TLS AND CRYPTOGRAPHY

Hardening TLS using ATECC508A

Prerequisites

- **Hardware Prerequisites**
    - AWS Zero Touch Secure Provisioning Kit
- **Software Prerequisites**
    - Windows 7 or above

Introduction

The world we live in is a connected world. Today we rely on our phones, computers and soon IoT devices to communicate, buy goods, travel and work. It is expected that connected devices or IoT devices, increase exponentially in the near future. The fact is that there are more phones than people today. The number of smart phones is measure by the billions and is increasing in a fast pace.

All these devices that are connected to the internet have one thing in common – They rely on the protocol called TLS (Transport Layer Security) to protect their information in transit.

TLS is a cryptographic protocol designed to provide secure communication over an insecure infrastructure. This means that, if this protocol is properly deployed, you can open a communication channel to an arbitrary service on the internet and be reasonably sure that you're talking to the correct server, and exchange information safely knowing that your data won't fall into the wrong hands and that it will be received intact.

These is not the case in the real world. Poorly designed systems along with software bugs can open a back door to an attacker. Aside from this, the simplicity of the RSA algorithm (which is widely used in most of the systems running TLS), has known weaknesses, such as the Private Key being stored in software. Anyone with the access to the corresponding server's private key can decrypt the communication between the client and it. This type of attack does not need to happen in real time. An attacker could stablish a long term operation and record all the encrypted traffic and wait until he obtains the Key. After the Key has been compromised, it's possible to decrypt all previously recorded traffic.

As we will see in the following sections of this document ATECC508 will serve as a secure hardware storage device to store Keys. It will also provide the hardware acceleration functionality to preform ECDSA verify and ECDH(E) to stablish the TLS handshake and secure session establishment.

Table of Contents

Icon Key Identifiers

TIP	**Highlights Useful Tips and Techniques**
INFO	**Highlights Objectives to be Completed**
RESULT	**Highlights the Expected Result of an Assignment Step**
WARNING	**Indicates Important Information**
EXECUTE	**Highlights Actions to be Executed Out of the Target**

1 Networking Layers

The internet at its core is built on top protocols called Internet Protocol (IP) and Transmission Control Protocol. These are used to package data into small packages to be transport. Because the core protocol don't provide any security at all by themselves anyone with access to the communication link can gain full access to the data as well as change the traffic without detection.

When encryption is taken un to account, the attacker might be able to gain access to the encrypted data, but it wouldn't be able to decrypt it or modify it. To prevent impersonation attacks TLS, rely on another important technology besides cryptography called *Public Key Infrastructure* **PKI** which ensures that the traffic is sent to the correct recipient.

To have a better understanding where TLS fit, we will look at the Open Systems Interconnection (OSI) model. This is a conceptual model that can be used to explain network communications. All functionality is mapped in 7 different layers. The bottom layer is the closest to the physical communication link and at the top is the application layer.

#	OSI Layer	Description	Example protocols
7	Application	Application data	HTTP, SMTP, IMAP
6	Presentation	Data representation, conversion, encryption	SSL/TLS
5	Session	Management of multiple connections	-
4	Transport	Reliable delivery of packets and streams	TCP, UDP
3	Network	Routing and delivery of datagrams between network nodes	IP, IPSec
2	Data link	Reliable local data connection (LAN)	Ethernet
1	Physical	Direct physical data connection (cables)	CAT5

Figure 1-1. Open Systems Interconnection (OSI) model

TLS sits above the TCP but below the higher-level protocols such as HTTP.

2 TLS and Cryptography

Transport Layer Security (TLS) is a protocol that provides communication security to communications on the internet. It is the most widely security protocol used today. As we will see in subsequent sections TLS is composed of two layers: The TLS Handshake Protocol Layer that allows the Server and Client to authenticate each other, and the TLS Record Protocol Layer which provides connection security.

2.1 Transport Layer Security

As mentioned earlier, the TLS protocol protects the communication link or transport layer, which is where the name comes from.

Security is not the only goal of TLS. It actually has four main goals:

- **Cryptographic Security**

This is the main purpose of TLS, to enable secure communications between any two parties who wish to exchange information.

- **Interoperability**

It should be possible for programs and libraries to be created and are able to communicate with each other using common cryptographic parameters.

- **Extensibility**

TLS is effectively a framework for development and deployment of cryptographic protocol

2.2 Symmetric Encryption

Symmetric encryption is a method for obfuscation that enables secure transport of data over insecure communication channels. This method is also known as private-key cryptography by the fact that it uses the same cryptographic keys for encrypting the plain-text and decryption of the cipher-text.

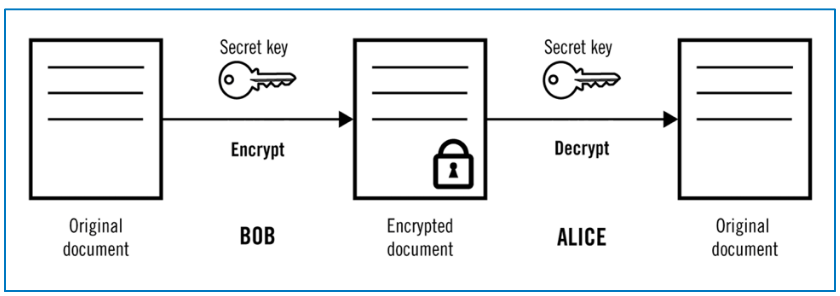

Figure 2-1. Bob and Alice share the same Private-Key for encryption and decryption

2.3 Asymmetric Encryption

Symmetric encryption does a great job t handling large amount dog data at great speeds, but it’s not that efficient as soon as the number of parties involved increases:

- Members of the same group must share the same key. The more people join in the group, the more expose the group is.

- A different key could be used for each person joining the group, but this collapses as the group gets bigger
- Symmetric encryption can't be used on unattended systems to secure data. This is do the fact that the process can be reversed by using the same key. A compromise to such a system leads to the compromise of all the data stored in the system.

Asymmetric encryption is a different approach to encryption that uses two keys instead of one. One of the keys is called a Private-Key and the other one is known as the Public-Key. As the name indicates one of the keys must be kept private and the other can be shared.

There's a special mathematical relationship between these keys that enables useful features such as sign-verify or encryption-decryption.

Asymmetric encryption makes communication in large groups easier. This is because you can share your public key widely and anyone can send you a message that you can read and also verify by them signing the message with their private key.

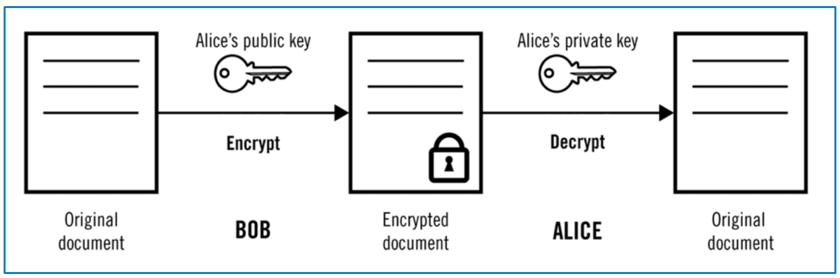

Figure 2-2. Bob uses a Public-Key to encrypt and Alice uses a Private-Key to Decrypt

2.4 Digital Signatures

Digital Signatures is a cryptographic scheme that allows us to verify the authenticity of a digital message or document. The Message Authentication Code (MAC) is a type of digital signature. A MAC is a cryptographic function that extend hashing with authentication, in other words, is a keyed-hash. Only does in possession of the hashing key can produce a valid MAC.

Digital signatures are possible with the help of public key cryptography. Its asymmetric nature can be exploit to device an algorithm that allows a message to be signed by a private key and be verified with its corresponding public key. The sign and verify process depends on the selected public key cryptosystem. For example, Elliptic Curve Cryptography (ECC), where the ECDSA cryptographic algorithm will sign a message using a private key and verify the message using its corresponding public key.

The process for signing a message is as follows

1. Calculate the hash of the document you want to sign using for example SHA256
2. Use ECDSA to sign the hash of the data with the private key
3. To verify the message, the public key associated with the private key needs to be send with the message and its signature
4. Used ECDSA to verify the message, having as inputs the Hash of the message, its signature and the public key.

3 TLS Record Protocol

TLS is a cryptographic protocol designed to secure communications that consist of an arbitrary number of messages between two parties. At a high level, TLS is implemented via the record protocol, which is in charge of the following aspects of the communication:

- **Message Transport**

  The record protocol transports opaque data buffers submitted to it by other layers in packets of max 16382 bytes. If a data payload is longer, it will split it into smaller chucks.

- **Encryption and Message Validation**

  Initially in a brand new connection, messages are transported without any protection. This is necessary so the first negotiation can take place. However, once the handshake is complete, the record layer starts to apply encryption and integrity validation according to the negotiated connection parameters.

- **Compression**

  This feature is no longer used.

- **Extensibility**

  The record protocol takes care of data transport and encryption, but delegates all other features to sub-protocols. This approach makes TLS extensible, because new sub-protocols can be added easily. With encryption handled by the record protocol, all sub-protocols are automatically protected using the negotiated connection parameters.

4 TLS Handshake Protocol

The TLS Handshake is the process during which the sides negotiate connection parameters and perform authentication. There can be many variations in the exchange, depending on the configuration and supported protocol extensions, but three are the ones that are most widely use:

- Full Handshake with server authentication
- Abbreviated handshake that resumes an earlier session
- Handshake with client and server authentication

In this document we will present the Full Handshake with Server Authentication

4.1 Understanding the TLS Handshake Procedure

Every TLS connection starts with a handshake. If the client hasn't previously established a session with the server, the two sides will execute a full handshake in order to negotiate a TLS session. During this handshake, the client and server will perform four main activities:

- Exchange capabilities and agree on desired connection parameters
- Validate the presented certificates or authentication
- Agree on a shared master secret that will be used to protect the session
- Verify that the handshake messages haven't been modified by a third party

In practice, steps 2 and 3 are part of a single step called **key exchange** or **session establishment**

This is the sequence followed by a complete TLS handshake with server authentication

1. Client begins a new handshake and submits its capabilities to the server
2. Server selects connections parameters
3. Server sends its certificate chain
4. Depending on the selected key exchange, the server sends additional information required to generate the master secret
5. Server communicates acceptable Certificate Public Key and Signature Algorithm
6. Server indicates completion of its side of the negotiation
7. Client sends Certificate Chain
8. Client sends additional information required to generate the master secret
9. Client proves the possession of the Private Key corresponding to the Public Key in previously sent Certificate by signing all the Hand Shake messages exchanged until this point
10. Client switches to encryption and informs the Server
11. Ready for Encrypted Application Data
12. Server switches to encryption and informs the Client
13. Ready for Encrypted Application Data

At this point the connection is established and the parties can begin to send application data securely.

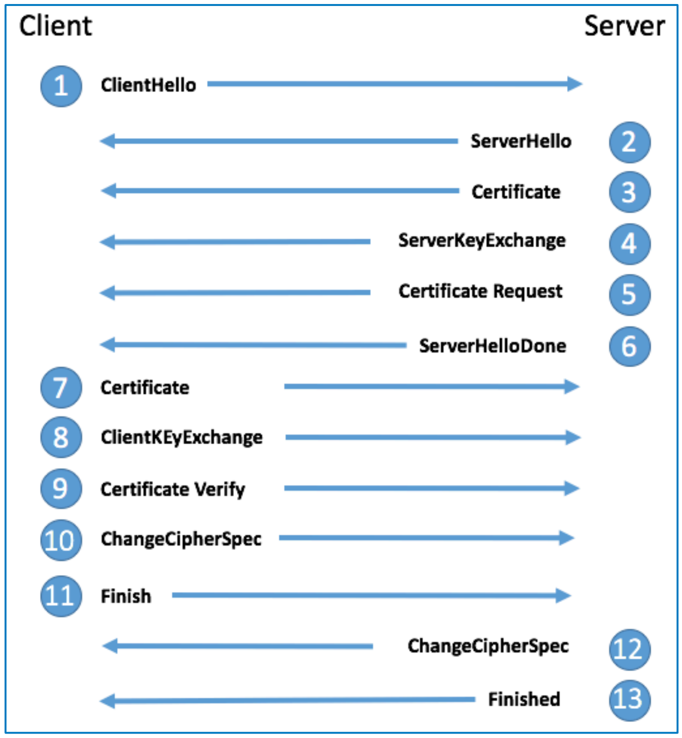

Figure 4-1. Full TLS handshake with both Client and Server Authentication

4.1.2 TLS Client Hello

The Client Hello message is always the first message sent in a new handshake. It's used to communicate the client capabilities and preferences to the server. Clients send this message at the beginning of a new connection. A TLS session empowered by the ATECC508, will have the Client Hello message that looks like the one below.

