**一、总结前一天的学习**

通过上一章我们知道、了解并掌握了Web Server结合App Server是怎么样的一种架构，并且亲手通过Apache的Http Server与Tomcat6进行了整合的实验。

**这样的架构的好处在于：**

ü   减轻App Server端的压力，用Web Server来分压，即Web Server只负责处理静态HTML内容，而App Server专职负责处理Java请求，这对系统的performance是一个极大的提升。

ü   安全，Web Server端没有任何Java源代码包括编译后的东西，对Internet开放的只有Web Server，因此黑客就算通过80端口攻入了我们的Web Server，他能得到什么？除了静态HTML内容，任何逻辑，口令他都得不到，为什么？喏。。。因为我们的App Server“躲”在Web Server的屁股后面呢。

**需要注意的地方：**

ü   如果以这样的架构出现，你的J2EE 工程，必须在web.xml里把那些个划分清楚，比如说：

我们可以知道\*.do, \*.action, \*.jsp是属于JAVA需要解析的东西对吧！

但是，如果你的servlet写成这样

/abc

/123

/def

那么当我们在作映射时，需要把/abc, /123, /def分别写成一行行的JKMount语句，是不是。。。OK，假设我们这个工程有100个servlet（这个算少的哦），你该不会在httpd.conf文件中给我写这样的无聊的东西100行吧？

所以，我们在规划我们的servlet时需要有矩可循，即pattern，因此我才一直强调，大家在servlet命名时必须统一成：

/servlet/myServletabc

这样，我在做这个Web Server到App Server的Mapping时，是不是只要一句：JkMount /servlet/\* ajp13就可以搞定啦？

ü   同样的架构有不同的变种：

²  IIS+Tomcat

因 为微软的IIS本身就是一个Web Server，因此通过IIS和Tomcat的一个插件叫”isapi”的也可以作到这样的架构，但是我强烈不推荐，因为JAVA源于Unix系统，归于 Unix系统，Unix可是不认什么IIS的，一定请一定用Apache，你是JAVA不是多奶（dot net）。

²  Apache+Weblogic

²  IBM HttpServer(Apache的一个变种)+IBM WAS6.x/WAS7.X

²  Tomcat集群

Apache挂N多个tomcat，由tomcat1…tomcat2…tomcat3…等组成

²  Weblogic集群

Apache挂N多个weblogic，由weblogic1…weblogic2…weblogic3…等组成

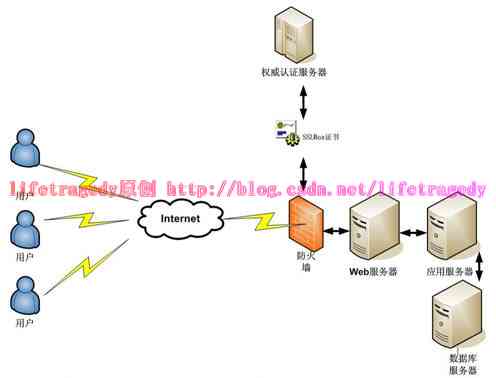
²  WASND（IBMWebsphere App Server Network Deployment）

IBM HttpServer挂N多个WAS，由WAS1…WAS2…WAS3…等组成

**二、HTTPS**

**2.1 HTTPS介绍**

先来看HTTPS的概念



我们一般的http走的是80端口，而https走的是443端口，有什么不一样的地方吗？

很简单，我们拿个telnet命令来作个实验：

telnet127.0.0.1 80，直接就登进了80端口（如果你机器上的Apache开放的话），这样好极了，所有的http中的get, put, post全部可以被我们截获，你的上网帐号，你提交的表单信息全部被别人拦截，就算你对一些信息加了密，对于黑客来说，这些加密被解密只是时间问题，而且一般黑客可以利用云计算，群集计算对你的加密可以进行“硬杀伤”，即穷举算法，利用超大规模集群解密的你的算法会很快，电影里的几十秒解开一个128位的加密不是神话，是真的！！！

因此，我们要让黑客一开始就攻不进来，连门都进不来，何谈拿到我里面的东西，对不对？



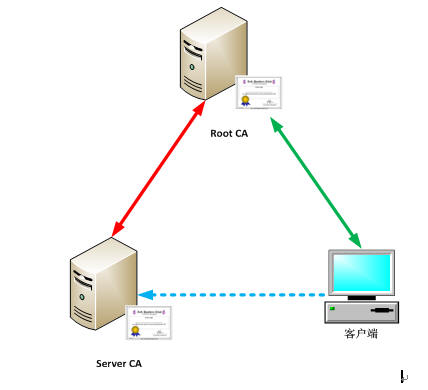
现在我们把我的http通道，变成了https，同时关闭80端口，因此用用户要访问必须经过443端口。好了，我们再来用:

telnet localhost 443

你连telnet都进不去，因此，你就无法再获取http通道内传输的东西了。因此黑客要进入你的网站先要突破这个https，而https使用的是 RSA非对称128位加密，如果是安全交易类甚至会使用RSA 1024位加密算法，除非是世界上最高明的黑客才能突破我们的防线。

**2.2 HTTPS的构成**

要构成HTTPS，我们需要有一张“根证书”，一张“服务器认证”才能做到一般的https，HTTPS还分成“双向认证”，在双向认证的结构中我们需要3 张证书即“根证书”，“服务器认证”，“客户端认证”。为什么需要这么多证书？嘿嘿，下面我们来看HTTPS是怎么构成这个“信任关系链”的。



ü   证书我们称为CA；

ü   根证书叫Root CA；

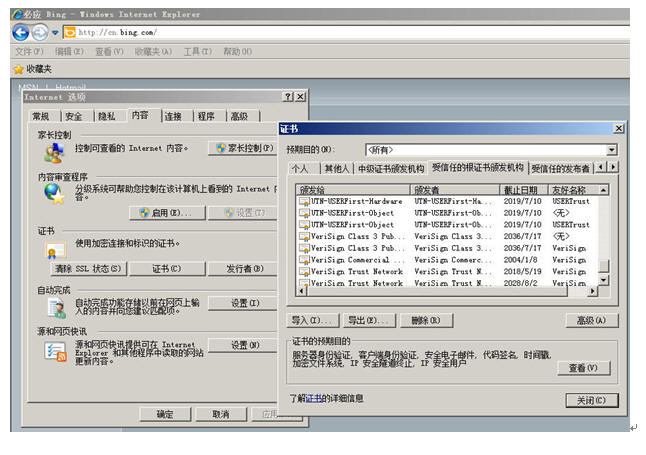
上述这个图什么意思？

**首先**，RootCA是全球的根，这个“树”的根是全球任何IE、FireFox、Safari里的证书库里都有这个RootCA的，因为它们是权威，所以全球的电子证书拿它们做“根”，这些证书比较具有代表性的是：

ü   Verisign

ü   RSA

这两家公司是世界上所有加密算法的“鼻祖”，因此被拜为全球所信任，我们可以在我们的IE中看到这些“根”。



其此，全球的计算器客户默认在装完系统后，都会带有这些ROOT CA，因此“**由ROOT CA签出来的服务器证书将自动被客户端所信任**”。

所以，这个信任关系，就此建立。

在HTTPS是SSL的一种，它们间是如何进行加密传输的呢，就是这个“**信任关系**”，先建立起信任关系，然后再开始数据传输，在加密的世界中“建立信任”就需要用到至少2张证书，即ROOT CA， SERVER CA，我们把这个信任建立的过程称为“**Hands Shake**”，握手协议。

