Dash Riprock

January 28, 2017

1 riprock

Dash Riprock[1], `riprock` for short, is a repository of example Python code that will provision an Amazon IoT Button[2] to send voice and text messages whenever the button is pressed.

AWS IoT[3] is used to accept and process button presses and to invoke an AWS Lambda[4] handler. Because AWS does not support voice messaging, the handler uses Twilio[5] for both voice and SMS messages.

The code is intended to be a brief guide to accomplishing the above entirely via software based on Boto3[6], the AWS SDK for Python; with it, it should not be necessary to use the Amazon Management Console for setup and operation. Note that the code is *not* intended to be general-purpose IoT Button / messaging software – it is very specifically tailored to its task (in other words, it's quite brittle).

2 Prerequisites

To successfully run this code, you will need:

- An Amazon AWS account.
- A user created in Amazon Identity and Access Management[7] (IAM), with sufficient permissions to manage IoT, IAM, S3, and SNS.

[1]The original brand-based IoT button from Amazon is called the Dash. Dash Riprock was Elly Mae Clampett's boyfriend in The Beverly Hillbillies.
[2]IoT Button: https://aws.amazon.com/iotbutton/
[3]AWS IoT: https://aws.amazon.com/documentation/iot/
[4]AWS Lambda: https://aws.amazon.com/documentation/lambda/
[5]Twilio: https://www.twilio.com
[6]Boto3: https://boto3.readthedocs.io/en/latest/
[7]AWS IAM: https://aws.amazon.com/iam/

- A *profile* for the above IAM user stored in your AWS credentials file, `~/.aws/credentials`.

- An Amazon IoT Button. Having said that, it should also be pointed out that the code includes and IoT Button simulator, so you can start without a real button.

3 Provisioning the IoT Button

3.1 Introduction

Amazon's guide *Getting Started with AWS IOT*.[8] steps through setting up an IoT button using the AWS web interface. This code roughly follows the guide, but accomplishes its goal through code rather than clicks on a web page.

The file `riprock.py` contains the command line processor which invokes the appropriate routines. `riprock` reads the contents of `riprock.conf` for certain configuration values. You will want to edit it and set `aws_profile_name` to the *profile* of the IAM user mentioned earlier.

3.2 About *Things*

IoT devices are termed *Things* within AWS IoT. *Things* are associated with *Thing Types*, which describe certain, but not necessarily all, attributes of a specific class of device. When a *Thing* is created, its *Thing Type* is specified. Attribute values are intended to be static over the life of a thing (e.g, a thing's serial number); they are not intended to be dynamic (e.g., a thing's current voltage level).

How does AWS know who a *Thing* belongs to and what it is allowed to do? Each *Thing* is authenticated via AWS device credentials associated with the *Thing's* unique X.509 certificate. Permissions granted to a *Thing* are determined by the IoT Policy associated with the *Thing's* X.509 certificate.

An Amazon IoT Button *Thing* is programmed to publish to the MQTT topic: `topic/iotbutton/button-serial-number`. An IoT Policy can be broad (e.g., allow any *Thing* to publish to any topic) or very constrained (e.g., allow IoT Button serial number 1234 to publish only to `topic/iotbutton/1234`.

[8] IoT Getting Started: http://docs.aws.amazon.com/iot/latest/developerguide/iot-gs.html

3.3 Provisioning AWS IoT

3.3.1 Provisioning

`riprock.py` does everything necessary to get an IoT Button up and communicating with AWS IoT.

1. `$ ./riprock.py create-policy`

   Creates an IoT Policy named `EM247PubSub`. This policy grants permission to publish to IoT and to subscribe to IoT topics.

   Strictly speaking, only publish permission is necessary for the IoT button. However, we also later use the button's credentials to listen to MQTT traffic for testing purposes. This latter use requires subscribe permission.

2. `$ ./riprock.py create-type`

   Creates a *Thing Type* named `EM247Button`.

3. `$ ./riprock.py create-button SERIALNUM`

   Creates an IoT *Thing* for the SERIALNUM provided. It is assigned to the `EM247Button` *Thing Type*.

   This must be done once for each IoT Button to be used.

4. `$ ./riprock.py create-certs SERIALNUM`

   Creates a 2048-bit RSA key pair and issues an X.509 certificate using the issued public key. AWS calls the X.509 certificate a *Principal* or a *Security Principal*.

   All of the credentials are stored in `./certs/`. `SERIALNUM` is prepended to each credentials file, thereby allowing one to identify the credentials for a particular device serial number.

   This must be done once for each IoT Button to be used.

   Note that the certificate and credentials are created by AWS IoT, but they are *not*, at this point, associated with any device or policy – they simply exist. Associations between them and devices/policies take place in subsequent steps.

5. `$ ./riprock.py attach-principal-policy SERIALNUM`

   Associate the device's X.509 certificate (i.e., its *Principal*) with the `EM247Pub` IoT Policy.

6. `$ ./riprock.py attach-button-principal SERIALNUM`

   Associate the *Thing* created by `create-button` with the device's X.509 certificate.

3.3.2 Testing (optional)

At this point, one can test their IoT setup, without using the IoT Button. `riprock` can emulate a button and deliver simulated clicks to IoT. Additionally, `riprock` contains a command that will print all MQTT traffic received by your IoT account. The two can be used in combination to confirm that the AWS side of things is setup properly.

