Bas de Cort

Mobile developer

Protecting your users with certificate pinning

JULY 18, 2018JULY 20, 2018 / BASDECORT

These days a lot of apps are in the news because they are hacked or data from the app has leaked. A breach in your security can not only cost a lot of money but also the trust of your users. In this article I will briefly cover secure network communication, but mostly focus on taking security to the next level with certificate pinning.

Building secure mobile apps is difficult because most of the times your app runs in a untrusted environment. Users are often not aware of security risks (https://www.csoonline.com/article/3241727/mobile-security/5-mobile-security-threats-you-should-take-seriously-in-2018.html) and therefor you want to protect them if possible. Public WiFi networks are a great example. If your user is on public WiFi and your app communicates over an un-safe connection (HTTP for example), it's child's play to intercept the network traffic. This traffic may contain passwords or other sensitive information that you don't want anyone else to see.

To prevent this, HTTP Secure (HTTPS (https://en.wikipedia.org/wiki/HTTPS)) was built. By using HTTPS, communication is encrypted with Transport Layer Security (TLS), formerly known as Secure Sockets Layer (SSL). When your app communicates over HTTPS, messages can still be intercepted, but will not be readable to the interceptor. Using HTTPS instead of HTTP should be a no brainer, some mobile operating systems (like iOS (https://developer.apple.com/library/content/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/SecureNetworking/SecureNetw don't even allow communicating over HTTP anymore.

Although HTTPS will improve your security vastly, it's still not bulletproof. By default your OS contains a set of trusted root certificates (e.g.: trusted on iOS 11 (https://support.apple.com/en-us/HT208125)). If for some reason a compromised certificate is installed on your device or hackers are able to get a valid certificate from a Certificate authority (https://en.wikipedia.org/wiki/Certificate_authority), communication over HTTPS is not so safe anymore. Malicious people will then be able to read or even manipulate your network traffic. This can be done with a Man In The Middle Attack (MITM) (https://en.wikipedia.org/wiki/Man-in-the-middle_attack), ARP spoofing (https://en.wikipedia.org/wiki/ARP_spoofing) or DNS spoofing (https://en.wikipedia.org/wiki/DNS_spoofing). Certificate pinning can help you prevent these attacks by verifying that the server is responding with the expected certificate.

Certificate pinning

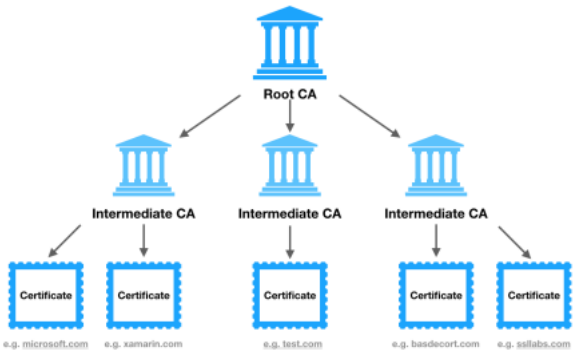
Certificate pinning can be used to verify the integrity of the system you are communicating with. The certificate can be verified in a few different ways:

- Certificate pinning: This is the easiest way of pinning. At runtime you will compare the server certificate with an embedded certificate, when it doesn't match the request will fail. A downside of this method, is when the certificate changes, you also need to update your app.
- Public key pinning: This way of pinning is a bit more trickier because you might need to take some extra steps (depending on the platform) to extract the public key from your server certificate. When the public key is extracted it will be compared with an embedded key in your app and when it doesn't match the request will fail. Because the public key is static and won't change when the certificate is renewed (if requested with same certificate request), you don't need to update your app on certificate renewal. Although this is convenient, some companies might have policies on key rotation and therefor app updates might still be required once in a while.
- Subject Public Key Info (SPKI) pinning: will verify the fingerprint of the certificate to match a hash of the SPKI (https://tools.ietf.org/html/rfc7469#section-2.4). The SPKI consist of the algorithm of the public key, along with the public key. This way of pinning is similar to public key pinning, but uses a different payload. Just like public key pinning, your app doesn't require an update when a certificate is renewed with the same request.