前面说到了，这个握手分单向和双向，包括上述这个图就是一个单向握手，什么叫单向，什么叫双向呢？我们下面来讲解：

ü   单向握手信任

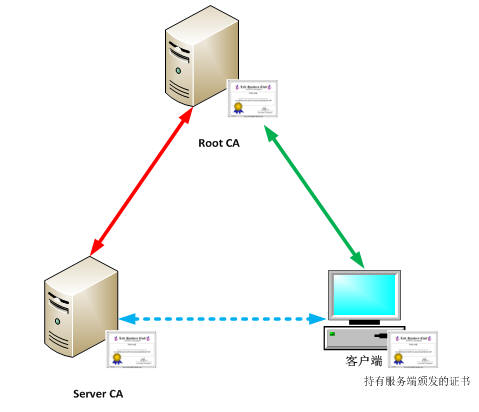
我们又称它为“包二奶协议”，大家想一下，贪官包二奶和二奶说“你跟着我，我每月给你1万块”，他说的这个话，能不能写下来？ 能吗？当然不能，写下来还得了，将来二奶一不爽把这份白纸黑字的东西交到中纪委还不把这烂货给双规了哈？

所以，二奶单向里信任贪官，这就是二奶协议，即客户端认为我访问的这台服务器“是安全的”，因此客户端可以向服务器发送和提交任何东西。

ü   双向握手信任

我们又称它为“君子协定”，呵呵，从这个词表面上来看就知道这个协议有多牢靠了，首先，它是写在字面上的，其次，双方都签署协议这个信任关系怎么样啊？非常牢靠！

即客户端信任服务器，因此客户端可以向服务器发送和提交任何东西。同时，服务器也信任客户端，允许该客户端向我发送和提交东西。



客户端单向信任服务器很简单，只要这个服务器是我客户端信任的顶级根签发出来的证书就行，而服务器如何信任客户端呢？记住下面三句话：

**首先，你这个客户端到我这边来登记一下；**

**其次，我给你签一张证书，你带回家装在你的IE里；**

**最后，每次访问时因为你的IE里装着我服务器签出的证书，所以你是我的会员，所以我信任你；**

**2.3 证书与如何生成证书的基本概念**

通过上面的概念，我们知道了，如果建立起这个HTTPS的环境我们需要至少一张服务器证书，对不对？而且这张服务器证书是需要由客户端信任的“ROOT”级机构所签发出来的。

所以一般生成证书由以下几个步骤构成：

1.       生成一对不对称密钥，即公钥public key和私钥 private key

2.       用密钥产生请求，同时把我的请求交给ROOT机构

3.       ROOT机构对我提交的请求进行“签名”Sign

这个被签完名后的“请求”就称为证书。

上面多出来了密钥，公钥，私钥三个名词，下面来做解释。

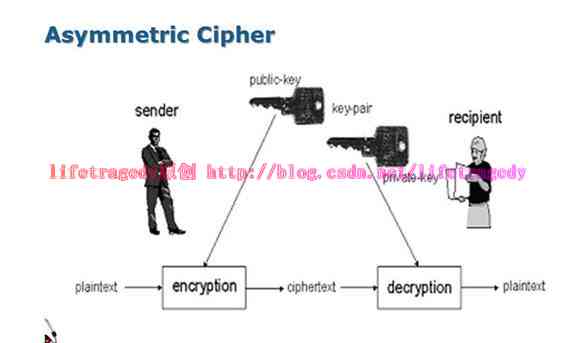
先来看一个真理：

**1）密钥，密钥为一对，即一把公钥，一把私钥**

**2）一把私钥可以对应多把公钥，而这些公钥只可能来源于一把私钥**

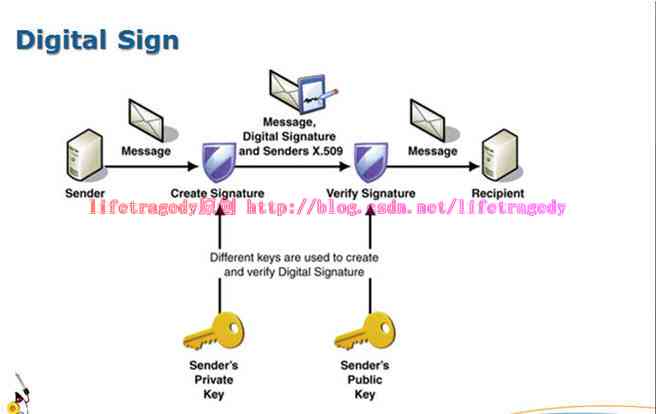
**3）公钥加密，私钥解密**

大家想一下，公钥加密，这是“公”就是人人有这把KEY，都可以用来加密，但是能打开我这扇门的因该只有一个人是吧？这就是为什么说“私钥”解密。



倒过来

**私钥“加密”（被称为签名），公钥“解密”（被称为认证）**



把上面这个“真理”倒过来，呵呵，好玩了，因为public key只能用来加密而解密必须是private key，因此公钥是不能加密的，公钥也不能用来解密，那么它们该怎么做？

HP公司是卖打印机的，它有100个代理，都为它销售打印机。

客户来到了某个代理公司，问：你凭什么说你是HP的代理

代理公司说：请看，这是HP公司给我证书

客户问：你的证书不能伪造吗？

代理公司答：请看，下面有一个防伪条形码

于是，客户把这个防伪条形码用手机拍下来，来到HP公司说：这是不是你们的代理？

HP公司说：你等一下！ **随后HP公司拿出以前给这家代理公司的公钥，对这个签名做一个“解密”（被称为杂凑算法）**，**也算出来**一个防伪条形码，**然后拿这个防伪条形码和客户带来的防伪条形码一比较，完全一致**，所以HP和客户说：您可以完全相信这家公司，这家公司是我代理的。

**这边这个防伪条形码就是代理公司用HP在颁发证书时给它时用HP的私钥的“签名”；**

**HP公司用为当初给代理商发放证书时同时生成的密钥对里的公钥对这个签个名做一个杂凑算法，然后拿算出来的防伪条形码再和客户带来的这个防伪条形码比对的这个过程被称为“认证”；**

一起来看看证书里的防伪条形码吧



我们把它称为电子“指纹”，一般用的是SHA或者是MD5算法，因为MD5和SHA是不可逆唯一性算法，所以把它比喻成“指纹”再恰当不过了。

**2.4 实际开发实验中如何产生证书**

实际产生证书时，我们需要生成请求，但不是说我们把请求交给Verisign或者一些信息机构，它们就帮我们签的，这是要收费的，一般签个名：50-500美金不等，有时还分为每年必须去签一次，要不然就会失效。

所以我们在实际开发环境中，为了做实验或者是搭建模拟环境，不可能会去花钱买个证书的，有时我们环境要搭建几套，怎么办？这钱花的没有明堂的。

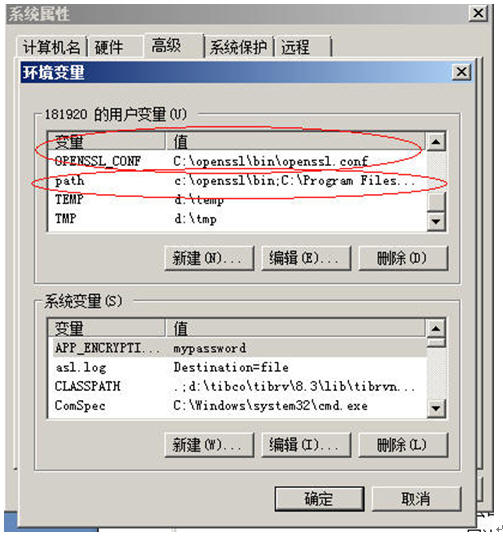
**所以，在实际开发环境中，我们自己来模拟这个ROOTCA，然后用自己模拟出来的ROOTCA去签我们服务器的证书，这个过程就被称为“自签”**。

**2.5 使用OpenSSL来签证书**

OpenSSL 就是这么一个自签，加密的命令行工具，它是从UNIX下分离出来的一个项目，但也有FOR WINDOWS平台的，比如说我给你们用的这个OPENSSL，就是For WIN的，但是由于它是从UNIX/LINUX下产生的，因此它内部的配置还是用的是LINUX/UNIX的盘符与路径，需要手动去校正，当然我已经做好了校正，因此直接打了个压缩包放在了FTP上，大家拿下来后解压后就可以直接用了。

