

Overview of Secure Controls Framework Implementation

Version 1. February 23, 2004.
A. D. Petrov, apetrov@fnal.gov

The Secure Controls Framework (SCF) was proposed in [1] as a way to improve the communication between Java™ controls applications and server-side data providers over RMI. This document gives an overview of the core SCF implementation, existed now.

Architecture

SCF consists of:

1. Client tier, which initiates and maintains RMI connections to the server and provides the high-level API.
2. Data server, which accepts the incoming connections and handles data requests. It can be started either standalone, or along with the Data Acquisition Engine (DAE).
3. Kerberos module, which provides Kerberos V5 user authentication.

Client and server parts communicate over RMI, Remote Method Invocation protocol. SSL/TLS is used by RMI as a secure transport level for client→server calls.

In order to be authenticated by SCF, the user must obtain the Kerberos ticket on the local host by using an external utility, *kinit* or *Leash32*.

Implementation

All parts are implemented on J2SE version 1.4. The Kerberos module has some small native program used to access the memory cache. The code is located in the following packages:

Table 1. SCF code packages

<i>Package</i>	<i>Description</i>	<i>Required on the client</i>
<code>gov.fnal.controls.↵ scf</code>	Implementation of the client-side API	✓
<code>gov.fnal.controls.↵ scf.example</code>	Test utilities and examples	

gov.fnal.controls.← scf.items	Data acquisition support	✓
gov.fnal.controls.← scf.more	Miscellaneous Swing components	✓
gov.fnal.controls.← scf.naming	Naming service support	✓
gov.fnal.controls.← scf.remote	Client-server communication support	✓
gov.fnal.controls.← scf.server	Implementation of services, and supporting classes	
gov.fnal.controls.← servers.scf	Server loader and web manager	
gov.fnal.controls.← tools.kerberos.base	Kerberos module	✓

The client tier does not require any additional libraries. The server needs gov.fnal.controls.db and jconn2.jar in the *classpath* for the database access.

Kerberos Module

This module provides Kerberos V5 user authentication.

In SCF, the user authentication is initiated by the server and consists of three steps:

1. The client reads the cached Kerberos ticket and creates a `Subject` representing the user.
2. Then, it prepares a token (equivalent of the forwarded ticket) in order to pass user credentials to the server.
3. The server accepts the token and determines the remote user name.

The Kerberos module API is implemented in `Krb5` class. It has five static methods:

```
public static Subject login()
public static void logout()
public static String getPrincipalName()
```

```

public static byte[] initGSSContext(
    String serverPrincipal
)
public static GSSContext acceptGSSContext(
    String serverPrincipal, byte[] context
)

```

The `login` method starts the authentication. It tries to acquire credentials of the local user or process, and, if successful, returns a `Subject` associated with it. The `logout` method clears the internal variables and resets the `Subject`. The implementation of `login` and `logout` methods is based on Java Authentication and Authorization Service (JAAS) API [2].

Krb5 supports two types of authentication: user authentication and service authentication.

By default, if no additional settings were made, the Kerberos module performs the *user authentication*. Krb5 creates an instance of `FermiLoginModule` that reads the ticket from a cache. Technically, the file cache is read by the `CacheReader`*; on Windows, the memory cache is loaded through Java Native Interface (JNI) by `WinCacheAdapter`.

Table 2. Cache reader compatibility

<i>Operating System</i>	<i>Cache Locations</i>
Windows	API: FILE:<user-tmp-dir>\krb5cc
SunOS, FreeBSD, Linux	\$KRB5CCNAME FILE:/tmp/krb5cc_<uid>
MacOS†	FILE:/tmp/krb5cc_<uid>

In order to accept the incoming tokens, a process running on the server must be authenticated with the service principal. This is the *service authentication*. The service does not have cached credentials. It reads the Kerberos password from a keytab, presents this password along with the principal name to the Kerberos server, and acquires the credentials. In order to perform the service

* The generic analog, `sun.security.krb5.internal.ccache.FileCredentialsCache`, apparently has a bug in the integer format representation that prevents it from being used on FreeBSD.

† There is also a memory cache in OS X, but the access to it is not yet implemented.

authentication, the keytab file name and the service principal name must be specified as system properties (see `Krb5` JavaDoc for details).

The `Krb5`'s `initGSSContext` and `acceptGSSContext` methods are used to establish a common security context between two peers using locally acquired credentials. The first one is called on the client side. It generates a token that can be sent to the remote host in order to present the local user. The `acceptGSSContext` method accepts the incoming token on the remote host.

The implementation of `initGSSContext` and `acceptGSSContext` methods is based on Java Generic Security Service API (JGSS-API) [3]

Tests have shown that:

- The ticket can be accepted only by the service, whose principal name was specified in `initGSSContext` method.
- The ticket is valid in a limited period of time (around couple of minutes).
- The ticket is not reusable and can not be presented more than once.

Some Authentication Issues

The clients running in the Control rooms can not be authenticated by using cached Kerberos tickets, because those workstations do not have it. In this case, an alternative way of authentication by IP address is used.

