The Internet of Things on AWS – Official Blog

Understanding the AWS IoT Security Model

by Nick Corbett | on 11 MAY 2017 | in Technical How-To | Permalink | Share

According to Gartner, the Internet of Things (IoT) has enormous potential for data generation across the rough billion endpoints expected to be in use in 2020(1). Internet of Things (IoT) devices in use. Its easy to get excited about this rapid growth and envisage a future where the digital world extends further into our world. Before you take the decision to deploy devices into the wild, it's vital to understand how you will maintain your security perimeter.

In this post, I will walk you through the security model used by AWS IoT. I will show you how devices can authenticate to the AWS IoT platform and how they are authorized to carry out actions.

To do this, imagine that you are the forward-thinking owner of a Pizza Restaurant. A few years ago, most of your customers would have picked up the phone and actually spoken to you when ordering a pizza. Then it all moved on-line. You now want to give your customers a new experience, similar to the Amazon Dash Button. One press of an order button and you will deliver a pizza to your customer.

The starting point for your solution will be the AWS IoT Button. This is a programmable button based on Amazon Dash Button hardware. If you choose to use an AWS IoT Button, the easiest way to get up and running is to follow one of the Quickstart guides. Alternatively, you can use the Getting Started with AWS IoT section of the AWS Documentation.

Who's Calling?

When someone presses an AWS IoT Button to order a pizza, it's important to know who they are. This is obviously important as you will need to know where to deliver their pizza, but you also only want genuine customers to order. In much the same way as existing, on-line customers identify themselves with a username, each AWS IoT Button needs an identity. In AWS IoT, for devices using MQTT to communicate, this is done with an X.509 certificate.

Before I explain how a device uses an X.509 certificate for identity, it is important to understand public key cryptography, sometimes called asymmetric cryptography (feel free to skip to the next section if you are already familiar with this). Public key cryptography uses a pair of keys to enable messages to be securely transferred. A message can be encrypted using a public key and the only way to decrypt it is to use the corresponding private key:

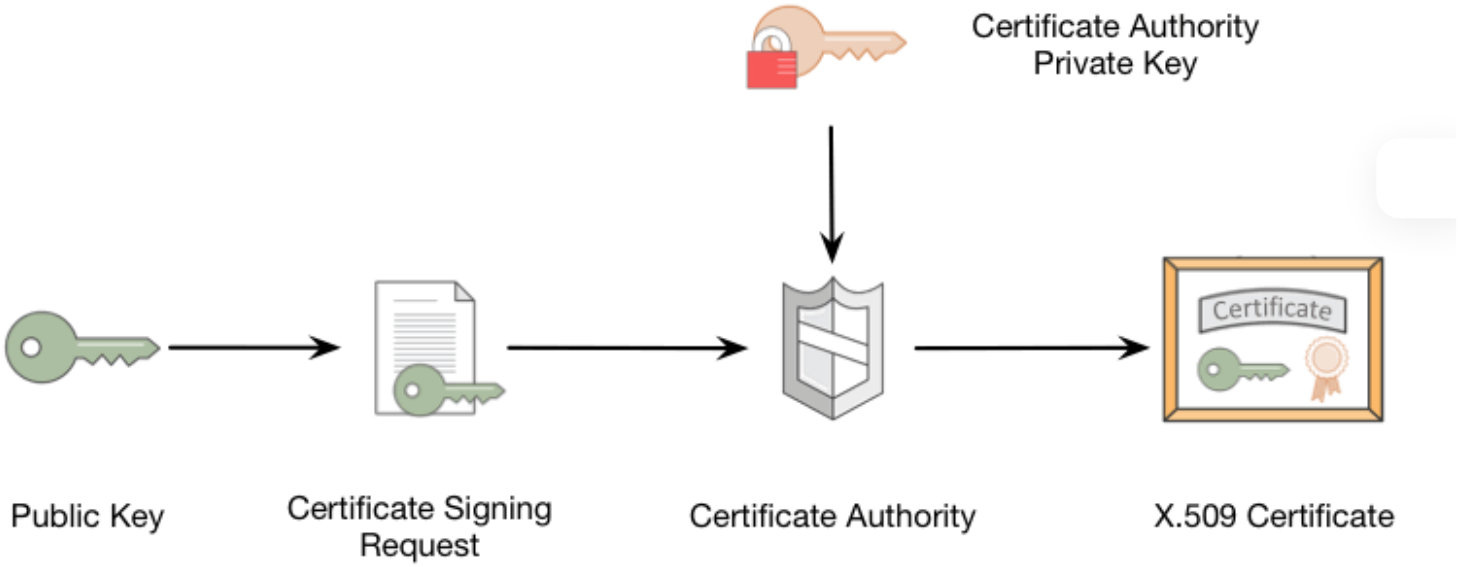
A key pair is a great way for others to send you secret data: if you keep your private key secure, anyone with access to the public key can send you an encrypted message that only you can decrypt and read.