**如果你们是自己从网上官方网站下载的OPENSSL需要手动去改它的盘符和路径，要不然是用不起来的。**

ü   设置环境变量



把c:\openssl\bin\openssl.cnf设成OPENSSL\_CONF这样的一个变量，同时把c:\openssl\bin目录加到你的path里去（根据你们自己的解压后的openssl的实际路径）。

ü   生成根证书所用的密钥

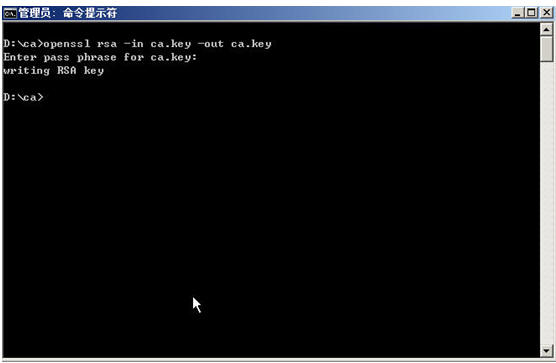


提示输入密码我们使用：aaaaaa

再次输入确认密码

（密钥，由其是private key是由口令保护的）

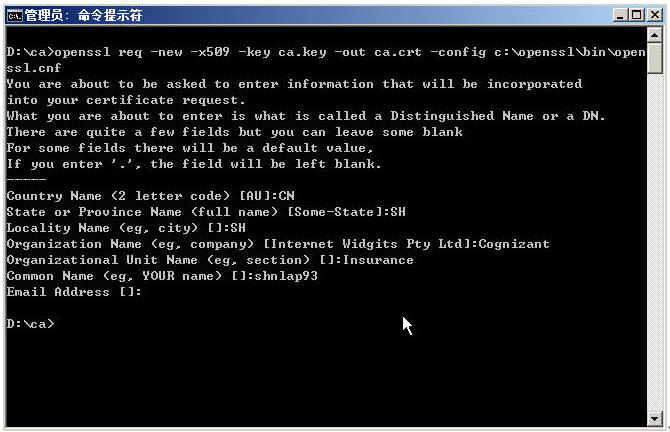
去除CA密钥的口令



为什么我们要把好好的口令保护给去除呢？**这边不是去除而是代表这个证书在被应用程序启动时不需要显示的提示用户输入口令**，要不然我们会出现下面这种情况：

*在启动HTTPS协议的服务器时，一般我们点一下service->apache2.x启动，就启动了，但如果这个https所带的证书是没有经过上述这道手续后处理的话，这个服务在启动时会失败，而需要切换成手动命令行启动，就是黑屏！在黑屏状态下，apache2.x服务器启动时会提示你要求：输入口令，这个太麻烦了，一般启动服务器服务的一定是超级管理员，因此一般情况下没必要在启动相关服务时再输入一遍口令了。*

ü   生成CA即ROOT CA证书并自签



网上有很多说法，说是先产生CA的Request请求，再用ca.key去自签，我给大家介绍一条一步到位的产生ca ROOT证书的命令，为了安全，我们在最后加上“-configC:\openssl\bin\openssl.cnf”，以使openssl工具可以找到相应的config文件（有些系统在指定了OPENSSL\_CONF环境变量后一般就不需要在命令行里去手工指定这个-config变量了）。

由于我们产生的证书为:X509格式，因此需要按照X509格式填入相关的值。

²  AU-国家家的缩写，如：CHINA=CN，美国=USA，英国=UK，日本=JP

²  State or Province Name-省/洲的缩写或者是全称，如：上海=SH

²  Locality Name-城市的全称或者是缩写，如：上海=SH

²  Organization Name-公司名，如：Cognizant

²  Common Name-要安装这台证书的主机名，**证书是和主机名绑定的**，**如果证书里的主机名和你实际的主机名不符，这张证书就是非法的证书**。

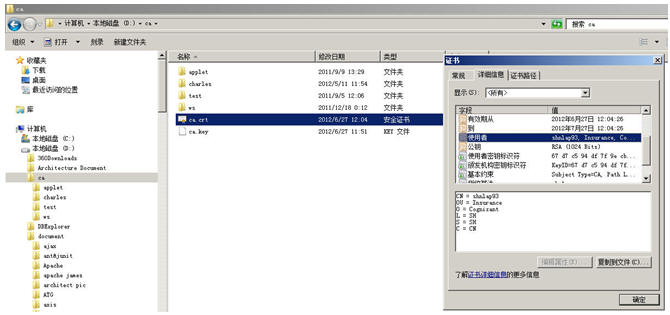
**我们不能够填IP，一定一定要填主机名即域名**[**www.xxx.com**](http://www.xxx.com/)**这样的东西**，比如说我填的是shnlap93，但我的主机怎么知道shnlap93是指:10.225.106.35或者说是指localhost这台机器呢？

