CA certificate authority 证书颁发机构

签名，使用私钥对需要传输的文本的摘要进行加密，得到的密文即被称为该次传输过程的签名。

SSL：（Secure Socket Layer，安全套接字层），位于可靠的面向连接的网络层协议和应用层协议之间的一种协议层。SSL 通过互相认证、使用数字签名确保完整性、使用加密确保私密性，以实现客户端和服务器之间的安全通讯。该协议由两层组成：SSL 记录协议和 SSL 握手协议。

TLS：(Transport Layer Security，传输层安全协议)，用于两个应用程序之间提供保密性和数据完整性。该协议由两层组成：TLS 记录协议和 TLS 握手协议。

SSL 是 Netscape 开发的专门用户保护 Web 通讯的，目前版本为 3.0。

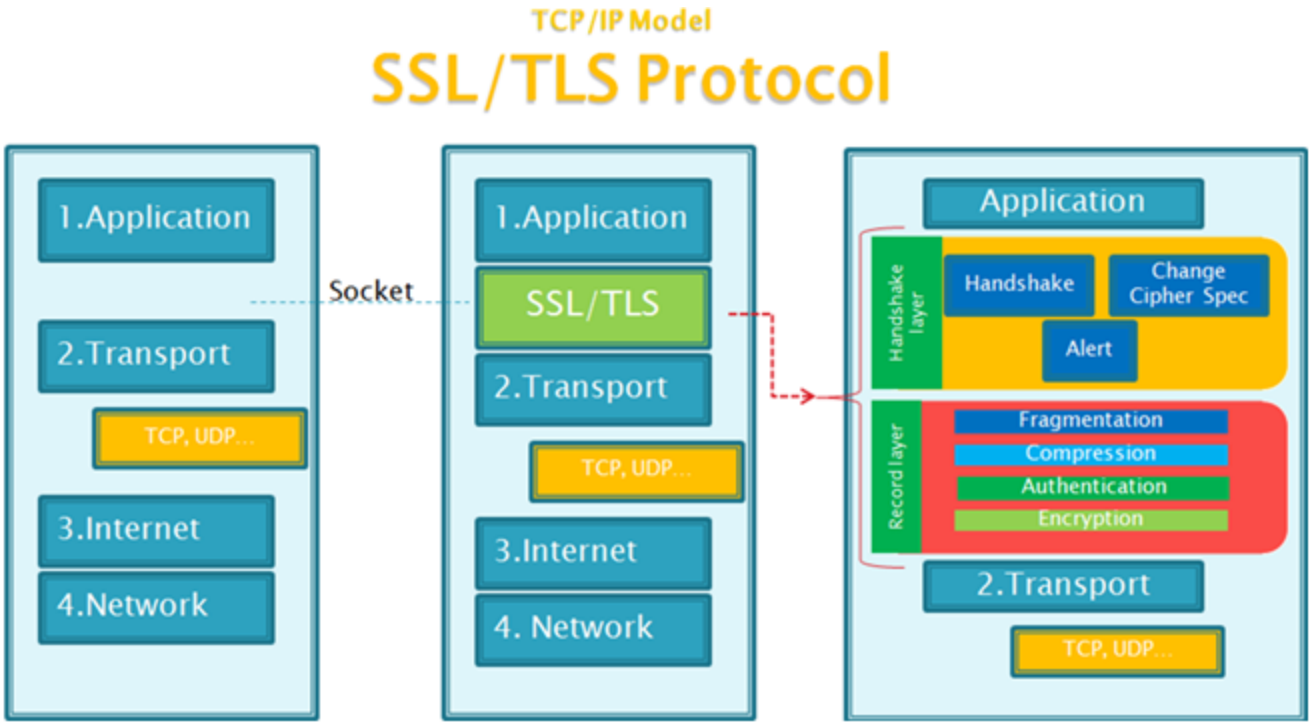
最新版本的 TLS 1.0 是 IETF(工程任务组)制定的一种新的协议，它建立在 SSL 3.0 协议规范之上，是 SSL 3.0 的后续版本。

SSL (Secure Socket Layer)

