Coda Authentication and Protection

Peter Braam

Contents

1 Coda Security

In this section we will describe the Coda security mechanisms as they are currently implemented. The basic reference for this material is the paper "Integrating Security in a Large Distributed System", (M.

Satyanarayanan, ACM Transactions on Computer Systems, Vol 7., N. 3. August 1989, Pages 247- 280.) There are a number of areas where the current system could fairly easily be improved and we will point out these areas.

Basically the security in Coda falls into two parts:

1. **Authentication and secure connections**: Coda's RPC2 package has a mechanism for securily authenticating clients to servers and vice versa and setting up encrypted channels between them. A key element of this scheme is passwords for Coda users.
2. **Access control and protection datbases**: The files on Coda servers are protect with directory access control lists. Such lists grant permissions to users or groups and the structure of these is stored in a Coda protection database.

The control of access to files and directories is simpler. Access control is established only after and authenticated connection has been established. It then uses the vice.pcf file in conjunction with the ACL's to determine permissions. The file vol.tk is the shared secret between the volutil clients and the volutil server routines inside srv. This token is used to establish and authenticated connection to the server from a volutil client.

Note: It might make sense to allow a system administrator to get volutil priviliges through a Coda token.

1.1 Vice Configuration Files and authentication

In terms of configuration files on the server, the authentication process uses several:

1. auth2.tk: the shared secret between the file server and the authentication server.
2. the flag "dreuss " given to initpw is used to encrypt passwords to prevent accidental disclosure
3. auth2.pwd contains the (uid, encrypted password pairs) using the "dreuss " key
4. vice.pcf, the binary equivalent of vice.pdb is used to look up uid's from user names

We will now proceed to describe the two components for security.

2 Authentication and secure connections.

The system of authentication and encryption of data going between clients and servers is very close to that employed by Kerberos 5, see ???.

The key ingredients in the mechanisms are as follows:

- *Shared secret cryptography*: in the protocols it is assumed in numerous places that the two *principals* on a connection share a "secret" key, allowing the sender to encrypt data and the receiver to decrypt it. The acquisition of shared keys is involved since they cannot be sent over the net in the clear.
- *Establishing secure connections*: this comprises a mechanisms to establish a secure connections if the two parties share a secret key and a public `ClientIdent`. Secure means *authenticated* and *encrypted*.

- *Authentication protocol*: this protocol is a variant of the Needham Schroeder authentication protocol. It allows the generation of a shared secret, called a session or handshake key for use between client and fileserver based on communication with an *authentication server*. The result of the an authetication session is that the client acquires a session key on the basis of supplying a correct password. This key is initially only known to the authentication server and client.
- The authentication server and the fileserver share a secret and can communicate the session keys for certain clients securely. In this way the fileserver can set up a secure connection with the client.

Let us look at the details.

2.1 Encryption

The files rpc2.h and secure.c in the RPC2 package contain the cryptographic routines. For export reasons only RPC2_XOR is in use, but it would be easy to use DES too.

A bit of cleanup seems to be needed, also in the random number routines.

2.2 Secure and authenticated connections

RPC2 allows to bind to a server (i.e. set up a connection) in 4 ways:

OpenKimono:
: neither authenticated nor encrypted

AuthOnly:
: authenticated but not encrypted

HeadersOnly:
: authenticated and headers of RPC packets are encrypted

Secure:
: authenticated and RPC packets are encrypted.

The security implications for the different types of connections are discussed in Satya's paper, as well as the algorithm employed in the bind routine (RPC2_NewBinding, rpc2a.c).

Of interest here is that the encryption type is given to the bind routine as a variable. A small amount of cleanup in secure.c should help to make DES encryption an easy switch. It would be interesting to measure the effect of encryption on performance.

2.3 Authentication tokens

In the preceeding two subsections we did not explain how a shared secret between the server and client can be established. The following three methods of acquiring such a secret are in place:

1. upon authentication the auth server and the client can share the password of the user to establish a secure connection

2. in the final stages of authentication the auth server can send the client a session key, which it also shares with the server
3. venus and the server use the session key from the authentication protocol to establish a secure connection

Let us go into detail for case 1. Coda employs the clog program to get a session key. clog looks up the username in `/etc/passwd` and tries a secure binding with the auth server using the username as ClientIdent and the password, which it got from the user, as the Secret. It then executes a standard bind request to the auth server.

The auth server receives this request (using RPC2_GetRequest in auth2.cc) which points to the routine PWGetKeys as a means to acquire the shared secret for the bind routine.