In addition, public and private keys also allow you to sign documents. Here, a private key is used to add a digital signature to a message. Anyone with the public key can check the signature and know the original message hasn't been altered:

In addition to demonstrating a message hasn't been tampered, a digital signature can be used to prove ownership of a private key. Anyone with the public key can verify a signature and be confident that when the message was signed the signer was in possession of the private key.

Create an Identity

An X.509 certificate is a document that is used to prove ownership of a public key. To make a new X.509 certificate you need to create a Certificate Signing Request (CSR) and give it to a Certificate Authority (CA). The CSR is a digital document that contains your public key and other identifying information. When you send a CSR to a CA it first validates that the identifying information you've supplied is correct, for example you may be asked to prove ownership of a domain by responding to an email. Once your identity has been verified, the CA creates a certificate and signs it with a private key. Anyone can now validate your certificate by checking its digital signature with the CA's public key.



At this point you may be wondering why you should trust the CA and how you know the public key it gave you is genuine. The CA makes it easy to prove the ownership of its public key by publishing it in an X.509 certificate. The CA's certificate is itself signed by another CA. This sets up a chain of trust where one CA vouches for another. This chain goes back until a self-signed root certificate is reached.

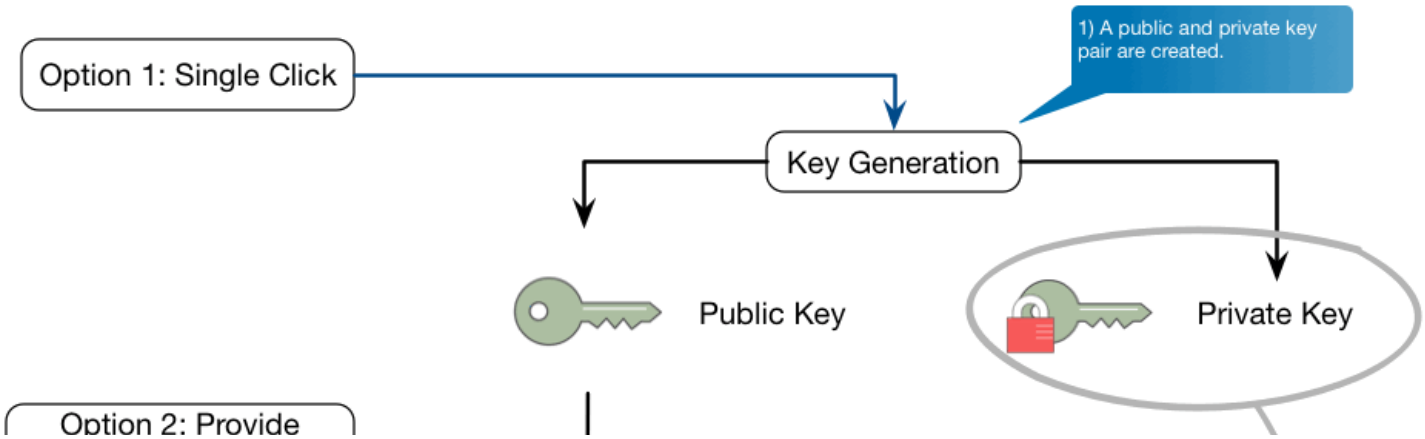
There are a small number of well-known root certificates. For example you can find lists of certificates that are installed in MacOS Sierra or available to Windows computers as part of the Microsoft Trusted Root Certification Program (free TechNet account needed to view). The chain of trust allows anyone to check the authenticity of any certificate by examining it all the way to a well-known, trusted root certificate:

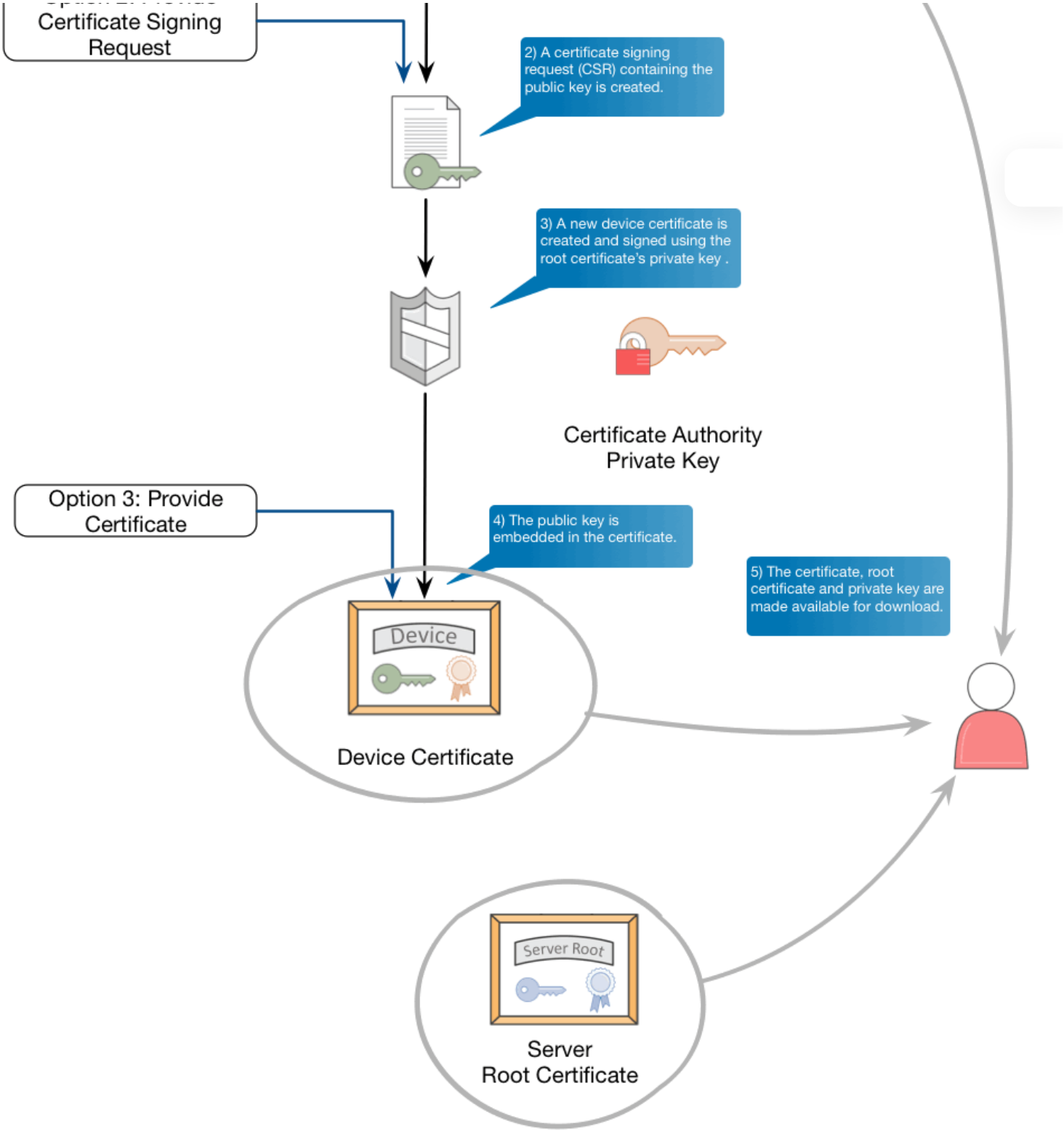
Root Certificate:
- Self-Signed, well known
- Has Root Certificate public key
- Signed by Root Certificate private key

Certificate 1:
- Has own public key
- Signed by Root Certificate private key
- Use Root Certificate public key to prove authenticity

Certificate 2:
- Has own public key
- Signed by Certificate 1 private key
- Use Certificate 1 public key to prove authenticity

