AWS IoT

Setting up a new AWS IOT Thing (with eLinux example on a raspberry pi)

Author: James McAnanama

Version: 28

Date: 06-Jan-2017 20:36

Table of Contents

1 Introduction

This article shows how to setup an embedded IOT Thing to work with AWS. Goals:

- We will setup a raspberry pi to be a data logger.
- We will create an abstract thing type of 'data logger' and describe our first thing as such.

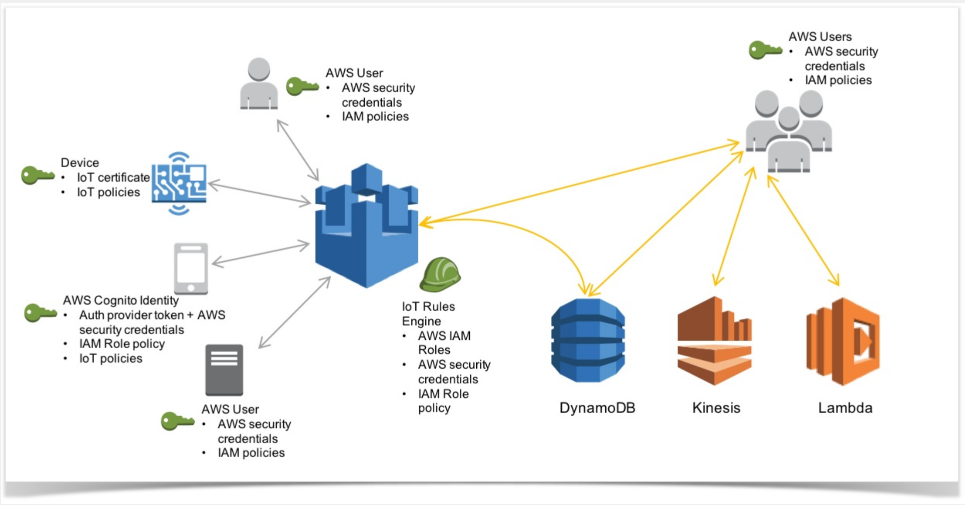

Image credit.

2 Create a Device in the Thing Registry

AWS IoT provides a thing registry that helps you manage your things. A thing is a representation of a specific device or logical entity. It can be a physical device or sensor (for example, a light bulb or a switch on a wall). It can also be a logical entity like an instance of an application or physical entity that does not connect to AWS IoT but is related to other devices that do (for example, a car that has engine sensors or a control panel).

Things are identified by a name. Things can also have up to three attributes, which are name-value pairs you can use to store information about the thing, such as its serial number or manufacturer. If you also assign a thing type (below), then you can have up to 50 attributes, three of which are searchable.

A typical device use case involves the use of the thing name as the default MQTT client ID. Although AWS does not enforce a mapping between a thing's registry name and its use of MQTT client IDs, certificates, or shadow state, they recommend you choose a thing name and use it as the MQTT client ID for both the thing registry and the Thing Shadows service. This provides organization and convenience to your IoT fleet without removing the flexibility of the underlying device certificate model or thing shadows.

You do not need to create a thing in the thing registry to connect it to AWS IoT. Adding your things in the thing registry allows you to manage and search for them more easily.

2.1 Thing Types

Thing types allow you to store description and configuration information that is common to all things associated with the same thing type. This simplifies the management of things in the thing registry. For example, you can define a data_logger thing type. All things associated with the data_logger thing type share a set of attributes such as: model, serial number, and firmware info. We may also want to add things like mac address, installed options etc, but this has to be added per instance as the thing type is limited to three. *(The approach I am using is to design the thing type with maintenance in mind; what are the three key attributes that we would like to search to quarantine and repair our fleet.)* When you create a thing of type data_logger (or change the type of an existing thing to data_logger) you can specify values for each of the attributes defined in the data_logger thing type.

Although thing types are optional, their use provides better discovery of things.