Besides different ways of pinning, you also have to choose the level of verification. To choose a level, it's important to have an understanding on how certificates are signed. Every Operating System has a list of companies that are allowed to issue certificates. The companies in this list are called Certificate Authorities and the list itself is often referred to as Trusted Root Certification Authorities Certificate Store. If you visit a website over HTTPS, your OS or browser will verify if the certificate is signed by a CA that is listed in your Trusted Root CA Store. If the CA is in your store, the public key of the certificate will be used to encrypt your communication. If it's not in the store, the website will be marked as "Not Secure". With this in mind, it's clear that you don't want to install a malicious certificate in your local Trusted Root CA store as this will allow someone to intercept or manipulate your data. With this in mind, choosing a level of verification will also impact the level of security and the number of required updates. The levels you need to choose from are:

- Leaf: This is the most secure verification because it will only allow the actual certificate of your server. Other certificates issued by the same (valid) CA will not pass verification. If for some reason your private key get's compromised, your app will be bricked until you've updated the embedded certificate.
- Intermediate: Intermediate verification will verify that the certificate is issued by a specific CA. This will often be the company where you bought your certificate. With this verification you are still relying on the CA to only issue certificates to trustworthy companies. This level of pinning is commonly used because it is secure enough in most cases and allows you to renew certificates without updating the app. If your CA is compromised

you still need to update your app, but this is not very likely to happen (except for Diginotar (https://www.eff.org/deeplinks/2011/09/post-mortem-iranian-diginotar-attack)).
- Root: This is the least secure verification because when you trust the root, you also trust all it's child CA's. This approach is not very often used in app development.



Implementing in Xamarin

There are different ways to implement certificate pinning in Xamarin. You can choose for the native approach which will give you fine grained control or you can choose a cross-platform way which is easier to implement, but also has limitations.

Your choice really depends on what HttpClient implementation you are using. If you're using the Managed HttpClient, you will be able to implement certificate pinning with the ServicePointManager class. This approach is by far the easiest way to implement certificate pinning, but unfortunately doesn't work with the native HttpClient handlers (like NSUrlSession/CfNetwork or AndroidClientHandler). If you want to use the native handlers, you also have to implement certificate pinning in a native fashion.

Option 1: Cross-platform (ServicePointManager)

The ServicePointManager class exposes a ServerCertificateValidationCallback (https://msdn.microsoft.com/nl-nl/library/system.net.servicepointmanager.servercertificatevalidationcallback(v=vs.110).aspx) that will be called every time you make a network request:

```
ServicePointManager.ServerCertificateValidationCallback += (sender, certificate, chain, sslPolicyErrors) =>
{
    return _allowedPublicKeys.Contains(certificate?.GetPublicKeyString());
};
```

In this code sample we are checking if the public key of the server is in the list we've embedded in our app. If not, the call wil fail, otherwise it will proceed as it would without pinning. A code sample on intermediate pinning can de found on GitHub (https://github.com/basdecort/Xamarin/blob/master/CertificatePinning/CertificatePinning/Services/CertficateValidation.cs).

As previously mentioned, this approach doesn't work is you're using the native HttpClient handlers, but there is a workaround with the NuGet package ModernHttpClient (https://github.com/paulcbetts/ModernHttpClient). If you install this NuGet package and pass the provided handler to your HttpClient, the ServicePointManager will be triggered while using the native network stack. Sadly, since Xamarin added support for native handlers, this NuGet package isn't maintained anymore. More info can be found on GitHub. (https://github.com/paulcbetts/ModernHttpClient)

Option 2: Native (Xamarin.Android) – AndroidClientHandler

On the Android side of things there are a few different ways of implementing certificate pinning. The preferred way is to use Network Security Configuration (https://developer.android.com/training/articles/security-config) (NSC). NSC allows you to configure certificate pinning in XML format pretty easily. Unfortunately this requires Android 7.0 (API 24) at a minimum.

Most of the time you also want to support lower versions of Android and therefor NSC is not the best way for now. A commonly used alternative on Android is the OkHttp client (http://square.github.io/okhttp/), which has some methods (https://square.github.io/okhttp/3.x/okhttp/okhttp3/CertificatePinner.html) to easily verify certificates. There is a binding available on NuGet (https://www.nuget.org/packages/Square.OkHttp3/) that can be used in Xamarin.Android, but the documentation is pretty limited. Also, this cannot be used combined with the HttpClient class and therefor you cannot use this in shared code. This is approach might work for native Android, but is not great for Xamarin.Android.

An alternative (and in most cases the best) way to implement certificate pinning is by building your own TrustManager class (https://developer.android.com/reference/kotlin/javax/net/ssl/TrustManager). This class is responsible for the validation of external parties you're communicating with. There are a few ways to use your own TrustManager, but I found the easiest way to set your handler on the current TLS context:

```
private void SetHandler()
{
   var algoritm = TrustManagerFactory.DefaultAlgorithm;

   var trustManagerFactory = TrustManagerFactory.GetInstance(algoritm);
   trustManagerFactory.Init((KeyStore)null);

   var tm = new ITrustManager[] { new PublicKeyManager() };

   var sslContext = SSLContext.GetInstance("TLS");
   sslContext.Init(null, tm, null);
   SSLContext.Default = sslContext;
   HttpsURLConnection.DefaultSSLSocketFactory = sslContext.SocketFactory;
}
```

For more info, check the full sample code on GitHub (https://github.com/basdecort/Xamarin/blob/master/CertificatePinning/CertificatePinning.Android/Handlers/PublicKeyHandler.cs).