Since each of your pizza order buttons will need a separate identity, you will need an X.509 certificate for each device. The diagram below shows how a new X.509 certificate is made for a device by AWS IoT. When creating a new certificate, you have three choices. The easiest (option 1 below) is to use the one-click generation. Here, AWS will create a public and private key and follow the process through to create a new certificate signed by the AWS IoT CA. The second option is to provide your own CSR. This has the advantage that you never give AWS sight of your private key. As with option 1, the new certificate generated from the CSR is signed by the AWS IoT CA. The final option is to bring your own certificate signed by your own trusted CA. This choice is best if you already generate your own certificates as part of your device manufacture or you already have a large number of devices in the field. You can find out more about using your own certificates in this blog post.





At the end of this you should be in possession of both the new device certificate and its private key. Whether you need to download these from AWS depends whether you choose option 1 (you need to download the certificate and the private key), option 2 (you just need to download the certificate) or option 3 (you already have both the certificate and the key, so don't need to download anything).

At this point, you also need to get a copy of the root certificate used by the AWS IoT server. As you will see below, this is important when establishing an authenticated link with the AWS IoT service.

All three files (the private key, the device certificate and the AWS IoT server certificate) need to be put onto your pizza ordering button. Note that if you are using an AWS IoT Button, you don't need to put the root certificate onto the device explicitly because it was put onto the device for you when it was manufactured.

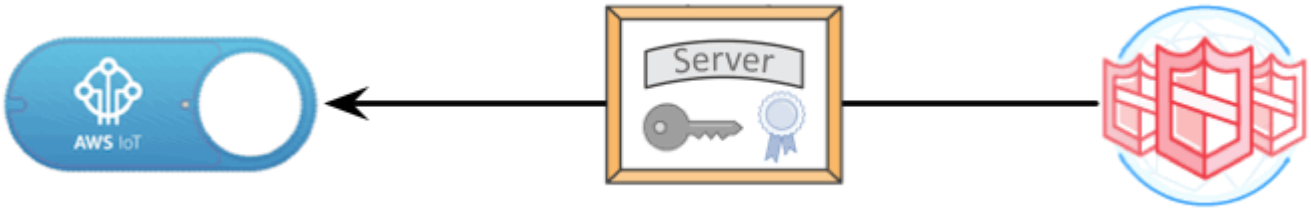
Authenticating to AWS IoT

Now that the certificates and private key are on our AWS IoT Button, you are ready to establish a connection to AWS IoT and authenticate. The protocol used is Transport Layer Security (TLS) 1.2, which is the successor to Secure Sockets Layer (SSL). This is the same protocol that you use to securely shop or bank on the internet but, in addition to server authentication, the client also uses a X.509 certificate prove its identity.

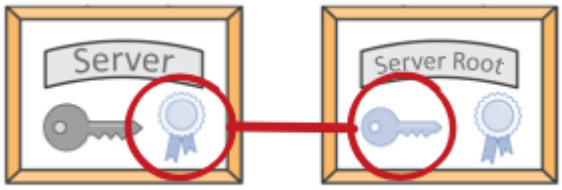
The connection starts with the AWS IoT Button contacting the Authentication and Authorization component of AWS IoT with a **hello** message:

The hello message is the start of a TLS handshake, which will establish a secure communication channel between the AWS IoT Button and AWS IoT. During the handshake, the client and server will agree on a shared secret, rather like a password, which will be used to encrypt all messages. A shared secret is preferred over using asymmetric keys as it less expensive in terms of computing power needed to encrypt messages, so you can get better communication throughput. The hello message contains details of the various cryptographic methods that the AWS IoT Button is able to use.

When the server receives a hello message it picks the cryptographic method it wants to use to establish the shared secret and returns this, together with its server certificate, to the AWS IoT Button:



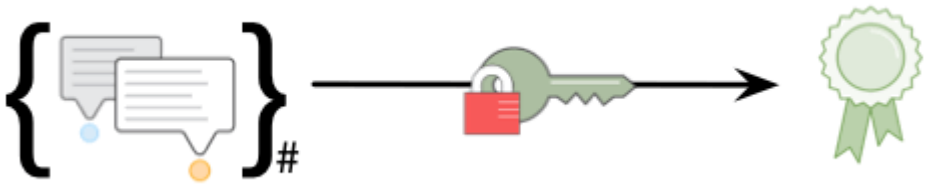
Now that the AWS IoT Button has a copy of the server certificate it can check that it is really talking to AWS IoT. It does that by using the AWS IoT Service root certificate, that you downloaded and put on the device. The public key that's embedded in the root certificate is used to validate the digital signature on the server certificate:



If the digital signature checks out with the root certificate's public key then the AWS IoT Button trusts that it has made contact with the AWS IoT service. It now needs to do two things; first it needs to authenticate itself with AWS IoT and second it needs to establish a shared secret for future communication.