- *Things can have up to 50 attributes (3 searchable attributes defined by the thing type, and an additional 47 instance attributes)*

- *Things without a thing type can have up to three attributes.*
- *A thing can only be associated with one thing type.*
- *There is no limit on the number of thing types you can create in your account.*

Thing types are immutable. You cannot change a thing type name after it has been created. You can deprecate a thing type at any time to prevent new things from being associated with it. You can also delete thing types that have no things associated with them.

```
Create a JSON Thing Type Definintion File:

#Let's get AWS to create a skeleton thing type definition file
for us to edit:
$ aws iot create-thing-type  --generate-cli-skeleton >
data_logger_type.json

$ vi data_logger_type.json

#make these changes to the resulting data_logger_type.json file:
{
    "thingTypeName": "data_logger",
    "thingTypeProperties": {
        "thingTypeDescription": "data loggers",
        "searchableAttributes": [
            "model",
            "serial_number",
            "fw_info",
        ]
    }
}
```

```
Create a Thing Type:

#Create our data_logger type on aws:
$ aws iot create-thing-type --thing-type-name "data_logger" \
                            --cli-input-json file://data_logger_ty
pe.json
{
   "thingTypeName": "data_logger",
   "thingTypeArn": "arn:aws:iot:us-east-1:560429543710:thingtype
/data_logger"
}
```

```
Create a Thing:

#Now let's create a thing using the aws cli, also using the --
generate-cli-skeleton trick to define the structure of our json
configuration:
$ aws iot create-thing --generate-cli-skeleton >
data_logger_thing.json
# edit data_logger_thing.json:
{
    "thingName": "ras_pi_fc2de3e3",
    "thingTypeName": "data_logger",
    "attributePayload": {
        "attributes": {
            "model"         : "Pi-B-Rev-2.0-512Mb",
            "serial_number" : "00000000fc2de3e3",
            "fw_info"       : "",
            "mac"           : "b8:27:eb:2d:e3:e3",
            "location"      : "my_desk",
            "description"   : "protoytpe_AWS_IOT_data_logger"
        },
        "merge": false
    }
}

$ aws iot create-thing --cli-input-json file://data_logger_thing.
json
{
   "thingArn": "arn:aws:iot:us-east-1:560429543710:thing/MyRasPi",
   "thingName": "ras_pi_fc2de3e3"
}
```

3 Create and Activate a Device Certificate

Each connected device must have a credential to access the message broker or the Thing Shadows service. All traffic to and from AWS IoT must be encrypted over Transport Layer Security (TLS). Device credentials must be kept safe in order to send data securely to the message broker. After data reaches the message broker, AWS cloud security mechanisms protect data as it moves between AWS IoT and other devices or AWS services.

- We are responsible for managing device credentials (X.509 certificates, AWS credentials) on our devices and policies in AWS IoT. We are responsible for assigning unique identities to each device and managing the permissions for a device or group of devices.
- Devices connect using our choice of identity (X.509 certificates, IAM users and groups, or Amazon Cognito identities) over a secure connection according to the AWS IoT connection model.
- The AWS IoT message broker authenticates and authorizes all actions in our account. The message broker is responsible for authenticating our devices, securely ingesting device data, and adhering to the access permissions we place on devices using policies.
- The AWS IoT rules engine forwards device data to other devices and other AWS services according to rules we define. It is responsible for leveraging AWS access management systems to securely transfer data to its final destination.

3.1 Transport Security

The AWS IoT message broker and Thing Shadows service encrypt all communication with TLS. TLS is used to ensure the confidentiality of the application protocols (MQTT, HTTP) supported by AWS IoT. TLS is available in a number of programming languages and operating systems.

For MQTT, TLS encrypts the connection between the device and the broker. TLS client authentication is used by AWS IoT to identify devices. For HTTP, TLS encrypts the connection between the device and the broker. Authentication is delegated to AWS Signature Version 4.

3.2 Authentication in AWS IoT

AWS IoT supports three types of identity principals for authentication:

-
    - X.509 certificates
    - IAM users, groups, and roles
    - Amazon Cognito identities

Each identity type supports different use cases for accessing the AWS IoT message broker and Thing Shadows service.

The identity type you use depends on your choice of application protocol. If you use HTTP, use IAM (users, groups, roles) or Amazon Cognito identities. If you use MQTT, use X.509 certificates.

We are going to use the MQTT protocol and so will use X.509 TLS to connect our things to AWS. We will use IAM for fanning out into other AWS services or across AWS accounts. Read more: X.509 Certificates and AWS IoT.

3.3 Ours or Theirs?

Our signing CA certificate must be registered with AWS before we can get X.509 certificates. By default, AWS will use their own CA certificate and no action is required by us to register an alternative CA.