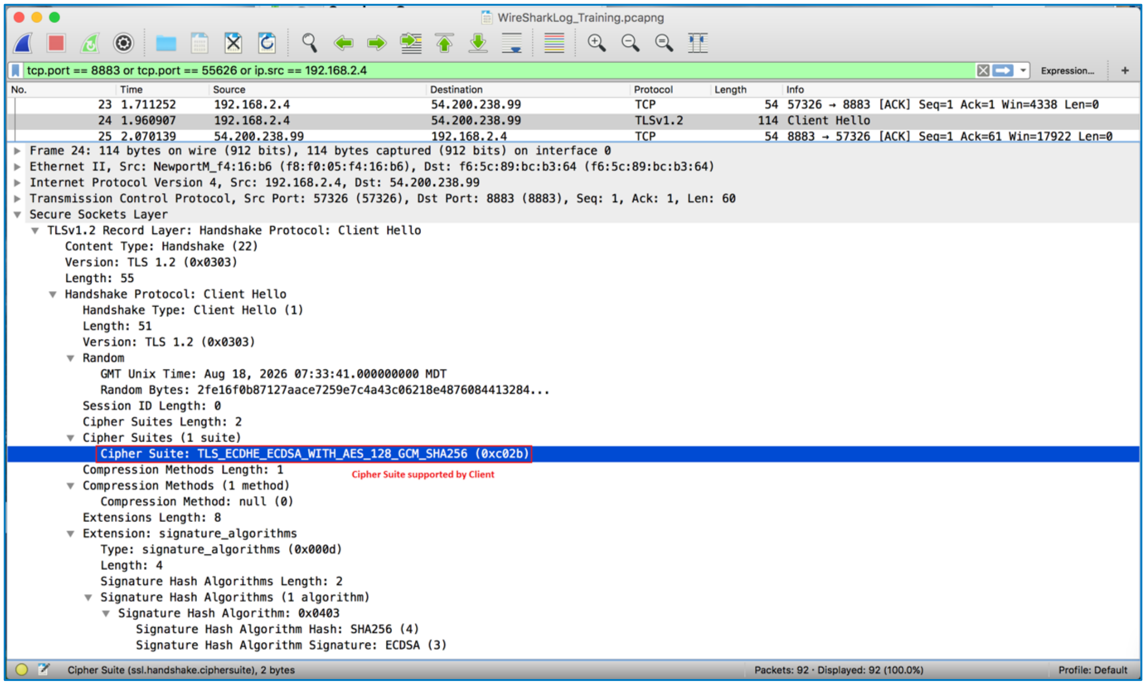

Figure 4-2. ECC508A TLS Client Hello Message

In this case the client sends to the server that its capabilities are:

- Elliptic Curve secp256r1
- ECDHE
- ECDSA
- AES128
- SHA256

Based on ECC508A capabilities (ECC p256, ECDHE, ECDSA and SHA256)

4.1.3 TLS Server Hello

The TLS Server Hello message will communicate the selected connection parameters back to the client. This message is similar in structure to Client Hello but contains only one option per field. A TLS session empowered by the ATECC508, will have the Server Hello message that looks like the one below.

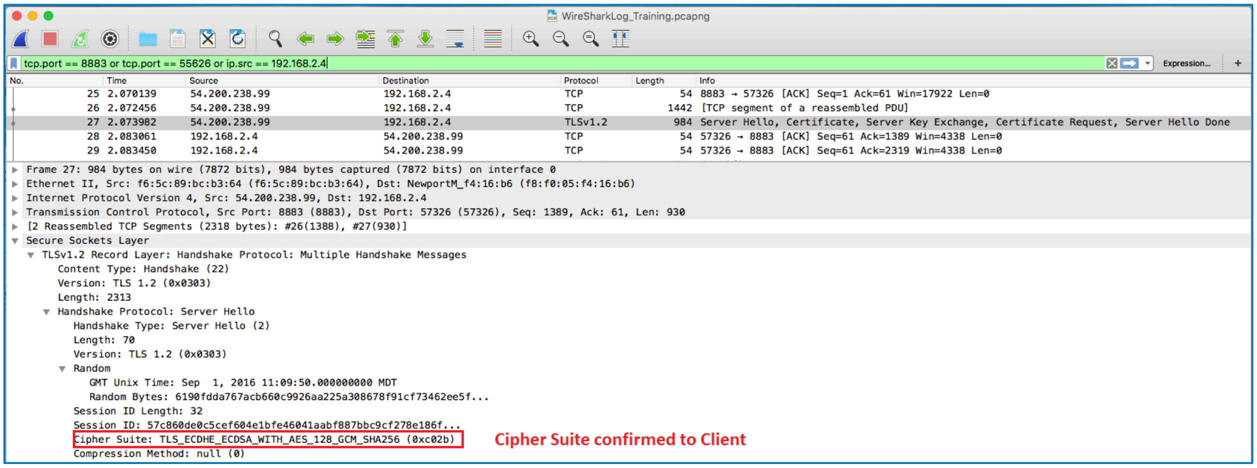

Figure 4-3. ECC508A TLS Server Hello Message

Now, the Server will acknowledge the same connection parameters that the Client requested

- Elliptic Curve secp256r1
- ECDHE
- ECDSA
- AES128
- SHA256

Based on ECC508A capabilities (ECC p256, ECDHE, ECDSA and SHA256)

4.1.4 TLS Certificate

In the TLS Certificate message, the Server will carry its X.509 certificate chain. These are provided one after another in **ASN.1 DER encoding**. The main certificate must be sent first, with all of the intermediate certificates following in the correct order. A TLS session empowered by the ATECC508, will have the Certificate message that looks look like the one below.

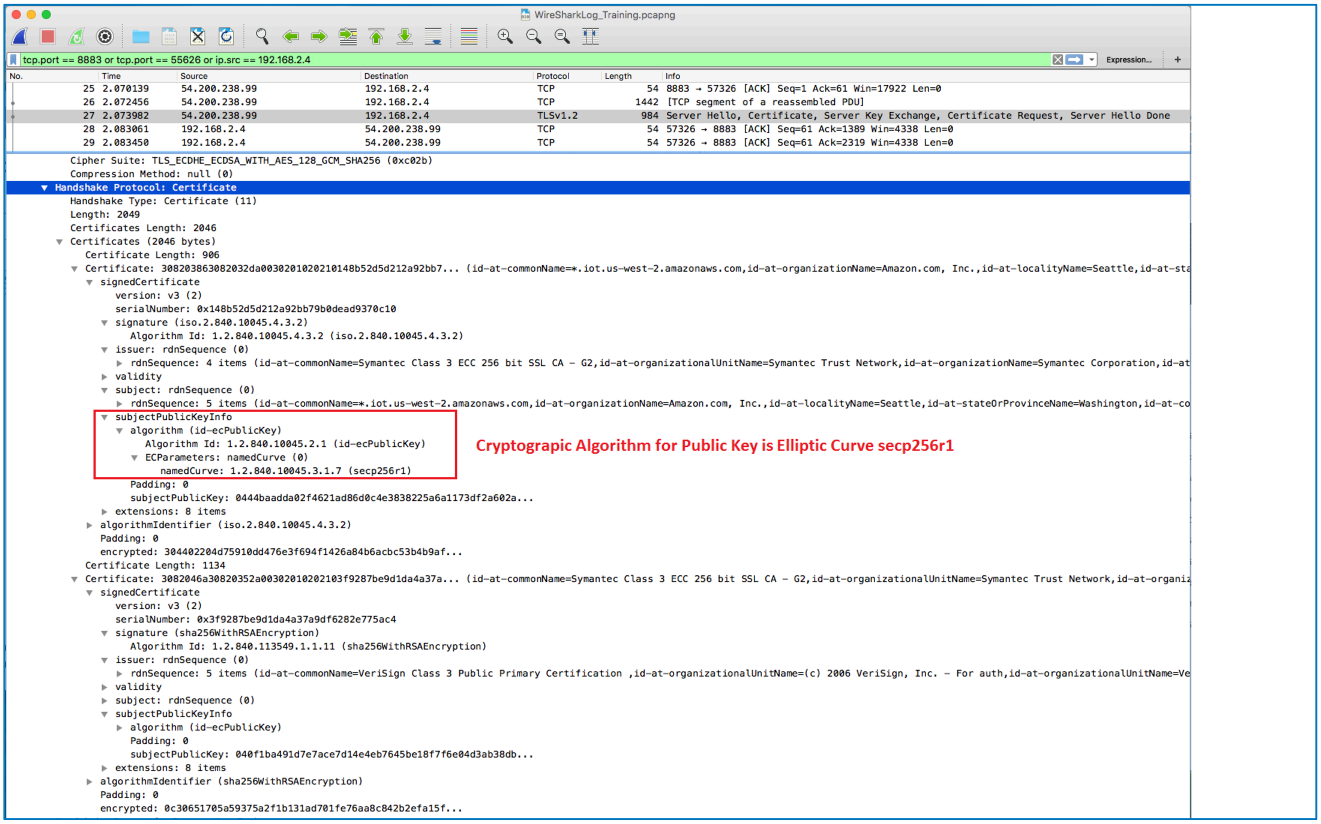

Figure 4-4. ECC508A TLS Certificate Message

4.1.5 TLS Server Key Exchange

The purpose of the Server Key Exchange message is to carry additional information that is needed for the key exchange. A TLS session empowered by the ATECC508, will have the Server Key Exchange message that looks like the one below.

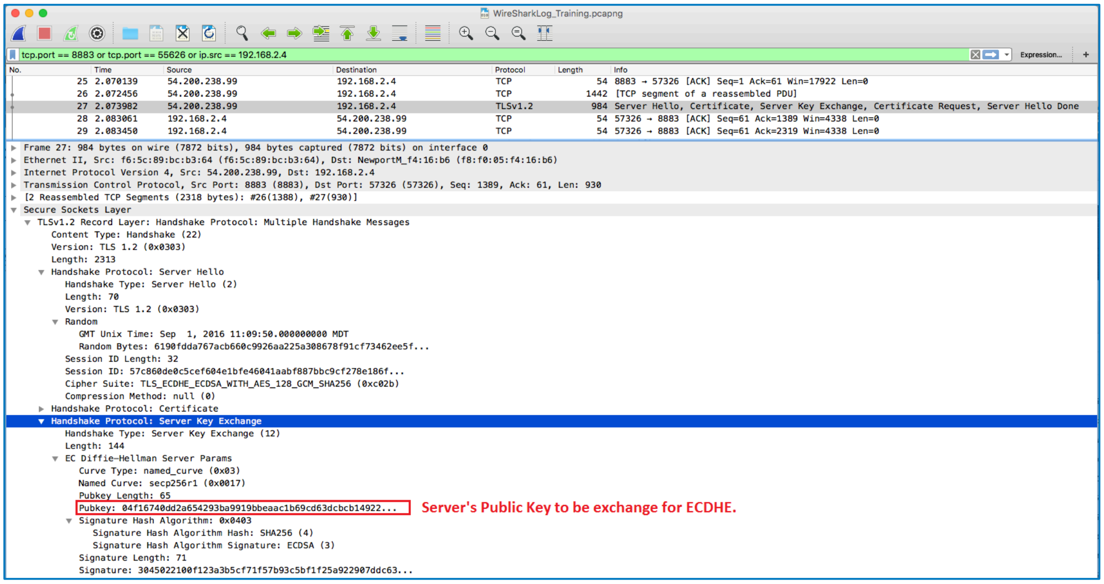

Figure 4-5. ECC508A TLS Server Key Exchange Message

In this case, the server will exchange its Public-Key with the Client. This will be used during the creation or the encryption key by applying ECDHE. This process takes place after the certificate chain has been verified.

4.1.6 TLS Certificate Request

By sending the TLS Certificate Request the Server request the client authentication and communicates acceptable Certificate, Public-Key and Signature algorithms to the Client. A TLS session empowered by the ATECC508, will have the Server Certificate Request message that looks like the one below

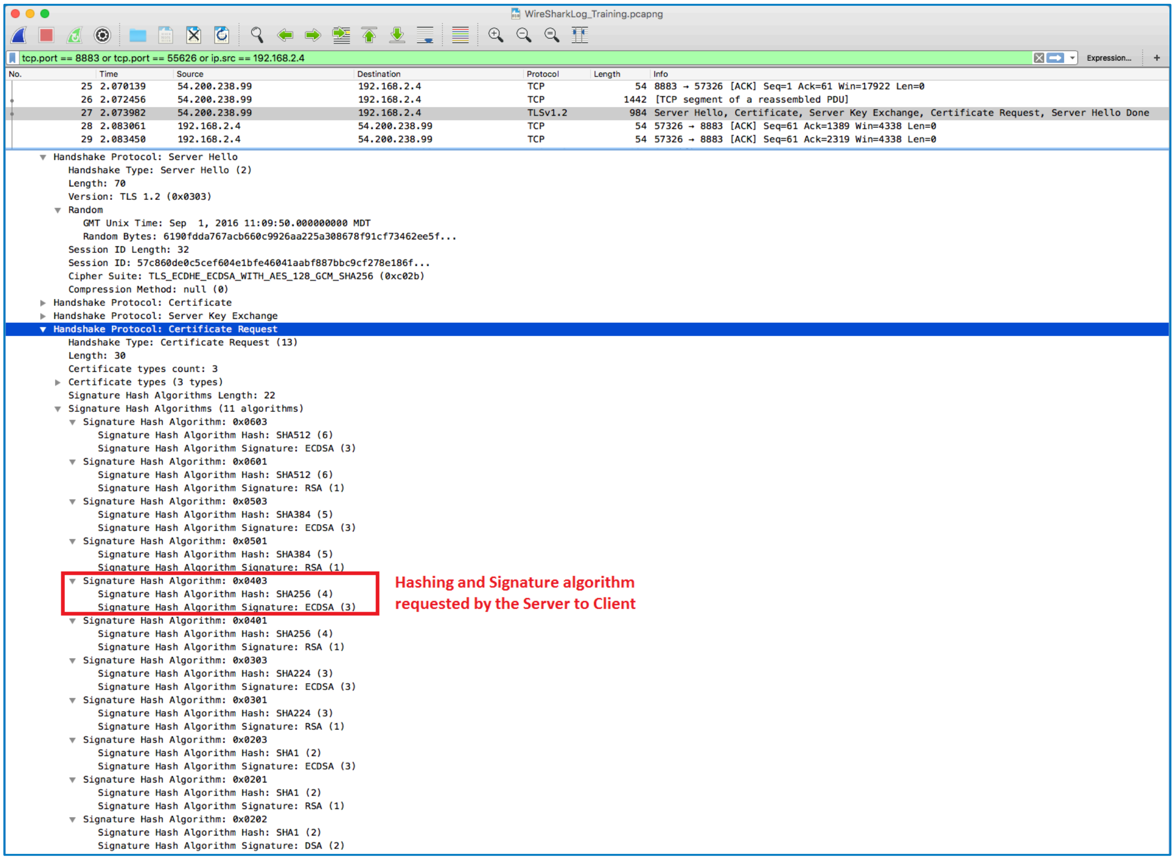

Figure 4-6. ECC508A TLS Server Certificate Request Message

4.1.7 TLS Hello Done

The Server Hello message indicates that the Server has sent all necessary messages for the Hand Shake to take place. The Server will now wait for further messages from the client.

