Segurança com HTTPS e SSL

O Secure Sockets Layer (SSL) — agora conhecido como [Transport Layer Security (TLS)](http://en.wikipedia.org/wiki/Transport_Layer_Security) — é uma base comum para estabelecer comunicações criptografadas entre clientes e servidores. É possível que um aplicativo use o SSL incorretamente de forma que entidades mal-intencionadas consigam interceptar seus dados na rede. Para ajudar a garantir que isso não aconteça com seu aplicativo, este artigo destaca os riscos comuns ao usar protocolos de rede segura e aborda algumas das maiores preocupações sobre o uso da [Infraestrutura de Chave Pública (PKI)](http://en.wikipedia.org/wiki/Public-key_infrastructure).

Conceitos

Em um cenário de uso de SSL típico, um servidor é configurado com um certificado que contém uma chave pública e uma chave privada correspondente. Como parte do handshake entre um cliente e um servidor SSL, o servidor prova que tem a chave privada assinando seu certificado com a [criptografia de chave pública](http://en.wikipedia.org/wiki/Public-key_cryptography).

No entanto, qualquer pessoa pode gerar seu próprio certificado e sua chave privada, portanto, um simples handshake não prova nada sobre o servidor além do fato de ele ter a chave privada que corresponde à chave pública do certificado. Uma maneira de solucionar esse problema é fazer com que o cliente tenha um conjunto de um ou mais certificados de confiança. Se o certificado não estiver nesse conjunto, o servidor não é confiável.

Essa abordagem simples tem várias desvantagens. Os servidores devem poder ser atualizados para chaves mais fortes ao longo do tempo (“rotação de chaves”), o que substitui a chave pública do certificado por uma nova. Infelizmente, isso faz com que o aplicativo do cliente precise ser atualizado devido ao que é, em essência, uma mudança de configuração do servidor. Isso é especialmente problemático se o servidor não for controlado pelo desenvolvedor do aplicativo, por exemplo, se for um serviço da Web de terceiros. Essa abordagem também apresenta problemas se o aplicativo precisar se comunicar com servidores arbitrários, como um navegador da Web ou um aplicativo de e-mail.

Para solucionar essas desvantagens, os servidores geralmente são configurados com emissores conhecidos, chamados [Autoridades de Certificação (CAs)](http://en.wikipedia.org/wiki/Certificate_authority). A plataforma do host geralmente contém uma lista de CAs conhecidas nas quais ela confia. A partir do Android 4.2 (Jelly Bean), o Android contém mais de 100 CAs que são atualizadas a cada versão. Semelhante a um servidor, uma CA tem um certificado e uma chave privada. Ao emitir um certificado para um servidor, a CA [assina](http://en.wikipedia.org/wiki/Digital_signature) o certificado do servidor usando sua chave privada. Então, o cliente pode verificar se o servidor tem um certificado emitido por uma CA conhecida pela plataforma.

No entanto, embora solucione alguns problemas, o uso de CAs introduz um novo. Como a CA emite certificados para muitos servidores, você ainda precisa de uma maneira para garantir que esteja se comunicando com o servidor desejado. Para solucionar isso, o certificado emitido pela CA identifica o servidor com um nome específico, como *gmail.com*, ou com um conjunto curinga de hosts, como *\*.google.com*.

O exemplo a seguir tornará esses conceitos um pouco mais concretos. No snippet abaixo de uma linha de comando, o comando s\_client da ferramenta [openssl](http://www.openssl.org/docs/apps/openssl.html) verifica as informações de certificado do servidor da Wikipedia. Ele especifica a porta 443, pois ela é a porta padrão para HTTPS. O comando envia a saída de openssl s\_client para openssl x509, que formata as informações de certificados de acordo com o [padrão X.509](http://en.wikipedia.org/wiki/X.509). Especificamente, o comando solicita o assunto, que contém as informações do nome do servidor e o emissor, o que identifica a CA.