SSL 协议位于 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。SSL 协议可分为两层： **SSL 记录协议**（SSL Record Protocol）：它建立在可靠的传输协议（如 TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。 **SSL 握手协议**（SSL Handshake Protocol）：它建立在 SSL 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。

应用层数据不再直接传递给传输层，而是传递给 SSL 层，SSL 层对从应用层收到的数据进行 n 加密，并增加自己的 SSL 头。

SSL 并不依赖于 TCP，它可以建立在任何可靠的传输层协议（比如 TCP）之上。也就是说 SSL 是不能建立在 UDP 之上 的。这是显然的，如果传输都不可靠，偶尔丢两个包或者包的顺序换一换的话，怎么保证安全呢？



SSL 协议提供的服务主要有：

1）认证用户和服务器，确保数据发送到正确的客户机和服务器；

2）加密数据以防止数据中途被窃取；

3）维护数据的完整性，确保数据在传输过程中不被改变。

SSL 协议的工作流程：

服务器认证阶段：

1）客户端向服务器发送一个开始信息“Hello”（ClientHello），进行请求以便开始一个新的会话连接；

2）服务器返回证书(Certificate)也就是公钥，并根据客户的信息确定是否需要生成新的主密钥，如需要则服务器在响应客户的“Hello”信息（ServerHello）时将包含生成主密钥所需的信息；

3）客户根据收到的服务器响应信息，产生一个(随机的、对称的）主密钥(ClientKeyExchange)，并用服务器的公开密钥加密后传给服务器；

4）服务器使用私有密钥恢复该主密钥，并返回给客户一个用主密钥认证的信息，以此让客户认证服务器。

用户认证阶段：

在此之前，服务器已经通过了客户认证，这一阶段主要完成对客户的认证。经认证的服务器发送一个提问给客户，客户则返回（数字）签名后的提问和其公开密钥，从而向服务器提供认证。

