深入浅出HTTPS工作原理

深入浅出HTTPS工作原理

HTTP协议由于是明文传送，所以存在三大风险：

1、被窃听的风险：第三方可以截获并查看你的内容

2、被篡改的危险：第三方可以截获并修改你的内容

3、被冒充的风险：第三方可以伪装成通信方与你通信

HTTP因为存在以上三大安全风险，所以才有了HTTPS的出现。

HTTPS涉及到了很多概念，比如SSL/TLS，数字证书、数字签名、加密、认证、公钥和私钥等，比较容易混淆。我们先从一次简单的安全通信故事讲起吧，其中穿插复习一些密码学的概念。

一. 关于Bob与他好朋友通信的故事

这个故事的原文是：

http://www.youdzone.com/signature.html

阮一峰老师也翻译过：

http://www.ruanyifeng.com/blog/2011/08/what_is_a_digital_signature.html

（不过阮老师里面没有很好的区分加密和认证的概念，以及最后HTTPS的说明不够严谨，评论区的针对这些问题的讨论比较激烈，挺有意思的）

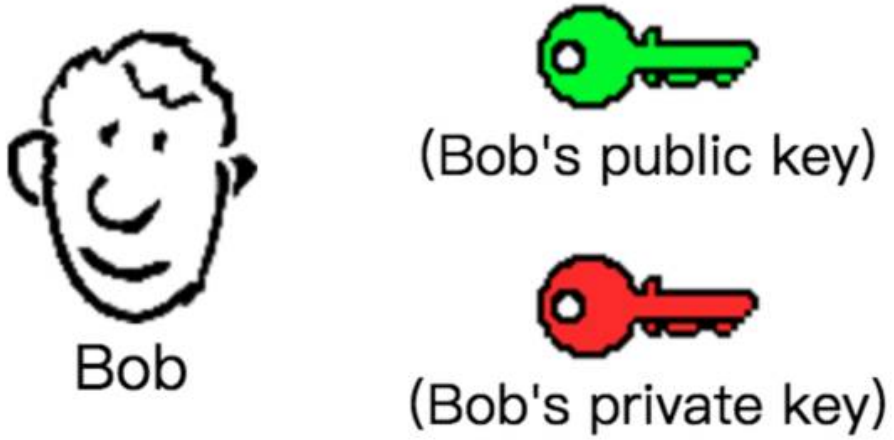
这里重新叙述一下这个故事：

故事的主人公是Bob，他有三个好朋友Pat、Doug和Susan。Bob经常跟他们写信，因为他的信是明文传输的，在传递过程可能被人截获偷窥，也可能被人截获然后又篡改了，更有可能别人伪装成Bob本人跟他的好朋友通信，总之是不安全的。他很苦恼，经过一番苦苦探索，诶，他发现计算机安全学里有一种叫非对称加密算法的东东，好像可以帮助他解决这个问题

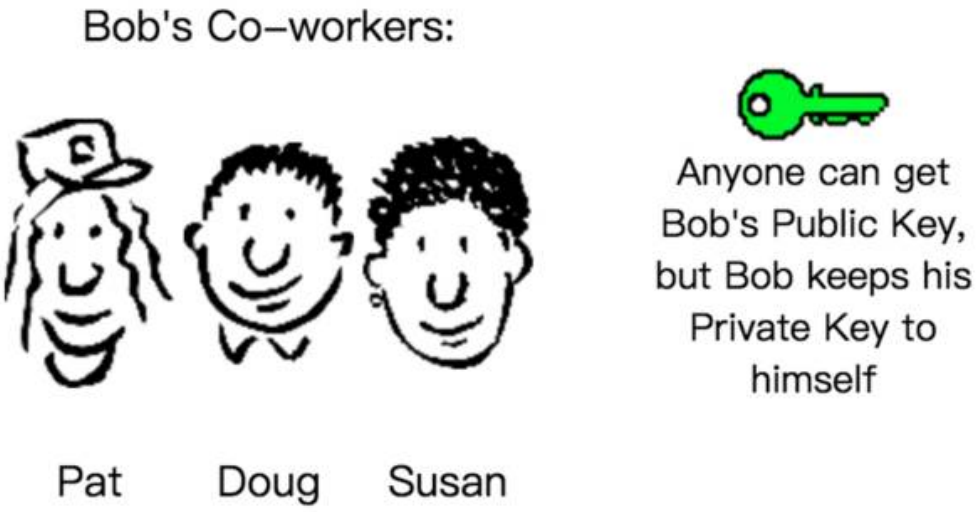
说明：非对称加密算法（RSA）是内容加密的一类算法，它有两个秘钥：公钥与私钥。公钥是公开的钥匙，所有人都可以知道，私钥是保密的，只有持有者知道。通过公钥加密的内容，只能通过私钥解开。非对称加密算法的安全性很高，但是因为计算量庞大，比较消耗性能。