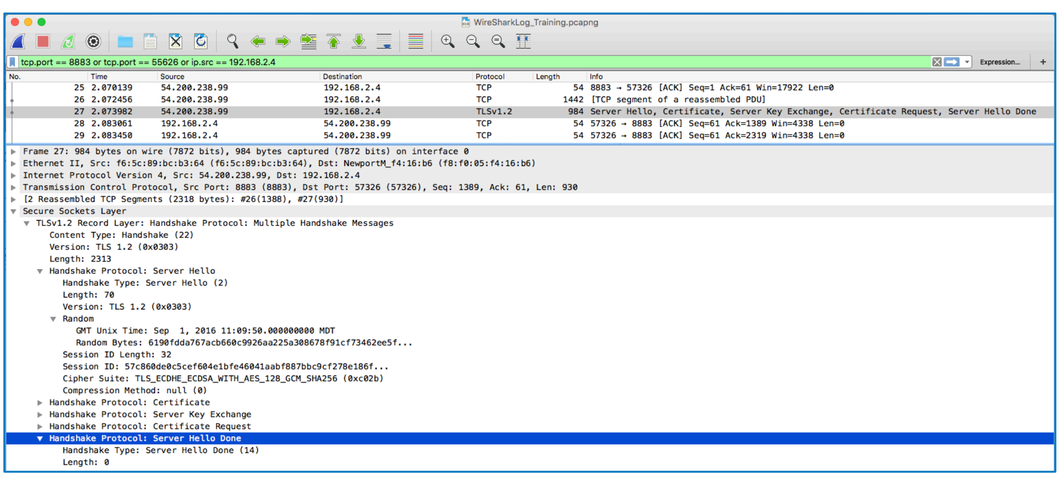

Figure 4-7. ECC508A TLS Server Hello Done Message

4.1.8 TLS Certificate

In the TLS Client Certificate Message, Client sends its Certificate chain to Server.

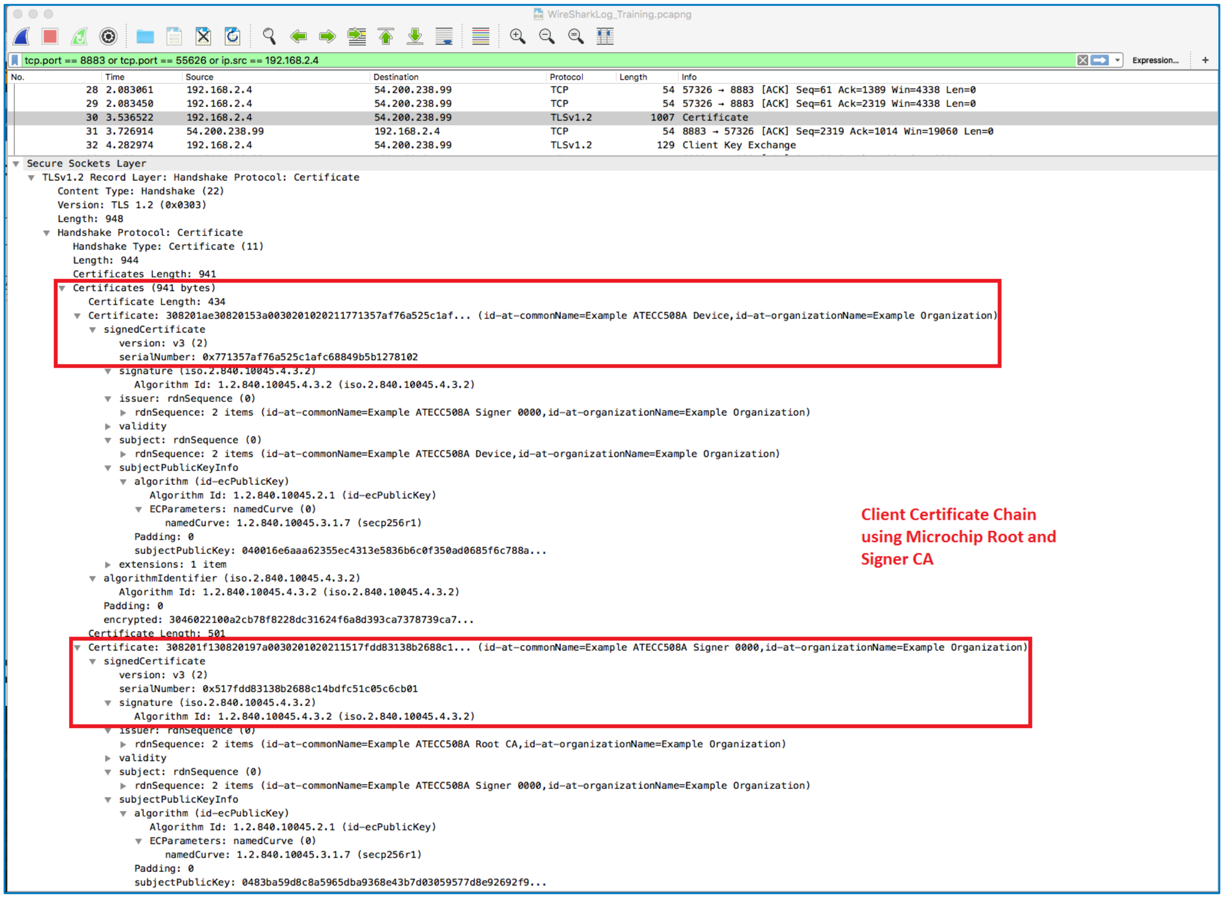

Figure 4-8. ECC508A TLS Client Certificate Message

4.1.9 TLS Client Key Exchange

The Client Key Exchange message carries the Client's contribution to the key exchange. Its content depends on the negotiated cipher suite. A TLS session empowered by the ATECC508 will have the Client Key Exchange message that looks like the one below

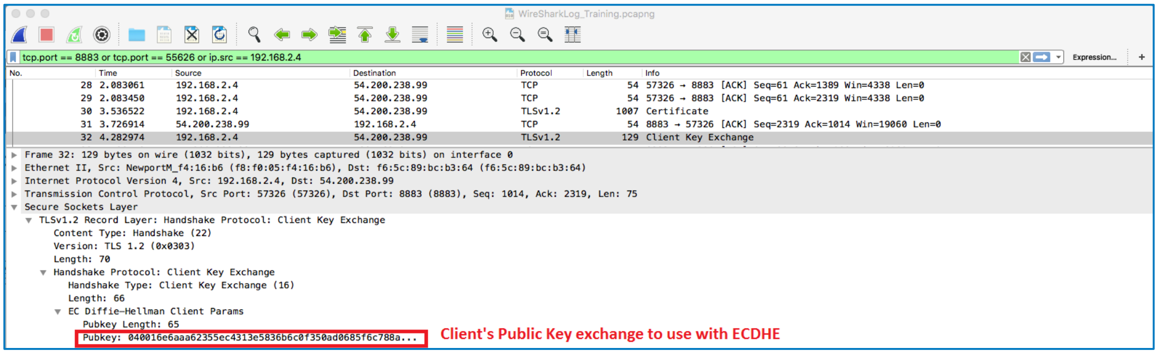

Figure 4-9. ECC508A TLS Client Key Exchange Message

4.1.10 TLS Certificate Verify

The Client uses the Certificate Verify to prove the possession of the Private Key corresponding to the Public Key in the previously sent Client Certificate. This message contains a Signature of all the handshake messages exchanged until this point. A TLS session empowered by the ATECC508 will have the Certificate Verify message that looks like the one below

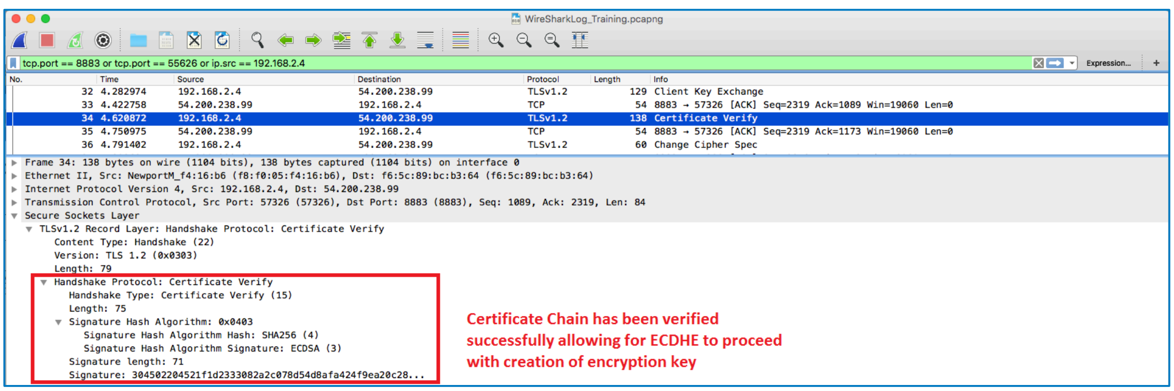

Figure 4-10. ECC508A TLS Verify Certificate Message

4.1.11 TLS Client Change Cipher Spec

The Client Change Cipher Spec message is a signal that the Client obtained enough information to create the connection parameters, generated the encryption keys and will now switch to encrypted communications.

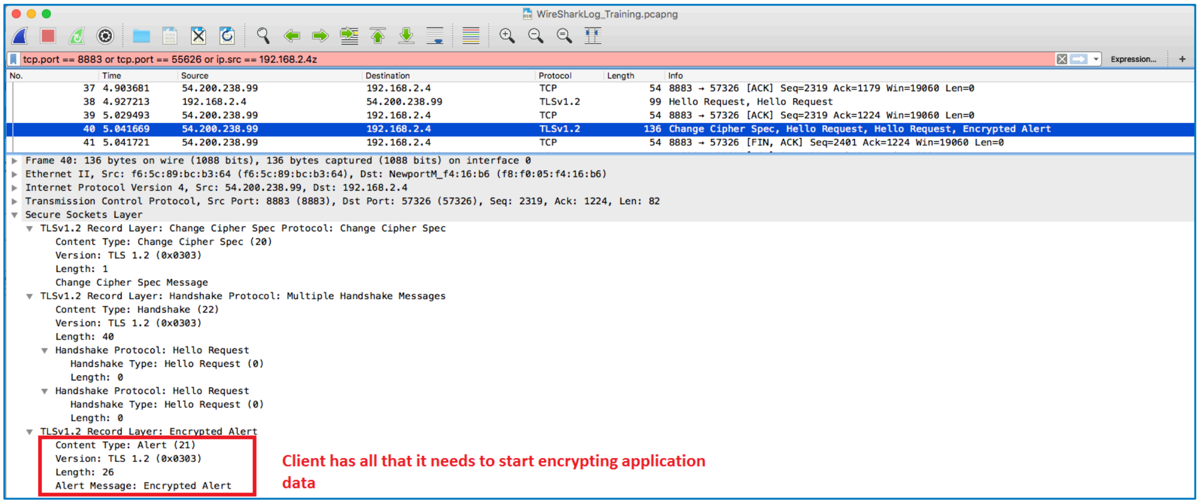

Figure 4-11. ECC508A TLS Client Change Cipher Spec Message

4.1.12 TLS Client Finish

The Client Finish message is a sign that the Handshake process is complete from the Client's perspective

4.1.13 TLS Server Change Cipher Spec

The Server Change Cipher Spec message is a signal that the Server obtained enough information to create the connection parameters, generated the encryption keys and will now switch to encrypted communications.

4.1.14 TLS Server Finish

The Server Finish message is a sign that the Handshake process is complete from the Server's perspective.

4.1.15 TLS Encrypting Message

All application data shared from this point forward will be encrypted data.

5 PKI and ECC508

5.1 Public Key Infrastructure

The Public Key Infrastructure (PKI) was created so we can communicate with someone that we have never meet before securely, just by sharing its Public Key. The PKI model relies on a trusted entity known as Certificate Authority which job is to issue trusted Certificates.

This certificate issued by a Certificate Authority will become the systems Root of Trust. The Root of Trust will extend a certificate to the OEM which will turn in to the OEM-CA.

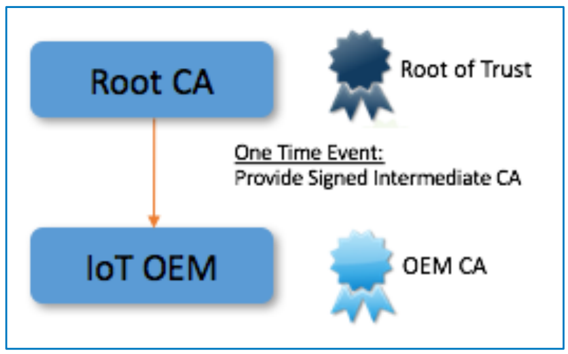

Figure 5-1. Establishing your PKI infrastructure

The OEM-CA will create an intermediate CA or Production Signer. The end device will request a certificate issuance (CSR) to the Signer.

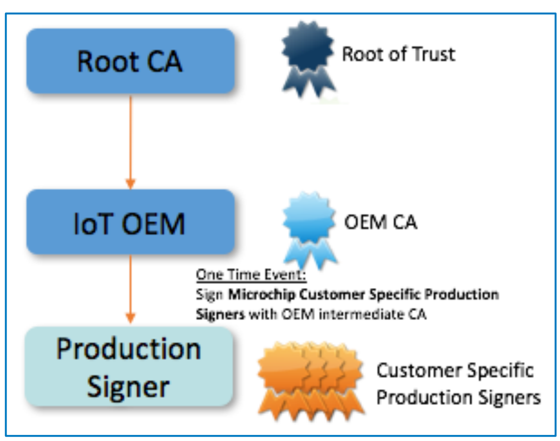

Figure 5-2. Creating a Production Signer that will sign the End Device CSR's

The production signer will sign the end device CSR, and thus will give an identity to the end device.