$ openssl s\_client -connect wikipedia.org:443 | openssl x509 -noout -subject -issuer

**subject=** /serialNumber=sOrr2rKpMVP70Z6E9BT5reY008SJEdYv/C=US/O=\*.wikipedia.org/OU=GT03314600/OU=See www.rapidssl.com/resources/cps (c)11/OU=Domain Control Validated - RapidSSL(R)/**CN=\*.wikipedia.org**

**issuer=** /C=US/O=GeoTrust, Inc./CN=**RapidSSL CA**

Você pode ver que o certificado foi emitido para servidores que correspondem a *\*.wikipedia.org* pela CA RapidSSL.

Um exemplo de HTTP

Pressupondo que você tem um servidor Web com um certificado emitido por uma CA conhecida, você pode fazer uma solicitação segura com um código simples, como este:

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

Sim, pode ser simples assim. Se quiser adaptar a solicitação HTTP, você pode transmiti-la para um [HttpURLConnection](https://developer.android.com/reference/java/net/HttpURLConnection.html). A documentação do Android para [HttpURLConnection](https://developer.android.com/reference/java/net/HttpURLConnection.html) tem mais exemplos sobre como lidar com cabeçalhos de solicitação e resposta, a publicação de conteúdo, o gerenciamento de cookies, o uso de proxies, o armazenamento de respostas em cache e assim por diante. No entanto, em termos de detalhes sobre a verificação de certificados e nomes de host, a estrutura do Android cuida de tudo para você utilizando essa APIs. Essa é a solução ideal, se possível. Dito isso, apresentamos abaixo algumas considerações.

Problemas comuns ao verificar certificados de servidor

Suponha que, em vez de receber o conteúdo de [getInputStream()](https://developer.android.com/reference/java/net/URLConnection.html#getInputStream()), ele aciona uma exceção:

javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException: Trust anchor for certification path not found.

at org.apache.harmony.xnet.provider.jsse.OpenSSLSocketImpl.startHandshake(OpenSSLSocketImpl.java:374)

at libcore.net.http.HttpConnection.setupSecureSocket(HttpConnection.java:209)

at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.makeSslConnection(HttpsURLConnectionImpl.java:478)

at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLConnectionImpl.java:433)

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.java:282)

at libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnectionImpl.java:177)

at libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnectionImpl.java:271)

Isso pode ocorrer por vários motivos, incluindo:

1. [A CA que emitiu o certificado do servidor era desconhecida](https://developer.android.com/training/articles/security-ssl#UnknownCa)
2. [O certificado do servidor não foi assinado por uma CA, mas sim autoassinado](https://developer.android.com/training/articles/security-ssl#SelfSigned)
3. [A configuração do servidor não incluía uma CA intermediária](https://developer.android.com/training/articles/security-ssl#MissingCa)

As seções a seguir discutem como solucionar esses problemas e manter sua conexão com o servidor segura.

Autoridade de certificação desconhecida

Nesse caso, a [SSLHandshakeException](https://developer.android.com/reference/javax/net/ssl/SSLHandshakeException.html) ocorre porque você tem uma CA na qual o sistema não confia. Pode ser que você tenha um certificado de uma nova CA na qual o Android ainda não confia ou que seu aplicativo esteja executando uma versão mais antiga sem a CA. Com frequência, a CA é desconhecida por não ser pública, mas sim uma CA privada emitida por uma organização como o governo, uma corporação ou uma instituição educacional para uso interno.

Felizmente, você pode ensinar o [HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html) a confiar um conjunto específico de CAs. O procedimento pode ser um pouco complicado, então apresentamos abaixo um exemplo que usa uma CA específica de um [InputStream](https://developer.android.com/reference/java/io/InputStream.html) para criar um [KeyStore](https://developer.android.com/reference/java/security/KeyStore.html), que então é usado para criar e inicializar um [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html). Um [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) é utilizado pelo sistema para validar certificados do servidor e — ao criar de um [KeyStore](https://developer.android.com/reference/java/security/KeyStore.html) com uma ou mais CAs — essas serão as únicas CAs nas quais esse [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) confiará.

