Hadoop认证源码分析

Hadoop安全机制由认证、授权、审计等构成，其中认证是对一个实体的身份进行判断，确保用户是自己声称的用户，对用户身份进行验证是安全的核心。Hadoop的认证主要包括两个方面：

1）客户端与服务端的认证，例如客户端与NameNode交互获取数据信息、从DN中获取数据及写入数据、向YARN提交作业并进行作业的管理等。在Hadoop中，客户端与服务端的认证机制协议主要包括两个部分：

* Kerbeoros认证协议，使用Kerberos认证协议来建立一个安全的网络连接，之后客户端通过该连接从服务端获取共享密钥
* Token（令牌）认证协议，客户端通过上述获取到的共享密钥进行服务认证

Kerberos是一种基于可信任的第三方服务的认证机制，在高并发的情况下，效率较低。为了解决该问题，客户端一旦通过Kerberos和服务端之间建立安全网络连接后，就通过该连接从服务端获取一个秘钥，该秘钥仅有客户端和服务端知道，客户端并可使用该共享秘钥获取服务的认证，即基于授权令牌(Delegation Token)的认证机制。

1. 服务端与服务端的认证，Hadoop运行组件之间的认证，例如DataNode与NameNode之间的数据交互、NodeManager与ResourceManager之间的作业运行交互等，要防止非法程序伪装成DN/NM接收NN/RM的任务指派。服务端之间的认证完全基于Kerberos协议，对所有服务分发Keytab，服务组件之间使用私钥进行通信。

由于Hadoop认证过程比较复杂，这里仅以客户端向YARN提交数据处理作业为例来阐述客户端与服务端之间的认证过程。

# Hadoop Kerberos认证

上文已经大概介绍了Hadoop认证过程，不管是客户端与服务端，还是服务端之间的认证首先都要经过Kerberos认证建立安全连接通道。默认情况下，Kerberos认证机制是关闭的，管理员可以通过设置参数：

*<property>*

*<name>hadoop.security.authentication</name>*

*<value>kerberos</value>*

*</property>*

启动认证。下面是安全管理相关的术语：

* Kebrberos（Krb），基于第三方服务的认证协议，用户只需要输入一次身份验证信息就可以凭借此验证获得Ticket访问多个服务。Kerberos认证过程的实现不依赖于主机操作系统的认证，不基于主机地址的信息，也不要求网络上所有主机都是物理安全的。
* Principal，Hadoop集群中被认证或者授权的主体，主要包括用户、Hadoop服务、Container、Application、Localizer、Shuffle Data等
* KDC(Key distribution center)，是Krb的一个网络服务，提供ticket和临时会话秘钥
* AS(Authentication Server)，认证服务器
* TGS(Ticket Granting Server)，许可证服务器
* Ticket，客户端向服务器证明自己身份的记录，包括客户标识、会话秘钥、时间戳

使用Krb进行认证服务流程如下图所示：



客户端请求服务需要获取两个票据（Ticket），许可票据和服务票据：在许可票据中存放了客户端与KDC之间的加密信息，该信息只能通过客户端和KDC之间的密钥才能解密，主要过程如下：

1）客户端请求其他服务之前，TGS根据客户端发送过来的<许可票据，服务信息-服务名>信息，为客户端和服务之间提供Session Key，KDC将<SessionKey,用户,IP,服务,有效期,时间戳>包装成服务票据，服务票据是经过KDC与服务之间的密钥加密，客户端无法更改Ticket信息。

