**通向八段的道路**

**第十四天**

lifetragedy 08/08/2012

目录

[一、 加密保护我们的web service传输 3](#_Toc332212496)

[二、 基本概念 3](#_Toc332212497)

[2.1 加密解密的基本概念 3](#_Toc332212498)

[2.1.1 对称加密 3](#_Toc332212499)

[2.1.2 非对称加密 4](#_Toc332212500)

[2.1.3 数字签名 4](#_Toc332212501)

[2.1.4 口令保护 5](#_Toc332212502)

[2.2 回过头来看https与证书、公钥、私钥的关系 6](#_Toc332212503)

[2.3 使用jks文件来实现Server-Client间的“公钥加密，私钥解密” 6](#_Toc332212504)

[三、 使用Axis2 Rampart实现Web Service安全规范 7](#_Toc332212505)

[3.1 Axis2与Rampart 7](#_Toc332212506)

[3.2 动手来做例子 8](#_Toc332212507)

[3.2.1 搭建环境 9](#_Toc332212508)

[3.2.2 Service端 10](#_Toc332212509)

[1） org.sky.axis2.security. MyRampartService 10](#_Toc332212510)

[2） service.xml文件 10](#_Toc332212511)

[3）service.properties文件 13](#_Toc332212512)

[3.2.3 编写client端前的准备 14](#_Toc332212513)

[3.2.4 编写client端 17](#_Toc332212514)

[1） org.sky.axis2.security. MyRampartServiceClient 18](#_Toc332212515)

[2） 工程的src\repository\axis2.xml文件 20](#_Toc332212516)

[3） client.properties文件 21](#_Toc332212517)

[四、换一种Web Service Security规范来试试同样的例子 22](#_Toc332212518)

[4.1 换service端的配置 22](#_Toc332212519)

[4.2 换client端的配置 23](#_Toc332212520)

# 加密保护我们的web service传输

在上一天的教程中，我们讲了一个简单的基于” security-constraint”的以指定用户名和密码来保护一个Web Service以及如何用https对这个web service的通讯过程进行保护。虽然它用https来进行保护了，但是我们抛开https，这个web service之间传输的用户名，密码，数据都是明文的。

在我之间教程中曾经提到过，有一种黑客工具叫作sniffer，或者使用MIM-ATTACK（中间件拦截）的方式，也是可以把客户端的流拦截住并且发往黑客主机的，这样我们的用户名和密码就可以被黑客所获取了。

因此，今天我们要讲述的就是如何在web service传输时，使得这个用户名和密码以及相关数据也能被加密。

# 基本概念

我们先来看基本概念，这个基本概念将涉及到PKI的相关领域，**请仔细看完这一章**，要不然后面你将云里雾里然后我劝你从头来过，我将参照麻省理工大学的教程-RSA公司出版的“计算机加密与解密原理”，用最实际的例子和最简化的语言把PKI中最重要的几个概念给大家说清楚。

这次应该是我们第三次要求生成证书请求，证书，签名了，挺折腾的！！！

不折腾你们不行，我要把大家折腾的蛋疼，这次折腾过后就彻底明白了。

被折腾着，痛苦着并最后快活着，好了我废话又多了，下面开始。

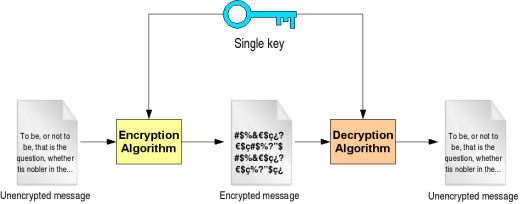
## 2.1 加密解密的基本概念

我们的加密解密分两种：

1. 对称加密（Symmetric Cipher）
2. 非对称加密（Asymmetric Cipher）

### 2.1.1 对称加密

即采用一个密码（密钥）来对一串String进行解密，同样这个密码（密钥）也能对被加密的密文进行解密，至始至终只有一个密码（密钥），因此它叫做对称加密。



### 2.1.2 非对称加密

**这个是最重要的概念之一**

我们知道，对称加密只有一把密钥（你可以把这个密钥看成一个密码）。而非对称加密呢？它有2把密钥，

* 一把我们称为私钥即private key，一把私钥可以对应着无数把公钥，公钥是可以“散播”的。
* 一把我们称为公钥即public key，一堆公钥只能对应着仅有的一把私钥，私钥是绝对不可以“散播”的。

这两把密钥在产生时是被一起产生的，相当于同年同月生一样，即生成私钥时也伴随着生成了公钥。

下面公式来了：

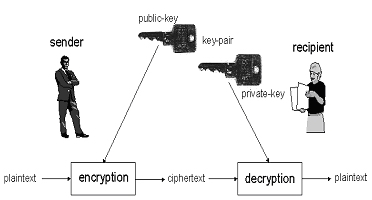
**公钥加密，私钥解密**

大家试想一下哈，我有两把钥匙，一把是用来专门锁门的（加密），一把是专门用来开门的（解密）。那么我用来锁门的那把key掉了，被其它人捡到了，要不要紧？大不了别人可以锁我家的门。

