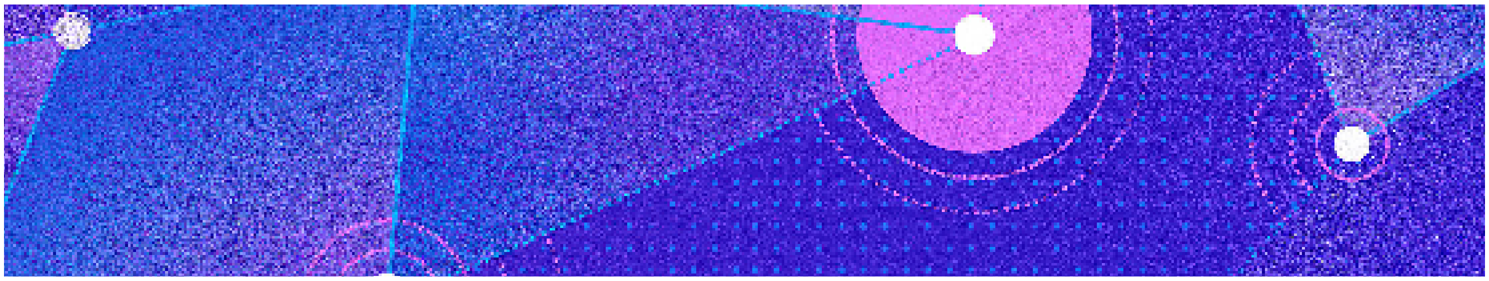


How to Prepare Your Android App for a Pentest — Networking Edition

Renato Turić

14 min read

Making your app as secure as possible is a must when developing an application, especially if you deal with sensitive user information. To identify the weak spots of your application's security, it is good practice to have it tested by mobile security experts. The testing method they use for this is called penetration testing.

Penetration tests, aka *pentests*, are simulated cyber attacks on your application designed to find exploitable vulnerabilities. The results of a penetration test are then used to improve the app's security.

The comprehensive guide to pentesting

In this blog post, we will concentrate on networking – specifically on the TLS protocol, and tips on making your app as secure as possible when connecting to a specific web service.

when having your app pentested, as well as provide recommendations on the neccessary preparations ahead of it.

Disclaimer: In our examples, we will use OkHttp for the integration part, because it is a well established library and probably the most popular one in the Android community.

TLS connection

Most applications today communicate with a web service of some kind. Therefore, it's very important to have a secure connection.

The worst case scenario is having an app which uses HTTP connection without the Transport Layer Security (TLS) protocol. It is inevitably going to fail because your data will be transferred in plain text – and therefore, very easy to track. TLS is a core part of encrypted communication, which makes HTTPS calls secure and authenticated.

Even though it is based on SSL 3.0, you will often hear people using SSL as a synonym for TLS- That's not 100% correct as SSL, a predecessor for TLS, was deprecated in 2015 by the IETF.

In fact, most of the big companies are switching to TLS 1.2 as the Internet's new minimum standard, starting from early 2020. (Mozzila, Google, Microsoft, Apple).

What does Android OS offer, with regard to TLS?

According to the documentation, Android supports TLS 1.2 since API level 16, and is enabled by default since level 21. Unfortunately, this is not 100% correct either.

Yes, you can rely on TLS 1.2 from API level 21/22 and above. However, you cannot count on that for API levels 16 to 19. There is an excellent article written by Ankush Gupta that describes this problem in more detail, so feel free to check it out if you are want to know more about the how and why.

INFINUM

With the knowledge of what Android OS has to offer and the goal to achieve a secure connection using at least TLS 1.2, we're moving on to the implementation guide.

Preparation phase – What you'll need to know

In this section, we will cover two specific cases. The first case is when your minSdk is between API levels 16 and 19. In the other case, your minSdk is API level 21 or higher.