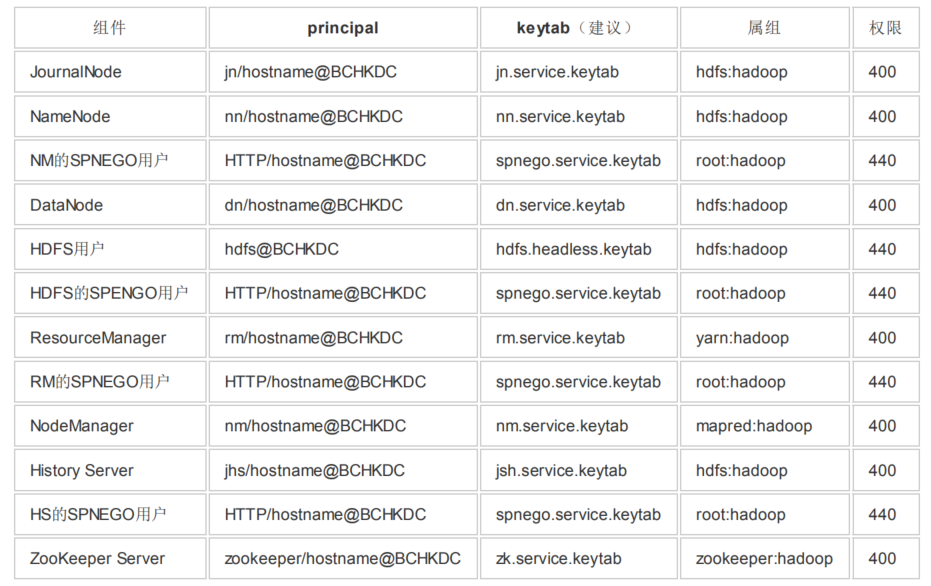
2）客户端请求服务时，向服务发送两类信息：

* + 服务票据，<SessionKey,用户,IP,服务,有效期,时间戳>
  + 客户端Authenticator信息，用服务票据中的SessionKey将<用户,IP>加密

3）服务用于KDC之间的密钥解析服务票据，获取SessionKey，然后将Authenticator信息解密，然后验证服务票据和Authenticator中的用户名、IP信息

## 服务端注册KDC（获取许可票据）

Hadoop开启Kerberos认证后，需要为Service和客户端创建Principal，通过Ambari启用Kerberos，服务与Principal的对应关系如下表所示：



启动之前，需要在KDC中创建相应的principal，命令格式如下：

*kadmin.local: adddprinc -randkey nn/\_HOST@BCHKDC*

-randkey参数表示随机生成密码。根据上述的principal，生成对应的keytab文件

*kadmin.lcoal: xst -k nnm.service.keytab nn/\_HOST@BCHKDC*

在nn.service.keytab中，保存了principal的密码等其他信息。服务端(NameNode)启动后，使用nn.service.keytab将服务注册到KDC中，源码如下所示：

*void loginAsNameNodeUser(Configuration conf) throws IOException {  
 InetSocketAddress socAddr = getRpcServerAddress(conf);  
 SecurityUtil.login(conf, DFS\_NAMENODE\_KEYTAB\_FILE\_KEY,  
 DFS\_NAMENODE\_KERBEROS\_PRINCIPAL\_KEY, socAddr.getHostName());*

*//<principal,keytab file,hostname>  
}*

执行流程图如下所示：



Server端获取Ticket Granting Ticket的用途主要是与KDC交互，验证客户端发送的Credential是否有效，这里的客户端包括DFS Client与DataNode等。

## 客户端获取服务访问许可

客户端访问服务首先要获取Ticket Granting Ticket（TGT），用于与KDC交互获取Service Ticket，其执行过程与Service相同，如上图所示，不再详述。客户端访问Service的执行流程图如下所示：



以DFSClient访问NameNode为例，在发起流程之前客户端已经获取到了许可票据

1. 客户端发起Rpc连接，通过RPC#Client创建连接

*Client#getConnection*

*connection = new Connection(remoteId, serviceClass);*

*boolean trySasl = UserGroupInformation.isSecurityEnabled() || //开启Kerberos后  
 (ticket != null && !ticket.getTokens().isEmpty()); //AuthProtocol为SASL  
 this.authProtocol = trySasl ? AuthProtocol.SASL : AuthProtocol.NONE;*