但是，如果我用来开门的这么key掉了？怎么办？被人捡到了人家就可以开我家的门进我家了。

因此，公钥永远被用来加密，可以有多把被多人持有，而私钥永远用来解密且只能主人自己拥有。

**公钥加密，私钥解密**！老老记住，这是永远的公式，也是真理！



### 2.1.3 数字签名

看了上面的“真理”即“**公钥加密，私钥解密**”后有人说了，我偏不信邪我就是要把它们倒过来，好好好！我们一起来看倒过来是什么样的，即成了“**私钥加密，公钥解密**”了。但话不能这样说，真理是不容否定的，但倒过来不是也行？

行，行是行，不过这句话就不能这样说了。

我们知道，公钥是可以多把的，私钥只有一把，因此：

1. 如果我们先把我们的明文用MD5或者SHA1这样的杂凑算法做一个杂凑，得到一堆杂凑值我们称它为报文。
2. 然后呢我们拿着我们的私钥来对着这个得到的杂凑码不管它是MD5还是SHA1，做一个加密运算，就得到了一个“摘要”即Digest。
3. 然后我们把这个摘要和我们的明文一起发送给接收方。
4. 接收方首先用与发送方一样的杂凑函数与接收到的原始明文中计算出这个杂凑计算，得到杂凑值即报文。
5. 接着再用发送方的公用密钥来对这个报文附加的数字签名进行解密，这样，在接收方手上就会有两样东西了。

* 接收方用发送方的公钥与所谓的原始明文运算得到的杂凑值或者称为报文也可称为摘要即digest。
* 接收方收到的由发送方发过来的摘要

1. 将这两要东西，就是两个摘要，它通宵是如下的格式：

|  |
| --- |
| ‎0a f5 b0 3f 38 6b 97 9c 08 62 9b 8b df d7 a0 c6 fe 00 12 08 |

把这两个摘要一比较，完全一致我们就可以说：

从接收方发送来的摘要是出自某某某之手！为什么？

举例来说：

因为我们的公钥和密钥在产生时是一对的，Andy保留了私钥，他把公钥给了Forest。

**If**

Forest用这把公钥和明文得到的摘要如果==Andy用私钥和明文做了杂凑后发来的摘要

**Then**

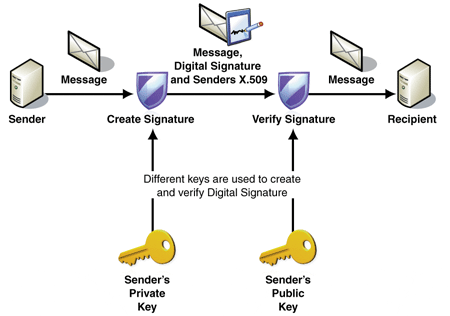
这条消息一定是Andy发过来的。

除非Andy把他的私钥交给了其它人并授权其它人代理他来做这个“私钥签证”

所以，我们得到另一条公式：

**私钥签名，公钥认证**

这也是一条真理，不能违背，这条真理也被称为“数字签名”，这边的“认证”也可以称为“被信任”。



### 2.1.4 口令保护

我们的private key，public key如果一旦真的出现了private key被丢失的情况下怎么办？

不要紧，我们在private key上加一道锁：

Snap12.jpg

这下成了最右边那个带锁的信封了，是不是，这个带锁的信封就是我们的**钥匙袋**。

**你要拿到我的私钥就必须要先打开这个钥匙带，打开这个钥匙带你就必须再需要一把钥匙**。

一般这把钥匙就是一个密码，我们称之为“**口令**”。

来看如下的一个jks的生成来理解吧：

|  |
| --- |
| keytool -genkey -alias shnlap93client -keyalg RSA -keysize 1024 -dname "CN=shnlap93, OU=insurance, O=CTS, L=SH, S=SH, C=CN" -keypass bbbbbb -keystore shnlap93client.jks -storepass bbbbbb |

JKS文件就是“公钥私钥证书”在一起的一种证书格式。

**一般证书如IE中的证书都只带着公钥（public key），而jsk是同时带有公钥和私钥的证书。**

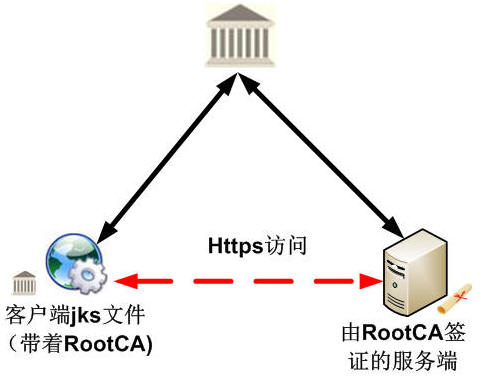
因此，如果你要得到这对密钥，你就必须要“解信封”，解开信封时你就要输入密码（口令），这边的口令就是：keypass即bbbbbb六个小写的b。