好了，来看看Bob是怎么应用非对称加密算法与他的好朋友通信的：

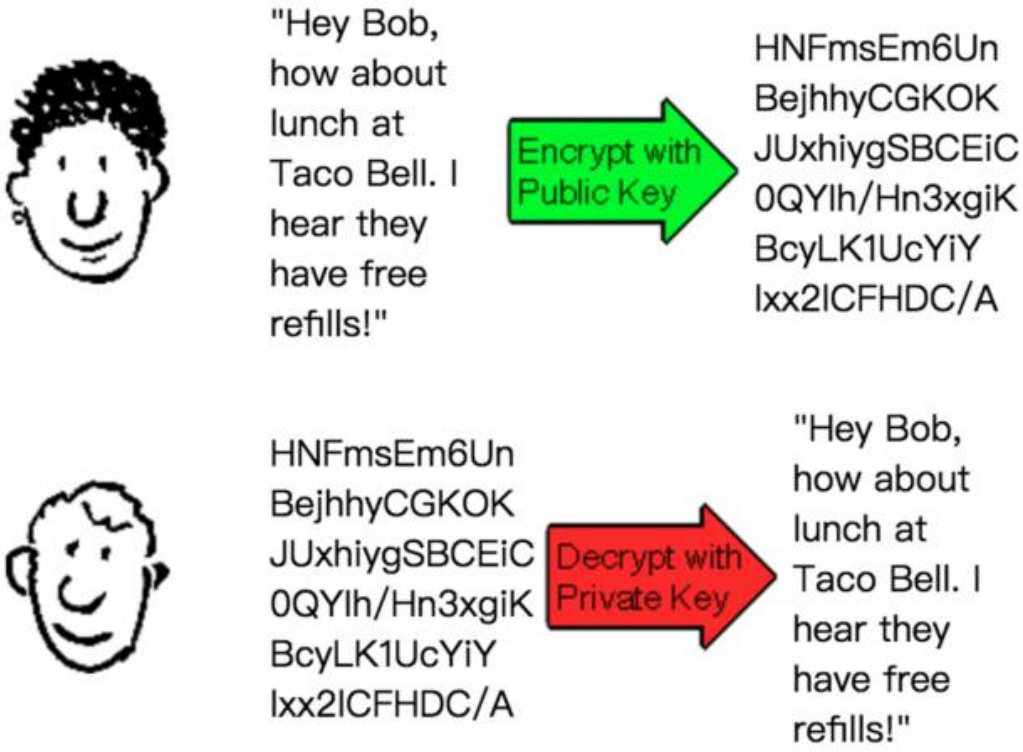
1、首先Bob弄到了两把钥匙：公钥和私钥。



2、Bob自己保留下了私钥，把公钥复制成三份送给了他的三个好朋友Pat、Doug和Susan。

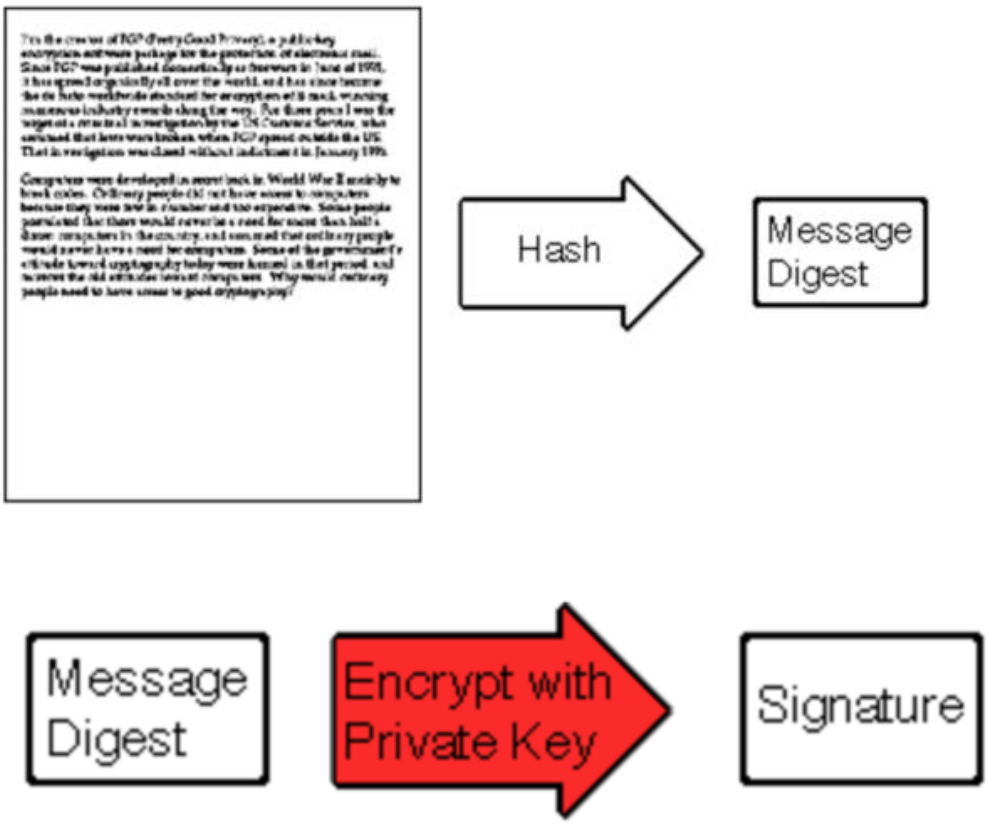


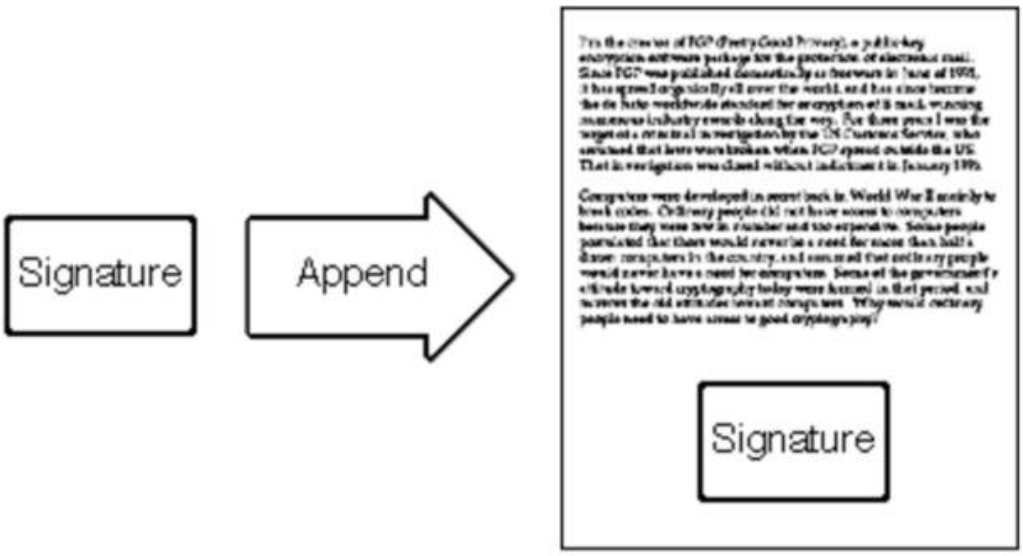
3、此时，Bob总算可以安心地和他的好朋友愉快地通信了。比如Susan要和他讨论关于去哪吃午饭的事情，Susan就可以先把自己的内容（明文）首先用Bob送给他的公钥做一次加密，然后把加密的内容传送给Bob。Bob收到信后，再用自己的私钥解开信的内容。