PWGetKeys is the heart of password authentication. The viceid for the user is looked up (see the documentation regarding the protection database) and password file auth2.pw is opened to find her password. The password can be decrypted with the FileKey ("dreuss "). At this moment PWGetKeys returns and the client and server share a secret. (XXX: the routines copying passwords copy buffers of a fixed length and might not deal with passwords which are too short gracefully).

It tries a host listed as an argument to clog to find an auth server or cycles through the hosts in vstab. If the binding is successful this is taken as a guarantee that the password was correct, and clog proceeds by placing the AuthGetTokens rpc. The server

The next rpc which clog makes invokes AuthGetTokens which sets the last used time one the ui component of the clientident and returns two tokens to the client:

- The clear token (`ctoken`): contains
  1. the handshake key (also called the session key)
  2. an authhandle (always -1)
  3. a begin time (always 0)
  4. a final time (always
  5. the viceid for the user
- the secret token (`stoken`): containing:
  1. the above information
  2. random fields

  The secret token is encrypted with the TokenKey which is obtained from auth2.tk.

Of course this information is returned over the connection secured by the users password.

We now enter the last stage of clog: it must tell Venus what about the two keys it acquired for the user. This is the routine U_SetLocalTokens, avenus.cc. This routine passes the two tokens to Venus through a pioctl. (similar routines get the tokens or remove the tokens).

2.4 Bugs to fix and things to improve

- There seems to be a problem with the handling of U_SetLocalTokens in vproc_pioctl.cc (VIOCSETTOK): it gets the ue for which it enters the token from the packet passed up by pioctl. This is wrong. The tokens were acquired for the user listed as an argument to clog and these are not necessarily the same. Remove the mess and let the routine get the user entry from the clear token (and perhaps an extra parameter passed with the pioctl).
- GetPathName in avenus.cc should return /coda by default not /cmu.
- Clean up the code a bit
- Check that the cycling through hosts in vstab is working
- Figure out if the encryption of passwords works with variable length keys.
- Check that passwords can be of variable length
- Check that clog uses the username (not the client userid) and that if it uses the id a translation localuid to viceid is maintained appropiately. This will need some modifications in venus too, but is important for cells and system administration.
- in GetPWKeys the statting of the auth.pw file is done at the wrong moment
- the "feature" that auth2 checks for the presence of "new" auth2 binaries in `/vice/bin/auth2` seems over the top and causes system administration nightmares. This code should just be removed.
- It seems pretty mysterious that the "direct assigment of fields doesn't work" (user.cc in coda-src/venus). I would like to know why.
- user.cc contains some routines that call abort. In fact these routines should be removed so that the compiler can check if those culprits are ever called.

3 Authentication to the fileserver

3.1 Authenticated connections in Venus and Vice

When a client establishes a new connection this is done in userent::Connect. The userent class knows the secret token and if it is still valid. An authenticated bind is attempted. If it fails the client removes the tokens.

The request arrives in the ServerLWP routine where it is calling RPC2_GetRequest. This time the hook to get the keys is the routine GetKeysFromToken. The secret token included with the clients bind request is decrypted with the tokens found in auth2.tk and auth2.tk.BAK. The latter allows the server password to be changed without the server having to be restarted.

The fileserver will attach a current protection subdomain (see below) to a ClientEntry when it executes the CheckSemantics calls. This is only done for authenticated connections.

3.2 Kerberos for Coda

It is appropiate to end this section with a short discussion of the things that need to be done to get kerberos to work with Coda. While the detailed names and calling conventions will remain to be sorted out the following summarizes the task. We start by very briefly reviewing Kerberos authentication.

Kerberos authentication is very similar to Coda authentication, with one difference: the kerberos server, or Key Distribution Center gives a Ticket Granting Ticket (TGT) which can be used to get specific session keys for a variety of servers from the TGS (Ticket Granting Service). Once a user has authenticated to the KDC using kinit, is given a TGT which kerberized applications use to get session keys for client server connections.

The client is issues a session key encrypted with the secret the server shares with the KDC as well as an unencrypted session key.

Instead of the dialogue with the auth2 server and the functioning of clog the following will need to be done.

1. The user authenticates to the Kerberos KDC and acquires a TGT.
2. When Venus establishes a new connection to the server it checks if it has a session key for this user. If not, it tries to acquire one using the TGT which should be available on the machine. If the session key is present but has expired it does the same. There is a standard routine in libkrb.a to get this session key.
3. It now proceeds to connect to the fileserver using the Kerberos DES encryption routine and the session key acquired from the KDC in the normal fashion of auth2 bind.
4. The server must acquire the shared secret between the KDC and the server to decrypt the packet coming from the client. Again there is a standard routine in libkrb.a to get this key. DES decryption will then give the client's request.
5. We now proceed as before using the kerberos session key instead of the auth2 session key.