## 2.2 回过头来看https与证书、公钥、私钥的关系

CA证书签出一张服务器证书，在签出签书时CA使用自己的私钥，签完后怎么叫作这张被签的证书生效呢？因为这张被签的证书中含有着CA证书中的“公钥”。

然后我们产生一个客户端的证书，该客户端的证书生成后我们把CA的公钥也“烧”到客户端的证书里。

因此，我们有了下面这样的一个关系（见第十三天教程中的3.3小节-需要为我们的Axis2的调用客户端也建立起https中的互信）。



这边的客户端jks文件（带着RootCA），就是带着RootCA的公钥，因为公钥是可以多把的吗，因此RootCA可以把自己的公钥散播给其它人。

于是我们来套“**私钥签名，公钥认证**”这个公式就得到了：

1. 因为客户端的jks文件的公钥可以“认证”RootCA，即RootCA被客户端信任。
2. 因为服务端的ks文件也带着RootCA的公钥，可以“认证”RootCA，即RootCA可以被服务端信任。
3. 因此，客户端可以信任服务端

## 2.3 使用jks文件来实现Server-Client间的“公钥加密，私钥解密”

上面说了，客户端拥有RootCA的public key，服务端也拥有RootCA的public key，所以客户端与服务端可以彼此间也建立起这条信任链。

但是，信任不代表它们可以实现“公钥加密，私钥解密”。

**来看一个例子，这是今天的核心：**

**需求一、如果我们的客户端对一个web service进行加密，然后传到服务端进行解密。**

对于需求一来说：

我们需要在客户端用**服务端的公钥加密**欲传给服务端的消息，然后在服务端得到客户端传过来的加密消息后我们拿着**服务端的私钥进行解密**。

**需求二、**服务端将返回的web service的内容先用**客户端的公钥加密**后传给客户端**，**客户端拿自己的**私钥进行解密。**

对于需求二来说：我们在服务端返回给客户端消息之间先用**客户端的公钥对消息进行加密**，等消息传到客户端的时候，客户端再拿着**客户端自己的私钥进行解密**。

是不是就可以满足上述两个需求了？

于是，根据上面的描述我们来做一件事，即：

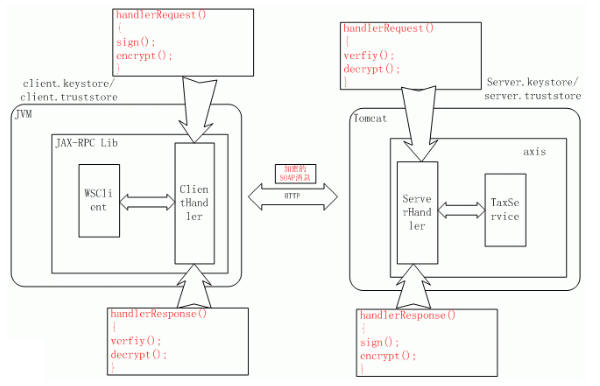
* 把客户端的公钥烧到服务端的jks文件里
* 把服务端的公钥烧到客户端的jks文件里

即可达到上面两个需求中的文字描述了，是不是？

# 使用Axis2 Rampart实现Web Service安全规范

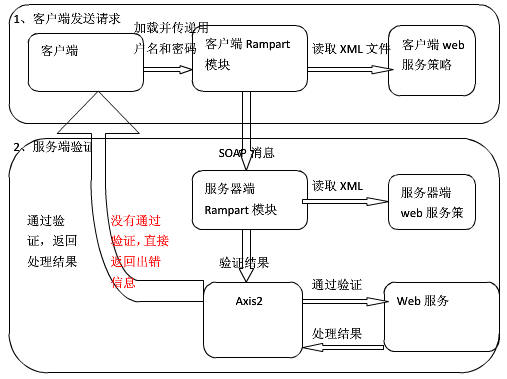
## 3.1 Axis2与Rampart

Web Service安全规范其实就是在讲一套客户端与服务端的webservice之间如何进行认证、加密的手段。在Axis2中，是以handler的模式来实现web service安全规范的。



在旧的Axis版本中，使用的是自己写handler然后来实现客户端与服务端之间的web service加密、认证。而到了Axis21.4版即后续版中，这个handler已经不需要我们手工写了，Axis2为我们提供了一个组件，这个组件叫rampart。

rampart是axis2实现ws-security的一个必须模块。它基于wss4j来完成安全相关的任务，以下是rampart的工作原理：



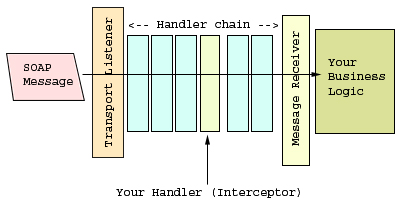
因为Rampart是基于Handler模式的，是对handler的一种封装，它很像一个拦截器，因此有了Rampart我们可以在不需要改动客户端、服务端的代码的情况下只通过修改service端与client端的xml即可实现不同的web service安全规范，而且有了rampart我们只需要提供client端与server端的jks以及相关的“口令”和“用户名（即jks的alias-别名）Rampart就可以自动为我们进行：

