# Design and build secure IoT solutions, Part 1: Securing IoT devices and gateways

## From authentication, including certificate-based authentication, to authorization, to application ID validation

Amitranjan Gantait
Joy Patra
Ayan Mukherjee

February 28, 2018
(First published May 16, 2016)

In this 3 part series of developing secure IoT solutions, the authors introduce IoT vulnerabilities and design challenges for secure IoT apps and then provide tested techniques for securing devices and gateways. For example, they discuss device authentication, including certificate-based authentication, device authorization, and application ID validation.

View more content in this series

The Internet of Things (IoT) introduces huge opportunities for businesses and consumers, especially in the areas of healthcare, warehousing, transportation, and logistics. Along with this widespread adoption, developers face new challenges to make sure that IoT applications are sufficiently secure because these applications handle a lot of sensitive data. Many security breaches have already been reported for IoT solutions, so developers must focus on building security into their IoT applications when they design and implement such solutions.

This series of articles focuses on the architectural tiers of an IoT application that is based on IBM cloud platforms. The articles in this series describe a solution-based approach to minimizing security risks in IoT applications by using services that are readily available in IBM cloud platforms. The articles provide tested techniques for securing IoT applications.

Part 1 of this series describes the various approaches for securing devices or gateways. Part 2 focuses on the security aspects of the network and transport tier, which includes the IBM Watson IoT Platform. Part 3 provides the security requirements for the application tier, and an implementation approach for an analytics IoT application that is created in the IBM Bluemix platform.

*“ As IoT applications collect more and more previously unexposed—often private—data, and allow access to various control functions over the internet, security becomes a major challenge. ”*

## IoT security basics

IoT solutions involve a complex network of smart devices, such as vehicles, machines, buildings, or home appliances, that are embedded with electronics, software, sensors, and network connectivity, which enable these "things" to collect and exchange data. The "things" in the Internet of Things allows developers to provide a broad range of new services based on these cloud-enabled, connected physical devices. As IoT applications collect more and more previously unexposed—often private—data, and allow access to various control functions over the internet, security becomes a major challenge. Therefore, an IoT application must:

- **Prevent system breaches or compromises**.
  Each tier of the IoT application must implement effective preventive measures to keep the hackers out. For example, you need to ***harden*** the device to make sure communication from the device to the cloud is secure.
- **Support continuous monitoring**.
  Even the best secured systems still leave many vulnerabilities. Also, today's best secured solution (both hardware and software) might not be good enough to prevent attacks in the future. Therefore, you must supplement your security measures with continuous monitoring and constant upgrading of the system to protect against the latest forms of attack.
- **Be resilient**.
  Finally, if a breach does occur, damage must be minimized and the system must recover as quickly as possible.

### IoT vulnerabilities

Developers have so many ways that they can apply IoT technologies to create IoT solutions. They can create a simple home monitoring system that provides alerts to smartphones and smart watches, or they can create complex healthcare systems that collect data and control a network of patient devices—and many opportunities for solutions we can't yet imagine.

But connecting objects like cars, homes, and machines exposes a lot of sensitive data, such as the location of people in a building or medical records of patients. This data must be protected in accordance with the key information security principles, the CIA triad: *confidentiality*, *integrity*, and *availability*.

Read the article that describes the connected car vulnerability, "Hackers Remotely Kill a Jeep on the Highway."

Any device that has network connectivity is vulnerable. Personal data that is collected by IoT devices is always of value to data hackers and identity thieves. Also, a cyber attack on IoT

solutions has the potential to cripple physical services and infrastructure. For example, hackers attacked a Jeep Cherokee while it was being driven on a highway. Therefore, secure IoT applications are not only critical for the reputation of the enterprise, but also for the physical health and well-being of the clients and users of the solution.

> Read the articles that describe these vulnerabilities in more detail: "It's Insanely Easy to Hack Hospital Equipment" and "9 baby monitors wide open to hacks...".

In a two-year study, security vulnerabilities of connected devices in hospitals were demonstrated. One of the main problems they found concerned embedded web services that allowed devices to communicate with one another and feed digital data directly to patient medical records—without an appropriate authentication or encryption mechanism! A similar study demonstrated the security vulnerabilities of baby monitors.

### IoT security design challenges

While the importance of IoT security is widely understood and agreed upon, the actual design and implementation of IoT security brings new challenges and opportunities for creativity. In the design of most any app, developers always face a trade-off between security and usability. For IoT solutions, it becomes even more problematic. IoT devices often have limited computing power and memory capacity, making it difficult to use complex cryptographic algorithms that require more resources than the devices provide.

Another challenge is updating IoT devices with regular security fixes and updates. Rolling out security patches to all devices at once can be very difficult in unreliable, low-bandwidth device networks, and many existing security measures, such as web browser security, might not be available to IoT applications.

> You can read more about MQTT and CoAP on the sites for these open standards: MQTT or CoAP.

In addition, security mechanisms might need to be developed or enhanced for new protocols that are designed specifically for the Internet of Things, such as Message Queuing Telemetry Transport (MQTT) and Constrained Application Protocol (CoAP). Therefore, it is especially important to factor in security considerations from the very beginning when you design IoT apps.

## Developing secured IoT applications

Most IoT solutions consist of three main tiers. IoT solution components that run in each tier need to incorporate specific security measures to protect against various vulnerabilities.