We can generate our own private:public key pair and then issue a certificate signing request (CSR) to AWS to get an X.509 digital identity certificate (see create-certificate-from-csr). We may want to do this if we are using special crypto hardware such as the Atmel ATECC508A chip. Alternatively, you can have AWS provide the key pair and certificate for your device.

```
Create a certificate:

#Time to create a private key, public key, and X.509 certificate
using the aws cli.
$ aws iot create-keys-and-certificate --set-as-active   --
certificate-pem-outfile ras_pi_fc2de3e3_cert.pem \
                                                        --public-
key-outfile  ras_pi_fc2de3e3_publicKey.pem \
                                                        --private-
key-outfile ras_pi_fc2de3e3_privkey.pem

{
    "certificateArn": "arn:aws:iot:us-east-1:560429543710:cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46",
    "certificatePem": "-----BEGIN CERTIFICATE----- redacted -----
END CERTIFICATE-----\n",
    "keyPair":
{
        "PublicKey": "-----BEGIN PUBLIC KEY----- redacted -----
END PUBLIC KEY-----\n",
        "PrivateKey": "-----BEGIN RSA PRIVATE KEY----- definitely
redacted -----END RSA PRIVATE KEY-----\n"

},
    "certificateId": "5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7
af17780835cde1dd8b46"
}
```

This create-keys-and-certificate command results in the following files:

1. certificatePem, saved locally as ras_pi_fc2de3e3_cert.pem (the filename was chosen by us based on the name and serial number of our data logger): This is our certificate signed by the CA registered for us at AWS (in this case we are using AWS's CA). When we want to connect our thing to AWS, AWS will request this cert and will use it to:
   a. Validate the status of the certificate and or AWS account.
   b. Send our thing a cryptographic challenge to prove we have the private key associated with the public key included in the certificate. As long as we keep our private key secret, only our thing can successfully respond to the challenge.
2. PrivateKey, saved locally as ras_pi_fc2de3e3_privkey.pem: This is our private key for our data logger. This must be kept secret otherwise anyone who has a copy can impersonate our thing and decrypt messages.

3. PublicKey, saved locally ras_pi_fc2de3e3_publicKey.pem: The is our public key for out data logger. It is shared with AWS in the certificate. Anyone with the public key can encrypt messages that only we can decrypt.
4. certificateArn and certificateId: This is the distinguishable name for the certificate for when we need to tell AWS which cert we are referring to.

Enterprise Thinking

James McAnanama, TODO: How would we manage a fleet of things? Where would we store the certificates and keys? How would we keep the private keys secure? This is where hardware like the ATECC508A comes in handy. Note, any provisioning automation would have to parse the certificateId/ARN from the command response and somehow save it, cross referenced to the cert and thing.

The certificate, key provisioning gives us a means to authenticate our thing with AWS. Next, we will need a way to establish what our thing is authorized to do. This is accomplished with an AWS IoT Policy.

4 Create an AWS IoT Policy

4.1 Authentication vs Authorization on AWS IoT

Communication with AWS IoT follows the principle of least privilege. An identity can execute AWS IoT operations only if we grant the appropriate permission. We create AWS IoT and IAM policies to give permissions to authenticated identities in AWS IoT.

Certificates and key pairs are used to authenticate the identity of the thing to AWS and allows AWS to cross reference the thing to an account and to the thing within the account's "thing registry". However, policies give permissions to AWS IoT things. To control which resources a device can access, attach one or more AWS IoT policies to the certificate associated with the device. Likewise, we will have to control which resources a web or mobile application can access. Referring to the image at the top of this page, we will need to attach one or more AWS IoT policies to the Amazon Cognito identity pool associated with the application. AWS IoT policies control access to AWS IoT resources (MQTT topics, devices, thing shadows, and so on). IAM policies control access to other AWS services and are attached to IAM users, groups, and roles.

Policy-based authorization is a powerful tool. It gives you complete control over the topics and topic filters in your AWS account. For example, consider a device connecting to AWS IoT with a certificate. You can open its access to all topics, or you can restrict its access to a single topic. The latter example allows you to assign a topic per device. For example, the device ID 123ABC can subscribe to `/device/123ABC` and you can grant other identities permission to subscribe to this topic, effectively opening a communication channel to this device.

