarios and corresponding protocols like SCEP and EST will certainly gain more attention in the future, providing a major step forward for managing secure device identities.