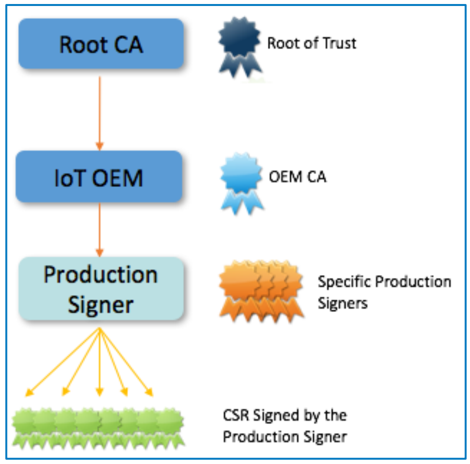

Figure 5-3. The Production Signer will now Sign the CSR's

5.2 Using ECC508A in a Public Key Infrastructure

The ECC508A is a device that can store Private Keys securely and use these private Keys internally in cryptographic algorithms to support a PKI. The cryptographic algorithms that ECC508A supports are the Elliptic Curve p256, Elliptic Curve Digital Signature Algorithm (ECDSA) and the Elliptic Curve Diffie-Hellman algorithm (ECDH(E)) and SHA256.

- ECDSA
  - It's a signing algorithm that uses elliptic curve cryptography.
- ECDH
  - It's a Key exchange agreement that allows two parties, each having an elliptic curve public-private key, to establish a common key to secure their communications.
- Elliptic Curve p256 Key Generator
- RNG (FIPS Compliant)
  - Random Number Generator
- SHA256
  - SHA256 Hashing algorithm engine

The ECC508A will also allow you to store compressed X.509 certificates that the Host can decompress.

These features allow the ECC508A to be used as a cryptographic coprocessor for stablishing a Public Key Infrastructure and also be used in a TLS session establishment between a Host and a Client.

The PKI in its most complete form using an ECC508A device to store the Private Keys and its relevant compressed certificates would look like this.

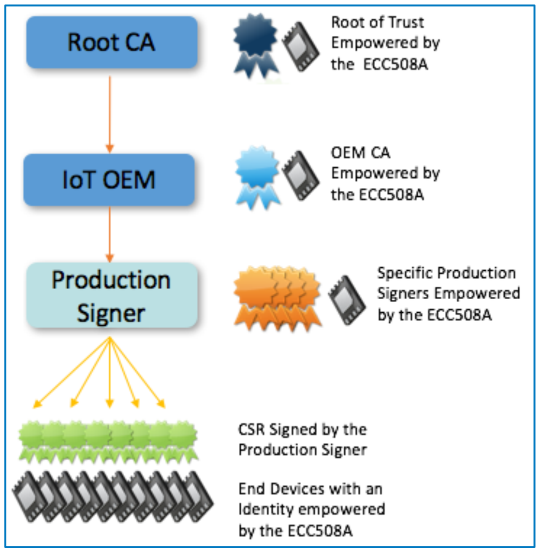

Figure 5-4. A Public Key Infrastructure empowered by the ECC508A

6 TLS and ECC508

As mentioned in the previous section the ECC508A will act as a cryptographic coprocessor and HW key storage device. During the provisioning phase of the ECC508A, it will create its Elliptic Curve Private Key and lock the slot where it is stored. Once the ECC508A device is configured and locked the secret keys that were stored in the device will never leave it. All the cryptographic operations where this private key is needed will happen within the device itself.

By following this approach, the level of security that TLS offers increases since the private keys will never be exposed in software.

During the TLS handshake and session establishment process the ECC508A will

- Perform the ECDSA verification of the Certificates being exchanged.

- It will use ECDH to generate a shared key that can be used to established a secure session between the Server and the Client

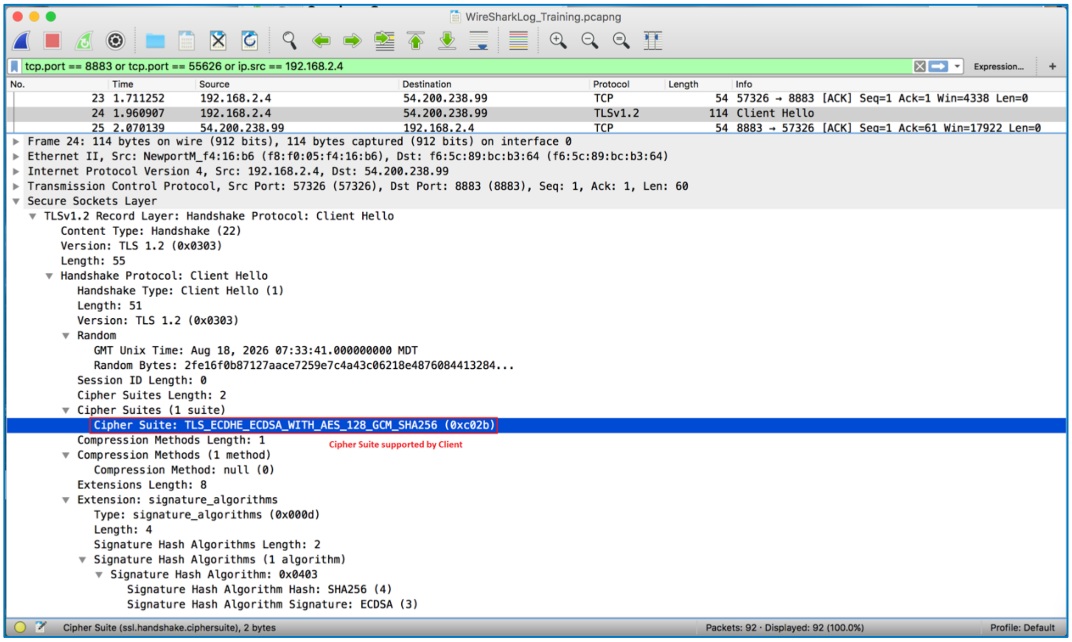

Figure 6-1. Using ECC508A ECDSA and ECDH to establish a TLS session.

For a detailed process on how the TLS handshake and session establishment works, please refer to section 4 of this document.

7 Connecting to AWS IoT using ATECC508A

Now that we understand how the ATECC508A can be used to harden the TLS session establishment during the handshake and secure public key exchange, we will use it to connect the device to AWS IoT securely. The steps that we will follow basically are:

- Creating the Root CA that will sign our production Signer using our Root Module.
- Register our Signer Certificate with AWS Server by using the Bring Your Own Certificate (BYOC) capability of AWS IoT.
- Establish a secure TLS session using ECC508A to register our IoT device by using the Just In Time Registration capability (JITR) of AWS IoT.
- Securely exchange MQTT messages between our IoT device and AWS IoT once the TLS session has been created.

We have created a GitHub repository containing the necessary information on how to run this hands-on. It has also links to AWS documentation that will support this application note if more detailed information is needed

MicrochipTech Secure Insight on Things GitHub repository

https://github.com/MicrochipTech/AWS-Secure-Insight

7.1 Creating the Root of Trust and Production Signer

From section 5, we know that:

- The purpose of a **Root of CA** is to give authority to the Production Signer to sign the IoT device.
    - The Root CA is basically signing the Production Signer's certificate using its Private Key.
    - It will allow the user to stablish another layer of security to its network by securing the application layer and be able to verify the PKI chain back to it.

In our AWS Zero Touch Provisioning kit you will find a Root Module that we have provided for the purpose of creating a Root CA.

Figure 7-1. Root Module. Contains an ATECC508 configured to be used as a Root CA

- The purpose of the **Production Signer** is to validate the identity of the IoT device.
    - The Identity of the IoT device will be given when the Production Signer signs the End Node's certificate using its Private Key

In our AWS Zero Touch Provisioning kit you will find a Signer Module that we have provided for the purpose of creating the Production Signer.

Figure 7-2. Signer Module. Contains an ATECC508 configured to be a Signer CA

The creation of the Root CA and the Production Signer and will be done by the GUI in interaction with the Root and Signer Modules. It is the first step that the GUI takes during the process of registering the Production Signer into AWS IoT. Let's review this process in the next section.

7.2 Registering our Production Signer with AWS IoT

As we have mentioned in the previous sections, our objective is to have IoT devices (end nodes), that will have an identity that AWS IoT can trust. The way we achieve this is by registering our Production Signer CA certificate with AWS IoT along with a verification certificate which will provide the means to verify that you have access to both the Production Signer CA and the private Key associated to it.

To prove you have ownership of the Signer CA private key we generate a verification certificate using the Signer CA certificate, the verification code used to generate a verification certificate and sign it with the Signer CA private key.

Below is the sequence diagram that we will follow to register or Production Signer CA certificate in AWS IoT.

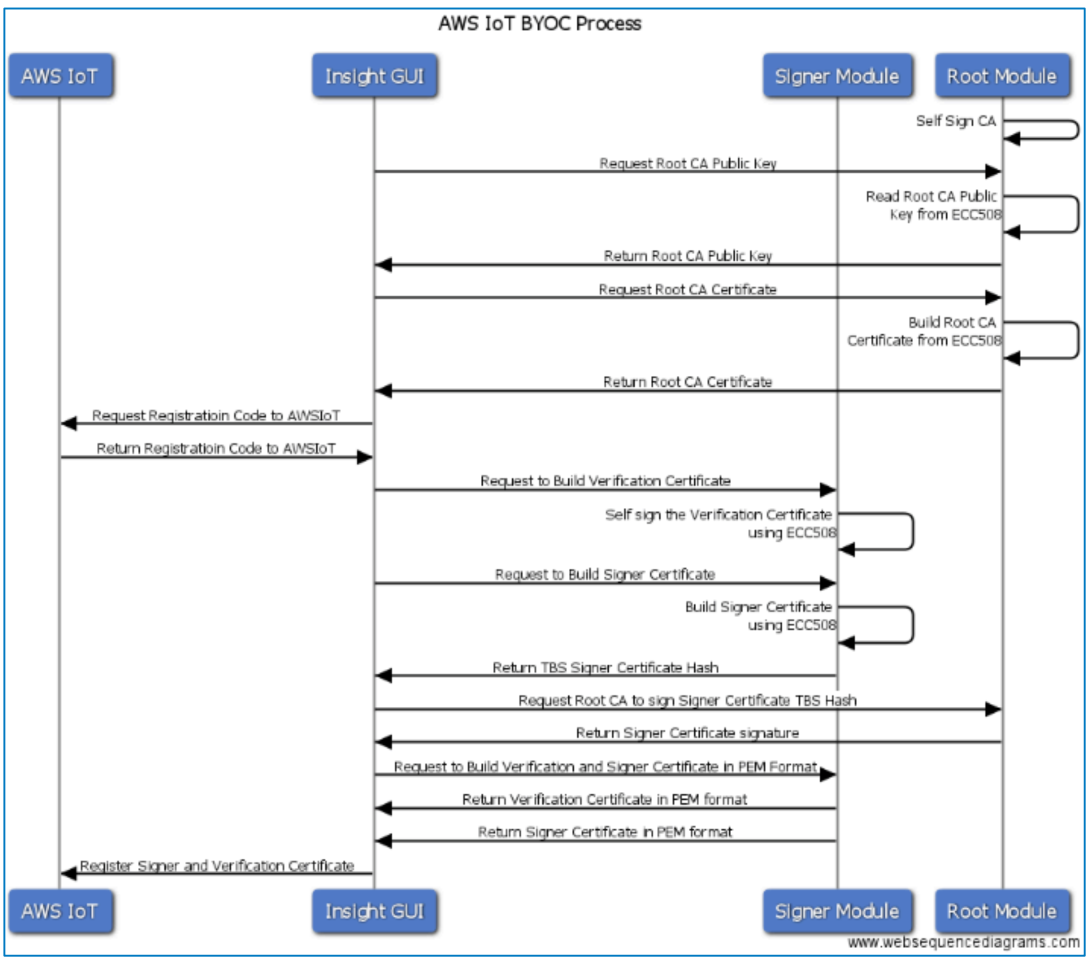

Figure 7-3. AWS IoT BYOC Registration process using the Insight GUI, Root and Signer modules

7.2.2 AWS IoT BYOC Hands On

The objective of this hands on will be to demonstrate the process that involves registering your Production Signer to AWS IoT by using the BYOC (Bring Your Own Certificate), functionality.

We will be using the following setup:

- AWS IoT device compose of:
    - ATSAMG55
    - Winc1500 (updated to FW 19.4.4)
    - CryptoAuthXplained Pro
- Secure Insight of Things GUI
- Root Module
- Signer Module

Setting-up your ATSAMG55 AWS IoT device

Our first task is to set up our G55 AWS IoT device. Take out from your **AWS Zero Touch Secure Provisioning kit** the following tools:

- AWS IoT Thing compose of:
    - ATSAMG55
    - Winc1500 (updated to FW 19.4.4)
    - CryptoAuthXplained Pro

Now assemble them as shown in the image below.