Considerando o novo [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html), o exemplo inicializa um novo [SSLContext](https://developer.android.com/reference/javax/net/ssl/SSLContext.html) que fornece um [SSLSocketFactory](https://developer.android.com/reference/javax/net/ssl/SSLSocketFactory.html) que pode ser usando para modificar o [SSLSocketFactory](https://developer.android.com/reference/javax/net/ssl/SSLSocketFactory.html) padrão do [HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html). Dessa forma, a conexão usará suas CAs para validar os certificados.

Veja a seguir um exemplo completo que usa uma CA organizacional da Universidade de Washington:

// Load CAs from an InputStream  
// (could be from a resource or ByteArrayInputStream or ...)  
CertificateFactory cf = CertificateFactory.getInstance("X.509");  
// From https://www.washington.edu/itconnect/security/ca/load-der.crt  
InputStream caInput = new BufferedInputStream(new FileInputStream("load-der.crt"));  
Certificate ca;  
try {  
    ca = cf.generateCertificate(caInput);  
    System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());  
} finally {  
    caInput.close();  
}  
  
// Create a KeyStore containing our trusted CAs  
String keyStoreType = KeyStore.getDefaultType();  
KeyStore keyStore = KeyStore.getInstance(keyStoreType);  
keyStore.load(null, null);  
keyStore.setCertificateEntry("ca", ca);  
  
// 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);

Com um [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) personalizado que conhece seus CAs, o sistema pode validar que seu certificado de servidor foi emitido por uma fonte confiável.

**Atenção:** Muitos sites descrevem uma solução alternativa insuficiente, que consiste em instalar um [**TrustManager**](https://developer.android.com/reference/javax/net/ssl/TrustManager.html)que não faz nada. Fazer isso seria o mesmo que não criptografar suas comunicações, pois qualquer pessoa pode atacar seus usuários em um ponto de acesso de Wi-Fi público usando truques de DNS para enviar o tráfego dos seus usuários por um proxy da escolha dela, que finge ser seu servidor. Com isso, esse indivíduo pode registrar senhas e outros dados pessoais. Isso funciona porque essa pessoa pode gerar um certificado e — sem um [**TrustManager**](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) que realmente valide que o certificado foi emitido por uma fonte confiável — seu aplicativo pode estar se comunicando com qualquer um. Dessa forma, não faça isso, mesmo que temporariamente. Você sempre pode fazer com que seu aplicativo confie no emissor do certificado do servidor, então faça isso.

Certificado de servidor autoassinado

O segundo caso de [SSLHandshakeException](https://developer.android.com/reference/javax/net/ssl/SSLHandshakeException.html) ocorre devido a um certificado autoassinado, o que significa que o servidor se comporta como sua própria CA. Isso é semelhante a uma autoridade de certificação desconhecida, então você pode usar a mesma abordagem da seção anterior.

Você pode criar seu próprio [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html), dessa vez confiando no certificado do servidor diretamente. Essa abordagem tem todas as desvantagens discutidas anteriormente de vincular seu aplicativo diretamente a um certificado, mas ela pode ser adotada de forma segura. No entanto, não deixe de garantir que seu certificado autoassinado tenha uma chave razoavelmente forte. Desde 2012, uma assinatura RSA de 2048 bits com um expoente de 65537 que expira anualmente é aceitável. Ao alternar as chaves, verifique as [recomendações](http://csrc.nist.gov/groups/ST/key_mgmt/index.html) de uma autoridade (como a [NIST](http://www.nist.gov/)) sobre o que é aceitável.

