TLS For Accessory Developers

This thread has been locked by a moderator.

313

I often get questions about disabling the HTTPS default server trust evaluation done by `NSURLSession`. My general advice is “Don’t do that!” However, there is one case where you *have* to do that, namely when dealing with a hardware accessory on the local network. This post contains my advice on how to deal with that situation.

IMPORTANT If you found your way here and you’re not developing a network-based accessory, I have two suggestions for you:

- If you’re trying to override HTTPS server trust evaluation for testing purposes, see QA1948 HTTPS and Test Servers.
- If not, post a question here on DevForums and we’ll try to get you heading in the right direction. Tag it with *Security*, and either *CFNetwork* or *Network* depending on which API you’re using.

Share and Enjoy
—
Quinn “The Eskimo!” @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

TLS For Accessory Developers

Communicating securely with a network-based accessory is tricky because the standard network security protocol, TLS, was designed for the wider Internet, where servers have a stable DNS name. A devices on your local network doesn’t have a stable DNS name. The DNS name `my-waffle-maker.local.` might resolve to your waffle maker today, but tomorrow it might resolve to a different waffle maker or, worse yet, a malicious waffle *varnisher*. I guarantee that you won’t like them waffles!

Note If you’re unfamiliar with TLS in general, I encourage you to read TLS for App Developers first.

TLS puts the *S* into *HTTPS*. This post focuses on `URLSession` and HTTPS, because that’s where this most commonly crops up, but similar logic applies to other networking APIs.

Doing the Minimum: The SSH Approach

Many accessory firmware developers support TLS by generating a self-signed digital identity on the accessory and using that for their HTTPS server. IMO they should do better — the subject of later sections in this post — but sometimes you have to work with what you’ve got.

If you’re working with an accessory like this, and you can’t convince the firmware developer to do better, I recommend that you adopt the **SSH approach**. In this approach you accept the certificate offered by the accessory the first time and then, when you connect again in the future, check the accessory still has the same certificate.

Note I call this the SSH approach because that’s how many folks use SSH.

To implement the SSH approach, first disable App Transport Security for local networking by setting the `NSAllowsLocalNetworking` property. On modern systems this isn’t strictly necessary but, as it says in the docs, this is a good “declaration of intent”.

Next, override HTTPS server trust evaluation as shown below:

```
var expectedCertificate: SecCertificate? = nil

func shouldAllowHTTPSConnection(trust: SecTrust) async -> Bool {
    guard
        let chain = SecTrustCopyCertificateChain(trust) as? [SecCertificate],
        let actual = chain.first
    else {
        return false
    }
    guard let expected = self.expectedCertificate else {
        // A. First connection
        self.expectedCertificate = actual
        return true
    }
    // B. Subsequent connections
    return CFEqual(expected, actual)
}

func urlSession(_ session: URLSession, didReceive challenge: URLAuthenticationChallenge) async ->
(URLSession.AuthChallengeDisposition, URLCredential?) {
    switch challenge.protectionSpace.authenticationMethod {
    case NSURLAuthenticationMethodServerTrust:
        let trust = challenge.protectionSpace.serverTrust!
        guard await self.shouldAllowHTTPSConnection(trust: trust) else {
            return (.cancelAuthenticationChallenge, nil)
        }
        let credential = URLCredential(trust: trust)
        return (.useCredential, credential)
    default:
        return (.performDefaultHandling, nil)
    }
}
```

There are a few things to note here:

- The snippet uses the new Swift async/await syntax. That simplifies the code and it also avoids a gotcha with the old completion-handler approach, where it was easy to forget to call the completion handler.

  Having said that, you can achieve the same result without using Swift async/await, or indeed, in Objective-C.
- The authentication challenge handler only deals with HTTPS server trust challenges (`NSURLAuthenticationMethodServerTrust`). Critically, if it finds a challenge it doesn’t recognise, it resolves that in the default way (`.performDefaultHandling`).
- The `shouldAllowHTTPSConnection(trust:)` method checks to see if it’s connected to the server before. If not, it defaults to allowing the connection (case A). If so, it checks that the certificate hasn’t changed (case B).
- In a real app you would persist `expectedCertificate` so that this logic continues to apply the next time your app is launched. When you do that, store the certificate using a key that uniquely identifies the accessory. **Do not use the DNS name or IP address** because that can change.
- The first time you connect (case A) you might take other steps to confirm the identity of the accessory. This is something I’ll cover below. The thing to note here is that this function is async, so you can take your time doing that confirmation. For example, you could bounce over to the main actor and display a UI.

Doing Better: Add a Side Channel

The above represents the *absolute minimum* you should do; anything less is doing the user a disservice. Aim to do better!

