Client-Server mutual authentication

This guide describes how to enable secure communication between client and server using SASL mechanism. ZooKeeper supports Kerberos or DIGEST as your authentication scheme.

JIRA and Source Code

This feature was added in ZooKeeper 3.4.0+ version and is available in all higher versions. ZOOKEEPER-938 is the JIRA issue, and the patch is available linked from that JIRA.

ZooKeeper ACLs and SASL

This proposed implementation builds on the existing ZooKeeper authentication and authorization design in a straightforward way. To briefly review, ZooKeeper supports pluggable authentication schemes. A node may have any number of <scheme:expression,perms> pairs. The left member of the pair specifies authentication as the authentication scheme and the principal. The right member indicates what permissions are given to this principal. For example, one ACL pair on a given node might be:

```<ip:19.22.0.0/16 , READ>```

The left side, ip:19.22.0.0/16, means that the authentication scheme is by Internet address, and that any client whose IPv4 address begins with "19.22" has whatever permissions are indicated on the right side. The right side indicates that the user the permissions "READ" on the given node.

The designated name of the SASL authentication scheme is simply "sasl", so if you are using Kerberos, you may set a ZooKeeper's node to be:

```<sasl:myclient@EXAMPLE.COM , READ>```

meaning that the client whose Kerberos principal is myclient@EXAMPLE.COM may read the given node.

ZooKeeper command differences

create

In non-SASL ZooKeeper, you may add authentication credentials when you create a node, for example, using org.apache.zookeeper.server.auth.DigestAuthenticationProvider, you would do:

```password```

# create a digest form of the password "password":
$ java -cp build/classes:build/lib/log4j-1.2.15.jar org.apache.zookeeper.server.auth.DigestAuthenticationProvider user:password
user:password->user:tpUq/4Pn5A64fV2yQ0gOJ8ZWqkY=

Then, after connecting to ZooKeeper, you would do the following to grant all permissions to the user "user" using password "password":

```create```

create /mynode content digest:user:tpUq/4Pn5A64fV2yQ0gOJ8ZWqkY=:cdrwa

With SASL ZooKeeper, the password generation depends on the mechanism (currently DIGEST-MD5 or Kerberos). How to set passwords for both mechanisms is described below in the Configuration section. Unlike with DigestAuthenticationProvider as shown above, with SASL, the create command does not include password information. Instead, (assuming your Kerberos domain is EXAMPLE.COM):

```create```

create /mynode content sasl:user@EXAMPLE.COM:cdrwa
addauth

The SASL authentication scheme differs from certain other schemes in that the "addauth <scheme> <auth>" command has no effect if scheme is "sasl". This is because authentication is performed using SASL-enabled token exchange immediately after connection, rather than occurring any time after connection, as addauth is.

addAcl

As with create, you do not include credential information. So whereas with the DigestAuthenticationProvider you would do:

```addAcl
addAcl /mynode digest:user:tpUq/4Pn5A64fV2yQ0gOJ8ZwqY=:cdrwa
```

with SaslAuthenticationProvider, you instead do:

```addAcl
addAcl /mynode sasl:user@EXAMPLE.COM:cdrwa
```

SASL and existing authProviders

You may continue to use existing ZooKeeper authentication providers, such as DigestAuthenticationProvider together with SaslAuthenticationProvider, if you wish. Existing unit tests that test existing authentication providers still pass and code that uses these authentication providers should also work.

org.apache.zookeeper.LoginThread

LoginThread is a new class that starts a new thread that periodically refreshes the javax.security.auth.Subject credentials, and is used for this purpose on both the ZooKeeper client and server. If ZooKeeper is configured to use Kerberos (see "Server Configuration" below for how to do this), both client and server should be configured to use a keytab or credential cache that the LoginThread will use to refresh the Subject's credentials.

ZooKeeper Client Modifications

org.apache.zookeeper.ZooKeeper

If the System Property java.security.auth.login.config is defined, the ZooKeeper constructor initializes its member variable org.apache.zookeeper.LoginThread loginThread:
LoginThread loginThread = null;

if (System.getProperty("java.security.auth.login.config") != null)
{
    // zookeeper.client.ticket.renewal defaults to 19 hours (about 80% of 24 hours, which is a typical
ticket expiry interval).
    loginThread = new LoginThread("Client",new ClientCallbackHandler(null),Integer.getInteger("zookeeper.
client.ticket.renewal",19*60*60*1000));
}

cnxn = new ClientCnxn(connectStringParser.getChrootPath(),
hostProvider, sessionTimeout, this,
watchManager,
    getClientCnxnSocket(), canBeReadOnly, loginThread);
cnxn.start();

As shown above, the loginThread is then passed to the ClientCnxn constructor, whose class is discussed in the next section.

org.apache.zookeeper.ClientCnxn

ClientCnxn's constructor has one new parameter: LoginThread loginThread. The above code fragment shows how the ZooKeeper object initializes ClientCnxn using this new parameter.

ClientCnxn uses the supplied loginThread object to initialize its saslClient member variable in the startConnect() method, which is called during ClientCnxn's run() loop when the client attempts to connect to a ZooKeeper Quorum server.

The loginThread object is also used to generate SASL tokens to send to the ZooKeeper server, as will be shown below in the code fragment showing the definition of prepareSaslResponseToServer().

When the ZooKeeper client connects to a ZooKeeper Quorum member, it creates a ClientCnxn as shown above, which in turn starts an EventThread to communicate with the quorum member. If SASL is enabled, then the client goes from CONNECTING to SASL_INITIAL. At this state, the client checks whether its saslClient should send an initial response (which is a SASL-internal detail that depends on the mechanism). If it should send an initial response, it creates the initial token and sends it to the ZooKeeper server and goes to state SASL. If not, it simply goes to state CONNECTED.

If, on the other hand, SASL is not configured on the client, then the client simply goes from SASL_INITIAL to CONNECTED state. This allows non-SASL authenticated ZooKeeper clients to interact without modification with a SASL-configured ZooKeeper Quorum.

While the client is in CONNECTED state, it exchanges tokens with the ZooKeeper server until authentication has succeeded or failed. If the former, it goes to AUTH_FAILED state. The token-exchange process on the client side is done using packets of type SaslServerResponseCallback (the definition of this class is shown below). We modify the ClientCnxn's event thread to support processing packets of type SaslServerResponseCallback:
class EventThread {
  .
  .
  run() {
    .
    .
    processEvent(event);
    .
    .
  }
}

private void processEvent(Object event) {
  .
  .
  Packet p = (Packet) event;
  .
  .
  if (p.cb instanceof ServerSaslResponseCallback) {
    ServerSaslResponseCallback cb = (ServerSaslResponseCallback) p.cb;
    SetSASLResponse rsp = (SetSASLResponse) p.response;
    cb.processResult(rc, null, p.ctx, rsp.getToken(), null);
  }
  .
  .
}

The `processResult()` called in the above has the following implementation:

```java
static class ServerSaslResponseCallback implements DataCallback {
  public void processResult(int rc, String path, Object ctx, byte[] data, Stat stat) {
    // data[] contains the ZooKeeper Server's SASL token.
    // ctx is the ClientCnxn object. We use this object's prepareSaslResponseToServer() method
    // to reply to the ZooKeeper Server's SASL token
    ClientCnxn cnxn = (ClientCnxn)ctx;
    byte[] usedata = data;
    if (data != null) {
      LOG.debug("ServerSaslResponseCallback(): saslToken server response: (length=" + usedata.length + ")");
    } else {
      usedata = new byte[0];
      LOG.debug("ServerSaslResponseCallback(): using empty data[] as server response (length=" + usedata.length + ")");
    }
    cnxn.prepareSaslResponseToServer(usedata);
  }
}
```

The `cnxn.prepareSaslResponseToServer()` called in the above is implemented as:
private byte[] saslToken = new byte[0];

public void prepareSaslResponseToServer(byte[] serverToken) {
    saslToken = serverToken;
    LOG.debug("saslToken (server) length: "+ saslToken.length);

    if (!(saslClient.isComplete() == true)) {
        try {
            saslToken = createSaslToken(saslToken, saslClient);
            if (saslToken != null) {
                LOG.debug("saslToken (client) length: "+ saslToken.length);
                queueSaslPacket(saslToken);
            }
            if (saslClient.isComplete() == true) {
                LOG.info("SASL authentication with ZooKeeper server is successful.");
            }
            } catch (SaslException e) {
                LOG.error("SASL authentication failed.");
            }
    }
}

Finally, createSaslToken is defined as follows (with some exception-handling code not shown):

Subject subject = this.loginThread.getLogin().getSubject();
if (subject != null) {
    synchronized(this.loginThread) {
        try {
            final byte[] retval = Subject.doAs(subject, new PrivilegedExceptionAction<byte[]>() {
                public byte[] run() throws SaslException {
                    try {
                        LOG.debug("ClientCnxn:createSaslToken(): ->saslClient.evaluateChallenge(len="+saslToken.length+")");
                        return saslClient.evaluateChallenge(saslToken);
                    }
                    catch (SaslException e) {
                        LOG.error("SASL authentication failed.");
                    }
                }
            }());
        }
    }
}

Note the use of the javax.security.auth.Subject subject in the above: this allows use of a Kerberos-authenticated ZooKeeper client to generate tokens that allow the ZooKeeper server to authenticate it, and also allows the client to authenticate the ZooKeeper server. Similar code exists on the server side, shown below.

ZooKeeper Server Modifications

When a client connects to the server, the server creates a javax.security.SaslServer object using its own authentication information derived from its startup configuration (see Configuration in the next section). This authentication information is used by the server's SaslServer object to exchange SASL tokens with the client's SaslClient object, as shown in the following code:
public SaslServer createSaslServer() {
    synchronized (loginThread) {
        Subject subject = loginThread.getLogin().getSubject();
        if (subject != null) {
            // server is using a JAAS-authenticated subject: determine service principal name and hostname
            if (subject.getPrincipals().size() > 0) {
                try {
                    final Object[] principals = subject.getPrincipals().toArray();
                    final Principal servicePrincipal = (Principal)principals[0];

                    // e.g. servicePrincipalNameAndHostname := "zookeeper/myhost.foo.com@FOO.COM"
                    final String servicePrincipalNameAndHostname = servicePrincipal.getName();
                    int indexOf = servicePrincipalNameAndHostname.indexOf("/");
                    // e.g. servicePrincipalName := "zookeeper"
                    final String servicePrincipalName = servicePrincipalNameAndHostname.substring(0, indexOf);
                    // e.g. serviceHostnameAndKerbDomain := "myhost.foo.com@FOO.COM"
                    final String serviceHostnameAndKerbDomain = servicePrincipalNameAndHostname.substring(indexOf+1, servicePrincipalNameAndHostname.length());
                    indexOf = serviceHostnameAndKerbDomain.indexOf("@");
                    // e.g. serviceHostname := "myhost.foo.com"
                    final String serviceHostname = serviceHostnameAndKerbDomain.substring(0, indexOf);

                    final String mech = "GSSAPI";
                    try {
                        return Subject.doAs(subject, new PrivilegedExceptionAction<SaslServer>() {
                            public SaslServer run() {
                                try {
                                    SaslServer saslServer;
                                    saslServer = Sasl.createSaslServer(mech, servicePrincipalName, serviceHostname, null, saslServerCallbackHandler);
                                    return saslServer;
                                } catch (SaslException e) {
                                    return null;
                                }
                            }
                        });
                    }
                }
            }
        }
    }
}
case OpCode.sasl: {
    // client sent a SASL token: respond with our own SASL token in response.
    LOG.debug("FinalRequestProcessor:ProcessRequest():Responding to client SASL token.");
    lastOp = "SASL";
    GetSASLRequest clientTokenRecord = new GetSASLRequest();
    ZooKeeperServer.byteBuffer2Record(request.request,clientTokenRecord);

    byte[] clientToken = clientTokenRecord.getToken();
    LOG.debug("Size of client SASL token: " + clientToken.length);
    byte[] responseToken = null;

    try {
        SaslServer saslServer = cnxn.saslServer;
        try {
            // note that clientToken might be empty (clientToken.length == 0):
            // in the case of the DIGEST-MD5 mechanism, clientToken will be empty at the beginning of the
            // SASL negotiation process.
            responseToken = saslServer.evaluateResponse(clientToken);
            if (saslServer.isComplete() == true) {
                cnxn.addAuthInfo(new Id("sasl",saslServer.getAuthorizationID()));
            }
        }
        catch (SaslException e) {
            LOG.warn("Client failed to SASL authenticate: " + e);
            if ((System.getProperty("zookeeper.maintain_connection_despite_sasl_failure") != null)
                &&
                (System.getProperty("zookeeper.maintain_connection_despite_sasl_failure").equals("yes"))) {
                LOG.warn("Maintaining client connection despite SASL authentication failure.");
            } else {
                LOG.warn("Closing client connection due to SASL authentication failure.");
                cnxn.close();
            }
        }
    } catch (Exception e) {
        LOG.warn("Unknown exception while processing request: " + e);
    }
}

Note that the server uses the existing ServerCnxn.addAuthInfo() function to record that a connection is authenticated, just as other Authentication Providers do.

Note also from the above that clients that fail SASL authentication will be immediately disconnected unless the system property zookeeper. maintain_connection_despite_sasl_failure is set to yes.

Server Configuration

conf/zoo.cfg

requireClientAuthScheme=sasl is optional: if it is set to any value, it will only allow non-authenticated clients to ping, create session, close session, or sasl-authenticate.

zoo.cfg

authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
#requireClientAuthScheme=sasl

conf/java.env

java.env

SERVER_JVMFLAGS="-Djava.security.auth.login.config=/path/to/server/jaas/file.conf"
The configuration file indicated by the system property `java.security.auth.login.config` should be similar to one of the following examples, depending on whether you are using DIGEST-MD5 or Kerberos as your authentication mechanism. In either case, the `Server` header is required.

**JAAS conf file: Kerberos authentication**