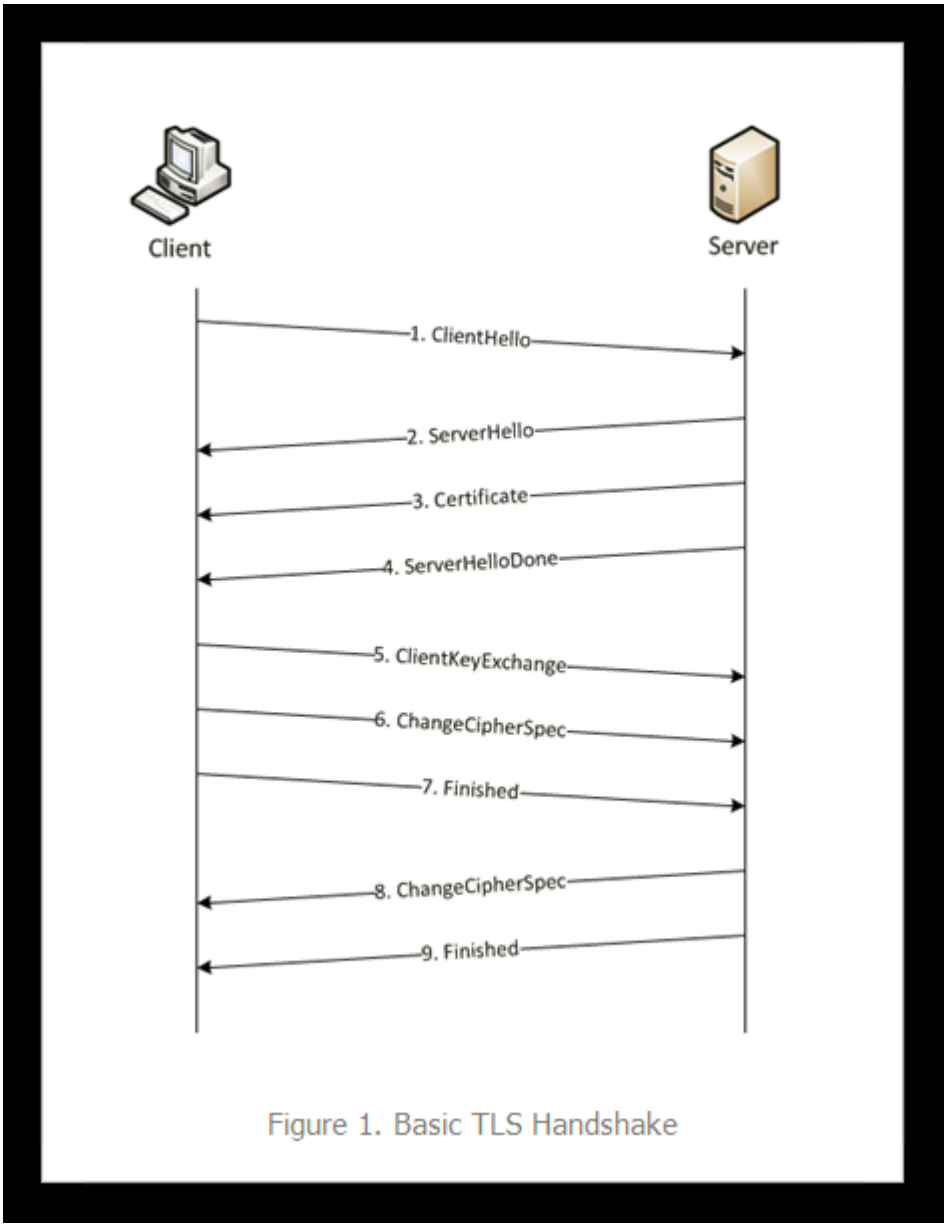


Figure 1. Basic TLS Handshake

RSA 性能是非常低的，原因在于寻找大素数、大数计算、数据分割需要耗费很多的 CPU 周期，所以一般的 HTTPS 连接只在第一次握手时使用非对称加密，通过握手交换对称加密密钥，在之后的通信走对称加密。

总结：
服务器 用 RSA 生成公钥和私钥
把公钥放在证书里发送给客户端，私钥自己保存
客户端首先向一个权威的服务器检查证书的合法性，如果证书合法，客户端产生一段随机数，这个随机数就作为通信的密钥，我们称之为对称密钥，用公钥加密这段随机数，然后发送到服务器
服务器用密钥解密获取对称密钥，然后，双方就已对称密钥进行加密解密通信了

SSL 和 TLS 的关系是**并列关系**

最新版本的 TLS（Transport Layer Security，传输层安全协议）是 IETF（Internet Engineering Task Force，Internet 工程任务组）制定的一种新的协议，它建立在 SSL 3.0 协议规范之上，是 SSL 3.0 的后续版本。在 TLS 与 SSL3.0 之间存在着显著的差别，主要是它们所支持的加密算法不同，所以 TLS 与 SSL3.0 不能互操作。

1．TLS 与 SSL 的差异

1）版本号：TLS 记录格式与 SSL 记录格式相同，但版本号的值不同，TLS 的版本 1.0 使用的版本号为 SSLv3.1。

2）报文鉴别码：SSLv3.0 和 TLS 的 MAC 算法及 MAC 计算的范围不同。TLS 使用了 RFC-2104 定义的 HMAC 算法。SSLv3.0 使用了相似的算法，两者差别在于 SSLv3.0 中，填充字节与密钥之间采用的是连接运算，而 HMAC 算法采用的是异或运算。但是两者的安全程度是相同的。

3）伪随机函数：TLS 使用了称为 PRF 的伪随机函数来将密钥扩展成数据块，是更安全的方式。

4）报警代码：TLS 支持几乎所有的 SSLv3.0 报警代码，而且 TLS 还补充定义了很多报警代码，如解密失败（decryption_failed）、记录溢出（record_overflow）、未知 CA（unknown_ca）、拒绝访问（access_denied）等。

5）密文族和客户证书：SSLv3.0 和 TLS 存在少量差别，即 TLS 不支持 Fortezza 密钥交换、加密算法和客户证书。

6）certificate_verify 和 finished 消息：SSLv3.0 和 TLS 在用 certificate_verify 和 finished 消息计算 MD5 和 SHA-1 散列码时，计算的输入有少许差别，但安全性相当。

