Security with HTTPS and SSL

The Secure Sockets Layer (SSL)—now technically known as Transport Layer Security (TLS) (http://en.wikipedia.org/wiki/Transport_Layer_Security)—is a common building block for encrypted communications between clients and servers. It's possible that an application might use SSL incorrectly such that malicious entities may be able to intercept an app's data over the network. To help you ensure that this does not happen to your app, this article highlights the common pitfalls when using secure network protocols and addresses some larger concerns about using Public-Key Infrastructure (PKI) (http://en.wikipedia.org/wiki/Public-key_infrastructure).

IN THIS DOCUMENT

Concepts
An HTTP Example
Common Problems Verifying Server Certificates
    Unknown certificate authority
    Self-signed server certificate
    Missing intermediate certificate authority
Common Problems with Hostname Verification
Warnings About Using SSLSocket Directly
Blacklisting
Pinning
Client Certificates

SEE ALSO

- Android Security Overview
- Permissions

Concepts

In a typical SSL usage scenario, a server is configured with a certificate containing a public key as well as a matching private key. As part of the handshake between an SSL client and server, the server proves it has the private key by signing its certificate with public-key cryptography (http://en.wikipedia.org/wiki/Public-key_cryptography).

However, anyone can generate their own certificate and private key, so a simple handshake doesn't prove anything about the server other than that the server knows the private key that matches the public key of the certificate. One way to solve this problem is to have the client have a set of one or more certificates it trusts. If the certificate is not in the set, the server is not to be trusted.

There are several downsides to this simple approach. Servers should be able to upgrade to stronger keys over time ("key rotation"), which replaces the public key in the certificate with a new one. Unfortunately, now the client app has to be updated due to what is essentially a server configuration change. This is especially problematic if the server is not under the app developer's control, for example if it is a third party web service. This approach also has issues if the app has to talk to arbitrary servers such as a web browser or email app.

In order to address these downsides, servers are typically configured with certificates from well known issuers called Certificate Authorities (CAs) (http://en.wikipedia.org/wiki/Certificate_authority). The host platform generally contains a list of well known CAs that it trusts. As of Android 4.2 (Jelly Bean), Android currently contains over 100 CAs that are updated in each release. Similar to a server, a CA has a certificate and a private key. When issuing a certificate for a server, the CA signs (http://en.wikipedia.org/wiki/Digital_signature) the server certificate using its private key. The client can then verify that the server has a certificate issued by a CA known to the platform.

However, while solving some problems, using CAs introduces another. Because the CA issues certificates for many servers, you still need some way to make sure you are talking to the server you want. To address this, the certificate issued by the CA identifies the server either with a specific name such as *gmail.com* or a wildcarded set of hosts such as **.google.com*.

The following example will make these concepts a little more concrete. In the snippet below from a command line, the `openssl` (http://www.openssl.org/docs/apps/openssl.html) tool's `s_client` command looks at Wikipedia's server certificate information. It specifies port 443 because that is the default for HTTPS. The command sends the output of `openssl s_client` to `openssl x509`, which formats information about certificates according to the X.509 standard (http://en.wikipedia.org/wiki/X.509). Specifically, the command asks for the subject, which contains the server name information, and the issuer, which identifies the CA.

```
$ openssl s_client -connect wikipedia.org:443 | openssl x509 -noout -subject -issuer
```

```
subject= /serialNumber=sOrr2rKpMVP70Z6E9BT5reY008SJEdYv/C=US/O=*.wikipedia.org/OU=GT03
issuer= /C=US/O=GeoTrust, Inc./CN=RapidSSL CA
```

You can see that the certificate was issued for servers matching **.wikipedia.org* by the RapidSSL CA.

An HTTPS Example

Assuming you have a web server with a certificate issued by a well known CA, you can make a secure request with code as simple this:

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Yes, it really can be that simple. If you want to tailor the HTTP request, you can cast to an `HttpURLConnection` (/reference/java/net/HttpURLConnection.html). The Android documentation for `HttpURLConnection` (/reference/java/net/HttpURLConnection.html) has further examples about how to deal with request and response headers, posting content, managing cookies, using proxies, caching responses, and so on. But in terms of the details for verifying certificates and hostnames, the Android framework takes care of it for you through these APIs. This is where you want to be if at all possible. That said, below are some other considerations.