The list of exempt hosts is stored in the database along with corresponding user names. Before requesting Kerberos user authentication, the server checks whether the client is running on an exempt node, and, if so, skips the explicit authentication procedure. The same mechanism is used in Application Index web interface [5].

As stated in [1], all clients are assigned to 5 logical zones, according to their IP address. A security policy (authentication lifetime, presence of exempt host, etc.) is defined for each zone. The zone support and exempt host authentication are implemented in `ZoneConfig.class`. More security policy attributes will be added as necessary.

Client Tier

The principal class of the SCF client tier is `DataConnection`. This is a singleton without public constructors. The shared instance of the class can be obtained through `getInstance` static method. The API provided by `DataConnection` allows the user to control the connection (`connect` and `disconnect` methods) and check the connection status (`getStatus` and `isAlive` methods). In order to be notified about connection status changes, the `ConnectionStatusListener` can be used, as well. The `DataConnection`

class also provides a way for other client-side parts to communicate with the data server (e.g., `getNamingFactory` method for JNDI).

The `DataConnection` class is able to restore the connection if the server becomes unavailable. The connection timeout can be set by `setTimeout` method. Most of the underlying client requests do not depend on whether the server is physically available in a specific moment; they are cached until the communication is restored.

Before opening the connection, the remote object `ClientContextImpl` is created and exported on a random TCP/IP port. This object is used by the server to get client properties and to start the authentication. Besides that, the client creates a random session ID (by a secure random generator) for every connection.

Data Server

The SCF data server has a pluggable architecture. The main class, `gov.fnal.controls.servers.scf.ScfServer` works as a loader and starts some number of services. All services are described in a XML configuration file. The default file name is `gov/fnal/controls/servers/scf/config.xml`. It can be overwritten in system properties.

Most of the services are remote RMI objects. Each of them is responsible for a particular task and is exported on a fixed TCP/IP port.

- `CheckpointImpl` is responsible for checking the clients in and out, authentication, and managing the connection pool.
- `StorekeeperImpl` provides database access for the client-side JNDI. Currently supported objects are: ACNET devices, parameter pages, and a Universal Repository of Serialized Objects.
- `ForemanImpl` is a bridge between the Data Acquisition Engine and client-side DAQ API (under development).

One service, the `WebManager`, is not an RMI object. It implements a simple web-server, which provides the user interface for monitoring and restarting.

The server activity logging is implemented on Java Logging API.

Transport Level

SCF is using both secured and unsecured connections in different situations:

- Secured connection is used for remote calls originated from the client.
- Unsecured connection is used for the remote calls originated from the server, and for RMI repository calls.

Secured connection is based on SSL/TLS protocol with authentication. It is implemented in `SecureClientSocketFactory` and `SecureServerSocketFactory`. The first class is used on both server and client, the second one—only on the server. `SSLCiphers.class` defines the used cipher suites.

The transport level authentication is based on X.509 certificates. The server-side certificate (contains both private and public keys) is saved in a Java keystore. This keystore (a file) and the corresponding keystore password are stored in a safe location, defined in the server configuration file. The client-side certificate (public key only) is distributed as a file, `gov/fnal/controls/scf/remote/scf.cer`, available in application classpath. If applications are started from a properly signed jar files, `scf.cer` does not necessarily have to be signed by a certification authority and can be self-issued.

Connection Scenario

The following diagram shows up the connection procedure:

