Practical IoT Cryptography on the Espressif ESP8266

Originalartikel

Backup

<html> The Espressif ESP8266 chipset makes three-dollar ‘Internet of Things’ development boards an economic reality. According to the popular automatic firmware-building site nodeMCU-builds, in the last 60 days there have been 13,341 custom firmware builds for that platform. Of those, only 19% have SSL support, and 10% include the cryptography module. We’re often critical of the lack of security in the IoT sector, and frequently cover botnets and other attacks, but will we hold our projects to the same standards we demand? Will we stop at identifying the problem, or can we be part of the solution? This article will focus on applying AES encryption and hash authorization functions to the MQTT protocol using the popular ESP8266 chip running NodeMCU firmware. Our purpose is not to provide a copy/paste panacea, but to go through the process step by step, identifying challenges and solutions along the way. The result is a system that’s end-to-end encrypted and authenticated, preventing eavesdropping along the way, and spoofing of valid data, without relying on SSL. We’re aware that there are also more powerful platforms that can easily support SSL (e.g. Raspberry Pi, Orange Pi, FriendlyARM), but let’s start with the cheapest hardware most of us have lying around, and a protocol suitable for many of our projects. AES is something you could implement on an AVR if you needed to. <h2>Theory</h2> MQTT is a lightweight messaging protocol that runs on top of TCP/IP and is frequently used for IoT projects. Client devices subscribe or publish to topics (e.g. sensors/temperature/kitchen), and these messages are relayed by an MQTT broker. More information on MQTT is available on their webpage or in our own getting-started series. The MQTT protocol doesn’t have any built-in security features beyond username/password authentication, so it’s common to encrypt and authenticate across a network with SSL. However, SSL can be rather demanding for the ESP8266 and when enabled, you’re left with much less memory for your application. As a lightweight alternative, you can encrypt only the data payload being sent, and use a session ID and hash function for authentication. A straightforward way to do this is using Lua and the NodeMCU Crypto module, which includes support for the AES algorithm in CBC mode as well as the HMAC hash function. Using AES encryption correctly requires three things to produce ciphertext: a message, a key, and an initialization vector (IV). Messages and keys are straightforward concepts, but the initialization vector is worth some discussion. When you encode a message in AES with a static key, it will always produce the same output. For example, the message “usernamepassword” encrypted with key “1234567890ABCDEF” might produce a result like “E40D86C04D723AFF”. If you run the encryption again with the same key and message, you will get the same result. This opens you to several common types of attack, especially pattern analysis and replay attacks. In a pattern analysis attack, you use the knowledge that a given piece of data will always produce the same ciphertext to guess

what the purpose or content of different messages are without actually knowing the secret key. For example, if the message “E40D86C04D723AFF” is sent prior to all other communications, one might quickly guess it is a login. In short, if the login system is simplistic, sending that packet (a replay attack) might be enough to identify yourself as an authorized user, and chaos ensues. IVs make pattern analysis more difficult. An IV is a piece of data sent along with the key that modifies the end ciphertext result. As the name suggests, it initializes the state of the encryption algorithm before the data enters. The IV needs to be different for each message sent so that repeated data encrypts into different ciphertext, and some ciphers (like AES-CBC) require it to be unpredictable – a practical way to accomplish this is just to randomize it each time. IVs do not have to be kept secret, but it’s typical to obfuscate them in some way. While this protects against pattern analysis, it doesn’t help with replay attacks. For example, retransmitting a given set of encrypted data will still duplicate the result. To prevent that, we need to authenticate the sender. We will use a public, pseudorandomly generated session ID for each message. This session ID can be generated by the receiving device by posting to an MQTT topic. Preventing these types of attacks is important in a couple of common use cases. Internet controlled stoves exist, and questionable utility aside, it would be nice if they didn’t use insecure commands. Secondly, if I’m datalogging from a hundred sensors, I don’t want anyone filling my database with garbage. <h2>Practical Encryption</h2> Implementing the above on the NodeMCU requires some effort. You will need firmware compiled to include the ‘crypto’ module in addition to any others you require for your application. SSL support is not required. First, let’s assume you’re connected to an MQTT broker with something like the following. You can implement this as a separate function from the cryptography to keep things clean. The client subscribes to a

```
sessionID
```

channel, which publishes suitably long, pseudorandom session IDs. You could encrypt them, but it’s not necessary. <pre class=„brush: plain; title: ; notranslate“ title=„“>
 m = mqtt.Client(„clientid“, 120)

 m:connect(„myserver.com“, 1883, 0,
 function(client)
 print(„connected“)
 client:subscribe(„mytopic/sessionID“, 0,
 function(client) print(„subscribe success“) end
)
 end,
 function(client, reason)
 print(„failed reason: “ .. reason)
 end
)

 m:on(„message“, function(client, topic, sessionID) end)
 </pre> Moving on, the node ID is a convenient way to help identify data sources. You can use any string you wish though:

```
nodeid = node.chipid()
```