打开C:\Windows\System32\drivers\etc\hosts这个文件，如下：

|  |
| --- |
| localhost shnlap93  **10.225.106.35 shnlap93** |

看到了吧？所以当我们使用pint shnlap93时，它是不是就可以知道shnlap93=10.225.106.35啦？

EmailAddress-邮件地址，爱填不填，可以跳过，反正我们是“自签”。



Look，我们的CA证书生成了，可以双击这张证书，查看信息后关闭它。

目前这张ROOT 证书，只是个自签的产品，因为是自签，**一般其它客户端的IE里因此是不会带有这张根证书的。**

要其实客户端也能信任这张根证书，我们必须怎么办？

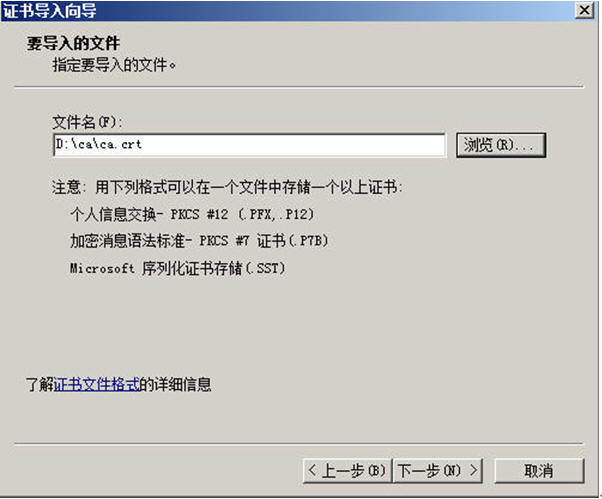
**将它安装到我们的IE的信任域里。**

ü   将ROOT CA导入客户端的根级信任域，有多少台客户端，每个客户端都要导一边这个证书！

**所以说如果我们拥有世界级的根证书该多好啊，电脑上默认就带有我们的证书，因此知道这帮世界级的根证书机构为什么能挣钱了吧？50-500美金签张证书，几秒钟的事，CALL！！！**

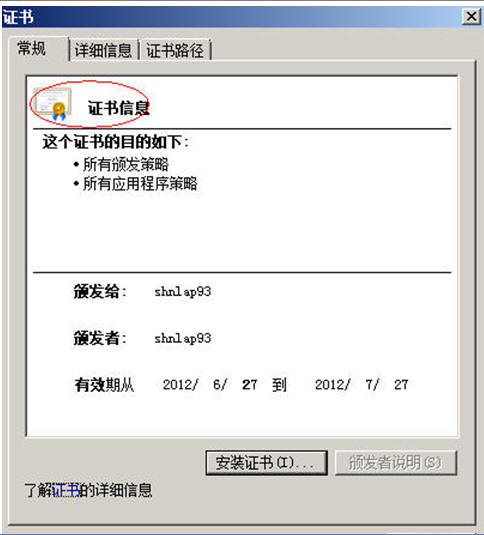


点[导入]按钮



下一步，下一步，此时会有一个弹出框，选“yes（是）”完成导入。

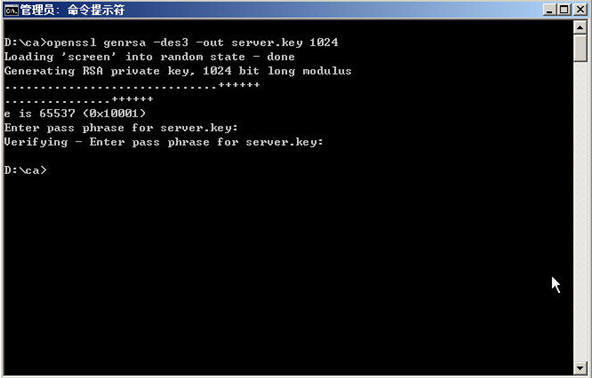
再来打开我们的ca.crt文件



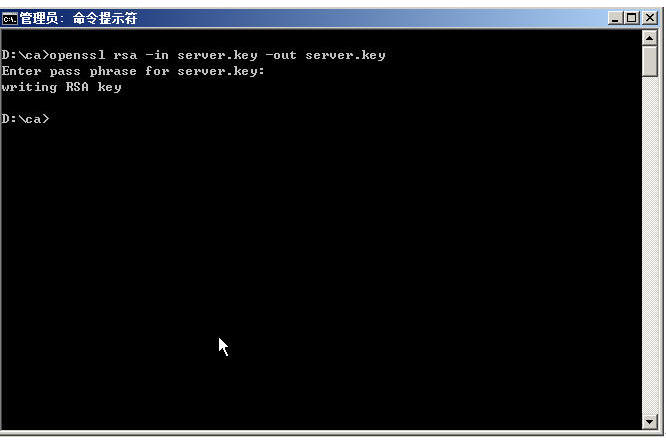
发现了没有，这张证书是有效的证书了，所以在“证书信息”前原有的一个红叉叉，消失了。

ü   生成Web服务器端证书密钥

我们的root证书有了，现在可以生成Web服务器端的证书了，并且用root ca去签名

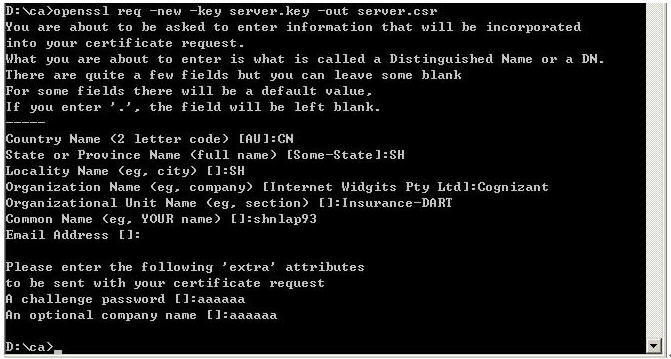


先生成密钥，密码6个a



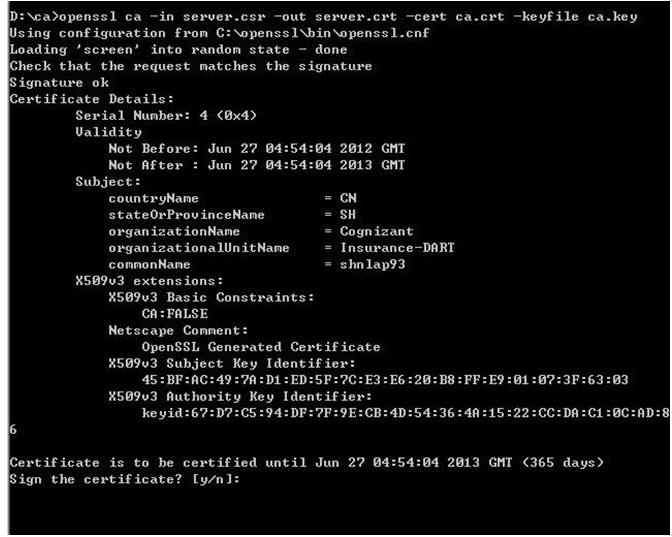
去除密码（提示：enter pass phrase for server.key时输入刚才生成密钥时的密码即6个a。

ü   生成Web服务器端证书的签名请求



生成服务器端证书请求时需要输入server端key的口令，我们为了方便，也用6个a。

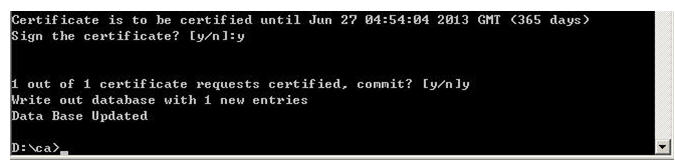
ü   用Root CA去对Web服务器的证书请求即csr(certificate request)进行签名认证



输入y并回车

此时它会提示：

1 out of 1certificate requests certified, commit? [y/n]，再 输入y并回车

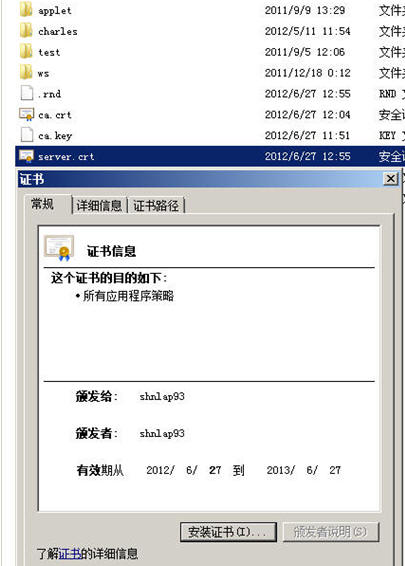


Web服务器的server.crt证书生成完毕。

注：

**如果在操作时有任何错，必须连同生成的.key,.csr, .crt文件全部删除重头来一遍**

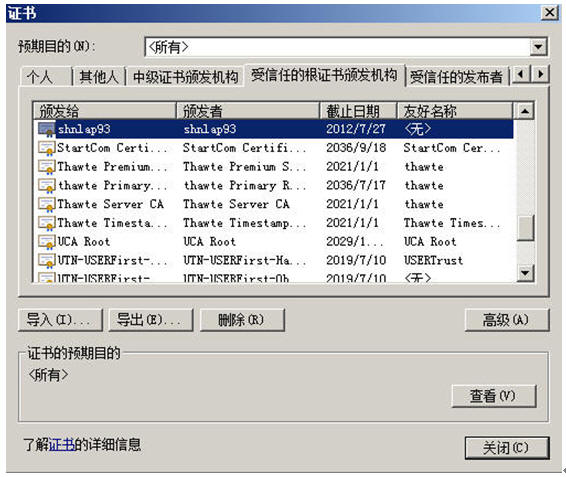
我们来看看，这个server.crt文件，双击它。



**首先，我们看到该证书的“证书信息”前没有红色的大叉，然后是证书信息正是我们刚才输入的内容，为什么没有大叉？**

*因为我们的RootCA根证书装在我们IE的根级信任域里，又因为我们的客户端信任我们的RootCA，因此当我们的客户端打开由RootCA签出来的server.crt时，这根“信任链”被建立了起来，所以客户端自动单向信任我们的server.crt，对不对？*