MinSdk between API levels 16 and 19

If you want TLS 1.2 enabled on versions before Android 5, you will have to extend the `SSLSocketFactory` and create your own implementation. The implementation is quite simple and straightforward, and would look something like this:

```
class TlsSocketFactory constructor(private val socketF

    override fun getDefaultCipherSuites(): Array<Strin
        return socketFactory.defaultCipherSuites
    }

    override fun getSupportedCipherSuites(): Array<Str
        return socketFactory.supportedCipherSuites
    }

    override fun createSocket(socket: Socket?, host: S
        return socketFactory.createSocket(socket, host
    }

    override fun createSocket(host: String?, port: Int
        return socketFactory.createSocket(host, port).
    }

    override fun createSocket(host: String?, port: Int
        return socketFactory.createSocket(host, port,
    override fun createSocket(host: InetAddress?, port
        return socketFactory.createSocket(host, port).
    }

    override fun createSocket(address: InetAddress?, p
        return socketFactory.createSocket(address, por
    }

    private fun Socket.enableTls(): Socket {
        if (this is SSLSocket) enabledProtocols += Tls
        return this
    }
}
```

This implementation contains our own `socketFactory` object, which we use as a delegate and simply update the enabled protocols.

Now that we have a custom `SSLSocketFactory` implementation, we have to tell OkHttp to use it via the `sslSocketFactory` builder parameter:

```
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.LOLLI
        val sslContext = SSLContext.getInstance(TlsVersion
        sslContext.init(null, *yourTrustManagers*, null)

        sslSocketFactory(TlsSocketFactory(sslContext.socke
    }
```

INFINUM

need custom implementation, you can always load the default one. You will learn how to load the default trust manager in the next section, in which we will discuss certificates.

Note that the above stated `TlsSocketFactory` implementation only sets the specified protocol as the default. It does not cover the case in which the protocol is not installed on the device. To cover that instance, you will have to use the `ProviderInstaller` from Google Play services.

Also, keep in mind that you can either call `installIfNeeded(context)` or `installIfNeededAsync(context)`, and that it has to be done prior to creating the TlsSocketFactory.

For a more specific implementation, you can check the sample implementation on these gists: installIfNeeded, installIfNeededAsync. These implementations are provided by the OWASP team and can be found in the OWASP-MSTG book 1.1.3 on pages 206, 207 and 208.

MinSdk API level 21+

If this is the case, you should use at least OkHttp 3.13.x – and it works only on Android 5+. If you're already using that version of Android, you're good to go, as this version of the library already uses TLS 1.2 as a default for all HTTPS calls.

In case you are using an older version of the OkHttp library and cannot update it for whatever reason, you will have to follow the steps described in the section *MinSdk between API levels 16 and 19* above.

Btw, if you are interested in the history of TLS configuration in OkHttp, this is a superb read.

Coming back to the topic – now that your app uses a more secure version of the TLS protocol, it's important to know how to successfully verify a TLS connection.

Verifying a TLS connection

- Certificate verification (Certificate pinning)
- Hostname verification

In the following paragraphs, we'll go over the specifics of each one.

Certificate verification (Certificate pinning)

The main focus of this section is going to be **certificate pinning** and how to implement it. In addition, we'll take a brief look on how certificate verification is done.

In order for a TLS connection to work as expected, the Client must have a way of verifying that a certificate used on the server is trusted and valid. In our case, the Client is the Android app. This is where **Certificate Authorities (CA)** enter the stage.

A CA is an entity that is eligible to issue a trusted, time-limited certificate. The certificate issued by a CA is a verifiable small data file which contains identity credentials to help websites, people, and devices represent their authentic online identity.

But how does our Android device differentiate between the certificates issued by a CA and those so-called **self-signed** certificates? With the help of a set of CAs, which the device has stored on the Android system level. As of 4.2, Android contains over 100 CAs, which are updated in each release.