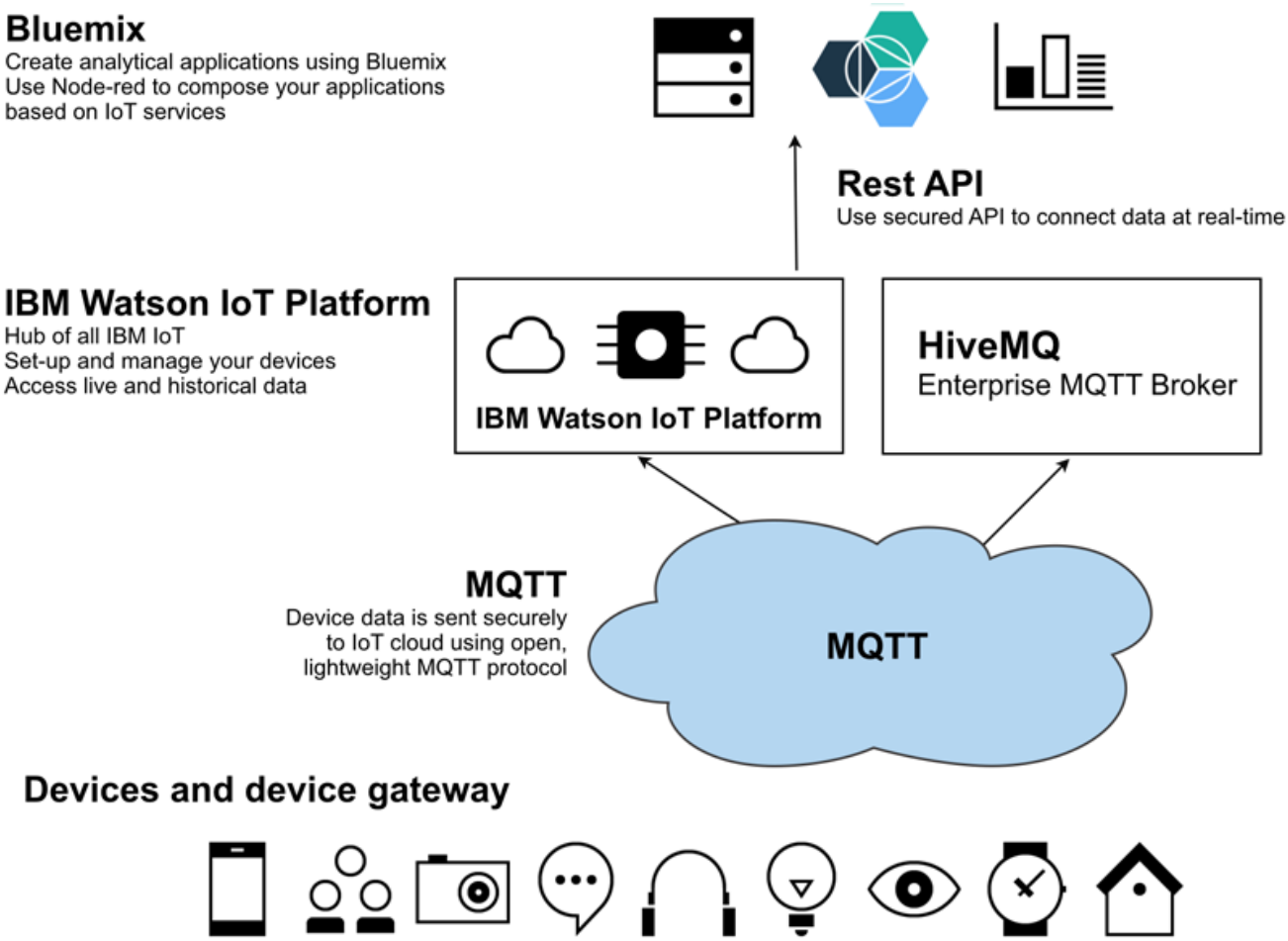
- **Devices/Gateways tier:** Protect against a "fake" server that sends malicious commands, or protect against a hacker that tries to listen to private sensor data being sent from the devices. Security considerations for this tier are discussed in Part 1 (this article).
- **Network/Transport tier:** Protect against a "fake" device that sends false measurements that might corrupt the data that is being persisted in the application. Security considerations for this tier will be discussed in Part 2.

- **Applications tier:** Protect against the invalid use of data, or protect against the manipulation of analytical processes that are running in the application tier. Security considerations for this tier will be discussed in Part 3.

The application layer of an IoT device provides the largest attack surface to hackers. The application layer includes any application that has connectivity with the IoT device, which can include local web applications, cloud-based applications, and mobile apps.

Application security must be an intrinsic part of the software development lifecycle (SDLC) for all IoT applications, particularly during the design, development, and testing stages. Within the planning or design stage of an IoT application, there must be a formal "top-to-bottom" assessment of the planned application's security and privacy requirements.

The diagram below shows the three tiers of a typical IoT application that uses IBM Watson IoT Platform in the network/transport tier and the IBM Bluemix cloud platform in the application tier.




The following table briefly describes each tier and the security considerations developers must focus on.

| Tier | Description | Security considerations |
|---|---|---|

| | | |
|---|---|---|
| **Application** | IoT applications are deployed in the Bluemix platform. | • Application security<br>• Secured API call for IBM Watson IoT Platform<br>• Node-RED security<br>• Decrypting messages<br>• Message checksum verification |
| **Network/Transport** | IBM Watson IoT Platform provides the MQTT-based messaging platform for the IoT applications. | • Authenticating devices (only trusted devices can send data)<br>• Authorization<br>• API security<br>• Security configuration<br>• Secured transport |
| **Devices/Gateways** | Devices (directly or through gateways) publish the sensor data to IBM Watson IoT Platform and receive instructions to execute control functions. | • Authentication<br>• Message payload encryption<br>• Certificate provisioning and verification<br>• Secured MQTT transport<br>• Secure booting<br>• Firewalls<br>• Firmware updates and patches |

## Securing devices

Device security focuses on ensuring that a trusted set of devices are used in the solution, and that those devices can trust the broker or application that is sending the control commands. This article discusses various kinds of security mechanisms that can be used to establish that trust.

In addition, we developed (in JavaFX) a device simulator program that demonstrates these security mechanisms:

- User ID/Password authentication
- One time password (OTP) authentication
- Server unique ID authentication
- Message payload authentication

See the following screen capture of the device simulator.

You can download the code for the device simulator program, and follow the instructions in the readme file to build and run it locally.

Get the code for the device simulator

MQTT is the most popular messaging protocol for IoT devices and applications, and it is supported by many key players in the IoT field. MQTT provides a lightweight, easy-to-use communication protocol for IoT solutions.

MQTT itself specifies few security mechanisms, but all common implementations support state-of-the-art security standards, such as SSL/TLS for transport security. MQTT does not enforce the use of a particular security approach for its applications, but instead leaves that to the application designer. Therefore, IoT solutions can be based on application context and specific security requirements.

Most deployments of MQTT use transport layer security (TLS), so the data is encrypted and its integrity is validated. Similarly, most implementations of MQTT (including the one in IBM Watson IoT Platform) also use authorization features in the MQTT server to control access.

In addition to the device simulator program, we provided a broker application client that displays the MQTT messages that are received from the device and sends sample commands to the device. This broker application client sample application generates the OTP key for device authentication and sends application unique ID for application verification by the devices. This client generates command messages —both valid and invalid messages—in order to test different scenarios.

You can download the code for the broker application client, and follow the instructions in the readme file to build and run it locally.

Get the code for the broker application client

## Device authentication

Authentication is part of the transport and application level security in MQTT. On the transport level, TLS can guarantee authentication of the client to the server by using client certificates, and of the server to the client by validating the server certificate. On the application level, the MQTT protocol provides user name/password authentication.

Developers can use several approaches to ensure the right device is registered with the broker. Selecting the right approach depends on the security needs of the solution, and on the capability of the device to run the needed approach.

The sections below describe some of these approaches. Eclipse Paho is used as the MQTT client library for the code samples.

## Authenticating with a user name and password

MQTT protocol provides `username` and `password` fields in the CONNECT message for device authentication. A client must send a user name and password when it connects to an MQTT broker.

The user name is a UTF-8 encoded string, and the password is binary data. Each has a 65535 byte max. MQTT protocol does not encrypt the user name or password, and unless transport encryption is used, they are sent in clear text format.