```java
JAAS configuration file, Kerberos mechanism

Server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/path/to/server/keytab"
    storeKey=true
    useTicketCache=false
    principal="zookeeper/yourzkhostname";
};
```

Note that the keytab file given in the `keyTab` section should not be readable by anyone other than the ZooKeeper server process user.

**JAAS configuration file: DIGEST-MD5 authentication**

```java
JAAS configuration file, DIGEST-MD5 mechanism

Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_super="adminsecret"
    user_bob="bobsecret";
};
```

Note that the passwords in the above are in plain text, so the JAAS configuration file should not be readable by anyone other than the ZooKeeper server process user.

**Client Configuration**

This is similar to the ZooKeeper server configuration, except there is no `zoo.cfg` for the client.

**conf/java.env**

```java
java

CLIENT_JVMFLAGS="-Djava.security.auth.login.config=/path/to/client/jaas/file.conf"
```

The configuration file indicated by the system property `java.security.auth.login.config` should be similar to one of the following examples, depending on whether you are using DIGEST-MD5 or Kerberos as your authentication mechanism. In either case, the `Client` header is required.

**JAAS conf file: Kerberos authentication**

```java
JAAS configuration file, Kerberos mechanism

Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/path/to/client/keytab"
    storeKey=true
    useTicketCache=false
    principal="yourzookeeperclient";
};
```
Note that the keytab file given in the keyTab section should not be readable by anyone other than the ZooKeeper client process user.

### JAAS configuration file: DIGEST-MD5 authentication

<table>
<thead>
<tr>
<th>JAAS configuration file, DIGEST-MD5 mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client {</td>
</tr>
<tr>
<td>org.apache.zookeeper.server.auth.DigestLoginModule required</td>
</tr>
<tr>
<td>username=&quot;bob&quot;</td>
</tr>
<tr>
<td>password=&quot;bobsecret&quot;;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Note that (as in the server configuration) the password in the above is in plain text, so the JAAS configuration file should not be readable by anyone other than the ZooKeeper client process user.

**Try it out for yourself!**

Setting up Kerberos and SASL with ZooKeeper is a complicated process for a beginner, so I’ve put detailed step-by-step instructions on [Up and Running with Secure ZooKeeper](#) to quickly get a simple Kerberos and SASLized ZooKeeper setup for your evaluation.