加密；

解密；

认证；

而不需要我们再去书写加密、解密、认证的底层API了，够方便吧。



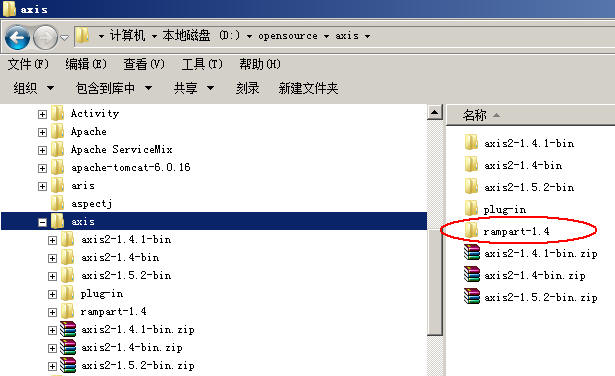
## 3.2 动手来做例子

需求：

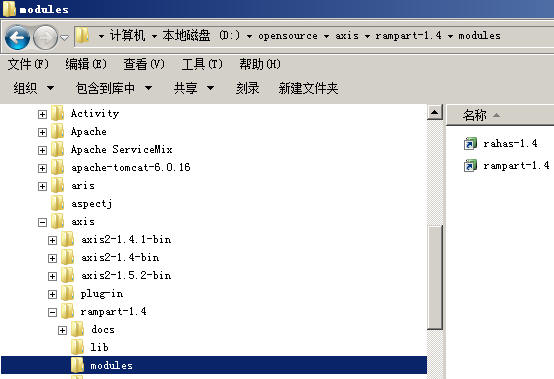
实现客户端访问一个webservice，彼此间只通过http因此需要对客户端传给服务端的message进行加密，服务端得到客户端传来的加密的message后解密并把要返回给客户端的消息进行加密返回给客户端。客户端得到服务端返回后的被加密的消息后进行解密，并显示在客户端。

### 3.2.1 搭建环境

请下载Rampart1.4(<http://axis.apache.org/axis2/java/rampart/download.html>)。



解压后为下面这样的目录结构：



* 把”modules”里的两个mar拷入你工程的WEB-INF\modules目录下；
* 把lib目录下的jar拷入你工程所在的WEB-INF\lib目录下并加载入eclipse的classpath路径中去；

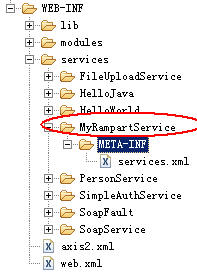
开始实现我们的web service。

### 3.2.2 Service端

### 1） org.sky.axis2.security. MyRampartService

|  |
| --- |
| package org.sky.axis2.security;  import javax.xml.stream.XMLStreamException;  import org.apache.axiom.om.OMElement;  public class MyRampartService {  public String rampartMethod(String userName) throws XMLStreamException {  return "hello: " + userName;  }  } |

### 2） service.xml文件

****

|  |
| --- |
| <service name="MyRampartService">  <description>test rampart service</description>  <operation name="rampartMethod">  <messageReceiver class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />  </operation>  <parameter name="ServiceClass" locked="false">org.sky.axis2.security.MyRampartService  </parameter>  **<module ref="rampart" />**  **<parameter name="InflowSecurity">**  <action>  <items>**Encrypt**</items>  <passwordCallbackClass>**org.sky.axis2.security.RampartPasswordCB**</passwordCallbackClass>  <decryptionPropFile>**service.properties**</decryptionPropFile>  </action>  </parameter>  **<parameter name="OutflowSecurity">**  <action>  <items>Encrypt</items>  <encryptionUser>**shnlap93client**</encryptionUser>  <passwordCallbackClass>**org.sky.axis2.security.RampartPasswordCB**</passwordCallbackClass>  <encryptionPropFile>**service.properties**</encryptionPropFile>  </action>  </parameter>  </service> |

通过上面的描述我们可以看到，这个Rampart的Service需要对两个流即：流出（InflowSecurity）与流进（ OutflowSecurity）进行拦截。

* InflowSecurity时

即client端使用**service端的公钥加密**后提交上来的数据，我们在In时需要调用**service端的私钥进行解密**，因此：

|  |
| --- |
| <action>  <items>Encrypt</items>  <passwordCallbackClass>org.sky.axis2.security.RampartPasswordCB</passwordCallbackClass>  <decryptionPropFile>service.properties</decryptionPropFile>  </action> |

它使用一个实现了PasswordCallBack接口的类来进行调用service端的private key进行解密。

前面我们说过了，由于key都由口令保护着：

Snap12.jpg

因此我们先来看这个” org.sky.axis2.security.RampartPasswordCB”类吧：

|  |
| --- |
| package org.sky.axis2.security;  import java.io.IOException;  import java.util.HashMap;  import java.util.Map;  import javax.security.auth.callback.Callback;  import javax.security.auth.callback.CallbackHandler;  import javax.security.auth.callback.UnsupportedCallbackException;  import org.apache.ws.security.WSPasswordCallback;  public class RampartPasswordCB implements CallbackHandler {  public void handle(Callback[] callbacks) throws IOException,  UnsupportedCallbackException {  for (int i = 0; i < callbacks.length; i++) {  WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];  String id = pwcb.getIdentifer();  System.out.println("id====" + id);  if ("shnlap93client".equals(id)) {  pwcb.setPassword("bbbbbb");  } else if ("shnlap93server".equals(id)) {  pwcb.setPassword("aaaaaa");  } else {  System.out.println("Your are not a authorized user");  throw new UnsupportedCallbackException(callbacks[i],  "Your are not a authorized user");  }  }  }  } |