These modifications seem minor. **Warning:** the sizes of a variety of structures may well be different in Kerberos and Coda.

3.3 Security holes to plug

There are a variety of security holes that we need to plug in Coda:

- the most serious issue that needs to be addressed is to use authenticated connections for the callbacks. A client should have an option to be started and not accept OPENKIMONO connections for callbacks. There seems no difficulty in implementing this.
- It would be a good idea to kerberize the volutil and
- advice monitor connections
- dispose of the insecure tcp socket for mariner and replace it with authenticated RPC2 connections.
- the failure package similarly should establish secure connections only

Do we know of any other holes?

4 Coda Protection

To quote Satya's paper: "The fundamental question is: Can agent X perform operation Y on object Z". The discussion here breaks up in several components:

- A protection database is used to group principals conveniently. This goes beyond Unix users and groups in that transtive group membership is taken into account
- Access Control Lists are attached to directory vnodes. Access Control lists allow much finer granularity of access principally through two mechanisms:
    1. Access for an arbitrary number of principals can be specified, not just for "owner", "group", "others"
    2. Negative rights can be assigned which override positive ones

In more detail the structure of these two components are held in:

1. **the protection database** is has entries which are users or groups. The attributes of users and groups are:
    (a) users, and groups. Groups and users are identified with 32 bit integers, positive for users, negative for groups. These integers should not be reused.
    (b) Users have a list of groups to which they belong and which they own. Groups have a list of groups to which they belong and of users and groups belonging to the group.
    (c) A precomputed Current Protection Subdomain, the CPS for each user and group. Informally the CPS of a user or group X is the set of groups of which X is a member, directly or indirectly, including X itself.
    (d) An access list which defines which groups and users may and may not change the entry in the protection database.
    (e) Some exceptional groups and users are defined, such as System:Administrators and System:Anyuser and user Anonymous.

    At the moment the precomputed CPS and access lists are not used.
2. The counter part are the **Access Control Lists** or **ACL**s, stored right behind the shared part of large Vnodes in server RVM. ACL's have attributes pairs of
    (a) a right or negative right; such rights are read, write, lookup, insert, delete, administer.
    (b) a group or user to which the right applies.

Note that transitive Membership and exclusion of membership relations among members and groups can be defined. Satya's papers explains several importance advantages and possibilities for using the system based on this scheme.

The total rights are the union of the rights of members of the CPS of the user. The negative rights are also a union and overide positive rights in case of conflict.

4.1 Checking permissions.

When a client first contacts a server the RPC routine ViceNewConnection (`srvproc2.cc`) is called. This calls BuildClient (`clientproc.cc`). BuildClient gets the ViceId from the username if necessary. A ClientEntry client is malloced and a the routine SetUsername is called to fill in more details. Finally AL_GetInternalCPS is called which builds up the CPS for the client entry.

AL_GetInternalCPS gets to the guts of the protection database file. It mallocs the length needed to encode the CPS and hangs it off the client entry.

The algorithm to check permissions is short and sweet and explained in Satya's paper. The detailed checks are in libal but it is instructive to step momentarily to the server code. Before performing an operation the server will do a Check_XX_Semantics routine which among other things checks permissions:

- It gets a pointer to the ACL for the large Vnode.
- It enters the GetRights routine which does some complicated checking of rights for anonymous users before calling AL_CheckRights using the CPS attached to the client and the ACL.

4.2 The authorization library.

This system is ready for a rewrite, although the ideas are fine. Let's step through some of the ingredients.

4.2.1 The .pdb .pcf mystery

When setting up a Coda server the system administrator can combine existing Unix password files and groups files into a .pdb file using the program pwd2pdb. This is merely a convenient tool to quickly get a usable .pdb file the structure of which has been described above.