## Listing 1. User name and password fields

```
try {
    MqttClient securedClient = new MqttClient(broker, clientId, persistence);
    MqttConnectOptions connOpts = new MqttConnectOptions();
    connOpts.setCleanSession(true);
    connOpts.setUserName(userName);
    connOpts.setPassword(password.toCharArray());
    System.out.println("Connecting to broker: "+broker);
    securedClient.connect(connOpts);
    System.out.println("Connected");
} catch(MqttException me) {
    System.out.println("reason "+me.getReasonCode());
    System.out.println("msg "+me.getMessage());
    System.out.println("loc "+me.getLocalizedMessage());
    System.out.println("cause "+me.getCause());
    System.out.println("excep "+me);
    me.printStackTrace();
}
```

## Authenticating with an access token

If the client has successfully retrieved an access token, it can be sent to the broker in the CONNECT message by using the `password` field. The user name can then be a special string for recognizing the access token. The size limit of a password in MQTT is 65535 bytes, so the token can't be longer than that limit.

The broker can use the token to perform various validations, such as:

- Check the validity of the signature from the token
- Check whether the expiration date of the token has already passed
- Check the authorization server to see whether the token was revoked

The same validations that are used when the device connects to the MQTT broker can be used when applications publish or subscribe. However, when publishing or subscribing, the broker must also authorize the application. This authorization can be done in two ways:

- The token includes the authorization for the client in the scope claim.
- The broker has a third-party source, such as a database or LDAP directory, look up the authorizations for the client.

IBM Watson IoT Platform applications can be authenticated by application ID, key, and token. The IoT application key and token can be generated during application registration and can be used when it connects to IBM Watson IoT Platform, as shown in the example below:

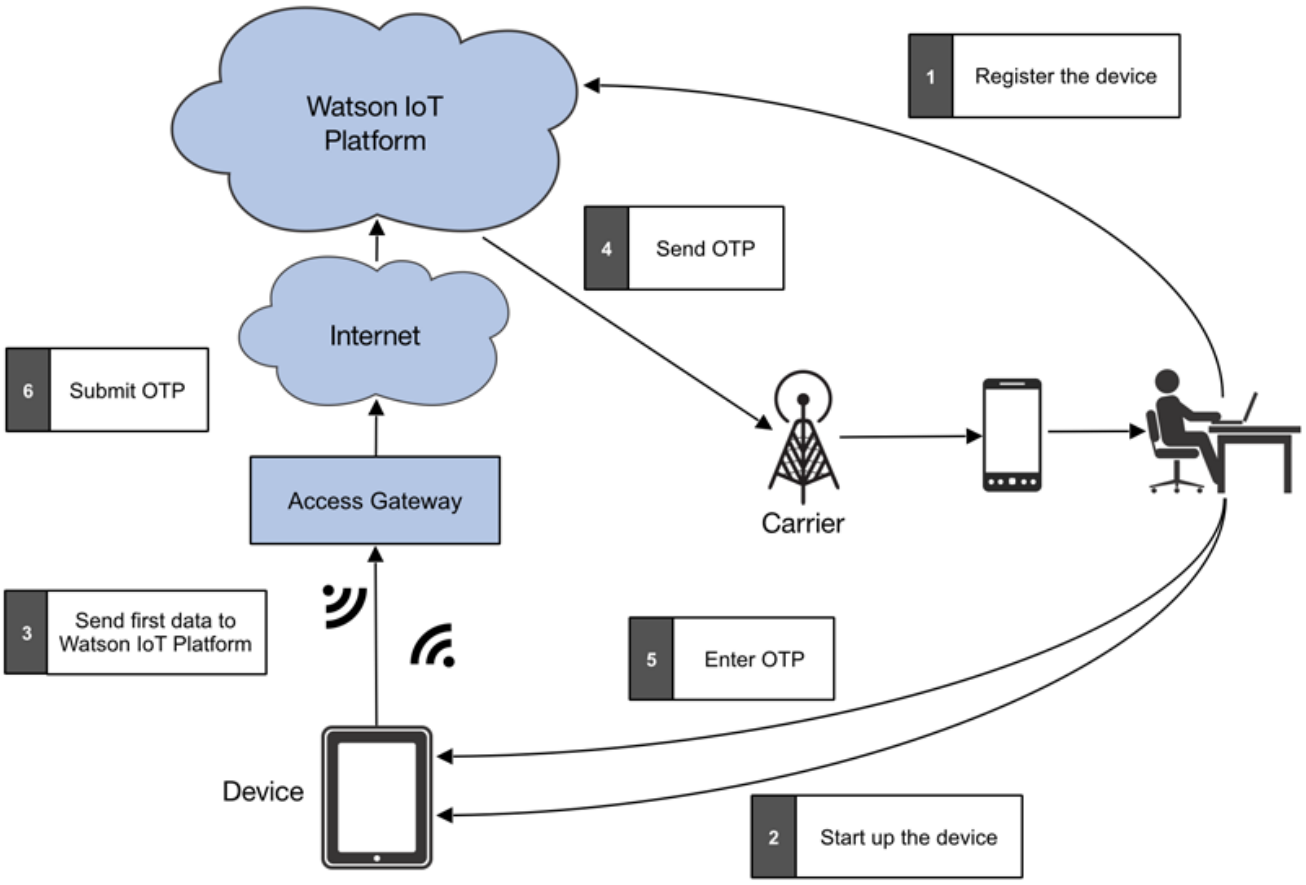
## Listing 2. Application keys and tokens

```
App properties

# A unique id you choose it by yourself, maybe, abcdefg123456
appid=<Your_application_id>

# The key field from App Keys info you copied previously
key=<Key>

# The Auth Token field from App Keys info you copied previously
token=<Token>

App code during connection:

strAuthMethod = props.getProperty("key");
strAuthToken = props.getProperty("token");

handler = new AppMqttHandler();
handler.connect(serverHost, clientId, strAuthMethod, strAuthToken, isSSL);
```

## Authenticating with one-time password (OTP) authentication

In addition to the MQTT-provided authentication mechanisms, IoT applications might need to implement additional security to identify the proper device. This article describes an approach for implementing OTP-based authentication for these situations. OTP authentication can be a useful mechanism for protecting a device from improper use by eliminating the risk of unauthorized users gaining access.

With this method, only authenticated users are able to start data communication with the IoT application after device startup. Since not all devices have keypad entry capability, a simple property switch can be implemented to enable or disable this security scheme based on the device type. If OTP authentication is enabled, the device sends an OTP request to the IoT broker application after startup by using normal MQTT messaging. The detailed flow is shown in the graphic below.




Listing 3 shows how OTP authentication can be turned on and off by using a device property. If OTP authentication is enabled, the device sends an OTP request to the IoT broker application after startup by using normal MQTT messaging.