1. 通过IpcStream，开始Sasl认证流程

*private synchronized void setupIOstreams(AtomicBoolean fallbackToSimpleAuth) {*

*UserGroupInformation ticket = remoteId.getTicket();*

*....*

*while (true) {*

*setupConnection(ticket); //创建Connection连接*

*ipcStreams = new IpcStreams(socket, maxResponseLength);*

*writeConnectionHeader(ipcStreams);*

*if (authProtocol == AuthProtocol.SASL) {*

*try {*

*authMethod = ticket*

*.doAs(new PrivilegedExceptionAction<AuthMethod>() {*

*@Override*

*public AuthMethod run()*

*throws IOException, InterruptedException {*

*return setupSaslConnection(ipcStreams); //开启sasl认证流程*

*}*

*});*

*}*

*......*

*}*

* 客户端SaslRpcClient发送Negotiate给SaslRpcServer，Server端返回NEGOTIATE状态后，SaslRpcClient与KDC TGS通信获取Service Ticket

*return Sasl.createSaslClient(  
 new String[] { mechanism }, saslUser, saslProtocol, saslServerName,  
 saslProperties, saslCallback);*

其最终调用CredentialsUtil#acquireServiceCreds

*private static Credentials serviceCreds(*

*ServiceName service, Credentials ccreds)*

*throws KrbException, IOException {*

*return new KrbTgsReq(ccreds, service).sendAndGetCreds();*

*}*

* SaslRpcClient开启CHALLEGE过程，验证TOKEN是否合理，发送证书到Server端

*byte[] responseToken = saslEvaluateToken(saslMessage, false);*

* SaslRpcServer收到客户端发送的Token后，与KDC Admin模块交互，验证是否有效

*RPC.Server#processSaslToken*

*saslToken = saslServer.evaluateResponse(saslToken);*

*return buildSaslResponse(  
 saslServer.isComplete() ? SaslState.SUCCESS : SaslState.CHALLENGE,  
 saslToken);*

验证成功后将SUCCESS状态，发送Succuess状态给SaslRpcClient，客户端结束认证流程，

*case SUCCESS: {*

*saslEvaluateToken(saslMessage, true);*

*done = true;*

*break; }*

验证失败，则直接抛出RemoteExeption，拒绝客户端访问。

客户端认证成功，就可以向Server端发送访问请求。

# Token认证

经过第一部分的过程，客户端与服务端已经创建了安全的网络连接，客户端可以向服务端发起请求，在这个阶段客户端并不知道服务端的私钥，该私钥由Krb中的KDC进行管理，在交互过程中要通过Session key进行加密，防止私钥在网络中明文传送。

但是Kerberos认证协议是一种基于可信任的第三方服务的认证机制，在高并发的情况下，效率较低。为了解决该问题，客户端通过Krb认证协议和服务端之间建立安全网络连接后，客户端并通过该连接从服务端获取一个秘钥，该秘钥仅有客户端和服务端知道，客户端并可使用该共享秘钥获取服务的认证，即基于授权令牌(Delegation Token)的认证机制。

* Token，Token是基于共享秘钥的双方身份认证机制，会被加入到当前UGI对象中，并以Credential对象的形式加入到JAAS的Subject中，在UGI.doAs上下文中执行RPC调用时，Subject信息将被推送到线程上下文。Token的示例如下：

[

*Kind: HDFS\_DELEGATION\_TOKEN,*

*Service: 10.139.17.58:8020,*

*Ident: {*

*token for hdfs: HDFS\_DELEGATION\_TOKEN*

*owner=hdfs-yarn300@YARN.FED,*

*renewer=yarn,*

*realUser=,*

*issueDate=1537336866605,*

*maxDate=1537941666605,*

*sequenceNumber=67,*

*masterKeyId=14*

*},*

*Password: 41890903...*

]

* UserGroupInformation(UGI)，该类是在JAAS框架上封装了Hadoop用户信息，JAAS是Java 认证和授权服务（Java Authentication and Authorization Service）的缩写，主要包括以下几个实体：

1. Subject，实体类，标志一个请求的来源，包含相关的Principal和证书(Credential)
2. Principal，认证的主体
3. Credential，认证过程使用的证书（也可以称为凭据）

在Hadoop认证体系中，将UGI作为认证信息的载体在各组件之间进行传输。

Hadoop Token定义在org.apache.hadoop.security.token.Token中，其基本构成：

|  |  |  |
| --- | --- | --- |
| 字段名称 | 字段类型 | 含义 |
| identifier | byte[] | TokenIdentifier中将序列化的identity信息 |
| password | byte[] | TokenIdentifier中经序列化的password信息 |
| kind | TEXT | TokenIdentifier种类 |
| service | TEXT | TokenIdentifier应用到的服务类型 |
| renewerr | TokenRenewer | 由ServiceLoader类为这类TokenIdentifier加载的插件 |