To authenticate itself with AWS IoT, the AWS IoT Button first sends a copy of its device certificate to the server:

To complete the authentication process, the AWS IoT Button calculates a hash over all the communication records that are part of the current session with the AWS IoT Server. It then calculates a digital signature for this hash using its private key:



The digital signature is then sent to AWS IoT.

AWS IoT is now in possession of the devices' public key (which was in the device certificate) and the digital signature. Whilst the TLS handshake has been proceeding, the AWS IoT Service has also been keeping a record of all communication and calculates the same hash as the AWS IoT Button. It uses the device's public key to check the accuracy of the digital signature:

If the signature checks out, AWS IoT can be confident that it is talking to a pizza ordering device belonging to one of your customers. By using the unique identifier of the certificate, it knows exactly which device is establishing a MQTT session.

The exact method by which a shared secret is established depends on the key exchange algorithm that the server and client agreed on at the beginning of the handshake. However, the process is started by the AWS IoT Button encrypting a message using the server's public key (which it got from the server's certificate). The message might be a pre-master-secret, a public key or nothing. This is sent to the server and can be decrypted using the server's private key. Both the server and the AWS IoT Button then use the contents of the message to establish a shared secret without needing further communication. From then on, all messages between the device and AWS IoT are secured using the shared secret.

Permission to Order

The pizza order button has used its X.509 certificate to prove its identity and secure the messages it exchanges with AWS IoT. It is now ready to order pizza. Each AWS IoT Button publishes MQTT messages to its own topic, for example:

```
iotbutton/G03XXXXXXXXXXXX
```

The second part is the serial number of the AWS IoT Button. It's important that your system implements least privilege security and only permits an AWS IoT Button to publish to its own topic. For example, a nefarious customer could re-program their button to publish to a neighbor's topic. When the pizza turns up, it's simple social engineering to intercept the delivery and claim a free meal.

As you've seen, a device certificate is similar to a user's username; it's their identity. To give this identity permissions, you need to attach a policy to the certificate, in much the same way as you would attach permissions or policies to an IAM user.

The default policy for an AWS IoT Button is shown below. The default policy grants the owner of the certificate rights to publish to the topic specified in the 'Resource' attribute.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "iot:Publish",
      "Effect": "Allow",
      "Resource": "arn:aws:iot:eu-west-1:[redacted] topic/iotbutton/G03[redacted]
    }
  ]
}
```

In this policy, the serial number is hard-coded. This solution will not scale well as you will need a separate policy for each AWS IoT Button.

Fortunately, the policy language can help us with variable substitutions. For example, the following policy can be applied to all our devices. Instead of hard coding the serial number, the AWS IoT Service obtains it from the certificate that was used to authenticate the device. This assumes that when you created the certificate, the serial number was part of the identifying information.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish"
      ],
      "Resource": [
        "arn:aws:iot:eu-west-1:[redacted]:topic/${iot:Certificate.Subject.SerialNumber}"
      ]
    }
  ]
}
```

You can check out the documentation for further information on AWS IoT Policies and the substitution variables that you can use.

Summary

In this post, I have introduced you to the AWS IoT Security model and showed you how devices are authenticated against the service and how devices can be authorised to carry out actions.

You can purchase your own AWS IoT Button here or, if you plan a more sophisticated solution, you may want to check out this page that has lots of idea for getting started on AWS IoT, including some starter kits.

If you have any questions or suggestions, please leave a comment below.

(1) Gartner, Press Release, Gartner Reveals Top Predictions for IT Organizations and Users in 2017 and Beyond, October 2016, http://www.gartner.com/newsroom/id/3482117

TAGS: AWS IoT, IoT Button