To view all certificates in an Android device programmatically, you can load the default `TrustManager`. That way, you will see all the file paths that correspond to system-level certificates, but also the user-installed certificates. These certificates are also known as **root certificates**. To retrieve the default `TrustManager` you can use this code:

```
private fun getDefaultTrustManager(): Array<TrustManag
    val trustManagerFactory = TrustManagerFactory.getI
    trustManagerFactory.init(null as KeyStore?)
```

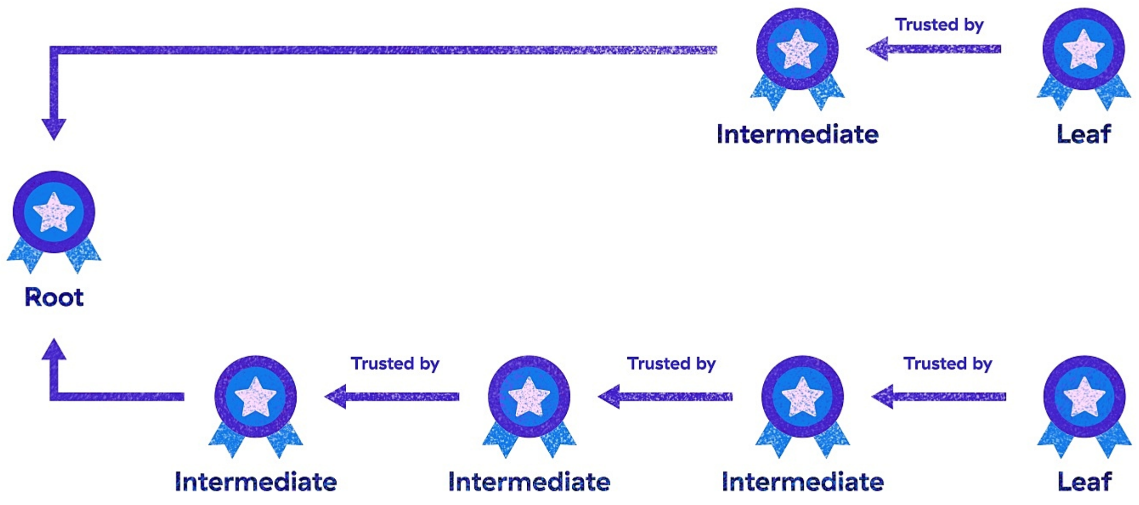
```
        if (trustManagers.size != 1 || trustManagers[0] !i
            throw IllegalStateException("Unexpected defaul
        }
        return trustManagers
    }
```

When an Android device tries to establish a secure connection with a service, it goes through a process called the *TLS handshake*. During connection setup, the server can send an entire chain of certificates which the device has to verify with its set of trusted CAs.

The certificate chain is also known as *the chain of trust,* which is basically a linked path of verification and validation from the end user (in our case the Android device) to a root certificate.

The certificates in the chain are consisted of a **root certificate**, zero or more **intermediate certificates** and a **leaf certificate**.

Check out the image below to clarify this confusion a bit:



Certificates in the chain are co-dependent

As you can see, the certificates in the chain are dependent on each other. This means that the chain of trust can be exploited even if just one of the issue a certificate that will be automatically trusted by your Android device, due to the default `TrustManager`.

To further protect your app from fraudulently issued certificates, you can use a technique known as *certificate pinning*. Certificate pinning is an extra defence mechanism against MITM (*ie. man-in-the-middle*) attacks, in which the developer implements a custom trust manager that contains only the certificates which can validate that web server that the app is communicating with. This means that your app will neither trust any other system-trusted CA nor the user-installed certificates.

Certificate pinning is one of the most common test cases in a penetration test, so the following paragraph will elaborate on how to prepare an app for it.

Preparation phase – What you'll need to know

In this section, we are going to show you how to successfully implement certificate pinning for both self-signed certificates and those issued by a CA. To pin a certificate, the first step is to decide which certificate from the certificate chain to pin.

It is often recommended to pin the **leaf certificate** as the attack surface is very small, thanks to the fact that the app interacts with that certificate first. If that certificate can't be verified, the app will not be usable.

Self-signed certificates

Pinning self-signed certificates is something you should only do for testing environments. To pin a certificate in an Android app, the first thing to do is to acquire the certificate, which can easily be done with OpenSSL va the terminal:

```
$ openssl s_client -connect example.com:443 -showcerts
```

The command above will return a list of the entire certificate chain for the specified website. If you want to save the certificates in a local file or download one via a browser, find out how to do that here.