说明：这其实是计算机安全学里**加密**的概念，加密的目的是为了不让别人看到传送的内容，加密的手段是通过一定的加密算法及约定的密钥进行的（比如上述用了非对称加密算法以及Bob的公钥），而解密则需要相关的解密算法及约定的秘钥（如上述用了非对称加密算法和Bob自己的私钥），可以看出加密是可逆的（可解密的）。

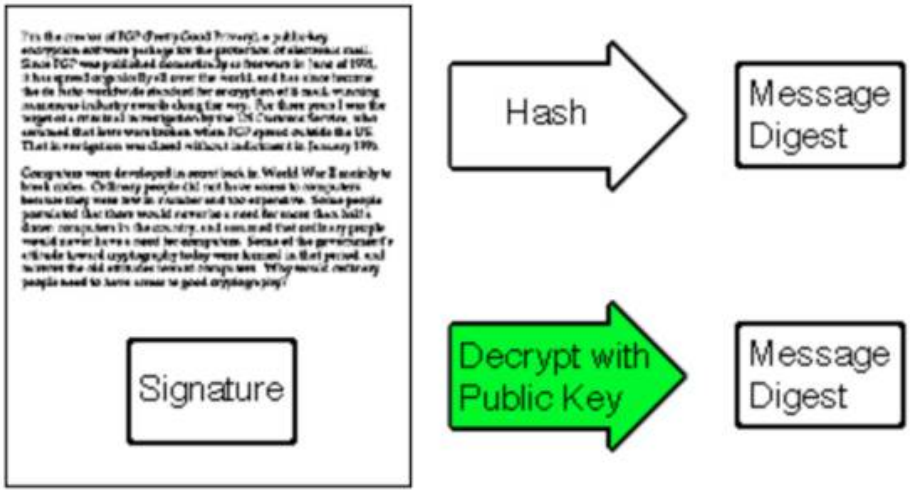
4、Bob看完信后，决定给Susan回一封信。为了防止信的内容被篡改（或者别人伪装成他的身份跟Susan通信），他决定先对信的内容用hash算法做一次处理，得到一个字符串哈希值，Bob又用自己的私钥对哈希值做了一次加密得到一个签名，然后把签名和信（明文的）一起发送给Susan。





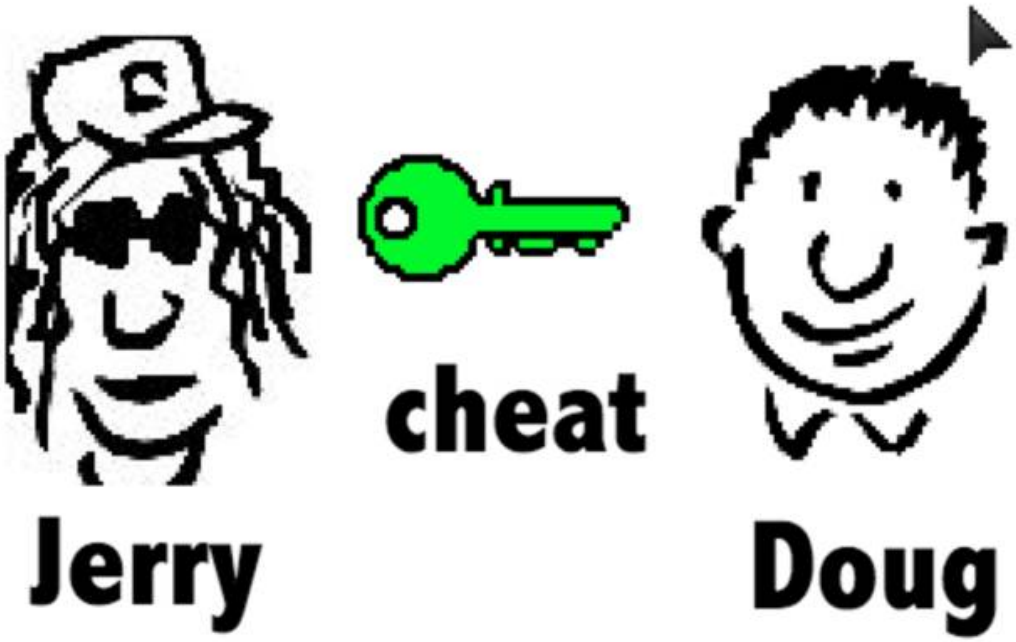
说明：Bob的内容实质是明文传输的，所以这个过程是可以被人截获和窥探的，但是Bob不担心被人窥探，他担心的是内容被人篡改或者有人冒充自己跟Susan通信。这里其实涉及到了计算机安全学中的**认证**概念，Bob要向Susan证明通信的对方是Bob本人，另外也需要确保自己的内容是完整的。

5、Susan接收到了Bob的信，首先用Bob给的公钥对签名作了解密处理，得到了哈希值A，然后Susan用了同样的Hash算法对信的内容作了一次哈希处理，得到另外一个哈希值B，对比A和B，如果这两个值是相同的，那么可以确认信就是Bob本人写的，并且内容没有被篡改过。