Common Problems Verifying Server Certificates

Suppose instead of receiving the content from `getInputStream()` (/reference/java/net/URLConnection.html#getInputStream()), it throws an exception:

```
javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException: Tr
        at org.apache.harmony.xnet.provider.jsse.OpenSSLSocketImpl.startHandshake(Oper
        at libcore.net.http.HttpConnection.setupSecureSocket(HttpConnection.java:209)
        at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.makeSslConnection(Https
        at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLConnect
        at libcore.net.http.HttpEngine.sendSocketRequest(HttpEngine.java:290)
        at libcore.net.http.HttpEngine.sendRequest(HttpEngine.java:240)
        at libcore.net.http.HttpURLConnectionImpl.getResponse(HttpURLConnectionImpl.ja
        at libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnectionImpl
        at libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnectionIn
```

This can happen for several reasons, including:

1. The CA that issued the server certificate was unknown
2. The server certificate wasn't signed by a CA, but was self signed
3. The server configuration is missing an intermediate CA

The following sections discuss how to address these problems while keeping your connection to the server secure.

Unknown certificate authority

In this case, the `SSLHandshakeException` (/reference/javax/net/ssl/SSLHandshakeException.html) occurs because you have a CA that isn't trusted by the system. It could be because you have a certificate from a new CA that isn't yet trusted by Android or your app is running on an older version without the CA. More often a CA is unknown because it isn't a public CA, but a private one issued by an organization such as a government, corporation, or education institution for their own use.

Fortunately, you can teach HttpsURLConnection (/reference/javax/net/ssl/HttpsURLConnection.html) to trust a specific set of CAs. The procedure can be a little convoluted, so below is an example that takes a specific CA from an InputStream (/reference/java/io/InputStream.html), uses it to create a KeyStore (/reference/java/security/KeyStore.html), which is then used to create and initialize a TrustManager (/reference/javax/net/ssl/TrustManager.html). A TrustManager [illegible]m uses to validate certificates from the [illegible]ecurity/KeyStore.html) with one or more CAs—those will be the only CAs trusted by that TrustManager (/reference/javax/net/ssl/TrustManager.html).

Given the new TrustManager [illegible]tManager.html), the example initializes a new SSLContext (/reference/javax/net/ssl/SSLContext.html) wh[illegible]ch provides an SSLSocketFactory (/reference/javax/net/ssl/SSLSocketFactory.html) you [illegible]e to override the default SSLSocketFactory (/reference/javax/net/ssl/SSLSocketFactory.html) from [illegible]sURLConnection (/reference/javax/net/ssl/HttpsURLConnection.html). This w[illegible]y the connection will use your CAs for certificate validation.