. Then, we set up a static initialization vector and a key. This is only used to obfuscate the randomized initialization vector sent with each message, NOT used for any data. We also choose a separate key for the data. These keys are 16-bit hex, just replace them with yours. Finally we’ll need a passphrase for a hash function we’ll be using later. A string of reasonable length is fine. <pre class=„brush: plain; title: ; notranslate“ title=„“>
 staticiv = „abcdef2345678901“
 ivkey = „2345678901abcdef“
 datakey = „0123456789abcdef“
 passphrase = „mypassphrase“
 </pre> We’ll also assume you have some source of data. For this example it will be a value read from the ADC.

```
data = adc.read(0)
```

 Now, we generate a pseudorandom initialization vector. A 16-digit hex number is too large for the pseudorandom number function, so we generate it in two halves (16^8 minus 1) and concatenate them. <pre class=„brush: plain; title: ; notranslate“ title=„“>
 half1 = node.random(4294967295)
 half2 = node.random(4294967295)
 I = string.format(„%8x“, half1)
 V = string.format(„%8x“, half2)
 iv = I .. V
 </pre> We can now run the actual encryption. Here we are encrypting the current initialization vector, the node ID, and one piece of sensor data. <pre class=„brush: plain; title: ; notranslate“ title=„“>
 encrypted_iv = crypto.encrypt(„AES-CBC“, ivkey, iv, staticiv)
 encrypted_nodeid = crypto.encrypt(„AES-CBC“, datakey, nodeid,iv)
 encrypted_data = crypto.encrypt(„AES-CBC“, datakey, data,iv)
 </pre> Now we apply the hash function for authentication. First we combine the

```
nodeid
```

,

```
iv
```

,

```
data
```

, and session ID into a single message, then compute a HMAC SHA1 hash using the passphrase we defined earlier. We convert it to hex to make it a bit more human-readable for any debugging. <pre class=„brush: plain; title: ; notranslate“ title=„“>
 fullmessage = nodeid .. iv .. data .. sessionID
 hmac = crypto.toHex(crypto.hmac(„sha1“, fullmessage, passphrase))
 </pre> Now that both encryption and authentication checks are in place, we can place all this information in some structure and send it. Here, we’ll use comma separated values as it’s convenient: <pre class=„brush: plain; title: ; notranslate“ title=„“>
 payload = table.concat({encrypted_iv, eid, data1, hmac}, „,“)
 m:publish(„yourMQTTtopic“, payload, 2, 1, function(client) p = „Sent“ print(p) end)
 </pre> When we run the above code on an actual NodeMCU, we would get output something like this: <pre>1d54dd1af0f75a91a00d4dcd8f4ad28d,
 d1a0b14d187c5adfc948dfd77c2b2ee5,
 564633a4a053153bcbd6ed25370346d5,
 c66697df7e7d467112757c841bfb6bce051d6289
 </pre> All together, the encryption program is as follows (MQTT sections excluded for clarity): <pre class=„brush: plain; title: ; notranslate“ title=„“>
 nodeid = node.chipid()
 staticiv = „abcdef2345678901“
 ivkey = „2345678901abcdef“
 datakey = „0123456789abcdef“
 passphrase = „mypassphrase“

 data = adc.read(0)
 half1 = node.random(4294967295)
 half2 = node.random(4294967295)
 I = string.format(„%8x“, half1)
 V = string.format(„%8x“, half2)
 iv = I .. V

 encrypted_iv = crypto.encrypt(„AES-CBC“, ivkey, iv, staticiv)
 encrypted_nodeid = crypto.encrypt(„AES-CBC“, datakey, nodeid,iv)
 encrypted_data = crypto.encrypt(„AES-CBC“, datakey, data,iv)
 fullmessage = nodeid .. iv .. data .. sessionID
 hmac = crypto.toHex(crypto.hmac(„sha1“,fullmessage,passphrase))
 payload = table.concat({encrypted_iv, encrypted_nodeid, encrypted_data, hmac}, „,“)
 </pre> <h2>Decryption</h2> Now, your MQTT broker doesn’t know or care that the data is encrypted, it just passes it on. So, your other MQTT clients subscribed to the topic will need to know how to decrypt the data. On NodeMCU this is rather easy. Just split the received data into strings via the commas, and do something like the below. Note this end will have generated the session ID so already knows it. <pre class=„brush: plain; title: ; notranslate“ title=„“>
 staticiv = „abcdef2345678901“
 ivkey = „2345678901abcdef“
 datakey =