- Winc1500 will connect to EXT 1
- CryptoAuthXpro will connect to EXT 4
- OLED1 Explained Pro will connect to EXT3

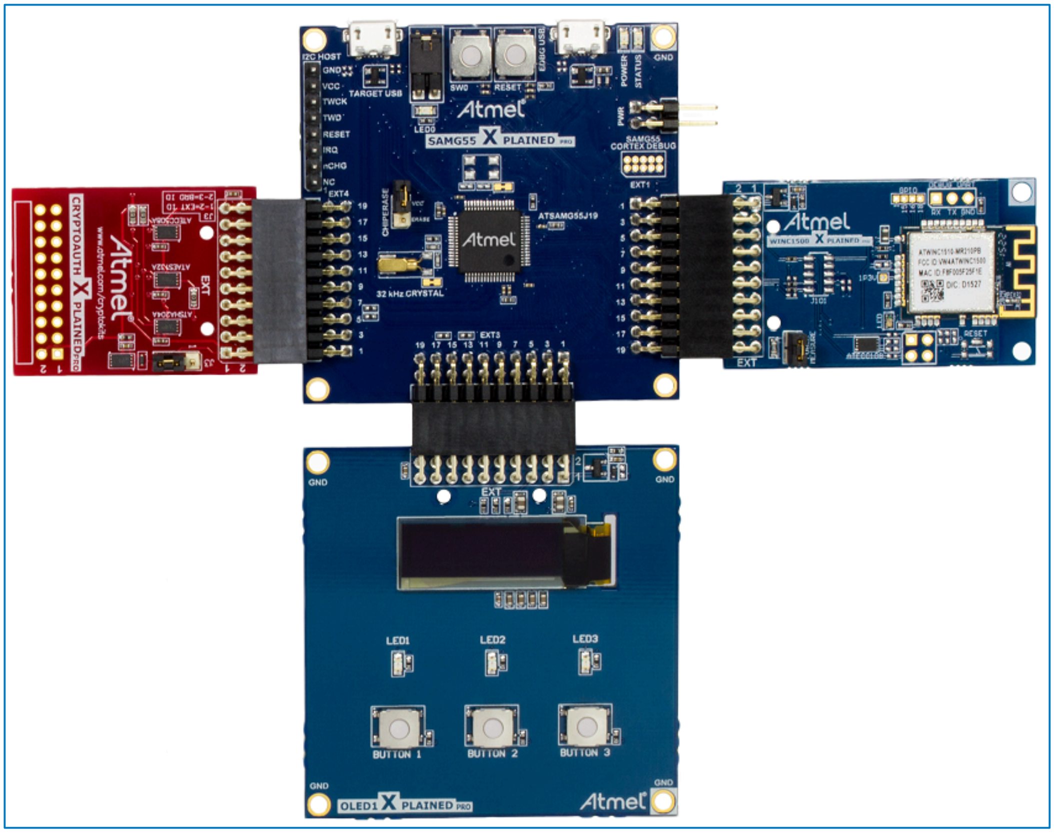

Figure 7-4. ATSAMG55 AWS IoT device

The **AWS Zero Touch Secure Development Kit** comes with a micro USB cable. Connect one end of the USB cable to the **USB Target** connector in ATSAMG55 setup, as shown in Figure 7-5.

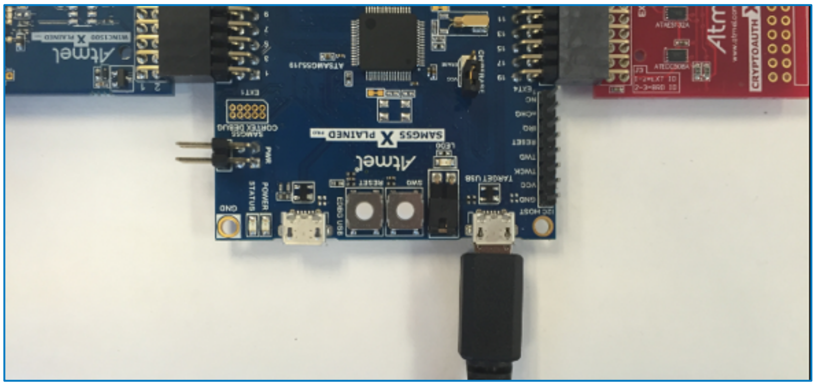

Figure 7-5. Connecting the USB cable to your ATSAMG55 setup.

Installing the AWS Secure Insight Application GUI

We will now download and install the **Installing the AWS Secure Insight on Things Application GUI** from GitHub. Open the link bellow.

https://github.com/MicrochipTech/AWS-Secure-Insight/blob/master/sw/GUI/Windows-x64/AwsSecureIoT.msi

Press the download button.

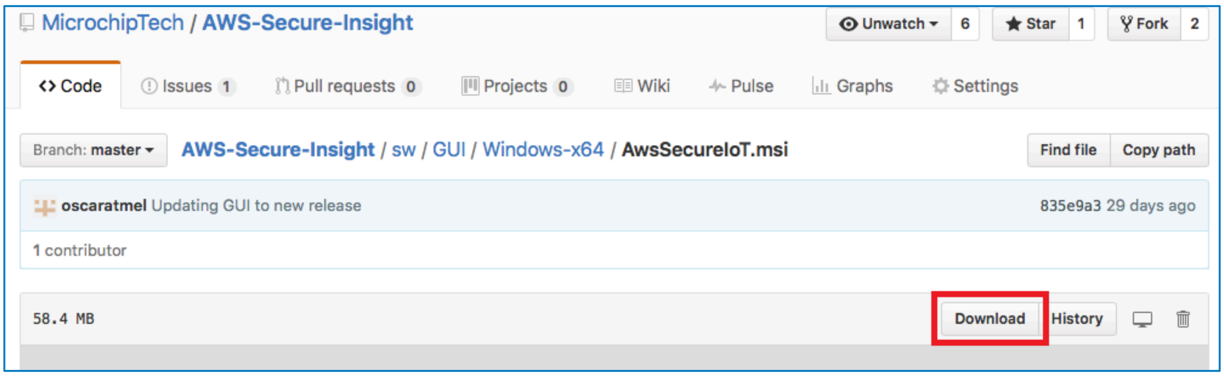

Figure 7-6. Downloading the AWS Secure Insight on Things Application GUI

Open the Installer and run. Follow the instructions from the wizard.

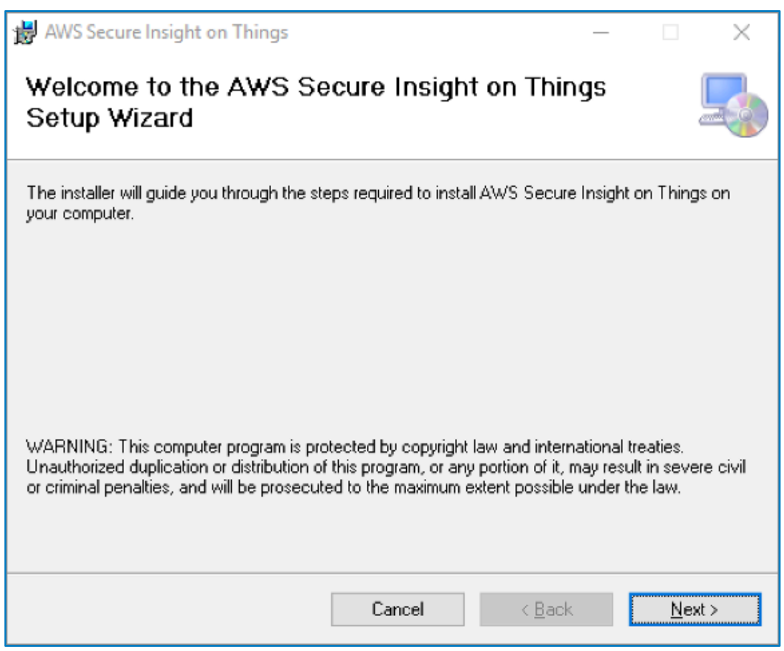

Figure 7-7. Installing the AWS Secure Insight on Things Application GUI

After the installation is completed you should see an icon in your desktop like the one bellow.

Figure 7-8. AWS Secure Insight GUI desktop shortcut

Connecting the Root and Signer Module and ATSAMG55

The **AWS Zero Touch Secure Development Kit** comes with a USB dongle equipped with an ECC508 configured as a Root Certificate Authority (Red Label) and Signer USB dongle (Green Label).

- Connect the two USB modules to your desktop.
- Connect the ATSAMG55 AWS IoT device to your desktop

Configuring the Secure Insight on Things GUI

Now that we have the USB Root and Signer Modules and the ATSAMG55 AWS IoT device connected to our desktop we will run the **AWS Secure Insight** GUI.

The first time you run the AWS Secure Insight GUI, a configuration window will appear like the one shown below in Figure 7-9.

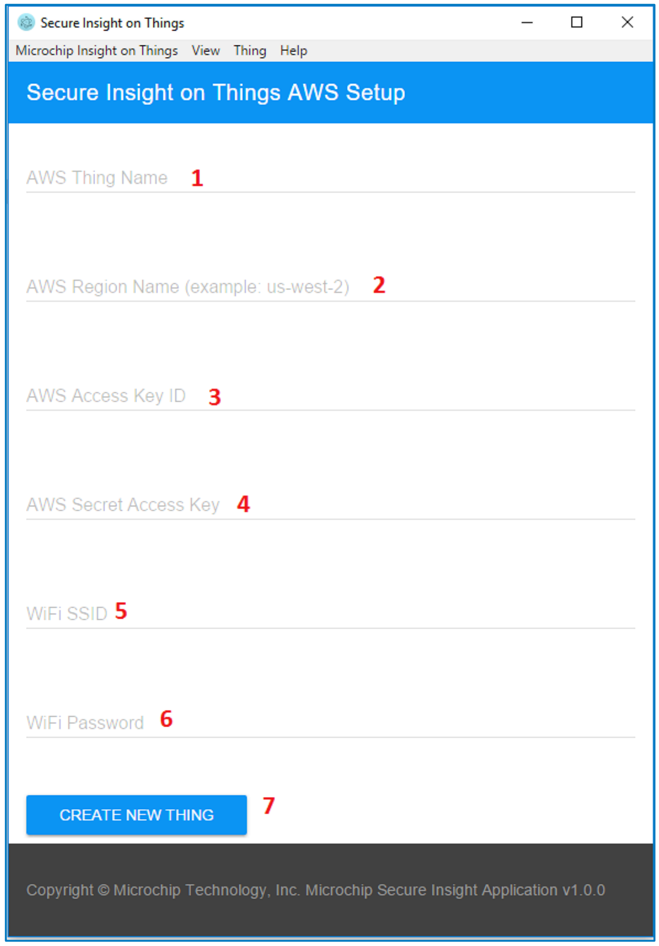

Figure 7-9. Secure Insight on Things configuration setup

The information that we need to enter in the configuration window is as follows

1. The name of you AWS IoT device
2. The AWS IoT Region Name we will be connecting to. Usually **us-west-2**
3. The AWS IoT Access Key ID. This is provided when you create an AWS IoT account.
4. AWS Secret Access Key. This is provided when create an AWS IoT account.
5. Your WiFi access point SSID
6. Your WiFi access point Password
7. After entering all the required information press the **Create New Thing Button**

Registering your Production Signer

We will now register our Production Signer using the **Secure Insight on Things GUI.**

Before we proceed we need to make sure that all of our devices (ATSAMG55, Root Module, Signer Module), are being detected. If they are, you should see them enumerated with in the GUI as shown below in Figure 7-10.

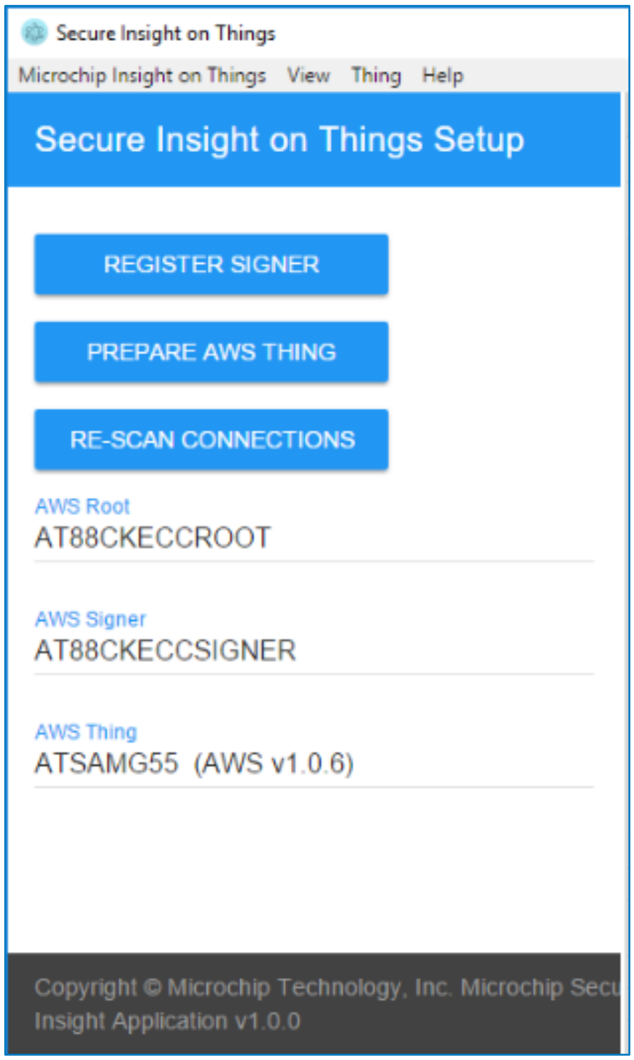

Figure 7-10. ATSAMG55, Root Module and Signer Module enumerated correctly

If you have all of the devices connected to your desktop and still do not see all of them showing up, go to **View** menu and select **Reload**.

In order to see the log of the Production Signer registration into AWS IoT, we will use as aid the console view with in our GUI. To enable this view, follow the next steps:

- Open the View menu.
- Select Toggle DevTools from the drop down menu.