Option 2: Native (Xamarin.iOS) – NSUrlSession

The most common way to implement certificate pinning in Swift / Objective-C is to use the TrustKit (https://github.com/datatheorem/TrustKit). Unfortunately, at this moment, there is no Xamarin binding available for TrustKit.

A different approach is to override the NSUrlSessionHandlerDelegate:DidReceiveChallenge. This method allows you to validate certificates and kill the request if it doesn't match. Unfortunately the NsUrlSessionHandler (https://github.com/xamarin/xamarin-macios/blob/master/src/Foundation/NSUrlSessionHandler.cs) from Xamarin uses a private NSUrlSessionHandlerDelegate which makes hard to override the DidReceiveChallenge. It can still be done, but you'll have to copy the NsUrlSessionHandler code into your project, which is not ideal. Recently CheeseBaron created an issue on GitHub (https://github.com/xamarin/xamarin-macios/issues/4170) to add support for implementing your own delegate, so this might get fixed in the near future. If you decide to copy the Xamarin classes (until the issue is solved), I recommend looking at Jonathan's example (https://github.com/jonathanpeppers/Xamarin.SSLPinning).

Verification can be done in the DidReceiveChallenge method. Before the end of the method, you need to call the completionHandler to perform or cancel the request:

```
if (IsValid(serverCertChain))
{
   // Proceed with the request
   completionHandler(NSUrlSessionAuthChallengeDisposition.PerformDefaultHandling, challenge.ProposedCredential);
} else {
   // Cancel the request
   completionHandler(NSUrlSessionAuthChallengeDisposition.CancelAuthenticationChallenge, null);
}
```

The code above will make sure only valid calls may proceed, but the actual validation is done in the IsValid method. This method takes in a parameter of type NSUrlAuthenticationChallenge (https://developer.xamarin.com/api/type/MonoTouch.Foundation.NSUrlAuthenticationChallenge/) which can be used to get validate the certificate tree. This implementation of IsValid will verify the public key:

```
private static bool IsValid(NSUrlAuthenticationChallenge challenge)
{
   var serverCertChain = challenge.ProtectionSpace.ServerSecTrust;
   var first = serverCertChain[0].DerData;
   var firstString = first.GetBase64EncodedString(NSDataBase64EncodingOptions.None);
   var cert = NSData.FromFile("xamarin.cer");
   var certString = cert.GetBase64EncodedString(NSDataBase64EncodingOptions.None);
   return firstString == certString;
}
```

Views

Most of your network calls are made through the HttpClient, but there are a few exceptions. Some (Xamarin Forms) Views like Image or WebView also make requests to a server. For most Views you can create a custom class and load the content of the View with your HttpClient that uses certificate pinning. Views that contain Web content are a bit more complex, because you might also want to verify al links that are loaded inside the (Web)View. To implement this, you'll need to create custom renderers.

Views approach:

```
public class SafeImage : Image
{
   private readonly SafeService _safeService;
   public SafeImage(SafeService safeService)
   {
      _safeService = safeService;
   }

   public async Task Load(string url)
   {
      var stream = await _safeService.GetStream(url);
      if (stream != null)
      {
         Source = ImageSource.FromStream(() => stream);
      }
   }
}
```

The sample above will make sure that the resource is verified before loading it in the ImageView. As mentioned, to verify WebViews, you'll need to create custom renderers:

Android custom renderer

To verify all the calls (also inside the WebView), you need create a custom WebViewClient and add this to your WebView Control. The WebViewClient contains a method for you to override called ShouldInterceptRequest (https://developer.xamarin.com/api/member/Android.Webkit.WebViewClient.ShouldInterceptRequest/p/Android.Webkit.WebView/System.String/). Inside this method you'll want to load the content with your own HttpClient and then set the loaded content in the Source property of your WebView. A code sample can be found on GitHub (https://github.com/basdecort/Xamarin/blob/master/CertificatePinning/CertificatePinning.Android/Renderers/CustomWebViewRenderer.cs).

iOS custom renderer

On iOS a custom implementation of the UIWebViewDelegate (https://developer.xamarin.com/api/type/MonoTouch.UIKit.UIWebViewDelegate/) can be used to verify all calls from the WebView. This delegate contains a method called ShouldStartLoad (https://developer.xamarin.com/api/member/MonoTouch.UIKit.UIWebViewDelegate.ShouldStartLoad/) that can be overridden for custom validation. If this method returns false, the request will be cancelled. When it returns true, it will continue proceed with the request. A code sample can be found on GitHub (https://github.com/basdecort/Xamarin/blob/master/CertificatePinning/CertificatePinning.iOS/Renderers/CustomWebViewRenderer.cs).