每个Token存在一个唯一的TokenIdentifier标识。

Token类图如下所示：



当客户端使用令牌向另外一个服务获取认证时，经过的步骤如下：

1. 客户端将TokenIdentifier发送给服务器端，不同类型的Token，TokenIdentifier包含的字段是不一样的
2. 服务端使用TokenIdentifier和masterKey(秘钥，由Kerberos验证后获取，或者客户端向服务端注册时领取，并通过周期性心跳获取最新的masterKey)，重新计算TokenAuthenticator和Token
3. 服务端会检查新的Token是否合法，当Token在Server的内存中存在及当前时间在有效期内

*protected DelegationTokenInformation checkToken(TokenIdent identifier)*

*throws InvalidToken {*

*DelegationTokenInformation info = getTokenInfo(identifier); //在内存中存在*

*if (info == null) {*

*throw new InvalidToken("token " + formatTokenId(identifier)*

*+ " can't be found in cache");*

*}*

*long now = Time.now();*

*if (info.getRenewDate() < now) { //在有效期内*

*throw new InvalidToken("token " + formatTokenId(identifier) + " is " +*

*"expired, current time: " + Time.formatTime(now) +*

*" expected renewal time: " + Time.formatTime(info.getRenewDate()));*

*}*

*return info;}*

1. Token合法，则客户端和服务端将TokenAuthenticator作为密钥、DIGEST-MD5作为认证协议进行双方认证。其中Delegation Token用于NameNode为客户端进行认证，当客户端初始访问NameNode，如果通过Kerberos认证，则NameNode会为它返回一个密钥，之后客户端只需要借助该密钥便可以进行NameNode认证。为了访问重启密钥丢失，NameNode将各个客户端对应的密钥持久化保存到镜像文件中。默认情况下，所有密钥每个24小时更新一次，且NameNode总会保存前7小时的密钥以保证之前的密钥可用。

以MR的执行为例，从NameNode中获取密钥的操作如下：

*JobSubmitter#populateTokenCache {*

*...... //其中Credentials，是从KDC中获取的Service Ticket*

*TokenCache.obtainTokensForNamenodes(credentials, inputPaths, getConf());*

*}*

在这个过程中获取secret keys及token。

下面以客户端向ResourceManager提交Job为例，其执行过程如下所示：



1. 客户端经过Kerberos认证后，与NameNode进行Rpc通信，首先是获取DelegationToken，在NN端的执行如下：

*Token<DelegationTokenIdentifier> getDelegationToken(Text renewer)*

*throws IOException {*

*....*

*DelegationTokenIdentifier dtId = new DelegationTokenIdentifier(owner,*

*renewer, realUser);*

*token = new Token<DelegationTokenIdentifier>(*

*dtId, dtSecretManager);*

*long expiryTime = dtSecretManager.getTokenExpiryTime(dtId);*

*...*

*return token;*

*}*

其中dtSecretManager为DelegationTokenSecretManager，其生成Token Password，并加入其缓存，如下：

*byte[] password = createPassword(identifier.getBytes(), currentKey.getKey());  
DelegationTokenInformation tokenInfo = new DelegationTokenInformation(now  
 + tokenRenewInterval, password, getTrackingIdIfEnabled(identifier));  
...  
storeToken(identifier, tokenInfo); //currentTokens.put(ident,tokenInfo)*

