1.你在互联网上的数据还在裸奔吗?

这是今年三月份有关移动市场的统计数据,移动app的数量已经突破10亿。移动安全也成为了一个全民关注的问题。从最初的app只针对功能实现，爆出来了一系列的高危漏洞之后，应运而生了包括移动app检测、app加固保护等工作来保护开发者以及使用者权益。同时，http的明文数据传输问题也得到了有效解决。我们本篇文章的讨论内容还是从数据传输过程中所引发的一系列安全问题。

1.1 数据裸奔时代

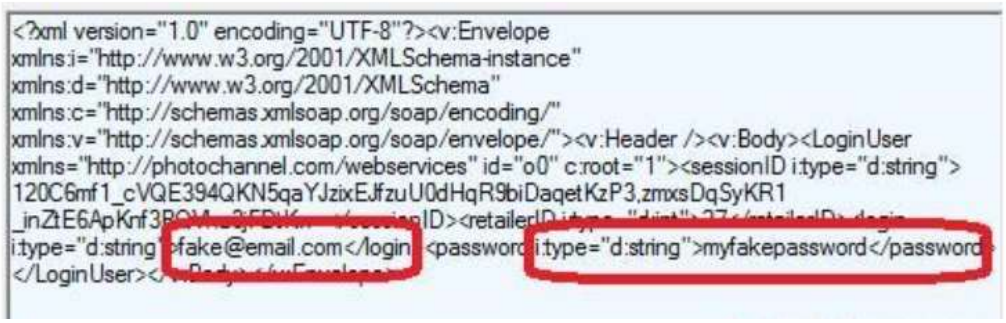
使用http协议的数据传输方式

- HyperText Transfer Protocol，超文本传输协议，是互联网上使用最广泛的一种协议，所有WWW文件必须遵循的标准。HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全。 使用TCP端口为：80

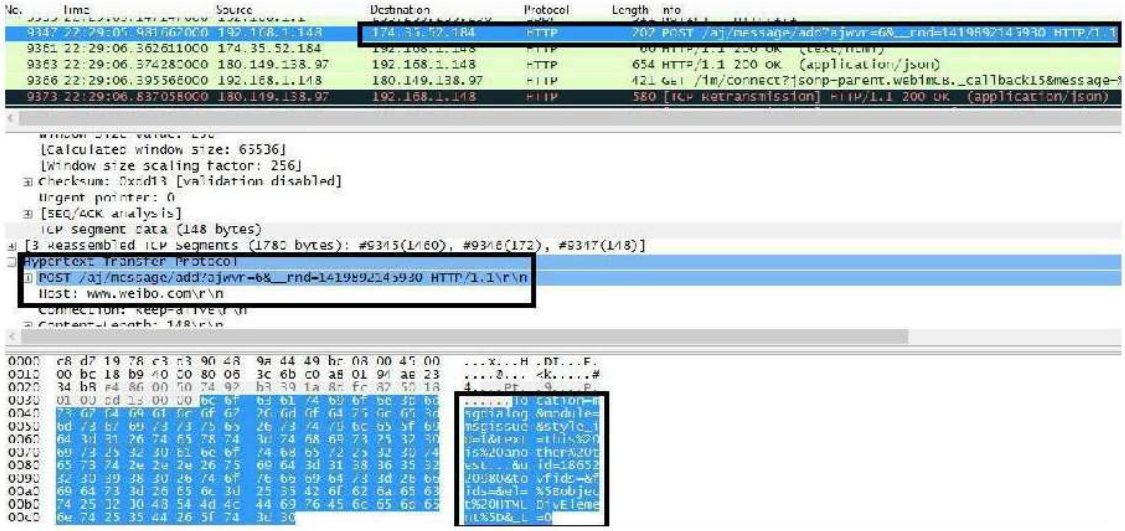
最初的移动app开发过程中，使用的大部分http协议来进行客户端跟服务端的通信。这个过程中传输的信息都是明文，继而引发了一系列的信息泄露等漏洞

Costco应用：赤裸裸的凭证

美国第二大零售商Costco应用程序就存在这个漏洞，安全研究人员测试发现其登录请求为明文HTTP请求。这说明了什么?当你使用手机连接到存在危险的公共无线网络进行网上购物时，黑客将会截取这些信息。



wireshark简单捕获就能看到明文隐私数据



当然上述极为不安全的数据传输，在2015年被大量爆出来之后，立即引起了app的开发人员以及使用着的重视。后续的数据传输使用了相对安全的基于SSL/TLS加密的安全的超文本传输协议https。

2.你所使用的加密数据传输真的有保证你的数据不被窃取吗?

