Java security编程简介

**1.背景：**

银企通项目开发中，由于接口面向的是银行。在针对银行接口进行编程的时候，100%会存在安全问题。针对与网络安全传输问题，不管银企通开发，还是银行开发人员都会用到加密，签名，甚至认证，https等安全技术来避免数据被篡改，数据泄露等安全问题。只是这些安全措施要么被银行自己开发，银企通方面只需要封装数据，要么就是银企通开发时按照银行的安全规范进行开发，进行加密等操作。

2-5节为概念，通俗易懂。

1. 为代码片段，源自花旗银行接口CITIReqAndRespUtil.java
2. 为资料链接

**2.常用的安全措施：**

签名（确定为对方发送，且未被篡改），加密（避免明文传输），认证授权（从权限持有方获取部分权限），https(传输的数据再加密，确认交互双方的合法性)

公共密钥加密技术（PKC:Public Key Cryptography）：

该技术基于私有密钥，公有密钥。设计思想：公共密钥可以告诉任何人，私有密钥自身持有，而不告诉任何人。公共密钥与私有密钥之间存在一定的数学关系。例如RSA加密算法就是以数值巨大的数字进行因式分解的困难性进行设计。

关于加密算法，属于密码学范畴，可自行研究学习。

**3.签名**

在实际生活中，签名是很常见的行为，比如，刷卡后，你需要在银行小票上签个字，这就是签名。签名的作用就是，当这个小票被拿到其他地方去的时候，其他人能够看出，这个小票出自于你，至于内容，由于内容是机打小票，所以内容的正确性得到了保障。其他例子？Ok,按指纹。

而在网络传输当中，签名的作用同小票签字一样，确认出自谁手。但是传输内容是流，流是可以被读取，然后被改变的。那么确保内容的正确性的方式就是签名了。签名不再是一个你的名字，而是通过数字签名算法，对内容进行处理，在内容（消息）的最后，生成一串数字签名串。不同的内容，生成的签名串则不同。所以当信息接收人收到被篡改的数据时，签名是不同的。

考虑签名场景，签名的数据，希望接收人能验证签名是否有效，显然接收人必然是一群人，而签名的人只是你个人。鉴于公钥，私钥的属性，显然，对于签名而言，进行签名时是使用的私钥，而验签时使用的公钥。

当收到已签名的数据时，接收人根据数据内容，通过公钥生成接收内容的签名串，通过比对生成的签名串和接收内容后面紧随的签名串，就可以发现是否内容被篡改。若相同，则内容没篡改。反之亦然。

**4.加密**

上文阐释了签名的过程，但是传输时，数据流被读出时，没有进行加密，则出现的明文。居心叵测者，可以通过拦截这些信息，获取他想知道的信息。比如你的银行流水，生辰八字，秘密安排等等。这样通过这些私密信息也能骗取无辜的吃瓜群众。

所以在进行文件传输时，将源文件，原数据进行某种算法的转换，让那些居心叵测者拦截下来是一串看似乱码的数据，他就跟吃了苍蝇一样难受。当然，此时也保证了数据的安全性。

同样的加密算法千千万，也属于密码学范畴，可自行研究学习。Java提供的算法有RSA,MD5,MGF1等等。

考虑加密使用的场景，如果，接收者使用公钥进行解密，想想，公钥是任何人都能从证书中获取的，所以任何人都能解密，即居心叵测者也能解密，读取明文。显然真正的接收者必须通过私钥解密。那么加密时就必须使用公钥。这点跟签名恰恰相反。

**5.https**

为了了解https。先给个例子，不妨想想一下，你写了一封情书，此时被变态的加密了，其他人看不懂，又有你的签名。看起来很安全了。但是你要把这封情书寄给太平洋对面的她。此时，你不需要别的。你需要的是一个信封（邮政来运输）。这个信封的作用无他，就是屏蔽信件本身，并且证明来自何方，去往何处。

同样的，数据进行了签名，加密，但是数据本身还是暴露的，https就像信封一样把数据包起来，外界看起来就是一个物件，而不是你的粉色信纸。当然https搞了更多事情，对数据和签名的整体还进行了加密，使得原有的粉色信纸变为了粗糙的泛黄信封。

当收信人看到这封信时，她不会直接看信的内容，而是确认信的合法性。她信任她心中的中国四川成都的某某小巷的寄件地址和寄件人，而不是来自阿联酋的莫某某。当确认了信封的合法信息之后才看信，否则她根本不关注其他的信。

同样的，https连接的双方，需要给对方ssl证书，在接收人处，会将该证书导入信任库中，这样当发送者，发送使用该证书加密的数据后，接收人先看数据的合法性，发现是在本身的信任库中的加密算法，则再关注数据，否则直接忽略对方的请求。

