Geode Security Framework

Scope

Geode can be configured to authenticate peer system members, clients, and remote gateways. Geode also provides for authorization of cache operations on a server from clients. This allows to block unauthenticated access to a Geode Distributed System, or block cache operations as per user defined policies. When the peer system members are configured for authentication, then the Geode system is required to use the locator service for discovery.

The Geode authentication and authorization sub-system is implemented as a framework allowing applications to plug-in external providers such as LDAP, Kerberos, etc. We find that most enterprise customers typically have a common infrastructure such as single sign-on or centralized authentication systems they need to integrate with. Geode provides sample implementations based on LDAP and PKCS along with source code that users can adapt to suit their requirements or roll their own.

It is also possible to use SSL (Secure Sockets Layer) for "on-the-wire" confidentiality in Geode.

Note that authentication and entitlement management is provided for each VM that joins a Geode Distributed System, or for client/gateway processes. Each member node can only represent a single 'principal' or 'user'. Authentication at a connection level (support for multiple users in a single process) will be added in a future release.

We first walk through the authentication interfaces provided as part of security framework with simple implementations. Then configuring Geode to use those implementations is described. The sample implementations provided with Geode are described next. In the latter part, more advanced implementations like integration with a Kerberos realm, object-level authorization are provided.

Interfaces for authentication callbacks

AuthInitialize

The first requirement is obtaining credentials for a client/peer which is catered to by the AuthInitialize interface. This is required for a client or peer or gateway that needs to connect to a Geode Distributed System which has security enabled (more on enabling security in section "Configuration of authentication callbacks for peers").

```
public interface AuthInitialize extends CacheCallback {
  public void init(LogWriter systemLogger, LogWriter securityLogger)
      throws AuthenticationFailedException;
  public Properties getCredentials(Properties securityProps,
      DistributedMember server, boolean isPeer)
      throws AuthenticationFailedException;
}
```

The first method "init" in the interface is to initialize the callback with the given LogWriters. Usually a callback will use the securityLogger for logging purpose which is a special logger for security that will mark log lines with a "security-" prefix and the logs can also be configured to go to a separate log file. A sample implementation is below:

```
public class SampleAuthInit implements AuthInitialize {
  private LogWriter logger;
  public void init(LogWriter systemLogger, LogWriter securityLogger)
      throws AuthenticationFailedException {
    this.logger = securityLogger;
  }
```

The work of obtaining credentials for a VM is done by the "getCredentials" method.

The *securityProps* argument contains the set of Geode properties containing the prefix "security-". The server argument is for the host, port and other membership information of remote server (for the case of clients or gateways) or locator/peer (for the case of peer members) which will authenticate this member. The *isPeer* argument is passed as true when the remote member is a peer or locator, while it is false when the remote member is a server. This is useful if a member is both a peer in a Distributed System and client/gateway for another Distributed System and the member needs to use different credentials for the two.

A simple user/password implementation may expect "security-username" property to be set for the name of user, and "security-password" property to be set for the password (latter would normally be set programmatically) so all that is needed to be done is pass back the two properties as below:

```
public Properties getCredentials(Properties securityProps, DistributedMember server, boolean isPeer)
 throws AuthenticationFailedException {
 Properties credentials = new Properties();
 String userName = securityProps.getProperty("security-username");
 credentials.setProperty("security-username", userName);
 String passwd = securityProps.getProperty("security-password");
 credentials.setProperty("security-password", passwd);
 logger.info("SampleAuthInit: successfully obtained credentials for user " + userName);
 return credentials;}
```

Adding a few checks in the above code, our sample implementation looks like below:

```
package com.gemstone.samples;
import java.util.Properties;
import com.gemstone.gemfire.LogWriter;
import com.gemstone.gemfire.distributed.DistributedMember;
import com.gemstone.gemfire.security.AuthInitialize;
import com.gemstone.gemfire.security.AuthenticationFailedException;
public class SampleAuthInit implements AuthInitialize {
  private LogWriter logger;
  public static final String USER_NAME = "security-username";
  public static final String PASSWORD = "security-password";
  public static AuthInitialize create() {
    return new SampleAuthInit();
  }
  public void init(LogWriter systemLogger, LogWriter securityLogger)
      throws AuthenticationFailedException {
    this.logger = securityLogger;
  }
  public Properties getCredentials(Properties securityProps,
      DistributedMember server, boolean isPeer)
      throws AuthenticationFailedException {
    Properties credentials = new Properties();
    String userName = securityProps.getProperty(USER_NAME);
    if (userName == null) {
      throw new AuthenticationFailedException(
          "SampleAuthInit: user name property [" + USER_NAME + "] not set.");
    }
    credentials.setProperty(USER_NAME, userName);
    String passwd = securityProps.getProperty(PASSWORD);
    if (passwd == null) {
      throw new AuthenticationFailedException(
          "SampleAuthInit: password property [" + PASSWORD + "] not set.");
    }
    credentials.setProperty(PASSWORD, passwd);
    logger.info("SampleAuthInit: successfully obtained credentials for user "
        + userName);
    return credentials;
  }
  public void close() {
  }
}
```