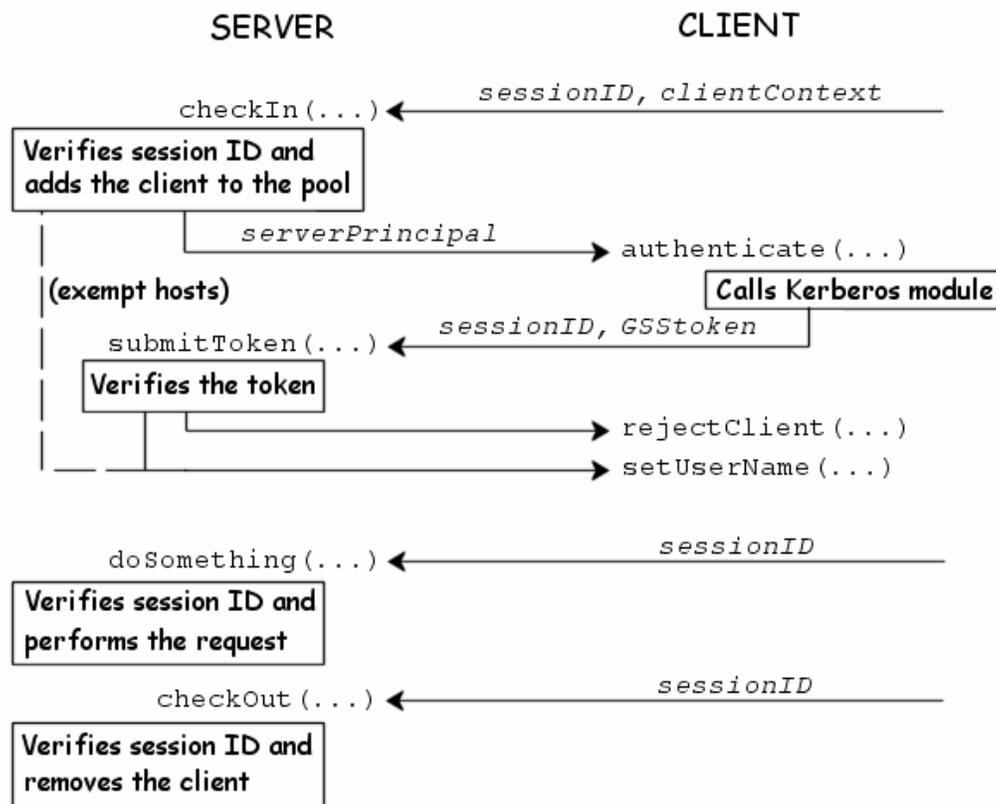


Fig. 1. Connection scenario

1. The client exports a `ClientContext` instance and creates a random session ID.
2. The client calls `checkpoint.checkIn` with the `clientContext` and session ID as parameters.
3. The server makes sure that:
 - a. The request originates from the same IP address as specified in the `clientContext` object;
 - b. The connection pool does not have another client with the same session ID.
4. If these tests succeed, the server accepts the client and adds it in the connection pool; otherwise a `SecurityException` is thrown.
5. The client gets authenticated:
 - a. If the client is running on an exempt host, the user name is taken from the database.
 - b. Otherwise, the server calls `clientContext.authenticate`; the client creates the token and calls `checkpoint.submitToken`; the server verifies this token.
 - c. If the authentication succeeds by either way, the server notifies the client by calling `clientContext.setUserName`; otherwise, the server calls `clientContext.rejectClient` and remove this connection from the pool.
6. After the client is authenticated, it is allowed to use other services (remote objects). The session ID is passed every time as one of the remote method call arguments and verified by the server.
7. In order to disconnect, the client calls `clientContext.checkOut`.
8. A routine on the server side periodically checks all connected clients by calling `clientContext.getTime`. If the client fails to respond several times, it gets disconnected. If the client authentication has expired, the client gets reauthenticated.

Authorization Issues

The methods of the user authorization depend on services.

The `StorekeeperImpl` gets permissions from custom database tables. Read-only access to ACNET devices and parameter pages is allowed for all authenticated users; writing is prohibited. The Universal Repository of Serialized Objects has more flexible UNIX-like security system, where permissions and the object owner are represented as `DirContext`'s attributes.

The `ForemanImpl` relies on the DAE security system and does not perform the authorization itself. The Kerberos user principal is translated to the VMS user name and, along with node and service attributes, is used by the engine. The DAE is using the existing mechanisms to define the user, node, and service privileges.

Analysis of Security

The ultimate goal of the SCF security system is to prevent unauthorized changes of data, namely: making DAQ settings and altering objects stored in the database. The information traveling in the opposite direction, from the server to the user, is not sensitive and it is unlikely that it can be targeted.

SCF does not address denial-of-service attacks. This problem has to be solved rather by deploying of several groups of servers, where each group is assigned to serve some specific IP zone.

As appears from the previous chapters, the connection security is based on 3 points:

1. Session ID, used in all remote calls;
2. GSS token that represents user credentials;
3. Verification of IP address.

There is no evidence, so far, that the current implementation of RMI protocol has any particular vulnerability. Also, there is no evidence against Kerberos. Theoretically, the potential intruder has the following options:

1. Using someone else's session ID:
 - a. Steal an existing session ID of the authenticated user (either by intercepting the communication and breaking SSL, or by another way);
 - b. Change somehow the own IP address and use the stolen session ID to connect from behalf of the other user.

Method of protection: keeping session ID in secrecy.

2. Replacing the server:

- a. Steal the SCF source code and make certain changes in server and client parts that do not affect remote interfaces;
- b. Starts the fake server;
- c. Somehow redirect SCF requests to the own server;
- d. Acquire the GSS token from the innocent client;
- e. Connect to the real server and present this token.

Method of protection: transport level authentication.

References

4. A. D. Petrov. Proposals on a Secure RMI Connection for Client Applications. Fermilab Beams-doc-953. <http://beamdocs.fnal.gov/cgi-bin/public/DocDB/ShowDocument?docid=953&version=1>
5. Java™ Authentication and Authorization Service (JAAS) Reference Guide. <http://java.sun.com/j2se/1.4.1/docs/guide/security/jaas/JAASRefGuide.html>
6. M. Upadhyay, R. Marty. Single Sign-on Using Kerberos in Java. Sun Microsystems, Inc. <http://java.sun.com/j2se/1.4.1/docs/guide/security/jgss/single-signon.html>
7. L. Gong, G. Ellison, M. Dageforde. Inside Java™ 2 Platform Security, Second Edition. Addison-Wesley ISBN 0-201-78791-1. p. 262.
8. Application Index web interface. <http://www-bd.fnal.gov/appix>
9. R. Lee. The JNDI Tutorial. <http://java.sun.com/products/jndi/tutorial>