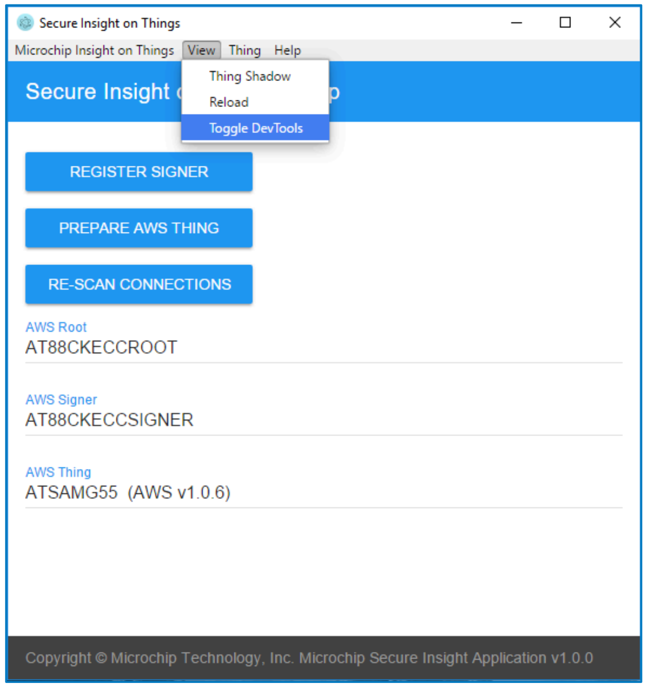

Figure 7-11. Enabling the Console view of the Secure Insight on Things GUI

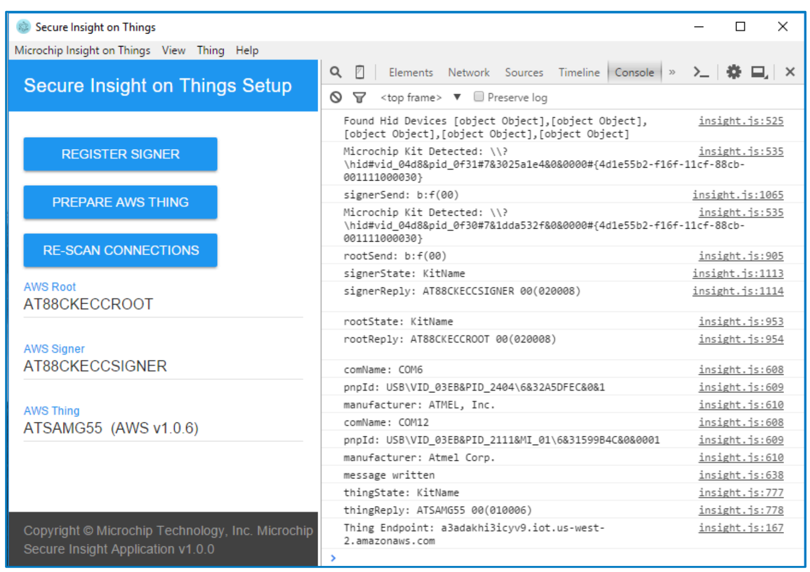

Figure 7-12. Console will show the ongoing process

From the new window pane that opened, select the Console option.

Before we start our Production Signer and Verification certificate registration we need to make sure that the initialization of the GUI was successful. We will know this because we would have received the AWS IoT device EndPoint which will be used during our process.

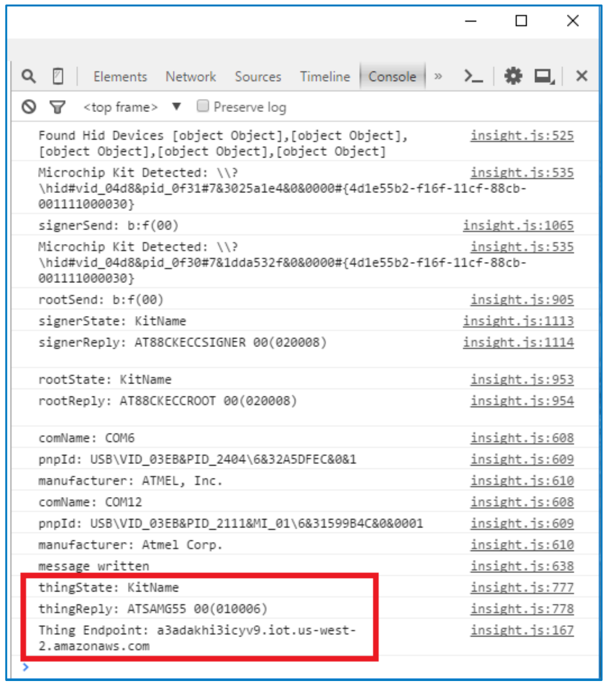

Figure 7-13. Secure Insight on Things GUI successfully initialized.

The Next steps will take place during the registration process of our Production Signer in to AWS IoT. To Start the process, press the "**Register Signer"** Button**.**

The following steps will take place during the registration processes as illustrated in the sequence diagram in Figure 7-3:

1. **Secure Insight GUI** will request the **Root CA's** public key to Root Module.
2. **Secure Insight GUI** will request the **Root CA's** certificate to the Root Module in PEM format.
3. **Secure Insight GUI** will request the **Verification Code** from AWS IoT.
4. **Secure Insight GUI** will request the **Signer Module** to build **Verification Certificate.**
5. **Signer Module** will self-sign the **Verification Certificate.**
6. **Secure Insight GUI** will request **Signer Module** to build the **Production Signer Certificate.**
7. **Secure Insight GUI** will request **Root Module** to sign the **Production Signer Certificate.**
8. Register **Production Signer** and **Verification Certificates** with AWS IoT in PEM format.

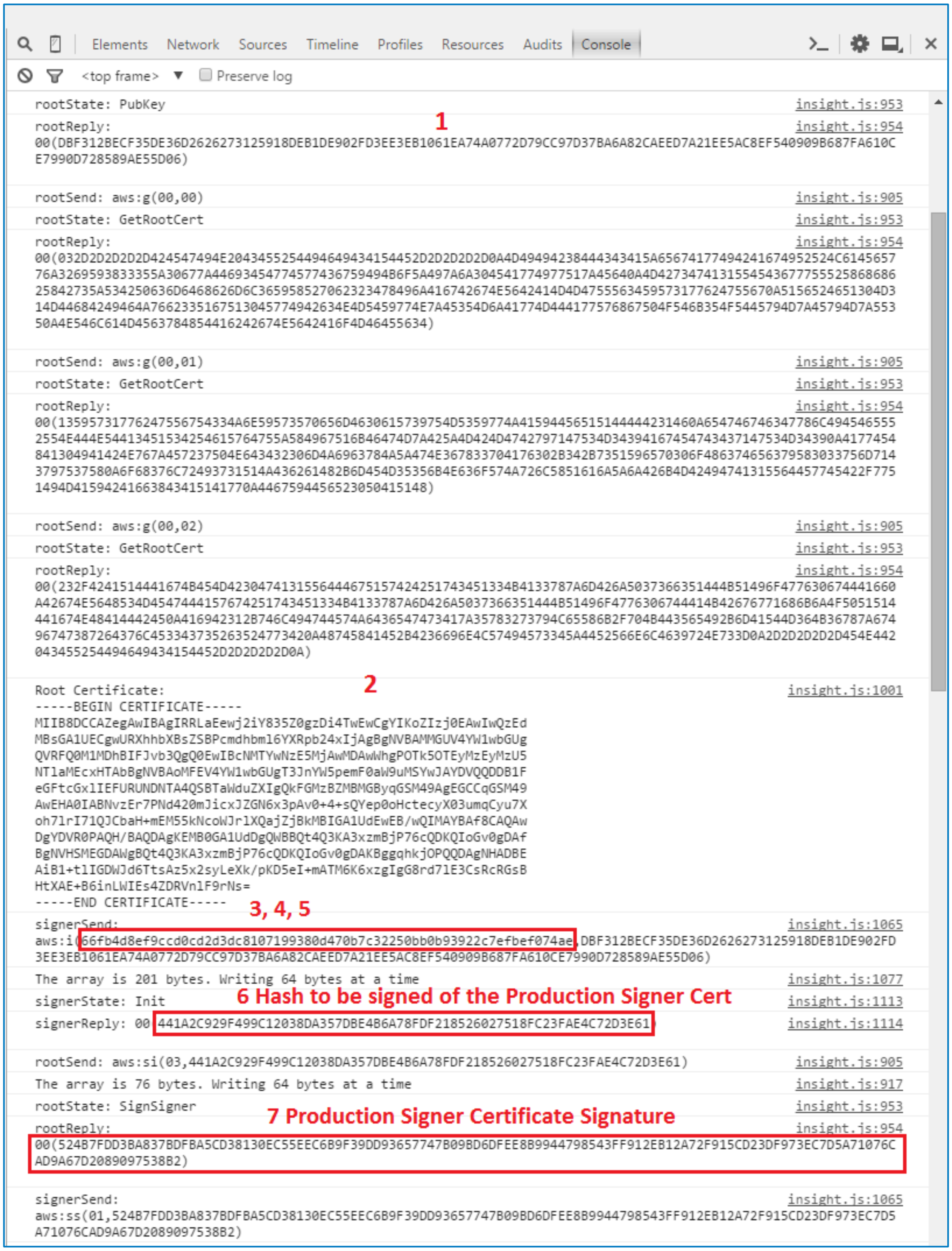

Figure 7-14. Registering the Production Signer and Verification Code Certificate into AWS IoT part 1

Figure 7-15. Registering the Production Signer and Verification Code Certificate into AWS IoT part 2

During the registration process with AWS IoT the GUI will also enable auto registration (JITR), functionality for our production signer CA certificate as shown below in the Figure 7-15 .

```
// Send the signer & verification cert to AWS
var params =
{
    caCertificate: signerCert, /* required */
    verificationCertificate: verifyCert, /* required */
    setAsActive: true,
    allowAutoRegistration: true
};
awsIot.registerCACertificate(params, function(err, data)
{
    if (err)
    {
        // an error occurred
        console.log("Failed to register Signing Module\n", err, err.stack);
    }
    else
    {
        // successful response
        //console.log(data);
        console.log("Signing Module Registration Successful!\n");
    }
});
```

Figure 7-16. Activating the Production Signer CA certificate and enabling auto-registration for JITR on the GUI

After we have successfully registered our Production Signer Certificate into AWS IoT we can go and access our account and verified the registration.

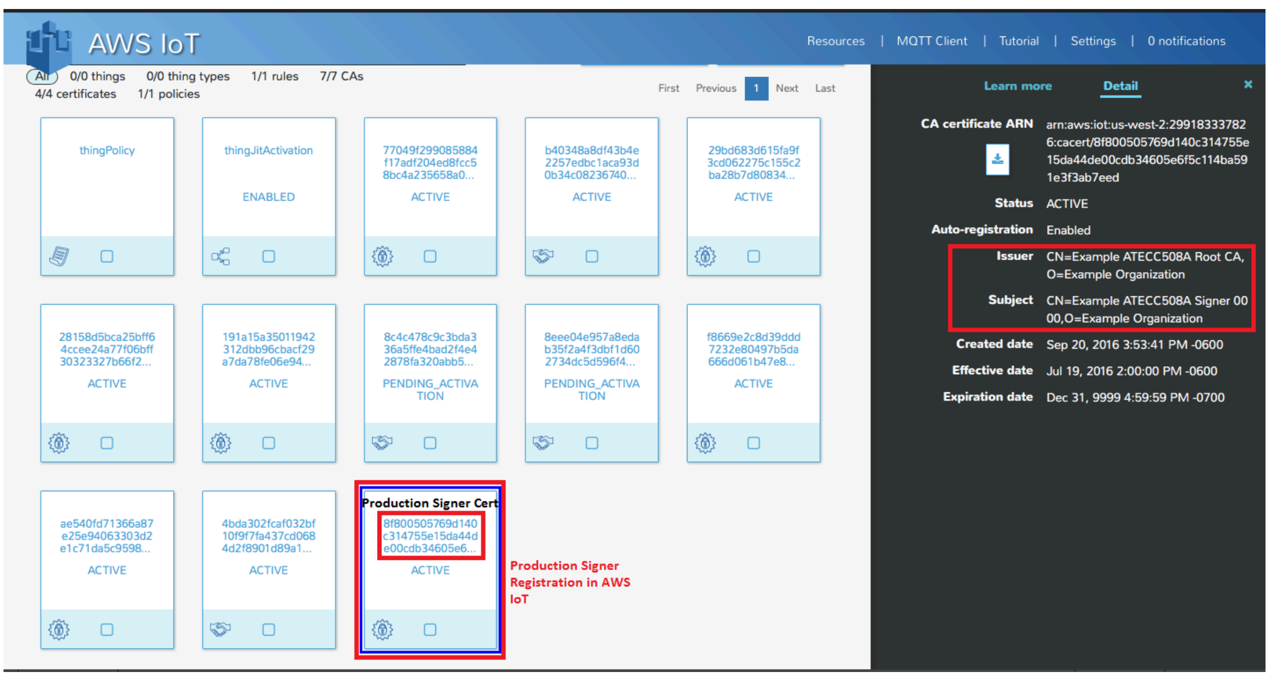

Figure 7-17. Production Signer Registration in AWS IoT

7.3 Just-In-Time Registration of the AWS IoT device

In the previous section we showed how use-your-own-certificate (**BYOC**), of AWS IoT will allow to use device certificates signed by a production signer, to connect and authenticate with AWS IoT.

In this section we will now use the Just-In-Time Registration support from AWS IoT. With JITR we will eliminate the need to manually registered any device that was signed by the Production Signer and turn this into an automated process.

To enable an AWS IoT device for JITR, the following steps need to take place

- Create, register and activate a CA certificate that will be used to sign your device certificate (like we did in the previous section).
- Enable auto-registration of certificates (like we did in the previous section).
- Create device certificates signed by your CA and install them on your device
- Create and attach a rule with an AWS Lambda action that activates the certificate and then creates ad attaches policies to the certificate.
- Connect to AWS IoT using the device certificate

7.3.1 AWS IoT TLS and JITR Hands- On

The objective of this hands on will be to demonstrate the process that involves the JITR functionality of AWS IoT.