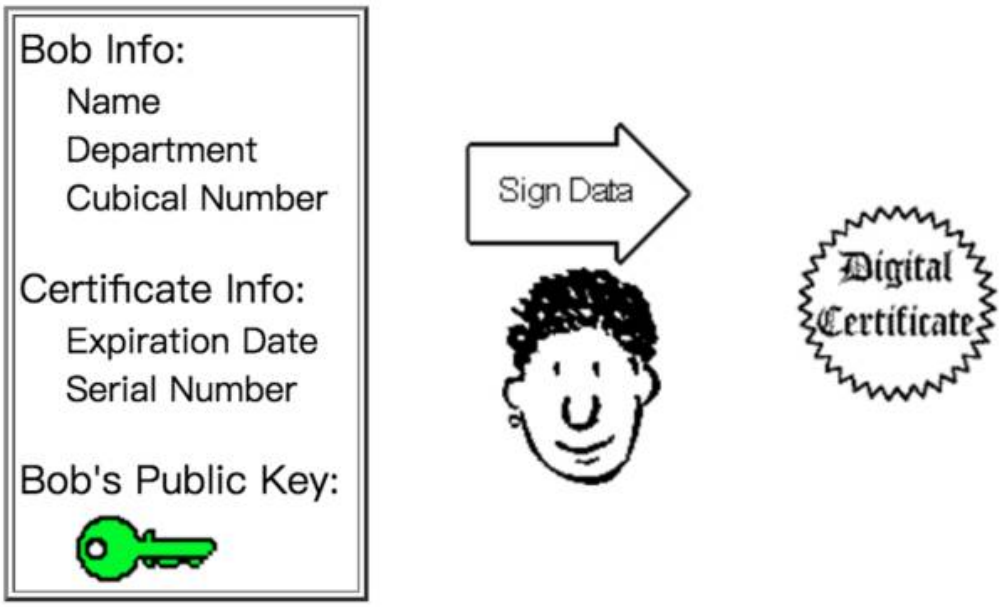


说明：4跟5其实构成了一次完整的通过**数字签名**进行认证的过程。数字签名的过程简述为：发送方通过不可逆算法对内容text1进行处理（哈希），得到的结果值hash1，然后用私钥加密hash1得到结果值encry1。对方接收text1和encry1，用公钥解密encry1得到hash1，然后用text1进行同等的不可逆处理得到hash2，对hash1和hash2进行对比即可认证发送方。

6、此时，另外一种比较复杂出现了，Bob是通过网络把公钥寄送给他的三个好朋友的，有一个不怀好意的家伙Jerry截获了Bob给Doug的公钥。Jerry开始伪装成Bob跟Doug通信，Doug感觉通信的对象不像是Bob，但是他又无法确认。



7、Bob最终发现了自己的公钥被Jerry截获了，他感觉自己的公钥通过网络传输给自己的小伙伴似乎也是不安全的，不怀好意的家伙可以截获这个明文传输的公钥。为此他想到了去第三方权威机构“证书中心”（certificate authority，简称CA）做认证。证书中心用自己的私钥对Bob的公钥和其它信息做了一次加密。这样Bob通过网络将数字证书传递给他的小伙伴后，小伙伴们先用CA给的公钥解密证书，这样就可以安全获取Bob的公钥了。



二、HTTPS通信过程

通过Bob与他的小伙伴的通信，我们已经可以大致了解一个安全通信的过程，也可以了解基本的加密、解密、认证等概念。HTTPS就是基于这样一个逻辑设计的。

首先看看组成HTTPS的协议：HTTP协议和SSL/TLS协议。HTTP协议就不用讲了，而SSL/TLS就是负责加密解密等安全处理的模块，所以HTTPS的核心在SSL/TLS上面。整个通信如下：

1、浏览器发起往服务器的443端口发起请求，请求携带了浏览器支持的加密算法和哈希算法。

2、服务器收到请求，选择浏览器支持的加密算法和哈希算法。

3、服务器下将数字证书返回给浏览器，这里的数字证书可以是向某个可靠机构申请的，也可以是自制的。

4、浏览器进入数字证书认证环节，这一部分是浏览器内置的TLS完成的：

4.1 首先浏览器会从内置的证书列表中索引，找到服务器下发证书对应的机构，如果没有找到，此时就会提示用户该证书是不是由权威机构颁发，是不可信任的。如果查到了对应的机构，则取出该机构颁发的公钥。

4.2 用机构的证书公钥解密得到证书的内容和证书签名，内容包括网站的网址、网站的公钥、证书的有效期等。浏览器会先验证证书签名的合法性（验证过程类似上面Bob和Susan的通信）。签名通过后，浏览器验证证书记录的网址是否和当前网址是一致的，不一致会提示用户。如果网址一致会检查证书有效期，证书过期了也会提示用户。这些都通过认证时，浏览器就可以安全使用证书中的网站公钥了。