An AWS IoT policy looks like the following:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action":["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:123456789012:topic/foo/

    },
    {
        "Effect": "Allow",
        "Action": ["iot:Connect"],
        "Resource": ["*"]
```

```
            }]
    }
```

The components are: *Version*

Must be set to `"2012-10-17"`. *Effect*

Must be set to `"Allow"` or `"Deny"`. *Action*

Must be set to "iot:*<operation-name>*" where <operation-name> is one of the following:

`"iot:Publish"`: MQTT publish.

`"iot:Subscribe"`: MQTT subscribe.

`"iot:UpdateThingShadow"`: Update a thing shadow.

`"iot:GetThingShadow"`:Retrieve a thing shadow.

`"iot:DeleteThingShadow`:Delete a thing shadow. *Resource*

Must be set to one of the following:

Client - arn:aws:iot:*<region>*:*<accountId>*:client/*<clientId>*

Topic ARN - arn:aws:iot:*<region>*:*<accountId>*:topic/*<topicName>*

Topic filter ARN - arn:aws:iot:*<region>*:*<accountId>*:topicfilter/*<topicFilter>*

This policy allows the principal to connect and publish messages to AWS IoT.

4.2 Authorization with an AWS IoT Policy

AWS IoT policies are JSON documents. They follow the same conventions as IAM policies. AWS IoT supports named policies so many identities can reference the same policy document. Named policies are versioned so they can be easily rolled back. The following are actions available for use with AWS IoT:

Action	Description	Resource
iot:Publish	Checked every time a PUBLISH request is sent to the broker. Used to allow clients to publish to specific topic patterns.	topic ARN
iot:Subscribe	Checked every time a SUBSCRIBE request is sent to the broker. Used to allow clients to subscribe to topics that match specific topic patterns.	topic filter ARN

Action	Description	Resource
iot:Receive	Checked every time a message is delivered to a client. Because the Receive permissions is checked on every delivery, it can be used to revoke permissions to clients that are currently subscribed to a topic.	topic ARN
iot:Connect	Checked every time a CONNECT request is sent to the broker. The message broker does not allow two clients with the same client ID to stay connected at the same time. After the second client connects, the broker detects this case and disconnects the oldest connections. The Connect permission can be can be used to ensure only authorized clients can connect using a specific client ID.	client ID ARN
iot: UpdateThingShadow	Checked every time a request is made to update the state of a thing shadow document.	thing ARN
iot:GetThingShadow	Checked every time a request is made to get the state of a thing shadow document.	thing ARN
iot: DeleteThingShadow	Checked every time a request is made to delete the thing shadow document.	thing ARN

4.3 Policy Variables

It is possible to create placeholders in a policy for evaluation at runtime using policy variables - when the policy is evaluated, the policy variables are replaced with values that come from the request itself. The following is a list of iot policy variables:

Variable	Resolves to
iot:ClientId	Client ID that sent an MQTT message. (Warning: do not use in iot: Connect - specifically allow things to connect. Failure to do so, can allow exploited clientID's like '+' that are topic wildcards).
aws:SourceIp	IP address from which the message originated.

Variable	Resolves to
iot:Connection. Thing. ThingName	The name of the thing for which the policy is being evaluated. The thing name is obtained from the client ID for the MQTT connection. This policy variable is only available when connecting over MQTT.
iot:Connection. Thing. ThingTypeName	The thing type associated with the thing for which the policy is being evaluated. The thing name is obtained from the client ID for the MQTT. TThe thing type name is obtained by a call to the DescribeThing API.
iot:Connection. Thing.Attributes[*attributeName*]	the value of the specified attribute associated with the thing for which the policy is being evaluated. A thing can have up to 50 attributes. Each attribute will be available as a policy variable: iot:Connection.Thing. Attributes[*attributeName*].
iot:Connection. Thing.IsAttached	This resolves to true if the thing has a certificate or Amazon Cognito attached.
X.509 Certificate Variables	Various items within the authentication certificate.

More information, along with sample policies is located here.

Enterprise Thinking

As a design goal, let's keep security at the forefront. Only devices created by us are allowed to connect to out account. In addition, each thing shall have its own unique ID, certificate and key pair. As a result, our policy shall only authorize a device that is attached to the authentication certificate to make a connection. In addition, let us protect ourselves from errors - to this end, we will corral our device types (e.g. our data_logger). We can attach multiple policies to a device and should further limit the privileges of a thing's instance (e.g. data_logger/engines, data_logger/hvac, data_logger/reentry_vehicle etc,)

Create a data_logger policy:

```
#Design goals:
# *Device shall be attached to the certificate for connection
authorization.
```

```
# *Device can only publish to topics about its device type and
name. (e.g. topics described as <device_type>/<device_name>.
# *Device can subscribe to all topics related to its device type
and name or master_control.  (e.g. topics described as
<device_type>/<device_name>|master_control.
# Creat policy JSON file:  data_logger_policy.json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "*"
      ],
      "Condition": {
        "Bool": {
          "iot:Connection.Thing.IsAttached": [
            "true"
          ]
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:560429543710:*${iot:Connection.
Thing.ThingTypeName}/${iot:Connection.Thing.ThingName}/*",
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:560429543710:*${iot:Connection.
Thing.ThingTypeName}/${iot:Connection.Thing.ThingName}/*",
        "arn:aws:iot:us-east-1:560429543710:*${iot:Connection.
Thing.ThingTypeName}/master_control/*"
      ]
    }
  ]
}
```

```
$ aws iot create-policy --policy-name "data_logger_policy" --
policy-document file://data_logger_policy.json
```

5 Attach an AWS IoT Policy to a Device Certificate

To control which resources a device can access, we need to attach one or more AWS IoT policies to the certificate associated with the device

Attach our data_logger policy to our thing's certificate

```
# When we created our certificate for our device, AWS informed us
that the certificate arn was: arn:aws:iot:us-east-1:560429543710:
cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46
# You can view your certificates with:
$ aws iot list-certificates
{
    "certificates":
[

{
            "certificateArn": "arn:aws:iot:us-east-1:560429543710:
cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46",
            "status": "ACTIVE",
            "creationDate":
1483650398.338,
            "certificateId": "5a6043866e3acdbcfbfd1fa68a9b09907bf6
0b0bfff7af17780835cde1dd8b46"

},

{
            "certificateArn": "arn:aws:iot:us-east-1:560429543710:
cert
/393d7422300d2118b14e452caedc8ccb7e6b85e29b7e6bd6ee4993646e319a11",
            "status": "ACTIVE",
            "creationDate":
1483402790.071,
            "certificateId": "393d7422300d2118b14e452caedc8ccb7e6b
85e29b7e6bd6ee4993646e319a11"

}

]
}
```

```
$ aws iot attach-principal-policy --policy-name "data_logger_polic
y" --principal "arn:aws:iot:us-east-1:560429543710:cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46"

# Verify the results:
$ aws iot list-policy-principals --policy-name "data_logger_policy
"
{
    "principals": [
        "arn:aws:iot:us-east-1:560429543710:cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46"
    ]
}
```

6 Attach a Thing to a Certificate

It is possible to have multiple things tied to a certificate. However, the message broker does not allow two clients with the same client ID to stay connected at the same time. To prevent an error where multiple things end up with the same client id, we will use a singleton of a certificate for our thing with a shared policy among singletons.

Attach our thing to our thing's certificate

```
# When we created our certificate for our device, AWS informed us
that the certificate arn was: arn:aws:iot:us-east-1:560429543710:
cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46
# You can view your certificates with:
$ aws iot list-certificates
{
    "certificates":
[

{
            "certificateArn": "arn:aws:iot:us-east-1:560429543710:
cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46",
            "status": "ACTIVE",
            "creationDate":
1483650398.338,
            "certificateId": "5a6043866e3acdbcfbfd1fa68a9b09907bf6
0b0bfff7af17780835cde1dd8b46"

},

{
            "certificateArn": "arn:aws:iot:us-east-1:560429543710:
cert
/393d7422300d2118b14e452caedc8ccb7e6b85e29b7e6bd6ee4993646e319a11",
            "status": "ACTIVE",
            "creationDate":
1483402790.071,
            "certificateId": "393d7422300d2118b14e452caedc8ccb7e6b
85e29b7e6bd6ee4993646e319a11"

}

]
}

$ aws iot attach-thing-principal --thing-name "ras_pi_fc2de3e3" --
principal "arn:aws:iot:us-east-1:560429543710:cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46"