2.1 https加密传输

- Hyper Text Transfer Protocol over Secure Socket Layer，安全的超文本传输协议，网景公式设计了SSL(Secure Sockets Layer)协议用于对Http协议传输的数据进行加密，保证会话过程中的安全性。 使用TCP端口默认为443
- SSL协议即用到了对称加密也用到了非对称加密(公钥加密)，在建立传输链路时，SSL首先对对称加密的密钥使用公钥进行非对称加密，链路建立好之后，SSL对传输内容使用对称加密。

> 对称加密 速度高，可加密内容较大，用来加密会话过程中的消息

> 公钥加密 加密速度较慢，但能提供更好的身份认证技术，用来加密对称加密的密钥

2.1.1.HTTPs单向认证机制

单向认证主要是客户端保存有服务端的公钥证书，自己本身是没有私钥证书的。

1.给服务器生成密钥方式：

- keytool -genkeypair -alias skxy -keyalg RSA -validity 3650 -keypass 123456 -storepass 123456 -keystore skxy.keystore

2. 给Tomcat服务器配置Https

- tomcat/config/server.xml修改connector配置

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11Protocol"
        maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
        clientAuth="false" sslProtocol="TLS"
        keystoreFile="conf/skxy.keystore"
        keystorePass="123456"/>
```

3.导出证书

- keytool -export -alias skxy -file skxy.cer -keystore skxy.keystore -storepass 123456

4.将证书放在android客户端，能够读取的地方比如assert目录 5.代码中执行网络请求，获取证书，读取https网站的数据

客户端单向认证代码实现部分

```
String path = "https://10.0.3.2:8443/Test/Hlloer";

  try {
    //获取证书
    InputStream stream = getAssets().open("skxy.cer");

    SSLContext tls = SSLContext.getInstance("TLS");
```

```
//使用默认证书
KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
//去掉系统默认证书
keystore.load(null);
Certificate certificate =
    CertificateFactory.getInstance("X.509").generateCertificate(stream);
//设置自己的证书
keystore.setCertificateEntry("skxy", certificate);
//通过信任管理器获取一个默认的算法
String algorithm = TrustManagerFactory.getDefaultAlgorithm();
//算法工厂创建
TrustManagerFactory instance = TrustManagerFactory.getInstance(algorithm);
instance.init(keystore);
tls.init(null, instance.getTrustManagers(), null);
SSLSocketFactory socketFactory = tls.getSocketFactory();
HttpsURLConnection.setDefaultSSLSocketFactory(socketFactory);

URL url = new URL(path);
HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
//设置ip授权认证：如果已经安装该证书，可以不设置，否则需要设置
conn.setHostnameVerifier(new HostnameVerifier() {
  @Override
  public boolean verify(String hostname, SSLSession session) {
    return true;
  }
});

InputStream inputStream = conn.getInputStream();
String result = getString(inputStream);
stream.close();
```

单向认证过程：

(1) 客户端向服务端发送SSL协议版本号、加密算法种类、随机数等信息。

(2) 服务端给客户端返回SSL协议版本号、加密算法种类、随机数等信息，同时也返回服务器端的证书，即公钥证书
(3) 客户端使用服务端返回的信息验证服务器的合法性，包括：
    证书是否过期
    发型服务器证书的CA是否可靠
    返回的公钥是否能正确解开返回证书中的数字签名
    服务器证书上的域名是否和服务器的实际域名相匹配
    验证通过后，将继续进行通信，否则，终止通信
(4) 客户端向服务端发送自己所能支持的对称加密方案，供服务器端进行选择
(5) 服务器端在客户端提供的加密方案中选择加密程度最高的加密方式。
(6) 服务器将选择好的加密方案通过明文方式返回给客户端
(7) 客户端接收到服务端返回的加密方式后，使用该加密方式生成产生随机码，用作通信过程中对称加密的密钥，使用服务端返回的公钥进行加密，将加密后的随机码发送至服务器
(8) 服务器收到客户端返回的加密信息后，使用自己的私钥进行解密，获取对称加密密钥。
在接下来的会话中，服务器和客户端将会使用该密码进行对称加密，保证通信过程中信息的安全。

2.1.2.Https双向认证机制

首先对于双向证书验证，也就是说，客户端有自己的密钥，并持有服务端的证书，服务端给客户端发送数据时，需要将服务端的证书发给客户端验证，验证通过才运行发送数据，同样，客户端请求服务器数据时，也需要将自己的证书发给服务端验证，通过才允许执行请求。

客户端双向认证代码实现部分