7）加密计算：TLS 与 SSLv3.0 在计算主密值（master secret）时采用的方式不同。

8）填充：用户数据加密之前需要增加的填充字节。在 SSL 中，填充后的数据长度要达到密文块长度的最小整数倍。而在 TLS 中，填充后的数据长度可以是密文块长度的任意整数倍（但填充的最大长度为 255 字节），这种方式可以防止基于对报文长度进行分析的攻击。

2．TLS 的主要增强内容

TLS 的主要目标是使 SSL 更安全，并使协议的规范更精确和完善。TLS 在 SSL v3.0 的基础上，提供了以下增强内容：

1）更安全的 MAC 算法

2）更严密的警报

3）“灰色区域”规范的更明确的定义

3．TLS 对于安全性的改进

1）对于消息认证使用密钥散列法：TLS 使用“消息认证代码的密钥散列法”（HMAC），当记录在开放的网络（如因特网）上传送时，该代

码确保记录不会被变更。SSLv3.0 还提供键控消息认证，但 HMAC 比 SSLv3.0 使用的（消息认证代码）MAC 功能更安全。

2）增强的伪随机功能（PRF）：PRF 生成密钥数据。在 TLS 中，HMAC 定义 PRF。PRF 使用两种散列算法保证其安全性。如果任一算法暴露了，只要第二种算法未暴露，则数据仍然是安全的。

3）改进的已完成消息验证：TLS 和 SSLv3.0 都对两个端点提供已完成的消息，该消息认证交换的消息没有被变更。然而，TLS 将此已完成消息基于 PRF 和 HMAC 值之上，这也比 SSLv3.0 更安全。

4）一致证书处理：与 SSLv3.0 不同，TLS 试图指定必须在 TLS 之间实现交换的证书类型。

5）特定警报消息：TLS 提供更多的特定和附加警报，以指示任一会话端点检测到的问题。TLS 还对何时应该发送某些警报进行记录。

SASL Simple Authentication and Security Layer 简单验证和安全层

SASL 为应用程序和共享库的开发者提供了用于验证、数据完整性检查和加密的机制。开发者可通过 SASL 对通用 API 进行编码。此方法避免了对特定机制的依赖性。SASL 特别适用于使用 IMAP、SMTP、ACAP 和 LDAP 协议的应用程序，因为这些协议全都支持 SASL。

SASL 通过 LDAP v3 和 IMAP v4 协议来确保鉴权是可插拔的。而不是硬编码一个鉴权方法到协议。

SASL 是一种 challange-response 协议，server 发布 challenge 到 client，而 client 基于 challange 发送 response。

这种交换直到 server 被满足且 b 不再发布 challage。challenge 和 response 是任意长度的二进制标记，封装协议诸如 LDAP、

IMAP 指定了这些标记如何被编码和交换的。例如 LDAP 指定了 SASL 标记被封装在 LDAP request 和 response 内。

Java SASL 根据这种应答和使用方式已经模板化了，SaslClient 和 SaslServer 接口，分别代表了 client-side 和

server-side 的机制。应用程序通过代表 challenge 和 response 的字节数组，使用这种机制交互。server-side 机

制：不断发布 challenge,处理 response 直到被满足，而 client-side 机制：不断评估 challenge，且发布 response
直到 server 满足。

Kafka Security（0.10.1.0）
在 0.9.0.0 版本中，kafka 社区加入了很多特性，这些特性或者单独被使用或者一起使用，提高了 Kafka 集群的安全性。这些特性被认为是衡量标准。以下安全测评现在是被支持的：

1. 从 broker 向 client、其他 borker 和工具连接的身份认证，使用 SSL 或者 SASL（Kerberos）。从 0.10.0.0 版本开始，SASL/PLAIN 也可以被使用；
2. 从 broker 向 zookeeper 连接的身份认证；
3. 在 broker 和 clients、brokers 或工具之间的使用 SSL 的传输数据的加密（注意使用 SSL 会使得性能有所下降，下降的量级由 CPU 的类型和 JVM 实施有关）
4. 由 client 完成的读写操作的认证
5. 认证是可插拔的并且对外部认证的整合是支持的