客户端获取Token后，将其加入Credentials，并通过ApplicationSubmissionContext启动AM，初始化如下：

*DataOutputBuffer dob = new DataOutputBuffer();*

*credentials.writeTokenStorageToStream(dob);*

*ByteBuffer securityTokens =*

*ByteBuffer.wrap(dob.getData(), 0, dob.getLength());*

*List<String> vargs = setupAMCommand(jobConf);*

*ContainerLaunchContext amContainer = setupContainerLaunchContextForAM(*

*jobConf, localResources, securityTokens, vargs);*

1. 与RM交互获取资源，启动Application，在这个过程中涉及到JobToken的获取及流程，不再详述，RM收到请求后，将appContext中Token添加到DelegationTokenRenewer中，启动TimerTask定期与NN交互，进行token的renew，其源码在RMAppMaster#submitApplication中，其执行如下：

*this.rmContext.getDelegationTokenRenewer()  
 .addApplicationAsync(applicationId,  
 BuilderUtils.parseCredentials(submissionContext), //从context中获取credentials  
 submissionContext.getCancelTokensWhenComplete(),  
 application.getUser(),  
 BuilderUtils.parseTokensConf(submissionContext));*

在DelegationTokenRenewer中，有DelegationTokenRenewerRunnable负责token的renew.

1. RM与NM通信，启动AM，在这个过程中通过ApplicationMasterService进行launchAM，其中appContext直接传给NM，其具体信息如下：

*StartContainerRequest scRequest =*

*StartContainerRequest.newInstance(launchContext,*

*masterContainer.getContainerToken());*

1. NM接收到AM的start请求，将launchContext中的crendetials写入文件:

*HADOOP\_TOKEN\_FILE\_LOCATION <= nmPrivateTokenPath*

并将其添加到launch\_container.sh的环境变量中。YarnChild启动时，UGI直接从该文件中读取Token信息，如下：

*UGI#createRemoteUser() {*

*tokenFileLocation = conf.get(HADOOP\_TOKEN\_FILES);*

*if (tokenFileLocation != null) {*

*for (String tokenFileName:*

*StringUtils.getTrimmedStrings(tokenFileLocation)) {*

*if (tokenFileName.length() > 0) {*

*File tokenFile = new File(tokenFileName);*

*if (tokenFile.exists() && tokenFile.isFile()) {*

*Credentials cred = Credentials.readTokenStorageFile(*

*tokenFile, conf);*

*loginUser.addCredentials(cred);*

*}*

1. AM与RM交互，获取资源并与NM交互，这个过程中涉及到ContainerToken，不再详述，在ContainerLaunchContext中会添加上述的DelegationToken，如下：

*private void createMapTasks(JobImpl job, long inputLength,*

*TaskSplitMetaInfo[] splits) {*

*for (int i=0; i < job.numMapTasks; ++i) {*

*TaskImpl task =*

*new MapTaskImpl(job.jobId, i,*

*job.eventHandler,*

*job.remoteJobConfFile,*

*job.conf, splits[i],*

*job.taskAttemptListener,*

*job.jobToken, job.jobCredentials,*

*job.clock,*

*job.applicationAttemptId.getAttemptId(),*

*job.metrics, job.appContext);*

*job.addTask(task);*

*}*

*}*

Task的初始化过程中对Token的处理相同，通过HADOOP\_DELEGATION\_TOKEN\_FILE来传递

1. Task在执行过程中需要从NN中获取数据，则通过FileSystem使用DelegationToken来进行交互，其具体的执行过程如下:



SaslRpcServer接收到Token验证信息后，与DelegationTokenSecretManager交互，验证Token是否有效

*SaslDigestCallbackHandler#*

*if (pc != null) {  
 TokenIdentifier tokenIdentifier = getIdentifier(nc.getDefaultName(),  
 secretManager);  
 char[] password = getPassword(tokenIdentifier);*

*//*DelegationTokenSecretManager#checkToken *UserGroupInformation user = null;  
 user = tokenIdentifier.getUser(); // may throw exception  
 connection.attemptingUser = user;  
 pc.setPassword(password);  
}*

至此完成了认证过程，如果认证成功则服务端向客户端返回请求响应信息。

参考链接：

<https://issues.apache.org/jira/browse/HADOOP-6419>

https://issues.apache.org/jira/browse/HADOOP-4487

<http://www.nosqlnotes.com/technotes/distributed-system-with-kerberos/>

https://blog.cloudera.com/blog/2017/12/hadoop-delegation-tokens-explained/