Autoridade de certificação intermediária ausente

O terceiro caso de [SSLHandshakeException](https://developer.android.com/reference/javax/net/ssl/SSLHandshakeException.html) ocorre devido a uma CA intermediária ausente. A maioria das CAs públicas não assinam certificados de servidores diretamente. Em vez disso, elas usam o certificado de sua CA principal, também conhecida como CA raiz, para assinar CAs intermediárias. Isso é feito para que a CA raiz possa ser armazenada off-line para reduzir os riscos de segurança. No entanto, sistemas operacionais como o Android geralmente confiam apenas em CAs raiz diretamente, o que deixa uma lacuna de confiança entre o certificado do servidor — assinado pela CA intermediária — e o verificador do certificado, que conhece a CA raiz. Para solucionar essa questão, o servidor não envia ao cliente somente seu certificado durante o handshake de SSL, mas sim uma cadeia de certificados da CA do servidor por meio de qualquer intermediária necessária para alcançar uma CA raiz de confiança.

Para ver essa abordagem em prática, apresentamos a seguir uma cadeia de certificados do *mail.google.com*visualizada pelo comando s\_client do [openssl](http://www.openssl.org/docs/apps/openssl.html):

$ 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

---

Isso mostra que o servidor envia um certificado para o *mail.google.com* emitido pela CA *Thawte SGC*, que é uma CA intermediária, e um segundo certificado para a CA *Thawte SGC* emitida por uma CA *Verisign*, que é a CA principal na qual o Android confia.

No entanto, não é incomum configurar um servidor para não incluir a CA intermediária necessária. Por exemplo, este é um servidor que pode causar um erro em navegadores do Android e exceções em aplicativos Android:

$ 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/OU=United States Citizenship and Immigration Services/OU=Terms of use at www.verisign.com/rpa (c)05/CN=egov.uscis.gov

i:/C=US/O=VeriSign, Inc./OU=VeriSign Trust Network/OU=Terms of use at https://www.verisign.com/rpa (c)10/CN=VeriSign Class 3 International Server CA - G3

---

É interessante observar que acessar esse servidor na maioria dos navegadores de computador não causa um erro da mesma forma que uma CA totalmente desconhecida ou um certificado de servidor autoassinado causaria. Isso ocorre porque a maioria desses navegadores armazena em cache CAs intermediárias confiáveis ao longo do tempo. Quando um navegador acessa e aprende sobre uma CA intermediária em um site, ele não precisa que essa CA intermediária seja incluída na cadeia de certificados na próxima vez.

Alguns sites fazem isso propositalmente para servidores Web secundários usados para fornecer recursos. Por exemplo, sua página HTML principal pode ser fornecida por um servidor com uma cadeia de certificados completa, mas ter servidores para recursos como imagens, CSS ou JavaScript que não incluem a CA, possivelmente para poupar largura de banda. Infelizmente, às vezes esses servidores podem fornecer um serviço Web que você está tentando chamar por seu aplicativo Android, que não é tão leniente.

Há duas abordagens para solucionar esse problema:

* Configure o servidor para incluir a CA intermediária na cadeia do servidor. A maioria das CAs fornecem documentos sobre como fazer isso para todos os servidores Web comuns. Essa é a única abordagem disponível se você precisa que o site funcione com navegadores Android padrão até a versão 4.2.
* Você também pode tratar a CA intermediária como qualquer outra CA desconhecida e criar um [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) para confiar nelas diretamente, como foi feito nas duas seções anteriores.

Problemas comuns com a verificação do nome do host

Como mencionamos no início deste artigo, a verificação de uma conexão SSL é dividida em duas partes essenciais. A primeira consiste em verificar se o certificado foi emitido por uma fonte confiável, o que era o foco da seção anterior. O foco desta seção é a segunda parte do processo: garantir que você esteja se comunicando com um servidor que apresente o certificado certo. Se isso não ocorrer, você normalmente verá um erro como este:

java.io.IOException: Hostname 'example.com' was not verified

at libcore.net.http.HttpConnection.verifySecureSocketHostname(HttpConnection.java:223)

at libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLConnectionImpl.java:446)

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.java:282)

at libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnectionImpl.java:177)

at libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnectionImpl.java:271)

Um dos motivos pelos quais isso pode acontecer é um erro de configuração do servidor. O servidor é configurado com um certificado que não tem campos de nomes de assunto ou assunto alternativo que correspondam com o servidor que você está tentando alcançar. É possível que um certificado seja usado por diversos servidores. Por exemplo, ao examinar o certificado do *google.com* com o [openssl](http://www.openssl.org/docs/apps/openssl.html) s\_client -connect google.com:443 | openssl x509 -text, você pode ver que ele tem um assunto compatível com *\*.google.com*, além de nomes alternativos de assunto para *\*.youtube.com*, *\*.android.com* e outros. O erro ocorre apenas quando o nome do servidor ao qual você está se conectando foi listado pelo certificado como aceitável.