# Verify the results:
$ aws iot list-thing-principals --thing-name "ras_pi_fc2de3e3"
{
    "principals": [
        "arn:aws:iot:us-east-1:560429543710:cert
/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af17780835cde1dd8b46"
    ]
}
#Likewise
$  aws iot list-principal-things --principal arn:aws:iot:us-east-
1:560429543710:cert/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af
17780835cde1dd8b46
{
    "things": [
        "ras_pi_fc2de3e3"
    ]
}
$ aws iot list-principal-policies --principal arn:aws:iot:us-east-
1:560429543710:cert/5a6043866e3acdbcfbfd1fa68a9b09907bf60b0bfff7af
17780835cde1dd8b46
{
    "policies": [
        {
            "policyName": "data_logger_policy",
            "policyArn": "arn:aws:iot:us-east-1:560429543710:
policy/data_logger_policy"
        }
    ]
}
```

7 Test Thing's Connection to AWS

Clone the aws device sdk, copy .pem to the certs directory and embedtls to the third party directory.

Lets run some test code on our raspberry pi

```
#Clone and setup the aws device sdk on your raspi:
pi@raspberrypi ~ $ mkdir dev
pi@raspberrypi ~ $ cd dev
pi@raspberrypi ~/dev $ git clone https://github.com/aws/aws-iot-
device-sdk-embedded-C.git
pi@raspberrypi ~/dev $ cd aws-iot-device-sdk-embedded-C/
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C $ cd certs
#Copy over your keys and cert you created above:
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/certs $ scp
<username>@<hostname>://<path to your keys and cert>/*.pem .
#Grab the root cert:
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/certs $ wget -
O rootCA.crt https://www.symantec.com/content/en/us/enterprise
/verisign/roots/VeriSign-Class%203-Public-Primary-Certification-
Authority-G5.pem
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/certs $ cd ../e
xternal_libs/
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/external_libs
$ git clone https://github.com/ARMmbed/mbedtls.git
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/external_libs
/mbedtls $ git checkout mbedtls-2.1.1
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/external_libs
/mbedtls $ cd ../../samples/
#Grab my altered sample (this includes the json files from the
rest of this tutorial as well).
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/samples $ git
clone https://github.com/jmcanana/aws_iot-data_logger.git
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/samples $ cd
aws_iot-data_logger/
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/samples $ make
#Go grab a coffee...
pi@raspberrypi ~/dev/aws-iot-device-sdk-embedded-C/samples
/aws_iot-data_logger $ ./subscribe_publish_sample
Connecting...
DEBUG:   iot_tls_connect L#120
  . Seeding the random number generator...
DEBUG:   iot_tls_connect L#128   . Loading the CA root
certificate ...
```

```
DEBUG:   iot_tls_connect L#134  ok (0 skipped)

DEBUG:   iot_tls_connect L#136   . Loading the client cert. and
key...
DEBUG:   iot_tls_connect L#149  ok

DEBUG:   iot_tls_connect L#151   . Connecting to data.iot.us-east-
1.amazonaws.com/8883...
DEBUG:   iot_tls_connect L#170  ok

DEBUG:   iot_tls_connect L#172   . Setting up the SSL/TLS
structure...
DEBUG:   iot_tls_connect L#204

SSL state connect : 0
DEBUG:   iot_tls_connect L#207  ok

DEBUG:   iot_tls_connect L#209

SSL state connect : 0
DEBUG:   iot_tls_connect L#210   . Performing the SSL/TLS
handshake...
DEBUG:   _iot_tls_verify_cert L#41
Verify requested for (Depth 2):

DEBUG:   _iot_tls_verify_cert L#43 cert. version     : 3
serial number     : 18:DA:D1:9E:26:7D:E8:BB:4A:21:58:CD:CC:6B:3B:
4A
issuer name       : C=US, O=VeriSign, Inc., OU=VeriSign Trust
Network, OU=(c) 2006 VeriSign, Inc. - For authorized use only,
CN=VeriSign Class 3 Public Primary Certification Authority - G5
subject name      : C=US, O=VeriSign, Inc., OU=VeriSign Trust
Network, OU=(c) 2006 VeriSign, Inc. - For authorized use only,
CN=VeriSign Class 3 Public Primary Certification Authority - G5
issued  on        : 2006-11-08 00:00:00
expires on        : 2036-07-16 23:59:59
signed using      : RSA with SHA1
RSA key size      : 2048 bits
basic constraints : CA=true
key usage         : Key Cert Sign, CRL Sign