## Listing 3. OTP request

```
 // Create the request for OTP
 JSONObject idObj1 = new JSONObject();
 try {
  idObj1.put("event", "server_otp_request");
  idObj1.put("deviceId", deviceIdentifier);
 } catch (JSONException e1) {
  System.out.println("Exception occurred");
  e1.printStackTrace();
 }
 new SendMessageToServer("server_otp_request", idObj1).start();
 System.out.println("otp request sent....");
}
```

The IoT app generates an OTP, sends it to the device owner separately and sends a notification to the device, as shown in the following listing.

## Listing 4. Generate OTP

```
   otp = IOTSecurityUtil.generateOTP();
```

```
  JSONObject jsonObj = new JSONObject();
  try {
   jsonObj.put("cmd", "server_otp_response");
   jsonObj.put("otp", otp);
   jsonObj.put("appid", strAppId);
   jsonObj.put("time",
               new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date()));

      // Server starts a timer of 5 mins during which the
      // OTP is valid.
   task = new TimeOutTask();
   t = new Timer();
   t.schedule(task, 30000000L);

  } catch (JSONException e) {
   e.printStackTrace();
  }
  System.out.println("Sending otp  -  " + otp);

  // Publish command to one specific device
  // iot-2/type/<type-id>/id/<device-id>/cmd/<cmd-id>/fmt/<format-id>
  new SendMessageToDevice(strDeviceId, "server_otp_response", jsonObj)
    .start();
```

As shown in Listing 5, the OTP is entered into the device and the device sends it to the broker app. The broker app validates the OTP sent by the device and sends a success/failure (incorrect OTP or timeout) message to the device. The device can retry OTP authentication based on a retry count from the configuration.

If OTP authentication is not successful even after a retry, the application shuts down. If OTP authentication is not enabled, the device will skip OTP authentication after startup.

## Listing 5. Validate OTP authentication

```
if (receivedOTP.equals(otp)) {
    if (task.isTimedOut) {
 // User took more than 100 seconds and hence the OTP is invalid
 System.out.println("Time out!");
 otpValidated = false;
 otpTimeOut = true;
    } else {
 System.out.println("OTP validated..");
 otpValidated = true;
 otpTimeOut = false
    }
} else {
    System.out.println("Incorrect OTP..");
    otpValidated = false;
    otpTimeOut = false;
}

JSONObject otpRespObj = new JSONObject();
try {
 otpRespObj.put("cmd", "server_otp_validate");
 otpRespObj.put("isOTPValid", String.valueOf(otpValidated));
 otpRespObj.put("isTimeOut", String.valueOf(otpTimeOut));
 otpRespObj.put("appid", strAppId);
 otpRespObj.put("time", new SimpleDateFormat(
     "yyyy-MM-dd HH:mm:ss").format(new Date()));
} catch (JSONException e) {
 e.printStackTrace();
}
System.out.println("Result of OTP validation -  " + otpValidated);
```

```
// Publish command to one specific device
    new sendMessageToDevice(strDeviceId, "server_otp_validate",
 otpRespObj).start();

 }
```

# Certificate-based authentication

Some brokers (such as the one included in IBM Watslno IoT Platform and such as HiveMQ, which is an MQTT broker that can be used to enable M2M and IoT capabilities for enterprises) support device certificates that can be used by the broker as part of a mutual authentication process. In applications where security requirements are very strict and devices can provision certificates, you should consider this type of authentication.

In this article, we use IBM Watson IoT Platform and HiveMQ to demonstrate certificate-based, two-way SSL authentication. They both use standard MQTT as a protocol for device communication.

## Certificate-based authentication in Watson IoT Platform

To configure certificate-based authentication in Watson IoT Platform, complete these steps.

1. **Create certificate extension file for the certificate authority (CA) certificates.** Create the `ext.cfg` file, and copy the following code into the file:
   ```
   [ req ]
   attributes = req_attributes
   req_extensions = v3_ca
   prompt = no
   [ req_attributes ]
   [ v3_ca ]
        basicConstraints = CA:true
   ```
2. **Create certificate extension file for servers.** Create the `srvext.cfg` file, and copy the following code into the file:
   ```
   [ req ]
   req_extensions = v3_req
   [ v3_req ]
   subjectAltName = DNS:<ORG>.messaging.internetofthings.ibmcloud.com
   ```
3. **Generate the root CA key and certificate.** In our example, we used a self-signed certificate using the `openssl` commands:
   ```
   openssl genrsa -aes256 -passout pass:<Password> -out rootCA_key.pem 2048
   openssl req -new -sha256 -x509 -days 3560 -subj "/C=IN/ST=WB/L=KOL/O=<ORG>/OU=<ORG>
   Corporate/CN=<ORG> Root CA" -extensions v3_ca -set_serial 1 -passin pass:<Password>
   -key rootCA_key.pem -out rootCA_certificate.pem -config openssl.cnf
   ```
4. **Generate the intermediate key, certificate, intermediate CA, and signed by root CA.** Again, we used these `openssl` commands:
   ```
   openssl genrsa -aes256 -passout pass:<Password> -out intermediateCA_key.pem 2048
   openssl req -new -sha256 -days 3650 -subj "/C=IN/ST=WB/L=KOL/O=<ORG>/OU=AIMDEV/
   CN=<ORG> Intermediate CA" -passin pass:<Password> -key intermediateCA_key.pem -out
   intermediateCA_crt.csr -config openssl.cnf
   ```

```
openssl x509 -days 3650 -in intermediateCA_crt.csr -out intermediateCA_crt.pem -req
-sha256 -CA rootCA_certificate.pem -passin pass:<Password> -CAkey rootCA_key.pem -
set_serial 13 -extensions v3_ca -extfile ext.cfg
```

5. **Upload the Root CA and the Intermediate CA certificates to Watson IoT Platform.** To enable TLS with client-side certificates, you need to upload the CAs to Watson IoT Platform, which authenticates the signature of the messaging server and the client-side certificates. For information on how to configure certificates in Watson IoT Platform, refer to the documentation.

6. **Set up the device certificate authentication** by following these steps:
   a. Generate device type key and certificate signed by the intermediate CA. Use these `openssl` commands:
   ```
   openssl genrsa -aes256 -passout pass:<Password> -out SecuredDevice_key.pem
   2048
   openssl req -new -sha256 -days 3560 -subj "/C=IN/ST=WB/L=KOL/
   O=<ORG>/OU=IMATEST/CN=d:<DEVICE TYPE>:" -passin pass:<Password> -key
   SecuredDevice_key.pem -out SecuredDevice_crt.csr -config openssl.cnf
   openssl x509 -days 3650 -in SecuredDevice_crt.csr -out SecuredDevice_crt.pem
   -req -sha256 -CA intermediateCA_crt.pem -passin pass:<Password> -CAkey
   intermediateCA_key.pem -extensions v3_req -extfile srvext.cfg -set_serial 131
   ```