```
public class MySSLSocketFactory {

    private static final String KEY_STORE_TYPE_BKS = "bks";//证书类型
    private static final String KEY_STORE_TYPE_P12 = "PKCS12";//证书类型


    private static final String KEY_STORE_PASSWORD = "****";//证书密码（应该是客户端证书密码）
    private static final String KEY_STORE_TRUST_PASSWORD = "***";//授信证书密码（应该是服务端证书密码）

    public static SSLSocketFactory getSocketFactory(Context context) {


        InputStream trust_input = context.getResources().openRawResource(R.raw.trust);//服务器授信证书
        InputStream client_input = context.getResources().openRawResource(R.raw.client);//客户端证书
        try {
                    SSLContext sslContext = SSLContext.getInstance("TLS");
                    KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
                    trustStore.load(trust_input, KEY_STORE_TRUST_PASSWORD.toCharArray());
                    KeyStore keyStore = KeyStore.getInstance(KEY_STORE_TYPE_P12);
                    keyStore.load(client_input, KEY_STORE_PASSWORD.toCharArray());
                    TrustManagerFactory trustManagerFactory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
                    trustManagerFactory.init(trustStore);

                    KeyManagerFactory keyManagerFactory =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
                   keyManagerFactory.init(keyStore, KEY_STORE_PASSWORD.toCharArray());
                   sslContext.init(keyManagerFactory.getKeyManagers(),
trustManagerFactory.getTrustManagers(), new SecureRandom());
                   SSLSocketFactory factory = sslContext.getSocketFactory();
                   return factory;
               } catch (Exception e) {
                            e.printStackTrace();
                           return null;
               } finally {
                           try {
                                 trust_input.close();
                                 client_input.close();
                           } catch (IOException e) {
                                e.printStackTrace();
                           }
               }
}
```

双向认证过程：

(1) 客户端向服务端发送SSL协议版本号、加密算法种类、随机数等信息。
(2) 服务端给客户端返回SSL协议版本号、加密算法种类、随机数等信息，同时也返回服务器端的证书，即公钥证书
客户端使用服务端返回的信息验证服务器的合法性，包括：
证书是否过期
发型服务器证书的CA是否可靠
返回的公钥是否能正确解开返回证书中的数字签名
服务器证书上的域名是否和服务器的实际域名相匹配
(3) 验证通过后，将继续进行通信，否则，终止通信
(4) 服务端要求客户端发送客户端的证书，客户端会将自己的证书发送至服务端
(5) 验证客户端的证书，通过验证后，会获得客户端的公钥
(6) 客户端向服务端发送自己所能支持的对称加密方案，供服务器端进行选择
(7) 服务器端在客户端提供的加密方案中选择加密程度最高的加密方式
(8) 将加密方案通过使用之前获取到的公钥进行加密，返回给客户端
(9) 客户端收到服务端返回的加密方案密文后，使用自己的私钥进行解密，获取具体加密方式，而后，产生该加密方式的随机码，用作加密过程中的密钥，使用之前从服务端证书中获取到的公钥进行加密后，发送给服务端
(10) 服务端收到客户端发送的消息后，使用自己的私钥进行解密，获取对称加密的密钥，在接下来的会话中，服务器和客户端将会使用该密码进行对称加密，保证通信过程中信息的安全。

2.2.我们身边的app中所使用的加密传输是怎样的呢?

某宝（金融类app）的数据加密分析（https单向认证） 为了更加清晰的了解https在实际项目中的应用，特意花了点时间分析了一个app的加密认证过程。app虽然加了部分混淆，但并没有加固，所以也不难分析。

（1）整个发送https post请求过程.ip以及域名都是固定的，证书也写死在app里。

```
static {
    ao.d = "211.151.121.41";
    ao.f = 1;
    ao.g = "-----BEGIN CERTIFICATE-----\nMIIDyDCCArACCQD9bypp/+5VujANBgkqhkiG9w0BAQUFADCBpDELMAkGA1UEBhMC\nQ04xEDAOBgNVBAgMB0J1aUppbmcxEDAOBgNVBAcMB0J
    ao.h = "utf-8";
    ao.i = new CRC32();
    ao.j = new bm(ao.g, "xdrig.com", ao.d, 443);  // 判断代理服务器
    ao.a = new HandlerThread("TDForwardManager");  // 创建一个新线程
    ao.b = null;
    ao.a.start();  // 启动线程
    ao.b = new ap(ao.a.getLooper());  // 发送数据
}
```

（2）https认证过程

- 判断代理服务器以及证书校验