Doing better requires some sort of side channel so that the user can confirm the identity of the accessory. For example:

- If the accessory has a display, you might present a UI asking the user to confirm that the code in your UI matches the one on the accessory’s display.
- If the accessory has a LED, it might blink that during this initial setup process.
- If the accessory has a serial number, you might ask the user to confirm that. Or scan that via a barcode.

Note For an example of the barcode approach, see Matt’s Configuring a Wi-Fi Accessory to Join the User’s Network sample. This sample uses a different technique for TLS, namely TLS-PSK. This has some advantages — most notably, it avoids messing around with certificates — but I’m not discussing it here because it’s not supported by `URLSession`.

If you adopt this approach make sure that you present a UI that the user can understand. If your UI contains raw certificate details, most users will just say “Sure, whatever.” and click OK, at which point you might as well have saved everyone’s time by accepting the connection on their behalf.

Take care not to blindly trust any information you get from the accessory. After all, the goal here is to check the identity of the accessory, and so you mustn’t trust it before completing that check.

Sometimes this can be quite subtle. Consider the accessory-with-a-display example above. In that case the accessory’s self-signed certificate might contain a common name of `Acme Waffle Maker CCC`, where `CCC` is the code showing on its display. Don’t display the entire string to the user. The presence of `Acme Waffle Maker` might lead the user to believe that the accessory is valid, even when it isn’t. After all, a malicious waffle varnisher can just as easily create a self-signed certificate with that common name. Rather, extract the code (`CCC`) and display just that. That way the user will focus on what’s important.

Doing Even Better: Proper Security

If the accessory’s firmware and hardware developers are on board, there are steps you can take to properly secure your communication with the accessory:

- Create a custom certificate authority (CA) for your accessories.
- At the factory, have that CA issue a certificate to each accessory as its produced, embedding identifying details, like the serial number, in that certificate.
- Then install that certificate and its associated private key (thus forming a digital identity) on the accessory as it comes off the line.
- Embed the CA’s certificate in your app.
- When you connect the accessory, verify that its certificate was issued by your CA. If it was, you can trust the identifying information, like the serial number, embedded in that certificate.

With this setup your app will never connect to a malicious accessory. The worst that can happen is that you accidentally connect to the wrong instance of your accessory.

IMPORTANT This is just an outline of one approach that I believe offers specific advantages. If you plan to deploy this in practice, hire a security profession to design your process. They can advise you on the details of creating a system that’s secure in practice. For example, they can help you:

- Create a mechanism that prevents your factory from leaking valid digital identities.
- Design your hardware to prevent an attacker from extracting an accessory’s digital identity from the accessory itself.

To implement the `URLSession` side of this, change the `shouldAllowHTTPSConnection(trust:)` method like so:

```
let waffleVarnisherCA: SecCertificate = {
    let u = Bundle.main.url(forResource: "WaffleVarnisherCA", withExtension: "cer")!
    let d = try! Data(contentsOf: u)
    let c = SecCertificateCreateWithData(nil, d as NSData)!
    return c
}()

func shouldAllowHTTPSConnection(trust: SecTrust) async -> Bool {
    var err = SecTrustSetPolicies(trust, SecPolicyCreateBasicX509())
    guard err == errSecSuccess else { return false }
    err = SecTrustSetAnchorCertificates(trust, [self.waffleVarnisherCA] as NSArray)
    guard err == errSecSuccess else { return false }
    err = SecTrustSetAnchorCertificatesOnly(trust, true)
    guard err == errSecSuccess else { return false }
    let wasIssuedByOurCA = SecTrustEvaluateWithError(trust, nil)
    guard wasIssuedByOurCA else { return false }
    guard
        let chain = SecTrustCopyCertificateChain(trust) as? [SecCertificate],
        let trustedLeaf = chain.first
    else {
        return false
    }
    // C. Check the now-trusted leaf certificate
    return true
}
```

There are a few things to note about this code:

- It changes the trust policy (`SecTrustSetPolicies`) to the basic X.509 policy (`SecPolicyCreateBasicX509`) because the standard policy for TLS connections checks the DNS name, and that’s not appropriate for an accessory on the local network.
- It applies a custom anchor (`SecTrustSetAnchorCertificates`) and then disables all the other anchors (`SecTrustSetAnchorCertificatesOnly`) [1]. That prevents someone from impersonating your accessory by getting some other CA to issue a certificate that looks like the accessory’s certificate.
- It evaluates trust to confirm that the certificate was issued by the accessory’s CA.
- At point C you can trust the details in the accessory’s certificate (`trustedLeaf`). Here you might add code to confirm the identity of the accessory, to prevent you from accidentally connecting to the wrong accessory.

[1] This is not strictly necessary because calling `SecTrustSetAnchorCertificates` disables other anchors. However, I’m applying both belt and braces [2] here.

[2] I’m using the British idiom because *belt and suspenders* is so wrong (-:

Security CFNetwork Network

Reply

Posted 6 months ago by eskimo

Add a Comment