   b. Generate device key and certificate signed by the intermediate CA. Use these `openssl` commands:
   ```
   openssl genrsa -aes256 -passout pass:<Password> -out
   SecuredDevice_0400213_key.pem 2048
   openssl req -new -sha256 -days 3560 -subj "/C=IN/ST=WB/L=KOL/O=<ORG>/
   OU=IMATEST/CN=d:<DEVICE TYPE>:<DEVICE ID>" -passin pass:<Password> -key
   SecuredDevice_0400213_key.pem -out SecuredDevice_0400213_crt.csr -config
   openssl.cnf
   openssl x509 -days 3650 -in SecuredDevice_0400213_crt.csr -out
   SecuredDevice_0400213_crt.pem -req -sha256 -CA intermediateCA_crt.pem -passin
   pass:<Password> -CAkey intermediateCA_key.pem -extensions v3_req -extfile
   srvext.cfg -set_serial 132
   ```

   c. Create pkcs12 key for the device certificate that will be used in the MQTT keystore. Use this `openssl` command:
   ```
   openssl pkcs12 -export -inkey SecuredDevice_0400213_key.pem -in
   SecuredDevice_0400213_crt.pem -out SecuredDevice_0400213.p12 -password
   pass:<Password>
   ```

   d. Upload the generated CA certificates to Watson IoT Platform like you did before.
   e. Create keystore and add the device pkcs12 key that you generated for the device. Use these keytool commands:
   ```
   keytool -keystore keystore.jks -genkey -alias keystore -storepass <Password>
   ```

```
keytool -importkeystore -srckeystore SecuredDevice_0400213.p12 -destkeystore
keystore.jks -srcstoretype pkcs12 -deststorepass <Password> -deststoretype jks
-srcstorepass <Password>
```

f. Set up the keystore in the Java MQTT client.

```
// SSL Configuration

// Keystore
    private static final String KEYSTORE_PW = "123456";
    private static final String KEYSTORE_TYPE = "jks";
    sslClientProps.put("com.ibm.ssl.keyStore", "C:\\demo\\new\\keystore.jks");
    sslClientProps.put("com.ibm.ssl.keyStoreType", KEYSTORE_TYPE);
    sslClientProps.put("com.ibm.ssl.keyStorePassword", KEYSTORE_PW);
```

7. **Complete the set up of certificate-based authentication for the message broker** by completing these steps:
   a. Generate MQTT server key and certificate using the root CA as the issuer. Use these `openssl` commands:

```
openssl genrsa -aes256 -passout pass:<Password> -out mqttServer_key.pem 2048
openssl req -new -sha256 -days 3560 -subj "/C=IN/ST=WB/L=KOL/O=<ORG>/
OU=IMATEST/CN=<ORG>.messaging.internetofthings.ibmcloud.com" -passin
pass:<Password> -key mqttServer_key.pem -out mqttServer_crt.csr -config
openssl.cnf
openssl x509 -days 3560 -in mqttServer_crt.csr -out mqttServer_crt.pem
-req -sha256 -CA rootCA_certificate.pem -passin pass:<Password> -CAkey
rootCA_key.pem -extensions v3_req -extfile srvext.cfg -set_serial 11
```

   b. **Upload the messaging service certificates to Watson IoT Platform.** Before you upload the messaging service certificates, make sure that you have already uploaded the Root CA certificate in the CA certificates section.

   c. **Create a truststore using the java keytool and add MQTT server certificate** that you previously uploaded into the truststore.

```
keytool -keystore truststore.jks -genkey -alias truststore -storepass
<Password>
keytool -import -trustcacerts -file mqttServer_crt.pem -keystore
truststore.jks -storepass <Password> -noprompt
```

   d. **Set up truststore in the MQTT client.**

```
// SSL Configuration

// Truststore

    private static final String TRUSTSTORE_PW = "123456";
    private static final String TRUSTSTORE_TYPE = "jks";
    sslClientProps.put("com.ibm.ssl.trustStore", "C:\\demo\\new\\truststore.jks");
    sslClientProps.put("com.ibm.ssl.trustStoreType", TRUSTSTORE_TYPE);
    sslClientProps.put("com.ibm.ssl.trustStorePassword", TRUSTSTORE_PW);
```

## Certificate-based authentication in HiveMQ

If you don't already have HiveMQ, you should install it if you want to follow along. It can be easily downloaded, installed, and started by following the steps outlined at hivemq.com. Optional plug-ins can be used to retrieve the retained messages from HiveMQ. The MQTT client works with HiveMQ in the same way it does with IBM Watson IoT Platform.

The device simulator program and the MQTT broker application client demo demonstrate certificate-based authentication. Both the device and the application perform mutual certificate verification with HiveMQ.

You can download the code for certificate authentication demo, and follow the instructions in the readme file to build and run it locally.

Get the code for the certificate authentication demo

### Generating the certificate

Follow the steps below to generate a certificate for authentication. This procedure uses the ***keytool*** that is bundled with Java Runtime Environment.

1. **Generate device key and keystore**
   ```
   keytool -genkey -alias iotdevice1 -keyalg RSA -keypass devicepass -storepass devicepass -keystore iot_device_keystore.jks -storetype jks
   ```
2. **Export device certificate from keystore**
   ```
   keytool -export -alias iotdevice1 -storepass devicepass -file iotdevice1.cer -keystore iot_device_keystore.jks
   ```
3. **Add device certificate into broker truststore**
   ```
   keytool -import -v -trustcacerts -alias iotdevice1 -file iotdevice1.cer -keystore iot_broker_truststore.jks -keypass devicepass -storepass brokerpass -storetype jks
   ```
4. **Generate broker key and keystore**
   ```
   keytool -genkey -alias broker -keyalg RSA -keypass brokerpass -storepass brokerpass -keystore iot_broker_keystore.jks -storetype jks
   ```
5. **Export broker certificate**
   ```
   keytool -export -alias broker -storepass brokerpass -file broker.cer -keystore iot_broker_keystore.jks
   ```
6. **Add the certificate into device truststore**
   ```
   keytool -import -v -trustcacerts -alias broker -file broker.cer -keystore iot_device_truststore.jks -keypass brokerpass -storepass brokerpass -storetype jks
   ```

The same approach can be extended for multiple devices. All device certificates must be added in the broker's `truststore`, and the broker's certificate must be in the `truststore` of all devices.

### Configuring HiveMQ for certificate-based authentication

As shown in Listing 6, HiveMQ is configured with the broker `keystore` and the broker `truststore` by using the `config.xml` file.