Infelizmente, isso também pode ocorrer por outro motivo: [hosting virtual](http://en.wikipedia.org/wiki/Virtual_hosting). Ao compartilhar um servidor para mais de um nome de host com HTTP, o servidor Web pode identificar pela solicitação HTTP/1.1 por qual nome de host de destino o cliente está procurando. Esse processo é complicado com o HTTPS, pois o servidor precisa saber qual certificado retornar antes de ver a solicitação HTTP. Para solucionar esse problema, novas versões do SSL, especificamente a TLSv.1.0 e posteriores, oferecem suporte à [Indicação de Nome do Servidor (SNI)](http://en.wikipedia.org/wiki/Server_Name_Indication), que permite que o cliente SSL especifique o nome de host pretendido para o servidor para que o certificado correto seja retornado.

Felizmente, o [HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html) oferece suporte à SNI desde o Android 2.3. Uma solução alternativa se precisar de suporte ao Android 2.2 (e versões mais antigas) e configurar um host virtual alternativo em uma porta exclusiva para que não haja dúvidas de qual certificado de servidor deve ser retornado.

A alternativa mais radical é substituir o [HostnameVerifier](https://developer.android.com/reference/javax/net/ssl/HostnameVerifier.html) por um que use não o nome do host virtual, mas o que foi retornado pelo servidor por padrão.

**Atenção:** Substituir o [**HostnameVerifier**](https://developer.android.com/reference/javax/net/ssl/HostnameVerifier.html) pode ser **muito perigoso** se o outro host virtual não estiver sob seu controle, pois um ataque de interceptação pode direcionar o tráfego para outro servidor sem seu conhecimento.

Se ainda assim você estiver certo de que deseja modificar a verificação do nome do host, veja um exemplo que substitui o verificador por um só [URLConnection](https://developer.android.com/reference/java/net/URLConnection.html) que tenha um que verifique que o nome do host esteja ao menos onde o aplicativo espera:

// 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);

Lembre-se: se você substituir a verificação do nome do host, especialmente devido ao hosting virtual, essa abordagem ainda é **muito perigosa** se o outro host virtual não estiver sob seu controle e é recomendável encontrar uma alternativa de hosting que evite esse problema.

Avisos sobre o uso direto do SSLSocket

Até o momento, os exemplos se concentraram no HTTPS usando [HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html). Às vezes, os aplicativos precisam usar SSL separado do HTTP. Por exemplo, um aplicativo de e-mail pode usar variantes de SSL como SMTP, POP3 ou IMAP. Nesses casos, o aplicativo desejaria usar [SSLSocket](https://developer.android.com/reference/javax/net/ssl/SSLSocket.html) diretamente, da mesma forma que o [HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html) faz internamente.

As técnicas descritas até o momento para solucionar problemas de verificação de certificado também se aplicam ao [SSLSocket](https://developer.android.com/reference/javax/net/ssl/SSLSocket.html). Na verdade, ao usar um [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) personalizado, o que é passado para o [HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html) é um [SSLSocketFactory](https://developer.android.com/reference/javax/net/ssl/SSLSocketFactory.html). Então, se precisar usar um [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) personalizado com um [SSLSocket](https://developer.android.com/reference/javax/net/ssl/SSLSocket.html), siga as mesmas etapas e use esse [SSLSocketFactory](https://developer.android.com/reference/javax/net/ssl/SSLSocketFactory.html) para criar seu [SSLSocket](https://developer.android.com/reference/javax/net/ssl/SSLSocket.html).