要调用一个jks文件的内容需要知道两大元素即：

1. 我们先要知道这个jks的Alias；
2. 我们要知道这个jks的保护口令；

比如说我们查看一个test.jks文件的内容因该用如下命令：

keytool –v –list –keystore test.jks

然后回车，此时console会提示我们输入相关的密码即口令，然后我们就可以看到这个jks文件的输出了。

**注意：**

**Jks文件的Alias名必须小写。**

因此我们回过头来看上面的这个类

|  |
| --- |
| if ("shnlap93client".equals(id)) {  pwcb.setPassword("bbbbbb");  } else if ("shnlap93server".equals(id)) {  pwcb.setPassword("aaaaaa");  } else {  System.out.println("Your are not a authorized user");  throw new UnsupportedCallbackException(callbacks[i],  "Your are not a authorized user");  } |

由其是这一段，它就代表着，我们在解密时调用service端的jks的私钥，要调用这个私钥我们必须输入：

Jks的alias与jks的key的保护口令。

这个类写完后请放着，我们在书写客户端时将使用相同的类（一点代码都不需要修改）。

InflowSecurity讲完了，我们继续看下去，来看这个OutflowSecurity

* OutflowSecurity时

流出时即Service端将要返回给Client端的内容加密然后输出，因此OutflowSecurity里做的应该是加密，对吧？

|  |
| --- |
| <parameter name="OutflowSecurity">  <action>  <items>Encrypt</items>  <encryptionUser>shnlap93client</encryptionUser>  <passwordCallbackClass>org.sky.axis2.security.RampartPasswordCB</passwordCallbackClass>  <encryptionPropFile>service.properties</encryptionPropFile>  </action>  </parameter> |

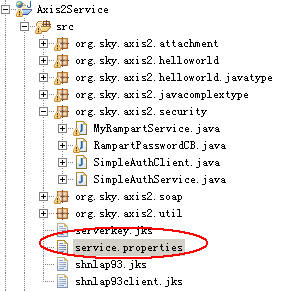
加密用的是client端的公钥，那么Client的公钥也需要输入jks的alias，那么客户端公钥的alias就是这个<encryptionUser>**shnlap93client**</encryptionUser>。

### 3）service.properties文件

最后来看这个service.properties文件：

|  |
| --- |
| org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin  org.apache.ws.security.crypto.merlin.keystore.type=jks  org.apache.ws.security.crypto.merlin.keystore.password=aaaaaa  org.apache.ws.security.crypto.merlin.file=shnlap93.jks |

它位于如下目录结构：



这个文件里描述的就是Service端解密时需要使用的private key的alias与口令。

### 3.2.3 编写client端前的准备

在编写client端前，我们需要生成service端与client端的两对密钥，对吧？我们来看下面的表格：

|  |  |  |  |
| --- | --- | --- | --- |
|  | Alias | Password | File Name |
| Service端 | shnlap93server | aaaaaa | shnlap93.jks |
| Client端 | Shnlap93client | bbbbbb | shnlap93client.jks |

全部为小写的哈！！！记住！

还记得我们前面已经做过几次jks与证书了？

**第一次：**

为tomcat的ssl我们生成了一个自签的CA，把CA放到了我们的IE的根信任域，然后生成了一个服务端的证书，并且用CA对这张证书进行签名，然后把生成的服务端的证书导成了jks，这个jks给了tomcat的server.xml进行配置https时用。**相关命令如下**：

|  |
| --- |
| **生成CA的KEY：**  openssl genrsa -des3 -out ca.key 1024  **从CA的KEY导出自签的CA证书：**  openssl req -new -x509 -key catest.key -out catest.crt  **生成服务端的jks：**  keytool -genkey -alias shnlap93server -keyalg RSA -keysize 1024 -dname "CN=shnlap93.cts.com, OU=insurance, O=CTS, L=SH, S=SH, C=CN" -keypass aaaaaa -keystore shnlap93.jks -storepass aaaaaa  **通过服务端的jks生成证书请求：**  keytool -certreq -alias shnlap93server -sigalg "MD5withRSA" -file shnlap93.csr -keypass aaaaaa -keystore shnlap93.jks -storepass aaaaaa  **用CA对服务端的证书请求进行签名并形成服务端的证书：**  openssl x509 -req -in shnlap93.csr -out shnlap93.crt -CA ca.crt -CAkey ca.key -days 3650 -CAcreateserial -sha1 -trustout -CA ca.crt -CAkey ca.key -days 3650 -CAserial ca.srl -sha1 –trustout  **先将CA导入服务端的jks的信任域即：trustcacerts中去：**  keytool -import -alias rootca -trustcacerts -file ca.crt -keystore shnlap93.jks -storepass aaaaaa  **再把被CA签了名的正式证书导入服务端的jks里**  keytool -import -alias shnlap93server -file shnlap93.crt -keystore shnlap93.jks -storepass aaaaaa |