1. In one terminal window, run the following command:

   ```
   $ ./riprock.py subscribe SERIALNUM
   ```

Replace `SERIALNUM` with the serial number used above when provisioning. This will print all MQTT traffic received by your IoT account. The SERIALNUM is used to get a valid set of credentials to use on contacting IoT as a MQTT client.

2. In another terminal window, simulate a button click by running the following command:

   ```
   $ ./riprock.py click --single SERIALNUM
   ```

   Shortly after this runs, you should see the message appear in the output of the `subscribe` command.

3.4 Provisioning the IoT Button

Once the credentials have been created, it is time to install the credentials in the IoT Button.

3.4.1 Provisioning

1. Each AWS account will have a dedicated IoT endpoint. Run `./riprock.py describe-endpoint` to get yours.
2. Connect to your IoT button over WiFi as described here: http://docs.aws.amazon.com/iot/latest/developerguide/configure-iot.html

   You will enter the following information on the web page that appears:

   a. *WiFi SSID*
   b. *WiFi Password*
   c. *IoT Certificate*
      This is located in `./certs/SERIALNUM-cert.pem`.
   d. *IoT Private Key*
      This is located in `./certs/SERIALNUM-private.pem`.

3.4.2 Testing

You can now take a moment to confirm that your button was properly provisioned and that AWS IoT is seeing its publishing activity. Use the `subscribe` command, as described above, for this purpose. When the IoT button is clicked, you should see its message appear shortly thereafter.

You can also use the IoT testing console to subscribe to the topic `iotbutton/+`.

4 What Happens When You Click

With the button working properly, it's helpful to take a moment and consider how all the parts come together and operate when the IoT Button is clicked. Upon button click:

1. Button wakes up and connect to your WiFi.
2. Button uses TLS to connect to your IoT endpoint using the certificate and private key.
3. AWS IoT authenticates the button using the X.509 certificate, thereby identifying it as a specific *Thing*.
4. The IoT Policy associated with the the X.509 certificate determines what the *Thing* can do. It this instance, this *Thing* can publish to IoT.
5. Button uses MQTT to publish to the topic `topics/iotbutton/SERIALNUM`.
6. Button disconnects from WiFi.

5 The Lambda Handler

At this point, the button is working, but it's not doing anything. This second half of the equation is taken care of by the AWS Lambda Handler.

5.1 The Handler Implementation

All of the code for the handler is located in the directory: `riprock/lambda`. ** All commands shown in this section are issued within `riprock/lambda`. **

The handler is implemented in `notifier.py`. Its messaging behavior is controlled by the contents of a file obtained from S3 at runtime. The location of the file is communicated to the handler through two environment variables (`BUCKET_NAME`, `KEY_NAME`).

5.2 IAM Role

The Lambda Handler requires an IAM Role that gives it permission to read S3 buckets and to write CloudWatch logs.

Such a role can be created by running `make create_role`, which invokes `iamrole.py`, which does the following:

- Creates an IAM Role named `EM247-Lambda`.
- Associates two standard IAM Policies (`AmazonS3ReadOnlyAccess` and `AWSLambdaBasicExecutionRole`) with the role.

5.3 Creating the Handler

The `lambda-uploader` tool is used to package the Lambda Handler implementation. Its behavior is controlled by `lambda.json`. Invoke it by running `make create_function`, which:

- Names the handler `EM247-Notifier`.
- Creates a virtualenv installs the `twilio`, `dotmap`, and `PyYAML` packages in it.
- Puts the handler implementation (i.e., `notifier.py`) and the contents of the virtualenv in a zip file. AWS Lambda calls this a Deployment Package.
- Uploads the deployment package to Lambda.
- Provides Lambda with the runtime values for the `BUCKET_NAME` and `KEY_NAME` environment variables, thereby specifying the S3 location of the messaging behavior.
- Associates the `EM247-Lambda` role with the handler.

5.4 Configuring the Messaging Behavior

Prior to using the handler, a YAML file specifying the messaging behavior must be uploaded to S3. This is done by running `make upload_config`.

The file `notifier.yml` contains an example of a messaging behavior specification. You will need to edit the file and, at a minimum, provide values for: `twilio`, `notifier.person`, `notifier.voice_numbers`, and `notifier.sms_numbers`. You are free to change the message content, but doing so isn't necessary to get started.

5.5 Creating an IoT Rule to Invoke the Handler

At this point, the Lambda handler isn't associated with anything - i.e., it will never be called. Running `make create-rule` will create an IOT rule which invokes the Lambda handler when the button on an IOT button is pressed.

5.6 Testing the Handler

The handler can be tested locally as well as on Lambda. Testing sends a fake IoT Button event to the handler.

5.6.1 Local Testing

Note that, even when run locally, the messaging behavior is obtained from S3. Therefore, prior to local testing:

- The YAML file must be uploaded to S3 (`make upload_config`).
- The IAM role must be created (`make create_role`). This is needed because the role grants the handler S3 read access.

Once the pre-requisites are in place, `make localtest` should result in messages being sent. Note that `localtest` sets the environment variables mentioned earlier prior to invoking the handler.

5.6.2 Lambda Testing

To invoke the handler on Lambda, run `make lambdatest`. This will use the AWS CLI to invoke the handler. Note that one can also go to the Lambda console and invoke the handler directly.