## Listing 6. Hive MQ configured with the broker `keystore` and `truststore`

```
<tls-tcp-listener>
            <port>8883</port>
            <bind-address>0.0.0.0</bind-address>
            <tls>
                <keystore>
                    <path><Your path>\iot_broker_keystore.jks</path>
                    <password>brokerpass</password>
                    <private-key-password>brokerpass</private-key-password>
                </keystore>
                <truststore>
                        <path>C:\certificates\iot_broker_truststore.jks</path>
                        <password>brokerpass</password>
                </truststore>
                <client-authentication-mode>REQUIRED</client-authentication-mode>
            </tls>
        </tls-tcp-listener>
```

The MQTT handler is then configured with the device `keystore` and `truststore` as shown in the following listing:

## Listing 7. MQTT handler configured with device `keystore` and `truststore`

```
if (isSSL) {
  java.util.Properties sslClientProps = new java.util.Properties();

            // Set the SSL properties
  sslClientProps.setProperty("com.ibm.ssl.protocol", "TLSv1.2");
  sslClientProps.setProperty("com.ibm.ssl.contextProvider", "SunJSSE");

            // Set the keystore properties
  sslClientProps.setProperty("com.ibm.ssl.keyStore", "<Your_path>/iot_device_keystore.jks");
            sslClientProps.setProperty("com.ibm.ssl.keyStorePassword", "devicepass");
  sslClientProps.setProperty("com.ibm.ssl.keyStoreType", "JKS");
  sslClientProps.setProperty("com.ibm.ssl.keyManager", "SunX509");

            // Set the trust store properties
  sslClientProps.setProperty("com.ibm.ssl.trustStore", "<Your_path>/iot_device_truststore.jks");
  sslClientProps.setProperty("com.ibm.ssl.trustStorePassword", "brokerpass");
  sslClientProps.setProperty("com.ibm.ssl.trustStoreType", "JKS");
  sslClientProps.setProperty("com.ibm.ssl.trustManager", "SunX509");

            // 'options' is an instance of MqttConnectOptions
  options.setSSLProperties(sslClientProps);
 }
```

### Authenticating with client certificates

While certificate-based authentication provides a high level of security for some applications, implementing this approach is not easy, and managing certificate lifecycles for numerous devices can be costly. But if an enterprise already has similar infrastructure in place and manages all MQTT components (devices and brokers), this approach might be considered.

MQTT can use TLS for transport encryption. To use TLS, the server must have a public/private key pair. When the TLS handshake takes place, clients need to validate the X509 certificate of the server—that also contains the public key of the server—before a secure connection can be established.

In addition to the server certificates, clients can also have a unique public/private key pair to implement the TLS handshake protocol. The client sends its certificate (which includes the public key of the client) as part of the TLS handshake after the server certificate is validated. The server is then able to verify the identity of the client and can cancel the handshake if the verification of the client certificate fails. This practice allows authentication of the client before a secure connection is established.

Implementing client certificates has the following advantages:

- Verification of the identity of MQTT clients
- Authentication of MQTT clients at the transport level
- Locking out invalid MQTT clients before MQTT CONNECT messages are sent

If you use client certificates, only trusted clients can establish a secured connection. This configuration can save resources on the broker side, especially if costly MQTT authentication mechanisms (like database lookups or web service calls) are used on the broker side. Because the authentication takes place in the TLS handshake, the authentication is done before a connection is established.

While X509 client certificates give an extra layer of security, this type of authentication comes at a cost. The MQTT client provisioning is more complex with client certificates and a certificate revocation mechanism is needed.

- **Deploying client certificates:** To use client certificates, a provisioning process must be defined. This process is defined when enterprises have control over their devices and have a well-defined firmware update process. Client certificates can be provisioned during the firmware update. Other considerations are required to manage the lifecycle (including expiration) of certificates in the devices.
- **Certificate revocation:** If a client certificate can no longer be trusted (for example, if it was leaked), then it's important to invalidate the client certificate. If the certificate was leaked and malicious clients are using the certificate, the server needs a way to identify the invalid certificate and prohibit clients connecting with that certificate.

## Advanced security policies in Watson IoT platform

IBM Watson IoT platform now supports several new security policies that can be enabled and configured to meet varying security needs of the IoT solution.

### Connection security

Connection between devices and Watson IoT platform can now be secured using different approaches. By using a connection security policy, you can set the default security level that is applied to all devices. You can also define custom connection security to apply different security level to your special devices. You can configure these security levels for either default or custom connection security:

- TLS Optional

- TLS with Token Authentication
- TLS with Client Certificate Authentication
- TLS with Client Certificate and Token Authentication
- TLS with either Client Certificate or Token Authentication

### Configuring rules for accessing devices

In Watson IoT Platform, in addition to configuring generic or device-specific connection policies, you can also configure rules that enable or prevent access for specific device or devices from a country.

- Blacklists block access from specific IP addresses and countries. Activating a blacklist disables an active whitelist.
- Whitelists allow access from specific IP addresses and countries. Activating a whitelist disables an active blacklist.

Refer to the IBM Bluemix documentation about setting up security polices in Watson IoT platform, including configuring blacklists and whitelists.

## Device authorization

An authorization mechanism ensures that there is no data leakage between two devices.

MQTT is a topic-based publish/subscribe protocol. Every message is published on a named topic, and every subscription has a topic filter that can include wildcards. So, authorization is in terms of publishing/subscribing and topic names. Most MQTT servers have some way of granting authority to publish and subscribe on topics.

In IBM Watson IoT Platform, this authorization is enforced by implementing secured messaging patterns. After the devices are authenticated, they are only authorized to publish and subscribe to a restricted topic space, for example:

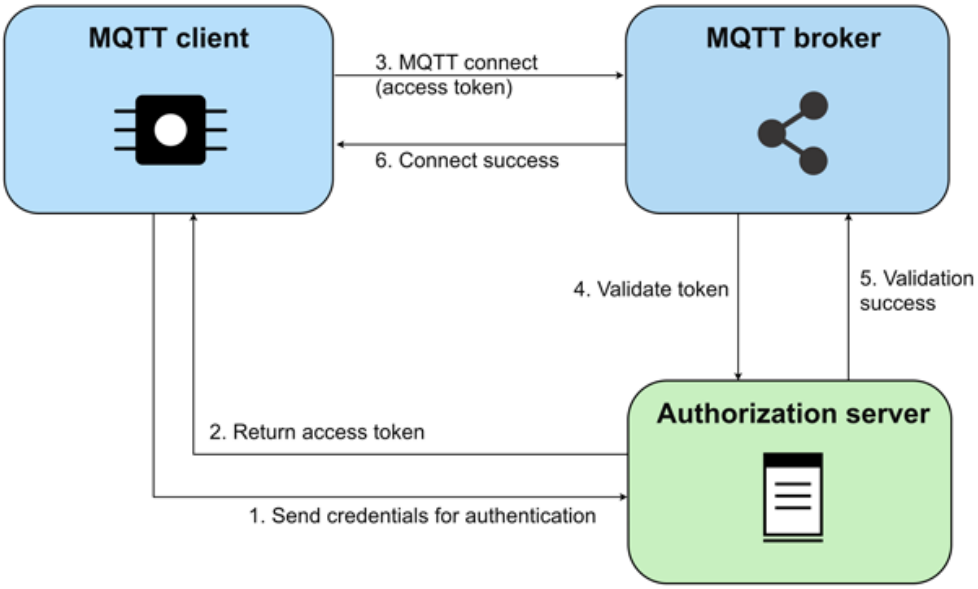
- `/iot-2/evt/+/fmt/+`
- `/iot-2/cmd/+`

All devices work with the same topic space. The authentication credentials that are provided by the connecting client dictate to which device this topic space will be scoped by IBM Watson IoT Platform. This configuration prevents devices from being able to impersonate another device. The only way to impersonate another device is by obtaining compromised security credentials for the device.