```
    private static final int j = 60000;

    public bm(String arg2, String arg3, String arg4, int arg5) {
        this(bm.g(arg2), arg3, arg4, arg5);  // https认证参数 证书、"xdrig.com","211.151.121.41", 443 /  /
    }
```

- 证书校验过程

在获取证书的过程中，仅仅读取了证书的信息，并没有实现校验证书是否安全可靠的代码。这里就留下了安全隐患。使用第三方证书一样可以截获数据。

```
        a(X509Certificate arg6) {  // 获取证书
            super(null);
            this.a = SSLContext.getInstance("TLS");
            this.a.init(null, new TrustManager[]{new b(arg6)}, null);
        }
```

```
    private static X509Certificate g(String arg3) {  // 获取证书
        X509Certificate v0_4;
        ByteArrayInputStream v0 = new ByteArrayInputStream(arg3.getBytes());
        X509Certificate v1 = null;
        try {
            Certificate v0_3 = CertificateFactory.getInstance("X.509").generateCertificate(((InputStream)v0));  // 创建证书对象
        }
        catch(Exception v0_1) {
            v0_4 = v1;
        }
        catch(Throwable v0_2) {
            throw v0_2;
        }

        return v0_4;
    }
```

```
public class bm {
    class a extends SSLSocketFactory {
        SSLContext a;

        a(X509Certificate arg6) {  // 获取证书
            super(null);
            this.a = SSLContext.getInstance("TLS");
            this.a.init(null, new TrustManager[]{new b(arg6)}, null);
        }

        public Socket createSocket() {
            return this.a.getSocketFactory().createSocket();
        }

        public Socket createSocket(Socket arg2, String arg3, int arg4, boolean arg5) {
            return this.a.getSocketFactory().createSocket(arg2, arg3, arg4, arg5);
        }
    }

    class b implements X509TrustManager {
        X509Certificate a;

        b(X509Certificate arg1) {
            super();
            this.a = arg1;
        }

        public void checkClientTrusted(X509Certificate[] arg1, String arg2) {
        }

        public void checkServerTrusted(X509Certificate[] arg5, String arg6) {
            int v1 = arg5.length;
            int v0;
            for(v0 = 0; v0 < v1; ++v0) {
                if(arg5[v0].equals(this.a)) {
                    return;
                }
```

- 数据解密过程

在数据解密过程也不够严谨，密钥和向量通过简单逆向分析就能获得

```
public static byte[] DesDecode(byte[] arg4, byte[] arg5) {  // 解密函数
    byte[] v0_2;
    try {
        SecretKey v0_1 = SecretKeyFactory.getInstance("DES").generateSecret(new DESKeySpec(arg5));
        Cipher v1 = Cipher.getInstance("DES/CBC/PKCS5Padding");
        v1.init(2, ((Key)v0_1), new IvParameterSpec(bt.n));
        v0_2 = v1.doFinal(arg4);
    }
    catch(Exception v0) {
        v0_2 = null;
    }

    return v0_2;
}
```

解密key的获取方式：数据包名的md5

```
static {
    an.b = "td_databaseTalkingData";
    an.c = "utf-8";
    an.a = new bf(aq.a, an.b);  // 随机读写文件
    String v0 = bt.c(w.a.getPackageName());  // 获取该包名的md5，作为对称加密的key
    an.d = w.a == null || v0 == null ? w.class.getSimpleName().getBytes() : v0.getBytes();  // 文件md5
}

public an() {
    super();
}

public static List a() {
    ArrayList v1;
    Class v2 = an.class;
    __monitor_enter(v2);
    List v0 = null;
    try {
        List v3 = an.a.a(100);
        if(v3.size() <= 0) {
            goto label_23;
        }

        v1 = new ArrayList();
        Iterator v3_1 = v3.iterator();
        while(v3_1.hasNext()) {
            Object v0_2 = v3_1.next();
            try {
                byte[] v4 = an.d;
                ((List)v1).add(new String(bt.DesDecode(((byte[])v0_2), v4)));
            }
            catch(Exception v0_3) {
            }
        }
    }
```

解密向量