**第二次：**

为了使得我们的Java应用程序也能和tomcat之间形成https的连接，我们生成了一个client端的jks，然后把CA的公钥导入了我们的client端的jks的信任域里即：trustcacert中，使得我们的client端与server端通过共同信任同一个CA的公钥而达成三角互信关系

相关命令如下：

|  |
| --- |
| **生成客户端的jks文件：**  keytool -genkey -alias **shnlap93client** -keyalg RSA -keysize 1024 -dname "CN=shnlap93, OU=insurance, O=CTS, L=SH, S=SH, C=CN" -keypass **bbbbbb** -keystore client.jks -storepass **bbbbbb**  **将CA导入客户端的信任域即trustcacerts中去：**  keytool -import -alias rootca -trustcacerts -file ca.crt -keystore **shnlap93client.jks** -storepass **bbbbbb** |

**第三次：**

就是这次，我们需要把

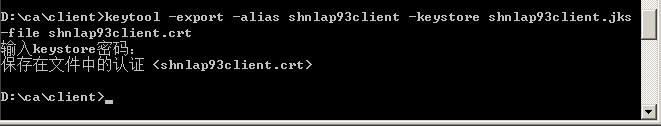
客户端的公钥导入服务端的jks，**使得服务端的jks文件不仅有自己的公钥与私钥同时还带着客户端的公钥**，服务端的这个jks给服务端用。

服务端的公钥导入客户端的jks，**使得客户端的jks文件不仅有自己的公钥与私钥同时还带着服务端的公钥**，客户端的这个jks给客户端用。

**相关命令如下：**

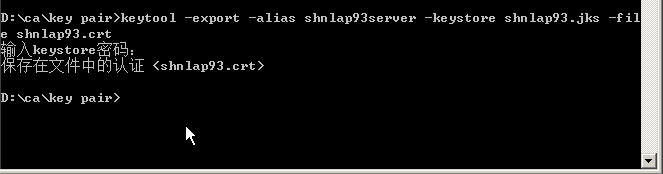
**第一步：**

先把客户端的jks文件导成crt，**jks导成crt后就会自动把jks里的public key带入到这个crt文件**，因此也可以把jks导成crt看成是**public key的导出**。



**第二步：**

再把服务端的jks文件导成crt



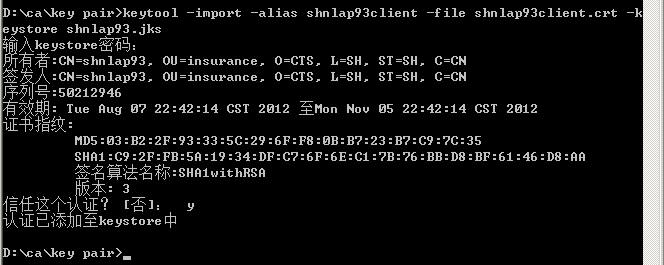
**第三步：**

两个public key都有了，然后互导！！

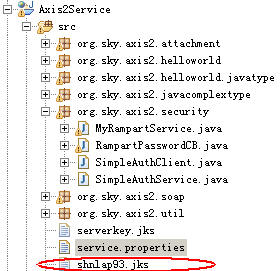
**先将服务端的公钥导入客户端的jks文件里**，由于此时要对client端的jks文件内容进行修改，因此系统会提示输入密码，这个密码就是client端的jks文件的口令即:bbbbbb六个b。



再将客户端的公钥导入服务端的jks文件里，由于此时要对server端的jks文件内容进行修改，因此系统会提示输入密码，这个密码就是server端的jks文件的口令即:aaaaaa六个a。



好了，两个jks文件shnlap93.jks与shnlap93client.jks文件都有了，把shnlap93.jks文件放入工程的src目录下使得工程在编译时会把这个shnlap93.jks编译进入我们的运行时的classpath

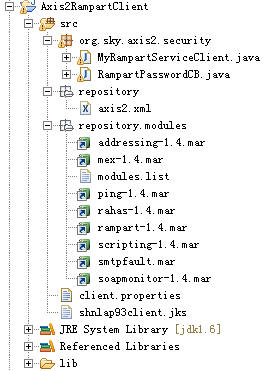


然后下面开始我们来编写我们的client端了。

### 3.2.4 编写client端

我们新建一个eclipse工程，一般Java工程即可。

工程结构如下：



* 把Rampart1.4中的modules目录里的.mar文件都拷入client工程的modules目录下；
* 把Rampart1.4中的lib目录里的jar文件与axis2的lib目录下的jar都拷入client工程的lib目录下并加载入classpath；
* 把我们在3.2.3小节中生成的client端的jks文件即shnlap93client.jks文件放入client 工程的src目录下。
* 把我们在3.2.2小节中生成的” org.sky.axis2.security.RampartPasswordCB”文件也拷入client工程。

