<https://maakbaas.com/esp8266-iot-framework/logs/https-requests/>

Обработка или публикация данных в Интернете - одна из основных задач устройства IoT. Выполнение этого через HTTP довольно хорошо реализовано в библиотеках Arduino ESP8266 по умолчанию, но с запросами HTTPS все сложнее. В этом посте я расскажу о наиболее распространенных подходах, используемых сообществом, и разработаю свой собственный метод безопасного выполнения произвольных HTTPS-запросов. Этот метод не требует ручного кодирования каких-либо сертификатов или отпечатков пальцев в приложении.

HTTPS, если метод выполнения HTTP-запроса через соединение TLS (ранее SSL). Таким образом, данные, которые передаются между вашим компьютером и сервером, зашифровываются и защищаются. Хорошей новостью является то, что этот протокол можно использовать с ESP8266 с классом WiFiClientSecure. Плохая новость в том, что у распространенных методов есть ряд серьезных недостатков.

Сначала я покажу два наиболее распространенных подхода, а затем опишу общее решение их проблем.

Несколько слов о сертификатах

Прежде чем углубляться в детали, я кратко объясню основные принципы безопасных HTTPS-запросов на языке непрофессионала.

Фактически у каждого сайта есть сертификат. Этот сертификат выдается кем-то, кого называют центром сертификации (ЦС). Каждый сертификат CA может быть выпущен другим CA, что приводит к так называемой цепочке сертификатов. На рисунке цепочка состоит из 3 сертификатов, но на самом деле длина цепочки может быть любой.

Цепочка сертификатов:

Рис. (В тексте по ссылке)

Сертификат верхнего уровня называется корневым сертификатом. Этот сертификат самозаверяющий, что означает, что ему можно доверять. Это связано с тем, что только несколько организаций могут выпускать корневые сертификаты, и они не могут предлагать поддельные или неправильные сертификаты.

Когда вы выполняете HTTPS-запрос к веб-сайту из своего браузера, браузер просматривает сертификат для веб-сайта и проверяет, действительно ли сертификат выпущен его родительским элементом. Это можно сделать, поскольку каждый сертификат подписан закрытым ключом вышестоящего сертификата. Разъяснение для чайников по работе с публичными и приватными ключами можно найти здесь

.

Когда подтверждается, что сертификат действительно выдан доверенным корневым центром сертификации, подтверждается, что домен в сертификате совпадает с фактическим доменом. Если это правда, мы знаем, что сервер является тем, кем он себя называет, и можно запустить безопасное соединение 👍.

Эти доверенные корневые сертификаты фактически хранятся как часть вашего браузера, чтобы иметь возможность проверять все другие сертификаты. Каждая ОС или браузер хранит немного другой набор примерно из 100-200 корневых сертификатов, которым можно доверять. Это называется хранилищем сертификатов, и это именно то, что я буду применять к ESP8266 позже в этой статье. Но сначала давайте начнем с двух самых популярных других подходов.

Отпечатки пальцев - безопасно, но раздражает

Метод, предложенный в официальной документации ESP8266 Arduino

состоит в том, чтобы извлечь отпечаток сертификата сайта и сохранить его в коде. Отпечаток пальца - это хэш сертификата. Поскольку очень маловероятно, что существует второй сертификат с тем же хешем, мы знаем, что веб-сайту можно доверять, если хэш совпадает с тем, который мы храним.

const char\* host = "https://api.github.com";

const char\* fingerpr = "CF 05 98 89 CA FF 8E D8 5E 5C E0 C2 E4 F7 E6 C3 C7 50 DD 5C";

WiFiClientSecure client;

client.connect(host, httpsPort);

if (client.verify(fingerpr, host))

{

http.begin(client, host);

String payload;

if (http.GET() == HTTP\_CODE\_OK)

payload = http.getString();

}

else

{

Serial.println("certificate doesn't match");

}

Этот подход прост, потому что цепочка сертификатов не нуждается в проверке, но для меня есть две основные проблемы:

Отпечаток пальца необходимо извлекать и сохранять вручную для каждой страницы, к которой должен подключаться ESP8266.

Еще более серьезная проблема заключается в том, что эти отпечатки пальцев меняются примерно раз в год, когда истекает срок действия сертификата. Это означает, что ваша программа может сломаться в любой момент, и вам нужно вручную обновить отпечаток пальца в вашем коде.

client.setInsecure () - легко, но небезопасно

Вы можете возразить, что безопасные соединения излишни для вашего приложения. Я первым признаю, что предпочитаю прагматичное решение там, где это возможно. Представьте, что все, что вы хотите сделать с помощью ESP8266, - это получить информацию о погоде из Интернета и отобразить ее в той или иной форме. Лично я бы не прочь сделать это небезопасным способом, поскольку здесь нет реальных опасностей.