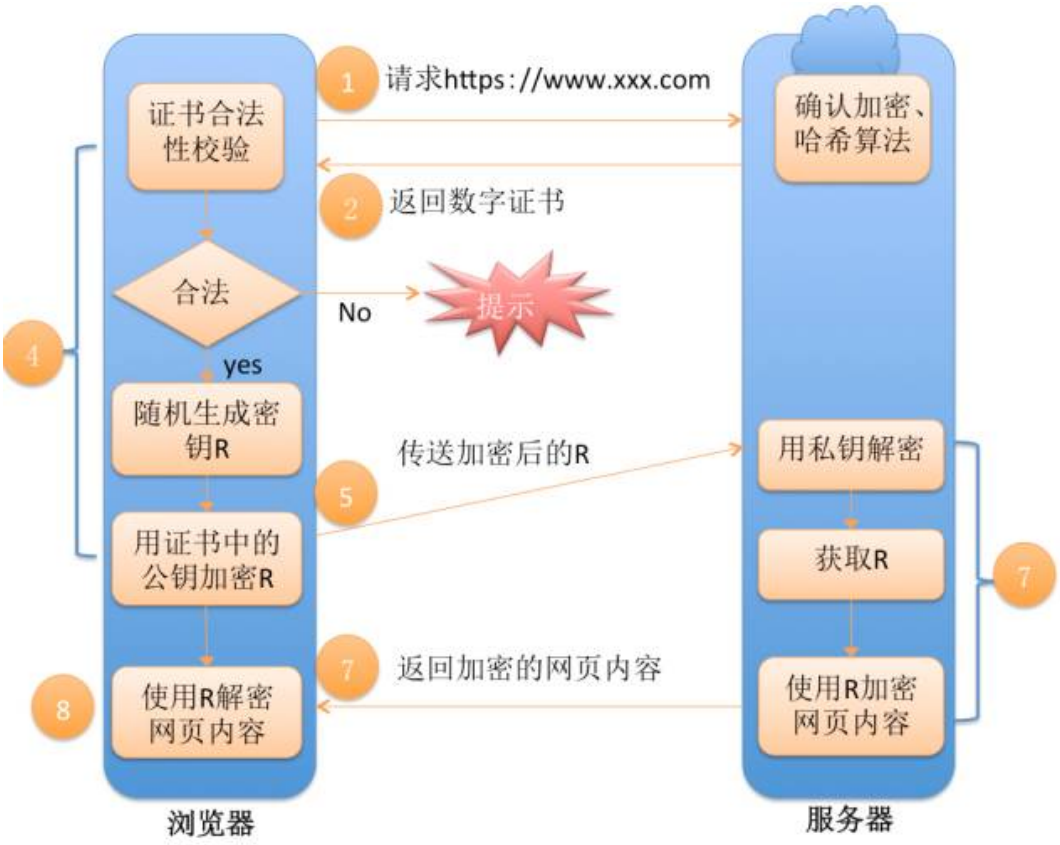
4.3 浏览器生成一个随机数R，并使用网站公钥对R进行加密。

5、浏览器将加密的R传送给服务器。

6、服务器用自己的私钥解密得到R。

7、服务器以R为密钥使用了**对称加密算法**加密网页内容并传输给浏览器。

8、浏览器以R为密钥使用之前约定好的解密算法获取网页内容。



备注1：前5步其实就是HTTPS的握手过程，这个过程主要是认证服务端证书（内置的公钥）的合法性。因为非对称加密计算量较大，**整个通信过程只会用到一次非对称加密算法**（主要是用来保护传输客户端生成的用于对称加密的随机数私钥）。**后续内容的加解密都是通过一开始约定好的对称加密算法进行的。**

备注2：SSL/TLS是HTTPS安全性的核心模块，TLS的前身是SSL，TLS1.0就是SSL3.1，TLS1.1是SSL3.2，TLS1.2则是SSL3.3。 SSL/TLS是建立在TCP协议之上，因而也是应用层级别的协议。其包括TLS Record Protocol和TLS Handshaking Protocols两个模块，后者负责握手过程中的身份认证，前者则保证数据传输过程中的完整性和私密性。

OpenSSL加密实践

对称加密与非对称加密

程序代码中我用的是对称加密。 （加密、解密都是一个密钥）

RPC中buffer格式如下：

```
[text]
<code style="line-height:normal;margin:auto;vertical-align:middle;height:auto;font-family:'Courier New', sans-serif;font-size:12px;border:1px solid rgb(204,204,204);padding-right:5px;padding-left:5px;background-color:rgb(245,245,245);">
[socket bufferLen] + [MD5 result] + {[business bufLen] + [business cmdCode] + [business buf]}
 4Bytes 32Bytes 4Bytes 4Bytes Unknow

- 密钥长度可以自己定义 (如果有性能要求的话，不需要太大，我们定义的是6bytes)
- [MD5 result] 是对 {花括号中的数据 + [密钥]} 进行MD5的值
- 接收方，知道我的密钥，先把{花括号中的数据 + [密钥]}做MD5得到的值，和[MD5 result]做对比，
    * 如果错误则记录 + 抛弃此信息
    * 如果正确，则做正常业务
</code>
```

非对称加密与OpenSSL via pannengzhi

随着个人隐私越来越受重视, HTTPS也渐渐的流行起来, 甚至有许多网站都做到了全站HTTPS, 然而这种加密和信任机制也不断遭遇挑战,比如戴尔根证书携带私钥,Xboxlive证书私钥泻露, 还有前一段时间的沃通错误颁发Github根域名SSL证书事件. 因此本文从非对称加密说起, 介绍了证书的签证流程, 并且通过openssl的命令行工具对这些过程都转化为相对具体的命令, 也算是一个温故知新的简要记录吧.

1.前言

一般来说,常见的数字加密方式都可以分为两类,即对称加密和非对称加密. 对于对称加密来说, 加密和解密用的是同一个密钥, 加密方法有AES,DES,RC4,BlowFish等; 对应的, 非对称加密在加密和解密时, 用的是不同的密钥, 分别称为公钥或私钥. 非对称加密的加密方法有RSA, DSA, Diffie-Hellman等.

OpenSSL是一个开源项目,为传输层安全(TLS)和安全套接字(SSL)协议提供了比较完整的实现, 同时也致力于将自身打造为一个通用的密码学工具集. 其中包括:

- libssl : 提供了SSL(包括SSLv3)和TLS的服务器端以及客户端的实现.
- libcrypto : 通用的密码学库以及对X.509的支持
- openssl : 一个多功能的命令行工具

本文主要使用openssl的命令行工具来示例非对称加密的流程, 如果有兴趣的话,也可以用其SDK 来实现更具体的操作.

2.加解密过程

2.1创建公私钥对

首先用openssl生成私钥:

```
openssl genrsa -out private.pem 1024
```

当然为了更加安全,可以在生成私钥的时候同时指定密码, 这样即使不小心泻露了私钥,也能增加别人的盗用难度:

```
openssl genrsa -aes256 -passout stdin -out private.pem 1024
openssl genrsa -aes256 -passout file:passwd.txt -out private.pem 1024
openssl genrsa -aes256 -passout pass:my_password -out private.pem 1024
```

其中`-passout` 指定密码的输入方式,可以分别是stdin,从文件中读取或者紧接着pass:后面输入.
有了私钥,便可以从其中提取出公钥:

```
openssl rsa -in private.pem -pubout -out public.pem
```

2.2用公私钥进行加解密

在一次秘密的信息传输中, 我们首先通过可信的方式(比如面对面)将公钥告知对方, 对方发送机密信息的时候
就可以用我们的公钥加密:

```
openssl rsautl -encrypt -pubin -inkey public.pem -in file.txt -out file.txt.enc
```

在发送的过程中即便泄露了文件,也无法查看文件的明文信息. 而我们收到密文后, 用私钥解密即可:

```
openssl rsautl -decrypt -inkey private.pem -in file.txt.enc -out file.txt.dec
```