最后来考虑信封的提供者（邮政），他提供的信封全世界都是一个规范，让任何人都相信这是一个普通的信封，这个物件是安全无害的，可信的。而不是你个人设计的，也许你设计得像大炮一样的信封，收件人看到就觉得怕，觉得危及生命财产安全。

同样的，证书的颁发者需要权威机构记性颁发，而非个人生成，否则收件人就会产生信任危及。

总结一下，https传输的好处就是对数据进行加密保护，改变原有所有的数据展现方式，然后在信息接收方进行验证，该数据的加密证书，是否可信，若可信再解析数据，否则拒绝访问，证书需要由第三方权威机构颁发，否则收方不确定对方证书的合法性。

**6.Java代码demo**

**6.1加载密钥库：**

//keyType为密钥库类型，jks,pkcs12等

KeyStore ks = KeyStore.*getInstance*(keyType);

//keyPath密钥库地址

FileInputStream fis = **new** FileInputStream(keyPath);

//pwd密钥库密码

ks.load(fis, pwd.toCharArray());

fis.close();

**6.2签名：**

// 获取客户端签名的私钥

PrivateKey privateSignKey = (PrivateKey) ks.getKey(driver.getClientSignKeyAlias(), driver.getKeyStorePwd().toCharArray());

X509Certificate signCert = (X509Certificate) ks.getCertificate(driver.getClientSignKeyAlias());

signCert.checkValidity();

// 对xml进行签名

org.apache.xml.security.Init.*init*();

ElementProxy.*setDefaultPrefix*(Constants.*SignatureSpecNS*, "ds");

XMLSignature sig = **new** XMLSignature(xmlDoc, "",XMLSignature.*ALGO\_ID\_SIGNATURE\_RSA*);

Element root = xmlDoc.getDocumentElement();

root.appendChild(sig.getElement());

Transforms transforms = **new** Transforms(xmlDoc);

transforms.addTransform(Transforms.*TRANSFORM\_ENVELOPED\_SIGNATURE*);

transforms.addTransform(Transforms.*TRANSFORM\_C14N\_OMIT\_COMMENTS*);

sig.addDocument("", transforms, Constants.*ALGO\_ID\_DIGEST\_SHA1*);

KeyInfo info = sig.getKeyInfo();

X509Data x509data = **new** X509Data(xmlDoc);

x509data.add(**new** XMLX509IssuerSerial(xmlDoc, signCert));

x509data.add(**new** XMLX509Certificate(xmlDoc, signCert));

info.add(x509data);

sig.sign(privateSignKey);

**6.3加密**

// 获取加密公钥

X509Certificate encryptCert = (X509Certificate) ks

.getCertificate(driver.getCitiEncryptKeyAlias());

encryptCert.checkValidity();

PublicKey publicEncryptKey = encryptCert.getPublicKey();

// 加密已签名的xml

String jceAlgorithmName = "DESede";

KeyGenerator keyGenerator = KeyGenerator.*getInstance*(jceAlgorithmName);

Key symmetricKey = keyGenerator.generateKey();

String algorithmURI = XMLCipher.*RSA\_v1dot5*;

XMLCipher keyCipher = XMLCipher.*getInstance*(algorithmURI);

keyCipher.init(XMLCipher.*WRAP\_MODE*, publicEncryptKey);

EncryptedKey encryptedKey = keyCipher.encryptKey(xmlDoc, symmetricKey);

Element rootElement = xmlDoc.getDocumentElement();

algorithmURI = XMLCipher.*TRIPLEDES*;

XMLCipher xmlCipher = XMLCipher.*getInstance*(algorithmURI);

xmlCipher.init(XMLCipher.*ENCRYPT\_MODE*, symmetricKey);

EncryptedData encryptedData = xmlCipher.getEncryptedData();

KeyInfo keyInfo = **new** KeyInfo(xmlDoc);

keyInfo.add(encryptedKey);

encryptedData.setKeyInfo(keyInfo);

xmlCipher.doFinal(xmlDoc, rootElement, **false**);

**6.4解密**

// 获取解密私钥

PrivateKey privateDecryptKey = (PrivateKey) ks.getKey(driver

.getClientDencryptKeyAlias(), driver.getKeyStorePwd()

.toCharArray());

X509Certificate decryptCert = (X509Certificate) ks

.getCertificate(driver.getClientDencryptKeyAlias());

decryptCert.checkValidity();

org.apache.xml.security.Init.*init*();

XMLCipher cipher = XMLCipher.*getInstance*();

cipher.init(XMLCipher.*DECRYPT\_MODE*, **null**);

EncryptedData encryptedData = cipher.loadEncryptedData(xmlDoc,(Element) dataEL);

EncryptedKey encryptedKey = cipher.loadEncryptedKey(xmlDoc, (Element) keyEL);

**if** (encryptedData != **null** && encryptedKey != **null**) {

String encAlgoURL = encryptedData.getEncryptionMethod().getAlgorithm();