Having acquired the needed certificate, the next step is to create your custom `TrustManager` which will contain the acquired certificate. This can be done as follows:

```
private fun createCustomTrustManager(context: Context)
    val keyStore = KeyStore.getInstance("BKS").apply {
    val certificateFactory = CertificateFactory.getIns
    val trustManagerFactory = TrustManagerFactory.getI

    context.resources.openRawResource(R.raw.leaf_cert_
        keyStore.setCertificateEntry("MyLeafCert", cer
    }

    trustManagerFactory.init(keyStore)
    return trustManagerFactory.trustManagers
}
```

This code loads a `leaf_cert_expires_1_1_2077.pem` file from the `raw` resource folder and sets it as a certificate entry inside our empty keystore, under *MyLeafCert* alias. After setting the required certificate in the keystore, use the same keystore to initialise the trustManagers.

With everything ready, it's time to pass the instance of our `TrustManager` to OkHttp via the `sslSocketFactory` builder function,

INFINUM

certificate in our app, and we have to upload a new version of the app that contains the new certificate every time the certificate changes or gets renewed.

Certificates signed by a trusted CA

The previous code sample can be used in the same way for certificates signed by a trusted CA. This is a good approach if you want just one implementation for all cases. However, OkHttp does provide a different mechanism for pinning non self-signed certificates, using the `CertificatePinner` class. More information about this class can be found here.

`CertificatePinner` employs a different kind of pinning, using the certificate's subject cryptographic public key for verification. Unlike the previously described approach, in which you need to bundle the pinned certificate with your app, public key pinning only needs the base64 SHA-256 hash value of the certificate public – considering that the hash can be safely stored as a plain string in your app without causing security issues.

The implementation is much more straightforward:

```
const val MY_LEAF_CERT_EXPIRES_1_1_2077 = "sha256/A23d

private fun buildCertificatePinner(): CertificatePinne
    return CertificatePinner.Builder()
        .add("yourHostname.com", MY_LEAF_CERT_EXPIRES_
        .build()
}

// Setting up OkHttp with the CertificatePinner object
OkHttpClient.Builder()
    .certificatePinner(buildCertificatePinner())
    .build()
```

So there you have it - a very simple and intuitive approach. In addition to easier implementation, this approach has another noticeable benefit. If the certificate expires, the server can just renew the certificate, without changing the cryptographic public key of the certificate.

This essentially means that you won't need to update the app if the certificate gets renewed. One of the downsides of this method is that it **does not support** self-signed certificates.

Also, in the implementation above, you probably noticed the *yourHostname.com* string. This value is used for hostname verification and we will talk about this in the next section.

Finally, keep in mind that this is only a brief introduction to certificate pinning, just enough to understand the basic and to be able to prepare your app for penetration tests. For easier implementation and maintenance, certificate pinning should be agreed upon with the administrators of the web service the app is communicating with, to see whether it is a good fit.

If you do decide on certificate pinning, make sure to have a better understanding of the entire process.

Hostname verification

Finally, we've reached the second key part in verifying a TLS connection, which is hostname verification.

A common mistake developers do is setting a permissive hostname verifier, or even worse – accepting all hostnames. This basically means that the attacker can issue a valid certificate with a compromised CA, choose *any* domain name for it, and execute a MITM attack.

If you use OkHttp, the default implementation of the `HostnameVerifier` will be enough to verify your connection to the host. Nevertheless, we will show you a couple of examples how to implement `HostnameVerifier` to gain a better understanding of how it

INFINUM

Preparation phase – What you'll need to know

If you aren't using `CertificatePinner` for hostname verification, you can use the Java's `HostnameVerifier`, as it is the base interface for hostname verification. This interface is supported by OkHttp, you just have to pass it to the hostnameVerifier builder function.

During the TLS handshake, the verification mechanism can call back to implementers of this interface to determine whether this connection should be allowed. The specific verification can be done using the `OkHostnameVerifier`, although you will stumble upon some implementations where `HttpsURLConnection.getDefaultHostnameVerifier()` is used.

Under the hood, this is still using the `OkHostnameVerifier` but from an internal Android version of the same class.