2.2和对称加密协作

虽然公私钥加密很好用, 但事实上非对称加密的缺点是加解密速度要远远慢于对称加密, 在某些极端情况下,
甚至能比非对称加密慢上千倍. 另外由于RSA算法的工作机理, 如果密钥是n比特的,那么其加密的信息容量就
不能大于(n-11)比特. 因此对于大文件的加密传输, 通常还是使用对称加密的方式, 例如

```
openssl rand -base64 128 -out aeskey.txt
openssl enc -aes-256-cbc -salt -in file.txt -out file.txt.aesenc -pass file:aeskey.txt
openssl enc -d -aes-256-cbc -in file.txt.aesenc -out file.txt.aesdec -pass file:aeskey.txt
```

其中aeskey.txt是我们随机生成密码文件, 并且用其可以对大文件进行对称的加解密, 在实际中,
通常还会将密码文件用公私钥加密的方式来发送给对方. 值得一提的是,这也正是PGP的工作方式,

3.证书

对任一个体来说, 它都有公钥,私钥和证书. 其中私钥用来加密发出去的信息,公钥用来解密收到的信息,
而证书则用来证明自己的身份. 一般来说,证书中包含自己的公钥以及额外的信息,如签发机构(CA, Certificate Authority),

证书用途(比如适用的域名)和有效时间等. CA通常是个第三方的可信机构, 比如VeriSign, GeoTrust, DigiCert和沃通等, 当然也可以是未知的主体, 比如说自己.

获得一张证书的流程通常是:

- 1)用私钥生成证书签名请求(csr),
- 2)将csr文件发送给CA,待其验证信息无误后,CA会用自己的私钥对其进行签名表示确认.

3.1 生成证书签名请求

证书签名请求(Certificate Signing Request)通常以.csr为后缀, 包含了请求方的公钥和主体的详细信息, 如域名,公司名,国家,城市等信息, 其完整内容可以参考这里. 使用openssl也能很方便地生成csr:

```
openssl req -new -key private.pem -out pppan.csr
```

默认会在stdin中根据提示交互地输入主体信息,也可以通过`-config`选项来从文件中读取.
生成完之后可以通过:

```
openssl req -in pppan.csr -noout -text
```

来查看csr文件中的详细信息.

3.2 CA对csr文件进行签名

当CA收到csr文件并且对请求方的域名,公司等内容校验无误后,便可以对csr请求进行确认(签名),

```
openssl req -x509 -newkey rsa:4096 -nodes -keyout cakey.pem -out cacert.pem -outform PEM
openssl ca -config openssl-ca.cnf -policy SP -extensions SR -infiles pppan.csr -out pppan.crt
```

虽然这不是重点, 但也稍微解释下这两个命令的意思吧. 第一个命令是CA一开始创建私钥和CA的证书,
第二个命令表示对csr文件进行签名确认, 用-config指定自定义的配置文件, 如果不指定则默认为/usr/lib/ssl/openssl.cnf,
SP和SR都是自定义于配置文件中的信息, 此外配置文件中还包括CA证书路径和私钥路径,以及对req的默认校验策略等, 有兴趣的可以查看详细解释.

另外值得一提的是, 我们用自己的私钥也可以生成证书, 并且也能用这个证书来对自己的csr进行签名,
这通常称为自签名(self-signed), 上面CA生成的证书cacert.pem就是自签名的. 一般来说,
如果是自己随便生成自签名证书, 通常会被认为是不可信的, 除非手动添加到对方的信任CA证书列表中.

3.3查看和验证证书

CA对csr进行签名后, 我们就能得到对应的证书, 这里是pppan.crt, 可以用openssl查看证书的详细信息:

```
openssl x509 -noout -text -in pppan.crt
```

可以看到具体的签发机构,签发时间和证书的有效时间等信息.
可以用命令验证证书是否有效:

```
openssl verify -CAfile Trusts.pem pppan.crt
```

其中Trusts.pem是一系列所信任的证书集合,其中也包括了上述CA的证书cacert.pem

4. 其他

上面所有用到的证书及其组件,如公钥,私钥,csr等,其格式都是PEM的,这也是最常见的一种格式,
可以用文本便及其打开,通常是以-----BEGIN XXX------开头, 以-----END XXX-----结束,
中间的部分则是实际密钥的base64编码, 其二进制表示也称为DER格式, 两者可以用base64转化,
因此都属于x509实现的证书格式.

还有比较常见的证书格式,为PKCS7和PKCS12. 其中PKCS7是由JAVA使用的开放标准,并且也被
Windows所支持, 其内是不包含私钥信息的; 而PKCS12则是一种非公开的标准,用来提供比PEM的
纯文本格式更高的安全性, 这是Windows建议使用的格式, 其中可以包含私钥信息.

不同格式的转换如下所示.

```
PEM <-> DER:
openssl x509 -in bar.pem -outform der -out bar.der
openssl x509 -inform der -in foo.der -out foo.pem
PEM <-> PKCS7:
openssl crl2pkcs7 -nocrl -certfile foocert.pem -out foocert.p7b
openssl pkcs7 -in foocert.p7b -print_certs -out barcert.pem
PEM <-> PKCS12:
openssl pkcs12 -inkey private.key -in foocert.pem -export -out foocert.pfx
openssl pkcs12 -in foocert.pfx -nodes -out barcert.pem
```

5. 后记

当今我们使用最多的https本质上就是在http协议的基础上对传输内容进行了非对称的加密,
当然实现过程多了很多复杂的交互, 感兴趣的可以去查看SSL和TLS协议. 我想说的是,
这一切信任机制的基石是对于CA的信任, 如果说CA的私钥泻露,或者我们错误地信任了一个坏CA,
那么https的隐私性也就不复存在了, 因为其可能对无效的csr进行签名, 从而使得https中间人攻击
成为现实. 据说早在两年前伟大的防火墙就已经可以对https进行监听,敏感词识别和连接重置,
后来因为某种原因才从大范围应用转为只对特殊对象使用,不过那是后话了.