DEBUG:   _iot_tls_verify_cert L#46   This certificate has no flags

DEBUG:   _iot_tls_verify_cert L#41
Verify requested for (Depth 1):

DEBUG:   _iot_tls_verify_cert L#43 cert. version     : 3
serial number     : 3F:92:87:BE:9D:1D:A4:A3:7A:9D:F6:28:2E:77:5A:
C4
```

```
issuer name       : C=US, O=VeriSign, Inc., OU=VeriSign Trust
Network, OU=(c) 2006 VeriSign, Inc. - For authorized use only,
CN=VeriSign Class 3 Public Primary Certification Authority - G5
subject name      : C=US, O=Symantec Corporation, OU=Symantec
Trust Network, CN=Symantec Class 3 ECC 256 bit SSL CA - G2
issued  on        : 2015-05-12 00:00:00
expires on        : 2025-05-11 23:59:59
signed using      : RSA with SHA-256
EC key size       : 256 bits
basic constraints : CA=true, max_pathlen=0
subject alt name  :
key usage         : Key Cert Sign, CRL Sign

DEBUG:   _iot_tls_verify_cert L#46   This certificate has no flags

DEBUG:   _iot_tls_verify_cert L#41
Verify requested for (Depth 0):

DEBUG:   _iot_tls_verify_cert L#43 cert. version     : 3
serial number     : 6B:37:E6:F1:E4:0E:19:2D:1A:D9:C1:EC:9F:2E:94:
26
issuer name       : C=US, O=Symantec Corporation, OU=Symantec
Trust Network, CN=Symantec Class 3 ECC 256 bit SSL CA - G2
subject name      : C=US, ST=Washington, L=Seattle, O=Amazon.com,
Inc., CN=*.iot.us-east-1.amazonaws.com
issued  on        : 2016-04-16 00:00:00
expires on        : 2017-04-17 23:59:59
signed using      : ECDSA with SHA256
EC key size       : 256 bits
basic constraints : CA=false
subject alt name  : iot.us-east-1.amazonaws.com, *.iot.us-east-1.
amazonaws.com
key usage         : Digital Signature
ext key usage     : TLS Web Server Authentication, TLS Web Client
Authentication

DEBUG:   _iot_tls_verify_cert L#46   This certificate has no flags

DEBUG:   iot_tls_connect L#227  ok
    [ Protocol is TLSv1.2 ]
    [ Ciphersuite is TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384 ]

DEBUG:   iot_tls_connect L#229     [ Record expansion is 29 ]

DEBUG:   iot_tls_connect L#234   . Verifying peer X.509
certificate...
DEBUG:   iot_tls_connect L#243  ok

DEBUG:   iot_tls_connect L#253   . Peer certificate
information    ...
```

```
DEBUG:   iot_tls_connect L#255       cert. version     : 3
      serial number     : 6B:37:E6:F1:E4:0E:19:2D:1A:D9:C1:EC:9F:
2E:94:26
      issuer name       : C=US, O=Symantec Corporation,
OU=Symantec Trust Network, CN=Symantec Class 3 ECC 256 bit SSL CA
- G2
      subject name      : C=US, ST=Washington, L=Seattle,
O=Amazon.com, Inc., CN=*.iot.us-east-1.amazonaws.com
      issued  on        : 2016-04-16 00:00:00
      expires on        : 2017-04-17 23:59:59
      signed using      : ECDSA with SHA256
      EC key size       : 256 bits
      basic constraints : CA=false
      subject alt name  : iot.us-east-1.amazonaws.com, *.iot.us-
east-1.amazonaws.com
      key usage         : Digital Signature
      ext key usage     : TLS Web Server Authentication, TLS Web
Client Authentication


Subscribing...
-->sleep
Subscribe callback
data_logger/ras_pi_fc2de3e3/sub hello from SDK QOS0 : 0
-->sleep
Subscribe callback
data_logger/ras_pi_fc2de3e3/sub hello from SDK QOS0 : 2
-->sleep
Subscribe callback
data_logger/ras_pi_fc2de3e3/sub hello from SDK QOS0 : 4
-->sleep
^C
#Go grab two fingers of scotch!
```

8 Configure and Test Rules

```
iot:Connection.Thing.ThingName
```