Но представьте, что ESP8266 управляет замком вашей двери или 3D-принтером, который может нагреваться и загораться. Или подумайте о случае, когда вы переносите личную информацию на какой-либо сайт или API или с них. В этих случаях лучше перестраховаться, чем сожалеть, и метод, описанный в этом разделе, использовать нельзя. Тем не менее, я покажу это здесь:

const char\* host = "https://api.github.com";

WiFiClientSecure client;

client.setInsecure(); //the magic line, use with caution

client.connect(host, httpsPort);

http.begin(client, host);

String payload;

if (http.GET() == HTTP\_CODE\_OK)

payload = http.getString();

Поэтому в основном все, что вам нужно сделать, это добавить client.setInsecure () в свой код, и он запустит соединение без проверки сертификата.

**Запросы HTTPS с фреймворком IoT**

После этого мы, наконец, переходим к реализации, которую я выбрал вместо моей ESP8266 IoT Framework, которая помещена в класс fetch.

const char\* host = "https://api.github.com";

String payload;

if (fetch.GET(host) == HTTP\_CODE\_OK)

payload = http.getString();

fetch.clean();

Выглядит достаточно просто, правда 😎? Так что же происходит за кулисами?

Практически то же самое, что и в обычном браузере. ESP8266 содержит полное хранилище всех доверенных корневых сертификатов в памяти PROGMEM. На данный момент это занимает примерно 170 Кбайт флеш-памяти, что в моем случае можно легко упустить. Это хранилище сертификатов создается автоматически при сборке программного обеспечения, никаких действий вручную не требуется. Это также означает, что вы можете выполнять защищенные HTTPS-запросы к любому URL-адресу (так что вы даже можете настроить или изменить URL-адрес после сборки).

Вы можете подумать, но эй! Срок действия этих сертификатов тоже истечет. И это правда. Единственное отличие от отпечатков пальцев заключается в том, что срок действия корневых сертификатов намного дольше и может составлять более 20 лет. А для некоторых сервисов отпечаток пальца может меняться каждые несколько месяцев.

В качестве отправной точки я нашел отличный, но скрытый пример в репозитории ESP8266 Arduino.

. Этот пример содержит сценарий Python, который получает все сертификаты из корневого хранилища сертификатов Mozilla и сохраняет их в виде файлов. Затем эти файлы будут загружены в SPIFFS и использованы при запросах HTTPS. Я адаптировал этот пример, чтобы вместо этого иметь возможность хранить сертификаты в PROGMEM.

Создание хранилища сертификатов

Когда запрос запускается, класс certStore сравнивает хэш издателя сертификата со всеми хэшами сохраненных корневых сертификатов. Если есть совпадение, будет проверена правильность домена и других свойств, и соединение будет инициализировано.

В примере Arduino по умолчанию эти хэши для сохраненных сертификатов генерируются в классе certStore. Мне кажется более логичным сделать это прямо в скрипте Python, чтобы сэкономить время вычислений на ESP8266, поэтому я переместил его туда. Кроме того, я адаптировал класс certStore (GitHub

), чтобы читать информацию из моих переменных PROGMEM, а не из файловой системы.

Последний скрипт Python для создания хранилища сертификатов показан ниже.

bytestr = der.read();

sizes.append(len(bytestr))

cert = Certificate.load(bytestr)

idxHash = hashlib.sha256(cert.issuer.dump()).digest()

# for each certificate store the binary data as a byte array

f.write("const uint8\_t cert\_" + str(idx) + "[] PROGMEM = {")

for j in range(0, len(bytestr)):

totalbytes+=1

f.write(hex(bytestr[j]))

if j<len(bytestr)-1:

f.write(", ")

f.write("};\n")

# for each hashed certificate issuer, store the binary data as a byte array

f.write("const uint8\_t idx\_" + str(idx) + "[] PROGMEM = {")

for j in range(0, len(idxHash)):

totalbytes+=1

f.write(hex(idxHash[j]))

if j<len(idxHash)-1:

f.write(", ")

f.write("};\n\n")

der.close()

idx = idx + 1

f.write("//global variables for certificates using " + str(totalbytes) + " bytes\n")

f.write("const uint16\_t numberOfCertificates PROGMEM = " + str(idx) + ";\n\n")

# store a vector with the length in bytes for each certificate

f.write("const uint16\_t certSizes[] PROGMEM = {")

for i in range(0, idx):

f.write(str(sizes[i]))

if i<idx-1:

f.write(", ")

f.write("};\n\n")

# store a vector with pointers to all certificates

f.write("const uint8\_t\* const certificates[] PROGMEM = {")

for i in range(0, idx):

f.write("cert\_" + str(i))

os.unlink(derFiles[i])

if i<idx-1:

f.write(", ")

f.write("};\n\n")

# store a vector with pointers to all certificate issuer hashes

f.write("const uint8\_t\* const indices[] PROGMEM = {")

for i in range(0, idx):

f.write("idx\_" + str(i))

if i<idx-1:

f.write(", ")

f.write("};\n\n#endif" + "\n")

f.close()

Сгенерированный файл заголовка сохраняется как Certific.h и включается в приложение. Скрипт python подключен

в PlatformIO для автоматического выполнения перед каждой сборкой с автоматическим включением последней версии хранилища сертификатов.

Полный исходный код

Этот пост содержал лишь несколько фрагментов кода, объясняющих применяемый высокоуровневый подход. Полная реализация инфраструктуры Интернета вещей ESP8266 находится на GitHub.

. Документацию для класса fetch можно найти здесь.