Here is the example in full using an organizational CA from the [illegible]niversity of Washington:

```
// Load CAs from an InputStream
// (could be from a resource or ByteArrayInp[illegible]eam or ...)
CertificateFactory cf = CertificateFactory.get[illegible]tance("X.509");
// From https://www.washington.edu/itconnect[illegible]rity/ca/load-der.crt
InputStream caInput = new BufferedInputStrea[illegible] FileInputStream("load-der.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
    System.out.println("ca=" + ((X509Certificate[illegible] ca).getSubjectDN());
} finally {
    caInput.close();
}

// Create a KeyStore [illegible]
String keyStoreType = [illegible]
KeyStore keyStore = K[illegible]Type);
keyStore.load(null, n[illegible]
keyStore.setCertifica[illegible]

// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);

// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);

// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL("https://certs.cac.washington.edu/CAtest/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

With a custom TrustManager (/reference/javax/net/ssl/TrustManager.html) that knows about your CAs, the system is able to validate that your server certificate come from a trusted issuer.

Develop > Training > Security with HTTPS and SSL

Graphics & Animation
Building Apps with Connectivity & the Cloud
Building Apps with User Info & Location
Building Apps for Wearables
Best Practices for Interaction & Engagement
Best Practices for User Interface
Best Practices for User Input
Best Practices for Background Jobs
Best Practices for Performance
Best Practices for Security & Privacy
Security Tips
Security with HTTPS and SSL
Developing for Enterprise

Caution: Many web sites describe a poor alternative solution which is to install a TrustManager (/reference/javax/net/ssl/TrustManager.html) that does nothing. If you do this you might as well not be encrypting your communication, because anyone can attack your users at a public Wi-Fi hotspot by using DNS tricks to send your users' traffic through a proxy of their own that pretends to be your server. The attacker can then record passwords and other personal data. This works because the attacker can generate a certificate and—without a TrustManager (/reference/javax/net/ssl/TrustManager.html) that actually validates that the certificate comes from a trusted source—your app could be talking to anyone. So don't do this, not even temporarily. You can always make your app trust the issuer of the server's certificate, so just do it.

Self-signed server certificate

The second case of SSLHandshakeException (/reference/javax/net/ssl/SSLHandshakeException.html) is due to a self-signed certificate, which means the server is behaving as its own CA. This is similar to an unknown certificate authority, so you can use the same approach from the previous section.

You can create your own TrustManager (/reference/javax/net/ssl/TrustManager.html), this time trusting the server certificate directly. This has all of the downsides discussed earlier of tying your app directly to a certificate, but can be done securely. However, you should be careful to make sure your self-signed certificate has a reasonably strong key. As of 2012, a 2048-bit RSA signature with an exponent of 65537 expiring yearly is acceptable. When rotating keys, you should check for recommendations (http://csrc.nist.gov/groups/ST/key_mgmt/index.html) from an authority (such as NIST (http://www.nist.gov/)) about what is acceptable.

Missing intermediate certificate authority

The third case of SSLHandshakeException (/reference/javax/net/ssl/SSLHandshakeException.html) occurs due to a missing intermediate CA. Most public CAs don't sign server certificates directly. Instead, they use their main CA certificate, referred to as the root CA, to sign intermediate CAs. They do this so the root CA can be stored offline to reduce risk of compromise. However, operating systems like Android typically trust only root CAs directly, which leaves a short gap of trust between the server certificate—signed by the intermediate CA—and the certificate verifier, which knows the root CA. To solve this, the server doesn't send the client only it's certificate during the SSL handshake, but a chain of certificates from the server CA through any intermediates necessary to reach a trusted root CA.

To see what this looks like in practice, here's the *mail.google.com* certificate chain as viewed by the openssl (http://www.openssl.org/docs/apps/openssl.html) `s_client` command:

```
$ openssl s_client -connect mail.google.com:443
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google Inc/CN=mail.google.com
   i:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
 1 s:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
   i:/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
---
```

This shows that the server sends a certificate for *mail.google.com* issued by the *Thawte SGC* CA, which is an intermediate CA, and a second certificate for the *Thawte SGC* CA issued by a *Verisign* CA, which is the primary CA that's trusted by Android.

However, it is not uncommon to configure a server to not include the necessary intermediate CA. For example, here is a server that can cause an error in Android browsers and exceptions in Android apps:

```
$ openssl s_client -connect egov.uscis.gov:443
---
Certificate chain
 0 s:/C=US/ST=District Of Columbia/L=Washington/O=U.S. Department of Homeland Security
   i:/C=US/O=VeriSign, Inc./OU=VeriSign Trust Network/OU=Terms of use at https://www.v
---
```

What is interesting to note here is that visiting this server in most desktop browsers does not cause an error like a completely unknown CA or self-signed server certificate would cause. This is because most desktop browsers cache trusted intermediate CAs over time. Once a browser has visited and learned about an intermediate CA from one site, it won't need to have the intermediate CA included in the certificate chain the next time.

Some sites do this intentionally for secondary web servers used to serve resources. For example, they might have their main HTML page served by a server with a full certificate chain, but have servers for resources such as images, CSS, or JavaScript not include the CA, presumably to save bandwidth. Unfortunately, sometimes these servers might be providing a web service you are trying to call from your Android app, which is not as forgiving.

There are two approaches to solve this issue:

- Configure the server to include the intermediate CA in the server chain. Most CAs provide documentation on how to do this for all common web servers. This is the only approach if you need the site to work with default Android browsers at least through Android 4.2.
- Or, treat the intermediate CA like any other unknown CA, and create a `TrustManager` to trust it directly, as done in the previous two sections.

Common Problems with Hostname Verification

As mentioned at the beginning of this article, there are two key parts to verifying an SSL connection. The first is to verify the certificate is from a trusted source, which was the focus of the previous section. The focus of this section is the second part: making sure the server you are talking to presents the right certificate. When it doesn't, you'll typically see an error like this:

```
java.io.IOException: Hostname 'example.com' was not verified
        at libcore.net.http.HttpConnection.verifySecureSocketHostname(HttpConnection.j
        at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLConnect
        at libcore.net.http.HttpEngine.sendSocketRequest(HttpEngine.java:290)
        at libcore.net.http.HttpEngine.sendRequest(HttpEngine.java:240)
        at libcore.net.http.HttpURLConnectionImpl.getResponse(HttpURLConnectionImpl.ja
        at libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnectionImpl
        at libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnectionIm
```

One reason this can happen is due to a server configuration error. The server is configured with a certificate that does not have a subject or subject alternative name fields that match the server you are trying to reach. It is possible to have one certificate be used with many different servers. For example, looking at the *google.com* certificate with `openssl` (http://www.openssl.org/docs/apps/openssl.html) `s_client -connect google.com:443 | openssl x509 -text` you can see that a subject that supports **.google.com* but also subject alternative names for **.youtube.com*, **.android.com*, and others. The error occurs only when the server name you are connecting to isn't listed by the certificate as acceptable.

Unfortunately this can happen for another reason as well: virtual hosting (http://en.wikipedia.org/wiki/Virtual_hosting). When sharing a server for more than one hostname with HTTP, the web server can tell from the HTTP/1.1 request which target hostname the client is looking for. Unfortunately this is complicated with HTTPS, because the server has to know which certificate to return before it sees the HTTP request. To address this problem, newer versions of SSL, specifically TLSv.1.0 and later, support Server Name Indication (SNI) (http://en.wikipedia.org/wiki/Server_Name_Indication), which allows the SSL client to specify the intended hostname to the server so the proper certificate can be returned.

Fortunately, `HttpsURLConnection (/reference/javax/net/ssl/HttpsURLConnection.html)` supports SNI since Android 2.3. Unfortunately, Apache HTTP Client does not, which is one of the many reasons we discourage its use. One workaround if you need to support Android 2.2 (and older) or Apache HTTP Client is to set up an alternative virtual host on a unique port so that it's unambiguous which server certificate to return.

The more drastic alternative is to replace `HostnameVerifier`

(/reference/javax/net/ssl/HostnameVerifier.html) with one that uses not the hostname of your virtual host, but the one returned by the server by default.

> **Caution:** Replacing HostnameVerifier (/reference/javax/net/ssl/HostnameVerifier.html) can be **very dangerous** if the other virtual host is not under your control, because a man-in-the-middle attack could direct traffic to another server without your knowledge.

If you are still sure you want to override hostname verification, here is an example that replaces the verifier for a single URLConnection (/reference/java/net/URLConnection.html) with one that still verifies that the hostname is at least on expected by the app:

```
// Create an HostnameVerifier that hardwires the expected hostname.
// Note that is different than the URL's hostname:
// example.com versus example.org
HostnameVerifier hostnameVerifier = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        HostnameVerifier hv =
            HttpsURLConnection.getDefaultHostnameVerifier();
        return hv.verify("example.com", session);
    }
};

// Tell the URLConnection to use our HostnameVerifier
URL url = new URL("https://example.org/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setHostnameVerifier(hostnameVerifier);
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

But remember, if you find yourself replacing hostname verification, especially due to virtual hosting, it's still **very dangerous** if the other virtual host is not under your control and you should find an alternative hosting arrangement that avoids this issue.