The "close" method in the above implementation comes from the *CacheCallback*interface that *AuthInitialize* extends. The static "create" method is used to create an instance of the interface which is used for registration of the callback.

Authenticator

Next is the *Authenticator* interface that is required to be implemented on a server/peer/locator that will authenticate a new client or peer member. This callback is provided the credentials of the joining member as a set of properties as obtained from*AuthInitialize#getCredentials* on the member.

```
public interface Authenticator extends CacheCallback {
 public void init(Properties securityProps, LogWriter systemLogger,
 LogWriter securityLogger) throws AuthenticationFailedException;
 public Principal authenticate(Properties props, DistributedMember member)
 throws AuthenticationFailedException;
}
```

This has a couple of methods (apart from the "close" method inherited from*CacheCallback* interface). The first one "init" is used to perform any initialization of the callback and provided LogWriters useful for logging. The *securityProps* argument provides all the Geode properties of this member that start with the prefix "security-". For our sample implementation we will again use the *securityLogger* provided.

```
 public class SampleAuthenticator implements Authenticator {
  private LogWriter logger;
  public void init(Properties securityProps, LogWriter systemLogger,
      LogWriter securityLogger) throws AuthenticationFailedException {
    this.logger = securityLogger;
  }
```

The "authenticate" method is the guts of callback that authenticates the client or peer member. It is provided the credentials of the joining member as a set of *Properties* in the*props* argument. This is the same set of properties that have been returned by the*AuthInitialize#getCredentials* on the member. Lastly the *member* argument provides the membership information of the joining client or peer member.

Continuing with the simple username/password based authentication example, our sample authenticator uses a JAAS username/password implementation to verify the credentials of the user. Firstly we need a *CallbackHandler* for JAAS that will just provide the username/password provided in the properties to the JAAS *LoginModule*. This is handled by the *SampleCallbackHandler* class below.

```
 package com.gemstone.samples;
import java.io.IOException;
import java.security.Principal;
import java.util.Properties;
import java.util.Set;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import com.gemstone.gemfire.LogWriter;
import com.gemstone.gemfire.distributed.DistributedMember;
import com.gemstone.gemfire.security.AuthenticationFailedException;
import com.gemstone.gemfire.security.Authenticator;
public class SampleAuthenticator implements Authenticator {
  private LogWriter logger;
  private String jaasEntry;
  private LoginContext currentContext;
  public static final String JAAS_ENTRY = "security-jaas-entry";
  public static Authenticator create() {
    return new SampleAuthenticator();
  }
  public void init(Properties securityProps, LogWriter systemLogger,
      LogWriter securityLogger) throws AuthenticationFailedException {
    this.logger = securityLogger;
    this.jaasEntry = securityProps.getProperty(JAAS_ENTRY);
  }
  public Principal authenticate(Properties props, DistributedMember member)
      throws AuthenticationFailedException {
    SampleCallbackHandler callbackHandler = new SampleCallbackHandler(props);
    try {
      this.currentContext = new LoginContext(this.jaasEntry, callbackHandler);
    } catch (LoginException ex) {
      throw new AuthenticationFailedException("SampleAuthenticator: failed "
          + "in creation of LoginContext for JAAS entry: "
          + this.jaasEntry, ex);
    }
    try {
```

```
      this.currentContext.login();
    } catch (LoginException ex) {
      throw new AuthenticationFailedException("SampleAuthenticator: "
          + "authentication failed for JAAS entry: " + this.jaasEntry, ex);
    }
    Set<Principal> principals = this.currentContext.getSubject()
        .getPrincipals();
    // assume only one Principal
    if (principals == null || principals.size() != 1) {
      throw new AuthenticationFailedException("SampleAuthenticator: expected "
          + "one Principal but got: " + principals);
    }
    logger.info("SampleAuthenticator: successfully authenticated member: "
        + callbackHandler.userName);
    return principals.iterator().next();
  }
  public void close() {
  }
  class SampleCallbackHandler implements CallbackHandler {
    private final String userName;
    private final String password;
    public SampleCallbackHandler(Properties props) {
      this.userName = props.getProperty(SampleAuthInit.USER_NAME);
      this.password = props.getProperty(SampleAuthInit.PASSWORD);
    }
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
      for (Callback callback : callbacks) {
        if (callback instanceof NameCallback) {
          ((NameCallback)callback).setName(this.userName);
        }
        else if (callback instanceof PasswordCallback) {
          ((PasswordCallback)callback).setPassword(this.password.toCharArray());
        }
      }
    }
  }
}
```