### 1） org.sky.axis2.security. MyRampartServiceClient

|  |
| --- |
| package org.sky.axis2.security;  import javax.xml.namespace.QName;  import org.apache.axiom.om.OMAbstractFactory;  import org.apache.axiom.om.OMElement;  import org.apache.axiom.om.OMFactory;  import org.apache.axiom.om.OMNamespace;  import org.apache.axis2.addressing.EndpointReference;  import org.apache.axis2.client.Options;  import org.apache.axis2.client.ServiceClient;  import org.apache.axis2.context.ConfigurationContext;  import org.apache.axis2.context.ConfigurationContextFactory;  public class MyRampartServiceClient {  private static EndpointReference targetEPR = new EndpointReference(  "http://localhost:8080/Axis2Service/services/MyRampartService");  private String getAxis2ConfPath() {  StringBuilder confPath = new StringBuilder();  confPath.append(this.getClass().getResource("/").getPath());  confPath.append("repository");  return confPath.toString();  }  private String getAxis2ConfFilePath() {  String confFilePath = "";  confFilePath = getAxis2ConfPath() + "/axis2.xml";  return confFilePath;  }  public void testMyRampartService() {  Options options = new Options();  options.setAction("urn:rampartMethod");  options.setTo(targetEPR);  ServiceClient sender = null;  String confFilePath = "";  String confPath = "";  try {  **confPath = this.getAxis2ConfPath();**  **confFilePath = getAxis2ConfFilePath();**  System.out.println("confPath======" + confPath);  System.out.println("confFilePath====" + confFilePath);  **ConfigurationContext configContext = ConfigurationContextFactory**  **.createConfigurationContextFromFileSystem(confPath,**  **confFilePath);**  **sender = new ServiceClient(configContext, null);**  sender.setOptions(options);  OMFactory fac = OMAbstractFactory.getOMFactory();  OMNamespace omNs = fac.createOMNamespace(  "http://security.axis2.sky.org", "");  OMElement callMethod = fac.createOMElement("rampartMethod", omNs);  OMElement nameEle = fac.createOMElement("name", omNs);  nameEle.setText("Wang Clare");  callMethod.addChild(nameEle);  OMElement response = sender.sendReceive(callMethod);  System.out.println("response====>" + response);  System.out.println(response.getFirstElement().getText());  } catch (Exception e) {  e.printStackTrace();  } finally {  if (sender != null)  sender.disengageModule("addressing");  try {  sender.cleanup();  } catch (Exception e) {  }  }  }  public static void main(String[] args) {  MyRampartServiceClient rampartServiceClient = new MyRampartServiceClient();  rampartServiceClient.testMyRampartService();  }  } |

这是一个标准的普通的web service访问客户端，没啥花头，唯一需要注意的就是：

|  |
| --- |
| **ConfigurationContext configContext = ConfigurationContextFactory**  **.createConfigurationContextFromFileSystem(confPath,**  **confFilePath);**  **sender = new ServiceClient(configContext, null);** |

这个configContext需要两个参数：

第一个参数指向：工程的src目录下的repository文件夹；

第二个参数指向：工程的src目录下的repository文件夹内的axis2.xml；

### 2） 工程的src\repository\axis2.xml文件

这个文件可以直接从axis2下载解压包中的conf目录下的axis2.xml文件的内容拷贝过来，拷贝过来后在里面添加如下内容（**红色加粗的部分**为我们手工添加的内容）：

|  |
| --- |
| <axisconfig name="AxisJava2.0">  **<module ref="rampart" />**  **<parameter name="OutflowSecurity">**  **<action>**  **<items>Encrypt</items>**  **<encryptionUser>shnlap93server</encryptionUser>**  **<encryptionPropFile>client.properties</encryptionPropFile>**  **</action>**  **</parameter>**  **<parameter name="InflowSecurity">**  **<action>**  **<items>Encrypt</items>**  **<passwordCallbackClass>org.sky.axis2.security.RampartPasswordCB</passwordCallbackClass>**  **<decryptionPropFile>client.properties</decryptionPropFile>**  **</action>**  **</parameter>** |

有了前面service.xml文件中的讲述，这个内容因该不难看懂了吧？

它无非就是：

* 在In时，需要把从服务端用客户端的公钥加密后的数据，在客户端用客户端的私钥进行解密；
* 在Out时，需要用服务端的公钥对数据进行加密再传给服务端。

### 3） client.properties文件

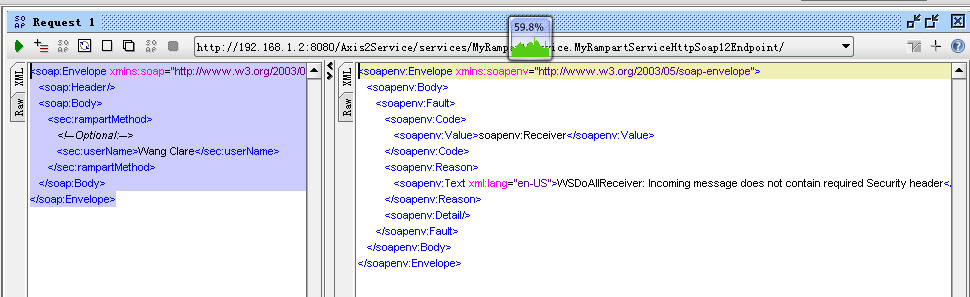
|  |
| --- |
| org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin  org.apache.ws.security.crypto.merlin.keystore.type=jks  org.apache.ws.security.crypto.merlin.keystore.password=bbbbbb  org.apache.ws.security.crypto.merlin.file=shnlap93client.jks |