A perl script could do this equally well and might easily do a precomputation of the CPS. The .pdb file is a text file which has a very useful comment at the top describing it's structure:

```
############################
# VICE protection database #
############################

# Lines such as these are comments. Comments and whitespace are ignored.

# This file consists of user entries and group entries in no particular order.
# An empty entry indicates the end.

# A user entry has the form:
# UserName      UserId
#               "Is a group I directly belong to"_List
#               "Is a group in my CPS"_List
#               "Is a group owned by me"_List
#               Access List
#               ;
```

```
# A group entry has the form:
# GroupName     GroupId OwnerId
#               "Is a group I directly belong to"_List
#               "Is a group in my CPS"_List
#               "Is a user or group who is a direct member of me"_List
#               Access List
#               ;


# A simple list has the form ( i1 i2 i3 ..... )

# An access list has two tuple lists:
#               one for positive and the other for negative rights:
#               (+ (i1 r1) (i2 r2) ...)
#               (- (i1 r1) (i2 r2) ...)
```

The program pcfgen is much more obscure, yet does something elementary: it creates a file vice.pcf containing 5 arrays:

USeeks

The offset of a user entry in the pdb file

GSeeks

The offset of a group entry in the pdb file

LitPool

An array of concatenated, null terminated strings of usernames and groupnames. The names are sorted alphabetically.

USorted

The sorted entry number of uid i

UOffset

The ofset of the i-th entry in LitPool

GSorted

Contains in spot i the sorted entry number of group with gid i in the LitPool

GOffset

The offset of entry i in the LitPool

4.3 A few of the AL_ routines

The **AL_Initialize** routine creates these arrays in VM for the server to access them. Finally the **AL_GetInternalCPS** builds a CPS by parsing the .pdb file at the offset given by the array entries, using a lex parser for pdb files. Transitive group membership and unexplained excluded entries from groups are not implemented.

4.4 Problematic aspects the ACL and PDB implementation.

Much of the AL implementation is now out of date or was incomplete to start with. The following points give a possible solution to this problem:

4.4.1 Local and Coda user id's.

It is not desirable that a userid should be the same on every client, particularly in the case of access to multiple coda clusters. Venus should have access to a translation table on the client (probably owned by root on the client) which translates local id's to remote Coda ids for various cells.

4.4.2 The pdb and pcf database

This database describes a wonderful structure but the implementation is poor:

1. It is undesirable to have to keep vice.pdb, vice.pcf and the password database auth2.pwd in sync. The utilities to move databases synchrnously indicate the history of these problems on busy servers.
2. There is no way a client can modify the pdb. All `au` operations only modify the auth2.pwd file.
3. The database is huge and large index tables and string table might needlessly be held in the Coda server's vm.
4. The system is extremely difficult to understand (particularly if one hasn't read this document or Satya's paper).

We propose therefore to make a pts server in one or two forms:

- The protection database could very conveniently be held in an LDAP database. These database servers could run on every fileserver and replicas can be held in sync using slurpd, the LDAP replication tool. This is similar but more sophisticated than the replication of kerberos databases between mastern and slave KDC's.
- Updates go to the master LDAP server, like in Kerberos.
- The same LDAP database could serve GetVolInfo requests. I will come back to this.
- LDAP is already kerberized and has tight access control.
- The list of routines to change is short and sweet:
  1. AL_GetInternalCPS
  2. AL_IdFromName
  3. AL_NameFromId
  4. some others
- The list of things we can remove from AL is long and spicy:
  1. the pdb stuff can go: it will be easy to create entries in the ldap pdb

2. the pcf stuff and the internal versions of the seek arrays can go. This eliminates major hassles with lex and flex we have had.
3. routines keeping cached pdb and pcf informatin up to date can be eliminated.

- A user of pts should be authenticated.
- CPS's attached to ClientEntry's should be refetched when updated (currently nothing is done when the .pdb file is modified to modify existing CPS's in client entry's . (How does ubiq do this?). Possibilities:
  1. Best: the ldap server when notified by slurpd of an update would prod the fileserver to re-fetch the CPS. This might not be possible without modifying ldapd.
  2. The pts client program gets all servers from LDAP and places an RPC2 with all of them to drop the cached CPS structures which it has just modified.
  3. A LDAP server should contain a table of userid's for which the pdb has been modified. Every fileserver should periodically read this list and refetch the client CPS's.
- Client programs to modify entries could be short and sweet, with one exception: they should recompute the precomputed CPS in ldap.
- The implementation of a fully features PTS server with a nice client program is very simple: the ldap server has all the basic routines.

Alternatively, we stick with the distribution of pdb databases through Update. The following modifications seems desirable:

- We need a pts type server to modify the pdb database. This has to run on the SCM and should change the auth2.pwd and pdb databases in conjunction.
- The database format for the pdb and pcf file is really not great and should be replaced by a collection of (g)dbm database files. This would also work for the volume databases.

The disadvantage of the second approach is that we effectively need to construct a networked database server for pdb files, while ldap is such a tool. The advantage is that we will control the code entirely rather than rely on 3rd parties construction.

5 Protection of files in Venus and the Coda filesystem on clients