```
static {
    boolean v0 = !bt.class.desiredAssertionStatus() ? true : false;
    bt.e = v0;
    bt.a = true;
    bt.c = false;
    bt.d = false;
    bt.g = "ge";
    bt.h = "tp";
    bt.i = "rop";
    bt.j = Executors.newSingleThreadExecutor();
    bt.m = new byte[]{65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84
    bt.n = new byte[]{1, 2, 3, 4, 5, 6, 7, 8};  // 向量
}
```

解密向量

通过这个简单分析，你还敢说你的数据是安全传输的吗?

2.3 安全隐患

2.3.1.因为开发方便而信任所有证书

手机银行开发人员在开发过程中为了解决ssl证书报错的问题（使用了自己生成了证书后，客户端发现证书无法与系统可信根CA形成信任链，出现了 CertificateException等异常。），会在客户端代码中信任客户端中所有证书的方式：

```
01 public static HttpClient getWapHttpClient() {
02     try {
03             KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultT
   ype());
04             trustStore.load(null, null);
05             SSLSocketFactory sf = new MySSLSocketFactory(trustStore);
06             sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERI
   FIER);
07             //此处信任手机中的所有证书，包括用户安装的第三方证书
08             HttpParams params = new BasicHttpParams();
09             HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
10             HttpProtocolParams.setContentCharset(params, HTTP.UTF_8);
11             SchemeRegistry registry = new SchemeRegistry();
12             registry.register(new Scheme("http", PlainSocketFactory.getSock
   etFactory(), 80));
13             registry.register(new Scheme("https", sf, 443));
14             ClientConnectionManager ccm = new ThreadSafeClientConnManager(p
   arams, registry);
15             return new DefaultHttpClient(ccm, params);
16     } catch (Exception e) {
17             return new DefaultHttpClient();
18     }
19 }
```

2.3.2.重写了校验机制，但并没有做任何检验SSL证书有效性。

而在客户端中覆盖google默认的证书检查机制（X509TrustManager），并且在代码中无任何校验SSL证书有效性相关代码：