它和service端的service.properties文件内容一样，是对client 端的私钥即shnlap93client.jks的口令与相关的alias的描述。

然后我们把service端布署入tomcat并启动得到我们的Web Service



我们先试着用SOAP UI来调用：

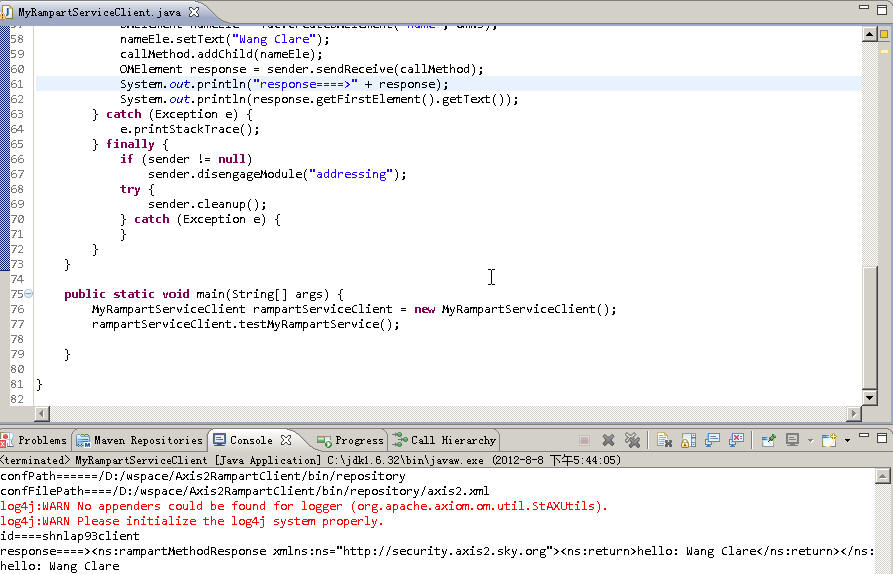


看到右边的输出没？不成功，为什么？

**WSDoAllReceiver: Incoming message does not contain required Security header**

我们的Web Service已经是用户名和密码保护并且我们的request和response都必须是加密的了。

现在我们来在eclipse里运行我们的客户端：



看到输出没？调用成功。

# 四、换一种Web Service Security规范来试试同样的例子

前面在3.1小节中我们说过了：

*Rampart是拦截器原理，因此有了Rampart我们可以在不需要改动客户端、服务端的代码的情况下只通过修改service端与client端的xml即可实现不同的web service安全规范*

我们前面这个例子用的是” Encrypting messages”方式。

我们现在来换成另一种更安全的叫” Sign and encrypt messages”的方式来对我们的web service的服务端与客户端进行安全传输：

## 4.1 换service端的配置

**service.xml文件**

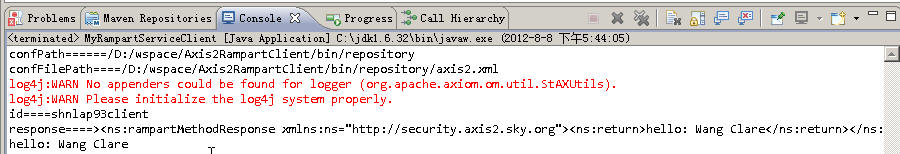
|  |
| --- |
| <module ref="rampart" />  <parameter name="InflowSecurity">  <action>  <items>**Timestamp Signature**</items>  <signaturePropFile>service.properties</signaturePropFile>  </action>  </parameter>    <parameter name="OutflowSecurity">  <action>  <items>**Timestamp Signature**</items>  <user>**shnlap93server**</user>  <passwordCallbackClass>org.sky.axis2.security.RampartPasswordCB</passwordCallbackClass>  <signaturePropFile>**service.properties**</signaturePropFile>  <signatureKeyIdentifier>**DirectReference**</signatureKeyIdentifier>  </action>  </parameter> |

## 4.2 换client端的配置

**axis2.xml文件**

|  |
| --- |
| <module ref="rampart" />  <parameter name="OutflowSecurity">  <action>  <items>**Timestamp Signature**</items>  <user>**shnlap93client**</user>  <signaturePropFile>**client.properties**</signaturePropFile>  <passwordCallbackClass>org.sky.axis2.security.RampartPasswordCB</passwordCallbackClass>  <signatureKeyIdentifier>**DirectReference**</signatureKeyIdentifier>  </action>  </parameter>  <parameter name="InflowSecurity">  <action>  <items>**Timestamp Signature**</items>  <signaturePropFile>**client.properties**</signaturePropFile>  </action>  </parameter> |

好了，除这两个xml配置文件中的内容换一下，其它的代码，properties，jks文件都保持不懂，再重新来运行我们的client端的程序：



成功，真棒！！！结束今天的教程，结束Axis2之旅。

有了这5天的Axis2教程，已经将Axis2的基本知识和概念介绍给了大家，大家通过这5天的学习再结合Axis2自带的Sample相信可以完全掌握Axis2了。