博客地址:

http://pppan.net
有价值炮灰-博客园
欢迎交流,文章转载请注明出处.

常用Java加密算法总结

简单的java加密算法有：

1. BASE64 严格地说，属于编码格式，而非加密算法
2. MD5(Message Digest algorithm 5，信息摘要算法)
3. SHA(Secure Hash Algorithm，安全散列算法)
4. HMAC(Hash Message Authentication Code，散列消息鉴别码)

1. BASE64

Base64是网络上最常见的用于传输8Bit字节代码的编码方式之一，大家可以查看RFC2045～RFC2049，上面有MIME的详细规范。Base64编码可用于在HTTP环境下传递较长的标识信息。例如，在Java Persistence系统Hibernate中，就采用了Base64来将一个较长的唯一标识符（一般为128-bit的UUID）编码为一个字符串，用作HTTP表单和HTTP GET URL中的参数。在其他应用程序中，也常常需要把二进制数据编码为适合放在URL（包括隐藏表单域）中的形式。此时，采用Base64编码具有不可读性，即所编码的数据不会被人用肉眼所直接看到。（来源百度百科）

java实现代码：

```
package com.cn.单向加密;

import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;
/*
BASE64的加密解密是双向的，可以求反解.
BASE64Encoder和BASE64Decoder是非官方JDK实现类。虽然可以在JDK里能找到并使用，但是在API里查不到。
JRE 中 sun 和 com.sun 开头包的类都是未被文档化的，他们属于 java, javax 类库的基础，其中的实现大多数与底层平台有关，
一般来说是不推荐使用的。
BASE64 严格地说，属于编码格式，而非加密算法
主要就是BASE64Encoder、BASE64Decoder两个类，我们只需要知道使用对应的方法即可。
另，BASE加密后产生的字节位数是8的倍数，如果不够位数以=符号填充。
BASE64
按照RFC2045的定义，Base64被定义为：Base64内容传送编码被设计用来把任意序列的8位字节描述为一种不易被人直接识别的形式。
（The Base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable.）
常见于邮件、http加密，截取http信息，你就会发现登录操作的用户名、密码字段通过BASE64加密的。
*/

public class BASE64 {
    /**
     * BASE64解密
     *
     * @param key
     * @return
     * @throws Exception
     */
    public static byte[] decryptBASE64(String key) throws Exception {
        return (new BASE64Decoder()).decodeBuffer(key);
    }

    /**
     * BASE64加密
     *
     * @param key
     * @return
     * @throws Exception
     */
    public static String encryptBASE64(byte[] key) throws Exception {
        return (new BASE64Encoder()).encodeBuffer(key);
    }
```

```
    public static void main(String[] args) {

     String  str="12345678";

        try {
        String  result1= BASE64.encryptBASE64(str.getBytes());
         System.out.println("result1=====加密数据==========="+result1);

         byte  result2[]= BASE64.decryptBASE64(result1);
         String  str2=new String(result2);
         System.out.println("str2========解密数据========"+str2);
    } catch (Exception e) {
        e.printStackTrace();
    }

    }

}
```

2. MD5

MD5即Message-Digest Algorithm 5（信息-摘要算法5），用于确保信息传输完整一致。是计算机广泛使用的杂凑算法之一（又译摘要算法、哈希算法），主流编程语言普遍已有MD5实现。将数据（如汉字）运算为另一固定长度值，是杂凑算法的基础原理，MD5的前身有MD2、MD3和MD4。广泛用于加密和解密技术，常用于文件校验。校验？不管文件多大，经过MD5后都能生成唯一的MD5值。好比现在的ISO校验，都是MD5校验。怎么用？当然是把ISO经过MD5后产生MD5的值。一般下载linux-ISO的朋友都见过下载链接旁边放着MD5的串。就是用来验证文件是否一致的。

java实现代码：

```
package com.cn.单向加密;

import java.math.BigInteger;
import java.security.MessageDigest;
/*
MD5(Message Digest algorithm 5，信息摘要算法)
通常我们不直接使用上述MD5加密。通常将MD5产生的字节数组交给BASE64再加密一把，得到相应的字符串
Digest:汇编
*/
public class MD5 {
    public static final String KEY_MD5 = "MD5";

    public static  String  getResult(String inputStr)
    {
        System.out.println("=======加密前的数据:"+inputStr);
        BigInteger bigInteger=null;

        try {
         MessageDigest md = MessageDigest.getInstance(KEY_MD5);
         byte[] inputData = inputStr.getBytes();
         md.update(inputData);
         bigInteger = new BigInteger(md.digest());
```

```
        } catch (Exception e) {e.printStackTrace();}
        System.out.println("MD5加密后:" + bigInteger.toString(16));
        return bigInteger.toString(16);
    }

    public static void main(String args[])
    {
        try {
            String inputStr = "简单加密8888888888888888888";
            getResult(inputStr);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

MD5算法具有以下特点：

1、压缩性：任意长度的数据，算出的MD5值长度都是固定的。

2、容易计算：从原数据计算出MD5值很容易。

3、抗修改性：对原数据进行任何改动，哪怕只修改1个字节，所得到的MD5值都有很大区别。

4、弱抗碰撞：已知原数据和其MD5值，想找到一个具有相同MD5值的数据（即伪造数据）是非常困难的。

5、强抗碰撞：想找到两个不同的数据，使它们具有相同的MD5值，是非常困难的。

MD5的作用是让大容量信息在用数字签名软件签署私人密钥前被"压缩"成一种保密的格式（就是把一个任意长度的字节串变换成一定长的十六进制数字串）。除了MD5以外，其中比较有名的还有sha-1、RIPEMD以及Haval等。

3.SHA

安全哈希算法（Secure Hash Algorithm）主要适用于数字签名标准（Digital Signature Standard DSS）里面定义的数字签名算法（Digital Signature Algorithm DSA）。对于长度小于2^64位的消息，SHA1会产生一个160位的消息摘要。该算法经过加密专家多年来的发展和改进已日益完善，并被广泛使用。该算法的思想是接收一段明文，然后以一种不可逆的方式将它转换成一段（通常更小）密文，也可以简单的理解为取一串输入码（称为预映射或信息），并把它们转化为长度较短、位数固定的输出序列即散列值（也称为信息摘要或信息认证代码）的过程。散列函数值可以说是对明文的一种"指纹"或是"摘要"所以对散列值的数字签名就可以视为对此明文的数字签名。

java实现代码：