For this portion you must have had completed the previous section successfully

We will be using the following setup:

- AWS IoT device compose of:
  - ATSAMG55
  - Winc1500 (updated to FW 19.4.4)
  - CryptoAuthXplained Pro
- Secure Insight of Things GUI
- Root Module
- Signer Module

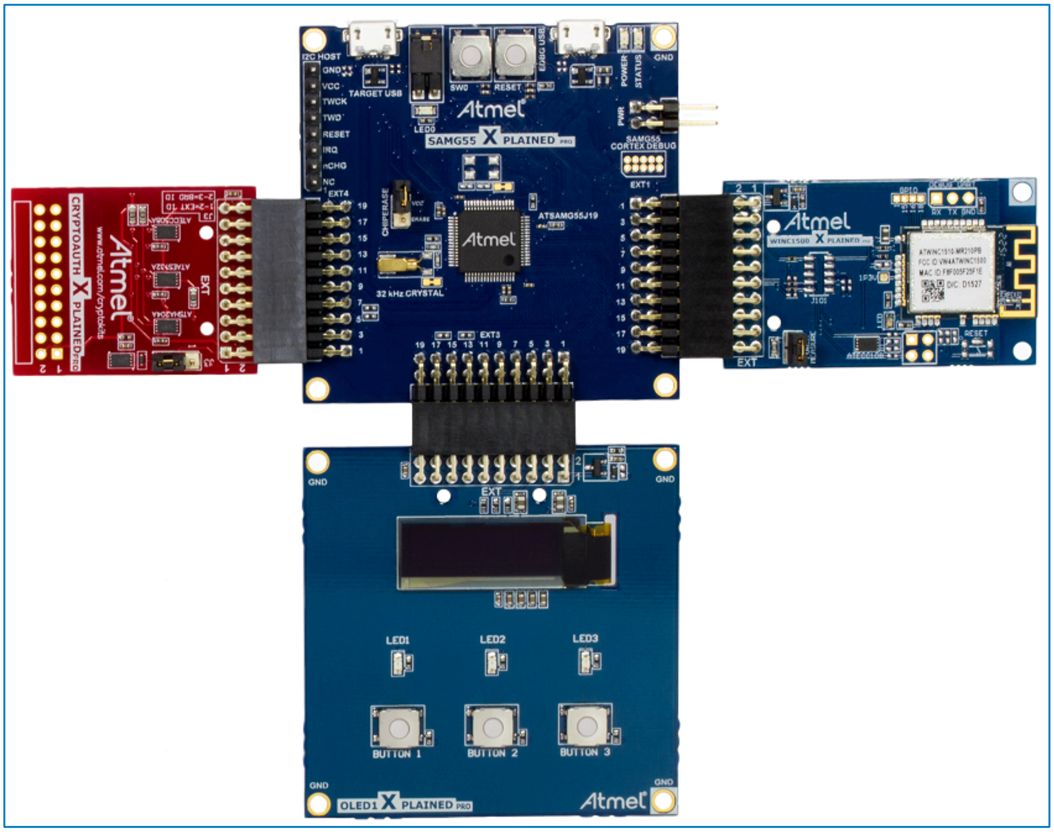

Figure 7-18. ATSAMG55 AWS IoT device

The **AWS Zero Touch Secure Development Kit** comes with a micro USB cable. Connect one end of the USB cable to the **USB Target** connector in ATSAMG55 setup, as shown in Figure 7-19.

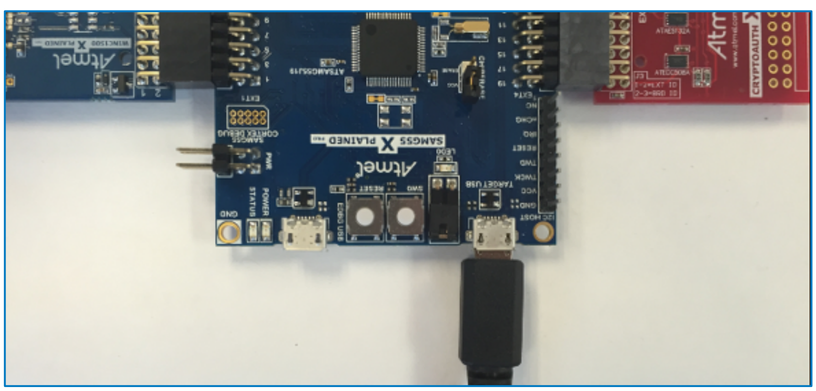

Figure 7-19. Connecting the USB cable to your ATSAMG55 setup.

Below is the sequence diagram that we will follow for AWS IoT JITR. Here we assume that the process for registering your Production Signer CA certificate and activating the JITR functionality has been completed.

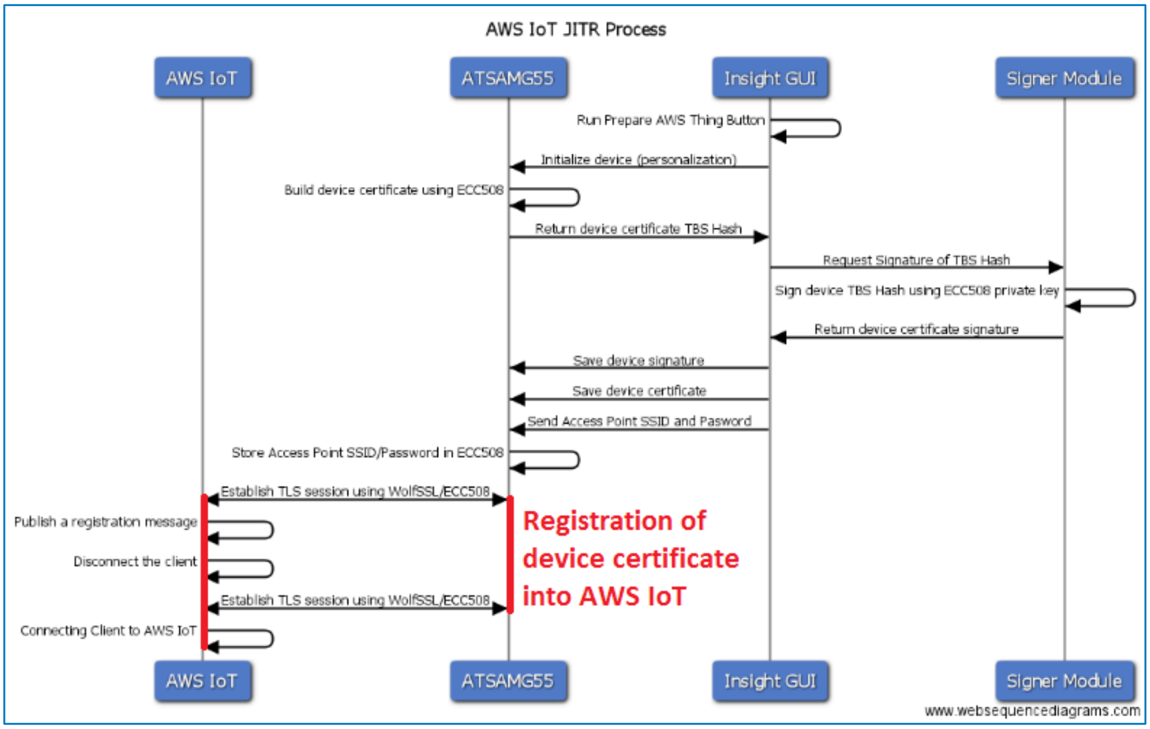

Figure 7-20. AWS IoT device JITR process

When a device attempts to connect with it's a certificate that it is not known to AWS IoT but was signed by a Signer CA certificate that was registered with AWS IoT, the device certificate will be auto-registered by AWS IoT in a new PENDING-ACTIVATION state. This means that the device is certificate was auto-registered but is not active. This is only controlled by AWS IoT.

The change of status of from PENDING-ACTIVATION to ACTIVE can be done by means of an AWS Lambda action on the registration topic that will activate the certificate, create and attach a policy to it.

For more information about creating a Lambda function and creating and attaching a policy, visit the AWS IoT site bellow.

https://aws.amazon.com/blogs/iot/just-in-time-registration-of-device-certificates-on-aws-iot/

For this hands-on below you can find the Lambda function we are using along with the policy that we are creating and device activation in the Appendix section of this document.

Registering the AWS IoT Device Certificate Using the GUI

Connect your Root Module, Signer Module and AWS IoT device to your desktop, then open the Secure Insight on Things GUI.

Before we proceed we need to make sure that all of our devices (ATSAMG55, Root Module, Signer Module), are being detected. If they are, you should see them enumerated with in the GUI as shown below in Figure 7-21.

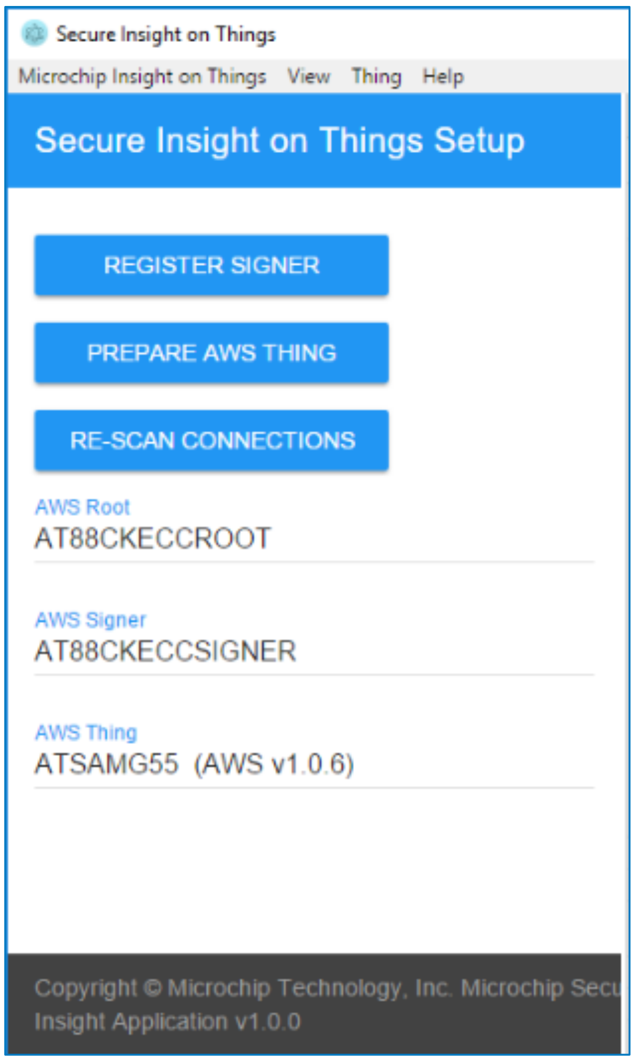

Figure 7-21. Secure Insight on Thing Gui

If you have all of the devices connected to your desktop and still do not see all of them showing up, go to **View** menu and select **Reload**.

In order to see the log of the Production Signer registration into AWS IoT, we will use as aid the console view with in our GUI. To enable this view, follow the next steps:

- Open the View menu.
- Select Toggle DevTools from the drop down menu.

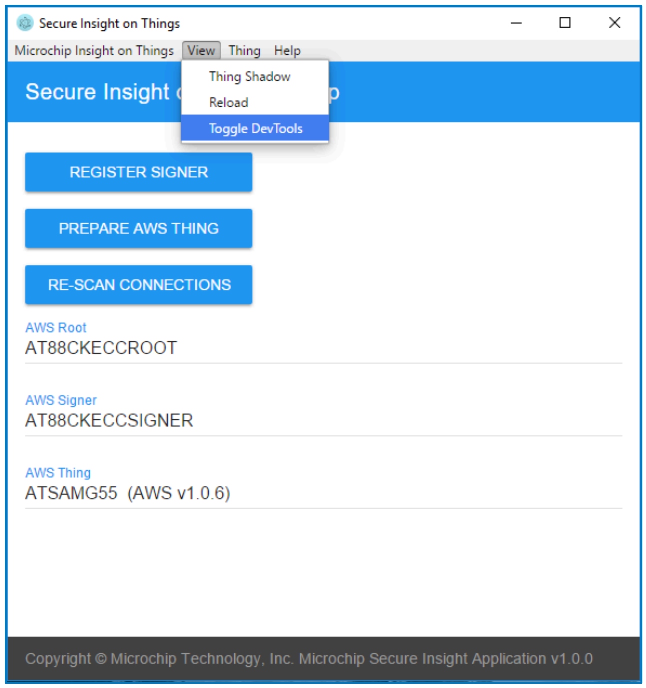

Figure 7-22. Enabling the Console view of the Secure Insight on Things GUI

Press the “**Prepare AWS Thing**” button. This will initiate the Device Registration which we previously discussed in the following order.