We're ready to check a sample implementation using `OkHostnameVerifier.INSTANCE.verify`:

```
hostnameVerifier { _, session ->
    OkHostnameVerifier.INSTANCE.verify("yourHostname.c
}
```

The code above will check whether the entered hostname *yourHostname.com* is contained inside the certificate of the current `SSLSession`. Only if it is, the verification will succeed. In case you want to trust an entire subdomain, you can use the wildcard pattern notation, but you will have to use the `verifyHostname` method like this:

INFINUM

```
hostnameVerifier { hostname, _ ->
    OkHostnameVerifier.INSTANCE.verifyHostname(hostnam
}
```

The implementation you choose will depend on your environment and all the services your app connects to. If you want to know more about the wildcard rules supported by this method, check this documentation.

Conclusion and notes

Hopefully by this point, you have a better understanding about the TLS connection and how to prepare your app for the next penetration test. Look out for the two key parts of a TLS connection, certificate and hostname verification. If one of these two verifications are broken, the entire TLS connection is nullified and the app becomes an easy target for MITTM attacks.

To deepen the knowledge on handling a proper TLS connection in a WebView and more, check out the OWASP Mobile Security Testing Guide.

Also, if you (have) encounter(ed) any problems during the setup of a proper TLS connection, please check ssl-debugging for answers and solutions.

One last thing – in this post we did not cover the Android Network security configuration, a powerful tool that lets apps customise their network security settings without modifying the app code.

Unfortunately, it is available only since Android 7, but if you are lucky enough to work on minSdk API level 24 or you are okay to have two this approach should definitely be the preferred way for handling your network configurations.

Until the next post on pentesting, have fun coding!

You might feel like the room is spinning if you stare into Marijana Šimag's illustration long enough.

Renato Turić

January 13th, 2020

Read more about

Testing

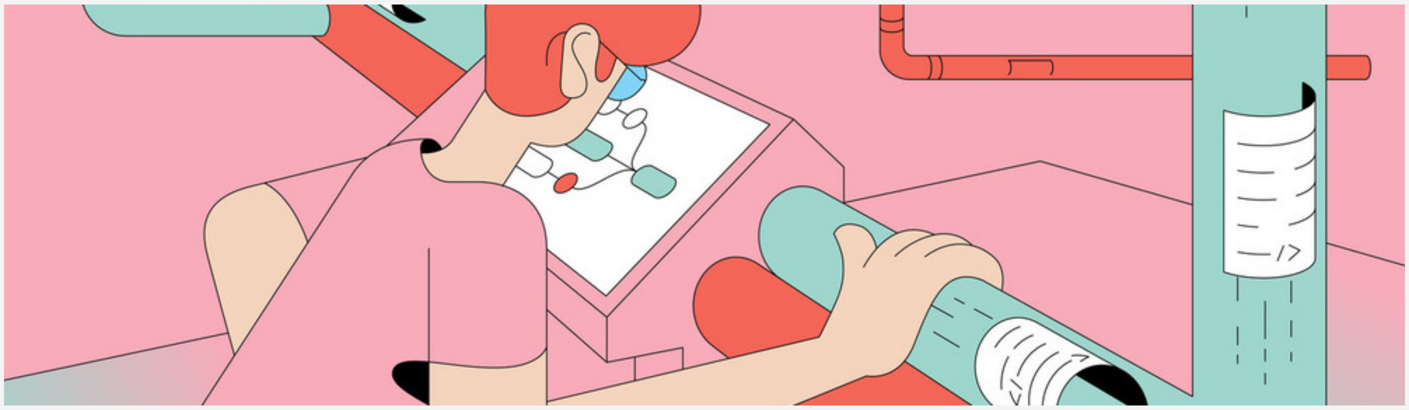
Android

Security

Mobile

Subscribe →

or keep reading articles from our blog...