```
package com.cn.单向加密;

import java.math.BigInteger;
import java.security.MessageDigest;

/*
SHA(Secure Hash Algorithm，安全散列算法），数字签名等密码学应用中重要的工具，
被广泛地应用于电子商务等信息安全领域。虽然，SHA与MD5通过碰撞法都被破解了，
但是SHA仍然是公认的安全加密算法，较之MD5更为安全*/
public class SHA {
    public static final String KEY_SHA = "SHA";

    public static  String  getResult(String inputStr)
    {
```

```
        BigInteger sha =null;
        System.out.println("=======加密前的数据:"+inputStr);
        byte[] inputData = inputStr.getBytes();
        try {
            MessageDigest messageDigest = MessageDigest.getInstance(KEY_SHA);
            messageDigest.update(inputData);
            sha = new BigInteger(messageDigest.digest());
            System.out.println("SHA加密后:" + sha.toString(32));
        } catch (Exception e) {e.printStackTrace();}
        return sha.toString(32);
    }

    public static void main(String args[])
    {
        try {
            String inputStr = "简单加密";
            getResult(inputStr);
        } catch (Exception e) {
           e.printStackTrace();
        }

    }

}
```

SHA-1与MD5的比较

因为二者均由MD4导出，SHA-1和MD5彼此很相似。相应的，他们的强度和其他特性也是相似，但还有以下几点不同：

l 对强行攻击的安全性：最显著和最重要的区别是SHA-1摘要比MD5摘要长32 位。使用强行技术，产生任何一个报文使其摘要等于给定报摘要的难度对MD5是2^128数量级的操作，而对SHA-1则是2^160数量级的操作。这样，SHA-1对强行攻击有更大的强度。

l 对密码分析的安全性：由于MD5的设计，易受密码分析的攻击，SHA-1显得不易受这样的攻击。

l 速度：在相同的硬件上，SHA-1的运行速度比MD5慢。

4.HMAC

HMAC(Hash Message Authentication Code，散列消息鉴别码，基于密钥的Hash算法的认证协议。消息鉴别码实现鉴别的原理是，用公开函数和密钥产生一个固定长度的值作为认证标识，用这个标识鉴别消息的完整性。使用一个密钥生成一个固定大小的小数据块，即MAC，并将其加入到消息中，然后传输。接收方利用与发送方共享的密钥进行鉴别认证等。

java实现代码：

```
package com.cn.单向加密;
/*
HMAC
HMAC(Hash Message Authentication Code，散列消息鉴别码，基于密钥的Hash算法的认证协议。
消息鉴别码实现鉴别的原理是，用公开函数和密钥产生一个固定长度的值作为认证标识，用这个标识鉴别消息的完整性。
使用一个密钥生成一个固定大小的小数据块，
即MAC，并将其加入到消息中，然后传输。接收方利用与发送方共享的密钥进行鉴别认证等。*/
import javax.crypto.KeyGenerator;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
```

```java
import javax.crypto.spec.SecretKeySpec;

import com.cn.comm.Tools;

/**
 * 基础加密组件
 */
public abstract class HMAC {
    public static final String KEY_MAC = "HmacMD5";

    /**
     * 初始化HMAC密钥
     *
     * @return
     * @throws Exception
     */
    public static String initMacKey() throws Exception {
        KeyGenerator keyGenerator = KeyGenerator.getInstance(KEY_MAC);
        SecretKey secretKey = keyGenerator.generateKey();
        return BASE64.encryptBASE64(secretKey.getEncoded());
    }

    /**
     * HMAC加密 ： 主要方法
     *
     * @param data
     * @param key
     * @return
     * @throws Exception
     */
    public static String encryptHMAC(byte[] data, String key) throws Exception {

        SecretKey secretKey = new SecretKeySpec(BASE64.decryptBASE64(key), KEY_MAC);
        Mac mac = Mac.getInstance(secretKey.getAlgorithm());
        mac.init(secretKey);
        return new String(mac.doFinal(data));

    }

    public static  String  getResult1(String inputStr)
    {
        String path=Tools.getClassPath();
        String fileSource=path+"/file/HMAC_key.txt";
        System.out.println("=======加密前的数据:"+inputStr);
        String  result=null;
        try {
            byte[] inputData = inputStr.getBytes();
            String key = HMAC.initMacKey(); /*产生密钥*/
            System.out.println("Mac密钥:===" + key);
            /*将密钥写文件*/
            Tools.WriteMyFile(fileSource,key);
            result= HMAC.encryptHMAC(inputData, key);
            System.out.println("HMAC加密后:===" + result);
        } catch (Exception e) {e.printStackTrace();}
```

```
        return result.toString();
    }

    public static  String  getResult2(String inputStr)
    {
       System.out.println("=======加密前的数据:"+inputStr);
        String path=Tools.getClassPath();
        String fileSource=path+"/file/HMAC_key.txt";
        String key=null;;
       try {
           /*将密钥从文件中读取*/
           key=Tools.ReadMyFile(fileSource);
           System.out.println("getResult2密钥:===" + key);
       } catch (Exception e1) {
          e1.printStackTrace();}
       String  result=null;
       try {
          byte[] inputData = inputStr.getBytes();
          /*对数据进行加密*/
          result= HMAC.encryptHMAC(inputData, key);
          System.out.println("HMAC加密后:===" + result);
       } catch (Exception e) {e.printStackTrace();}
      return result.toString();
    }

    public static void main(String args[])
    {
       try {
           String inputStr = "简单加密";
           /*使用同一密钥：对数据进行加密：查看两次加密的结果是否一样*/
           getResult1(inputStr);
           getResult2(inputStr);

       } catch (Exception e) {
          e.printStackTrace();
       }

    }

}
```