值得注意的是安全是可选的，不安全的集群是被支持的，除此之外还有认证的、非认证的、加密的、非加密的 clients 的混合。
下面的指导解释了如何配置并且使用 client 和 broker 各自的安全特性。
1.使用 SSL 的加密和认证
1.1 为每个 kafka broker 生成 SSL 密钥和证书
实施 HTTPS 的第一步就是为集群中的每一台机器生成一个密钥和证书。可以使用 java 的 keytool 组件来完成这项任务。
初始生成一个 key 到临时的 keystore 中，这样我们可以在之后导出并且使用 CA 对它进行签名（CA 也有一个证书，内含公钥和私钥）。

```
keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genkey
```

需要在上述命令中指定两个参数

1. keystore： 存储证书的 keystore 文件。keystore 文件包含了证书的私钥（当然还有公钥）；所以，它需要被安全地保存。
2. vality： 证书的有效时间。

注意：默认下属性 ssl.endpoint.identification.algorithm 是未定义的，所以 hostname 验证没有被进行。为了使 hostname 验证有效，设置如下属性：
ssl.endpoint.identification.algorithm=HTTPS
一旦使之有效，clients 就会根据以下两个 field 来验证 server 的 fully qualified domain name（FQDN）
1.Common Name (CN) 通用名
2.Subject Alternative Name (SAN) 主体别名

两个 field 都有效，但是 RFC-2818 推荐使用 SAN。SAN 也更灵活，允许多个 DNS entry 被声明。另一个好处是 CN 可以被设置成一个更有意义的值来认证。为了加入 SAN field，需要在 keytool 命令扩展如下参数-ext SAN=DNS:{FQDN}

```
keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genkey -ext SAN=DNS:{FQDN}
```

然后如下命令来验证生成的证书

```
keytool -list -v -keystore server.keystore.jks
```

1.2 生成自己的 CA

在第一步之后，每一台在集群中的机器都有一个公钥-私钥对（用于和 client 进行对称密钥的交换）和一个证书（用于证明自己的身份）。但是这个证书是未签名的，这意味着一个攻击者可以创建一个证书来冒充任意机器。

所以，对集群中的每台机器进行签名来防止假冒的证书。CA 就是负责对证书进行签名的。CA 像一个办理护照的政府（政府对护照进行盖章使其难以造假）。其他政府对印章进行验证来确认证书是官方的。相似的，CA 对证书进行签名，并且这种加密保证了一个签名的证书很难造假。所以只要保证 CA 是官方的，clients 就能确信它们连接的是官方的机器。

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

这个生成的 CA 不过是一个公钥-私钥对和证书，它是为了对其他证书签名的。

下一步是将生成的 CA 加入**clients' truststore**让 clients 可以信任它

```
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
```

注意：如果你配置了 kafka broker 来请求 client 的身份（通过在 kafka brokers config 中设置 ssl.client.auth 为"requested" 或 "required" ），那么你必须为 kafka brokers 提供一个 truststore 并且这个文件应该包含所有 client 的证书被签名时使用的 CA 的证书。

```
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
```

与在第一步中用于存储每台机器自己身份的 keystore 相反，一个 client 的 truststore 存储这个 client 应该信任的所有证书。向 truststore 导入一个证书也意味着信任由该证书签名的所有证书。这种属性被称为信任链，在大型 Kafka 集群上部署 SSL 时非常有用。你可以使用一个单独的 CA 对集群中的所有证书进行签名，并且让所有机器共享一个信任 CA 的 truststore。这样所有机器就可以去认证其他机器。

1.3 对证书签名

下一步是使用步骤 2 中证书对步骤 1 中的所有证书进行签名。首先，你需要从 keystore 导出该机器自己的证书：

```
keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-file
```

然后使用 CA 对它进行签名：

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days {validity} -CAcreateserial -passin pass:{ca-password}
```

最后，你需要向 keystore 中导入 CA 的证书和签名后的证书：

```
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
keytool -keystore server.keystore.jks -alias localhost -import -file cert-signed
```

参数定义如下：

1.keystore: the location of the keystore
2.ca-cert: the certificate of the CA
3.ca-key: the private key of the CA

4.ca-password: the passphrase of the CA
5.cert-file: the exported, unsigned certificate of the server
6.cert-signed: the signed certificate of the server
这里有一个以上步骤的 bash 脚本实例
注意命令之一假设有一个密码为 test1234，所以要么就使用这个密码要么在运行之前编辑命令