The static "create" method is used to create an instance of the interface which is used for registration of the callback. Next, the configuration of the above two callbacks is discussed.

Configuration of authentication callbacks for peers

As mentioned before, a Geode Distributed System must use locators for discovery for peer security i.e. multicast discovery is incompatible with peer security settings discussed below.

A peer attempting to join a secure distributed system presents its credentials to one of the authenticated locators. The first locator to join the Distributed System is assumed to be authenticated and subsequent locators authenticate against the first one. The list of locators is obtained from the "locators" Geode property. The credentials obtained from the *AuthIntialize#getCredentials* method is sent to one of the locators for authentication. The *security-peer-auth-init* property should be set to the name of a zero argument static method that returns an *AuthInitialize* object on the members while the *security-peer-authenticator* property should be set to the name of zero argument static method that returns an Authenticator object on the members and locators. Note that since the members also authenticate the VIEW messages sent out, so all members also need to be configured with the Authenticator in addition to the locators.

The settings required for the above example implementations are:

security-peer-auth-init – com.gemstone.samples.SampleAuthInit.create

security-username – a valid user name

security-password – password for the above user

security-peer-authenticator – com.gemstone.samples.SampleAuthenticator.create

_security-jaas-entry _– entry name in JAAS configuration file to use for authentication

The JAAS configuration file should be provided using the normal "java.security.auth.login.config" System property. These need to be set on all the peer members and locators of the Distributed System. Sample code to do this programmatically is below:

```
Properties props = new Properties();
props.setProperty("security-peer-auth-init",
    "com.gemstone.samples.SampleAuthInit.create");
props.setProperty(SampleAuthInit.USER_NAME, "user1");
props.setProperty(SampleAuthInit.PASSWORD, "xxx");
props.setProperty("security-peer-authenticator",
    "com.gemstone.samples.SampleAuthenticator.create");
props.setProperty("security-jaas-entry", "Sample");
DistributedSystem sys = DistributedSystem.connect(props);
```

If authentication for a peer member or locator fails, then the DistributedSystem.connect() method throws an *AuthenticationFailedException*. If the locators /peers have the *security-peer-authenticator* property set but the members do not have the *security-peer-auth-init*property set, then an *AuthenticationRequiredException* is thrown. All security exceptions have *GemFireSecurityException* as the base class so user code can choose to catch the base class exception where required.

Configuration of authentication callbacks for clients and servers

A client is authenticated for each handshake it initiates with a Geode cache server i.e. for each TCP connection from client to server. The client passes its credentials during the handshake and the server uses them to authenticate the client. The client must trust all the cache server host:bind-address[port] pairs in its endpoints list, since the client could connect to any server in the list and pass along its credentials. The credentials obtained from the *AuthInitialize#getCredentials* method is sent to the servers for authentication. The *security-client-auth-init* property should be set to the name of the zero argument static method that returns an *AuthInitialize* object on all the clients while the *security-client-authenticator* property should be set to the name of zero argument static method that returns an *Authenticator* object on all the servers.