1. **Secure Insight GUI** will request the **IoT Device** to build its certificate.
2. **Secure Insight GUI** will request the **Signer Module** to sign the TBS Hash.
3. **IoT Device** will save the Device Signature and Device Certificate.
4. **Secure Insight GUI** will send the access point SSID and Password to **IoT Device**
5. **IoT Device** will save the access point SSID and Password in ECC508.
6. **IoT Device** will stablish a TLS connection with **AWS IoT** using ECC508 to harden TLS session
7. **AWS IoT** will publish a registration message and set **IoT Device** in pending activation
8. **AWS IoT** will disconnect Client (**IoT Device**)

9. **IoT Device** will stablish a TLS connection with **AWS IoT** using ECC508 to harden TLS session
10. JITR has been successful and **IoT Device** is connected to **AWS IoT**

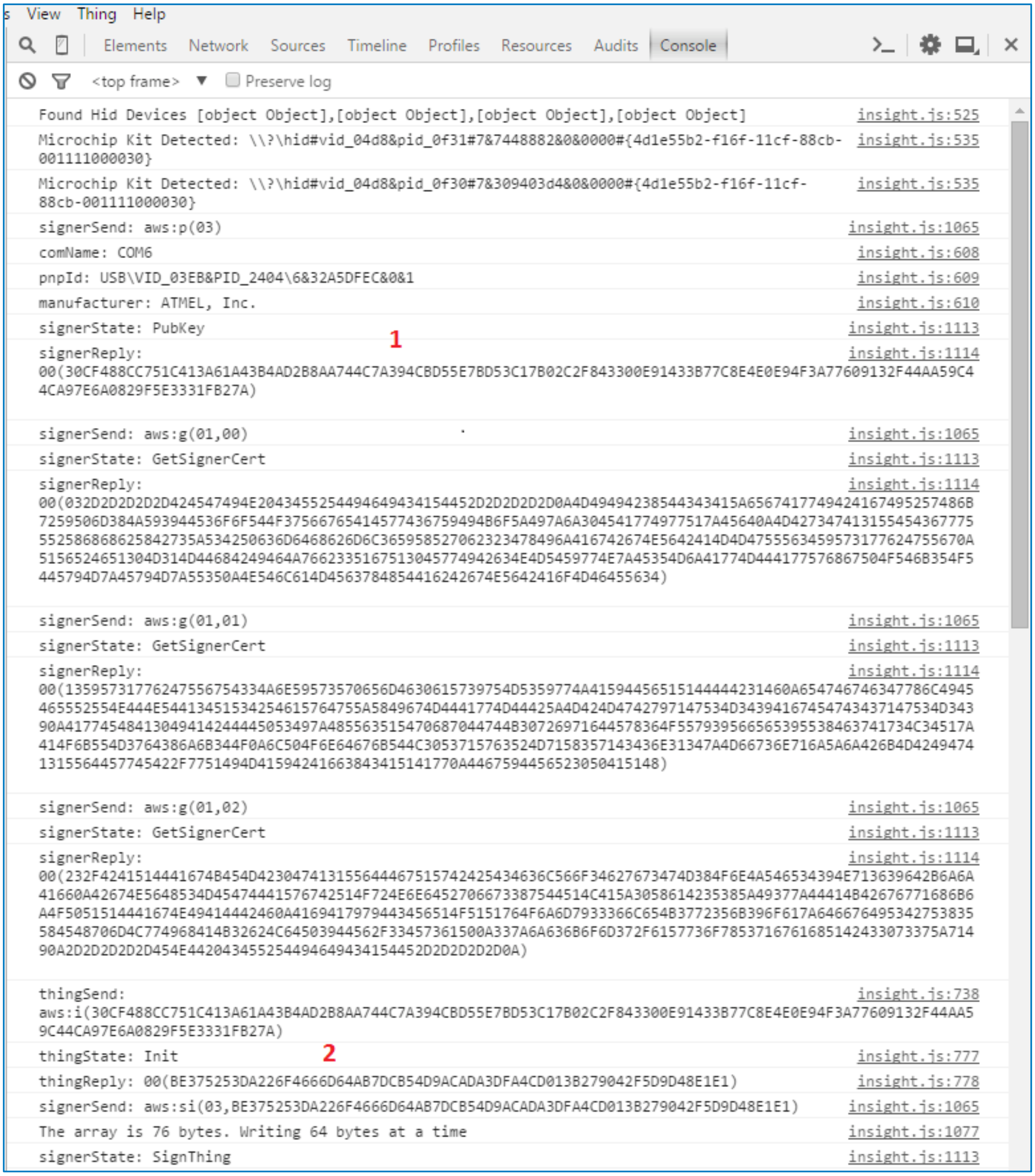

Figure 7-23. JITR process using GUI part 1

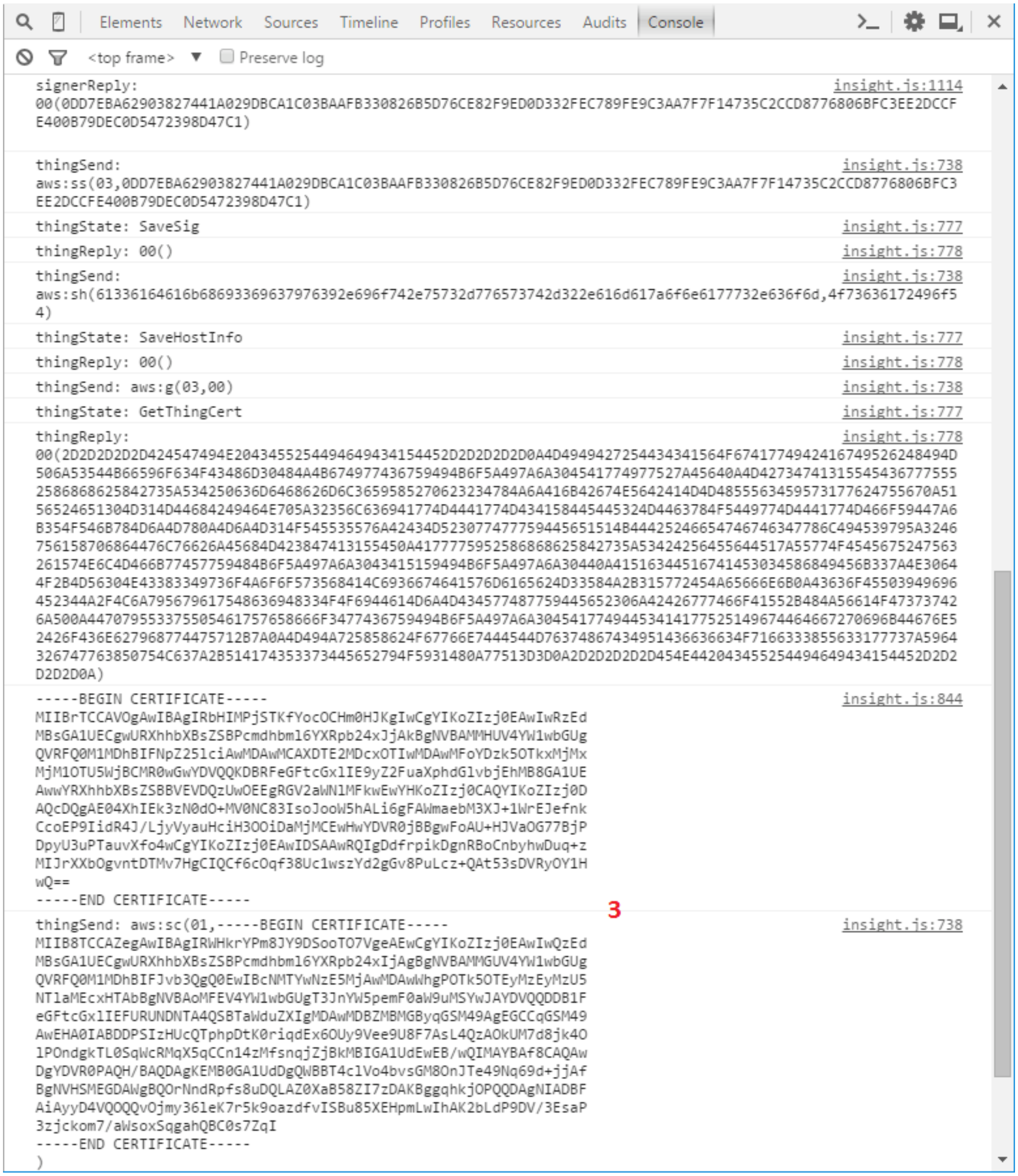

Figure 7-24. JITR process using GUI part 2

Once the device has been registered in to AWS IoT, we can go into our AWS IoT account and verify it, like the Figure below.

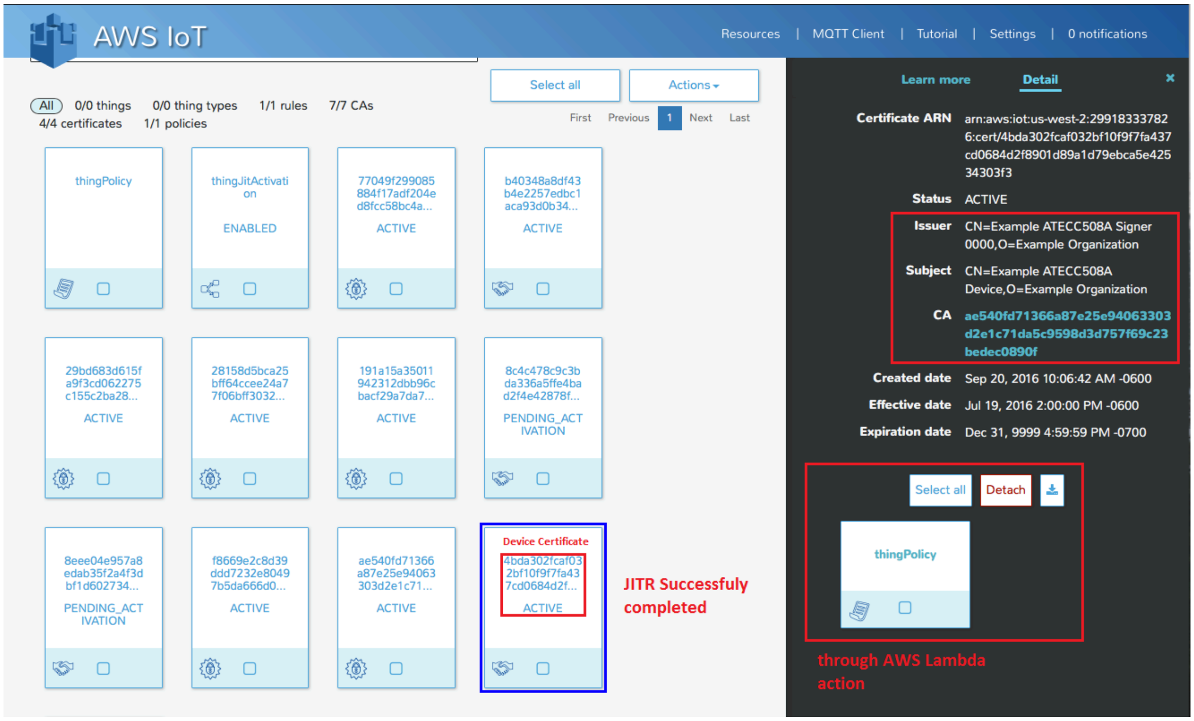

Figure 7-25. JITR successfully completed

8 Appendix

8.1 Lambda Function, Policy Attachment and Device Activation

```
/**
This node.js Lambda function code creates and attaches an IoT policy to the certificate
registered. It also activates the certificate. The Lambda function is attached as the rules engine action to
the registration topic aws/events/certificates/registered/<caCertificateID>

event JSON Structure:
{
  "certificateId": "<certificateID>",
  "caCertificateId": "<caCertificateId>",
  "timestamp": <timestamp>,
  "certificateStatus": "PENDING_ACTIVATION",
  "awsAccountId": "<awsAccountId>",
  "certificateRegistrationTimestamp": "<certificateRegistrationTimestamp>"
}
**/
var AWS = require('aws-sdk');
var region;
var iot;
var accountId;
var certificateARN;
var certificateId;
var thingPolicy;
const awsPolicyName = 'thingPolicy';

// Delay time tracking
var eventTime;
exports.handler = function(event, context, callback)
{
// Step 1: Create the policy.
// Step 2: Attach the policy to the certificate
// Step 3: Activate the certificate.
//Optionally, you can have your custom Certificate Revocation List (CRL) check logic here and
//ACTIVATE the certificate only if it is not in the CRL .Revoke the certificate if it is in the CRL
// Capture the event time & delay for Lambda execution
    var currentTime = (new Date()).getTime();
    eventTime = event.timestamp;
    var eventDelay = currentTime - eventTime;
    console.log("Lambda event delay: " + eventDelay);
```

```
// Replace it with the AWS region the lambda will be running in region = "us-west-2";

// Get the AWS account ID
accountId = event.awsAccountId.toString().trim();

// Create the Iot object
iot = new AWS.Iot({'region': region, apiVersion: '2015-05-28'});
certificateId = event.certificateId.toString().trim();

 // Construct the ARN for the Thing certificate
certificateARN = `arn:aws:iot:${region}:${accountId}:cert/${certificateId}`;

// Create and attach the thingPolicy
awsCreateSimplePolicy();
};


function awsCreateSimplePolicy()
{
    // Step 1: Create the policy
    // Policy definition
     var policy = {
            "Version": "2012-10-17",
            "Statement": [{
                    "Effect": "Allow",
                    "Action":["iot:*"],
                    "Resource": ["*"]
            }]
    };

    // Create the policy
    iot.createPolicy(
    {
        policyDocument: JSON.stringify(policy),
        policyName: awsPolicyName
    }, (err, data) =>
    {
        // Log the delay for the createPolicy() callback
        var currentTime = (new Date()).getTime();
        var callbackDelay = currentTime - eventTime;
        console.log("awsCreatePolicy() Delay: " + callbackDelay);
```

```
// Ignore if the policy already exists
        if (err && (!err.code || err.code !== 'ResourceAlreadyExistsException'))
        {
            console.log(err);
            return;
        }
        // Set the thingPolicy to the return data
        thingPolicy = data;

        // Step 2: Attach the policy to the certificate
        awsAttachPolicy();
    });
}

function awsAttachPolicy()
{
// Step 2: Attach the policy to the certificate
// Attach policy to certificate
iot.attachPrincipalPolicy(
{
   policyName: awsPolicyName,
   principal: certificateARN
}, (err, data) =>
{
    // Log the delay for the attachPrincipalPolicy() callback
     var currentTime = (new Date()).getTime();
     var callbackDelay = currentTime - eventTime;
     console.log("awsAttachPolicy() Delay: " + callbackDelay);

   // Ignore if the policy is already attached
   if (err && (!err.code || err.code !== 'ResourceAlreadyExistsException'))
   {
   console.log("Failed to attach Policy to \"Thing\" certificate\n" + err);
   return;
   }
   // We've attached the policy, now activate the certificate
   awsActivateThing();
   });
}
```

```
function awsActivateThing()
{
    //Step 3: Activate the certificate.
    //                              Optionally, you can have your custom Certificate Revocation List (CRL) check logic here and
    //                              ACTIVATE the certificate only if it is not in the CRL .Revoke the certificate if it is in the CRL
    iot.updateCertificate(
    {
        certificateId: certificateId,
        newStatus: 'ACTIVE'
    }, (err, data) =>
    {
        // Log the delay for the updateCertificate() (activate) callback
        var currentTime = (new Date()).getTime();
        var callbackDelay = currentTime - eventTime;
        console.log("awsActivateThing() Delay: " + callbackDelay);

        if (err)
        {
            console.log("Thing activation failed.");
        }
        else
        {
            console.log("Thing activated successfully.");
        }
    });
}
```

9 License Information

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**