```
#!/bin/bash
#Step 1
keytool -keystore server.keystore.jks -alias localhost -validity 365 -keyalg RSA -genkey
#Step 2
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
#Step 3
keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-file
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days 365 -
CAcreateserial -passin pass:test1234
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
keytool -keystore server.keystore.jks -alias localhost -import -file cert-signed
```

1.4 配置 kafka broker
kafka 支持在多个端口上面的对连接的监听。我们需要在 server.properties 去配置 listeners 属性，这个属性有一个或多个由逗号分开的值：
如果 SSL 没有被激活来进行 broker 之间交流（下面会讲解如何激活），那么 PALINTEXT 和 SSL 端口都将会是必要的。

```
listeners=PLAINTEXT://host.name:port,SSL://host.name:port
```

在 broker 端，需要进行如下 SSL 配置：

```
ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
ssl.truststore.password=test1234
```

如下可选项设置需要被考虑：
1.ssl.client.auth=none ("required" => client authentication is required, "requested" => client authentication is requested and client without certs can still connect. The usage of "requested" is discouraged as it provides a false sense of security and misconfigured clients will still connect successfully.)
2.ssl.cipher.suites (Optional). A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. (Default is an empty list)
3.ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1 (list out the SSL protocols that you are going to accept from clients. Do note that SSL is deprecated in favor of TLS and using SSL in production is not recommended)
4.ssl.keystore.type=JKS
5.ssl.truststore.type=JKS
6.ssl.secure.random.implementation=SHA1PRNG

如果要激活 broker 之间交流的 SSL，需要在 broker properties 文件中加入如下属性（默认是 PLAINTEXT）

security.inter.broker.protocol=SSL

由于在一些国家的 import 规定，在 Oracle 的实现中默认限制可用加密算法的长度。如果需要更强的算法（例如 256 位密钥的 AES），必须要拥有并且在 JDK/JRE 中安装 JCE Unlimited Strength Jurisdiction Policy Files。如需要更多信息，见 JCA Providers Documentation。

JRE/JDK 将拥有一个默认的 pseudo-random 数字生成器（PRNG）来进行加密操作，所以不需要去配置带有 ssl.secure.random.implementation 的实施。然而，在一些实现上会有一些性能问题（值得注意的是，在 Linux 系统的默认选项，使用一个全局锁）。为防止 SSL 连接的性能成为问题，考虑设置将被使用的实现。SHA1PING 实现是非阻塞的并且在高负载下表现了良好的性能（每台每秒 50MB 的被生产的消息（加上副本的传输））

一旦你开启了 broker，你应该能够在 server.log 看到

with addresses: PLAINTEXT -> EndPoint(192.168.64.1,9092,PLAINTEXT),SSL -> EndPoint(192.168.64.1,9093,SSL)

为了快速检查 keystore 和 truststore 是否被合理地建立，你可以运行如下命令：

openssl s_client -debug -connect localhost:9093 -tls1

（注意：TLSv1 应该在 ssl.enabled.protocols 下被列出）

在这条命令的输出中你应该看见 server 的证书

-----BEGIN CERTIFICATE-----

{variable sized random bytes}

-----END CERTIFICATE-----

subject=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=Sriharsha Chintalapani

issuer=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=kafka/emailAddress=test@test.com

1.5 配置 kafka clients

SSL 只在新的 Kafka Producer 和 Consumer 中被支持，老的 API 是不支持的。SSL 的配置对 producer 和 consumer 都是一样的。

如果 client 认证在 broker 处不被要求，那么如下是一个最简单的配置样例:

security.protocol=SSL

ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks

ssl.truststore.password=test1234

如果 client 认证被要求，那么一个如步骤 1 的 keystore 必须被创建并且必须配置如下：

ssl.keystore.location=/var/private/ssl/kafka.client.keystore.jks

ssl.keystore.password=test1234

ssl.key.password=test1234

根据我们的要求和 broker 配置，其他的配置设置可能也会需要

1.ssl.provider (Optional). The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.

2.ssl.cipher.suites (Optional). A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.

3.ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1. It should list at least one of the protocols configured on the broker side

4.ssl.truststore.type=JKS
5.ssl.keystore.type=JKS
使用 console-producer 和 console-consumer 的例子
kafka-console-producer.sh --broker-list localhost:9093 --topic test --producer.config client-ssl.properties
kafka-console-consumer.sh --bootstrap-server localhost:9093 --topic test --consumer.config client-ssl.properties