**Atenção:** o [**SSLSocket**](https://developer.android.com/reference/javax/net/ssl/SSLSocket.html) **não** realiza a verificação do nome do host. Cabe ao seu aplicativo fazer sua própria verificação do nome do host, preferencialmente chamando [**getDefaultHostnameVerifier()**](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html#getDefaultHostnameVerifier()) com o nome de host esperado. Esteja ciente também de que o [**HostnameVerifier.verify()**](https://developer.android.com/reference/javax/net/ssl/HostnameVerifier.html#verify(java.lang.String,%20javax.net.ssl.SSLSession)) não aciona uma exceção quando há um erro, mas sim retorna um resultado booleano que você deverá verificar explicitamente.

Veja um exemplo de como fazer isso. O exemplo mostra que, ao conectar-se ao *gmail.com* na porta 443 sem suporte à SNI, você receberá um certificado para *mail.google.com*. Isso é o esperado nesse caso, então verifique se o certificado realmente é para o *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();

Lista negra

O SSL depende significativamente de CAs para emitir certificados apenas para os proprietários verificados de servidores e domínios. Em casos raros, CAs são fraudados ou, no caso do [Comodo](http://en.wikipedia.org/wiki/Comodo_Group#Breach_of_security) ou do [DigiNotar](http://en.wikipedia.org/wiki/DigiNotar), violados, resultando em certificados para um nome de host emitidos por indivíduos que não são proprietários do servidor ou domínio.

Para reduzir esse risco, o Android pode inserir certos certificados ou mesmo CAs inteiras em uma lista negra. Embora essa lista tenha sido incorporada historicamente ao sistema operacional, a partir do Android 4.2, ela pode ser remotamente atualizada para solucionar futuros riscos de segurança.

Fixação

Um aplicativo pode se proteger ainda mais contra certificados fraudulentos utilizando uma técnica conhecida como fixação. Ela consiste, basicamente, em usar o exemplo fornecido no caso de CA desconhecida acima para restringir as CAs confiáveis do aplicativo a um pequeno conjunto conhecido que deva ser usado pelos servidores do aplicativo. Isso impede o comprometimento de uma das mais de 100 CAs do sistema, resultando em uma violação do canal seguro dos aplicativos.

Certificados de cliente

Este artigo se concentrou no uso do SSL para proteger as comunicações com servidores. O SSL também oferece suporte ao conceito de certificados de cliente que permitem que o servidor valide a identidade de um cliente. Embora elas estejam além do escopo deste artigo, as técnicas necessárias são semelhantes à especificação de um [TrustManager](https://developer.android.com/reference/javax/net/ssl/TrustManager.html) personalizado. Consulte a discussão sobre como criar um [KeyManager](https://developer.android.com/reference/javax/net/ssl/KeyManager.html) personalizado na documentação do [HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html).

Nogotofail: Uma ferramenta de teste de segurança do tráfego de rede

A Nogotofail é uma ferramenta que proporciona uma forma fácil de confirmar que seus aplicativos estão seguros contra vulnerabilidades e erros de configuração conhecidos do TLS/SSL. Trata-se de uma ferramenta automatizada, eficaz e escalável para testar problemas de segurança de rede em qualquer dispositivo cujo tráfego de rede possa passar por ela.

A Nogotofail é útil em três casos de uso principais:

* Encontrar erros e vulnerabilidades.
* Verificar correções e administrar regressões.
* Entender quais aplicativos e dispositivos geram qual tráfego.

A Nogotofail é compatível com Android, iOS, Linux, Windows, Chrome OS, OSX e, na verdade, com qualquer dispositivo que possa ser usado para conectar-se à Internet. Ela tem um cliente de fácil uso que pode ser usado para definir as configurações e receber notificações no Android e no Linux, além no próprio mecanismo de ataque, que pode ser implantado como roteador, servidor de VPN ou proxy.

Você pode acessar essa ferramenta no [projeto de código aberto Nogotofail](https://github.com/google/nogotofail).