Get the certificate (public key)

There are a lot of tools to get retrieve a public key from a website or certificate. I've found C# to be the easiest way:

```
var cert = X509Certificate.CreateFromCertFile("{filepath}.cer");
var publicKey = cert.GetPublicKeyString();
```

If you don't have the .cer file, you can use Google Chrome to download it from your API / website:

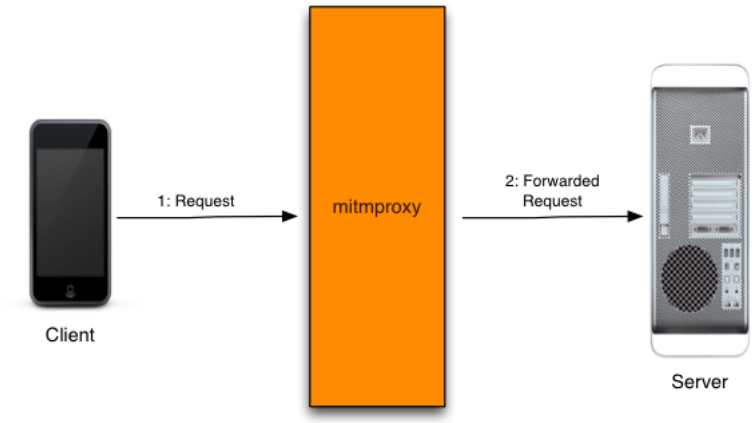
1. Click the Secure button in your address bar.
2. Then click "Certificate".
3. This will now show the applicable certificates for this site. Select your certificate and drag the certificate icon to your file explorer. This will download the certificate.

How to verify

A very simple verification can be done during development by altering your embedded key, so it becomes invalid. After altering your key, requests should be cancelled as expected.

To really put your pinning to the test in a more practical situation, you can use a proxy tool like Fiddler (https://www.telerik.com/fiddler), Charles (https://www.charlesproxy.com/) or Mitmproxy (https://mitmproxy.org/). I've found Mitmproxy to be the easiest solution. There is a great blogpost on how to configure Mitmproxy (https://medium.com/sean3z/debugging-mobile-apps-with-mitmproxy-4596e56b3da2).

For a few dollars, you can also buy the Charles iOS app (https://www.charlesproxy.com/documentation/ios/). This app will work as proxy without having to install software on your computer.



Despite what tool you are using, if certificate pinning is implemented correctly, requests will fail when the proxy is intercepting your requests.

In addition to the above, Kerry W. Lothrop (https://twitter.com/kwlothrop) created a great serie of blogposts (https://kerry.lothrop.de/wedskaas1/) on App Security and recorded a Xamarin Show (https://youtu.be/blTFwasG21U) with James Montemagno. If you want to know more, see his resources and the resources below. Happy pinning!

Related links

- Android Security – SSL pinning (https://medium.com/@appmattus/android-security-ssl-pinning-1db8acb6621e)
- Swift Talk – #57 Certificate Pinning (https://talk.objc.io/episodes/S01E57-certificate-pinning)
- Robert Heaton – How does HTTPS actually work? (https://robertheaton.com/2014/03/27/how-does-https-actually-work/)
- Asymmetric encryption – Simply explained (https://www.youtube.com/watch?v=AQDCe585Lnc)
- Google Support – Secure your site with HTTPS (https://support.google.com/webmasters/answer/6073543?hl=en)
- The Chicken Coop – Increasing your trust: Certificate Pinning on iOS (https://fastchicken.co.nz/2016/03/21/increasing-your-trust-certificate-pinning-on-ios/)
- Tim Taubert – Public Key pinning explained (https://timtaubert.de/blog/2014/10/http-public-key-pinning-explained/)
- Coursera – Public key infrastructure (https://www.coursera.org/learn/it-security/lecture/tjlhH/public-key-infrastructure)
- Tim Klingeleers – Certificate pinning in Xamarin (https://tim.klingeleers.be/2017/04/21/security-xamarin-certificate-pinning/)
- OWASP – Pinning Cheat Sheet (https://www.owasp.org/index.php/Pinning_Cheat_Sheet)
- Thomas Bandt – Certificate and public key pinning with Xamarin (https://thomasbandt.com/certificate-and-public-key-pinning-with-xamarin)

Categories: Android, iOS, Mobile, Xamarin Tags: Certificate pinning, HTTPS, SSL, TLS

One thought on “Protecting your users with certificate pinning”

1. Pingback: Dew Drop - July 18, 2018 (#2768) - Morning Dew