2.使用 SASL 进行认证

2.1 对 kafka broker 的 SASL 配置

2.1.1.在 broker 中选择一种或多种被支持的机制来激活。 GSSAPI 和 PLAIN 都是 Kafka 中当下被支持的机制.

2.1.2.按照建立 GSSAPI（Kerberos）（2.3 使用 SASL/Kerberos 进行验证） 或者 PLAIN（2.4 使用 SASL/PLAIN 进行验证） 的样例中的描述，为选择的机制加入一个 JAAS 配置文件

2.1.3.将 JAAS 配置文件作为 JVM 参数传给每个 Kafka broker。例如，
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf

2.1.4.在 server.properties 中配置一个 SASL 端口，向 listener 参数加入至少 SASL_PLAINTEXT 或 SASL_SSL 其中一个（前面表示 broker 之间的协议，后面表示 broker 和 client 之间的协议）
listeners=SASL_PLAINTEXT://host.name:port
如果使用了 SASL_SSL，那么 SSL 必须被配置（参看上面的 1）。如果只配置了一个 SASL 端口（换言之你希望 kafka broker 使用 SASL 来验证彼此），那么请确认你为 broker 之间交流设置了一样的 SASL 协议。
security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)

2.1.5.在 server.properties 激活一个或多个 SASL 机制：
sasl.enabled.mechanisms=GSSAPI (,PLAIN)

2.1.6.如果在 broker 之间交流中使用了 SASL，需要配置 broker 之间的 SASL 交流机制：
sasl.mechanism.inter.broker.protocol=GSSAPI (or PLAIN)

2.1.7.遵照 在 GSSAPI（Kerberos）（2.3 使用 SASL/Kerberos 进行验证）或者 PLAIN（2.4 使用 SASL/PLAIN 进行验证）中的步骤为相应的激活的机制对 SASL 进行配置。为了在 broker 中激活多种机制，遵循下面 4 中的步骤。

重要的注意事项：

2.1.7.1. KafkaServer 是被每一个 KafkaServer/Broker 在 JAAS 文件中使用的 section 名称。这个 section 为该 broker 包括任何由该 broker 完成的 SASL client 连接提供了 SASL 配置选项来进行 broker 间的交流。

2.1.7.2. client section 被用于认证一个与 zookeeper 的 SASL 连接。它也允许 broker 来设置在 zookeeper 节点上 SASL ACL 配置选项，这样这些节点只能被 broker 修改。在所有 broker 之间都使用相同的 principle 名字是必要的。如果你想要使用一个 section name 而不是 Client，设置系统属性 zookeeper.sasl.client 为一个合适的名字（例如-Dzookeeper.sasl.client=ZkClient）

2.1.7.3 Zookeeper 默认使用 zookeeper 为服务名字。如果你想要改变这个，设置系统属性 zookeeper.sasl.client.username 为合适的名字（例如 zookeeper.sasl.client.username = zk）

2.2 对 kafka clients 进行 SASL 配置
SASL 同 SSL 一样，只对新的 kafka producer 和 consumer 支持。
为了在 client 上配置 SASL 认证
2.2.1 为认证选择一个 SASL 机制
2.2.2 为被选择的机制添加一个 JAAS 配置文件（遵照 在 GSSAPI（Kerberos）（2.3 使用 SASL/Kerberos 进行验证）或者 PLAIN（2.4 使用 SASL/PLAIN 进行验证）中 的样例）。
KafkaClient 是 JAAS 文件中 kafka clients 使用的 section 名字
2.2.3 将 JAAS 配置文件作为 JVM 参数传给每个 Kafka broker。例如，
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
2.2.4 在 producer.properties 或者 consumer.properties 中配置如下属性
security.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism=GSSAPI (or PLAIN)
2.2.5 遵照 在 GSSAPI（Kerberos）（2.3 使用 SASL/Kerberos 进行验证）或者 PLAIN（2.4 使用 SASL/PLAIN 进行验证）中的步骤来为选择的机制配置 SASL

2.3 使用 SASL/Kerberos 进行认证
2.3.1.Prerequisites 先决条件
2.3.1. 1.Kerberos
If your organization is already using a Kerberos server (for example, by using Active Directory), there is no need to install a new server just for Kafka. Otherwise you will need to install one, your Linux vendor likely has packages for Kerberos and a short guidHe on how to install and configure it (Ubuntu, Redhat). Note that if you are using Oracle Java, you will need to download JCE policy files for your Java version and copy them to $JAVA_HOME/jre/lib/security.
2.3.1. 2.Create Kerberos Principals
If you are using the organization's Kerberos or Active Directory server, ask your Kerberos administrator for a principal for each Kafka broker in your cluster and for every operating system user that will access Kafka with Kerberos authentication (via clients and tools).
If you have installed your own Kerberos, you will need to create these principals yourself using the following commands:
sudo /usr/sbin/kadmin.local -q 'addprinc -randkey kafka/{hostname}@{REALM}'
sudo /usr/sbin/kadmin.local -q "ktadd -k /etc/security/keytabs/{keytabname}.keytab kafka/{hostname}@{REALM}"
2.3.1. 3.Make sure all hosts can be reachable using hostnames - it is a Kerberos requirement that all your hosts can be resolved with their FQDNs.

2.3. 2.Configuring Kafka Brokers
2.3.2. 1.Add a suitably modified JAAS file similar to the one below to each Kafka broker's config directory, let's call it kafka_server_jaas.conf for this example (note that each broker should have its own keytab):