下面我们来做一个实验，把我们的Root CA从我们的根级信任域中删除。



选中这个shnlap93的根级证书，点[删除]，会弹出两次确认框，选“yes”确认删除掉它。

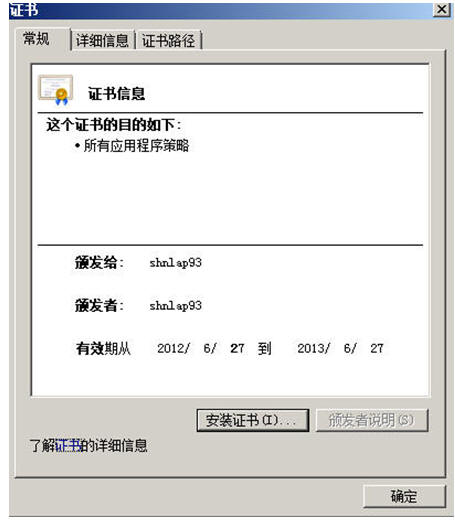
关闭IE，然后我们再次双击我们的server.crt文件，来查看证书内容。



我们看到了什么？“不能验证该证书。

重新导入我们的Root CA至IE的根级信任域（见**将ROOT CA导入客户端的根级信任域**）。

再次打开server.crt查看证书内容。



一切回复正常了。

**2.6 为Apache HttpServer布署https协议**

ü   用文本编辑器打开httpd.conf文件，找到如下这一行

|  |
| --- |
| #Include conf/extra/httpd-ssl.conf |

这行默认是被注释掉的，因此请把它放开，修改成如下

|  |
| --- |
| Include conf/extra/httpd-ssl.conf |

ü   打开D:\tools\httpd\conf\extra\里的httpd-ssl.conf文件

在开头处添加如下这一行语句

|  |
| --- |
| #  # This is the Apache server configuration file providing SSL support.  # It contains the configuration directives to instruct the server how to  # serve pages over an https connection. For detailing information about these  # directives see  #  # Do NOT simply read the instructions in here without understanding  # what they do.  They're here only as hints or reminders.  If you are unsure  # consult the online docs. You have been warned.  #  **LoadModule ssl\_module modules/mod\_ssl.so** |

然后找到下面这一行

|  |
| --- |
| SSLCertificateFile "D:/tools/httpd/ |

把它改成：

|  |
| --- |
| **SSLCertificateFile "D:/tools/httpd/cert/server.crt"** |

再找到下面这一行

|  |
| --- |
| SSLCertificateKeyFile "D:/tools/httpd/ |

把它改成

|  |
| --- |
| **SSLCertificateKeyFile "D:/tools/httpd/cert/server.key"** |

然后把我们在我们的Apache HttpServer的安装目录下手工建一个目录叫cert的目录，并把我们在前面生成的server.crt与server.key文件拷入d:\tools\httpd\cert目录内。

**在httpd.conf文件中搜索“ServerName”**

搜到下面这样的一句

|  |
| --- |
| **ServerName 10.225.101.35:80** |

把它改成你的主机名

|  |
| --- |
| **ServerName shnlap93:80** |

**此处的shnlap93是你的主机名**

再继续在httpd.conf文件中搜索“VirtualHost \*”

搜到下面这一句

|  |
| --- |
|  |

把它改成

|  |
| --- |
|  |

**在D:\tools\httpd\conf\extra\httpd-ssl.conf文件中查找**

搜“VirtualHost \_default\_:443”

然后把位于< VirtualHost \_default\_:443>段内的头三行改成如下格式

²  确保你的http的发布目录在d:/www

²  确保你的HTTPS的主机名为shnlap93:443(**这边的名字和生成证书里的common name必须完全一模一样连大小写都必须一样**)

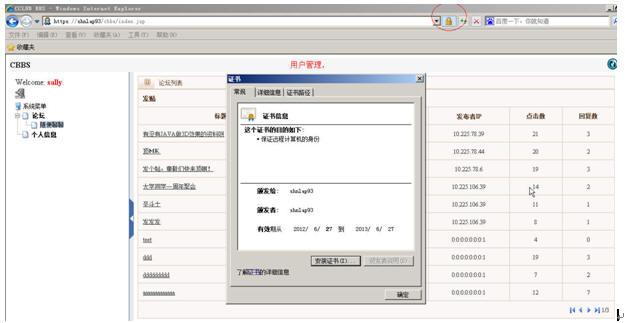
|  |
| --- |
| DocumentRoot "D:/www"  ServerName shnlap93:443  ServerAdmin admin@localhost |

然后在下一个“  ”结束前，填入下面这几行语句

|  |
| --- |
| DirectoryIndex index.html index.htm index.jsp index.action    JkMount /\*WEB-INF ajp13  JkMount /\*j\_spring\_security\_check ajp13  JkMount /\*.action ajp13  JkMount /servlet/\* ajp13  JkMount /\*.jsp ajp13  JkMount /\*.do ajp13  JkMount /\*.action ajp13    JkMount /\*fckeditor/editor/filemanager/connectors/\*.\* ajp13  JkMount /fckeditor/editor/filemanager/connectors/\* ajp13 |

在重启我们的Apache服务前先Test Configuration一下，如果一切无误，可以重启了。

然后我们来实验一下我们的Web Server的https的效果



看到没有，这个红圈圈起来的地方，目前是正常的，显示金黄色的一把钥匙。

**如果你的Root CA没有装入IE的根级信任域，此时你敲入**[**https://shnlap93/cbbs**](https://shnlap93/cbbs)**时，你会被提示说“该证书不被任何”，然后让你点一下“确认”按钮，点完信任后能进入我们的Web应用，但是，原先应该显示“金黄色钥匙”的地方会显示一个红色的圈圈，并且当你查看证书信息时，这个地方也会显示“证书不受信任”，并且显示一个红色的大叉。**

**2.7 为Tomcat也布署https协议**

我们的Apache HttpServer已经走https协议了，不是已经enough了吗？NO，远远不够，如果你没有用到任何App Server即tomcat/weblogic/was那么我们说我们的应用已经走https协议了，**但是因为我们的架构是Web Server + App Server**，**因此，我们的App Server也必须走https协议**。

*如果只是Web Server走https协议，而App Server没有走https协议，这就叫“假https架构”，是一种极其偷赖和不负责任的做法。*

概念同产生Apache的HttpServer的证书一样，只是这边的信任域有点不一样。

Web的信任域就是你的IE里的内容里的证书里的“根级信任域”，App Server的信任域是打不开也不能访问这块地方的，**而且App Server的信任域格式也不是crt文件，而是.jks(javakey store的简称)**。

下面来做一个tomcat的https布署。

原有ca.crt和ca.key继续有用，因为ROOT CA都是一个，而且必须一定始终是唯一的一个，对吧？

我们直接从server.jks来做起。

说JKS，这东西好玩的很，**jks文件其实就是把key文件与crt文件合在一起，以java key store的格式来存储而己**。

**2.8 生成Tomcat的SSL证书**

为Tomcat的server所在的服务器生成一个server.jks文件

很多网上的资料是拿原先的server.crt文件转成keystore文件，其实是不对的，需要单独生成一张server.jks文件。

怎么生成证书？回顾一下上文：

1）  生成KEY

2）  生成证书请求

3）  用CA签名

下面开始使用%JAVA\_HOME%\bin目录下的keytool工具来产生证书

ü   生成JKS密钥对，密码使用6个a，alias代表“别名”，CN代表Common Name，必须与主机名完全一致，错了不要怪我自己负责。

|  |
| --- |
| keytool -genkey -alias shnlap93X509 -keyalg RSA -keysize 1024 -dname "CN=shnlap93, OU=insurance-dart, O=Cognizant, L=SH, S=SH, C=CN" -keypass aaaaaa -keystore shnlap93.jks -storepass aaaaaa |

ü   生成JSK的CSR

|  |
| --- |
| keytool -certreq -alias shnlap93X509 -sigalg "MD5withRSA" -file shnlap93.csr -keypass aaaaaa -keystore shnlap93.jks -storepass aaaaaa |