## Application authorization with OAuth 2.0

In cases where enterprises want to use their centralized authorization mechanism for MQTT devices, an OAuth-based framework can be used. OAuth 2.0 enables separation of the authorization server from a resource server, such as an MQTT server. When you use OAuth 2.0, the client presents its credentials to the authorization server, which then performs the authentication check and returns an access token that allows permission to access a resource.

The access token is then used to connect to the MQTT server. The MQTT server validates the access token, usually by communicating with the authorization server, then grants access to the resource. The flow is depicted in the following diagram:

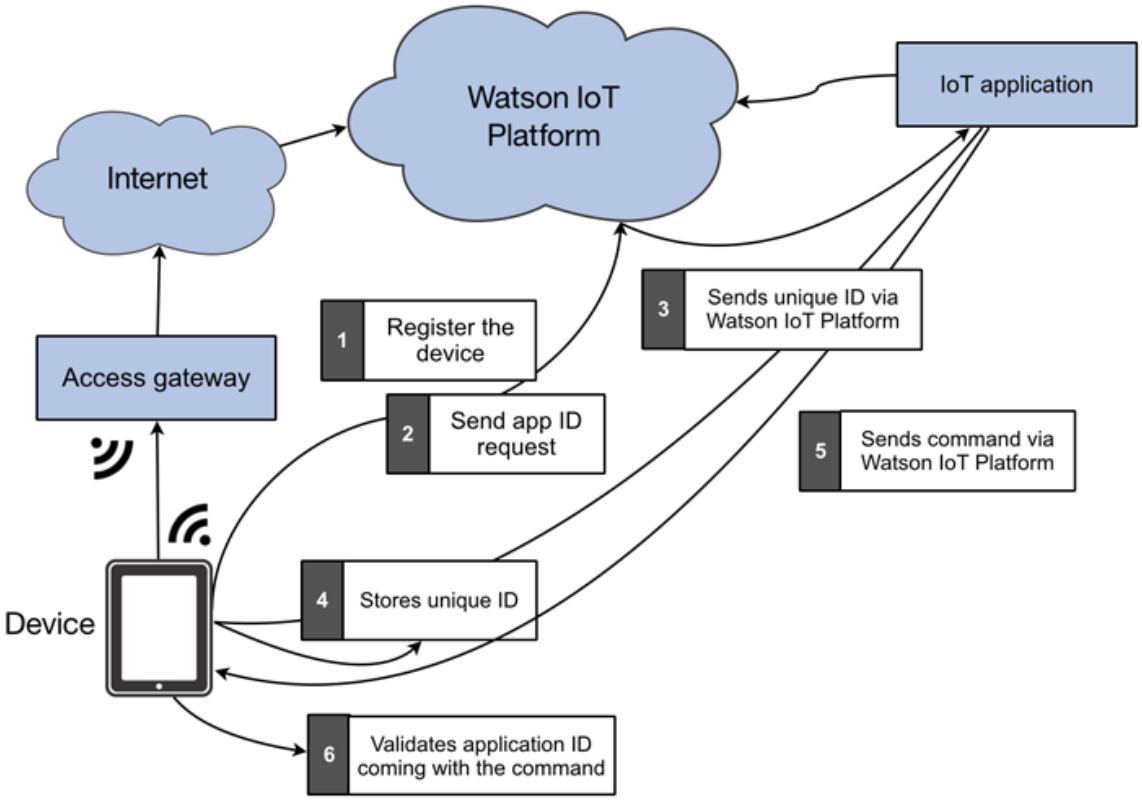



## Application ID validation

Application ID validation is an extra level of security between the IoT application and the device to ensure that no fake application can send commands to the device. This mechanism can be used both as startup security and as a communication security mechanism. By using this scheme, the device stores the unique ID of the IoT application and validates it when it processes the commands that are coming from the IoT application.

If the IoT application sends an invalid unique ID with a command, the command is ignored by the device. If the device has storage capability, the IoT application unique ID can be encrypted and stored. In that case, the unique ID request is not necessary after every restart.

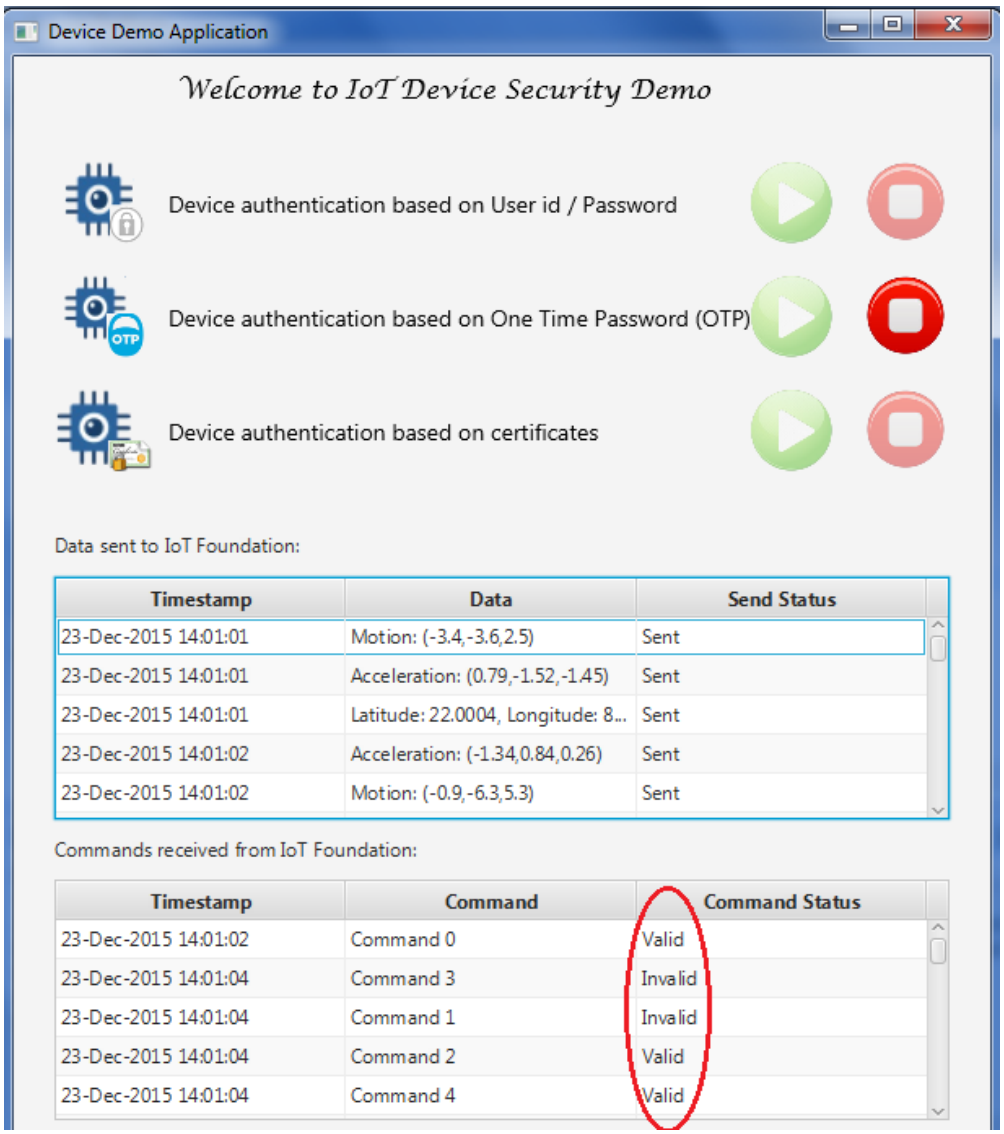
The detailed flow is shown in the following diagram:




The above diagram depicts the following aspects of the flow:

- The application ID validation can be turned on and off based on device capability.
- During restart, if application ID validation is enabled and unique application ID storage is enabled, the device tries to restore the IoT application unique ID from an encrypted file.
- If the device cannot load the unique application ID, it initiates a request to get the application.
- Upon receiving the request, the IoT application sends the unique ID to the device.
- The device stores the unique ID in memory and in a file (if it has storage capability).
- After that, the device expects the same application ID in each command coming from the IoT application.
- If there is a mismatch, the command is ignored by the device.

The following screen capture from the device simulator sample application shows the use of the application ID. In each instance, the application unique ID that comes with the command is validated against the stored application ID, and the command is executed or ignored accordingly.




## Protecting devices from other security issues

In addition to authentication, authorization, and communication issues that we've discussed so far, IoT devices need to be protected from various other forms of attack, by using these kinds of security measures:

- **Secured booting**.
  A secure boot is important for protecting the start-up code from attacks. Authenticity and integrity of the device must be verified during the initial booting process. A digital signature that is attached to the device software can be used to verify the authenticity of the device.
- **Device firmware upgrade**.
  The device must be protected from all known and unknown malicious attacks. A mechanism to install regular security patches must be developed to ensure all the devices get regular upgrades.
- **Incident logging and monitoring**.

All possible security incidents need to be logged and continuously monitored. Any suspicious activity in the device should immediately trigger de-registering the device from the broker.

## Secured transport

IBM Watson IoT Platform supports TLS 1.2, which can be used to secure the in-flight messages between the device and the broker. The risk of channel hacking and unauthorized access to the messages can be eliminated if the underlying transport layer is protected by using SSL. In IBM Watson IoT Platform, port **8883** is used for SSL-supported connections, as shown in the listing below.

## Listing 8. Encrypted client communication

```
//tcp://<org-id>.messaging.internetofthings.ibmcloud.com:1883
//ssl://<org-id>.messaging.internetofthings.ibmcloud.com:8883
 if (isSSL) {
  connectionUri = "ssl://" + serverHost + ":" + DEFAULT_SSL_PORT;
 } else {
  connectionUri = "tcp://" + serverHost + ":" + DEFAULT_TCP_PORT;
 }

 //If SSL is used, do not forget to use TLSv1.2
 if (isSSL) {
  java.util.Properties sslClientProps = new java.util.Properties();
  sslClientProps.setProperty("com.ibm.ssl.protocol", "TLSv1.2");
  options.setSSLProperties(sslClientProps);
 }
```

The IBM IoT starter library for Android (iot-starter-for-android GitHub repo) and iOS (iot-starter-for-ios GitHub repo) makes it easy to use MQTT and SSL in your application. You can use the provided `IoTClient` wrapper class as shown in the following listing:

## Listing 9. IoTClient wrapper class

```
IoTClient iotClient = IoTClient.getInstance(
        context,
        orgId,
        deviceId,
        deviceType,
        authToken
);

try {
    IMqttToken mqttToken = iotClient.connectDevice(
            new DemoIoTCallbackHandler(),
            new DemoIoTActionListenerHandler(),
            (usingSSL ? createSslSocketFactory(context) : null)
    );

    mqttToken.waitForCompletion(DemoIoTApplication.DEFAULT_MQTT_CONNECT_TIMEOUT);
}
catch(MqttException e) {
    Log.e(“Error connecting to IBM Watson IoT Platform: “+e);
}
```

The method `createSslSocketFactory` is defined as follows.

## Listing 10. The method `createSslSocketFactory`

```
private SocketFactory createSslSocketFactory(Context context) {
    SocketFactory factory = null;
```

```
    try {
        ProviderInstaller.installIfNeeded(context);

        KeyStore ks = KeyStore.getInstance("bks");
        ks.load(
                context.getResources().openRawResource(R.raw.iot),
                "password".toCharArray());

        TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
        tmf.init(ks);

        TrustManager[] tm = tmf.getTrustManagers();

        SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
        sslContext.init(null, tm, null);

        factory = sslContext.getSocketFactory();
    }
    catch (Exception e) {
        Log.e(TAG, "Error creating SSL factory: "+e);
    }

    return factory;
}
```

### Custom encrypted messages

Flexible options to impose multi-layered security and encryption can give users peace of mind and protect sensitive data from being compromised. Users sometimes prefer custom encryption over SSL protection, or they may be forced to use it when secured communication is not supported.

In this scheme, the application encrypts the message before it sends it to the MQTT broker. The cipher is exchanged between the device and the IoT application and is used to encrypt or decrypt the messages.

Details of custom encryption-based communication are described in Part 2 of this series.

## Conclusion

The correct approach, or approaches, for device authentication depends on the solution that you are designing and what kind of data the solution will be handling. Some key points to remember:

- Use secured connections if the IoT data is sensitive. Both authentication and secured (encrypted) communication are very important.
- Use OTP when an extra level of security is required, and when physical device security is not high, for instance, when anyone can access the physical device.
- Based on broker capabilities, use mechanisms like OAuth to support external authorization providers.
- Tightly control authorization to subscriptions, probably by using specific fixed-topic strings, which prevents a malicious client from making a massive number of subscriptions.
- Limit the size of messages to ensure that no one client or device can block the broker on its own.
- If certificate management is not a problem, consider issuing client certificates and only accepting connections from clients with certificates.

## Related topics

- Top 10 IoT security challenges
- Combating IoT cyber threats: Top security best practices for IoT applications
- IBM Watson IoT Platform Security Documentation
- MQTT Security Fundamentals on the HiveMQ blog
- Open Web Application Security Project: Internet of Things (IoT) Project