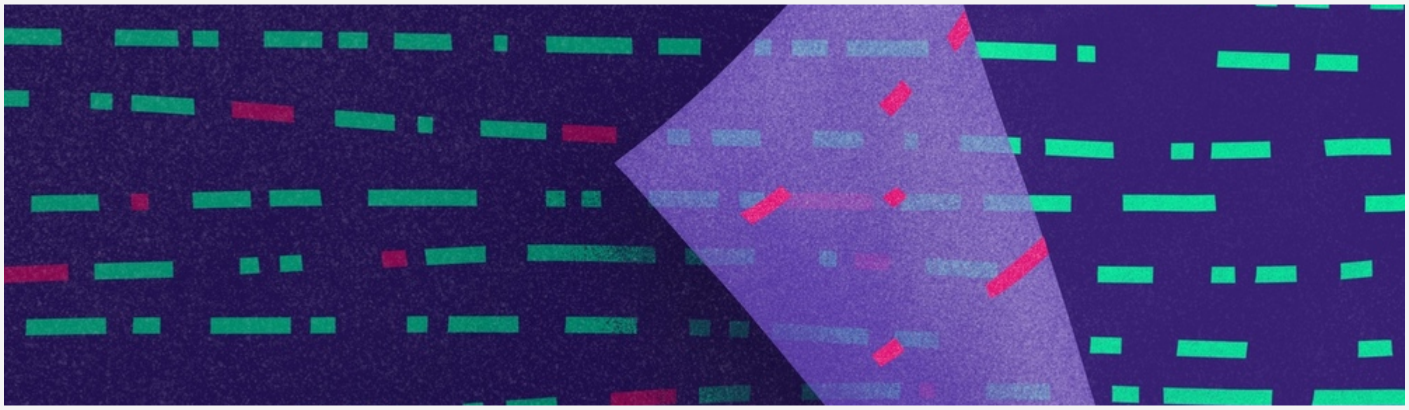


Bitrise vs. CircleCI for Android in a Head-to-H...

Josip K.

7 min read

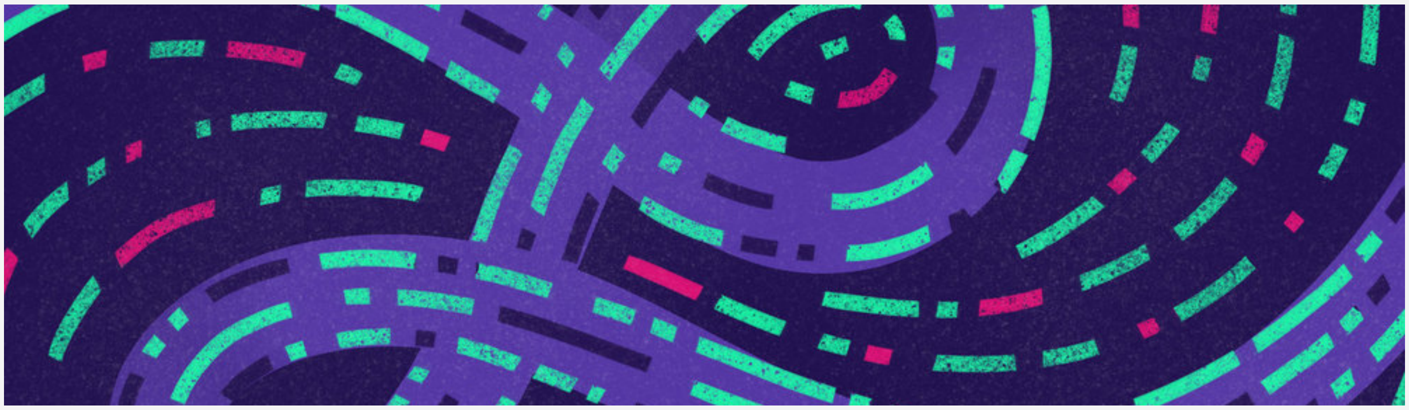
In mobile development, usually, developers are the ones who will set up and maintain CI/CD. In...



What Is Android Lint and How It Helps Write Mai...

INFINUM

When developers are not careful enough, things can go south. Classic developer oversights include...



How to Test Custom Lint Checks

Sven V.
6 min read

In my previous article, I've talked about writing Lint checks and benefits they can bring to...

View all posts →