此处注意：

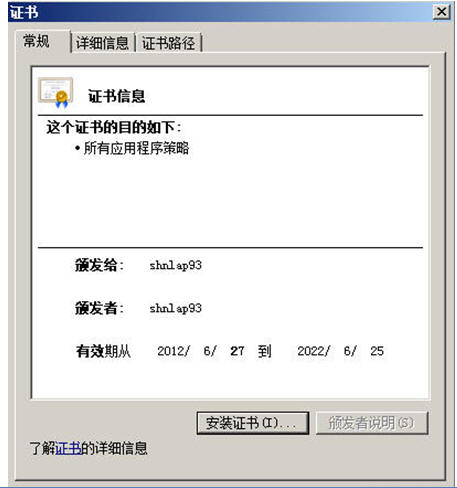
Alias名必须和上面一致

密码和上面一致

ü   使用openssl结合ca.crt与ca.key为jsk的csr来签名认证并产生jks格式的crt

|  |
| --- |
| 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 |

提示yes和no时选yes

         于是，我们有了一张符合jks格式的crt证书叫shnlap93.crt文件，来查看它。

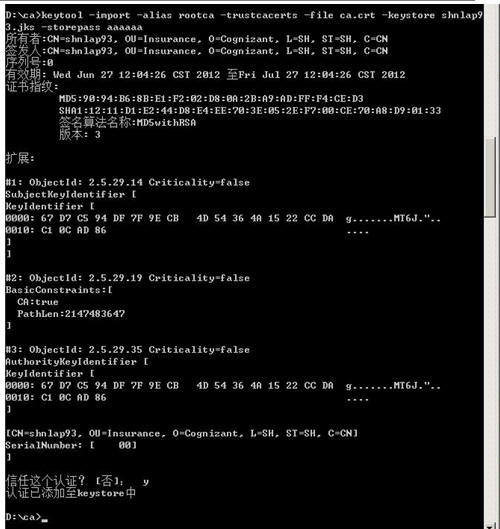
证书上没有红色的大叉，因为我们的Root CA装在我们的IE的根级信任域中，OK，下面来了，生成这个JKS，就是把crt和key合在一起，来了！

ü   生成符合x509格式的jks文件

1)      将Root CA导入jks信任域

|  |
| --- |
| keytool -import -alias rootca -trustcacerts -file ca.crt -keystore shnlap93.jks -storepass aaaaaa |

前面我说了，jks信任域是读不到IE的根级信任域的，因此要手动把ca.crt文件导入jks的信任域



1)      现在我们的shnlap93.jks文件中有两个realm(域)，一个是：

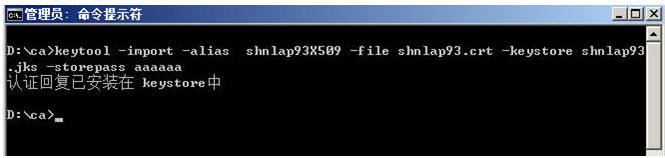
²  本身的csr(**产生的签书请求认证的域**，***还没被认证***，**因为认证的内容变成了shnlap93.crt文件了是不是**？)

²  刚才步骤中导入的Root CA认证域

那么客户端信任Root CA没有问题，但由于server端本身的信任域还只是处于“请求被签名”的状态，那么客户端如何去信任这个jks文件呢？

答案就是：**补链**，**补这根信任链！**

|  |
| --- |
| keytool -import -alias  shnlap93X509 -file shnlap93.crt -keystore shnlap93.jks -storepass aaaaaa |



我们不是生成过shnlap93.crt吗？把它导入jks不就是能够把原有的“正在处于请求被签名认证”这个状态改成“已经被Root CA签名认证” 了吗？对吧？

所以，我们用于布署tomcat的ssl证书的jks格式的文件shnlap93.jks已经完整了。

**2.9 布署Tomcat上的Https协议**

Apache HttpServer走的是443端口，Tomcat走的就是8443端口。

打开tomcat的conf目录下的server.xml，我们来找下面这一段：

|  |
| --- |
|  |

默认情况下，它是被由“ ”这样的标签给注释起来的，我们把它放开

|  |
| --- |
| port="8443" protocol="HTTP/1.1"                connectionTimeout="20000"                secure="true" SSLEnabled="true"                clientAuth="false" sslProtocol="TLS"                keystoreFile="d:/tomcat/conf/shnlap93.jks" keystorePass="aaaaaa" /> |

²  clientAuth=”false”

如果该值为”true”就代表要启用双向认证

²  keystoreFile就是我们生成的keystore文件所在的完全路径

²  keystorePass就是我们生成keystore时的password

重启tomcat,输入<https://shnlap93:8080/cbbs>, 可以看到登录网页,且右上方的SSL连接信息为“金黄色的钥匙”而不是红色大叉，那么一切就成功了。

重启Apache，然后在ie地址栏输入: <https://shnlap93/cbbs>，用sally/abcdefg，一切成功。

注意：

***当启用了https协议后，你在ie地址栏就不能再用localhost了，为什么？因为我们的证书中的CN(Common Name)填入的是主机名，如果你还用:***[***https://localhost***](https://localhost/)***，那么你也能正常进入网址，只是多了几步https不被信任的警告框，并且你右上角的ssl连接信息为红色的大叉，对于这样的https连接，如果换成是购物网站，你敢信任吗？***

**2.10 apache https + tomcat https**

²  假https

前面说过，如果你是在Apache上启用了https而没有在tomcat上启用https协议，那么我们在tomcat中布署一个servlet，含一条打印语句： System.out.println(“”+request.getScheme())，那么它将打印出来**http**。

²  真https

如果我们在Apache上启用了https同时在tomcat上也启用了https，那么我们如果有这样的一条语句：System.out.println(“”+request.getScheme())，它打印出来的将是：**https**。

# Https补充说明

## SSL/TLS 协议的介绍

SSL/TLS 协议（RFC2246 RFC4346）处于 TCP/IP 协议与各种应用层协议之间，为数据通讯提供安全支持。

从协议内部的功能层面上来看，SSL/TLS 协议可分为两层：

1. SSL/TLS 记录协议（SSL/TLS Record Protocol），它建立在可靠的传输层协议（如 TCP）之上，为上层协议提供数据封装、压缩、加密等基本功能。

2. SSL/TLS 握手协议（SSL/TLS Handshake Protocol），它建立在 SSL/TLS 记录协议之上，用于在实际的数据传输开始前，通讯双方进行身份认证、协商加密算法、交换加密密钥等初始化协商功能。

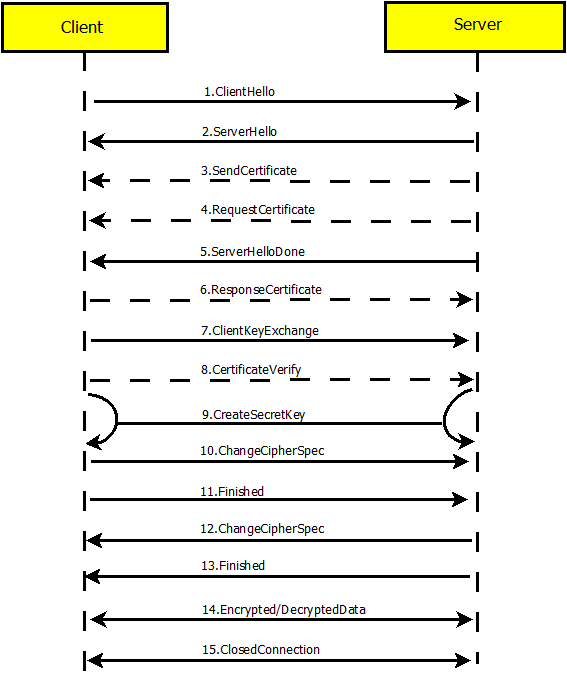
从协议使用方式来看，又可以分成两种类型：

1. SSL/TLS 单向认证，就是用户到服务器之间只存在单方面的认证，即客户端会认证服务器端身份，而服务器端不会去对客户端身份进行验证。首先，客户端发起握手请求，服务器收到握手请求后，会选择适合双方的协议版本和加密方式。然后，再将协商的结果和服务器端的公钥一起发送给客户端。客户端利用服务器端的公钥，对要发送的数据进行加密，并发送给服务器端。服务器端收到后，会用本地私钥对收到的客户端加密数据进行解密。然后，通讯双方都会使用这些数据来产生双方之间通讯的加密密钥。接下来，双方就可以开始安全通讯过程了。