Warnings About Using SSLSocket Directly

So far, the examples have focused on HTTPS using HttpsURLConnection (/reference/javax/net/ssl/HttpsURLConnection.html). Sometimes apps need to use SSL separate from HTTP. For example, an email app might use SSL variants of SMTP, POP3, or IMAP. In those cases, the app would want to use SSLSocket (/reference/javax/net/ssl/SSLSocket.html) directly, much the same way that HttpsURLConnection (/reference/javax/net/ssl/HttpsURLConnection.html) does internally.

The techniques described so far to deal with certificate verification issues also apply to SSLSocket (/reference/javax/net/ssl/SSLSocket.html). In fact, when using a custom TrustManager (/reference/javax/net/ssl/TrustManager.html), what is passed to HttpsURLConnection (/reference/javax/net/ssl/HttpsURLConnection.html) is an SSLSocketFactory (/reference/javax/net/ssl/SSLSocketFactory.html). So if you need to use a custom TrustManager (/reference/javax/net/ssl/TrustManager.html) with an SSLSocket (/reference/javax/net/ssl/SSLSocket.html), follow the same steps and use that SSLSocketFactory (/reference/javax/net/ssl/SSLSocketFactory.html) to create your SSLSocket (/reference/javax/net/ssl/SSLSocket.html).

> **Caution:** SSLSocket (/reference/javax/net/ssl/SSLSocket.html) **does not** perform hostname verification. It is up the your app to do its own hostname verification, preferably by calling getDefaultHostnameVerifier() (/reference/javax/net/ssl/HttpsURLConnection.html#getDefaultHostnameVerifier()) with the expected

> hostname. Further beware that HostnameVerifier.verify() (/reference/javax/net/ssl/HostnameVerifier.html#verify(java.lang.String, javax.net.ssl.SSLSession)) doesn't throw an exception on error but instead returns a boolean result that you must explicitly check.

Here is an example showing how you can do this. It shows that when connecting to *gmail.com* port 443 without SNI support, you'll receive a certificate for *mail.google.com*. This is expected in this case, so check to make sure that the certificate is indeed for *mail.google.com*:

```
// Open SSLSocket directly to gmail.com
SocketFactory sf = SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) sf.createSocket("gmail.com", 443);
HostnameVerifier hv = HttpsURLConnection.getDefaultHostnameVerifier();
SSLSession s = socket.getSession();

// Verify that the certicate hostname is for mail.google.com
// This is due to lack of SNI support in the current SSLSocket.
if (!hv.verify("mail.google.com", s)) {
    throw new SSLHandshakeException("Expected mail.google.com, "
                                    "found " + s.getPeerPrincipal());
}

// At this point SSLSocket performed certificate verificaiton and
// we have performed hostname verification, so it is safe to proceed.

// ... use socket ...
socket.close();
```

Blacklisting

SSL relies heavily on CAs to issue certificates to only the properly verified owners of servers and domains. In rare cases, CAs are either tricked or, in the case of Comodo (http://en.wikipedia.org/wiki/Comodo_Group#Breach_of_security) or DigiNotar (http://en.wikipedia.org/wiki/DigiNotar), breached, resulting in the certificates for a hostname to be issued to someone other than the owner of the server or domain.

In order to mitigate this risk, Android has the ability to blacklist certain certificates or even whole CAs. While this list was historically built into the operating system, starting in Android 4.2 this list can be remotely updated to deal with future compromises.

Pinning

An app can further protect itself from fraudulently issued certificates by a technique known as pinning. This is basically using the example provided in the unknown CA case above to restrict an app's trusted CAs to a small set known to be used by the app's servers. This prevents the compromise of one of the other 100+ CAs in the system from resulting in a breach of the apps secure channel.

Client Certificates

This article has focused on the user of SSL to secure communications with servers. SSL also supports the notion of client certificates that allow the server to validate the identity of a client. While beyond the scope of this article, the techniques involved are similar to specifying a custom TrustManager (/reference/javax/net/ssl/TrustManager.html). See the discussion about creating a custom KeyManager (/reference/javax/net/ssl/KeyManager.html) in the documentation for HttpsURLConnection (/reference/javax/net/ssl/HttpsURLConnection.html).