„0123456789abcdef“
 passphrase = „mypassphrase“

 iv = crypto.decrypt(„AES-CBC“, ivkey, encrypted_iv, staticiv)
 nodeid = crypto.decrypt(„AES-CBC“, datakey, encrypted_nodeid,iv)
 data = crypto.decrypt(„AES-CBC“,datakey, encrypted_data,iv)
 fullmessage = nodeid .. iv .. data .. sessionID
 hmac = crypto.toHex(crypto.hmac(„sha1“,fullmessage,passphrase))
 </pre> Then compare the received and computed HMAC, and regardless of the result, invalidate that session ID by generating a new one. <h2>Once More, In Python</h2> For a little variety, consider how we would handle decryption in Python, if we had an MQTT client on the same virtual machine as the broker that was analysing the data or storing it in a database. Lets assume you’ve received the data as a string “payload”, from something like the excellent Paho MQTT Client for Python. In this case it’s convenient to hex encode the encrypted data on the NodeMCU before transmitting. So on the NodeMCU we convert all encrypted data to hex, for example:

```
encrypted_iv = crypto.toHex(crypto.encrypt("AES-CBC", ivkey, iv, staticiv))
```

 Publishing a randomized sessionID is not discussed below, but is easy enough using os.urandom() and the Paho MQTT Client. The decryption is handled as follows: <pre class=„brush: plain; title: ; notranslate“ title=„“>
 from Crypto.Cipher import AES
 import binascii
 from Crypto.Hash import SHA, HMAC

 # define all keys
 ivkey = '2345678901abcdef'
 datakey = '0123456789abcdef'
 staticiv = 'abcdef2345678901'
 passphrase = 'mypassphrase'

 # Convert the received string to a list
 data = payload.split(„,“)

 # extract list items
 encrypted_iv = binascii.unhexlify(data[0])
 encrypted_nodeid = binascii.unhexlify(data[1])
 encrypted_data = binascii.unhexlify(data[2])
 received_hash = binascii.unhexlify(data[3])

 # decrypt the initialization vector
 iv_decryption_suite = AES.new(ivkey,AES.MODE_CBC, staticiv)
 iv = iv_decryption_suite.decrypt(encrypted_iv)

 # decrypt the data using the initialization vector
 id_decryption_suite = AES.new(datakey,AES.MODE_CBC, iv)
 nodeid = id_decryption_suite.decrypt(encrypted_nodeid)
 data_decryption_suite = AES.new(datakey,AES.MODE_CBC, iv)
 sensordata = data_decryption_suite.decrypt(encrypted_data)

 # compute hash function to compare to received_hash
 fullmessage = s.join([nodeid,iv,sensordata,sessionID])
 hmac = HMAC.new(passphrase,fullmessage,SHA)
 computed_hash = hmac.hexdigest()

 # see docs.python.org/2/library/hmac.html for how to compare hashes securely
 </pre> <h2>The End, The Beginning</h2> Now we have a system that sends encrypted, authenticated messages through an MQTT server to either another ESP8266 client or a larger system running Python. There are still important loose ends for you to tie up if you implement this yourself. The keys are all stored in the ESP8266s’ flash memory, so you will want to control access to these devices to prevent reverse engineering. The keys are also stored in the code on the computer receiving the data, here running Python. Further, you probably want each client to have a different key and passphrase. That’s a lot of secret material to keep safe and potentially update when necessary. Solving the key distribution problem is left as an exercise for the motivated reader. And on a closing note, one of the dreadful things about writing an article involving cryptography is the possibility of being wrong on the Internet. This is a fairly straightforward application of the tested-and-true AES-CBC mode with HMAC, so it should be pretty solid. Nonetheless, if you find any interesting shortcomings in the above, please let us know in the comments. </html>

From:
https://schnipsl.qgelm.de/ - **Qgelm**

Permanent link:
https://schnipsl.qgelm.de/doku.php?id=wallabag:practical-iot-cryptography-on-the-espressif-esp8266

Last update: **2021/12/06 15:24**