2.SSL/TLS 双向认证，就是双方都会互相认证，也就是两者之间将会交换证书。基本的过程和单向认证完全一样，只是在协商阶段多了几个步骤。在服务器端将协商的结果和服务器端的公钥一起发送给客户端后，会请求客户端的证书，客户端则会将证书发送给服务器端。然后，在客户端给服务器端发送加密数据后，客户端会将私钥生成的数字签名发送给服务器端。而服务器端则会用客户端证书中的公钥来验证数字签名的合法性。建立握手之后过程则和单向通讯完全保持一致。

SSL/TLS 协议建立通讯的基本流程如图 1 所示，

##### 图 1. SSL/TLS 基本流程图



**步骤 1.** ClientHello – 客户端发送所支持的 SSL/TLS 最高协议版本号和所支持的加密算法集合及压缩方法集合等信息给服务器端。

**步骤 2.** ServerHello – 服务器端收到客户端信息后，选定双方都能够支持的 SSL/TLS 协议版本和加密方法及压缩方法，返回给客户端。

**（可选）步骤 3.** SendCertificate – 服务器端发送服务端证书给客户端。

**（可选）步骤 4.** RequestCertificate – 如果选择双向验证，服务器端向客户端请求客户端证书。

**步骤 5.** ServerHelloDone – 服务器端通知客户端初始协商结束。

**（可选）步骤 6.** ResponseCertificate – 如果选择双向验证，客户端向服务器端发送客户端证书。

**步骤 7.** ClientKeyExchange – 客户端使用服务器端的公钥，对客户端公钥和密钥种子进行加密，再发送给服务器端。

**（可选）步骤 8.** CertificateVerify – 如果选择双向验证，客户端用本地私钥生成数字签名，并发送给服务器端，让其通过收到的客户端公钥进行身份验证。

**步骤 9.** CreateSecretKey – 通讯双方基于密钥种子等信息生成通讯密钥。

**步骤 10.** ChangeCipherSpec – 客户端通知服务器端已将通讯方式切换到加密模式。

**步骤 11.** Finished – 客户端做好加密通讯的准备。

**步骤 12.** ChangeCipherSpec – 服务器端通知客户端已将通讯方式切换到加密模式。

**步骤 13.** Finished – 服务器做好加密通讯的准备。

**步骤 14.** Encrypted/DecryptedData – 双方使用客户端密钥，通过对称加密算法对通讯内容进行加密。

**步骤 15.** ClosedConnection – 通讯结束后，任何一方发出断开 SSL 连接的消息。

除了以上的基本流程，SSL/TLS 协议本身还有一些概念需要在此解释说明一下。

**Key：**Key 是一个比特（bit）字符串，用来加密解密数据的，就像是一把开锁的钥匙。

**对称算法（symmetric cryptography）：**就是需要双方使用一样的 key 来加密解密消息算法，常用密钥算法有 Data Encryption Standard（DES）、triple-strength DES（3DES）、Rivest Cipher 2 （RC2）和 Rivest Cipher 4（RC4）。因为对称算法效率相对较高，因此 SSL 会话中的敏感数据都用通过密钥算法加密。

**非对称算法（asymmetric cryptography）：**就是 key 的组成是公钥私钥对 （key-pair），公钥传递给对方私钥自己保留。公钥私钥算法是互逆的，一个用来加密，另一个可以解密。常用的算法有 Rivest Shamir Adleman（RSA）、Diffie-Hellman（DH）。非对称算法计算量大比较慢，因此仅适用于少量数据加密，如对密钥加密，而不适合大量数据的通讯加密。

**公钥证书（public key certificate）：**公钥证书类似数字护照，由受信机构颁发。受信组织的公钥证书就是 certificate authority（CA）。多证书可以连接成证书串，第一个是发送人，下一个是给其颁发证书实体，往上到根证书是世界范围受信组织，包括 VeriSign, Entrust, 和 GTE CyberTrust。公钥证书让非对称算法的公钥传递更安全，可以避免身份伪造，比如 C 创建了公钥私钥，对并冒充 A 将公钥传递给 B，这样 C 与 B 之间进行的通讯会让 B 误认是 A 与 B 之间通讯。

**加密哈希功能（Cryptographic Hash Functions）：** 加密哈希功能与 checksum 功能相似。不同之处在于，checksum 用来侦测意外的数据变化而前者用来侦测故意的数据篡改。数据被哈希后产生一小串比特字符串，微小的数据改变将导致哈希串的变化。发送加密数据时，SSL 会使用加密哈希功能来确保数据一致性，用来阻止第三方破坏通讯数据完整性。SSL 常用的哈希算法有 Message Digest 5（MD5）和 Secure Hash Algorithm（SHA）。

**消息认证码（Message Authentication Code）：** 消息认证码与加密哈希功能相似，除了它需要基于密钥。密钥信息与加密哈希功能产生的数据结合就是哈希消息认证码（HMAC）。如果 A 要确保给 B 发的消息不被 C 篡改，他要按如下步骤做 --A 首先要计算出一个 HMAC 值，将其添加到原始消息后面。用 A 与 B 之间通讯的密钥加密消息体，然后发送给 B。B 收到消息后用密钥解密，然后重新计算出一个 HMAC，来判断消息是否在传输中被篡改。SSL 用 HMAC 来保证数据传输的安全。

**数字签名（Digital Signature）：**一个消息的加密哈希被创建后，哈希值用发送者的私钥加密，加密的结果就是叫做数字签名。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-ssltls/#ibm-pcon)

## JSSE（Java Secure Socket Extension）使用介绍

在 Java SDK 中有一个叫 JSSE（javax.net.ssl）包，这个包中提供了一些类来建立 SSL/TLS 连接。通过这些类，开发者就可以忽略复杂的协议建立流程，较为简单地在网络上建成安全的通讯通道。JSSE 包中主要包括以下一些部分：

* 安全套接字（secure socket）和安全服务器端套接字
* 非阻塞式 SSL/TLS 数据处理引擎（SSLEngine）
* 套接字创建工厂 , 用来产生 SSL 套接字和服务器端套接字
* 套接字上下文 , 用来保存用于创建和数据引擎处理过程中的信息
* 符合 X.509 规范密码匙和安全管理接口

下面将通过一个简单的例子来展示如何通过 JSSE，在客户端和服务器端建立一个 SSL/TLS 连接。设计两个类 SSLClient 和 SSLServer，分别来表示客户端和服务器端。客户端将会向服务器端发起连接请求，在通过服务器端验证建立 SSL 连接后，服务器端将会向客户端发送一串内容，客户端将会把收到的内容打印出来。样例代码如下，

SSLClient Source code：

package example.ssl.codes;

import java.io.\*;

import javax.net.ssl.SSLSocket;

import javax.net.ssl.SSLSocketFactory;

class SSLClient {

private SSLSocket socket = null;

public SSLClient() throws IOException {

// 通过套接字工厂，获取一个客户端套接字

SSLSocketFactory socketFactory = (SSLSocketFactory)

SSLSocketFactory.getDefault();

socket = (SSLSocket) socketFactory.createSocket("localhost", 7070);

}

public void connect() {

try {

// 获取客户端套接字输出流

PrintWriter output = new PrintWriter(

new OutputStreamWriter(socket.getOutputStream()));

// 将用户名和密码通过输出流发送到服务器端

String userName = "principal";

output.println(userName);

String password = "credential";

output.println(password);

output.flush();

// 获取客户端套接字输入流

BufferedReader input = new BufferedReader(

new InputStreamReader(socket.getInputStream()));

// 从输入流中读取服务器端传送的数据内容，并打印出来

String response = input.readLine();

response += "\n " + input.readLine();

System.out.println(response);

// 关闭流资源和套接字资源

output.close();

input.close();

socket.close();

} catch (IOException ioException) {

ioException.printStackTrace();

} finally {

System.exit(0);

}

}

public static void main(String args[]) throws IOException {

new SSLClient().connect();