```
01 public class MySSLSocketFactory extends SSLSocketFactory {
02     SSLContext sslContext = SSLContext.getInstance("TLS");
03
04     public MySSLSocketFactory(KeyStore truststore)
05         throws NoSuchAlgorithmException, KeyManagementException,
06             KeyStoreException, UnrecoverableKeyException {
07         super(truststore);
08
09         TrustManager tm = new X509TrustManager() {
10                 public void checkClientTrusted(X509Certificate[] chain,
11                     String authType) throws CertificateException {
12                 }
13
14                  //客户端并未对SSL证书的有效性进行校验，并且使用了自定义方法的方式覆盖android自带的校验方法
15                 public void checkServerTrusted(X509Certificate[] chain,
16                     String authType) throws CertificateException {
17                 }
18
19                 public X509Certificate[] getAcceptedIssuers() {
20                     return null;
21                 }
22             };
23
24         sslContext.init(null, new TrustManager[] { tm }, null);
25     }
26 }
```

3. 扩展 Java Security安全体系知识延伸

3.1.Java Security 背景知识

Java Security其实是Java平台中一个比较独立的模块。除了软件实现上内容外，它实际上对应了一系列的规范。从Java2开始，Java Security包含主要三个重要的规范：

- JavaCryptography Extension（简写为JCE），JCE所包含的内容有加解密，密钥交换，消息摘要（Message Digest，比如MD5等），密钥管理等。本文所涉及的大部分内容都属于JCE的范畴。
- JavaSecure Socket Extension（简写为JSSE），JSSE所包含的内容就是Java层的SSL/TLS。简单点说，使用JSSE就可以创建SSL/TLS socket了。
- JavaAuthentication and Authorization Service（简写为JAAS），JSSA和认证/授权有关。这部分内容在客户端接触得会比较少一点，所以本文不拟讨论它。

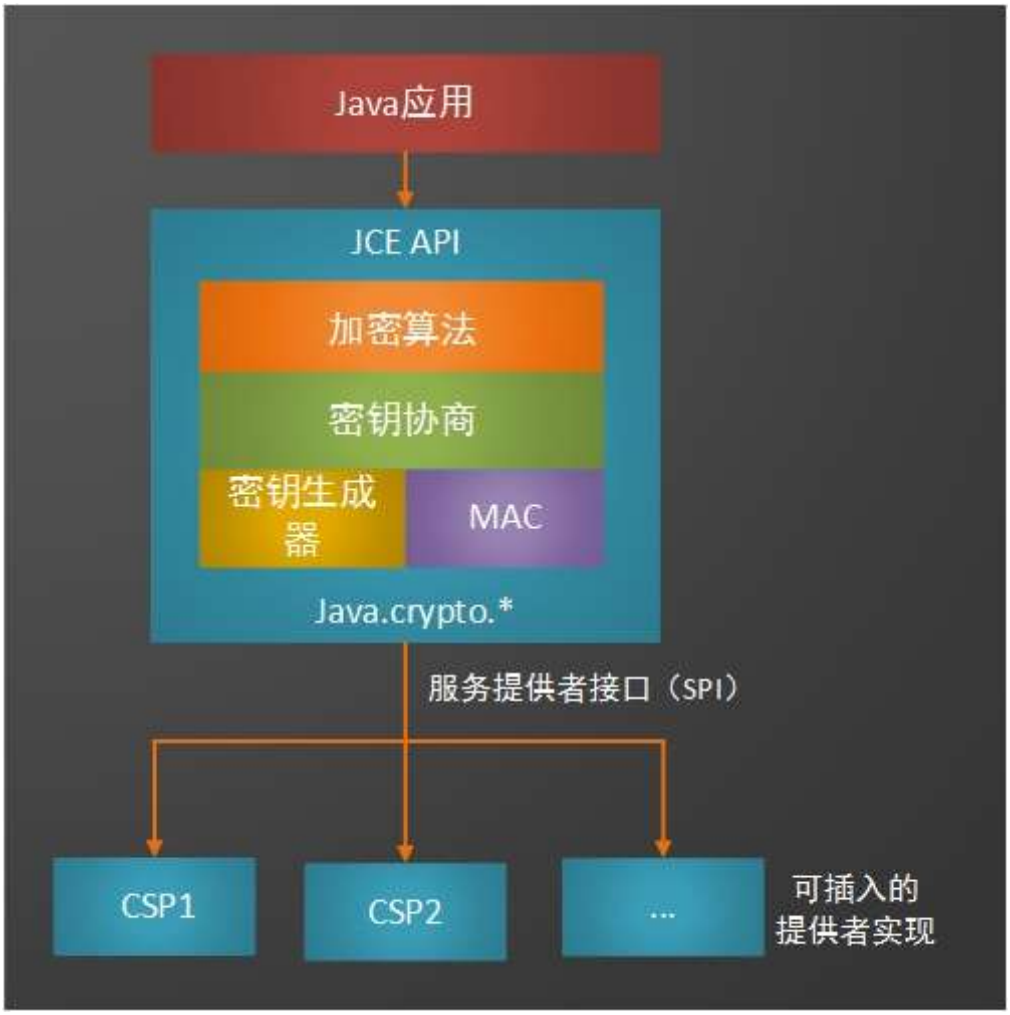
在上述三个子模块或规范中，JCE是JavaSecurity的大头，其他两个子模块JSSE和JAAS都依赖于它，比如SSL/TLS在工作过程中需要使用密钥对数据进行加解密，那么密钥的创建和使用就依靠JCE子模块了。 另外，既然和安全相关，那么对安全敏感的相关部门或政府肯定会有所干涉。Java是在美国被发明的，所以美国政府对于Java

Security方面的出口（比如哪些模块，哪些功能能给其他国家使用）有相关的限制。例如，不允许出口的JCE（从软件实现上看，可能就是从Java官网上下载到的几个Jar包文件）支持一些高级的加解密功能（比如在密钥长度等方面有所限制）。

3.2 JCE的介绍

JCE最初是作为JCA的扩展包开发的，旨在提供受美国出口控制条例管制的加密服务API和实现。JCE提供一个提供者实现和一组相关的API和包，以支持加密和解密，密钥的生成和协商以及消息验证算法，其中对加密和解密的支持包括对称加密、非对称加密、块加密和流加密。JCE还支持安全流和封装流对象。

JCE的架构模型如下图所示：



issue

1.不要忽略证书校验

2.保护好自己的密钥

3.尽量使用规范的https协议

参考

1.http://blog.csdn.net/xdd19910505/article/details/51926540

2.https://www.cnblogs.com/xiekeli/p/5607107.html

3.http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2014/0607/1621.html

4.https://www.waitalone.cn/bank-ssl-cap.html