XMLCipher keyCipher = XMLCipher.*getInstance*();

keyCipher.init(XMLCipher.*UNWRAP\_MODE*, privateDecryptKey);

Key encryptionKey = keyCipher.decryptKey( encryptedKey, encAlgoURL );

cipher = XMLCipher.*getInstance*();

cipher.init(XMLCipher.*DECRYPT\_MODE*, encryptionKey);

}

Document decryptedDoc = cipher.doFinal(xmlDoc, (Element) dataEL);

**6.5解签：**

X509Certificate signVerifyCert = (X509Certificate) ks.getCertificate(driver.getCitiDsignKeyAlias());

signVerifyCert.checkValidity();

**boolean** verifySignStatus = **false**;

NodeList sigElement = decryptedDoc.getElementsByTagName("Signature");

**if** (sigElement == **null** || sigElement.getLength() == 0) {

**throw** **new** Exception(

"No XML Digital Signature Found - unable to check the signature");

} **else** {

String BaseURI = "file:";

XMLSignature signature = **new** XMLSignature(

(Element) sigElement.item(0), BaseURI);

KeyInfo keyInfo = signature.getKeyInfo();

**if** (keyInfo == **null**) {

**throw** **new** Exception(

"Could not locate KeyInfo element - unable to check the signature");

} **else** {

**if** (keyInfo.containsX509Data()) {

X509Certificate certFromDoc = keyInfo.getX509Certificate();

**if** (certFromDoc != **null**) {

**int** enCodeCertLengthFrmDocCert = certFromDoc.getEncoded().length;

**int** enCodeCertLengthTobeValidated = signVerifyCert.getEncoded().length;

**if** (enCodeCertLengthFrmDocCert == enCodeCertLengthTobeValidated) {

verifySignStatus = signature.checkSignatureValue(signVerifyCert);

} **else** {

**throw** **new** Exception(

"Signature Verification Failed as Cert available in XML & configured on Plugin Properties are different");

}

}

} **else** {

PublicKey pk = keyInfo.getPublicKey();

**if** (pk != **null**) {

verifySignStatus = signature.checkSignatureValue(signVerifyCert);

} **else** {

**throw** **new** Exception(

"X509 cert and PublicKey not found on signature of XML");

}

}

}

}

**6.6 https**

SSLContext ctx = CitiReqAndRespUtil.*getSSLContext*(driver.getKeyStorePwd(),

driver.getKeyStoreFilePath(), driver.getTrustStorePath());// 初始化ssl上下文，方法详见1111111处

SSLSocketFactory ssf = ctx.getSocketFactory();

// 建立连接

HttpsURLConnection connect = (HttpsURLConnection) connectUrl.openConnection();

connect.setSSLSocketFactory(ssf);

connect.setHostnameVerifier(**new** HostnameVerifier() {

**public** **boolean** verify(String arg0, SSLSession arg1) {

**return** **true**;

}

});

//1111111处，构建ssl上下文

**public** **static** SSLContext getSSLContext(String password,

String keyStorePath, String trustStorePath) **throws** Exception {

// 实例化密钥库

KeyManagerFactory keyManagerFactory = KeyManagerFactory

.*getInstance*(KeyManagerFactory.*getDefaultAlgorithm*());

// 获得密钥库

KeyStore keyStore = *loadKeyStore*("JKS", keyStorePath, password);

// 初始化密钥工厂

keyManagerFactory.init(keyStore, password.toCharArray());

// 实例化信任库

TrustManagerFactory trustManagerFactory = TrustManagerFactory

.*getInstance*(TrustManagerFactory.*getDefaultAlgorithm*());

// 获得信任库

KeyStore trustStore = *loadKeyStore*("JKS",trustStorePath,password);

// 初始化信任库

trustManagerFactory.init(trustStore);

// 实例化SSL上下文

SSLContext ctx = SSLContext.*getInstance*("SSL");

// 初始化SSL上下文

ctx.init(keyManagerFactory.getKeyManagers(),

trustManagerFactory.getTrustManagers(), **null**);

**return** ctx;

}

**7.资料链接**

\* 密钥库工具keytool操作，来一波：

<http://www.cnblogs.com/xdp-gacl/p/3750965.html>

\* https理解：

<http://kingj.iteye.com/blog/2103662>

\* 授权协议：oauth2.0

<http://blog.csdn.net/seccloud/article/details/8192707>

\* Xml定位：XPath

<http://blog.csdn.net/jiangchao858/article/details/63314426>

\* 简单易懂的ssl，不看绝壁后悔：

<http://www.iteye.com/topic/1125183>

\* 最后，关于各类证书，不必在意什么格式，比如cer，der，pem等等。他们都是需要被导入密钥库中才能进行操作。所以需要在意的是，什么类型的密钥库中包含什么格式的证书，这两者的对应关系：

<http://blog.csdn.net/liaomin416100569/article/details/76020675>