}

}

SSLServer Source code：

package example.ssl.codes;

import java.io.\*;

import javax.net.ssl.SSLServerSocket;

import javax.net.ssl.SSLServerSocketFactory;

import javax.net.ssl.SSLSocket;

class SSLServer {

// 服务器端授权的用户名和密码

private static final String USER\_NAME = "principal";

private static final String PASSWORD = "credential";

// 服务器端保密内容

private static final String SECRET\_CONTENT =

"This is confidential content from server X, for your eye!";

private SSLServerSocket serverSocket = null;

public SSLServer() throws Exception {

// 通过套接字工厂，获取一个服务器端套接字

SSLServerSocketFactory socketFactory = (SSLServerSocketFactory)

SSLServerSocketFactory.getDefault();

serverSocket = (SSLServerSocket)socketFactory.createServerSocket(7070);

}

private void runServer() {

while (true) {

try {

System.out.println("Waiting for connection...");

// 服务器端套接字进入阻塞状态，等待来自客户端的连接请求

SSLSocket socket = (SSLSocket) serverSocket.accept();

// 获取服务器端套接字输入流

BufferedReader input = new BufferedReader(

new InputStreamReader(socket.getInputStream()));

// 从输入流中读取客户端用户名和密码

String userName = input.readLine();

String password = input.readLine();

// 获取服务器端套接字输出流

PrintWriter output = new PrintWriter(

new OutputStreamWriter(socket.getOutputStream()));

// 对请求进行认证，如果通过则将保密内容发送给客户端

if (userName.equals(USER\_NAME) && password.equals(PASSWORD)) {

output.println("Welcome, " + userName);

output.println(SECRET\_CONTENT);

} else {

output.println("Authentication failed, you have no

access to server X...");

}

// 关闭流资源和套接字资源

output.close();

input.close();

socket.close();

} catch (IOException ioException) {

ioException.printStackTrace();

}

}

}

public static void main(String args[]) throws Exception {

SSLServer server = new SSLServer();

server.runServer();

}

}

SSL 样例程序：

java -cp ./build/classes example.ssl.codes.SSLServer

java -cp ./build/classes example.ssl.codes.SSLClient

执行结果如下：

服务器端输出：

Waiting for connection...

javax.net.ssl.SSLHandshakeException: no cipher suites in common

Waiting for connection...

at sun.security.ssl.Alerts.getSSLException(Alerts.java:192)

at sun.security.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1836)

at sun.security.ssl.Handshaker.fatalSE(Handshaker.java:276)

at sun.security.ssl.Handshaker.fatalSE(Handshaker.java:266)

客户端输出：

javax.net.ssl.SSLException: Connection has been shutdown:

javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake\_failure

at sun.security.ssl.SSLSocketImpl.checkEOF(SSLSocketImpl.java:1426)

at sun.security.ssl.AppInputStream.read(AppInputStream.java:92)

at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:283)

at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:325)

at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:177)

at java.io.InputStreamReader.read(InputStreamReader.java:184)

at java.io.BufferedReader.fill(BufferedReader.java:154)

at java.io.BufferedReader.readLine(BufferedReader.java:317)

at java.io.BufferedReader.readLine(BufferedReader.java:382)

at example.ssl.codes.SSLClient.connect(SSLClient.java:29)

at example.ssl.codes.SSLClient.main(SSLClient.java:44)

Caused by: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake\_failure

at sun.security.ssl.Alerts.getSSLException(Alerts.java:192)

at sun.security.ssl.Alerts.getSSLException(Alerts.java:154)

at sun.security.ssl.SSLSocketImpl.recvAlert(SSLSocketImpl.java:1911)

at sun.security.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:1027)

at sun.security.ssl.SSLSocketImpl.performInitialHandshake

(SSLSocketImpl.java:1262)

at sun.security.ssl.SSLSocketImpl.writeRecord(SSLSocketImpl.java:680)

at sun.security.ssl.AppOutputStream.write(AppOutputStream.java:85)

at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:221)

at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:291)

at sun.nio.cs.StreamEncoder.implFlush(StreamEncoder.java:295)

at sun.nio.cs.StreamEncoder.flush(StreamEncoder.java:141)

at java.io.OutputStreamWriter.flush(OutputStreamWriter.java:229)

at java.io.PrintWriter.flush(PrintWriter.java:320)

at example.ssl.codes.SSLClient.connect(SSLClient.java:25)

... 1 more

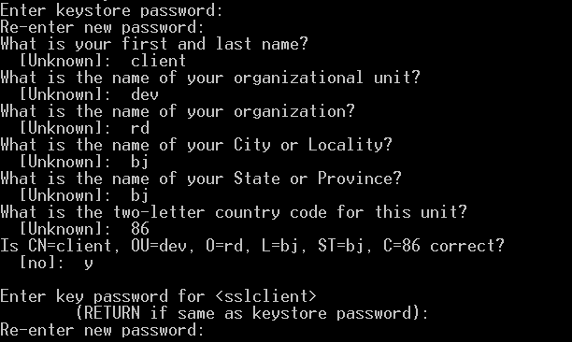
通过程序的错误输出，我们能够发现 SSL 建立失败了，在握手阶段双方没有能够协商出加密方法等信息。这是因为默认情况下，java 虚拟机没有与 SSL 相关的配置，需要开发者自己按照文档进行一些配置。在 JDK 中提供了一个安全钥匙与证书的管理工具 Keytool。Keytool 把钥匙，证书以及和与它们相关联的证书链储存到一个 keystore 中，默任的实现 keystore 的是一个文件，它本身有一个访问密码来保护存储在其中的内容。就本样例程序而言，只需要配置客户端和服务器端双方信任就可以了。可以按照如下几步来完成：

1. 进入本地的 java 安装位置的 bin 目录中 cd /java/bin

2. 创建一个客户端 keystore 文件，如图 2 所示

keytool -genkey -alias sslclient -keystore sslclientkeys

##### 图 2. 创建 keystore 文件



3. 将客户端 keystore 文件导出成证书格式

keytool -export -alias sslclient -keystore sslclientkeys -file sslclient.cer

4. 创建一个服务器端 keystore 文件

keytool -genkey -alias sslserver -keystore sslserverkeys

5. 将服务器端 keystore 文件导出成证书格式

keytool -export -alias sslserver -keystore sslserverkeys -file sslserver.cer

6. 将客户端证书导入到服务器端受信任的 keystore 中

keytool -import -alias sslclient -keystore sslservertrust -file sslclient.cer

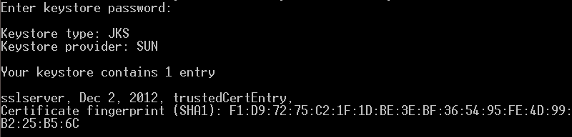
7. 将服务器端证书导入到客户端受信任的 keystore 中

keytool -import -alias sslserver -keystore sslclienttrust -file sslserver.cer

以上所有步骤都完成后，还可以通过命令来查看 keystore 文件基本信息，如图 3 所示

keytool -list -keystore sslclienttrust

##### 图 3. 查看 keystore 文件



将前面创建的所有 keystore 文件从 java 的 bin 目录中剪切出来，移动到样例程序的执行目录中，通过运行程序时候的系统属性来指定这些文件，重新执行一遍样例程序。

java -cp ./build/classes

-Djavax.net.ssl.keyStore=sslserverkeys

-Djavax.net.ssl.keyStorePassword=123456

-Djavax.net.ssl.trustStore=sslservertrust

-Djavax.net.ssl.trustStorePassword=123456

example.ssl.codes.SSLServer

java -cp ./build/classes

-Djavax.net.ssl.keyStore=sslclientkeys

-Djavax.net.ssl.keyStorePassword=123456

-Djavax.net.ssl.trustStore=sslclienttrust

-Djavax.net.ssl.trustStorePassword=123456

example.ssl.codes.SSLClient

执行结果如下：

客户端输出

Welcome, principal

This is confidential content from server X, for your eye!

客户端与服务器端成功建立起 SSL 的连接，然后服务器端成功将字符串发送给客户端，客户端将其打印出来。