The settings required for the above example implementations are:

For clients:

security-client-auth-init – com.gemstone.samples.SampleAuthInit.create

security-username – a valid user name

security-password – password for the above user

For servers:

security-client-authenticator – com.gemstone.samples.SampleAuthenticator.create

security-jaas-entry – entry name in JAAS configuration file to use for authentication

As before the JAAS configuration file should be provided using the normal "java.security.auth.login.config" System property. These need to be set on all the peer members and locators of the Distributed System. Sample code to do this programmatically is below:

For clients:

```
Properties props = new Properties();
props.setProperty("security-client-auth-init",
    "com.gemstone.samples.SampleAuthInit.create");
props.setProperty(SampleAuthInit.USER_NAME, "user1");
props.setProperty(SampleAuthInit.PASSWORD, "xxx");
DistributedSystem sys = DistributedSystem.connect(props);
```

For servers:

```
Properties props = new Properties();
props.setProperty("security-client-authenticator",
"com.gemstone.samples.SampleAuthenticator.create");
props.setProperty("security-jaas-entry", "Sample");
DistributedSystem sys = DistributedSystem.connect(props);
```

Unlike for peers, the authentication of clients is performed for each client-server connection that are created dynamically during Region operations. If authentication for a client fails, then the Region API method that requires to go to the server throws an*AuthenticationFailedException*. If the servers have the *security-client-authenticator*property set but the clients do not have the *security-client-auth-init* property set, then an*AuthenticationRequiredException* is thrown by the Region API methods.

Interface for authorization callbacks

Authorization for cache operations is currently provided for clients that should first authenticate to the server as above. Once a client has authenticated to a server as above, the *Principal _object returned by _Authenticator#authenticate* method is associated to the client. This is then passed on to the authorization callback on the server, if any.

There are two places where authorization of cache operations can be performed: one in the pre-operation phase before an operation is performed, and second in the post-operation phase after the operation is complete on the server and before sending result back to the client (for get/query kind of operations that return a result). In addition, notifications sent to clients by servers are also authorized in the post-operation phase.

```
public interface AccessControl extends CacheCallback {
 public void init(Principal principal, DistributedMember remoteMember,
 Cache cache) throws NotAuthorizedException;
 public boolean authorizeOperation(String regionName, OperationContext context);
}
```

The "init" method is invoked to initialize the callback for a client. The *Principal* object obtained for the authenticated client (result of *Authenticator#authenticate* method) is passed as the first argument to the method. Membership information for the client is provided in the remoteMember argument, while the Geode Cache is passed as the third argument. The "authorizeOperation" method is invoked for each client cache operation on the authenticated connection. It is provided the region name of the operation and an*OperationContext* object that encapsulates information of the current cache operation.

Continuing with our username/password example, we assume that the JAAS authentication module in above sample implementations returns a *Principal* (*MyPrincipal*class) that provides an "isReader" method that will return true if the member should be given read-only permissions, while other members have all permissions. The code for pre-authorization callback is below:

```
package com.gemstone.samples;
import java.security.Principal;
import com.gemstone.gemfire.cache.Cache;
import com.gemstone.gemfire.cache.operations.OperationContext;
import com.gemstone.gemfire.cache.operations.OperationContext.OperationCode;
import com.gemstone.gemfire.distributed.DistributedMember;
import com.gemstone.gemfire.security.AccessControl;
import com.gemstone.gemfire.security.NotAuthorizedException;
public class SampleAccessControl implements AccessControl {
 private boolean isReader;
 public static AccessControl create() {
 return new SampleAccessControl();
 }
 public void init(Principal principal, DistributedMember remoteMember,
 Cache cache) throws NotAuthorizedException {
 if (principal instanceof MyPrincipal) {
 this.isReader = ((MyPrincipal)principal).isReader();
 }
 else {
 this.isReader = false;
 }
 }
 public boolean authorizeOperation(String regionName, OperationContext context) {
 if (this.isReader) {
 OperationCode opCode = context.getOperationCode();
 // these cache operations do not modify data
 return (opCode.isGet() || opCode.isQuery() || opCode.isContainsKey()
 || opCode.isKeySet() || opCode.isRegisterInterest()
 || opCode.isUnregisterInterest() || opCode.isExecuteCQ()
 || opCode.isCloseCQ() || opCode.isStopCQ());
 }
 else {
 return true;
 }
 }
 public void close() {
 }
}
```

The "init" method caches the "isReader" flag for the client since the *isReader* method may be potentially expensive. The "authorizeOperation" method then allows read-only operations when the "isReader" flag is true.

Configuration of authorization callbacks

The setting required on the servers for the above example implementation is:

security-client-access-control – com.gemstone.samples.SampleAccessControl.create

If the authorization for a cache operation is denied by the server (i.e. *authorizeOperation*method on server returns false), then the client receives a *NotAuth orizedException* for the operation.