```
KafkaServer {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  storeKey=true
  keyTab="/etc/security/keytabs/kafka_server.keytab"
```

```
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};

// Zookeeper client authentication
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  storeKey=true
  keyTab="/etc/security/keytabs/kafka_server.keytab"
  principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};
```

KafkaServer section in the JAAS file tells the broker which principal to use and the location of the keytab where this principal is stored. It allows the broker to login using the keytab specified in this section. See notes(2.1.7) for more details on Zookeeper SASL configuration.
2.3.2. 2.Pass the JAAS and optionally the krb5 file locations as JVM parameters to each Kafka broker (see here （https://docs.oracle.com/javase/8/docs/technotes/guides/security/jgss/tutorials/KerberosReq.html） for more details):

```
-Djava.security.krb5.conf=/etc/kafka/krb5.conf
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

2.3.2. 3.Make sure the keytabs configured in the JAAS file are readable by the operating system user who is starting kafka broker.
2.3.2. 4.Configure SASL port and SASL mechanisms in server.properties as described here. For example:

```
listeners=SASL_PLAINTEXT://host.name:port
security.inter.broker.protocol=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.enabled.mechanisms=GSSAPI
```

We must also configure the service name in server.properties, which should match the principal name of the kafka brokers. In the above example, principal is "kafka/kafka1.hostname.com@EXAMPLE.com", so: sasl.kerberos.service.name=kafka

2.3. 3.Configuring Kafka Clients
To configure SASL authentication on the clients: 1. Clients (producers, consumers, connect workers, etc) will authenticate to the cluster with their own principal (usually with the same name as the user running the client), so obtain or create these principals as needed. Then create a JAAS file for each principal. The KafkaClient section describes how the clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client using a keytab (recommended for long-running processes):

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
```

useKeyTab=true
storeKey=true
keyTab="/etc/security/keytabs/kafka_client.keytab"
principal="kafka-client-1@EXAMPLE.COM";
};

For command-line utilities like kafka-console-consumer or kafka-console-producer, kinit can be used along with "useTicketCache=true" as in: KafkaClient {
com.sun.security.auth.module.Krb5LoginModule required
useTicketCache=true;
};

2.Pass the JAAS and optionally krb5 file locations as JVM parameters to each client JVM (see here for more details):

-Djava.security.krb5.conf=/etc/kafka/krb5.conf
-Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf

3.Make sure the keytabs configured in the kafka_client_jaas.conf are readable by the operating system user who is starting kafka client.

2.3. 4.Configure the following properties in producer.properties or consumer.properties:

security.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka